UNIVERSITAT JAUME I DE CASTELLÓ
ESCOLA DE DOCTORAT DE LA UNIVERSITAT JAUME I

Multithreaded Dense Linear Algebra on Asymmetric Multi-core Processors

Castelló de la Plana, November 2017

Ph.D. Thesis

Presented by: Sandra Catalán Pallarés
Supervised by: Enrique S. Quintana Ortí
Rafael Rodríguez Sánchez

Programa de Doctorat en Informàtica

Escola de Doctorat de la Universitat Jaume I

Multithreaded Dense Linear Algebra on Asymmetric Multi-core Processors

Memòria presentada per Sandra Catalán Pallarés per optar al grau de doctor/a per la Universitat Jaume I.

Sandra Catalán Pallarés

Enrique S. Quintana Ortí, Rafael Rodríguez Sánchez

Castelló de la Plana, desembre de 2017

Agraïments Institucionals

Esta tesi ha estat finançada pel projecte FP7 318793 "EXA2GREEN" de la Comisió Europea, l'ajuda predoctoral de la Universitat Jaume I i per l'ajuda predoctoral FPU del Ministeri de Ciència i Educació.

Contents

List of Figures

List of Tables

I'm not telling you it is going to be easy, I'm telling you it's going to be worth it.

Nicholas Negroponte

Summary

Nowadays, there exists a large variety of scientific, industry and engineering applications that require high computational power and storage, and their demands continue to grow; in order to obtain more precise solutions in these applications, scientists need to elaborate and work with more sophisticated and complex physical and mathematical models. Consequently, the capacity of new data processing systems and High Performance Computing (HPC) centers is saturated shortly after of their set up [2, 17, 44, 50]. Nonetheless, these resources are still used: scientific computation (or computational sciences, that is, the elaboration of mathematical models and the use of computers to analyze and solve scientific problems) is an effective tool in scientific discovering, complementary to more traditional methods based on theory and experimentation [44, 50].

Large-scale HPC systems are large energy consumers, that employ computing resources and auxiliary systems to operate [44, 49, 51, 74]. This consumption has a direct impact on the operational and maintenance costs of computing centers. However, electricity cost is not the only problem; in general, energy consumption turns into carbon emissions that are dangerous to the environment and public health, and the heat reduces the reliability of the hardware [51]. The situation requires additional measures: studying the Green500 list of June 2017 [1] we can see that, nowadays, the most efficient HPC systems in terms of power consumption attain 14,110 MFLOPS per Watt (MFLOPS/W). A simple calculation reveals that reaching the EXAFLOPS rate with the current technology will require 70.9 MFLOPS/W approximately, with an approximate cost of 70.9 million dollar per year. Although EXAFLOPS challenge will unleash innovative scientific discoveries, it is also true that more efficient hardware and software technologies are required from the energy point of view [6, 13, 18].

The pressure of HPC centers has forced hardware manufacturers to improve their designs to increase energy efficiency: Central Processing Unit (CPU), memory and disks (three of the large energy consumers in computing systems, with the remaining ones being the interconnection network and the power supply) integrate energy saving strategies, based on the system transition to low power states or the dynamic reduction of frequency and voltage (DVFS or Dynamic Voltage Frequency Scaling). On the other hand, software systems, communication libraries and, especially, computational libraries and application codes running in HPC centers have been, in general, oblivious to energy consumption. The Top500 [4] list is a good example. Computers in this list are classified according to the sustained performance (in floating-point arithmetic operations per second (FLOPS)) that the Linpack benchmark attains (basically, the solution of a dense linear system of large dimension). However, the numerical method behind this test, the LU factorization, is far from representative of the real performance attained by most scientific codes [18].

Even though this is a mature topic in other segments, the development of energy-aware solutions for HPC applications, which optimize both the execution time and the energy consumption, is still only in its early stages, despite the huge benefits that it may produce [6, 51]. The HPC community is now aware of the energy costs, as was demonstrated with the creation of the Green500 [1] list.

As a response to this situation, the general objective of this thesis is the study, design, development and analysis of experimental solutions that are energy-aware for the execution of scientific and engineering numerical applications on low power architectures, more specifically asymmetric platforms. With the aim of demonstrating the benefits of these contributions, we selected diverse dense linear algebra operations that arise in very different areas, such as image processing, molecular dynamics simulation, and big data analytics, among others.

Resumen

En la actualidad existe una gran variedad de aplicaciones científicas, industriales y de ingeniería que requieren un alto poder computacional y de almacenamiento, y su demanda continúa creciendo; para obtener soluciones más precisas en estas aplicaciones, los científicos necesitan elaborar y trabajar con modelos físicos y matemáticos mucho más sofisticados. Como consecuencia, los nuevos equipos de procesamiento de datos y los centros de Computación de Altas Prestaciones (CAP) se saturan a las pocas semanas de haber sido puestos en funcionamiento [2, 17, 44, 50]. Todos estos recursos no se desaprovechan: la computación científica (o ciencias de la computación, es decir, la elaboración de modelos matemáticos y el uso de computadoras para analizar y resolver problemas científicos) es una herramienta eficaz para el descubrimiento científico, complementaria a los métodos más tradicionales basados en la teoría y la experimentación [44, 50].

Los sistemas de CAP de gran escala son grandes consumidores de energía, que emplean los recursos de computación y los sistemas auxiliares para funcionar [44, 49, 51, 74]. Este consumo tiene un impacto directo en los costes de funcionamiento y mantenimiento de los centros de computación, poniendo en peligro su existencia y perjudicando la puesta en marcha de nuevas instalaciones. Sin embargo, el coste de la electricidad no es el único problema; en general, el consumo de energía resulta en emisiones de dióxido de carbono que son un peligro para el medio ambiente y la salud pública, y el calor reduce la fiabilidad de los componentes de hardware [51]. La situación requiere medidas adicionales: estudiando la lista Green500 de junio de 2017 [1] se puede observar que, a día de hoy, los sistemas de CAP más eficientes en cuanto a potencia consumida ofrecen 14.110 MFLOPS por vatio (MFLOPS/W). Así, un simple cálculo revela que alcanzar la tasa de EXAFLOPS con la tecnología actual requeriría, aproximadamente, 70,9 MFLOPS/W, con un coste también aproximado de 70,9 millones de dólares por año solo en electricidad. Aunque el desafío del EXAFLOPS hará, sin duda, que se produzcan descubrimientos científicos innovadores, también es cierto que se necesita que la tecnología hardware y el software del sistema sean más eficientes desde el punto de vista energético [6, 13, 18].

La presión de los centros de CAP ha obligado a los fabricantes de hardware a mejorar sus diseños para conseguir una mejora en la eficiencia energética: Unidad Central de Procesamiento (CPU), memoria y discos (tres de los grandes consumidores de energía en los sistemas informáticos, siendo los otros dos la red de interconexión y la fuente de alimentación) cuentan con algunas estrategias de ahorro, basadas en la transición del sistema a algún estado de bajo consumo o en la reducción de la frecuencia y el voltaje de forma dinámica (DVFS o Dynamic Voltage Frequency Scaling). Por otro lado, los sistemas software, las bibliotecas de comunicación y, en especial, las bibliotecas computacionales y los códigos de aplicaciones usados en los centros de CAP han sido, en general,

insensibles al consumo de energía. La lista Top500 [4] es un claro ejemplo. Los computadores de esta lista quedan clasificados en función del rendimiento sostenido (en operaciones aritméticas en coma flotante por segundo (FLOPS)) que alcanzan en el test Linpack (básicamente, la solución de un sistema lineal denso de dimensión escalable). Sin embargo, el método numérico que hay detrás de este test, la factorización LU, es poco representativo del rendimiento real conseguido por la mayoría de códigos científicos [18].

Si bien es un tema maduro en otros segmentos, el desarrollo de soluciones conscientes de la energía para las aplicaciones de CAP, que optimizan tanto el tiempo de ejecución como la conservación de la energía, se encuentra todavía en sus inicios, a pesar de los enormes beneficios que puede producir [6, 51]. La comunidad que trabaja en CAP ha empezado a tomar consciencia de los costes energéticos, y así se ha creado una clasificación como la lista Green500 [1], que utiliza este tipo de métricas para comparar y clasificar los supercomputadores en el mundo de acuerdo a su eficiencia energética.

En respuesta a esta situación, el objetivo general de esta tesis es el estudio, diseño, desarrollo y análisis experimental de soluciones conscientes de la proporcionalidad energética para aplicaciones científicas y de ingeniería numéricos sobre arquitecturas de bajo consumo, concretamente en plataformas asimétricas. Con el propósito de demostrar los beneficios de estas aportaciones, se han escogido distintas operaciones de álgebra lineal presentes en aplicaciones pertenecientes a campos tan distintos como el procesado de imágenes, la simulación de dinámica molecular o el análisis de grandes volúmenes de datos.

Agradecimientos

El tiempo pasa volando y prueba de ello han sido los últimos años que han dado lugar a esta tesis doctoral. Ha sido un periodo intenso, en el que el esfuerzo, la dedicación y el sacrificio han sido claves; pero todo ha sido posible gracias a la motivación y satisfacción de trabajar en lo que más me gusta. Ha sido una etapa llena de retos y experiencias, tanto a nivel personal como profesional, lo que me ha permitido disfrutar de la novedad de las circunstancias. Sin embargo, nada de esto hubiese sido posible sin la ayuda de las personas que me han brindado la oportunidad de trabajar con ellas, y las que, desinteresadamente, me han acompañado en cada momento de esta aventura. A todas ellas, me gustaría expresar mi más sincero agradecimiento.

A mis directores, Enrique S. Quintana Ortí y Rafael Rodríguez Sánchez. A Enrique, por su apoyo, por ser una fuente infinita de conocimientos y trabajador incansable. A Rafa, por su dedicación y su ánimo en mis malos momentos.

A las personas del grupo HPC&A de la Universitat Jaume I, a las que en algún momento han estado en él y a sus visitantes, José, José Manuel, Sergio B., Asun, Maribel, Juan Carlos, Germán L., Germán F., Merche, Rafa M., Alfredo, Manel, Toni, Fran, José Antonio, Maria, Rocío, Sergio I., Héctor, Adrián, Sisco, Sonia, Andrés, Goran, José Ramón y Pedro. Gracias por el buen ambiente y por vuestra colaboración. También quiero agradecer su ayuda a los técnicos del Departamento de Ingeniería y Ciencia de los Computadores, Gustavo y Vicente.

A Costas Bekas, por darme la oportunidad de conocer de primera mano la investigación en el entorno empresarial. A Cristiano, por su ayuda, guía y atención. También a los internos de verano de IBM, que hicieron de la estancia una experiencia inolvidable. A Robert van de Geijn y su grupo, por ofrecerme la oportunidad de integrarme en su entorno de trabajo y por su contribución en la tesis. Asimismo, me gustaría dar las gracias por su hospitalidad a los compañeros del grupo *Programming Models* del BSC, especialmente a Vicenç.

A mis amigos, por las conversaciones infinitas dándome apoyo y sabios consejos.

A mi familia y muy especialmente a mis padres, Manolo y María José, por su apoyo, comprensión y paciencia, por estar ahí simpre.

A Rubén, por hacer de mi proyecto parte del suyo. Por sus ánimos en los momentos de frustración y por compartir los logros.

– ¡Gracias! · Gràcies! · Thanks! · Danke! –

CHAPTER 1

Introduction

For many decades, the evolution of computer technologies and architectures has enlarged the performance gap between the throughput of processors and the memory access rate [63]. While the difference is no longer growing at a significant rate, the scenario that we face nowadays is that, in order to deliver very high performance, programmers have to explicitly take into account the cost of data movements. This is particularly the case of dense linear algebra (DLA) operations for which there exist highly tuned implementations, almost for any architecture, from the past vector processors to the current multicore designs, and the fancier graphics processing units and co-processors such as the Intel Xeon Phi.

This dissertation targets two important problems. The first one is the design of low-level DLA kernels for architectures comprising two (or more) classes of cores. The main question we have to address here is how to attain a balanced distribution of the computational workload among the heterogeneous cores while taking into account that some of the resources, in particular cache levels, are either shared or private. The second question is partially related to the first one. Concretely, this dissertation explores an alternative to runtime-based systems in order to extract "sufficient" parallelism from complex DLA operations while making an efficient use of the cache hierarchy of the architecture.

1.1 Motivation

Asymmetry-aware BLAS. DLA is at the bottom of the "food chain" for many scientific and engineering applications, which require kernels to tackle linear systems, linear least squares problems or eigenvalue computations, among others [38]. To address this, the scientific community has reacted by creating the Basic Linear Algebra Subroutines (BLAS) [47] in order to define standard domain-specific interfaces for a few dozens of fundamental DLA operations (such as the vector norm, the matrix-vector product, and the matrix-matrix multiplication) and improve performance portability across a wide range of computer architectures.

Highly-tuned implementations of BLAS (for example, Intel MKL [66], IBM ESSL [64], NVIDIA cuBLAS [81], GotoBLAS [57, 58], OpenBLAS [97], ATLAS [96] or BLIS [95]) aim to deliver high performance by carefully orchestrating the movement of data across the memory hierarchy to hide this overhead. To attain this goal, these implementations interleave data packing operations with

computations to maximize the access to data that is closer to the arithmetic units and favour the use of vector instructions. When the target architecture comprises multiple cores, these instances of BLAS also exploit *loop-parallelism* to distribute the workload in a balanced manner, ensuring that the cores make an efficient use of their private cache memories, and collaborate in the access to the shared cache levels instead of competing for them.

Asymmetric processors are heterogeneous architectures in principle conceived as a low-power designs for embedded systems. However, as we progress along the final steps of Moore's Law, the constraints imposed by the end of Dennard's scaling [40] and the high energy proportionality [17] of asymmetric processors are promoting them into a building block to assemble large-scale facilities for high performance computing [3]. Asymmetry can also appear in a conventional multicore processor when some of the cores are enforced to operate at a lower frequency, due for example to constraints imposed in the processor's power budget.

From the point of view of a parallel BLAS, asymmetric processors present the difficulty of distributing the work among a heterogeneous collection of resources (in practice, two classes). In order to attain high performance, the scheduling has then to take into consideration the distinct computational capabilities of the cores (e.g., due to different width/dimension of the floating-point units or different core frequency). In addition, for processors that are physically-asymmetric, the parallelization has to consider also the distinct cache configurations. As we will expose in this dissertation, even for simple and regular operations such as those composing BLAS, the challenge is far from trivial when the goal is to squeeze the last few drops of floating-point performance.

Cache-aware matrix factorizations. The Linear Algebra Package (LAPACK) [11] specifies a collection of DLA operations, built on top of the BLAS, with a functionality that is closer to the application level. For example, LAPACK includes driver routines for the solution of linear systems, the calculation of the singular value decomposition, the computation of the eigenvalues of a symmetric matrix, etc.

For multicore processors, the conventional approach to exploit parallelism from LAPACK has relied, for many years, on the use of a multi-threaded instance of the BLAS. This solution exerts a strict control over the data movements and can be expected to make an extremely efficient use of the cache memories. Unfortunately, for complex DLA operations, the exploitation of parallelism only within the BLAS constrains the concurrency that can be leveraged by imposing an artificial fork-join model of execution. Specifically, with this solution, parallelism does not expand across multiple invocations to BLAS kernels even if they are independent and, therefore, could be executed in parallel.

The increase in hardware concurrency of multicore processors in recent years, at the pace dictated by Moore's Law, has led to the development of parallel versions of some DLA operations that exploit *task-parallelism* via a runtime. (See, for example, the efforts with OmpSs [48], PLASMA-Quark [5], StarPU [91] and `libflame`-SuperMatrix [52].) In some detail, these task-parallel approaches decompose a DLA operation into a collection of fine-grained tasks with dependencies, and issue the execution of each task to a single core, simultaneously executing independent tasks on different cores. The runtime-based solutions are better equipped to tackle the increasing number of cores of current and future architectures, because they leverage the natural concurrency that is present in the application. However, with this type of solution, the cores ferociously compete for the shared memory resources and may not amortize completely the overheads of BLAS (e.g., for packing) due to the use of fine-grain tasks.

In the mixed scenario described in the above paragraphs, this dissertation investigates whether it is possible to extract nested task- and loop-parallelism to feed all the cores of a current multi-

socket architecture. The complementary objective is to make an efficient use of the cache hierarchy. In the framework of our study, the target is obviously a DLA operation.

