Static Scheduling of the LU Factorization with Look-Ahead on Asymmetric Multicore Processors

Sandra Catalán[a], José R. Herrero[b], Enrique S. Quintana-Ortí[a], Rafael Rodríguez-Sánchez[a,*]

[a]*Dept. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain*
[b]*Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain*

Abstract

We analyze the benefits of look-ahead in the parallel execution of the LU factorization with partial pivoting (LUpp) in two distinct "asymmetric" multicore scenarios. The first one corresponds to an actual hardware-asymmetric architecture such as the Samsung Exynos 5422 system-on-chip (SoC), equipped with an ARM big.LITTLE processor consisting of a quad-core Cortex-A15 cluster plus a quad-core Cortex-A7 cluster. For this scenario, we propose a careful mapping of the different types of tasks appearing in LUpp to the computational resources, in order to produce an efficient architecture-aware exploitation of the computational resources integrated in this SoC. The second asymmetric configuration appears in a hardware-symmetric multicore architecture where the cores can individually operate at a different frequency levels. In this scenario, we show how to employ the frequency slack to accelerate the tasks in the critical path of LUpp in order to produce a faster global execution as well as a lower energy consumption.

Keywords: Dense linear algebra, LU factorization, look-ahead, asymmetric multicore processors, multi-threading, frequency scaling

1. Introduction

The LU factorization with partial pivoting (LUpp) [1] is a crucial operation for the solution of dense linear systems that is supported by LA-

*Corresponding author
Email addresses: `catalans@uji.es` (Sandra Catalán), `josepr@ac.upc.edu` (José R. Herrero), `quintana@uji.es` (Enrique S. Quintana-Ortí), `rarodrig@uji.es` (Rafael Rodríguez-Sánchez)

PACK [2], `libflame` [3] as well as by commercial libraries such as Intel MKL [4]. High performance implementations of LUpp decompose this operation into a series of computational building blocks or *tasks*. In the legacy version of the operation in LAPACK, these tasks mainly correspond to kernels from BLAS (*basic linear algebra subroutines* [5]), for which there exist vendor-specific implementations as well as highly competitive open alternatives (e.g., GotoBLAS [6, 7], OpenBLAS [8], ATLAS [9], BLIS [10], etc.).

In this paper, we examine the parallelization of LUpp in the context of a parallel execution on asymmetric multicore processors (AMPs), considering two sources of asymmetry:

- Hardware architecture: The ARM big.LITTLE (v7/v8) architectures integrate two types of cores, combining a few high performance yet power-hungry cores with several energy-efficient but low performance cores. For linear algebra operations, performance is of paramount importance and, therefore, exploiting both types of cores is crucial.
- Core frequency: Recent processors from Intel can adjust the frequency (and voltage) of the hardware cores at execution time, on a per-core basis. In a power-constrained scenario, either because of thermal design limits of the architecture or due to external constrains imposed by the facility, the cores can be set to run at different frequencies to better leverage the resources [11], yielding an asymmetric architecture from the point of view of performance.

In our previous work in [12], we proposed several asymmetry-aware enhancements for the efficient execution of LUpp, enhanced with a technique known as static look-ahead [13, 14], which aims to eliminate the sequential panel factorization from the critical path of the global operation, on ARM big.LITTLE AMPs. In the present paper, we extend that work, making the following new contributions:

- For the hardware-asymmetric scenario, we develop several specialized versions of the BLAS and LAPACK kernels appearing in LUpp, for the ARM big.LITTLE multicore processor integrated into the Samsung Exynos 5422 system-on-chip (SoC). These versions include a new asymmetry-aware parallel scheme of the partial pivoting routine (LASWP) as well as some extra tuned configurations for the triangular system solve (TRSM) and matrix multiplication (GEMM).
- For the frequency-asymmetric scenario, we evaluate the performance benefits that can be obtained by carefully adjusting the frequency of

the cores in charge of the critical tasks during the factorization. While this strategy can be automatically applied by the hardware via, e.g., the Linux kernel when the proper governor is set, in our approach we depart from this strategy to use application-level information in order to set the frequency configuration on a task-level basis.

The rest of the paper is structured as follows. In Section 2 we describe the basic algorithms for LUpp, and discuss their parallelization options on a symmetric multicore processor. In Section 3, we perform a complete performance analysis of different parallelization variants of LUpp on a hardware-asymmetric platform equipped with the Samsung Exynos 5422 (ARM big.LITTLE v7) SoC. In Section 4, we study the performance, power and energy consumption of LUpp with static look-ahead on a frequency-asymmetric platform. Finally, in Section 5, we present the conclusions.

2. Parallel LUpp on Symmetric Multi-threaded Architectures

Given a square matrix $A \in \mathbb{R}^{n \times n}$, the LU factorization with partial pivoting produces a unit lower triangular factor $L \in \mathbb{R}^{n \times n}$, an upper triangular factor $U \in \mathbb{R}^{n \times n}$, and a permutation matrix $P \in \mathbb{R}^{n \times n}$, such that $PA = LU$ [1]. In this section, we revisit two blocked algorithms for LUpp, discussing several approaches to obtain a multi-threaded execution on a generic (symmetric) multicore processor. For simplicity, we do not include pivoting in the description of the algorithms, though all our actual implementations integrate the standard partial pivoting. The arithmetic cost of computing LUpp via this algorithm is $2n^3/3 + O(n^2)$ floating-point arithmetic operations (flops).

