Multi-Threaded Dense Linear Algebra Libraries for Low-Power Asymmetric Multicore Processors

Sandra Catalán[a], José R. Herrero[b], Francisco D. Igual[c], Rafael Rodríguez-Sánchez[a], Enrique S. Quintana-Ortí[a], Chris Adeniyi-Jones[d]

[a]*Depto. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain.*
[b]*Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain.*
[c]*Depto. de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Spain.*
[d]*ARM Research, Software and Large Scale Systems, Cambridge, United Kingdom.*

Abstract

Dense linear algebra libraries, such as BLAS and LAPACK, provide a relevant collection of numerical tools for many scientific and engineering applications. While there exist high performance implementations of the BLAS (and LAPACK) functionality for many current multi-threaded architectures, the adaption of these libraries for asymmetric multicore processors (AMPs) is still pending. In this paper we address this challenge by developing an asymmetry-aware implementation of the BLAS, based on the BLIS framework, and tailored for AMPs equipped with two types of cores: fast/power-hungry versus slow/energy-efficient. For this purpose, we integrate coarse-grain and fine-grain parallelization strategies into the library routines which, respectively, dynamically distribute the workload between the two core types and statically repartition this work among the cores of the same type.

Our results on an ARM® big.LITTLE™ processor embedded in the Exynos 5422 SoC, using the asymmetry-aware version of the BLAS and a plain migration of the legacy version of LAPACK, experimentally assess the benefits, limitations, and potential of this approach from the perspectives of both throughput and energy efficiency.

Email addresses: `catalans@uji.es` (Sandra Catalán), `josepr@ac.upc.edu` (José R. Herrero), `figual@ucm.es` (Francisco D. Igual), `rarodrig@uji.es` (Rafael Rodríguez-Sánchez), `quintana@uji.es` (Enrique S. Quintana-Ortí), `Chris.AdeniyiJones@arm.com` (Chris Adeniyi-Jones)

Keywords: Dense linear algebra, BLAS, LAPACK, asymmetric multicore processors, multi-threading, high performance computing.

1. Introduction

Dense linear algebra (DLA) is at the bottom of the "food chain" for many scientific and engineering applications, which can be often decomposed into a collection of linear systems of equations, linear least squares (LLS) problems, rank-revealing computations, and eigenvalue problems [1]. The importance of these linear algebra operations is well recognized and, from the numerical point of view, when they involve *dense* matrices, their solution can be reliably addressed using the *Linear Algebra PACKage* (LAPACK) [2].

To attain portable performance, LAPACK routines cast a major fraction of their computations in terms of a reduced number of *Basic Linear Algebra Subprograms* (BLAS) [3, 4, 5], employing an implementation of the BLAS specifically optimized for the target platform. Therefore, it comes as no surprise that nowadays there exist both commercial and open source implementations of the BLAS targeting a plethora of architectures, available among others in AMD ACML [6], IBM ESSL [7], Intel MKL [8], NVIDIA CUBLAS [9], ATLAS [10], GotoBLAS [11], OpenBLAS [12], and BLIS [13]. Many of these implementations offer multi-threaded kernels that can exploit the hardware parallelism of a general-purpose multicore processor or, in the case of NVIDIA's BLAS, even those in a many-core graphics processing unit (GPU).

Asymmetric multicore processors (AMPs), such as the recent ARM® big.LITTLE$^{\text{TM}}$ systems-on-chip (SoC), are a particular class of heterogeneous architectures that combine a few high performance (but power hungry) cores with a collection of energy efficient (though slower) cores.[1] With the end of Dennard scaling [14], but the steady doubling of transistors in CMOS chips at the pace dictated by Moore's law [15], AMPs have gained considerable appeal as, in theory, they can deliver much higher performance for the same power budget [16, 17, 18, 19].

In past work [20], we demonstrated how to adapt BLIS in order to attain high performance for the multiplication of two square matrices, on an ARM

[1]AMPs differ from a heterogeneous SoC like the NVIDIA Tegra TK1, in that the cores of the AMP share the same instruction set architecture (ISA).

big.LITTLE AMP consisting of ARM Cortex-A15 and Cortex-A7 clusters. In this paper, we significantly extend our previous work by applying similar parallelization principles to the complete Level-3 BLAS (BLAS-3), and we evaluate the impact of these optimizations on LAPACK. In particular, our work makes the following contributions:

- Starting from the reference implementation of the BLIS library (version 0.1.8), we develop a multi-threaded parallelization of the complete BLAS-3 for any generic AMPs, tailoring it for the ARM big.LITTLE AMP embedded in the Samsung Exynos 5422 SoC in particular. Furthermore, we demonstrate the generality of the approach by applying the same parallelization principles to develop a tuned version of BLIS for the 64-bit ARM big.LITTLE AMP in the Juno ARM development platform.
  These tuned kernels not only distinguish between different operations (e.g., paying special care to the parallelization of the triangular system solve), but also take into consideration the operands' dimensions (shapes). This is especially interesting because, in general, the BLAS-3 are often invoked from LAPACK to operate on highly non-square matrix blocks.

- We validate the correction of the new BLIS-3 by integrating them with the legacy implementation of LAPACK (version 3.5.0) from the netlib public repository.[2]

- We illustrate the computational performance and practical energy efficiency that can be attained from a straight-forward migration and execution of LAPACK, on top of the new BLIS-3 for the Exynos 5422, that basically adjusts the algorithmic block sizes and only carries out other minor modifications.
  In particular, our experiments with three relevant matrix routines from LAPACK, key for the solution of linear systems and symmetric eigenvalue problems, show a case of success for a matrix factorization routine; a second scenario where a significant modification of the LAPACK routine could yield important performance gains; and a third case where performance is limited by the memory bandwidth, but a

[2]Available at `http://www.netlib.org/lapack`.

multi-threaded implementation of the Level-2 BLAS [4] could render a moderate improvement in the results.

To conclude, we emphasize that the general parallelization approach proposed in this paper for AMPs can be ported, with little effort, to present and future instances of the ARM big.LITTLE architecture as well as to any other asymmetric design in general (e.g. the Intel QuickIA prototype [21], or general-purpose SMPs with cores running at different frequencies).

The rest of the paper is structured as follows. In Section 2, we briefly review the foundations of BLIS, and we discuss two distinct approaches (though complementary under certain conditions) to extract parallelism from LAPACK, based on a runtime that exploits task-parallelism and/or by leveraging a multi-threaded implementation of the BLAS. In Section 3, we introduce and evaluate our multi-threaded implementation of the complete BLIS-3, for matrix operands of distinct shapes, tuned for the big.LITTLE AMP architectures in the Exynos 5422 SoC and the ARM Juno platform. In Section 4, we illustrate the impact of leveraging our platform-specific BLIS-3 from LAPACK using three key operations. Finally, in Section 5 we offer a few concluding remarks and discuss future work.

2. BLIS and other Related Work

2.1. BLIS

The conventional and easiest approach to obtain a parallel execution of LAPACK, on a multicore architecture, simply leverages a multi-threaded implementation of the BLAS that partitions the work among the computational resources, thus isolating LAPACK from this task. For problems of small to moderate dimension, platforms with a low number of cores, and/or DLA operations with simple data dependencies (like those in the BLAS-3), this approach usually provides optimal efficiency. Indeed, this is basically the preferred option adopted by many commercial implementations of LAPACK.

Most modern implementations of the BLAS follow the path pioneered by GotoBLAS to implement the kernels in BLAS-3 as three nested loops around two packing routines, which orchestrate the transfer of data between consecutive levels of the cache-memory hierarchy, and a macro-kernel in charge of performing the actual computations. BLIS internally decomposes the macro-kernel into two additional loops around a micro-kernel that, in turn, is implemented as a loop around a symmetric rank-1 update (see Figure 1). In

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

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

practice, the micro-kernel is encoded in assembly or in C enhanced with vector intrinsics; see [13] for details.

A multi-threaded parallelization of the matrix multiplication (GEMM) in BLIS for conventional symmetric multicore processors (SMPs) and modern many-threaded architectures was presented in [22, 23]. These parallel implementations exploit the concurrency available in the nested five–loop organization of GEMM at one or more levels (i.e., loops), taking into account the cache organization of the target platform, the granularity of the computations, and the risk of race conditions, among other factors.