1.2 Contributions

This dissertation makes the following contributions to advance the state-of-the-art on the high performance implementation of DLA operations:

- We introduce efficient multi-threaded implementations of some kernels of the Level-2 BLAS and the complete Level-3 BLAS for asymmetric architectures composed of two types of cores with the following properties:
  - Our solution leverages the multi-threaded implementation of the matrix-matrix multiplication and the sequential implementation of the matrix-vector product kernels in the BLIS library [95], which decompose the operation into a collection of nested loops around a micro-kernel. Starting from these reference codes, we propose a modification of the loop stride configuration and scheduling to distribute the workload comprised by certain loops among the two core types while taking into account their distinct computational performance and cache organization.
  - We demonstrate the generality of the approach by applying the same parallelization principles to develop tuned versions of the BLAS for 32-bit and 64-bit ARM big.LITTLE processors consisting, respectively, of 4+4 (slow+fast) cores and 4+2 (slow+fast cores). The kernels not only distinguish between different operations, paying special care to the parallelization of the triangular system solve, but also take into consideration the operands' dimensions (shapes).
  - The correction of the new asymmetry-aware BLAS is validated by integrating them with the reference implementation of LAPACK from the netlib public repository as well as some new routines for the reduction to compact band forms involved in the solution of symmetric eigenvalue problems and the computation of the singular value decomposition (SVD).
- We propose a novel nested parallelization strategy for dense matrix factorizations that applies a static look-ahead to expose a certain degree of task concurrency in combination with the use of a multi-threaded implementation of the BLAS in order to attain an efficient use of the cache system. In more detail, our solution makes the following specific contributions:
  - We demonstrate the benefits of this nested approach by targeting the LU factorization with partial pivoting, an operation that is representative of many other matrix decompositions which are key for the solution of key DLA problems. Furthermore, our solution can be expected to carry over to any matrix factorization that is composed by a loop that iterates over the column/rows of a matrix, with a loop body that is composed of a "sequential" *panel factorization* and a highly parallel *trailing update*. This is the case of the blocked right-looking variants for the LU, QR, Cholesky and LDL^T factorization and, to some extent, also of some decompositions for the reduction to compact forms that are employed in the solution of symmetric eigenvalue problems and the computation of the singular value decomposition (SVD).
  - We introduce a *malleable thread-level* implementation of BLIS that allows to change the number of threads that participate in the execution of a BLAS kernel at execution

time. This technique departs from the current, inflexible solution adopted in all implementations of the BLAS, which presuppose that, once issued, the number of threads participating in the execution of a BLAS kernel will not vary.

– In case the panel factorization is less expensive than the update of the trailing submatrix, we leverage the malleable instance of the BLAS to improve workload balancing and performance, by allowing the thread team in charge of the panel factorization to be re-allocated to the execution of the trailing update.

– To tackle the opposite case, where panel factorization is more expensive than the update of the trailing submatrix we design an *early termination* (ET) mechanism that allows the thread team in charge of the trailing update to communicate the alternative team of this event.This alert forces an ET of the panel factorization, and the advance of the factorization into the next iteration

1.3 Organization

This chapter presents the motivation and main contributions of this thesis.. In addition, the structure of the whole document is detailed next.

Chapter 2 reviews the current state of the Dense Linear Algebra (DLA) libraries BLAS and LAPACK. The evolution, structure and main features of those libraries are presented in this chapter. The systems used in this thesis, the multi-core Intel Xeon E5-2603 and the asymmetric multi-core processors Odroid-XU3 and Juno are described next. That chapter also includes the description of the PMLib framework (used for the power consumption measurements) and its interaction with tracing and visualization tools.

In Chapter 3, the design, implementation and evaluation of BLAS-2 and BLAS-3 routines for asymmetric multi-core processors is carried out. Routines GEMM and GEMV are studied as the reference kernels for the corresponding levels, BLAS-3 and BLAS-2 respectively. Moreover, we identify the flaws of the initial asymmetry-aware implementation and, after a thorough performance analysis, present the selected solution in each case.

Chapter 4 analyzes several advanced operations for DLA. The results of the direct migration of the Cholesky and LU factorizations relying on the asymmetric-aware version of BLAS-3 are presented. That chapter includes the proposal of a new technique, thread-level malleability, which allows a more efficient use of the resources. The implementation of the technique and evaluation of its benefits are presented there.

In Chapter 5, the study of LAPACK routines in combination with the asymmetry-aware implementation of BLIS is extended in order to expose the benefits of the BLAS-2 asymmetry-aware version. To this end, we evaluate three routines that perform two-sided orthogonal reductions (TSOR) of a dense matrix to a condensed form. In addition, we present a model in order to select specific parameters that may increase the performance of the TSOR. The study presented in this chapter completes the validation of the new asymmetry-aware BLAS, started in Chapter 3 for BLAS-3, by integrating the new version of the library with the reference implementation of LAPACK.

Finally, in Chapter 6, we present the general conclusions of the thesis, the main results obtained as a list of publications, and a collection of open research lines.

CHAPTER 2

Background

In this chapter we offer a short review of the DLA libraries, tools and platforms used in this thesis. DLA operations are classified in two categories: basic and advanced. Basic DLA operations are implemented in Basic Linear Algebra Subprograms (BLAS) library, while LAPACK supports more sophisticated functionality. The first part of this chapter elaborates on the history of both libraries, their structure and main features. Then, a description of the platforms used in the thesis is carried out. Finally the PMLib framework is presented as the framework to collect power samples in our tests.

2.1 Basic Linear Algebra Subprograms (BLAS)

Fundamental problems of linear algebra, such as the solution of linear (equation) systems or the computation of eigenvalues or singular values, underlie a large number of scientific and engineering applications. These problems are found in very different fields such as structure computations, automatic control, integrated circuits design, and chemistry simulations, to name only a few. In the same vein, while solving these problems, a small set of basic operations are often used, e.g., a scalar vector multiplication, a triangular system solve or a matrix-matrix multiplication. In this scenario, BLAS specifies a set of routines that perform these basic DLA operations, defining the functionality and interface of each routine of the library.

The specification of BLAS was carried out by experts of different fields [47], which makes it specially valuable, as a result of an interdisciplinary collaboration. BLAS is a compromise between functionality and simplicity. On one hand, it defines a reasonable number of routines with a limited number of parameters; on the other hand, it provides a wide functionality.

There exists a reference implementation of BLAS published on the Internet [78], but the main strength of BLAS are its different implementations for specific architectures. Since the very beginning, the development of optimized implementations of BLAS routines has been expected of processor manufacturers. As a result, there are propietary implementations from AMD (AMD Core Math Library (ACML)), Intel (MKL), IBM (ESSL) and Nvidia (cuBLAS). In addition, there are also open source implementations such as GotoBLAS, OpenBLAS, ATLAS, and BLIS. In general, these libraries implement each routine taking into account the organization of the target architecture to exploit its resources in order to optimize performance.

The first implementation of BLAS [72] was made in the 1970s and it has been relevant in the solution of dense linear algebra problems ever since. Due to its reliability, robustness and efficiency, several libraries were designed in order to use BLAS routines internally. In addition to reliability and efficiency, BLAS presents the following advantages:

- Code readability: the name of the routines express their functionality.
- Portability and efficiency: when migrating the code to a new platform, the efficiency will remain reasonable as long as a specific version of BLAS for the new architecture is used.
- Documentation: all BLAS routines are thoroughly documented.

The most common implementations of BLAS are implemented in C and Fortran. However, a limited amount of assembly code is sometimes present in order to generate more efficient code.

2.1.1 Levels of BLAS

At the beginning of the development of BLAS, in the early 1970s, the most powerful computers integrated vector processors. The original BLAS were designed with this type of processor as the target, creating a small set of operations that worked on vectors (Level-1 BLAS or BLAS-1). The main objective of that specification was to foster the development of optimized implementations depending on the target architecture. In this way, from the very beginning, the BLAS creators defended the importance of "outsourcing" the implementation of efficient basic operations to the architecture designers, who were able to generate more robust and efficient code [71].

The initial set of routines comprised by the BLAS-1 provided a reduced functionality, since it only supported a limited number of vector operations. In 1987, the BLAS were extended with a new set of routines that constitute the second level of BLAS (BLAS-2). Those new routines implement matrix-vector operations, involving a quadratic amount of arithmetic operations and data. A public reference implementation of the BLAS-2 was released [46]. From that implementation it is worth highlighting the fact that matrix storage is done by columns, following the storage convention of Fortran. As done for BLAS-1, manufacturers were encouraged to create specific implementations and, to this end, some advice were given such as guidelines in order to adapt the inner loop of the routine to the specific architecture, the use of assembly code, or compiler directives for the target architecture.

Eventually, the growing disparity between the processor frequency and the memory bandwidth resulted in the design of architectures with multiple levels of cache memory (memory hierarchy). With the new memory structure, the BLAS-1 and BLAS-2 were not able to attain a reasonable level of performance, since in both cases the ratio between the number of operations and data movement is $O(1)$, while the gap between the processor and memory throughput is much higher. Consequently, the performance of BLAS-1 and BLAS-2 is constrained by the speed of the data transfer from memory.

To tackle this problem, a third set of routines was defined in 1989, the Level-3 BLAS (BLAS-3) [47], that implemented operations with computations of cubic order versus data movements of quadratic order. The difference between the number of computations and memory data transfers, when using a properly designed algorithm, allows to exploit the principle of locality in architectures with a hierarchical organization of the memory, hiding the memory access latency and attaining performances closer to the processor peak. From the algorithmic point of view, this is attained via so-called blocked algorithms. These algorithms divide the matrix into submatrices (or blocks) and carefully orchestrate the movement of these blocks across the memory hierarchy in order to increase

the probability of finding the data in those memory levels that are closer to the processor. Hence, blocked algorithms attain higher performance by making a better use of the memory hierarchy.

As was the case in the previous levels of BLAS, the third level of BLAS presents a compromise between complexity and functionality. An example of simplicity is that there are no specific routines for trapezoidal matrices, since this fact would increase the number of parameters in the routines. However, in order to increase functionality, implicitly transposed operands are considered because, otherwise, the user should transpose the matrix previously and this operation may be highly expensive due to the amount of memory accesses.

In summary, the BLAS routines are organized into three levels, named after the number of computations performed in each one:

- Level 1: Operations with vectors (lineal order of computations).
- Level 2: Matrix-vector operations (quadratic order of computations).
- Level 3: Matrix-matrix operations (cubic order of computations).

Regarding performance, the differences among the three levels is due to the ratio between the number of computations and the amount of data. This ratio is crucial in architectures with a hierarchical organization of memory as, in case the number of operations is higher than the amount of data accessed, it is possible to perform several operations per memory access and, consequently, increase the productivity. Therefore, the three levels are also defined according to the ratio between the number of computations and the data needed by the operation:

- Level 1: The number of computations and data grows linearly with the problem size.
- Level 2: The number of computations and data grows quadratically with the problem size.
- Level 3: The number of computations grows cubically with the problem size, while the amount of data grows quadratically only.

Multi-threaded versions of BLAS for multiprocessors with shared memory deserve special attention. Those versions implement a multi-threaded code that distributes the computations among the processor cores in order to make an efficient use of the resources. Such implementations are especially useful for BLAS-3 routines.

The blocked BLAS-3 routines have two additional advantages [45]. On one hand, they allow different processors to work on distinct blocks simultaneously and, on the other hand, within the same block, operations on different scalars or vectors may be performed simultaneously.

From the performance point of view, the following conclusions can be extracted:

- The performance of BLAS-1 and BLAS-2 is constrained by the memory bandwidth.
- BLAS-3 is the most efficient level, due to the fact that a higher number of computations can be performed per memory access. In addition, the BLAS-3 are more suitable for parallelization, providing significant improvements on multiprocessor architectures with shared memory.

2.1.2 The BLAS library election

Among the great variety of BLAS implementations upon which to build, in this dissertation we selected BLAS-like Library Instantiation Software Framework (BLIS) [95]. This instance of the BLAS, which will be described in detail in Chapter 3, is an open-source framework developed at The University of Texas at Austin that provides a framework to implement BLAS-like operations.

The flexibility of this library is one of the main reasons behind its election, along with the complete exposition of the code that allows the manipulation of the library at different points. Moreover, in terms of performance, BLIS is competitive with other commercial and open-source libraries. Due to these features, BLIS is the most suitable library, among those considered, to carry out the work towards the objectives of this dissertation.

2.2 LAPACK

LAPACK [79] is a library that provides routines to solve fundamental linear algebra problems via the current state in numerical methods. In the same vein as BLAS, LAPACK provides support for dense matrices. However, it tackles more complex problems, for example, systems of linear equations as well as linear least squares (LLS), eigenvalue and singular value problems. LAPACK emerged as the result of a project started in the late 80s. The objective was to obtain a library that gathered the functionalities and improved the performance of EISPACK [54] and LINPACK [39]. Those libraries, designed for vector processors, do not provide acceptable performance on current high performance processors, with segmented pipelines and hierarchical memories. The main reason of their inefficiency is the fact that they are based on BLAS-1 operations, which do not make an optimal use of the hierarchical memory for the reasons exposed earlier in this chapter. The performance improvements of LAPACK arise from a reorganization of the algorithms in order to make an intensive use of the efficient level-3 BLAS.

Regarding the integration of new algorithms, improvements were introduced in almost all linear algebra problems supported by the library, being especially relevant those applied to the solution of eigenvalue problems.

For multiprocessor systems, LAPACK extracts parallelism from within a parallel version of BLAS. In other words, LAPACK routines do not include any type of explicit parallelism in the code, but rely on a multithreaded implementation of BLAS for this purpose.

2.3 Target architectures

2.3.1 Power consumption

Supercomputers are key nowadays in solving challenging problems from very different areas (from engineering design, and financial analysis to disaster prediction); however, they also incur a huge power consumption, not only due to the computation, but also for cooling. For decades, performance and price/performance have been the natural targets of the supercomputing community, looking for the fastest computer (in terms of FLOPS) as advocated in the TOP500 [4] list. In this sense vast efforts have been made and supercomputers' performance has grown from 59.7 billions of floating-point arithmetic operations per second (GFLOPS) in 1993 to 93,014.6 TFLOPS in 2017.

The performance gains in supercomputers has been largely attained via the combined increase of the number of processors in the system, the amount of transistors per processor, and the processor's frequency. Nevertheless, power consumption was not considered as a main factor to take into account until mid of the last decade. The reliable operation of supercomputers depend on the continuous cooling of the machine, which results in enormous costs due to electrical power consumption. For example, the Chinese Sunway TaihuLight, ranked as the top supercomputer in the June 2017 TOP500 list consumes 15 MW. Considering that one MW costs approximately one million dollar per year, the cost of operating and cooling down that system is around 15 million dollars per year.

Energy consumption came officially into the picture a decade ago, when the Green500 [1] list was created. Since that moment, different approaches have been taken in order to improve the energy efficiency of supercomputers. The improvement of energy efficiency has been impressive over the last ten years, moving from 0.4 GFLOPS/W in 2008 to 11.1 GFLOPS/W in 2017. Part of this improvement is due to heterogeneity via hardware accelerators, as the introduction of Graphics Processing Units (GPUs) and manycore processors (e.g., Intel Xeon Phi) added more energy efficient resources to the traditional multi-core systems through the combination of cores of different nature. However, the supercomputing community is still looking for new alternatives in order to reduce energy consumption. For that reason a new approach with low-power architectures has been taken. Fujitsu announced recently the use of ARMv8-based cores for their first exascale supercomputer, the Post-K system [53]. In the same line, the Marenostrum 4 supercomputer [32], from Barcelona Supercomputing Center, will include the same technology as an update of the current supercomputer.

The Mont-Blanc project [3] is the reference example of the low-power architecture trend in supercomputing. This project, started in 2011, advocates for the assembly of supercomputers from low-power processors, more specifically mobile systems-on-chips (SoCs). Several prototypes have been created as an output of the project, starting with Tibidabo in 2011, which features Nvidia Tegra2 SoCs with two ARM Cortex-A9 at 1 GHz. The next prototype was Pedraforca (2013), a moderate-scale prototype with 78 Tegra3 nodes with 4 Cortex-A9 cores running at 1.3 GHz. The last prototype is Mont-Blanc (2015) which contains 1080 Exynos 5250 SoCs organized as 72 compute blades over two racks.

2.3.2 Asymmetric multi-core processors

Nowadays, diversity is the new trend in computing systems. Workloads of different nature run on a wide variety of systems (from mobile devices to datacenters) and require resources with distinct capabilities. Against this background, traditional multi-core systems are not enough; their sophisticated and highly optimized cores are usually underutilized by most workloads yielding a waste of energy. On the other hand, many-core systems that feature simpler cores with low power consumption may not be sufficient for workloads that require high performance.

Heterogeneity emerged as a response to this situation, combining cores of different nature on the same platform in order to provide a variety of resources that meet the necessities of distinct workloads. A specific type of heterogeneity is that provided by AMPs, which combine two (or more) types of cores in the same processor. These platforms provide more flexibility, thanks to their processor specialization, which makes them target either performance or energy efficiency.

However, complexity in AMPs is increasing due to different reasons, such as the variety of cores or the use of different instruction set architectures (ISAs) and, consequently, they are becoming harder to program. Moreover, diversity has made terminology about AMPs confusing due to the fact that many authors use different names to refer to the same type of platform. According to [77], our definition of an AMP matches *Physical asymmetric* in [90] and *Performance asymmetry* in [69]; that is, two sets of cores with same ISA but different microarchitecture.

2.3.2.1 ARM big.LITTLE

All the experimentation carried out throughout this thesis has been performed on a ARM big.LITTLE platform. This technology, designed by ARM, integrates *LITTLE* (or slower and energy-saving) processor cores with *big* (or more powerful and power-hungry) cores. Both types of processors are memory coherent and share the same ISA, but feature different microarchitectures

	Armv7-A/R	Armv8-A/R	Armv8-A
		AArch32	AArch64
Floating-point	32-bit	16-bit*/32-bit	16-bit*/32-bit/64-bit
Integer	8-bit/16-bit/32-bit	8-bit/16-bit/32-bit/64-bit	8-bit/16-bit/32-bit/64-bit

Table 2.1: Size of NEON operations on ARM architectures.

in order to implement their performance and power characteristics required by the big.LITTLE concept. The big cores are more complex, apply out-of-order execution, and have a multi-issue pipeline. The LITTLE cores are simpler, their execution is in-order, and have a simple multistage pipeline.