2.1. Basic algorithms and conventional parallelization

The algorithms in Figure 1 show the blocked left-looking (LL) and right-looking (RL) variants of LUpp, using the FLAME notation [15]. For simplicity, we only describe next the RL variant, which is the algorithm implemented in LAPACK and `libflame`. At each iteration, this variant of LUpp relies on an unblocked factorization algorithm to process the "current" panel A_p, composed of b columns, where b is often referred to as the algorithmic block size. Next, it updates the trailing submatrix, consisting of A_{12} and A_{22}, via a triangular system solve (TRSM, RL2) followed by a matrix multiplication (GEMM, RL3). Provided $b \ll n$, in this blocked algorithm most flops are cast in terms of the GEMM kernel. Upon completion, the triangular factors L and U respectively overwrite the strictly lower and upper triangular parts of the input matrix A.

```
Algorithm: $[A] := \text{LU\_BLK}(A)$
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where  $A_{TL}$ is $0 \times 0$
while $n(A_{TL}) < n(A)$ do
    Determine block size $b$
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        where  $A_{11}$ is $b \times b$,
        Define $A_p = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$

    LL1. $A_{01} := \text{TRILU}(A_{00})^{-1} A_{01}$        RL1. $A_p := \text{LU\_UNB}(A_p)$
    LL2. $A_p := A_p - \begin{bmatrix} A_{10} \\ A_{20} \end{bmatrix} A_{01}$        RL2. $A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$
    LL3. $A_p := \text{LU\_UNB}(A_p)$        RL3. $A_{22} := A_{22} - A_{21} A_{12}$

    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

Figure 1: blocked LL and RL algorithms (left and right operations in the loop-body, respectively) for LUpp. In the notation, $n(\cdot)$ returns the number of columns of its argument, and TRILU$(\cdot)$ returns the strictly lower triangular part of its matrix argument, setting the diagonal entries of the result to ones.

The conventional parallelization of LUpp for multicore processors simply relies on multi-threaded instances of TRSM and GEMM. Unfortunately, the factorization of the panel A_p (RL1) lies in the critical path of the algorithm and this kernel exhibits a reduced degree of concurrency. This performance bottleneck can be ameliorated by computing the factorization of the panel A_p via a "recursive" call to the blocked variant, with a block size $b' < b$.

2.2. Dynamic look-ahead

The "fork-join model" of the conventional parallelization of LUpp artificially constrains the algorithmic concurrency of the operation to that intrinsic in the individual BLAS kernels [16, 17]. Recent runtime-based alternatives extract task-level parallelism dynamically in an attempt to overcome some of the problems of the fork-join model [16, 17, 18]. In these dynamic parallel versions, the matrix is partitioned into a collection of column blocks (or panels), so that the operations on each panel, during each iteration of the algorithm, become a single task. In contrast with the conventional parallelization, in the runtime-based solution *i)* a task comprises multiple calls to BLAS kernels; *ii)* each of these invocations to BLAS relies on a sequential implementation of the corresponding kernel; and *iii)* the runtime-based parallelization schedules a concurrent execution of independent tasks.

The runtime-based parallelization overlap the execution of tasks from the same or different iterations (potentially overcoming the bottleneck imposed by the panel factorization), with the only restriction of fulfilling the real dependencies among them. However, this runtime-based parallelization often produces a suboptimal use of the cache hierarchy because the threads then compete for the use of shared resources instead of sharing them [14].

2.3. Static look-ahead

Look-ahead [13] is a technique that tackles the performance bottleneck represented by the factorization of A_p by overlapping the factorization of the "next" panel (say, that involved in iteration $k+1$ of the main loop in the blocked RL algorithm for LUpp) with the update of the "current" trailing submatrix (for iteration k).

Figure 2 re-organizes the blocked RL algorithm for LUpp to expose the look-ahead technique. The partitioning of the trailing submatrix into two column panels creates the two coarse-grain independent tasks which can be computed concurrently: the factorization of the next panel T_{PF} (consisting of PF1, PF2, PF3); and the remainder update of the current trailing submatrix T_{RU} (composed of RU1 and RU2). By adjusting the amount of computational resources (thread teams) dedicated to each of the two independent tasks, T_{PF} and T_{RU}, this static look-ahead can partially overcome the bottleneck represented by the panel factorization.

In [14] we improved the static look-ahead variant via two workload-balancing techniques:

- *Worker Sharing (WS):* At each iteration, if the team of threads dedicated to T_{PF} completes this task before T_{RU} is finished, a *malleable thread-level (MTL) implementation* of BLIS (*BLAS-like Library Instatiation Software Framework [10]*) incorporates these idle threads to the ongoing computation of T_{RU}. Note that this degree of malleability is not possible in standard multi-threaded implementations of the BLAS, as once a kernel (such as TRSM in RU1 or GEMM in RU2) is invoked with a certain number of threads, these versions of the BLAS do not allow variations in the thread concurrency until the execution is completed. We refer to our flexible implementation as the MTL BLIS,