In [20] we leverage similar design principles to propose a high performance implementation of the GEMM kernel from BLIS for an ARM big.LITTLE SoC with two quad-core clusters, consisting of ARM Cortex-A15 and ARM Cortex-A7 cores. Specifically, starting from the BLIS code for GEMM, we modify the loop stride configuration and scheduling policy to carefully distribute the micro-kernels comprised by certain loops among the ARM Cortex-A15 and Cortex-A7 clusters and cores taking into consideration their computational power and cache organization.

2.2. Runtime-based task-parallel LAPACK

Extracting task parallelism has been recently proved to yield an efficient means to tackle the computational power of current general-purpose multicore and many-core processors. Following the path pioneered by Cilk [24], several research efforts ease the development and improve the performance of task-parallel programs by embedding task scheduling inside a *runtime*. The benefits of this approach for complex DLA operations have been reported, among others, by OmpSs [25], StarPU [26], PLASMA [27, 28], Kaapi [29], and `libflame` [30]. In general, the runtimes underlying these tools decompose a DLA routine into a collection of numerical kernels (or *tasks*), and then take into account the dependencies between the tasks in order to correctly issue their execution to the system cores. The tasks are therefore the "indivisible" scheduling unit while the cores constitute the basic computational resources.

The application of a runtime-based approach to schedule DLA operations in an AMP is still quite immature. Botlev-OmpSs [31] is an instance of the OmpSs runtime that embeds a Criticality-Aware Task Scheduler (CATS) specifically designed with AMPs in mind. This asymmetry-conscious runtime relies on bottom-level longest-path priorities, and keeps track of the criticality of the individual tasks to place them in either a critical queue or a non-critical one. Furthermore, tasks enqueued in the critical queue can only be executed by the fast cores, and the enhanced scheduler integrates uni- or bi-directional work stealing between fast and slow cores.

Botlev-OmpSs required an important redesign of the underlying scheduling policy to exploit the asymmetric architecture. Alternatively, in [32] we proposed an approach to refactor any asymmetry-oblivious runtime task scheduler by *i)* aggregating the cores of the AMP into a number of *symmetric virtual cores*, which become the only computational resources visible to the runtime scheduler; and *ii)* hiding the difficulties intrinsic to dealing with an asymmetric architecture inside an asymmetry-aware implementation of each individual task, in our case invocations to BLAS-3.

The benefits of these two AMP-specific approaches have been demonstrated in [31, 32] for the Cholesky factorization. Unfortunately, applying the same principles to the full contents of a library as complex as LAPACK is a daunting task. First, one would need to transform all the algorithms underlying the library to produce task-parallel versions, which can then be adapted for and fed to a specific runtime scheduler. While this work has been done for some combinations of basic matrix factorizations (for the solution of linear systems, LLS, and eigenvalue problems), runtimes, and target platforms [33, 34, 35], the effort is far from negligible.

In this paper we depart from previous work by hiding the asymmetry-aware optimization inside a parallel implementation of the complete BLAS-3

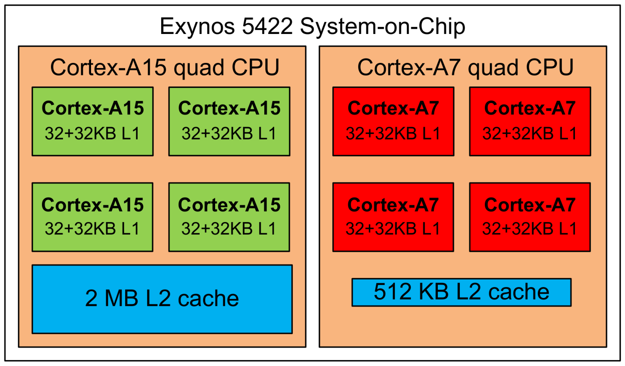


Figure 2: Exynos 5422 block diagram.

for ARM big.LITTLE architectures, which is then invoked from the legacy implementation of LAPACK. We note that, though we do not address the BLAS-1 and BLAS-2 in our work, the parallelization of the kernels in these two levels of the BLAS is straight-forward, even for an AMP.

3. Asymmetric-Aware BLAS for the Exynos 5422 SoC

3.1. Target architecture

The AMP employed in the experimentation is an ODROID-XU3 board furnished with a Samsung Exynos 5422 SoC. This processor comprises an ARM Cortex-A15 quad-core processing cluster (1.4 GHz) plus a Cortex-A7 quad-core processing cluster (1.6 GHz). Each core has its own private 32+32-Kbyte L1 (instruction+data) cache. The four ARM Cortex-A15 cores are out-of-order processors and share a 2-Mbyte L2 cache; the four ARM Cortex-A7 cores are in-order processors and share a smaller 512-Kbyte L2 cache. In addition, the two clusters access a DDR3 RAM (2 Gbytes) via 128-bit coherent bus interfaces; see Figure 2.

3.2. BLIS kernels

The specification of the BLAS-3 [5] basically comprises 6–9 kernels offering the following functionality:

1. Compute (general) matrix multiplication (GEMM), as well as specialized versions of this operation where one of the input operands is symmetric/Hermitian (SYMM/HEMM) or triangular (TRMM).

2. Solve a triangular linear system (TRSM).
3. Compute a symmetric/Hermitian rank-k or rank-$2k$ update (SYRK/HERK or SYR2K/HER2K, respectively).

The specification accommodates two data types (real or complex) and two precisions (single or double), as well as operands with different "properties" (e.g., upper/lower triangular, transpose or not, etc.). Note that HEMM, HERK, and HER2K are only defined for complex data, providing the same functionality as that of SYMM, SYRK, and SYR2K for real data.

For brevity, in the following study we will address the real double-precision version of these operations. Furthermore, we will only target the cases in Table 1, where we will consider upper triangular matrices and we will operate with/on the upper triangular part of symmetric matrices. However, we note that, due to the organization of BLIS, our optimized implementations for the Exynos 5422 SoC cover all other cases.

Kernel	Operation	Operands		
		A	B	C
GEMM	$C := C + AB$	$m \times k$	$k \times n$	$m \times n$
SYMM	$C := C + AB$ or $C := C + BA$	Symmetric $m \times m$ Symmetric $n \times n$	$m \times n$	$m \times n$
TRMM	$B := AB$ or $B := BA$	Triangular $m \times m$ Triangular $n \times n$	$m \times n$	–
TRSM	$B := A^{-1}B$ or $B := BA^{-1}$	Triangular $m \times m$ Triangular $n \times n$	$m \times n$	–
SYRK	$C := C + A^T A$	$k \times n$	–	$n \times n$
SYR2K	$C := C + A^T B + B^T A$	$k \times n$	$k \times n$	$n \times n$

Table 1: Kernels of BLIS-3 considered in the evaluation.

The steps to attain high performance from these kernels in the Exynos 5422 SoC require:

1. to develop highly optimized implementations of the underlying micro-kernels for the ARM Cortex-A15 and Cortex-A7 cores;
2. to tune the configuration parameters m_c, n_c, k_c (see Figure 1) to the target core type; and
3. to enforce a balanced distribution of the workload between both types of cores.

The following subsection offers some hints on the first two tasks, which have been carried out following a development and experimental optimization approach similar to those necessary in a homogeneous (non-asymmetric) architecture.

Our major contribution is introduced next, in subsection 3.4, where we investigate the best parallelization strategy depending on the kernel and the operands' shape. This is a crucial task as, in practice, the invocations to the BLAS-3 kernels from LAPACK generally involve nonsquare operands with one (or more) small dimension(s).

3.3. Cache optimization of BLIS

For the ARM Cortex-A15 and Cortex-A7 core architectures, the BLIS micro-kernels are manually encoded with $m_r = n_r = 4$. Furthermore, via an extensive experimental study, the configuration parameters are set to $m_c = (152, 80)$ for the ARM (Cortex-A15,Cortex-A7) cores; and $k_c = 352, n_c = 4096$ for both types of cores. With these values, the buffer A_c, of dimension $m_c \times k_c$ (408 KBytes for the Cortex-A15 and 215 KBytes for the Cortex-A7), fits into the L2 cache of the corresponding cluster, while a micro-panel of B_c, of dimension $k_c \times n_r$ (11 KBytes), fits into the L1 cache of the each core. The micro-kernel thus streams A_c together with the micro-panel of B_c into the floating-point units (FPUs) from the L2 and L1 caches, respectively; see [20] for details.