The big.LITTLE platforms used in our tests include ARM NEON [12] technology, a Single Instruction Multiple Data (SIMD) architecture extension. In this technology, registers are considered vectors of elements (of the same data type) that are processed simultaneously increasing the performance of the processor. The data types and the architectures that support them are presented in Table 2.1 (values marked with * refer to the ARMv8.2-A architecture).

2.3.2.2 Odroid

ODROID refers to a series of single-board computers manufactured by Hardkernel Co. Ltd. that include some implementations of the big.LITTLE design announced by ARM in 2011. The first generation of ODROID that implemented the big.LITTLE architecture (ODROID-XU) was launched in 2013 and combined a single Cortex-A15 quad-core and a single Cortex-A7 quad-core (Samsung Exynos 5410). This platform did not allow the simultaneous operation of both clusters, an issue that was solved later with the ODROID-XU3.

For the tests presented in this thesis using an ODROID platform, we used either an ODROID-XU3 or an ODROID-XU4 both furnished with a Samsung Exynos 5422 SoC. The ODROID-XU4 is the evolution of the ODROID-XU3, with similar performance and energy efficiency hardware, but smaller board. Moreover, the ODROID-XU3 includes power sensors that are not present on the ODROID-XU4 board. Hereafter ODROID will be used to refer indistinctly to any of these platforms.

The ODROID SoC comprises an ARM Cortex-A15 quad-core processing cluster (big) plus an ARM Cortex-A7 quad-core processing cluster (LITTLE), both implementing the ARMv7 microarchitecture; that is a 32-bit architecture. Each Cortex core has its own private 32-Kbyte L1 (data) cache. The four ARM Cortex-A15 cores share a 2-Mbyte L2 cache, and the four ARM Cortex-A7 cores share a smaller 512-Kbyte L2 cache; see Figure 2.1. Moreover, the two clusters access a common 2-Gbyte DDR3 RAM. Additionally, the Cortex-A7 cores may operate from 200 MHz to 1.4 GHz, while the Cortex-A15 may work from 200 MHz to 2 GHz.

2.3.2.3 Juno

Juno is the first ARMv8-A 64-bit development board featuring a Cortex-A57 dual-core processing cluster and a Cortex-A53 quad-core processing cluster. Each Cortex-A57 core has a private 48+32-Kbyte L1 (instruction+data) cache, while each Cortex-A53 core has a private 32+32-Kbyte L1 (instruction+data) cache. The four Cortex-A57 cores share a 2-Mbyte L2 cache and the four Cortex-A53 cores share a smaller 1-Mbyte L2 cache; both clusters access a common 8-Gbyte DDR3 RAM (Figure 2.2). Furthermore, the Cortex-A53 cluster may operate from 450 MHz to 850 MHz and the Cortex-A57 from 450 MHz to 1.1 GHz.

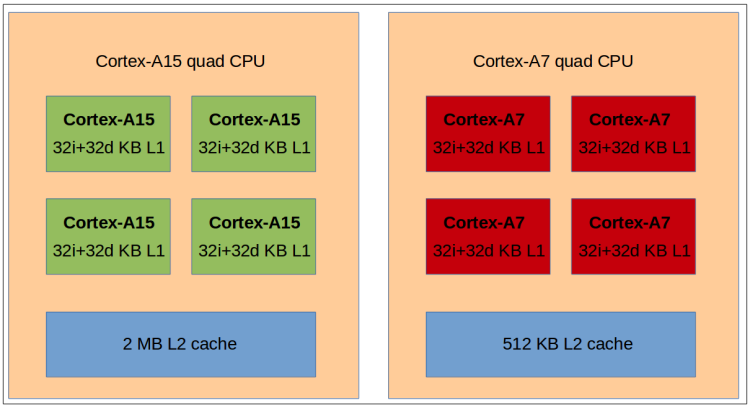

Figure 2.1: Exynos 5422 block diagram.

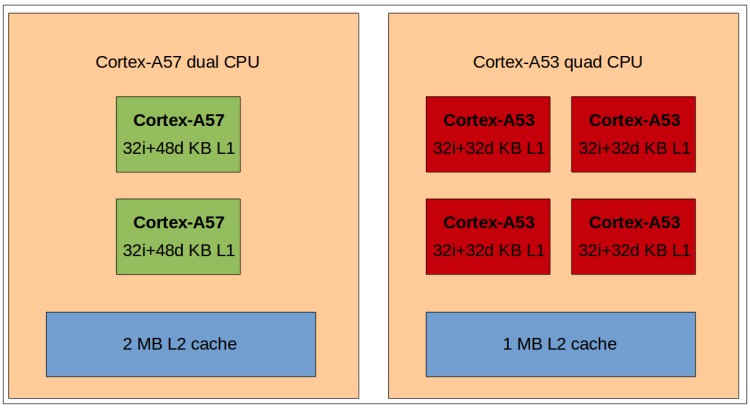

Figure 2.2: ARM Juno SoC block diagram.

2.3.3 Intel

For experiments with the thread-level malleability technique, we have used an Intel Xeon E5-2603 v3. This platform features two homogeneous sockets with 6 cores each. Each core may run in the frequency range 1.2-2.4 GHz, has a private 32 KByte L1 cache, and a private 256 KByte L2 cache. All the cores in the same socket share a 20 MByte L3 cache. Moreover, the platform is furnished with 64 GB of DDR3 RAM (Figure 2.3).

Although this platform is homogeneous or symmetric, it has been used to illustrate the advantages of thread-level malleability in a simpler way. However, the initial approach tested on this platform was adapted in order to be ported to the ODROID platforms as a part of this dissertation.

2.4 Measuring power consumption

Obtaining power dissipation measurements has been key in order to analyze the behavior of the target DLA library on the different platforms. To do so, we have used PMLib, a portable framework developed at Universitat Jaume I designed to interact with power measurement units.

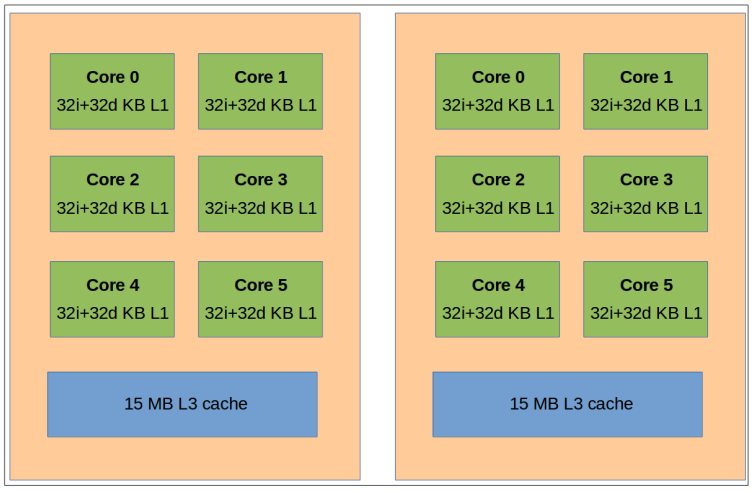

Figure 2.3: Intel Xeon block diagram.

The current implementation of PMLib provides an interface to access internal/external Wattmeters and a number of tracing tools. Power measurement is retrieved by the application using a collection of routines that allow the user to retrieve information from the power measurement units, create counters associated with a device where power data is stored, start/continue/terminate power sampling, etc. All this information is managed by the PMLib server, which is in charge of acquiring data from the devices and sending back the appropriate answers to the invoking client application via the proper PMLib routines. The PMLib server is implemented as two different types of daemons in the framework: the external-PMLib server and the internal-PMLib server. The former runs in a separate system and collects power samples from one or more Wattmeters attached to the target platform where the application is running. In this case interferences from the running daemon are avoided and noise is removed from the power measurements. However, there is some information that is only accessible locally (like, e.g., power measurements from local sensors). In those situations the use of the internal-PMLib daemon running in each node of the target platform is used.

Figure 2.4 illustrates the PMLib framework and its interaction with the target application and different tools, such as a performance tracing suite and a visualization tool. The starting point is a scientific application, instrumented with PMLib, that runs on the target platform. Attached to the application node(s) there are different internal/external power measurement devices (managed by the external-PMLib daemon). The users' code running on the target platform makes calls to the PMLib Application Programming Interface (API) to instruct the external-PMLib server [14] to do distinct operations (e.g., start/ stop collecting the data captured by the power measurement devices). Once the users' application is executed, and thanks to the smooth integration of the PMLib traces with different tracing suites (e.g., Extrae [87], TAU [88], VampirTrace [61], HDTrace [70]), the power trace can be inspected along with performance traces obtained from the tracing suites with the appropriate visualization tool (e.g., Paraver [82]).

2.4.1 PMLib and big.LITTLE platforms

As explained in the previous section, PMLib is composed by two daemons. The external-PMLib daemon is in charge of all the hardware mesuring devices, which can be either internal Direct

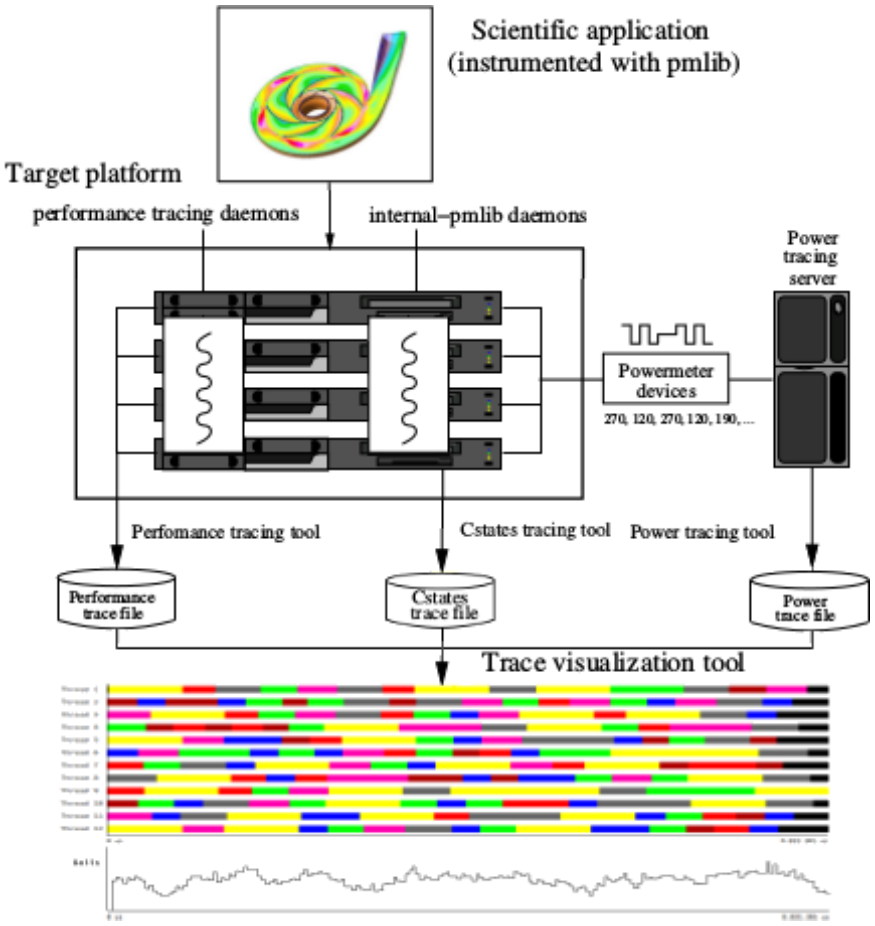

Figure 2.4: PMLib diagram.

Current (DC) or external Alternating Current (AC) Wattmetters. On the other hand, the internal-PMLib server manages information that can be retrieved only locally; this information includes the C-states of the processor and power measurements from energy sensors (e.g., RAPL [65], NVML counters [80], ARM current sensors [62]).

Power samples on the big.LITTLE ODROID-XU3 platform are collected in this thesis using the internal-PMLib daemon and the current/voltage sensors present in this SoC. There exist four sensors on the ODROID-XU3 that measure the power consumption of the Cortex-A15 cluster, Cortex-A7 cluster, GPU and DRAM respectively every 250 ms. The PMLib framework retrieves power consumption samples from this platform at 4Hz, and dumps them to the standard output or a text/paraver file.

2.5 Summary

In this chapter, we presented the BLAS and LAPACK DLA libraries, which are used throughout this thesis. We also described the platforms used in our experiments, two ARM platforms with different architectures and distinct architectures. Finally, we reviewed PMLib, a framework designed to collect power samples from a wide variety of Wattmeters.

CHAPTER 3

Basic Linear Algebra Subprograms (BLAS)

In this chapter, a wide description of two (BLAS-2 and BLAS-3) out of the three BLAS levels is performed for the BLIS library. This specific library is used for the study since its structure allows to adopt a low level approach and extract detailed information about the source of the costs of the operations, something that is not possible with other solutions that follow a black-box approach,.e.g., MKL. An adaption of the library for AMPs is presented along with a detailed study of the performance and energy consumption gains derived from this new version. That implementation includes a full asymmetry-aware BLAS-3 version and two asymmetry-aware routines (SYMV and GEMV) from BLAS-2. The asymmetry-aware version of the library is tested on two recent ARM big.LITTLE SoCs, both implementing a particular class of heterogeneous architecture that combines a few high performance (but power hungry) cores with a collection of energy efficient (though slower) cores. In order to carry out a complete study, different operand shapes and precisions are used in the tests.

3.1 Blas-like Library Instantiation Software (BLIS)

BLIS is a framework for instantiating the functionality of BLAS that increases the productivity of developers when implementing BLAS and BLAS-like operations. In the same vein as BLAS, BLIS is organized in three levels (hereinafter named as BLIS-x), plus 4 sublevels in BLIS-1, that provide the same functionality as BLAS, but also offers additional options that allow the user to implement tuned operations thanks to its open source philosophy:

- Level-1 is composed of 4 sublevels that tackle operations with operands of different nature:
  - Level-1v targets operations on vectors.
  - Level-1d tackles element-wise operations only on matrix diagonals.
  - Level-1m performs element-wise operations on matrices.
  - Level-1f focus on fused operations on multiple vectors, meaning that one call to the routine performs the specific operation on f vectors simultaneously.
- Level-2 is in charge of operations with one matrix and (at least) one vector operand.

- Level-3 deals with operations that take one or more matrices as operands.

BLIS follows the GotoBLAS approach [57] but further modularizes operations or kernels in order to isolate a micro-kernel. This micro-kernel performs the targeted operation and is written in assembly or using vector intrinsics, bringing out the most of the underlying architecture. Thus, to guarantee portability, the developer only needs to implement the micro-kernel in order to obtain an implementation for a different architecture. In addition, if high-performance is desired, a few configuration parameters need to be adjusted to optimize the micro-kernel and the loops around it.

The configuration parameters of BLIS are m_r and n_r, referring to register configuration parameters, and m_c, k_c and n_c, identified as cache configuration parameters. Register configuration parameters are essential in the implementation of the micro-kernel and their values depend on the throughput and latency of the SIMD units and the number of available registers. Moreover, the cache configuration parameter k_c is also crucial for the micro-kernel, since it defines the number of updates that the micro-kernel performs. k_c along with m_c and n_c dictate the strides of the loops around the micro-kernel and their value depends on the size and associativity degree of the cache levels. These cache configuration parameters are also used by the packing routines, which are in charge of arranging data in memory in order to minimize cache misses. Given an input matrix and the cache configuration parameters, the packing routine produces as an output a submatrix (or *micro-panel*) of the dimension dictated by the cache parameters with its values arranged properly. The goal, thus, is that the micro-panels obtained as the output of the packing routines fit in the appropriate cache levels in order to fully amortize these transfers with enough computation from within the micro-kernel, as explained in [73].

Moreover, BLIS offers multiple levels of multithreading for nearly all level-3 operations via OpenMP or POSIX threads. This multi-level parallelism allows the partitioning of the matrices in several dimensions to attain high performance on multi-core architectures. In order to encode the information about the logical thread topology, the execution of all routines is encoded as a *control-tree*, a recursive data structure that holds all the information necessary to combine the basic building blocks offered by BLIS. The control-tree for a given BLAS-3 operation governs, among others, which combination of loops have to be executed to complete the operation (that is, the exact *algorithmic variant* to execute at each level of the general algorithm), the stride for each loop (specific to each target architecture), and the exact points at which packing must occur. In BLIS, there is a single control-tree per operation composed by as many recursive levels as loops comprise the operation. Thus, when executing a BLIS-3 kernel, at a given level of the algorithm, the control-tree indicates the loop stride of the current loop; whether any of the input operands has to be packed at that point; the parallelism degree of the current loop; whether any of the input operands needs to be transposed; and the next step of the algorithm (next loop or the micro-kernel instantiation).

3.2 BLIS-3

In this section, we present the level-3 BLAS in BLIS (hereafter BLIS-3) and a detailed description of its multi-threaded implementation. Given that all BLIS-3 kernels rely on the general matrix multiplication (GEMM) , this operation is explained in the first place. Next, we describe how the same idea presented for the GEMM is extended to the rest of BLIS-3 operations. Finally, we describe the triangular system solve (TRSM) due to its specific features.