```
Algorithm: $[A] := \text{LU\_LA\_BLK}(A)$
Determine block size $b$
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $A_{BR} \rightarrow \left(\begin{array}{c|c} A_{BR}^{\mathsf{PF}} & A_{BR}^{\mathsf{RU}} \end{array}\right)$
  where $A_{TL}$ is $0 \times 0$, $A_{BR}^{\mathsf{PF}}$ has $b$ columns
$A_{BR}^{\mathsf{PF}} := \text{LU\_BLK}\left(A_{BR}^{\mathsf{PF}}\right)$
while $n(A_{TL}) < n(A)$ do
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
    where $A_{11}$ is $b \times b$

    Determine block size $b$
    % Partition into panel factorization and remainder
    $\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) \rightarrow \left(\begin{array}{c|c} A_{12}^{\mathsf{PF}} & A_{12}^{\mathsf{RU}} \\ \hline A_{22}^{\mathsf{PF}} & A_{22}^{\mathsf{RU}} \end{array}\right)$
    where both $A_{12}^{\mathsf{PF}}$, $A_{22}^{\mathsf{PF}}$ have $b$ columns

    % Panel factorization, $\mathsf{T_{PF}}$
    PF1. $A_{12}^{\mathsf{PF}} := \text{TRILU}(A_{11})^{-1} A_{12}^{\mathsf{PF}}$
    PF2. $A_{22}^{\mathsf{PF}} := A_{22}^{\mathsf{PF}} - A_{21} A_{12}^{\mathsf{PF}}$
    PF3. $A_{22}^{\mathsf{PF}} := \text{LU\_BLK}\left(A_{22}^{\mathsf{PF}}\right)$

    % Remainder update, $\mathsf{T_{RU}}$
    RU1. $A_{12}^{\mathsf{RU}} := \text{TRILU}(A_{11})^{-1} A_{12}^{\mathsf{RU}}$
    RU2. $A_{22}^{\mathsf{RU}} := A_{22}^{\mathsf{RU}} - A_{21} A_{12}^{\mathsf{RU}}$

    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

Figure 2: Blocked RL algorithm enhanced with static look-ahead for the LU factorization.

and we view the migration of threads into an ongoing kernel execution as *worker sharing* (WS). For further details on the parallelization of BLIS on multicore processors, in combination with MTL BLIS, we refer the reader to [14].

- *Early Termination (ET):* At each iteration, if $\mathsf{T_{RU}}$ is completed before $\mathsf{T_{PF}}$, we notify this event to the team of threads in charge of the latter so that they stop the factorization of the panel, and both teams inmediatly proceed to work on the next iteration of the static look-ahead algorithm. This produces an automatic tuning of the block size.

The implementation of LUpp with look-ahead, WS, MTL BLIS and ET, delivers notable performance on a conventional multicore processor equipped with Intel Xeon cores; see [14] for details.

3. Parallel LUpp on ARM big.LITTLE AMPs

The parallelization of the LUpp with static look-ahead on a multicore platform requires the design of a specialized strategy to distribute the cores between the $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$ thread teams. In comparison with a symmetric multicore processor, an AMP such as the Exynos 5422 thus offers a richer collection of possibilities since, in addition to the number of threads that are assigned to each team, we also need to decide the type of the cores [12].

After a brief description of the experimental setup, in this section we further tune our adaption of the LUpp with static look-ahead for AMPs in [12]. Specifically, we evaluate several new parallel configurations that were enabled by the use of a more flexible implementation of GEMM and TRSM; furthermore, we also inspect the parallel execution of LASWP.

3.1. Experimental setup

All the experiments in the following subsections were performed employing IEEE double precision arithmetic on a Samsung Exynos 5422 SoC consisting of four ARM Cortex-A15 plus four ARM Cortex-A7 cores. The frequency was set to 1.3 GHz for both types of cores.

In the experiments, we consider square matrices of dimension n=500 to 8,000 in steps of 500, with random entries uniformly distributed in the interval $(0, 1)$. The algorithmic block size was tested for values b=32 to 512 in steps of 32; the inner block size for the panel factorization was b_i=32. The performance results are rated in terms of GFLOPS (billions of flops per second), using the standard flop count of $2n^3/3$ for LUpp.

3.2. Variants with static look-ahead

The starting point for the experimental evaluation of the LUpp with static look-ahead is an analysis of the impact caused by the distribution of the cores between the teams that execute the GEMM kernels appearing in $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$, as this particular BLAS operation dominates the cost of the complete factorization. This study then impacts the distribution of the threads for the TRSM kernels, constraining the variety of parallel configurations that can be employed for the execution of this second BLAS kernel.

The parallel configurations evaluated in the following subsections are identified using a naming scheme of the form "(w+x|y+z)", where "w+x" are two numbers, in the range 0–4, used to specify the amount of Cortex-A7+Cortex-A15 cores (in that order) mapped to the execution of the BLAS kernels appearing in $\mathsf{T_{PF}}$; and "y+z", in the same range, play the same role for those mapped to the execution of the BLAS kernels appearing in $\mathsf{T_{RU}}$.