3.4. Multi-threaded parallelization of the BLIS-3

3.4.1. Asymmetric BLIS

The development of multi-threaded versions of BLIS has been previously analyzed for conventional symmetric multicore processors (SMPs) [22] and modern many-threaded architectures [23]. The insights gained from these studies about the loop(s) in Figure 1 to be parallelized can be summarized as follows:

- Loop 1. This option is equivalent to extracting the parallelism outside of BLIS. Each thread packs its own memory buffers so that this is reasonable in a multi-socket with separate L3 caches.

- Loop 2. The parallelization of Loop 2 is discarded because it requires a synchronization mechanism to deal with race conditions which may reduce performance.

- Loop 3. This option uses a common B_c memory buffer, but each thread packs it own A_c memory buffer. When there is a shared L2 cache, the size of A_c will have to be reduced by a factor equal to the degree of parallelization of this loop. However, reducing m_c is equivalent to parallelizing Loop 5.

- Loop 4. Different threads access the same memory buffer A_c in the L2 cache. In general, the amount of parallelism is larger than in Loop 5 since n_c is usually larger than m_c.

- Loop 5. This is similar to the previous case, but with less parallelism.

Taking into account the previous guidelines, and the lack of an L3 cache in the target Exynos 5422 chip, we adopted a two-level parallelization strategy. The coarse-grain (or inter-cluster) partitioning extracts *2-way parallelism* in either Loop 1 or Loop 3 to distribute the iteration space between the two clusters. These loops are appropriate candidates for parallelization across cores with a separated and isolated L2 cache, as is the case of each cluster in the Exynos 5422 SoC. The coarse-grain partitioning cannot be carried out in Loops 4 nor 5 because the threads would access to the same A_c memory buffer, which would have to be replicated in both L2 caches, degrading performance. In addition, the fine-grain (or intra-cluster) partitioning leverages up to *4-way parallelism* in either Loop 4 or Loop 5 to distribute the iteration space among the four cores of the same cluster. These loops are good candidates for parallelization across cores sharing a common L2 cache, as is the case of cores in the same cluster of the Exynos 5422 SoC. The fine-grain partitioning is set for the threads that share the same memory buffer A_c, which in BLIS is located in the L2 cache.

With this partitioning in mind, the coarse-grain (or inter-cluster) partitioning can be used to distribute the computational workload among the big and LITTLE clusters leading to a `Static` schedule. If the differences in performance between the two types of cores is known in advance, a predefined distribution ratio can be employed to distribute the computational workload among the clusters. For example, a distribution ratio `R` indicating that a big core is `R` times more powerful than a LITTLE core requires that the "amount of workload" mapped to the big cluster is `R` times larger than to the LITTLE one. However, a `Dynamic` schedule can provide a more flexible implementation that is able to adapt itself to the computational resources (e.g., due to a variation of operating frequency). In our particular case, a dynamic schedule of a given loop means that, at each iteration of that loop, a single thread bound to a big core and a single thread bound to a LITTLE core select the current chunk size (2-way parallelism for inter-cluster partitioning), which depends on the configuration parameter m_c of each type of core. The fine-grain partitioning then distributes the computational workload among the cores of a given cluster with the same computing capabilities. Therefore, for the intra-cluster partitioning a `Static` schedule is sufficient.

In the remainder of this paper we use the letters "`S`" and "`D`" to respectively indicate that a `Static` or a `Dynamic` schedule is applied. The number following each letter identifies the loop to which this scheduling is applied. Thus, for example, `D3S4` denotes a strategy that extracts inter-cluster dynamic parallelism from Loop 3 and intra-cluster static parallelism from Loop 4.

3.4.2. Square operands in GEMM

The *asymmetry-aware parallelization* of this kernel in [20] targeted only a matrix multiplication with square operands ($m = n = k$), applying a *dynamic schedule* to Loop 3 in order to distribute its iteration space between the two types of clusters (coarse-grain partitioning). In addition, a *static schedule* was internally applied to distribute the iteration space of either Loop 4 or Loop 5 among the cores of the same cluster (fine-grain partitioning). However, the parallelization of Loop 1 was discarded, when applying the dynamic workload distribution, because $n_c = 4096$, and this large value turns very difficult to attain a balanced workload distribution between the two clusters.

Following the solution adopted in [20], we will use "`D3S4`" and "`D3S5`" to refer to strategies based on a dynamic coarse-grain parallelization of Loop 3 combined with a static fine-grain parallelization of either Loop 4 or Loop 5, respectively. To assess the efficiency of these two options, we will measure the GFLOPS rates (billions of floating-point arithmetic operations per second) they attain and compare those against an "`ideal`" execution where the eight cores incur no access conflicts and the workload is perfectly balanced. To estimate the latter, we experimentally evaluate the GFLOPS achieved with the serial GEMM kernel, using either a single ARM Cortex-A15 core or a single ARM Cortex-A7 core, and then consider the ideal peak performance as the aggregation of both rates multiplied by 4 (matching the number of big-LITTLE core pairs in the system). We have applied the same methodology to derive ideal peak execution rates for all BLAS kernels and LAPACK routines developed/evaluated in our work.

Unless otherwise stated, the stride configuration parameter for Loop 3 is set to $m_c = (152, 32)$ for the ARM (Cortex-A15,Cortex-A7) cores. The value selected for the Cortex-A7 architecture is thus smaller than the experimental optimal ($m_c = 80$), but this compromise was adopted to roughly match the ratio between the computational power of both types of cores as well as to improve the workload distribution.

The top-left plot in Figure 3 reports the performance attained with the dynamic-static parallelization strategies for a matrix multiplication involving square operands only. The results show that the two options, `D3S4` and `D3S5`, obtain a large fraction of the GFLOPS rate estimated for the ideal scenario, though the combination that parallelizes Loops 3+4 is consistently better. Concretely, from $m = n = k \geq 2000$, this option delivers between 12.4 and 12.7 GFLOPS, which roughly represents 93% of the ideal peak performance. In this plot, we also include the results for an strategy that parallelizes Loop 4 only, distributing its workload among the ARM Cortex-A15 and Cortex-A7 cores, but oblivious of their different computational power (line labelled as "`ObS4`"). With this asymmetry-agnostic option, the synchronization at the end of the parallel regions slows down the ARM Cortex-A15 cores, yielding the poor GFLOPS rate observed in the plot.

3.4.3. GEMM *with rectangular operands*

The remaining three plots in Figure 3 report the performance of the asymmetry-aware parallelization strategies when the matrix multiplication kernel is invoked, (e.g., from LAPACK,) to compute a product for the following "rectangular" cases (see Table 1):

1. GEPP (general panel-panel multiplication) for $m = n \neq k$;
2. GEMP (general matrix-panel multiplication) for $m = k \neq n$; and
3. GEPM (general panel-matrix multiplication) for $n = k \neq m$.

In these three specialized cases, we vary the two equal dimensions in the range $\mathcal{R} = \{100, 300, 500, 1000, 1500, \ldots, 6000\}$ and fix the remaining one to 256. (This specific value was selected because it is often used as the algorithmic block size for many LAPACK routines/target architectures.)

The plots for GEPP and GEMP (top-right and bottom-left in Figure 3) show GFLOPS rates that are similar to those attained when the same strategies are applied to the "square case" (top-left plot in the same figure), with `D3S4` outperforming `D3S5` again. Furthermore, the performances attained with this particular strategy, when the variable problem dimension is equal or larger than 2000 (11.8–12.4 GFLOPS for GEPP and 11.2–11.8 GFLOPS for GEMP), is around 90% of those expected in an ideal scenario. We can thus conclude that, for these particular matrix shapes, this specific parallelization option is reasonable.