3.2.1 General matrix-matrix multiplication (GEMM)

GEMM is a crucial operation for the optimization of the Level-3 BLAS [47], as portable and highly tuned versions of the remaining Level-3 kernels are in general built on top of GEMM. As exposed in [68], all BLAS-3 kernels can be implemented by means of GEMM if a suitable partitioning of the matrices is performed, since GEMM is applied to some of these partitions, and the "non-GEMM-wise" partitions can be handled by Level-1 and/or Level-2 kernels.

Modern high-performance implementations of GEMM for general-purpose architectures , including BLIS and OpenBLAS, follow the design pioneered by Goto [59]. BLIS in particular implements the GEMM $C = \alpha \cdot A \cdot B + \beta \cdot C$, where the sizes of A, B, C are respectively $m \times k$, $k \times n$, $m \times n$, as three nested loops around a *macro-kernel* plus two packing routines (see Loops 1–3 in Figure 3.1). The macro-kernel is then implemented in terms of two additional loops around a *micro-kernel* (Loops 4 and 5 in Figure 3.1). In BLIS, the micro-kernel is typically encoded as a loop around a rank–1 (i.e., outer product) update using assembly or with vector intrinsics, while the remaining five loops and packing routines are implemented in C.

Figure 3.2 illustrates how the loop ordering, together with the packing routines and an appropriate choice of the BLIS cache configuration parameters orchestrate a regular pattern of data transfers through the levels of the memory hierarchy. In practice, the cache parameters n_c, k_c and m_c, and the register parameters n_r and m_r dictate the strides of the five outermost loops with the goal of streaming a $k_c \times n_r$ micro-panel of B_c, say B_r, and the $m_c \times k_c$ macro-panel A_c into the FPUs from the L1 and L2 caches, respectively; while the $k_c \times n_c$ macro-panel B_c resides in the L3 cache (if present).

```
Loop 1   for jc = 0,...,n − 1 in steps of nc
Loop 2     for pc = 0,...,k − 1 in steps of kc
             B(pc : pc + kc − 1,jc : jc + nc − 1) → Bc                    // Pack into Bc
Loop 3       for ic = 0,...,m − 1 in steps of mc
               A(ic : ic + mc − 1,pc : pc + kc − 1) → Ac                  // Pack into Ac
Loop 4         for jr = 0,...,nc − 1 in steps of nr                       // Macro-kernel
Loop 5           for ir = 0,...,mc − 1 in steps of mr
                   Cc(ir : ir + mr − 1,jr : jr + nr − 1)   // Micro-kernel
                     += Ac(ir : ir + mr − 1,0 : kc − 1)
                     ·  Bc(0 : kc − 1,jr : jr + nr − 1)
                 endfor
               endfor
             endfor
           endfor
         endfor
```

Figure 3.1: High performance implementation of GEMM in BLIS. In the code, $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is 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.

The multi-threaded version of GEMM integrated in BLIS exploits the concurrency available in the nested five-loop organization at one or more levels (i.e., loops). Furthermore, the approach takes into account the cache organization of the target platform (e.g., the presence of multiple sockets, which cache levels are shared/private, etc.), while discarding the parallelization of loops that would incur race conditions as well as those that exhibit too-fine granularity. The insights gained from the analyses carried out in [94, 89] about the loop(s) to be parallelized in a multi-threaded implementation of GEMM can be summarized as follows:

- Loop 5 (indexed by i_r). With this option, different threads execute independent instances of the micro-kernel, while accessing the same micro-panel B_r in the L1 cache. The amount of

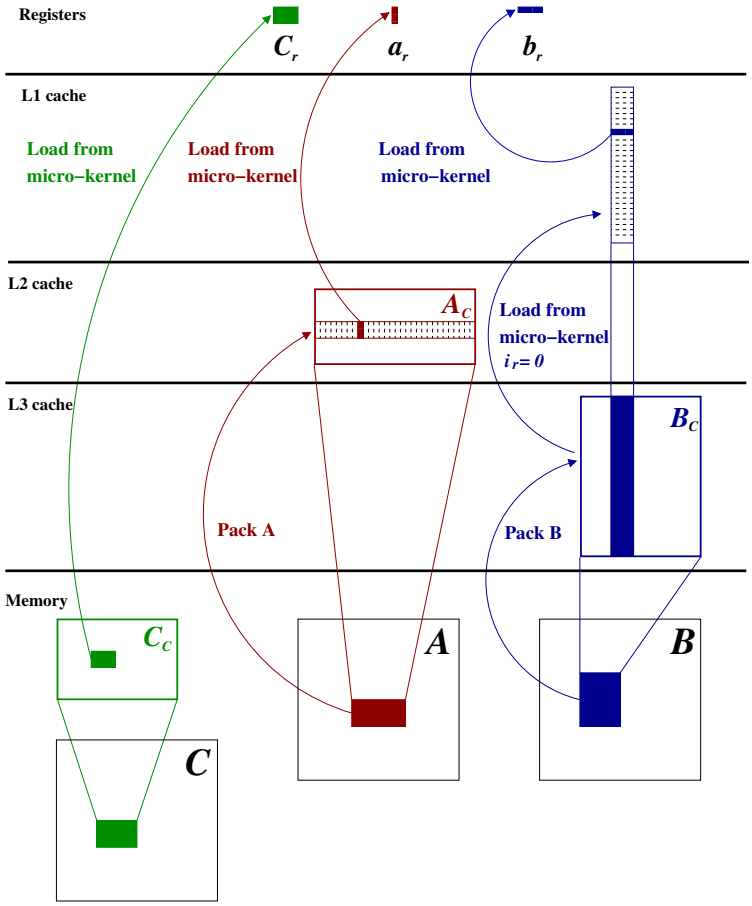

Figure 3.2: Data movement involved in the BLIS implementation of GEMM.

parallelism in this case, $\lceil \frac{m_c}{m_r} \rceil$, is scarce as, for many architectures, the optimal value for m_c is in the order of a few hundreds, and $m_r \in \{4, 8\}$ in general. Thus, if parallelized, the cost of moving B_r into the L1 cache may not be amortized over a sufficiently large number of flops.

- Loop 4 (indexed by j_r). Different threads operate on independent instances of the micro-kernel, but access the same macro-panel A_c in the L2 cache. The time spent in this loop amortizes the cost of packing (and, therefore, moving) A_c from main memory into the L2 cache. The amount of parallelism, $\lceil \frac{n_c}{n_r} \rceil$, is in general larger than in the previous case, as n_c is in the order of several hundreds up to a few thousands for many architectures, and often $n_r \in \{4, 8\}$.

- Loop 3 (indexed by i_c). Each thread packs a different macro-panel A_c into the L2 cache and executes a different instance of the macro-kernel. The number of iterations of this loop is not constrained by the cache parameters, but instead depends on the problem dimension m. When m is less than the product of m_c and the degree of parallelization of the loop, A_c will be smaller than the optimal dimension and performance may suffer. 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 in order to guarantee that A_c still fits in the appropriate cache level. However, reducing m_c is equivalent to parallelizing the first loop around the micro-kernel.

- Loop 2 (indexed by p_c). This is not a good choice because multiple threads simultaneously update the same parts of C, requiring a mechanism to prevent race conditions.
- Loop 1 (indexed by j_c). From a data-sharing perspective, this option is equivalent to extracting the parallelism outside of BLIS. This parallelization is reasonable in a multi-socket system where each CPU (socket) has a separate L3 cache.

To summarize, these are general guidelines to decide which loops are good candidates to be parallelized in order to fully exploit the cache hierarchy of a target architecture. At a glance, the appropriate combination of loops to parallelize strongly depends on which caches are private or shared. Usually, Loop 1 is a good candidate in a multi-socket platform with on-chip L3 caches; Loop 3 should be parallelized when each core has its own L2 cache; and Loops 4 and 5 are convenient choices if the cores share the L2 cache.

3.2.2 Generalization to BLAS-3

The specification of the BLAS-3 [47] basically comprises 9 kernels offering the following functionality:

1. Compute the (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 Hermitian operators, providing the same functionality as that of SYMM, SYRK, and SYR2K for symmetric inputs.

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 $n \times n$	$m \times n$	–
HEMM	$C := C + AB$ or $C := C + BA$	Hermitian $m \times m$ Hermitian $n \times n$	$m \times n$	$m \times n$
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	$C := C + A^T B + B^T A$	$k \times n$	$k \times n$	$n \times n$
HERK	$C := C + A^H A$	$k \times n$	–	$n \times n$
HER2K	$C := C + A^H B + B^H A$	$k \times n$	$k \times n$	$n \times n$

Table 3.1: Kernels of BLIS-3

In BLIS, all BLAS-3 kernels (Table 3.1) are implemented following the same structure as that presented for GEMM, featuring three nested loops that enclose two packing routines and a *macro-*

kernel. The macro-kernel is in turn implemented in terms of two additional loops around a *micro-kernel*. This GEMM micro-kernel is used by all BLIS-3 operations, since it provides an optimal implementation for the given architecture. Specific cases, such as symmetric, Hermitian and/or triangular matrices are tackled in the macro-kernel (Loops 4 and 5) by adapting the loop strides according to the operation parameters. For example, if the input matrix is symmetric, the macro-kernel will be the same as in the GEMM case, but it will be applied only to half of the matrix, consequently, the loop strides will need to be modified accordingly.

Regarding the multi-threaded implementation, the same conclusions extracted for GEMM can be applied to the rest of BLIS-3. The only exception is TRSM which will be discussed in the next section. This means that, thanks to the common loop structure of all BLAS-3 operations and the mechanisms provided by BLIS to this end, the user can parallelize one or more loops simultaneously in any BLAS-3 kernel. More precisely, for multi-threaded BLIS implementations, the control-tree will define which loops need to be parallelized and the level of concurrency to extract at each point of the algorithm.

3.2.3 Triangular system solve (TRSM)

As explained in previous sections, all BLIS-3 operations leverage the GEMM micro-kernel. The triangular system solve (TRSM) needs, in addition, a specific micro-kernel to deal with the diagonal blocks, where the triangular system is solved. Furthermore, this operation has some dependencies that shrink the options when parallelizing multiple loops.

In BLIS, the execution of a triangular system solve $A \cdot X = \alpha B$, where A is an $m \times m$ upper (or lower) triangular matrix, B is the $m \times n$ right-hand side matrix, and the $m \times n$ X is the sought-after solution requires both the invocation of the GEMM *micro-kernel* and the instantiation of a tailored GEMM *micro-kernel* (hereinafter TRSM *micro-kernel*) that performs the operation when the blocks that comprise the diagonal of A are involved. This specific TRSM *micro-kernel* performs the triangular solve with multiple right-hand sides and, in combination with the optimized GEMM *micro-kernel*, provides an accelerated implementation of TRSM. Nevertheless, a specific *micro-kernel* that combines a GEMM and a TRSM subproblem in order to perform the triangular system solve (known as fused GEMM-TRSM *micro-kernel*) can be implemented for the selected architecture. Between these micro-kernels the best option is to use the fused GEMM-TRSM *micro-kernel* (if provided for the architecture) because, in combination with the optimized GEMM *micro-kernel* needed by the blocks that not comprise the diagonal, will provide a fully optimized version of the TRSM thanks to the elimination of redundant memory operations that occur if two independent *micro-kernels* are invoked. In this sense, in TRSM operand B is completely packed while the amount of packed bytes of A is only half of those featured by GEMM.

Regarding the parallelization of TRSM, due to the data dependencies present in the kernel, the previous analysis presented for GEMM is not valid. For those systems where the triangular factor appears in the left-hand side of the operation, only Loops 1 and/or 4 can be parallelized. On the contrary, if the triangular operand lies on the right-hand side of the operation, only Loops 3 and/or 5 can be targeted. If a different parallelization was performed, distinct threads would be updating the same positions of matrix B incurring in race conditions.

3.3 BLIS-3 on ARM big.LITTLE

Given that BLIS does not provide a specific implementation for AMPs, the following experiment was designed to analyse the behavior, when combining the existing implementation of BLIS with an AMP (the Exynos 5422 SoC in our case), in terms of performance and energy, using GEMM as

an example. For the evaluation, given the guidelines in Section 3.2 and the lack of an L3 cache in this chip, the following two-level parallelization strategy is adopted:

- Coarse-grain (or inter-cluster): Loop 1 is tackled using *2-way parallelism* to statically distribute its iteration space between the two clusters. This loop (and also Loop 3) is a good candidate for parallelization across cores with a proprietary and isolated L2 cache, as is the case of each cluster in the Exynos 5422 SoC.

- Fine grain (or intra-cluster): Loop 4 is parallelized using *4-way parallelism* to statically distribute its iteration space among the four cores of the same cluster. This loop (as well as Loop 5) is a good candidate for parallelization across cores sharing a common L2 cache, as is the case of cores in the same cluster of the Exynos 5422 SoC.

In addition, the cache configuration parameters are set to those that are optimal for the Cortex-A15. Note that similar qualitative observations were obtained when parallelizing the alternative three combinations of Loops 1/3 and 4/5; and/or when the cache parameters were configured using the optimal values for the Cortex-A7.

Figure 3.3 illustrates the implications of the default BLIS scheduling strategy in terms of loop partitioning and assignment to threads. In total, eight threads (one per core) are created and bound to the cores so that in overall *8-way parallelism* is extracted within BLIS. As shown in the figure, the iterations of Loop 1 are distributed between two teams of threads. Each team is mapped to a cluster (big in green and LITTLE in red) and all its members are in charge of executing the assigned iterations, $n/2$ in this case. Then, in Loop 3, all threads execute m iterations. In Loop 4, each single thread constitutes a different team, which is in charge of $n_c/4$ iterations (note that, in this case, two blocks of n_c iterations are being processed by the two clusters). Finally, in Loop 5, all threads take part of the same team again and perform m_c iterations each. Here is important to emphasize that the iteration space for all loops is homogeneously distributed across the cores (i.e., without taking into account the core type).

Figure 3.4 reports the performance in GFLOPS and energy efficiency in GFLOPS per Watt (GFLOPS/W) using the (two-level) symmetric-static scheduling that parallelizes Loops 1 and 4. For reference, we also include the results from a parallelization of Loop 4 that separately exploits either the four cores in the Cortex-A15 cluster or the four cores in the Cortex-A7 cluster. The "Ideal" line in the performance graph corresponds to the aggregated performance of the configurations that use four cores of each of the two types in isolation (i.e., the performance of the four Cortex-A15 cores plus the performance of the four Cortex-A7 cores). The "Ideal–combined" line in the energy graph is an upper bound on the energy efficiency when both clusters are in operation. This is obtained using PMLib (Section 2.4) and calculated as the aggregated performance of both clusters (Ideal) divided by the power dissipated by the Cortex-A15 cores and Cortex-A7 cores when working in isolation plus the power of the memory and GPU. As expected, this curve for the ideal energy efficiency is above the results obtained when using only Cortex-A15 cores, but below the counterpart that only employs the Cortex-A7 cores. These ideal curves represent theoretical upper bounds for the performance and energy efficiency that can be attained when using an optimal scheduling strategy that exploits the asymmetry of the architecture.

This experiment reveals that a naive symmetric-static workload distribution, which does not consider the differences in the cache hierarchy or performance between the Cortex-A15 and the Cortex-A7, exploits the full system (8 cores) to deliver only about 40% of the highest performance that is observed when employing only the four Cortex-A15 cores. The reason is that, with this approach, BLIS performs a static partitioning and mapping of the iteration space to the processing cores in a homogeneous manner. This causes a severe workload imbalance, as the threads running

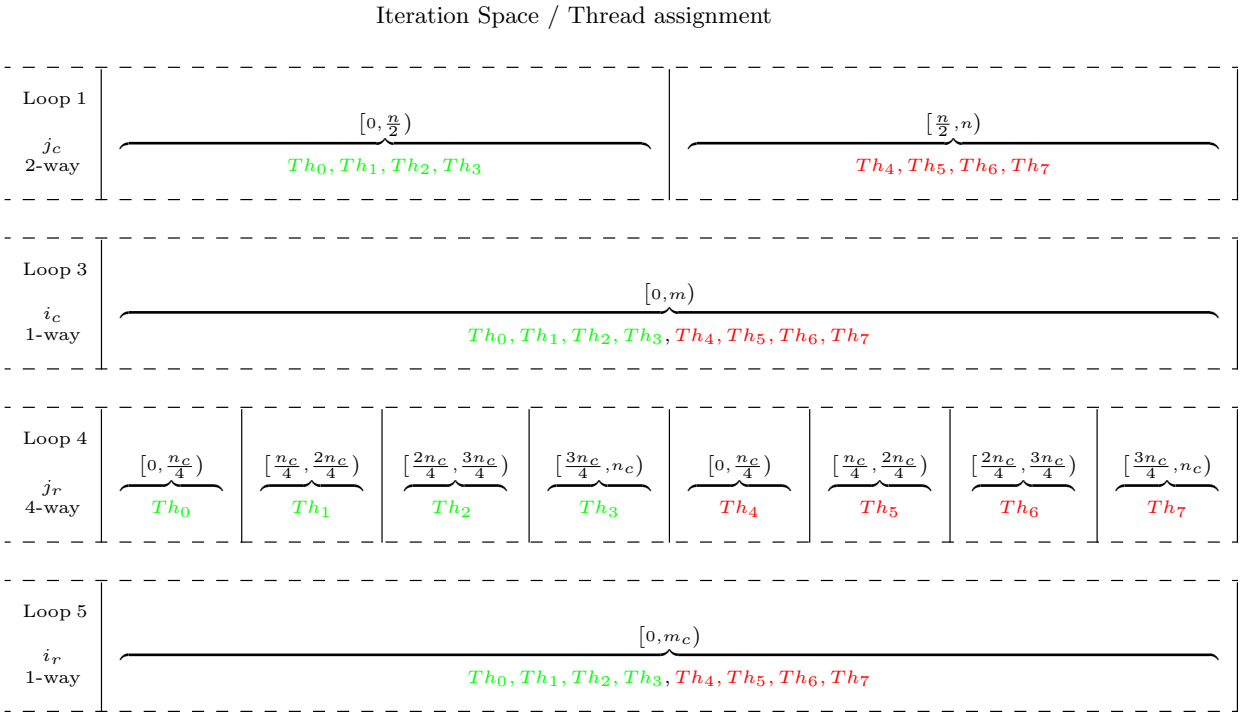

Figure 3.3: Partitioning of the iteration space and assignment to threads/cores for a multi-threaded BLIS implementation with 8-way parallelism that combines 2-way parallelism from Loop 1 and 4-way parallelism from Loop 4. Threads in green and red are respectively mapped to big and LITTLE cores.