At this point, we note that given that most flops occur inside T_{RU}, it is natural to dedicate more threads (or, at least, the most powerful ones) to the execution of the kernels in this task. We report results for a natural "homogenous" configurations that distributes the cores between the two execution branches (T_{PF} and T_{RU}) depending on the core type, and the best "heterogeneous" configurations among those that we tested.

3.2.1. Configuration of GEMM

Compared with our previous version in [12], this work introduces some changes in the parallelization of GEMM for AMPs in order to make the code for this operation more flexible. Concretely, our previous work required that, in a "heterogeneous" configuration, a thread team includes cores of the two distinct types, and the number of Cortex-A7 and Cortex-A15 cores in the teams were the same. In contrast, in the current version we have modified the code to accommodate heterogeneous configurations with any combination of Cortex-A7 and Cortex-A15 cores in a team. This is achieved by decoupling the code of both core types, via the implementation of two execution branches in the kernel code. This new structure offers greater flexibility and allows a finer control of the processing elements.

In this subsection, we propose three mappings of the Cortex-A7 and Cortex-A15 cores to the execution of the GEMM kernels:

- GEMM(4+0|0+4): Homogeneous configuration where the Cortex-A7 cluster executes PF2 and the GEMM kernels in PF3; the Cortex-A15 cluster is in charge of RU2.
- GEMM(1+1|3+3): Heteregeneous configuration with one Cortex-A7 core collaborating with one Cortex-A15 core in the execution of PF2 and the GEMM kernels in PF3; the remaining cores (three Cortex-A7 cores plus three Cortex-A15 cores) are mapped to RU2.
- GEMM(0+1|4+3): Heterogeneous configuration with a single Cortex-A15 core mapped to PF2 and the GEMM kernels in PF3; the remaining cores (four Cortex-A7 cores plus three Cortex-A15 cores) execute RU2.

For this initial experiment, the number of threads that participate in the execution of the TRSM kernels appearing in T_{PF} and T_{RU} (PF1 plus the invocations to TRSM from within PF3 and RU1, respectively) equals the number of Cortex-A15 cores that are in charge of the corresponding task. This means, for example, that in the configuration GEMM(1+1|3+3), a single Cortex-A15 executes PF1 and the TRSM kernels appearing in PF3, while the remaining three Cortex-A15 cores execute RU1. Furthermore, if the configuration assigns no Cortex-A15 cores to the execution of GEMM for a particular task, then the full Cortex-A7 cluster will participate in the execution of the TRSM kernels present in that task. The row permutations (LASWP) mostly occur outside the coarse-grain tasks T_{PF} and T_{RU}, and are carried out by all threads. The only exception are the row permutations that occur within the panel factorization PF3, which are performed by the same collection of threads that are mapped to the execution of the GEMM kernels in T_{PF}.

For the homogeneous configuration GEMM(4+0|0+4), at the beginning of each iteration, the execution employs symmetric kernels since PF2 and the GEMM kernels in PF3 are mapped to four Cortex-A7 cores only, while RU2 is executed by the four Cortex-A15 cores only. However, due to the integration of an instance of BLIS with thread-level malleability, as soon as the GEMM kernels in T_{PF} are completed, the active Cortex-A7 cores become part of the team in charge of T_{RU}, and the execution relies on asymmetric kernels from that point.

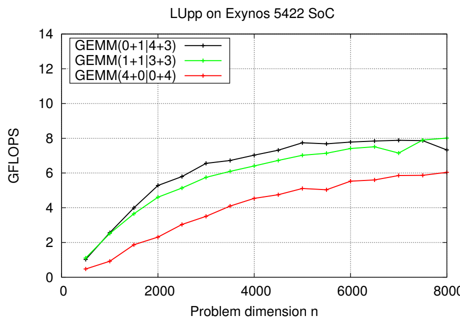

Figure 3: Performance of the blocked RL algorithm with static look-ahead enhanced with WS and ET for different configurations of GEMM.

The heterogeneous configurations differ in the combination of a few Cortex-A7 and Cortex-A15 in the execution of the GEMM operations appearing in T_{RU}. In the first case, GEMM(1+1|3+3), the same number of big and LITTLE cores are used in each task, devoting less resources to T_{PF}. As shown in Figure 3, combining two different types of cores in both tasks improves the performance of LUpp. On the other hand, GEMM(0+1|4+3) features an uneven number of cores for the execution of the GEMM kernels in both T_{PF} and T_{RU}. This mapping accelerates the execution of T_{RU} because more cores are devoted to the GEMM appearing in this task, which benefits an operation as parallel as GEMM.

In general, the best parallelization option, GEMM(0+1|4+3), assigns a single Cortex-A15 core to the execution of T_{PF} while the remaining resources are dedicated to T_{RU}. Fortunately, this choice is the optimal for almost all problem dimensions and, therefore, it is not necessary to modify the thread mapping during the progress of the factorization as smaller LUpp subproblems are encountered. This decision sets some conditions on the parallelization of TRSM next, as we cannot expect to change the mapping of the cores from between different kernels.

3.2.2. Configuration of TRSM

The parallelization of TRSM for AMPs presented in [12] could not accommodate any combination of Cortex-A7 and Cortex-A15 cores. One contribution of this work is the modification of TRSM code in order to make it as flexible as GEMM. We include the best configuration for TRSM from our previous work and, thanks to the finer control provided by the new implementation of TRSM, we present two new configurations in order to explore the impact of including Cortex-A7 cores in the execution of T_{RU}.

