Journal of Computational Science xxx (2016) xxx–xxx

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

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

A R T I C L E I N F O

Article history:
Received 6 November 2015
Received in revised form 30 August 2016
Accepted 31 October 2016
Available online xxx

Keywords:
Dense linear algebra
BLAS
LAPACK
Asymmetric multicore processors
Multi-threading
High performance computing

A B S T R A C T

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.

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) [3–5], 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™ 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 [16–19].

* Corresponding author.
E-mail addresses: catalans@uji.es (S. Catalán), josepr@ac.upc.edu (J.R. Herrero), figual@ucm.es (F.D. Igual), rarodrig@uji.es (R. Rodríguez-Sánchez), quintana@uji.es (E.S. Quintana-Ortí), Chris.AdeniyiJones@arm.com (C. Adeniyi-Jones).

[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).

http://dx.doi.org/10.1016/j.jocs.2016.10.020
1877-7503/© 2016 Elsevier B.V. All rights reserved.

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 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 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] Available at http://www.netlib.org/lapack.

```
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
```

Fig. 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.

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 Fig. 1). In 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,