The application of the same strategies to GEPM delivers mediocre results, though. The reason is that, when $m = 256$, a coarse-grain distribution of the workload that assigns chunks of $m_c = (152, 32)$ iterations of Loop 3 to the ARM (Cortex-A15,Cortex-A7) cores may be appropriate from the point of view of the cache utilization, but yields a highly unbalanced execution. This behaviour is exposed with an execution trace, obtained with the `Extrae` framework [36], in the top part of Figure 4.

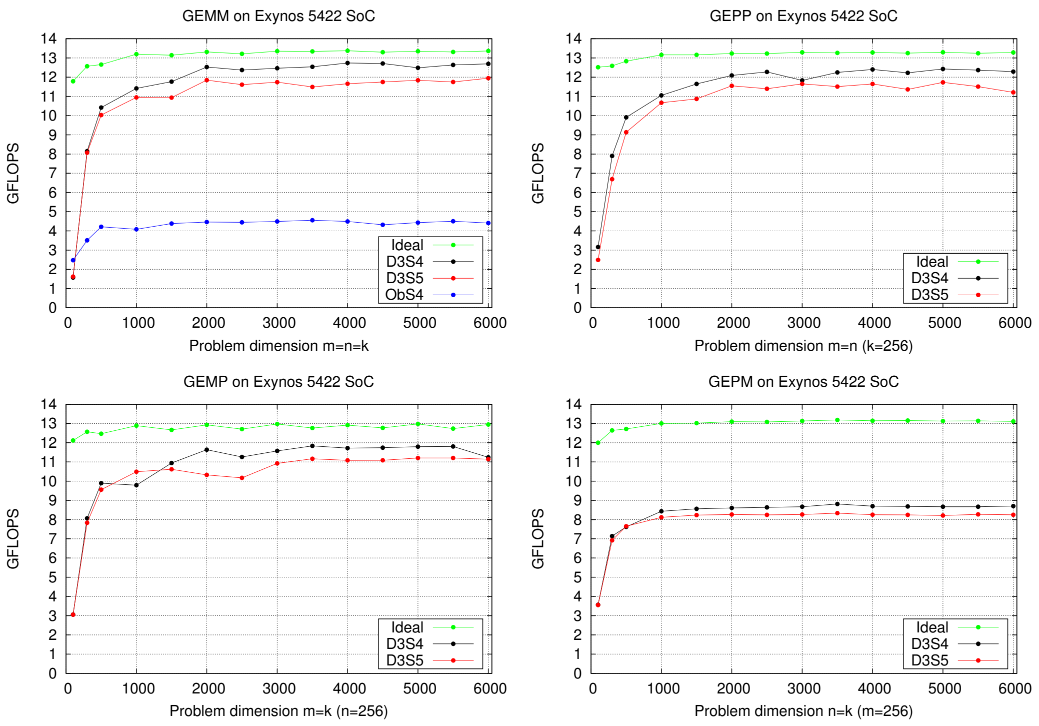
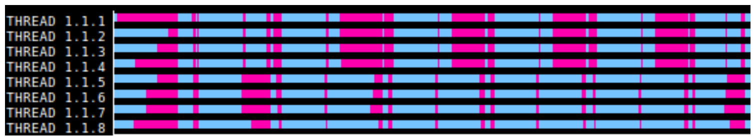


Figure 3: Performance of (general) matrix multiplication with square matrices: GEMM; and three rectangular cases with two equal dimensions: GEPP, GEMP, and GEPM.

To tackle the unbalanced workload distribution problem, we can reduce the values of m_c, at the cost of a less efficient usage of the cache memories. Figure 5 reports the effect of this compromise, revealing that the pair $m_c = (116, 24)$ presents a better trade-off between balanced workload distribution and cache optimization. For this operation, this specific pair delivers 11.8–12.4 GFLOPS which is slightly above 80% of the ideal peak performance. A direct comparison of the top and bottom traces in Figure 4 exposes the difference in workload distribution between the executions with $m_c = (152, 32)$ and $m_c = (116, 24)$, respectively.



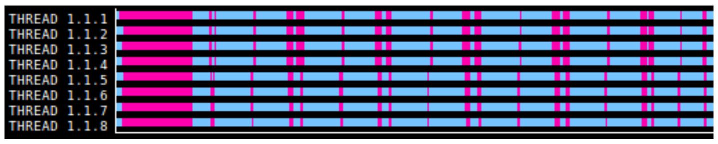


Figure 4: Execution traces of GEPM using the parallelization strategy `D3S4` for a problem of dimension $n = k = 2000$ and $m = 256$. The top plot corresponds to the use of cache configuration parameters $m_c = (152, 32)$ for the ARM (Cortex-A15,Cortex-A7) cores, respectively. The bottom plot reduces these values to $m_c = (116, 24)$. The blue periods correspond to actual work while the pink ones represent synchronization (idle time).

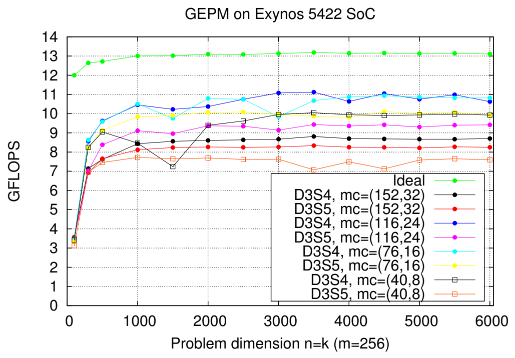


Figure 5: Performance of GEPM for different cache configuration parameters m_c.

3.5. Other BLIS-3 kernels with rectangular operands

Figure 6 reports the performance of the BLIS kernels for the symmetric matrix multiplication, the triangular matrix multiplication, and the triangular system solve when applied to two "rectangular" cases involving a symmetric/triangular matrix (see Table 1):

- SYMP, TRMP, TRSP when the symmetric/triangular matrix appears to the left-hand side of the operation (e.g., $C := C + AB$ in SYMP);
- SYPM, TRPM, TRPS when the symmetric/triangular matrix appears to the right-hand side of the operation (e.g., $C := C + BA$ in SYPM).

The row and column dimensions of the symmetric/triangular matrix vary in the range $\mathcal{R}$ and the remaining problem size is fixed to 256. Therefore, when the matrix with special structure is to the right-hand side of the operator, $m = 256$. On the other hand, when this matrix is to the left-hand side, $n = 256$. Also, in the left-hand side case, and for the same reasons argued for GEPM, we set $m_c = (116, 24)$ for the ARM (Cortex-A15,Cortex-A7) cores.

Let us analyze the performance of the symmetric and triangular matrix multiplication kernels first. From the plots in the top two rows of the figure, we can observe that `D3S4` is still the best option for both operations, independently of the side. When the problem dimension of the symmetric/triangular matrix equals or exceeds 2000, SYMP delivers 11.0–11.9 GFLOPS, SYPM 10.8–11.0 GFLOPS, TRMP 11.0–11.6 GFLOPS, and TRPM 7.8–8.9 GFLOPS. Compared with the corresponding ideal peak performances, these values approximately represent fractions of 91%, 95%, 90%, and 80%, respectively.

The triangular system solve is a special case due to the dependencies intrinsic to this operation. For this particular kernel, due to these dependencies, the BLIS implementation cannot parallelize Loops 1 nor 4 when the triangular matrix is on the left-hand side. For the same reasons, BLIS cannot parallelize Loops 3 nor 5 when this operator is on the right-hand side. Given these constraints, and the shapes of interest for the operands, we therefore select and evaluate the following three simple *static* parallelization strategies. The first variant, `S1S4`, is appropriate for TRSP and extracts coarse-grain parallelism from Loop 1 by statically dividing the complete iteration space for this loop (n) between the two clusters, assigning $r_c = 6\times$ more iterations to the ARM Cortex-A15 cluster than to the slower ARM Cortex-A7 cluster. (This ratio r_c was experimentally identified in [20] as a fair representation of the performance difference between the two types of cores available in these clusters.) In general, this strategy results in values for n_c that are smaller than the theoretical optimal; however, given that the Exynos 5422 SoC is not equipped with an L3 cache, the effect of this particular parameter is very small. At a finer grain, this variant `S1S4` statically distributes the iteration space of Loop 4 among the cores within the same cluster.