We next proceed to assess the performance of three different configurations for TRSM. These variants differ in the *specific loop* that is parallelized in BLIS [10] and the use of a homogeneous/heterogenoeus configuration for the execution of the TRSM kernels in T_{RU}:

- TRSM(L1L4;0+1|4+3): Heterogeneous configuration where a two-level parallelization is employed to distribute the iteration space of Loop 1 (outermost loop of the kernel) for TRSM between the Cortex-A15 and Cortex-A7 clusters; then, in Loop 4 of TRSM, each core within the same cluster is given an even part of the workload assigned to that cluster; see [19] for details.

- TRSM(L4;0+1|4+3): Heterogeneous configuration where the asymmetric distribution of the workload between the two types of clusters occurs in Loop 4 only.

- TRSM(L4;0+1|0+3): Homogeneous configuration where a symmetric implementation of TRSM takes advantage of the Cortex-A15 cores only.

Figure 4 reports the results for the different configurations of TRSM. Clearly, the worst option corresponds to the mapping that implements an asymmetric distribution of Loop 4 only. Curiously, although this reduces the number of data buffers and diminishes the volume of cache misses [10], it offers the lowest performance rate. This behaviour can be explained by the implementation of this configuration. Given that an asymmetric distribution of the workload is required at this level, a dynamic mechanism is needed in order to implement it [19]. However, the overhead introduced by the dynamic scheduling impairs performance when exploiting the asymmetry of the SoC, mainly due to the small workload that TRSM represents in the factorization (note that, for large matrix sizes, this trend changes). On the other hand, if the TRSM kernels are configured to leverage the dynamic distribution at a higher level (Loop 1), the overhead of this mechanism is significantly reduced and the performance improves considerably.

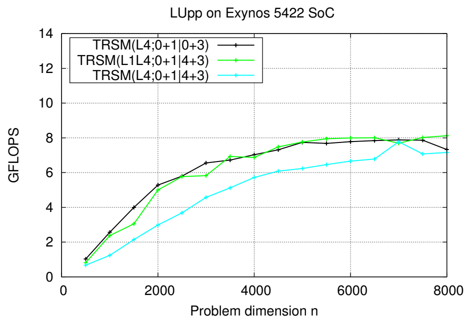

Figure 4: Performance of the blocked RL algorithm with static look-ahead enhanced with WS and ET for different configurations of TRSM.

To close the evaluation of this kernel, we tested the benefits of employing the Cortex-A15 cores to compute TRSM only. Interestingly, the results show that this is the best choice in order to optimize performance, since adding Cortex-A7 cores decreases the performance rate due to the low workload of this kernel. Concretely, the TRSM kernels appearing in LUpp operate most of the time on a square triangular matrix of small dimension of order b (often with $b = 256$) lessening the benefits of using additional resources.

3.2.3. Configuration of LASWP

To conclude the analysis of the three main building blocks present in LUpp when running on an AMP, we extend the work in [12] to analyze the impact of the row permutations (LASWP). For this analysis, we fix the configurations GEMM(0+1|4+3) and TRSM(L4;0+1|0+3). The major part of the row permutations occur after the operations in tasks T_{PF} and T_{RU} have been completed. The purpose of the following experiments is to determine which cores to employ in these permutations.

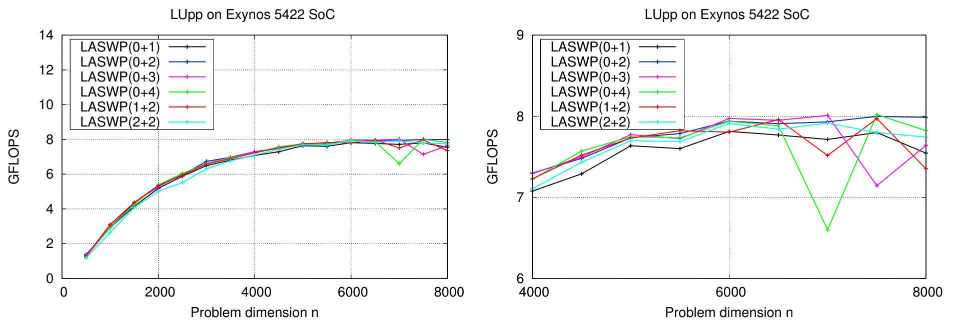

Figure 5: Performance of the conventional parallelization of the blocked RL algorithm with different LASWP parallelizations and static look-ahead enhanced with WS and ET.

The left-hand side plot in Figure 5 reports the performance of distinct mappings of threads to the execution of LASWP. From these results, we can conclude that, for small matrices, a homogeneous configuration that employs the Cortex-A15 cores only in general provides higher performance rates. The reason is that the workload is too small to take advantage of the Cortex-A7 cores. In addition, for the large matrices, it is hard to determine whether there is any difference in performance. When zooming in the results (see the right-hand side plot in the same figure), we can observe that adding one or more Cortex-A7 cores actually reduces the performance.