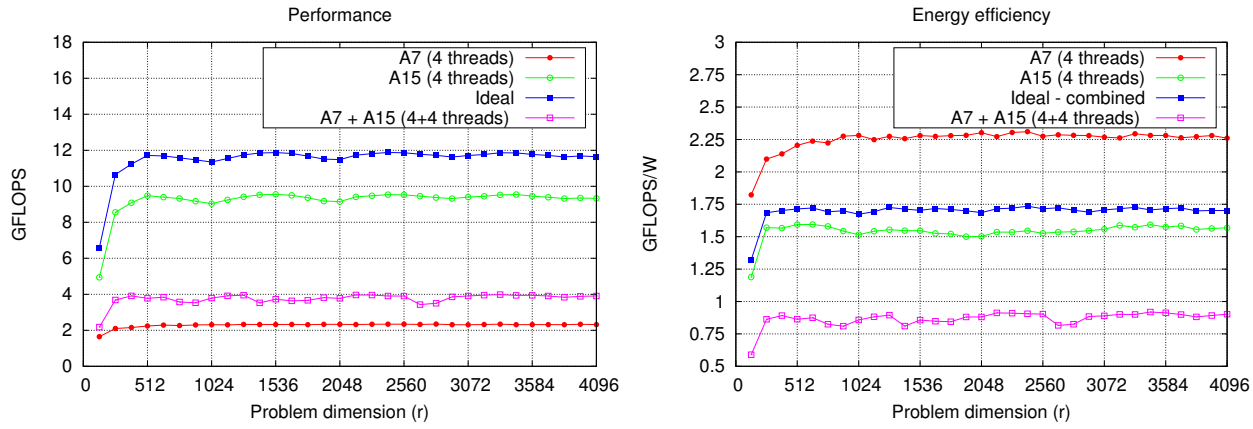

Figure 3.4: Performance (left) and energy efficiency (right) of the reference BLIS GEMM using exclusively one type of core in isolation, and the symmetric-static scheduling version with a coarse-grain parallelization of Loop 1 and the fine-grain parallelization of Loop 4 using 4 threads per cluster.

on the Cortex-A15 rapidly process their chunks, but then have to wait for the threads running on the slow Cortex-A7 cores to complete their work. The energy efficiency of the naive solution is also dramatically affected, and this configuration delivers the worst energy results.

In view of the results, we can state that the default approach adopted by BLIS to map BLAS-3 kernels on a multi-threaded CPU presents two main drawbacks when applied to leverage the asymmetric cores of an AMP:

- BLIS relies on a static partitioning and mapping of the loop iteration space among the threads, oblivious of the computational power of the cores these iteration chunks are assigned to. Therefore, independently of the chunk size and the specific loops that are parallelized, this strategy can only yield an unbalanced distribution of the workload (basically, the micro-kernels) among the asymmetric cores.

- In addition, BLIS employs constant values for the loop strides that, in order to attain high performance, need to match the optimal configuration parameters determined by the core cache organization. However, given that the system presents two different architectures (Cortex-A15 and Cortex-A7), and thus two distinct optimal cache parameters, ideally one should select different loop strides/configuration parameters for each type of core.

In conclusion, this experiment naturally motivates the need of an efficient alternative to the homogeneous symmetric-static scheduling partitioning of the iteration space integrated in the original multi-threaded implementation of BLIS GEMM.

3.4 Optimizing symmetric BLIS GEMM on big.LITTLE

In order to implement an asymmetry-aware version of BLIS-3, it is important to carry out a comparative study that makes it possible to measure how well BLIS will perform when extracting the most of all the cores in the SoC. To this end, it is essential to analyze the behavior of both architectures present in the platform, Cortex-A7 and Cortex-A15 in isolation.

3.4.1 Cache optimization for the big and LITTLE cores

An initial step in order to attain high performance with the implementation of BLIS GEMM requires the implementation of a specific *micro-kernel* for the target core in order to bring out the best of the SIMD units and available registers. This optimization was unnecessary in this work because this type of micro-kernels were already available in BLIS. After that, given a target precision (single or double), its essential to determine the configuration parameters n_c, k_c, m_c, n_r, and m_r for a single ARM core of each type, Cortex-A15 and Cortex-A7, that match the cache organization. The experimental effort towards this goal using IEEE 754 double-precision arithmetic is described next. The study in [73] shows that, in principle, this optimization is also possible via analytic derivation. According to that work, the optimal values for the register parameters m_r and n_r can be calculated taking into account the number of elements that can be hold in the vector registers, the throughput of the SIMD fused multiply-add instructions and their latency. Moreover, m_c, k_c and n_c can be calculated from the size of the caches, the replacement cache policy and the cache organization. As the authors probe in that work, their model provides very accurate results for double precision, hitting or providing a value very close to that determined by the expert developers.

The first aspect to note is that, in this architecture for double precision, n_c plays a minor role and, therefore, can be simply set to $n_c = 4{,}096$. This is explained because, in BLIS, n_c is connected to the dimension of the L3 cache, which is not present in the Exynos 5422 SoC. Furthermore, the micro-kernels for these core architectures and precision are thoroughly tuned with $m_r = 4$

and $n_r = 4$. In consequence, the optimization of GEMM in a single-core scenario boils down to determining the optimal values of m_c and k_c for each type of core. For this purpose, independent empirical searches using a single Cortex-A15 core and a single Cortex-A7 core were performed. In both cases, a coarse-grain search to detect potential optimal regions was initially applied, and the selected regions were further explored next with a finer granularity to detect the optimal configuration parameters. The result of this process is illustrated in Figure 3.5, where the top and bottom plots correspond to the coarse search and the fine-grain refinement, respectively.

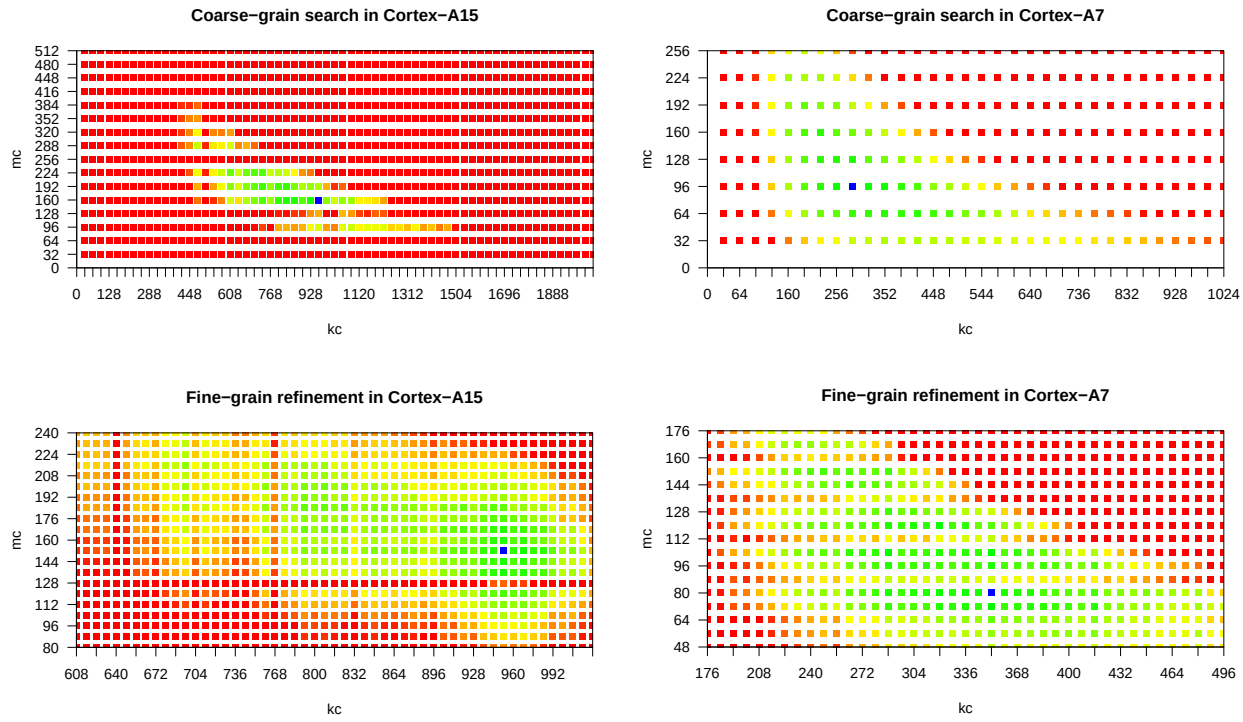

Figure 3.5: BLIS optimal cache configuration parameters m_c and k_c for the ARM Cortex-A15 (left) and Cortex-A7 (right) in the Samsung Exynos 5422 SoC. The performance ranges from red (lowest GFLOPS) to green (highest GFLOPS); the optimal (m_c, k_c) pair is marked as a blue dot.

The optimal configurations were detected at $m_c = 152$, $k_c = 952$ for the Cortex-A15 core and $m_c = 80$, $k_c = 352$ for the Cortex-A7 core. As could be expected, the optimal values for the Cortex-A15 core are larger than those of the Cortex-A7 core, since the L2 cache of the former is four times bigger. For both types of cores, the corresponding dimensions and the associative-degree of the caches ensure that the micro-panel B_r ($k_c \times n_r$) fits into the L1 cache while the macro-panel A_c ($m_c \times k_c$) resides into the L2 cache. Note that these values were obtained for a single-threaded version; consequently, when using several threads, the values may change in order to ensure that several blocks should fit in a specific cache level.

3.4.2 Multi-threaded Evaluation of BLIS GEMM on the big and LITTLE clusters

After determining the optimal configuration parameters for each core cache organization, we analyze the performance and energy efficiency of a multi-threaded implementation of BLIS GEMM

that operates in a homogeneous (symmetric) system consisting of up to four cores from either the Cortex-A15 cluster or the Cortex-A7 cluster.

In particular, given the guidelines exposed in Section 3.2, and the fact that the L2 cache is shared among the cores of the same cluster, we adopt a static parallelization of Loop 4 using 1–4 threads/cores. Similar qualitative conclusions were obtained from a static parallelization of Loop 5. In this section the two types of clusters are evaluated in isolation, so the scalability of the multi-threaded GEMM can be evaluated. In addition, the results of this experiment reveal the differences in performance and energy efficiency between the two types of cores.

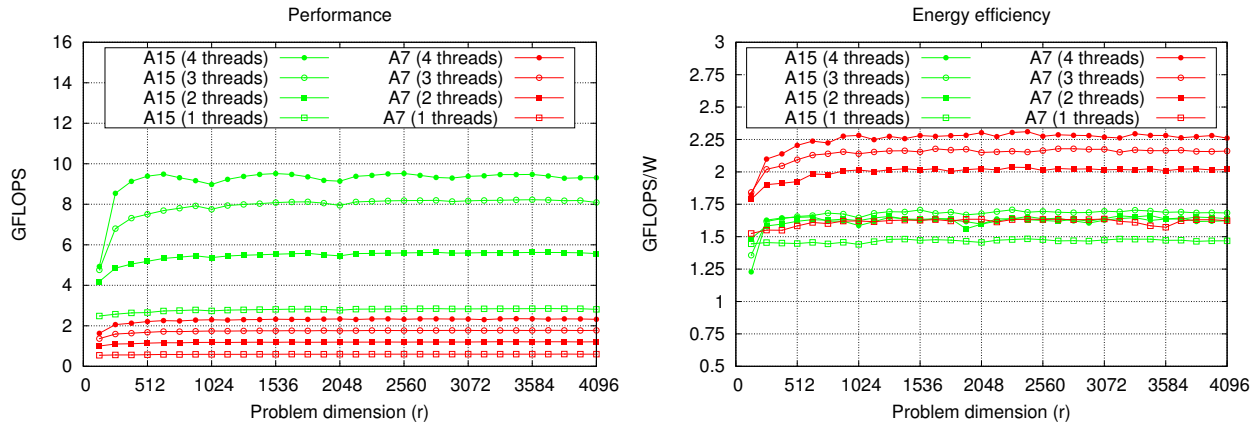

Figure 3.6: Performance (left) and energy efficiency (right) of the BLIS GEMM using exclusively one type of core, for a varying number of threads.

The plots in Figure 3.6 show the performance and energy efficiency of the multi-threaded GEMM implementation in BLIS when using the Cortex-A15 and the Cortex-A7 clusters in isolation. Note that, when calculating the energy efficiency of one type of cluster, the complementary (idle) cluster is deactivated so that it only dissipates a minor power rate (less than 0.1 W). In consequence, only the energy consumed by the active cluster and the memory is accounted for.

Focusing on performance first, the results expose that the Cortex-A15 cores deliver considerable higher performance than their Cortex-A7 counterparts. Specifically, the former type of cores renders an increase of 2.8 GFLOPS per added core when up to three cores are used, but the utilization of the fourth core yields a smaller increase, of 1.4 GFLOPS only. In conjunction, the four cores of the Cortex-A15 cluster attain a peak performance of 9.6 GFLOPS. For the Cortex-A7 cluster, the peak performance is close to 2.4 GFLOPS, also attained with four cores.

Regarding energy efficiency, the Cortex-A7 offers the best results in terms of GFLOPS/W and the benefits of increasing the number of threads are more significant when compared with those obtained with the Cortex-A15 cores. Concretely, the energy efficiency attained with the complete Cortex-A7 cluster is 70% higher than that observed with a single core of the same type. In contrast, the best energy efficiency for the Cortex-A15 is only 14% higher than that measured with a single Cortex-A15 core. This behavior can be explained using a detailed analysis of the power consumption of each type of core. Figure 3.7 reports power consumption for Cortex-A15 and Cortex-A7 cores when their number is increased. These results show that the increase in power consumption for the Cortex-A15 cores is higher (3x) than for the Cortex-A7 (2×). This trend along with the non-linear increase in performance for the big cores, which attain 3× higher performance when using the whole cluster (against the 4× obtained for the Cortex-A7 cores), makes their energy-efficiency

lower. Moreover, due to the same reason the most energy-efficient solution for big cores is obtained with three cores instead of the complete cluster.

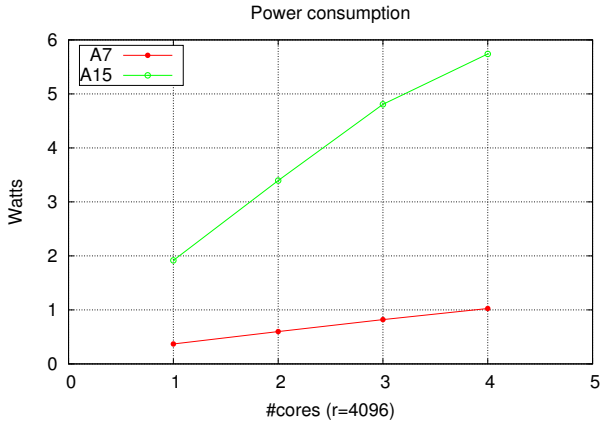

Figure 3.7: Power consumption of the BLIS GEMM for a matrix size of 4,096 using exclusively one type of core, for a varying number of threads.

3.4.3 Asymmetric BLIS GEMM on big.LITTLE

This section proposes two asymmetry-aware strategies (static and dynamic) for workload scheduling of the BLIS GEMM micro-kernels as well as a refinement to exploit the SoC cache hierarchy. Moreover, the impact of these techniques on performance and energy efficiency is evaluated. The optimized implementations can be described, at a high level, as follows:

- *Static-asymmetric scheduling* (SAS). This option statically partitions and assigns loop iterations to different thread types taking into account the performance differences between fast and slow cores.
- *Cache-aware static-asymmetric scheduling* (CA-SAS). This strategy enhances SAS by adapting the loop strides to the distinct cache configurations of the two computing clusters.
- *Cache-aware dynamic-asymmetric scheduling* (CA-DAS). This option improves the previous ones by replacing the static partitioning of the iteration space with a dynamic workload distribution across clusters. For this reason, no previous knowledge about the computational differences between the clusters is required.