The two other variants are designed for TRPS, and they parallelize either Loop 3 only, or both Loops 3 and 5 (denoted as `S3` and `S3S5`, respectively). In the first variant, the same ratio r_c is applied to statically distribute the iterations of Loop 3 between the two types of cores. In the second variant, the ratio statically partitions (coarse-grain parallelization) the iteration space of the same loop between the two clusters and, internally (fine-grain parallelization), the workload comprised by Loop 5 is distributed among the cores of the same cluster.

The plots in the bottom row of Figure 6 show that, for TRSP, the parallelization of Loops 1+4 yields between 9.6 and 9.8 GFLOPS, which corresponds to about 74% of the ideal peak performance; for TRPS, on the other hand, the parallelization of Loop 3 only is clearly superior to the combined parallelization of Loops 3 and 5, offering 7.2–8.0 GFLOPS, which is within 65–75% of the ideal peak performance.

To conclude the optimization and evaluation of the asymmetry-aware parallelization of BLIS in the ODROID-XU3 board, Figure 7 illustrates the performance of the symmetric rank-k and rank-$2k$ kernels, when operating with rectangular operand(s) of dimension $n \times k$. For these two kernels, we vary n in the range $\mathcal{R}$ and set $k = 256$ (see Table 1). The results reveal high GFLOPS rates, similar to those observed for GEMM, and again slightly better for `D3S4` compared with `D3S5`. In particular, the parallelization of Loops 3+4 renders GFLOPS figures that are 12.0–12.4 and 11.8–12.3 for SYRK and SYR2K, respectively, when n is equal or larger than 2000. These performance rates are thus about 93% of those estimated for an the ideal scenario.

From a practical point of view, the previous experimentation reveals `D3S4` as the best choice for all BLIS-3 kernels, except the triangular system solve; for the latter kernel we select `S1S4` when the operation/operands present a TRSP-shape or `S3` for operation/operands with TRPS-shape. Additionally, in case m is relatively small, our BLIS-3 kernels optimized for the Exynos 5422 SoC set $m_c = (116, 24)$, but rely on the default $m_c = (152, 32)$ otherwise.

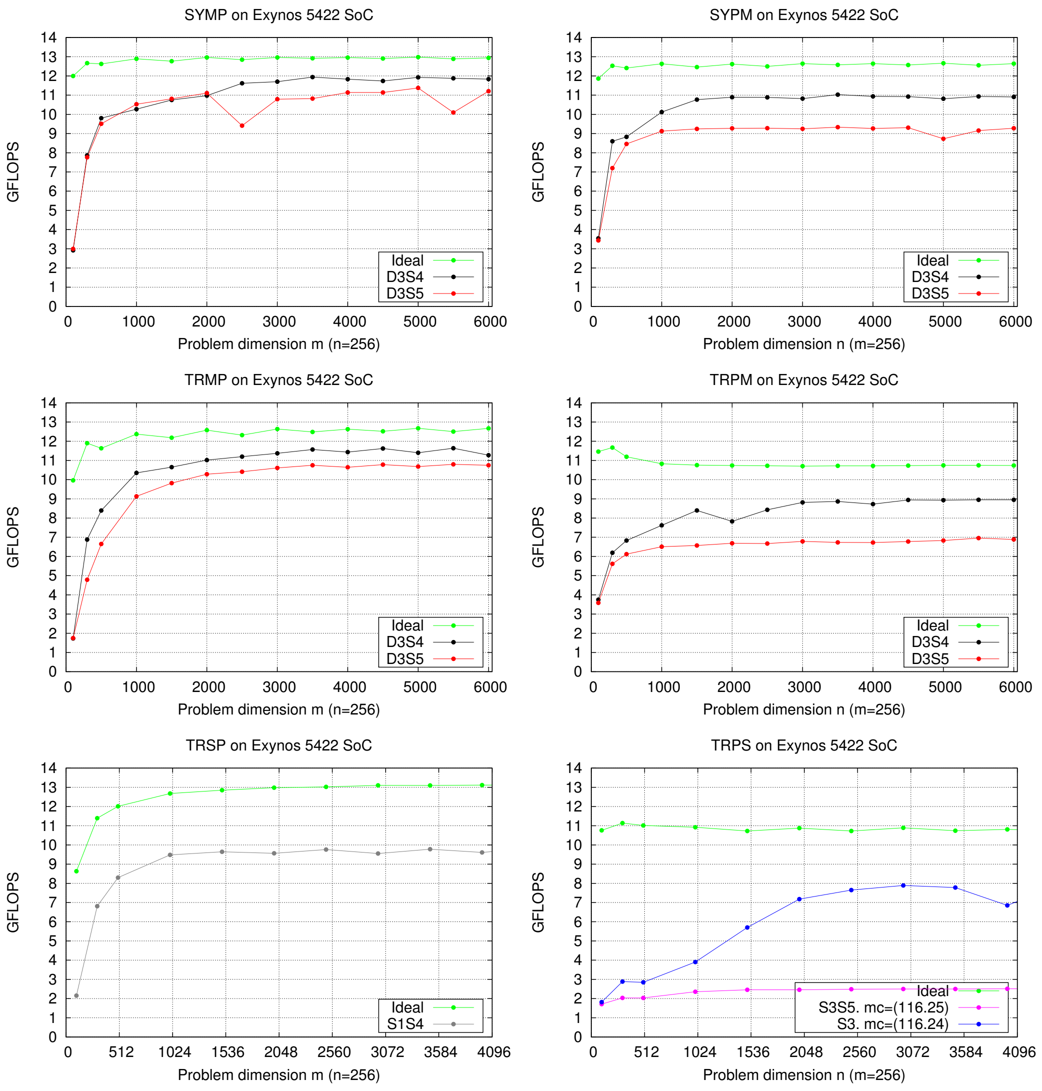
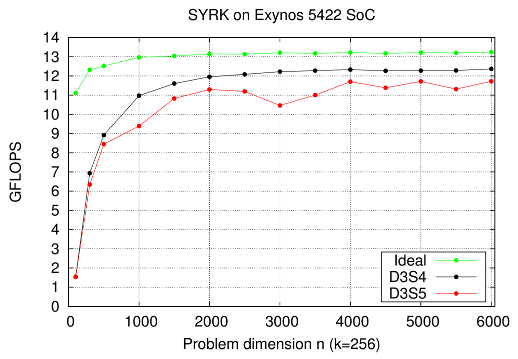


Figure 6: Performance of two rectangular cases of SYMM (SYMP for $C := C + AB$ and SYPM for $C := C + BA$), TRMM (TRMP for $B := AB$ and TRPM for $B := BA$), and TRSM (TRSP for $B := A^{-1}B$ and TRSM for $B := BA^{-1}$).

3.6. BLIS in 64-bit AMPs

In order to validate the generality of our approach for AMPs, we applied the same strategies to the 64-bit processor in a Juno ARM development platform. This processor features an ARM Cortex-A57 dual-core cluster



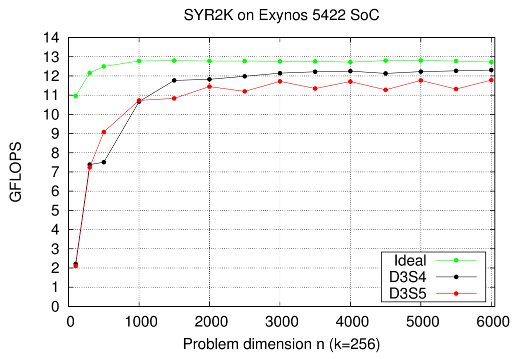


Figure 7: Performance of a rectangular case of SYRK and SYR2K.