In summary, we can conclude that, in order to attain high performance, the best option when parallelizing LASWP employs two Cortex-A15 cores only. The main reason for this is that LASWP is a memory-bound operation and adding more cores, simply saturates the memory bandwidth with no real benefit from the point of view of performance.

3.3. Global comparison

Figure 6 compares three parallel asymmetry-aware configurations for the execution of LUpp on ARM big.LITTLE AMPs:

- LU_AS: blocked RL algorithm for LUpp (without look-ahead) linked with asymmetric-aware implementation of the basic building blocks GEMM and TRSM that employs all 8 cores of the ARM SoC to extract parallelism from within each BLAS kernel.

- LU_LA: LUpp with look-ahead with the optimal configurations of the basic kernels determined in subsection 3.2: GEMM(0+1|4+3), TRSM(L4;0+1|0+3) and two Cortex-A15 for the execution of LASWP.

- LU_OS_VC: The best runtime-based option found in the literature which exploits the asymmetry of the SoC aggregating the computational resources into 4 homogeneous virtual cores (VCs), composed each of a single Cortex-A15 core plus a single Cortex-A7 core, and then relies on a runtime in order to introduce a dynamic look-ahead strategy in the implementation of LUpp; see [20, 12] for details.

In this plot, we can observe that the conventional parallelization delivers a reduced performance rate due to the bottleneck imposed by the panel factorization. This hurdle can be greatly palliated through the introduction of static look-ahead, enhanced with WS and ET, yielding an implementation that consistently delivers up to 2 GFLOPS more than the conventional algorithm for LUpp. Finally, the runtime-based implementation, which applies a dynamic look-ahead strategy, proves that the dynamic look-ahead is especially beneficial for small problems ($n < 2,000$). For mid-large problem dimensions though, our implementation with static look-ahead matches or slightly outperforms the runtime-based approach, at the cost of a more expensive programming and optimization cost.

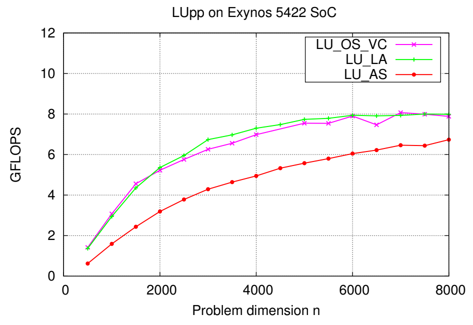

Figure 6: Performance of the best parallel configuration for each variant: conventional blocked RL algorithm, static look-ahead version enhanced with WS and ET, and runtime-based implementation.

4. Parallel LUpp on Frequency-Asymmetric Intel Xeon Cores

As argued in the introduction, even in a symmetric multicore processor, a potential asymmetric scenario appears if different cores operate at distinct frequency levels. This may occur, for example, because the power budget for the processor, server or complete facility enforces strict constrains in the highest power rate for the processor. As a response, in recent hardware architectures, power consumption can be finelly regulated (at the core level) by adjusting the voltage level and frequency of individual cores.

For LUpp with static look-ahead, in our parallelization we divided the main loop into two coarse-grain tasks, T_{PF} and T_{RU}, and explored how to distribute the computational resources (i.e., cores) in order to produce a balanced execution cost of both "branches", identifying two cases: *i)* When the block size is relatively large compared with the problem dimension, T_{PF} *dominates the total execution time of LUpp*; while, *ii)* in the opposite case, T_{RU} *is more costly than* T_{PF}.

In the remainder of this section, we will focus on the first case. For this particular situation, ET tackles the cost imbalance by stopping the panel factorization in T_{PF} at an early stage, to advance the computation to the next iteration, in practice reducing the algorithmic block size. As a side effect, if the parameter b becomes "too small" (usually, below 192, depending on the implementation of the BLAS and the cache hierarchy), the performance of the GEMM kernels appearing in the T_{RU} task is strongly reduced, and this renders a significant negative impact on the global performance of LUpp. At this point we remind that, in this T_{PF}-dominated scenario, increasing the amount of cores that compute T_{PF} cannot be expected to produce a significant positive effect, as the degree of parallelism of the panel factorization is very reduced.

In the same scenario (i.e., when the execution time of T_{PF} is significantly larger than that of T_{RU}, given the computational resources assigned to each task), a strategy to ameliorate the cost imbalance can exploit a potential surplus in the power budget to accelerate the execution of T_{PF}, via an increase of the frequency of the cores assigned to this task. Thus, the question we want to investigate is whether, by shifting part of the power budget to accelerate the execution of T_{PF}, we can reduce the total execution time of LUpp. Leaving the Linux governor in control of adjusting the voltage–frequency scaling will likely produce a suboptimal solution because, by the time T_{RU} is completed, and the governor shifts the surplus in the power budget to the execution of T_{PF}, a significant time may have been wasted. In contrast, given our detailed knowledge about the task-parallel organization of LUpp with look-ahead, we can set the cores in charge of T_{PF} to operate at a higher frequency than those that compute T_{RU} for the full execution of LUpp. We can refer to our approach as *algorithmic-based (voltage–)frequency scaling* (ABFS).