In our modifications of the BLIS framework, *control trees* (presented in Section 3.1) have been key in order to encode the differences between the original framework and our versions adapted for AMPs without affecting the rest of the BLAS implementation. In particular, we next focus on the necessary modifications and requirements to implement an asymmetric scheduling of the loop iteration space to big and LITTLE cores, and the modification of the loop strides in order to develop a cache-aware configuration for BLIS GEMM.

3.4.3.1 Static-asymmetric scheduling (SAS)

Taking into account the experiment presented in Section 3.3, the original multi-threaded implementation of BLIS GEMM was revamped to distinguish between the distinct computational power

of the two types of cores included in the ARM big.LITTLE architecture. In particular, the SAS version of BLIS GEMM integrates the following two new features, which modify the behavior of the default asymmetry-oblivious multi-threaded implementation at execution time: *i)* a mechanism to create "fast" and "slow" threads, which are bound upon initialization of the library to the big and LITTLE cores, respectively; and *ii)* a mechanism to decide which loop among those that are parallelized needs to be partitioned and assigned to fast/slow cores asymmetrically. The number of iteration chunks assigned to threads will thus no longer be the same. Instead, these numbers will be set according to the capabilities of each type of core.

The reimplementation also comprises, as a configuration knob, an interface to specify the *ratio* of performance between big and LITTLE cores, which should be known before run time. For the specific loop that is selected as the candidate to partition the computational workload between the two clusters, this configuration parameter controls the number of iteration chunks that are assigned to each cluster. The amount of threads/cores of each type, performance ratio and specific loop to be asymmetrically partitioned can thus be modified via ad-hoc environment variables; and they can all be fixed at execution time in order to tune the behavior of the library to other big.LITTLE setups (for example, to change the core frequency that affects the performance ratio between core types).

This new functionality is fully configurable and has been embedded into the internal control-tree structures that govern the parallelization of each loop in the general BLIS GEMM algorithm.

Evaluation of SAS

Given the memory organization of the Exynos 5422 SoC, and the guidelines for the parallelization of BLIS GEMM from Section 3.3, a two-level parallelization is defined as the combination of a coarse-grain parallelization that distributes the workload between the two clusters (at Loop 1 or Loop 3) and a fine-grain parallelization that partitions the workload among the cores in the same cluster (at Loop 4 or Loop 5).

To illustrate this, Figure 3.8 depicts the distribution of the iteration space across big and LITTLE cores (or threads bound to them) for an scenario in which the iteration space of Loop 1 is asymmetrically distributed between two teams (one composed of the big cores and the other by the LITTLE ones), using a ratio of 3, so that the fast threads are assigned three times more computations than the slow threads. Internally, Loop 4 is parallelized to distribute the work statically among the cores of the same cluster, so that each thread is in charge of executing $n_c/4$ iterations. Note that two iteration blocks are processed in parallel in both clusters. As in Figure 3.3, all threads perform all iterations in Loop 3 and 5.

The combination of the coarse-grain and fine-grain parallelization strategies for SAS yields four direct parallelization schemes. Additionally, two more configurations are possible, combining the coarse-grain parallelization of either Loop 1 or Loop 3 with the fine-grain parallelization of both Loops 4 and 5. Because the qualitative conclusions that can be extracted from these parallelization strategies are very similar, we only report the results when the iteration space is distributed between the clusters in Loop 1 and the macro-kernel is partitioned among homogeneous cores in Loop 4. In addition, the (distribution) ratios for the coarse-grain parallelization range from 1 to 7.

Figure 3.9 displays the results for this experiment. The evaluation shows that, when the appropriate workload distribution is applied, the asymmetric-aware SAS delivers a performance rate that is close to that of the ideal case. The "Ideal" line in the performance graph corresponds to the aggregated performance of the configurations that use four cores of each of the two types in isolation (i.e., the performance of the four Cortex-A15 cores plus the performance of the four Cortex-A7 cores). In particular, the left-hand side graph reveals that the worst performance is achieved when

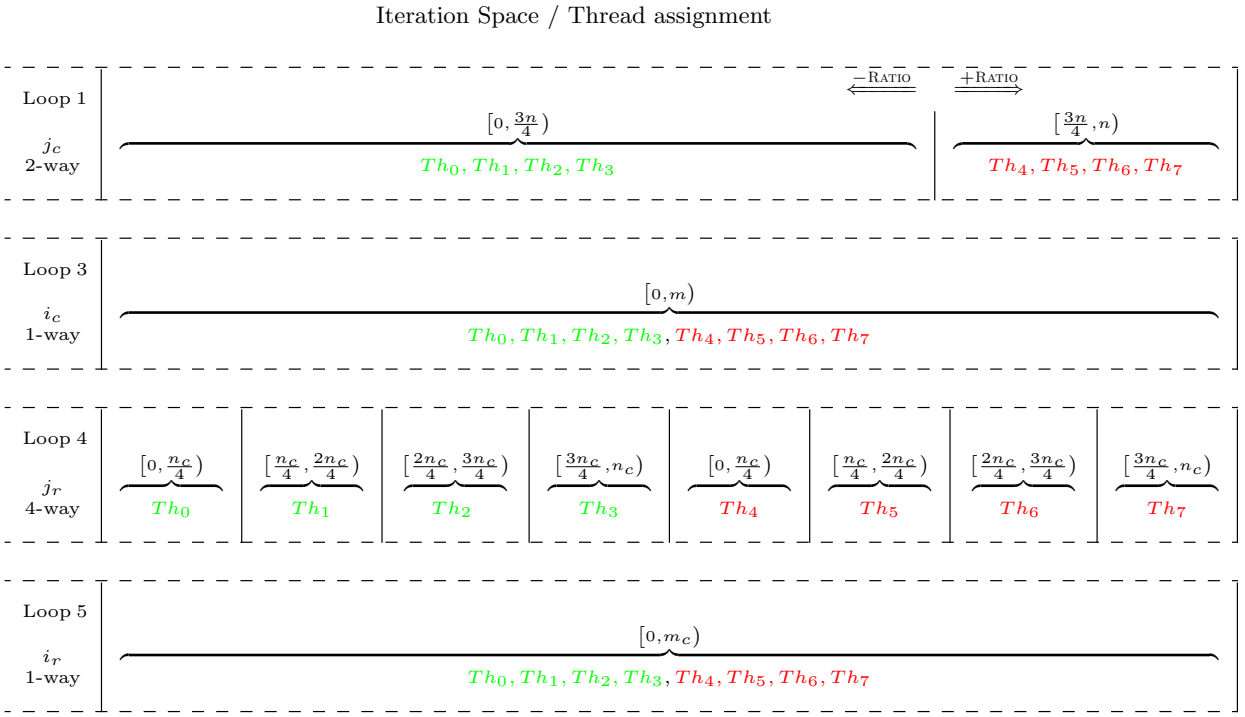

Figure 3.8: Partitioning of the iteration space and assignment to threads/cores for a multi-threaded BLIS implementation with 8-way parallelism that asymmetrically combines 2-way parallelism from Loop 1 (using a ratio between fast and slow cores of 3) and 4-way parallelism from Loop 4.

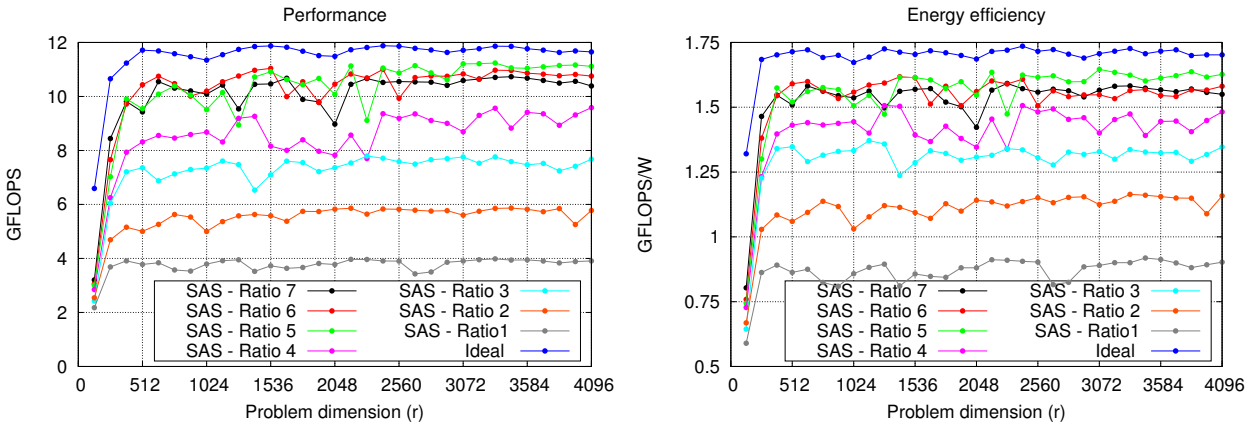

Figure 3.9: Performance (left) and energy-efficiency (right) of the SAS version of BLIS GEMM with a coarse-grain parallelization of Loop 1 and a fine-grain parallelization of Loop 4 using 4 threads per cluster.

the ratio is 1 (i.e., an homogeneous inter-cluster parallelization). Also, the performance grows until a ratio of 5–6 is used; and above 6, the performance in general declines with a lower limit existing at the line delivered by the Cortex-A15 cluster in isolation (not included in the figure for clarity).

These results indicate that ratios below 5 map too much work to the Cortex-A7 cluster, and ratios above 6 assign an excessive part of the work to the Cortex-A15 cluster.

For the largest tested problems, the increment of performance for SAS compared with the configuration that employs four Cortex-A15 cores only is close to 20%. However, SAS offers lower performance for the small problems, as the chunks assigned to the big and LITTLE cores are, in those cases, too small to exploit the asymmetric architecture.

In terms of energy efficiency, when the appropriate workload distribution is in place, SAS delivers similar GFLOPS/W as the setup that exclusively employs the Cortex-A15 cluster (see Figure 3.23). Moreover, the energy efficiency in this case attains 90% of the ideal estimation. On the other hand, when the workload is unbalanced, the energy performance is greatly affected, as the fast threads remain idle but active, polling and consuming energy, until the slow threads complete their work.

3.4.3.2 Cache-aware static-asymmetric scheduling (CA-SAS)

The original implementation of BLIS contains a single control-tree per operation, which implies that the GEMM routine can only employ the optimal cache configuration parameters for either the Cortex-A15 or the Cortex-A7. Our solution to this problem duplicates the control structure to set different configuration values for m_c and k_c, depending on the type of core. Specifically, two different control-trees are created upon initialization, for "fast" and "slow" threads, each setting the optimal loop strides/cache parameters for a different core architecture. In addition, this mechanism opens the door to the use of specific highly-tuned micro-kernels adapted to each micro-architecture in the AMP (and, therefore, optimal values for m_r and n_r), depending on the type of core that executes it. We note that, as argued earlier in Section 3.2, the performance of GEMM is quite independent of n_c, since there is no L3 cache in the Exynos 5422 SoC. Furthermore, we leverage the same micro-kernel for both the Cortex-A7 and Cortex-A15 clusters since, in this particular SoC, it is optimal for both.

An important caveat of this approach is that there may be some dependencies between the optimal configurations used for the clusters. Concretely, if the micro-kernels are distributed among the Cortex-A15 and Cortex-A7 clusters by parallelizing Loop 1, independent buffers are used for A_c and B_c, and no dependencies arise. However, if they are partitioned between the clusters by parallelizing Loop 3, then the buffer for B_c is shared, and it is necessary to employ a common value of k_c for the Cortex-A15 and the Cortex-A7. In this scenario the parameter is set to $k_c = 952$ in both control-trees, and a new (sub)optimal value for m_c has to be obtained for the Cortex-A7 threads. In order to do that, we carried out a similar search as that exposed in Section 3.4.1, finding the new optimal value at $m_c = 32$ for the Cortex-A7 (taking into account that the k_c parameter depends on the Cortex-A15). With these new optimal parameters, the performance peak attained with the Cortex-A7 cluster is slightly worse than that observed with the actual Cortex-A7-specific optimal parameters. However, it is still higher than that obtained with the cache parameters for the Cortex-A15 as, for those much larger values, the memory buffer A_c does not fit into the small L2 cache of the Cortex-A7.

Comparison of SAS and CA-SAS

The combination of the coarse-grain and fine-grain parallelization strategies described in Section 3.4.3.1 yields the same parallelization options for CA-SAS. For the same reasons, we only report next the results corresponding to an scenario where the iteration space is distributed between the clusters across Loop 1, while the macro-kernel is partitioned within clusters in Loop 4, using (dis-

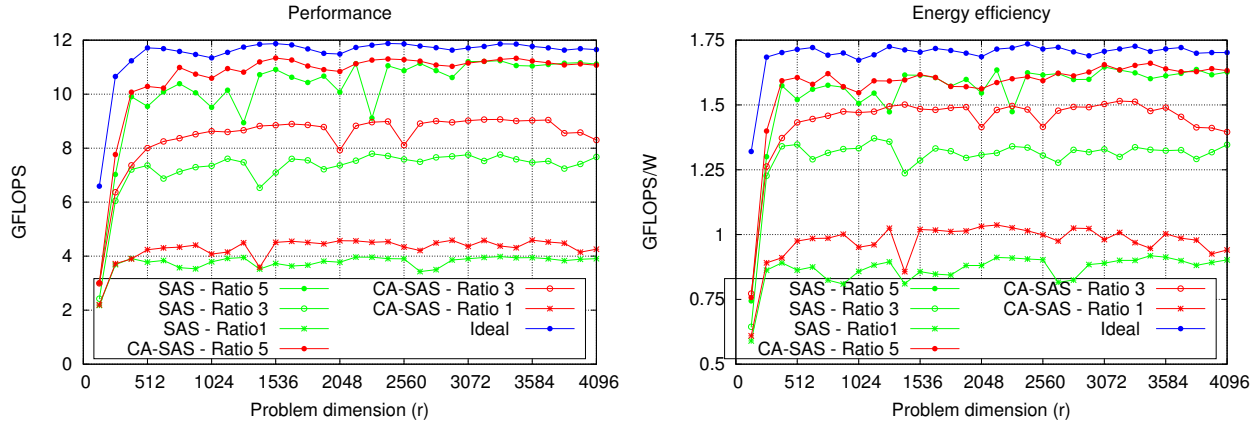

Figure 3.10: Performance (left) and energy-efficiency (right) of the SAS and CA-SAS versions of BLIS GEMM with a coarse-grain parallelization of Loop 1 and a fine-grain parallelization of Loop 4 using 4 threads per cluster.

tribution) ratios for the inter-cluster parallelization of 1, 3 and 5. For each distribution ratio, we include two lines, corresponding to the use of two control-trees (CA-SAS) and only one (SAS).

The plots in Figure 3.10 illustrate that, for both metrics of interest, better results are obtained with the option that integrates two control-trees. The increases of performance and energy efficiency are a direct consequence of the accelerated execution of the workload assigned to the Cortex-A7 cluster, derived from the use of more convenient cache configuration parameters for that architecture. We note that the improvements at this point are only visible when too much work is assigned to the Cortex-A7 cluster (i.e., for distribution ratios below 5). However, as we will expose later, this strategy has a more visible impact when a dynamic workload distribution is adopted.

To conclude the evaluation of the CA-SAS implementation of BLIS, we compare the four direct combinations (parallelization options) of the coarse-grain (Loop 1 or Loop 3) and fine-grain (Loop 4 or Loop 5) options, for a concrete distribution ratio of 5, using two control-trees. Figure 3.11 reports the outcome from this evaluation. The plots show that the fine-grain parallelization of Loop 4 yields performance curves closer to that of the ideal case than the alternatives that parallelize Loop 5. The reason is that n_c (linked to Loop 4) is usually much larger than m_c (linked to Loop 5) and, therefore, it is easier to attain a more balanced workload distribution with this option. Although it is not possible to leverage the actual optimal cache parameters specific to the Cortex-A7 cluster when Loop 3 is parallelized, the plots also reveal that, when the fine-grain parallelization is performed in Loop 4, there is no noticeable difference between distributing the computational workload in either Loop 1 or in Loop 3; however the difference is present when the fine-grain parallelization is set in Loop 5.

3.4.3.3 Cache-aware dynamic-asymmetric scheduling (CA-DAS)

The final step towards attaining a high performance implementation of BLIS GEMM for an AMP SoC integrates a mechanism that dynamically distributes the workload between the two SoC clusters. The main advantage of this option is that a predefined distribution ratio becomes unnecessary, although the target loop this feature is applied to still needs to be chosen with care.