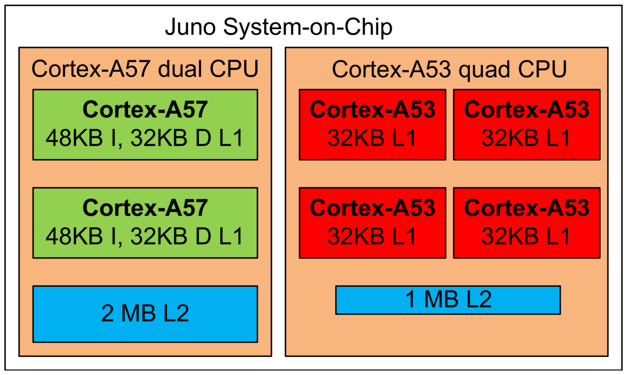


Figure 8: ARM Juno SoC block diagram.

(1.1 GHz) plus an ARM Cortex-A53 quad-core cluster (850 MHz), both implementing the ARMv8 microarchitecture; the two ARM Cortex-A57 cores share a 2-Mbyte L2 cache, while the four ARM Cortex-A53 cores share a smaller 1-Mbyte L2 cache; see Figure 8.

Figure 9 reports the performance of the BLIS kernels on the Juno board, using matrices with square operands ($m = n = k$). In our previous experiments, `D3S4` was identified as the best strategy to distribute the computational workload for all BLIS-3 routines except TRSM. Therefore, the same strategy was leveraged to produce the results in this graph comparing the performance of all BLIS-3 routines. Moreover, for comparison purposes, the graph also includes a curve for the ideal peak performance rate of GEMM, a routine which usually delivers the highest performance rate among all BLAS.

The results show that, for all routines, the D3S4 strategy delivers a large fraction of the GFLOPS estimated for the GEMM ideal scenario. Concretely, for $m = n = k = 2000$, this option renders between 10.4 and 11.1 GFLOPS, which roughly represents 82–87% of GEMM's ideal peak performance. For larger problem sizes, the graph reveals than even a larger fraction of the ideal is achieved, yielding between 12 and 12.5 GFLOPS which corresponds to 92–96% of GEMM's ideal. An additional observation from this experiment is that, for small problem dimensions, GEMM and SYMM consistently outperform the GFLOPS reported for all other routines, but for large problem dimensions only TRMM delivers slightly lower performance results.

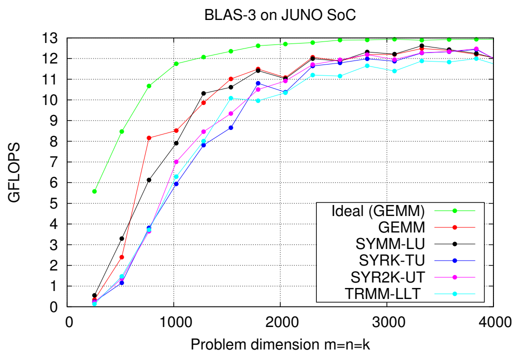


Figure 9: Performance of BLAS-3 on Juno SoC.

4. LAPACK for the Exynos 5422 SoC

Armed with the asymmetry-aware implementation of the BLIS-3 described in the previous section, we now target the execution of LAPACK on top of these optimized basic (level-3) kernels for the Exynos 5422 SoC. For this purpose, we employ version 3.5.0 of LAPACK from netlib. Here, our initial objective is to validate the soundness of our parallel version of BLIS-3 for the ARM big.LITTLE architecture, which was confirmed by successfully completing the correct execution of the testing suite included in the LAPACK installation package.

In the following subsections, we analyze the computational performance and energy efficiency of our migration of LAPACK to the Exynos 5422 SoC.

In this study, we are interested in assessing the (computational) performance of a "plain" migration; that is, one which does not carry out significant optimizations above the BLIS-3 layer. We point out that this is the usual approach when there exist no native implementation of the LAPACK for the target architecture, as is the case for the ARM big.LITTLE-based system. Furthermore, we include an analysis of the energy efficiency of LAPACK, using the GFLOPS/W (GFLOPS per Watt) metric, which determines the energy cost per flop. The impact of limiting the optimizations to this layer will be exposed via three crucial dense linear algebra operations [37], illustrative of quite different outcomes:

1. The Cholesky factorization for the solution of symmetric positive definite (s.p.d.) linear systems (routine POTRF).
2. The LU factorization (with partial row pivoting) for the solution of general linear systems (routine GETRF).
3. The reduction to tridiagonal form via similarity orthogonal transforms for the solution of symmetric eigenproblems (routine SYTRD).

For brevity, we will only consider the real double precision case.

For the practical evaluation of these computational routines, we only introduced the following minor modifications in some of the LAPACK contents related with these routines:

1. We set the algorithmic block size `NB` employed by these routines to $b = \texttt{NB} = 256$ by adjusting the values returned by LAPACK routine ILAENV.
2. For the Cholesky factorization, we modified the original LAPACK code to obtain a right-looking variant of the algorithm [37], numerically analogous to that implemented in the library, but which casts most of the flops in terms of a SYRK kernel with the shape and dimensions evaluated in the previous subsection, with n in general larger than $k = b = 256$.
3. For the Cholesky and LU factorizations, we changed the routines to (pseudo-)recursively invoke the blocked variant of the code (with block size $\tilde{b} = 32$) in order to process the "current" diagonal block and column panel, respectively [37].

4.1. Cholesky factorization

Figure 10 reports the GFLOPS and GFLOPS/W rates obtained with (our right-looking variant of the routine for) the Cholesky factorization (POTRF), executed on top of the asymmetry-aware BLIS-3 (`AA BLIS`), when applied to compute the upper triangular Cholesky factor. Following the kind of comparison done for the BLIS-3, in the performance plot we include the performance estimated for the ideal configuration (scale in the left-hand side y-axis). Additionally, in both plots we also include the execution of the factorization on top of the unmodified BLIS library using either four Cortex-A15 (`4A15`) or four Cortex-A7 cores (`4A7`). Furthermore, we offer the ratio that the actual GFLOPS rate represents compared with that estimated under the ideal conditions (line labeled as `Normalized`, with scale in the right-hand side y-axis).

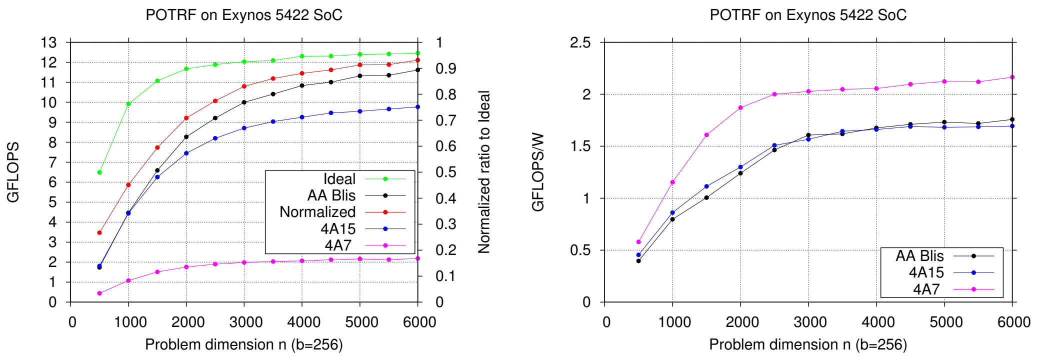


Figure 10: Performance and energy efficiency of POTRF for the solution s.p.d. linear systems.

For this particular factorization, as the problem dimension grows, the gap between the ideal peak performance and the actual GFLOPS rate rapidly shrinks. This is quantified in the columns labeled as `Normalized` in Table 2, which reflect the numerical values represented by the normalized curve in Figure 10. Here, for example, the implementation obtains over 70% and 88% of the ideal peak performance for $n = 2000$ and $n = 3000$, respectively.

This appealing behaviour is well explained by considering how this algorithm, rich in BLAS-3 kernels, proceeds. Concretely, at each iteration, the right-looking version decomposes the calculation into three kernels, with one of them being a symmetric rank-k update (SYRK) involving a row panel of $k = b$ rows [37]. Furthermore, as n grows, the cost of this update rapidly dominates the total cost of the decomposition; see the columns for the normalized flops in Table 2. As a result, the performance of this variant of the

Cholesky factorization approaches that of SYRK; see in Figure 7. Indeed, it is quite remarkable that, for $n = 6000$, the implementation of the Cholesky factorization attains slightly more than 93% of the ideal peak performance, which is basically the same fraction of the ideal peak observed for SYRK and a problem of dimension $n = 6000, k = 256$.