4.1. Experimental setup

Our experiments have been carried out on an Intel Xeon E5-2630 v3 that is furnished with 64 GB of DDR3 RAM and features two sockets with 8 cores each, which may run from 1.2 GHz to 2.4 GHz. In our tests we only employ one socket, assigning 1 core to T_{PF} and 7 cores to T_{RU}. All experiments are done in double precision arithmetic. Moreover, in the experiments we consider square matrices of dimension n=1,000 to 18,000 in steps of 2,000, with random entries uniformly distributed in the interval (0, 1). The algorithmic block size was set to $b = 256$ and the inner block size for the panel factorization was $b_i = 32$.

4.2. Global comparison

We remind that the case that we explore in this section corresponds to the following scenario:

- LUpp with static look-ahead. To simplify the analysis, we will not enhance this code with ET and WS, and we will use the regular BLIS instance of BLAS (without thread malleability). WS is only possible when using a MTL instance of the BLAS, which is currently only available as part of our implementation of BLIS. Furthermore, WS+MTL BLAS can only be expected to have a positive impact on performance when the cost of LUpp is dominated by T_{RU}. In our approach for frequency-asymmetry architectures, we consider the variation of frequency as an alternative to ET.

- Target hardware-symmetric multicore architecture, with voltage and frequency scaling at the core level.

- Small problem dimension compared with the amount of cores dedicated to computing the factorization.

In the remaining experiments in this subsection, the frequency of the core in charge of T_{PF} is set to the highest possible value, $f_{\mathsf{PF}} = f_{\max} = 2.4$ GHz, while the frequency of the remaining 7 cores, in charge of T_{RU}, is set to a "nominal" value $f_{\mathsf{RU}} = f_N$ ranging between 1.2 and 2.3 GHz, depending on the configuration. The results obtained for LUpp with look-ahead using these configurations are reported in Figure 7. There, each line represents the normalized value obtained for a specific asymmetric configuration with 1 core at f_{PF} and 7 cores at f_{RU}, with respect to the reference configuration that has all 8 cores set to operate at f_{RU}.

From the point of view of execution time, the top–left plot in the figure shows that, as expected, the acceleration factors are more relevant for the smaller problem dimensions, for which T_{PF} plays a more important role and dominates the total execution time. As the problem dimension grows, T_{RU} becomes more costly, and the time reductions become less important, though they are still visible as the algorithm for LUpp decomposes the operation into sub-problems of decreasing dimensions. In all experiments, the existence of a larger power surplus (i.e., the acceleration of the panel factorization by a larger margin) produces a more significant reduction of the execution time.

The top-right plot in Figure 7 reports the normalized average power consumption. The highest difference in the power dissipation rate appears for the configuration that sets f_{RU} to the lowest frequency, i.e., 1.2 GHz. Now, given that the dynamic power satisfies $P_{dynamic} \propto V^2 f \approx\propto' f^3$ [21], if we calculate the theoretical normalized power ratio between the asymmetric and symmetric configurations for that particular configuration, we have $P_{asym}/P_{sym} = (f^3_{\mathsf{PF}} + 7f^3_{\mathsf{RU}})/8f^3_{\mathsf{RU}} = 1.875$. In contrast, the comparison of the actual consumption rates shown in the plot offers an experimental ratio of 1.322. The difference between theory and practice can be explained here by the role of the static power consumption, which varies at a lower pace with respect to the frequency than that of the dynamic component of the power [21]. An additional explanation for the source of this difference is the temporal and energy costs of changing the frequency during the execution of the factorization.

The results for the normalized energy consumption are reported in the bottom plot in Figure 7. The trend for the energy consumption is similar to that observed for the execution time: small and medium problems largely benefit from setting the core executing T_{PF} to f_{PF}. However, for matrix sizes larger than $n = 10{,}000$, the asymmetric configuration of the platform increases the total energy consumption. The reason is that, although the asymmetric configuration reduces the execution time, the increase in power consumption exceeds that reduction, harming the total energy consumption.

5. Concluding Remarks

While the parallelization of dense linear algebra operations (DLA) on symmetric multicore processors has been largely analyzed, in this paper we have addressed the more arduous parallel execution of this type of operations on AMPs, using the LU factorization with partial pivoting (LUpp) and static-look-ahead as a representative case study. The rationale for our study steams from two asymmetric configurations, discussed next.