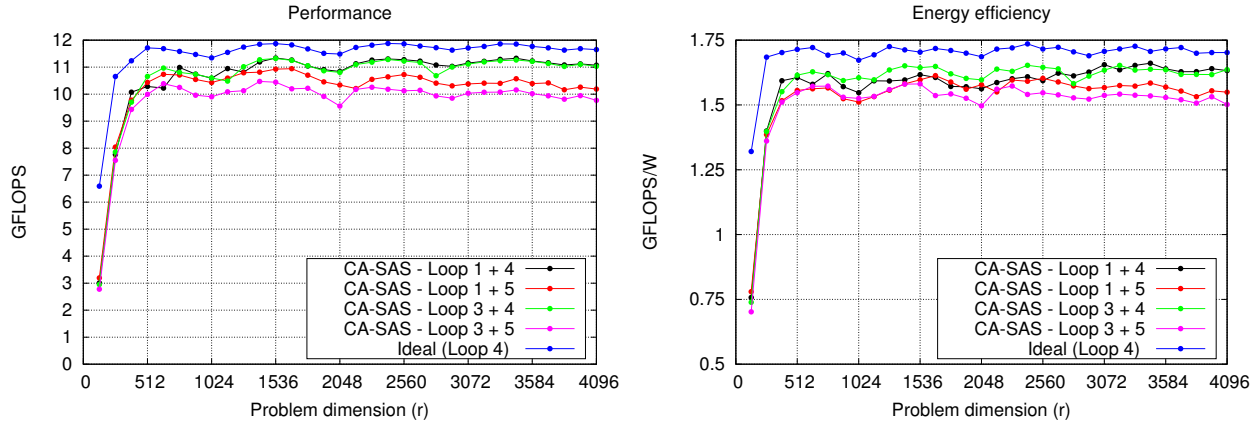

Figure 3.11: Performance (left) and energy-efficiency (right) of the CA-SAS version of BLIS GEMM with a coarse-grain parallelization of either Loop 1 or Loop 3; combined with a fine-grain parallelization of either Loop 4 or Loop 5, using a ratio 5 in both cases and 4 threads per cluster.

The candidates to apply a dynamic distribution are, obviously, Loop 1 and Loop 3, since these have been previously identified as the best options to distribute the computational workload between the two clusters. However, the cache parameter n_c (linked to the stride of Loop 1) is often in the order of several hundreds up to a few thousands and, therefore, in practice it is too large to dynamically distribute the iteration space. In contrast, the cache configuration parameter m_c (linked to the stride of Loop 3) is usually in the order of a few hundreds, and thus it is a good candidate to dynamically distribute the iterations. Diving into details, $n_c = 4{,}096$ for both types of cores, while $m_c = 32$ and 152 for the Cortex-A7 and Cortex-A15 cores, respectively. In consequence, the coarse-grain dynamic distribution of the workload will be carried out across Loop 3, with two independent control-trees in place bound to "fast" and "slow" threads. Note that, like in the CA-SAS scheduling strategy, the buffer B_c is shared by both clusters and, in consequence, k_c is set to 952 for both types of cores (cache-aware optimization).

The application of a dynamic scheduling strategy removes the static partitioning carried out before Loop 3. This is replaced by a mechanism where, at each iteration of Loop 3, a single thread bound to a "fast" core and a single thread bound to a "slow" core select the current chunk size, which depend on the configuration parameter m_c of each type of core. The selected workload is broadcast to the remaining threads of the same type. The fine-grain parallelization remains unmodified and targets Loop 4, Loop 5 or both. The chunk size selection is performed inside a critical section that controls the execution of Loop 3. The overhead of this synchronization point is fully amortized by the utilization of a more flexible workload distribution, except for small matrix sizes, where a static partition is more convenient.

Evaluation of CA-DAS

This set of experiments presents a more reduced number of options, since Loop 1 was identified as a poor choice to dynamically distribute the computational workload. We report results when the iteration space is dynamically distributed between clusters across Loop 3, and the macro-kernel is partitioned within clusters in either Loop 4 or Loop 5, using two control-trees (one for "fast" and one for "slow" threads, CA-DAS) or a single control-tree for both types of threads (DAS) in order

to check which option delivers greater benefits. Additionally, for comparison purposes, we include the performance lines of the best CA-SAS strategy with a distribution ratio of 5.

The plots in Figure 3.12 reveal that, for both metrics of interest, the best results are attained when the coarse-grain parallelization is dynamically applied to Loop 3 and the fine-grain parallelization is done at Loop 4. However, for small matrices ($r < 512$), the coarse-grain static parallelization at Loop 3 outperforms the results for the dynamic approach due to the overhead introduced by the critical section that controls its execution and to the fact that the matrix dimension is too small to perform a suitable distribution of the workload. Regarding the fine-grain parallelization, if it is set across Loop 5, the results are inferior to those reported for the static approach, since the amount of concurrency that can be extracted for Loop 5 is lower than for Loop 4 (see Figure 3.11 and the corresponding analysis for details). On the other hand, the plots show that the use of two control-trees has a great impact on both metrics. The use of a common control-tree implies that the chunk size assigned to both types of threads is the same. Therefore, due to the difference in performance of the Cortex-A7 and Cortex-A15 clusters, this produces a severe load unbalance for certain problem sizes.

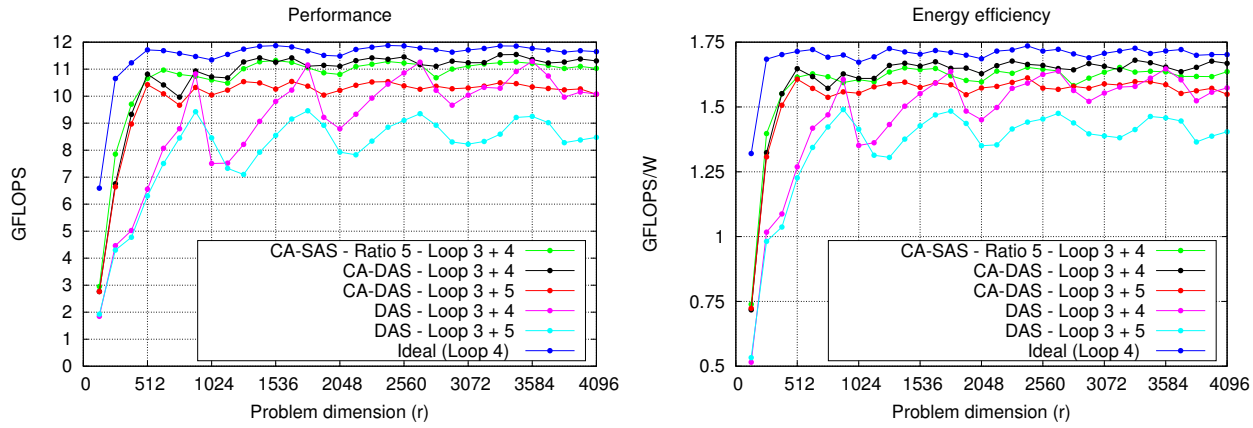

Figure 3.12: Performance (left) and energy-efficiency (right) of the CA-DAS and DAS versions of BLIS GEMM with a coarse-grain parallelization of Loop 3 and a fine-grain parallelization of either Loop 4 or Loop 5, using 4 threads per cluster in both cases.

3.5 Asymmetric BLIS-3 Evaluation

The insights gained from the previous study about the asymmetry-aware GEMM implementation and the special structure of BLIS, which makes all BLAS-3 kernels rely on the GEMM implementation, allowed us to adapt all routines in BLIS-3 in order to make them asymmetry-aware as well. In this section, a detailed analysis of the performance of all BLIS-3 kernels is carried out to verify the gains due to the asymmetry-aware implementation. In order to perform an exhaustive study, different shapes for the operands are considered and alternative parallelizations are explored in each case.

As exposed in the previous section, two different approaches can be used for scheduling: static or dynamic. Consequently, in the remainder of this evaluation the letters "`S`" and "`D`" are used 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.

3.5.1 Square operands in GEMM

Following the solution presented in the previous section, "`D3S4`" and "`D3S5`" will be used 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 measure the GFLOPS rates they attain and compare those against an "`ideal`" execution where the eight cores incur no access conflicts and the workload is perfectly balanced.

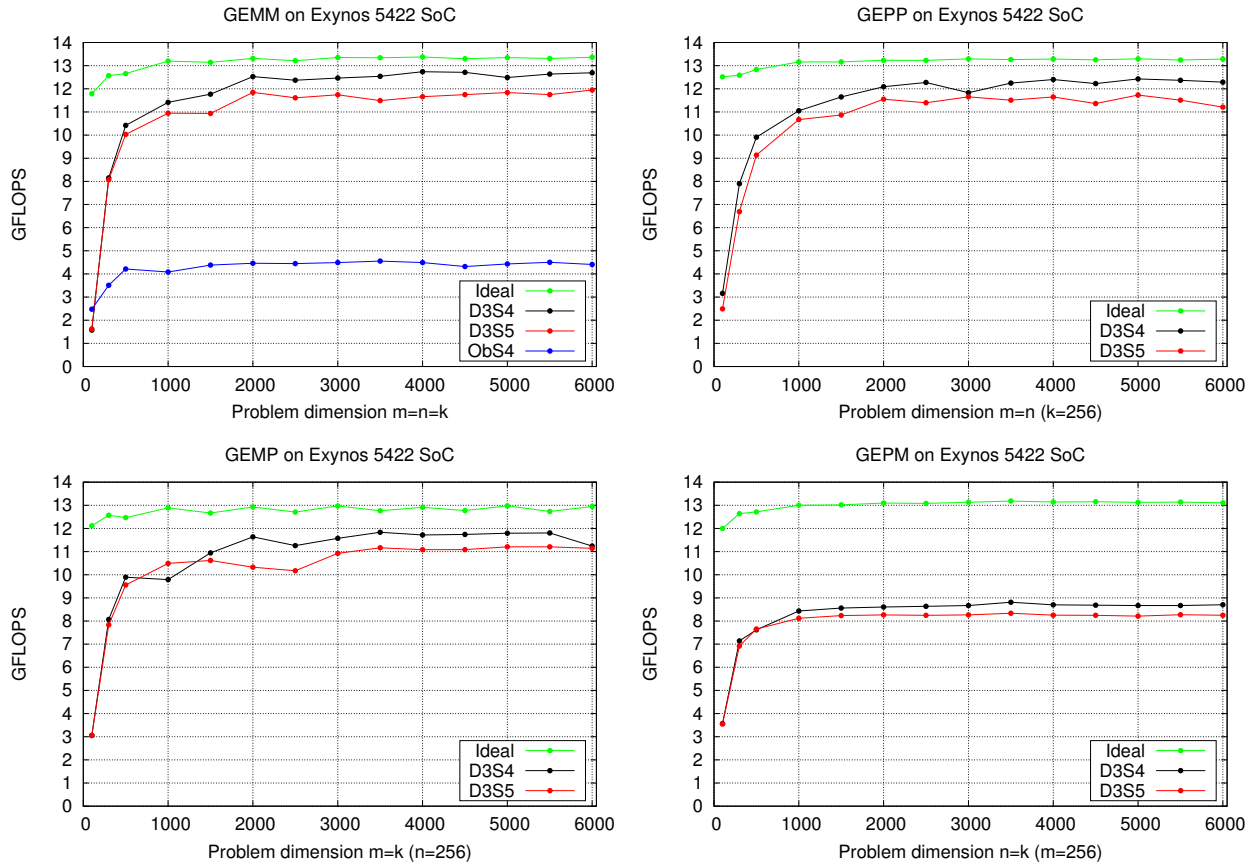

Figure 3.13: Performance of (general) matrix multiplication with square matrices: GEMM; and three rectangular cases with two equal dimensions: GEPP, GEMP, and GEPM.

The top-left plot in Figure 3.13 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, although 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.

3.5.2 GEMM with rectangular operands

The remaining three plots in Figure 3.13 report the performance of the asymmetry-aware parallelization strategies when the matrix-matrix multiplication kernel is invoked, (e.g., from LAPACK,) to compute a product for the following "rectangular" cases (see Table 3.1):

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, the two equal dimensions are varied in the range $\mathcal{R} = \{100,300,500, 1000, 1500, \ldots, 6000\}$ and the remaining one is fixed 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 3.13) 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 behavior is exposed with an execution trace, obtained with the `Extrae` framework [87], in the top part of Figure 3.14.

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 3.15 reports the effect of this compromise, revealing that the pair $m_c = (116, 24)$ for the ARM (Cortex-A15, Cortex-A7) cores presents a better trade-off between balanced workload distribution and cache optimization. For this operation, this concrete 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 3.14 exposes the difference in workload distribution between the executions with $m_c = (152, 32)$ and $m_c = (116, 24)$, respectively.

3.5.3 Other BLIS-3 kernels with rectangular operands

Figure 3.16 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 3.1):

- 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).

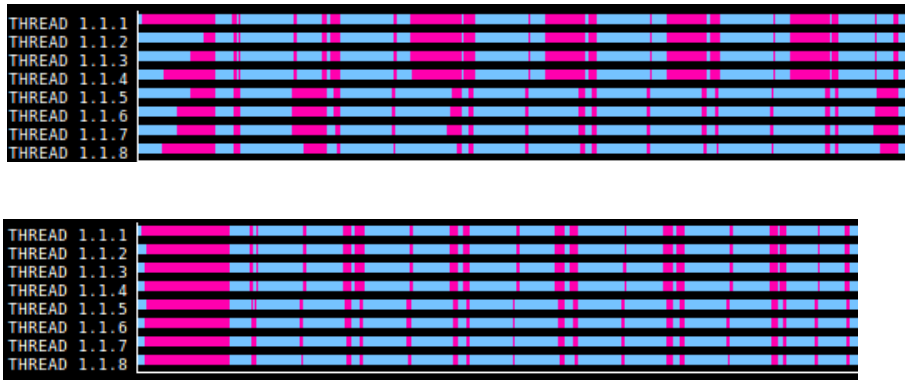

Figure 3.14: Execution traces of GEPM using the parallelization strategy `D3S4` for a problem of dimension $n = k = 2000$ and $m = 256$. The top plot corresponds to the use of cache configuration parameters $m_c = (152, 32)$ for the ARM (Cortex-A15,Cortex-A7) cores, respectively. The bottom plot reduces these values to $m_c = (116, 24)$. The blue periods correspond to actual work while the pink ones represent synchronization (idle time).

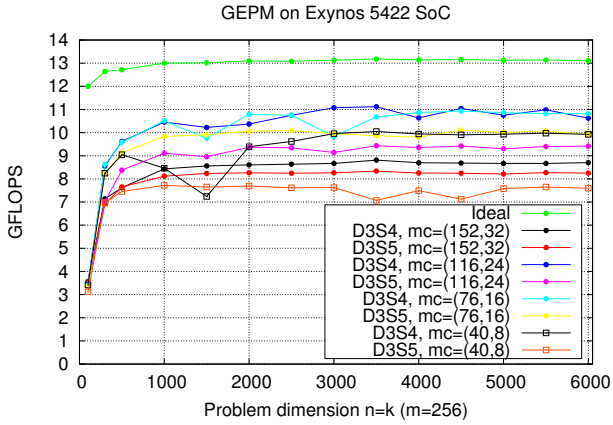

Figure 3.15: Performance of GEPM for different cache configuration parameters m_c.

The row and column dimensions of the symmetric/triangular matrix vary in the range $\mathcal{R} = \{100, 300, 500, 1000, 1500, \ldots, 6000\}$ 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.

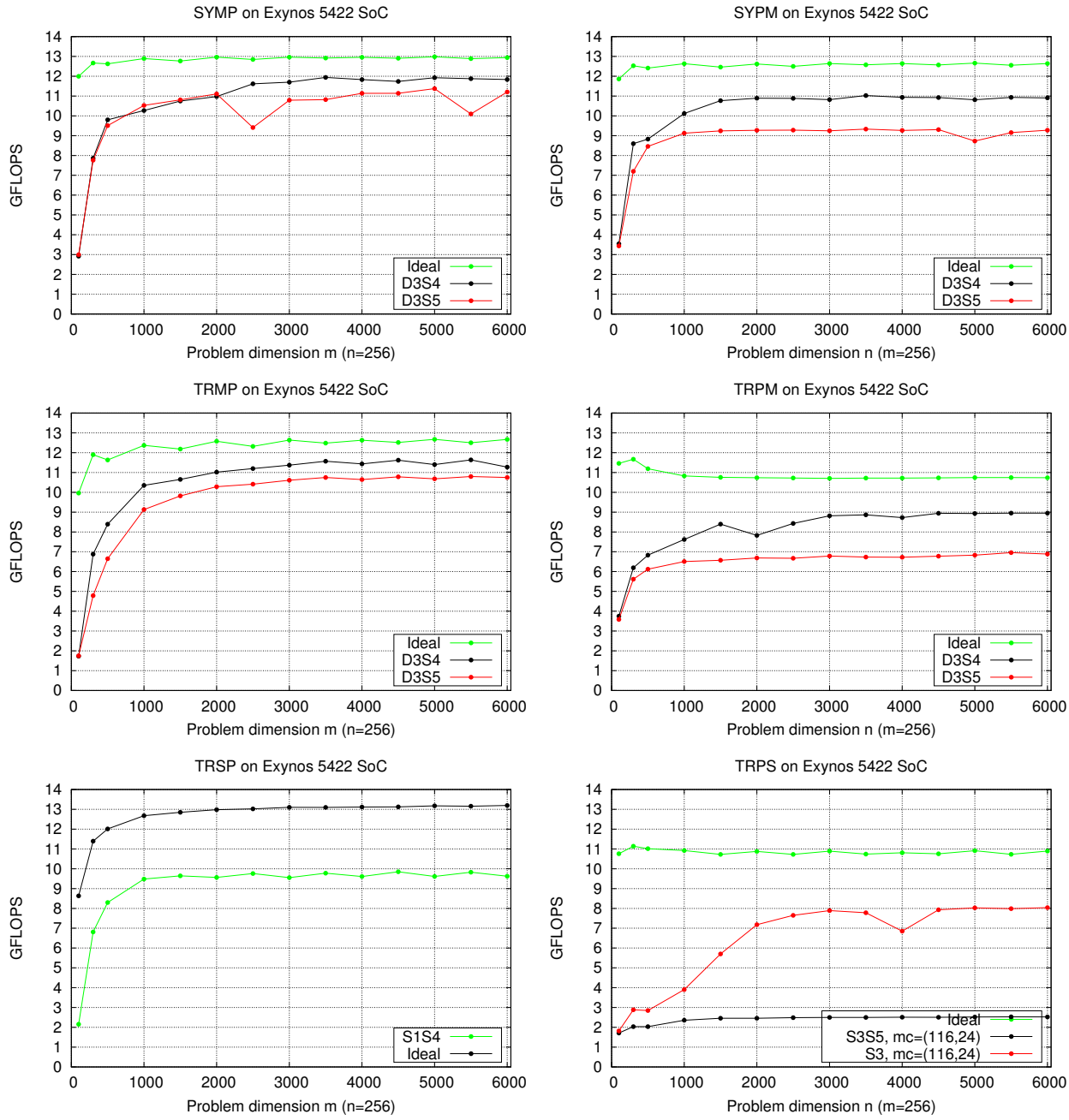

Figure 3.16: Performance of two rectangular cases of SYMM (SYMP for $C := C + AB$ and SYPM for $C := C + BA$), TRMM (TRMP for $B := AB$ and TRPM for $B := BA$), and TRSM (TRSP for $B := A^{-1}B$ and TRSM for $B := BA^{-1}$).