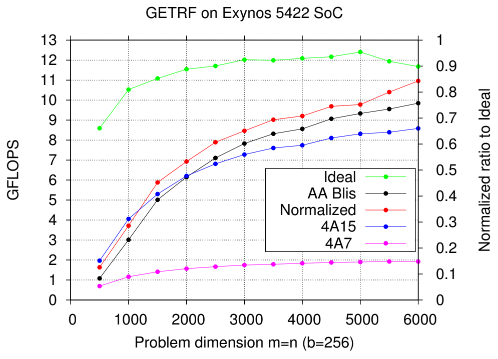
Focusing on energy efficiency, the first aspect to point out is that, as expected, the most energy efficient solution corresponds to the use of the Cortex-A7 only, though we note that this results in significantly lower performance. Second, for small problem dimensions, the performance of the asymmetry-aware BLIS-3 is similar to that obtained by using the Cortex-A15 cores only, yielding lower energy efficiency for the former as that option keeps all cores in operation. Third, for large problem dimensions, the energy efficiency of the asymmetry-aware BLIS-3 improves the energy efficiency of the alternative that relies on the Cortex-A15 cores only, since the increment in power dissipation is compensated by the increment in performance.

n	POTRF		GETRF	
	Normalized GFLOPS	Normalized flops of SYRK	Normalized GFLOPS	Normalized flops of GEPP
500	26.73	36.67	12.56	36.67
1000	45.12	64.97	28.59	64.97
1500	59.49	75.90	45.22	75.90
2000	70.85	81.65	53.27	81.65
2500	77.46	85.18	60.70	85.18
3000	83.06	87.58	65.11	87.58
3500	86.05	89.31	69.36	89.31
4000	88.06	90.61	70.80	90.61
4500	89.39	91.63	74.51	91.63
5000	91.29	92.46	75.21	92.46
5500	91.42	93.15	80.00	93.15
6000	93.16	93.69	84.30	93.69

Table 2: Performance of matrix factorizations for the solution of s.p.d. and general linear systems (POTRF and GETRF, respectively) normalized with respect to the ideal peak performance (in %); and corresponding theoretical costs of the underlying basic building blocks SYRK and GEPP normalized with respect to the total factorization cost (in %).

4.2. LU factorization

Figure 11 displays the GFLOPS and GFLOPS/W attained by the routine for the LU factorization with partial row pivoting (GETRF), linked with the asymmetry-aware BLIS-3, when applied to decompose square matrices of dimension $m = n$.



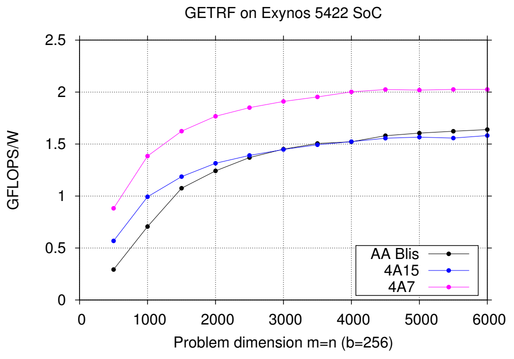


Figure 11: Performance and energy efficiency of GETRF for the solution general linear systems.

The actual performance and energy efficiency of the LU factorization follows the same general trend observed for the Cholesky factorization, though there are some differences worth of being discussed. First, the migration of the Cholesky factorization to the Exynos 5422 SoC was a story of success, while the LU factorization reflects a less pleasant case. For example, the routine for the LU factorization attains over 53% and 65.11% of the ideal peak performance for $n = 2000$ and $n = 3000$, respectively. Compared with this, the Cholesky factorization attained more than 70% and 83% at the same points. A case-by-case comparison can be quickly performed by inspecting the columns reporting the normalized GFLOPS for each factorization in Table 2.

Let us discuss this further. Like POTRF, routine GETRF casts most flops in terms of efficient BLAS-3 kernels, in this case the panel-panel multiplication GEPP. Nonetheless, its moderate performance behavior lies in the high practical cost (i.e., execution time) of the column panel factorization that is present at each iteration of the LU procedure. In particular, this panel factorization stands in the critical path of the algorithm and exhibits a limited amount of concurrency, easily becoming a serious bottleneck when the number of cores is large relative to the problem dimension. To illustrate this point, the LU factorization of the panel takes 27.79% of the total time during a parallel factorization of a matrix of order $m = n = 3000$. Compared with this, the decomposition of the diagonal block present in the Cholesky factorization, which plays an analogous role, represents only 10.42% of the execution time for the same problem dimension.

This is a known problem for which there exist *look-ahead* variants of the factorization procedure that overlap the update of the trailing submatrix with the factorization of the next panel, thus eliminating the latter from the critical path [38]. However, introducing a static look-ahead strategy into the code is by no means straight-forward, and therefore is in conflict with our goal of assessing the efficiency of a plain migration of LAPACK. As an alternative, one could rely on a runtime to produce the same effect, by (semi-)automatically introducing a sort of dynamic look-ahead into the execution of the factorization. However, the application of a runtime to a legacy code is not as simple as it may sound and, as argued in the discussion of related work, the development of asymmetry-aware runtimes is still immature.

4.3. Reduction to tridiagonal form

To conclude this section, Figure 12 reports the performance and energy efficiency behaviour of the LAPACK routine for the reduction to tridiagonal form, SYTRD. Here, we also execute the routine on top of the asymmetry-aware BLIS-3; and apply it to (the upper triangle of) symmetric matrices.

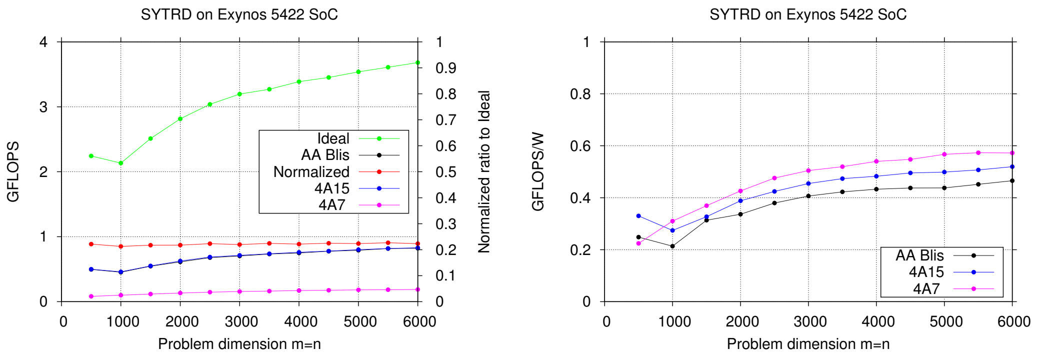


Figure 12: Performance and energy efficiency of SYTRD for the reduction to tridiagonal form.

The first difference to discuss between the results observed for this routine and those of the Cholesky and LU factorization is the scale of the left-hand side y-axis, with an upper limit at 4 GFLOPS for SYTRD against 13 for the other two. The reason is that the reduction procedure underlying SYTRD casts half of its flops in terms of the symmetric matrix-vector product, SYMV, a memory-bound kernel that belongs to the BLAS-2. Concretely, this kernel approximately performs 4 flops per memory access only and cannot take full advantage of the FPUs available in the system, which will be stalled most of the time waiting for data (the symmetric matrix entries) from memory. A second aspect to point out is the low fraction of the ideal peak performance attained with the asymmetry-aware implementation. Unfortunately, even though SYTRD performs the remaining 50% of its flops via the highly efficient SYR2K, the execution time of this other half is practically negligible compared with the execution of the SYMV kernels (for the problem dimensions evaluated in the paper, less than 5%). In addition, we note that BLIS does not provide parallel versions of the SYMV (nor any other routine from the Level-1 and Level-2 BLAS), which helps to explain the low performance attained with our plain migration of this LAPACK routine on top of a parallel BLIS-3 implementation. As a consequence, no visible benefits are obtained when using the asymmetry-aware BLIS-3, so that the performance reported is the same as that observed for the unmodified version of the library configured to use four Cortex-A15 cores only. For the same reason, the energy efficiency of the asymmetric-aware option is lower because all cores in the SoC are kept active.