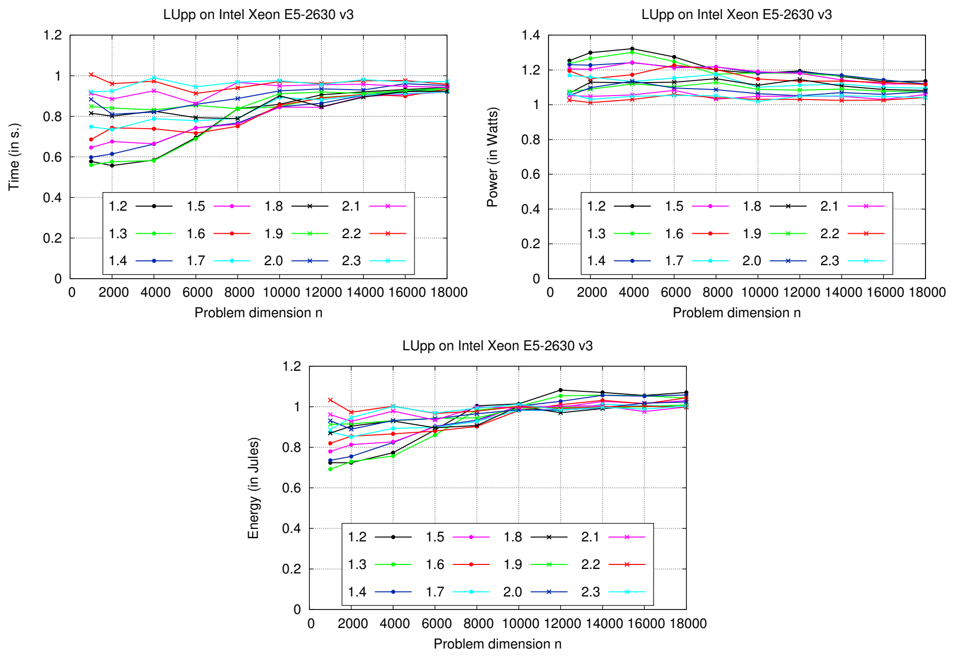

Figure 7: Normalized execution time (top left), average power (top right) and energy (bottom) of the blocked RL algorithm with static look-ahead for different core frequency configurations.

Hardware-based AMPs provide a means to tackle the power wall by utilizing the most appropriate type of core at each moment, while powering down the remaining computational resources (dark silicon). However, for dense linear algebra, attaining high performance is of paramount importance. In this situation, an appealing alternative is to set the cores to operate at a low frequency, but employ all of them to solve the problem. This leads to the question of how to coordinate all these resources efficiently for the parallel execution of LUpp with static look-ahead (enhanced with workload balancing techniques such as WS and ET). Our analysis and experiments show that this can be done by considering the distinct type of tasks appearing in LUpp, choosing the best number and type of cores on an individual basis.

Frequency-based AMPs stand at an intermediate point between standard multicore processors and hardware-based AMPs: like in a standard multicore architecture, all threads present the same architecture (e.g., cache organization) and instruction set; but if one or more cores operate at a different frequency, they reproduce the parallel programming challenge identified for AMPs. For this particular case, we demonstrate the advantages of utilizing the frequency slack to obtain an algorithmic-based frequency scaling (ABFS) of the critical tasks appearing in LUpp, in order to reduce both the execution time and energy consumption of this DLA operation.

In summary, when applied with care it is natural to expect that a manual distribution of the workload among the processor cores can outperform dynamic scheduling, at the cost of a more complex coding effort. We believe that delivering this message is nonetheless important for three reasons: *i)* Current development efforts are pointing in the opposite direction of introducing dynamic scheduling via a runtime; *ii)* for AMPs, the development of asymmetry-aware runtimes is rather inmature and quite more complex; and *iii)* given the right level of abstraction, modifying a factorization routine to manually introduce a static look-ahead is rather simple.

Acknowledgements

The researchers from Universidad Jaume I were supported by projects TIN2014-53495-R and TIN2017-82972-R of MINECO and FEDER, and the FPU program of MECD. The researcher from Universitat Politècnica de Catalunya was supported by projects TIN2015-65316-P from the Spanish Ministry of Education and 2017-SGR-1414 from the Generalitat de Catalunya.

References

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

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

[3] F. G. V. Zee, `libflame`: The Complete Reference, `www.lulu.com`, 2009.

[4] Intel, Math Kernel Library, `https://software.intel.com/en-us/intel-mkl` (2015).

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

[7] K. Goto, R. A. van de Geijn, High performance implementation of the level-3 BLAS, ACM Trans. Math. Soft. 35 (1) (2008) 4:1–4:14.

[8] `http://www.openblas.net` (2015).

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

[10] 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.

[11] M. Schulz, Power constrained HPC, slides of the talk at EnaHPC 2017, Frankfurt.

[12] S. Catalán, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, J. R. Herrero, Static versus dynamic task scheduling of the LU factorization on ARM big.LITTLE architectures, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 733–742.

[13] P. Strazdins, A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, Tech. Rep. TR-CS-98-07, Dept. of Computer Science, The Australian National University, Canberra, Australia (1998).

[14] S. Catalán, J. R. Herrero, E. S. Quintana-Ortí, R. Rodríguez-Sánchez, R. A. van de Geijn, A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting, CoRR abs/1611.06365.
URL `http://arxiv.org/abs/1611.06365`

[15] J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn, FLAME: Formal linear algebra methods environment, ACM Trans. Math. Softw. 27 (4) (2001) 422–455.

[16] 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.

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

[18] 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.

[19] S. Catalán, J. R. Herrero, F. D. Igual, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, C. Adeniyi-Jones, Multi-threaded dense linear algebra libraries for low-power asymmetric multicore processors, Journal of Computational Science.
URL `http://dx.doi.org/10.1016/j.jocs.2016.10.020`

[20] L. Costero, F. D. Igual, K. Olcoz, S. Catalán, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, Refactoring conventional task schedulers to exploit asymmetric ARM big.LITTLE architectures in dense linear algebra, in: IEEE Int. Parallel & Distributed Processing Symp. Workshops, 2016, pp. 692–701.

[21] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Pub., San Francisco, 2003.