5. Conclusions and Future Work

We have leveraged the flexibility of the BLIS framework in order to introduce an asymmetry-aware (and in most cases) high performance implementation of the BLAS-3 for AMPs, such as the ARM big.LITTLE SoC, that takes into consideration the operands' dimensions and shape. The key to our development is the integration of a coarse-grain scheduling policy, which dynamically distributes the workload between the two core types present in this architecture, combined with a complementary static schedule that repartitions this work among the cores of the same type. Our experimental results on the target platform in general show considerable performance acceleration for the BLAS-3 kernels, and more moderate for the triangular system solve.

In addition, we have migrated a legacy implementation of LAPACK that leverages the asymmetry-aware BLIS-3 to run on the target AMP. In doing so, we have explored the benefits and drawbacks of conducting a simple (plain) migration which does not perform any major optimizations in

LAPACK. Our experimentation with three major routines from LAPACK illustrates three distinct scenarios (cases), ranging from a compute-bound operation/routine (Cholesky factorization) where high performance/energy efficiency are easily attained from this plain migration; to a compute-bound operation (LU factorization) where the same level of success will require a significant reorganization of the code that introduces an advanced scheduling mechanism; and, finally, a memory-bound case (reduction to tridiagonal form) where an efficient parallelization of the BLAS-2 is key to obtain even moderate performance/energy efficiency.

As argued at the beginning of this section, the main goal in our work was to develop a high performance implementation of the BLAS-3 for asymmetric multicore architectures, and to evaluate its impact using several key routines from LAPACK. However, we recognize that, for low-power systems-on-chip, such as the Exynos 5422 or the Juno, an appealing complementary and/or alternative objective could be to minimize energy consumption instead of optimizing performance, or to minimize energy consumption while satisfying a certain response time (performance). As part of ongoing work, we are investigating these topics, especially for SYTRD, as the memory-bound behaviour of this operation constrains the performance benefits that can be achieved by adding more cores to the computation, significantly lowering the energy efficiency of the solution. As part of future work, we will also explore alternative parallelization strategies that better suite the triangular system solve kernel; we plan to introduce an asymmetry-aware static look-ahead scheduling into one-sided panel-operations such as the LU and QR factorizations; and we will develop an asymmetry-conscious version of the BLAS-2 from the BLIS framework.

Acknowledgments

The researchers from Universidad Jaume I were supported by projects CICYT TIN2011-23283 and TIN2014-53495-R of MINECO and FEDER, and the FPU program of MECD. The researcher from Universidad Complutense de Madrid was supported by project CICYT TIN2015-65277-R. The researcher from Universitat Politècnica de Catalunya was supported by projects TIN2015-65316-P from the Spanish Ministry of Education and 2014 SGR 1051 from the Generalitat de Catalunya, Dep. d´ Innovació, Universitats i Empresa.

References

[1] J. Demmel, Applied Numerical Linear Algebra, Society for Industrial and Applied Mathematics, 1997.

[2] E. Anderson et al, LAPACK Users' guide, 3rd Edition, SIAM, 1999.

[3] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic linear algebra subprograms for Fortran usage, ACM Transactions on Mathematical Software 5 (3) (1979) 308–323.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, An extended set of FORTRAN basic linear algebra subprograms, ACM Transactions on Mathematical Software 14 (1) (1988) 1–17.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.

[6] AMD, AMD Core Math Library, `http://developer.amd.com/tools/cpu/acml/pages/default.aspx` (2012).

[7] IBM, Engineering and Scientific Subroutine Library, `http://www.ibm.com/systems/software/essl/` (2012).

[8] Intel Corp., Intel math kernel library (MKL) 11.0, `http://software.intel.com/en-us/intel-mkl` (2014).

[9] NVIDIA, CUDA basic linear algebra subprograms, `https://developer.nvidia.com/cuBLAS` (2014).

[10] R. C. Whaley, J. J. Dongarra, Automatically tuned linear algebra software, in: Proceedings of SC'98, 1998.

[11] K. Goto, R. van de Geijn, Anatomy of a high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12:1–12:25.

[12] OpenBLAS, `http://xianyi.github.com/OpenBLAS/` (2012).

[13] F. G. Van Zee, R. A. van de Geijn, BLIS: A Framework for Rapidly Instantiating BLAS Functionality, ACM Trans. Math. Softw. 41 (3) (2015) 14:1–14:33. doi:10.1145/2764454.
URL `http://doi.acm.org/10.1145/2764454`

[14] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, A. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions, Solid-State Circuits, IEEE Journal of 9 (5) (1974) 256–268.

[15] G. Moore, Cramming more components onto integrated circuits, Electronics 38 (8) (1965) 114–117.

[16] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, K. I. Farkas, Single-ISA heterogeneous multi-core architectures for multithreaded workload performance, in: Proc. 31st Annual Int. Symp. on Computer Architecture, ISCA'04, 2004, p. 64.

[17] M. Hill, M. Marty, Amdahl's law in the multicore era, Computer 41 (7) (2008) 33–38.

[18] T. Morad, U. Weiser, A. Kolodny, M. Valero, E. Ayguade, Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors, Computer Architecture Letters 5 (1) (2006) 14–17.

[19] J. A. Winter, D. H. Albonesi, C. A. Shoemaker, Scalable thread scheduling and global power management for heterogeneous many-core architectures, in: Proc. 19th Int. Conf. Parallel Architectures and Compilation Techniques, PACT'10, 2010, pp. 29–40.

[20] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, Architecture-Aware Configuration and Scheduling of Matrix Multiplication on Asymmetric Multicore Processors, Cluster Computing (2016) 1–15.
URL `http://dx.doi.org/10.1007/s10586-016-0611-8`

[21] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, R. Iyer, Quickia: Exploring heterogeneous architectures on real prototypes, in: High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, 2012, pp. 1–8. doi:10.1109/HPCA.2012.6169046.

[22] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, R. A. V. D. Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. A. Gunnels, L. Killough, The BLIS Framework: Experiments

in Portability, ACM Trans. Math. Softw. 42 (2) (2016) 12:1–12:19. doi:10.1145/2755561.
URL `http://doi.acm.org/10.1145/2755561`

[23] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, F. G. Van Zee, Anatomy of high-performance many-threaded matrix multiplication, in: Proc. IEEE 28th Int. Parallel and Distributed Processing Symp., IPDPS'14, 2014, pp. 1049–1059.

[24] Cilk project home page, `http://supertech.csail.mit.edu/cilk/`.

[25] OmpSs project home page, `http://pm.bsc.es/ompss`, last visit: July 2016.

[26] StarPU project home page, `http://runtime.bordeaux.inria.fr/StarPU/`.

[27] PLASMA project home page, `http://icl.cs.utk.edu/plasma/`.

[28] MAGMA project home page, `http://icl.cs.utk.edu/magma/`.

[29] Kaapi project home page, `https://gforge.inria.fr/projects/kaapi`, last visit: July 2016.

[30] FLAME project home page, `http://www.cs.utexas.edu/users/flame/`.

[31] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, M. Valero, Criticality-aware dynamic task scheduling for heterogeneous architectures, in: Proceedings of ICS'15, 2015.

[32] L. Costero, F. D. Igual, K. Olcoz, E. S. Quintana-Ortí, Exploiting Asymmetry in ARM big.LITTLE Architectures for Dense Linear Algebra Using Conventional Task Schedulers, ArXiv e-prints 1590.02058.
URL `http://arxiv.org/abs/1590.02058`

[33] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using SMPSs, Conc. and Comp.: Pract. and Exper. 21 (2009) 2438–2456.

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

[35] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Trans. Math. Softw. 36 (3) (2009) 14:1–14:26.

[36] H. Servat, G. Llort, Extrae user guide manual for version 3.2.1, `http://www.bsc.es/computer-sciences/extrae` (2015).

[37] G. H. Golub, C. F. V. Loan, Matrix Computations, 3rd Edition, The Johns Hopkins University Press, Baltimore, 1996.

[38] P. Strazdins, 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).