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 [28] 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 3.16 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.

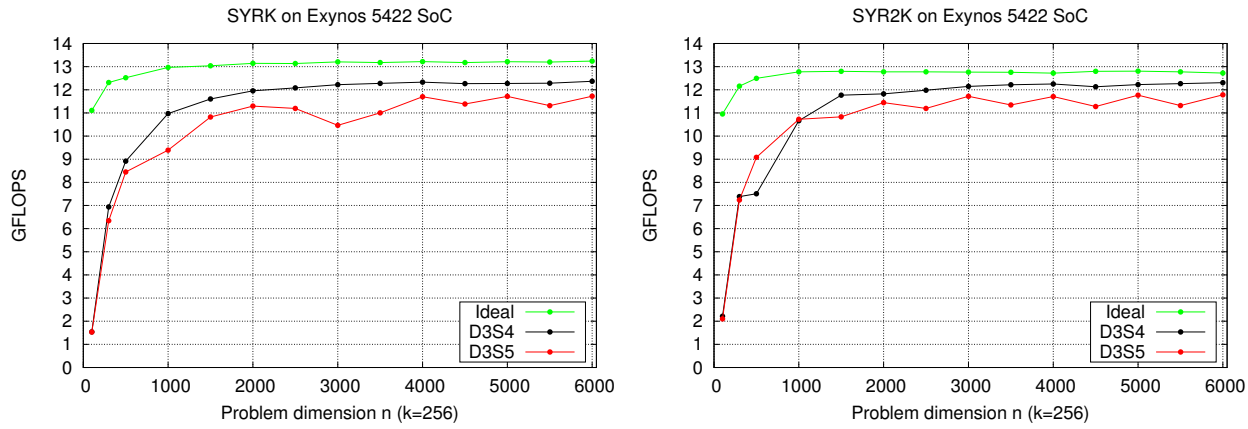

Figure 3.17: Performance of a rectangular case of SYRK and SYR2K.

To conclude the optimization and evaluation of the asymmetry-aware parallelization of BLIS, Figure 3.17 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 $\mathcal{R} = \{100,300,500,1000, 1500, \ldots, 6000\}$ and set $k = 256$ (see Table 3.1). 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, the BLIS-3 kernels optimized for the Exynos 5422 SoC set $m_c = (116,24)$, but rely on the default $m_c = (152,32)$ otherwise.

3.5.4 BLIS-3 in 64-bit AMPs

In order to validate the generality of our approach for AMPs, we applied the same strategies to a 64-bit processor in a Juno ARM development platform. The relevance of this experimentation is motivated not only because we employ a platform with a different register size, but also because of the higher asymmetry that it presents, since the Juno ARM comprises 4 slow cores and only 2 big cores.

In order to implement the asymmetry-aware version of BLIS for the Juno platform, specific *micro-kernels* were implemented for the Cortex-A57 and Cortex-A53 cores. In addition, cache and register configuration parameters were calculated as the result of an independent experiment. The values found for the register parameters were $m_r = 6$ and $n_r = 8$; while for the cache parameters $n_c = 3{,}072$ and $k_c = 240$; m_c was set to 48 for the Cortex-A53 and to 120 for the Cortex-A57.

Figure 3.18 reports the performance of the BLIS kernels on the Juno board, using matrices with square operands ($m = n = k$). In the 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.

3.6 BLIS-2

The routines in the Level 2 of BLAS target matrix-vector operations and, to this end, BLIS provides the kernels described in Table 3.2, including those that work on Hermitian operators (HEMV, HER, HER2). Past development efforts for BLIS have been devoted to support highly tuned micro-kernels and multi-threaded implementations for the Level-3 BLAS kernels [94, 89], while the Level-1 and Level-2 BLAS micro-kernels were left out for future work. Furthermore, the parallelization of these operations was also a pending task. Unfortunately, the experimental analyses clearly exposed that the Level-2 BLAS are essential to improve the efficiency of some LAPACK routines, for example the kernels that perform orthogonal reductions to condensed forms for eigenvalue computations.

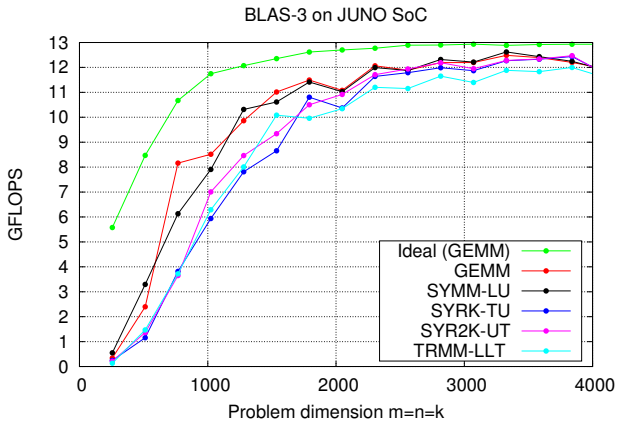

Figure 3.18: Performance of BLAS-3 on Juno SoC.

Kernel	Operation	Operands		
		A	x	y
GEMV	$y := y + Ax$ or $y := y + A^T x$	$m \times n$ $n \times m$	n m	m n
SYMV	$y := y + Ax$	Symmetric $n \times n$	n	n
TRMV	$x := Ax$ or $x := A^T x$	Triangular $n \times n$	n	–
HEMV	$y := y + Ax$	Hermitian $n \times n$	n	n
TRSV	$y := Ax$ or $y := A^T x$	Triangular $n \times n$	n	n
GER	$A := A + xy^T$	$m \times n$	m	n
SYR	$A := A + xx^T$	Symmetric $n \times n$	n	–
SYR2	$A := A + xy^T + yx^T$	Symmetric $n \times n$	n	n
HER	$A := A + xx^H$	Hermitian $n \times n$	n	–
HER2	$A := A + xy^H + yx^H$	Hermitian $n \times n$	n	n

Table 3.2: Kernels of BLIS-2

3.6.1 General matrix-vector multiplication (GEMV)

The implementation of the Level-2 BLIS kernels follows a general structure that is illustrated in this section through the general matrix-vector product GEMV $y := \alpha Mx + \beta y$, with the matrix M of dimension $m \times n$; and y, x vectors with m, n entries, respectively. For simplicity, we assume hereafter that $\alpha = \beta = 1$. This kernel is implemented in BLIS as two loops (see Loops 1 and 2 in Figure 3.19) around two packing routines and a macro-kernel. The GEMV macro-kernel contains an additional loop (Loop 3 in Figure 3.19) around a micro-kernel that casts each update as a fused vector-vector multiply-add [95]. The fusion factor depends on the width of the SIMD instructions and allows the execution in parallel of as many updates as the fusion factor indicates. The packing routines in the GEMV kernel copy the contents of y, x into contiguous buffers y_c, x_c, and unpack y_c into the result vector y (if these vectors were stored with a non-unit stride). No packing is performed on M since there is no reuse in the BLIS implementation of the general matrix-vector product. This approach differs from that introduced in [67], which fuses the two GEMV into a single kernel, in order to reduce the volume of memory transfers. By adhering to the BLIS style though,

the implementation maintains the loops around the macro-kernel which control parallelism and pack data if necessary.

```
Loop 1   for i_c = 0,...,m − 1 in steps of m_c
           y(i_c : i_c + m_c − 1) → y_c                          // Pack into y_c
Loop 2     for j_c = 0,...,n − 1 in steps of n_c
             x(j_c : j_c + n_c − 1) → x_c                        // Pack x into x_c
Loop 3       for j_r = j_c,...,j_c + n_c − 1 in steps of n_r     // Macro-kernel
               y_c += M(i_c : i_c + m_c − 1, j_r : j_r + n_r − 1) // Micro-kernel
                    · x_c(j_r − j_c : j_r − j_c + n_r − 1)
             endfor
           endfor
           y_c → y(i_c : i_c + m_c − 1)                          // Unpack y_c
         endfor
```

Figure 3.19: High performance implementation of the GEMV kernel in BLIS. In the code, x_c,y_c are buffers involved in data copies in case x,y are stored with a non-unit stride. Otherwise, they simply refer to the corresponding entries of the original vectors.

3.6.2 Symmetric matrix-vector multiplication (SYMV)

As pointed out earlier, the structure presented previously is the general approach followed by Level-2 operations. However, BLIS presents a slightly different implementation in the specific case of the symmetric matrix-vector multiplication (or SYMV) kernel $y := \alpha Mx + \beta y$, where M is a symmetric $m \times m$ matrix; y, x are vectors of m components; and α, β are scalars. For simplicity, hereafter we will assume that $\alpha = \beta = 1$, and the strictly upper triangular part of M is not accessed/referenced. This kernel is implemented in BLIS as a loop that traverses the matrix from the top/left corner to bottom/right one. The loop body comprises three packing routines (note that in the general case x is packed in Loop 2), two calls to the GEMV kernel, and one invocation to a specific macro-kernel for SYMV (see Figure 3.20). Note that, when invoking the GEMV kernel from the SYMV kernel, the entry point is Loop 2 from Figure 3.19, since Loop 1 from GEMV is replaced by Loop 1 of SYMV in order to perform the appropriate packs.

```
Loop 1  for i_c = 0,...,m − 1 in steps of m_c
          x(i_c : i_c + m_c − 1) → x_c          // Pack x into x_c
          y(i_c : i_c + m_c − 1) → y_c          // Pack y into y_c
          M(i_c : i_c + m_c − 1,                // Pack only lower
            i_c : i_c + m_c − 1) → M_c          //   triangle of M into M_c
          y_c += M_c · x_c                      // SYMV macro-kernel
          y_c += M_10 · x_0                     // GEMV kernel
          y_c += M_21^T · x_2                   // GEMV kernel
          y_c → y(i_c : i_c + m_c − 1)          // Unpack y
        endfor
```

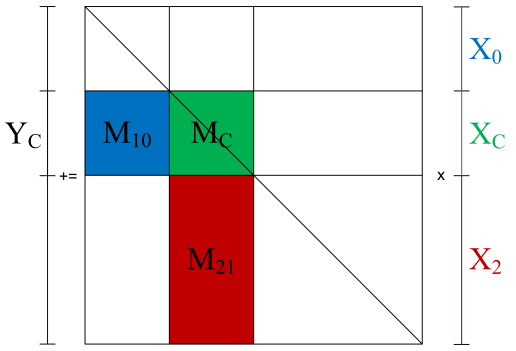

Figure 3.20: Implementation of SYMV in BLIS.

The SYMV macro-kernel is implemented as a separate loop around a micro-kernel (not included in the figure). The macro-kernel performs the operations that involve the $m_c \times m_c$ blocks on the diagonal of M, and casts the corresponding update of y_c in terms of a *fused vector-vector multiply-add* [95]. All input/output operands of the SYMV macro-kernel (M_c, x_c, y_c) are packed since there

is a mild reuse in the symmetric case. Furthermore, the symmetric structure of M is taken into account to access and pack only the lower triangular part of this matrix into M_c.

3.7 BLIS-2 on big.LITTLE

As pointed out in Section 3.6, optimization efforts have been focused on BLIS-3 kernels, implementing hand-tuned micro-kernels targeting different architectures. In addition, that is the only BLIS level that allows the user to parallelize its operations. Since none of these features are available in BLIS-2 for the Cortex-A cores, the insights gained after the adaption of BLIS-3 to these architectures will guide the development of an optimized version of BLIS-2. Therefore, we should follow the next steps:

- Develop hand-tuned micro-kernels for ARMv7 architectures.
- Implement parallel BLIS-2 kernels that exploit the underlying architecture asymmetry.
- Find cache optimization parameters for multi-threaded BLIS-2.

In this Section, the development of specific micro-kernels for ARMv7 architectures is presented along with the parallelization of the kernels for the general matrix-vector product and the symmetric matrix-vector product. Moreover, the selection of the appropriate cache optimization parameters is carried out.

3.7.1 Level-2 BLAS micro-kernels for ARMv7 architectures

To address the lack of BLIS-2 micro-kernels, we have developed hand-tuned micro-kernels for the ARM Cortex-A7 and Cortex-A15 cores that exploit the NEON units in these architectures, employing software prefetching and SIMD instructions. More precisely, two micro-kernels are needed for GEMV implementation depending on the transposition or non-transposition of the input matrix, and one micro-kernel is needed for SYMV.

Figure 3.21 reports the performance of SYMV (left) and GEMV (right) for square matrices with and without hand-coded micro-kernels for the non-transposed case. This plot shows remarkable accelerations for the versions that integrate tailored (i.e., architecture-aware) micro-kernels. Concretely, both kernels multiply the performance of the original codes by a factor close to 3× in the Cortex-A7 architecture, and double it in the Cortex-A15 core. In consequence, all experiments in the following employ our hand-tuned micro-kernels for SYMV and GEMV.

3.7.2 Asymmetry-aware SYMV and GEMV for ARM big.LITTLE AMPs

The current instance of BLIS offers multi-threaded implementations of the Level-3 BLAS, but *sequential versions only* of the Level-1 and Level-2 BLAS, including SYMV and GEMV. A straightforward solution to parallelize both kernels, on the AMP targeted in this work, is to extract the concurrency from Loop 1 via, e.g., an OpenMP construct that dynamically distributes its iteration space among the threads/cores; see Figure 3.20.

One important theoretical advantage of leveraging a dynamic schedule is that the workload is automatically adjusted to the performance capabilities of the two different types of cores in the ARM big.LITTLE AMP. Furthermore, by using distinct cache-aware values of m_c for the Cortex-A15 and the Cortex-A7, the kernels can take advantage of the cache memory hierarchies specific to each type of core. For clarity, Figure 3.22 shows a simple example of this strategy applied to

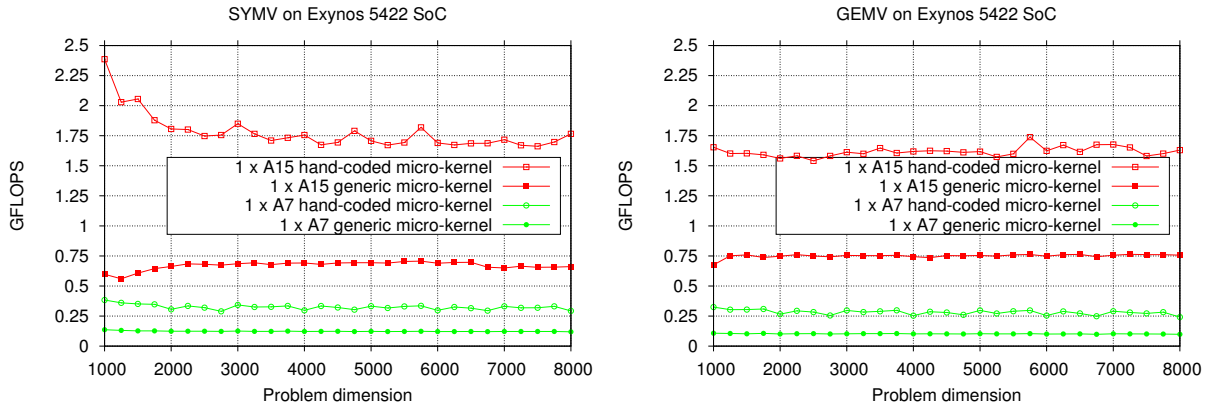

Figure 3.21: Impact of the use of architecture-aware micro-kernels on the performance of SYMV (left) and GEMV (right) for non-transposed matrices in each type of ARMv7 core embedded in the Exynos 5422 SoC.

SYMV (note that a similar approach is followed for GEMV), using two threads, one bound to a big core (Cortex-A15) and one to a LITTLE core (Cortex-A7). These threads are identified in the figure by means of the superscripts "b" and "L", respectively, and compute different parts of the output vector y. The workload distribution is actually implemented via an OpenMP critical section, incurring a synchronization overhead that can be easily compensated by the improvements in the workload balance. Additionally, the parallelization of Loop 1 does not require intermediate buffers for the output vector y, as would be the case if the parallelization targeted any of the two innermost loops of SYMV/GEMV (see Loops 2 and 3 in Figure 3.19).

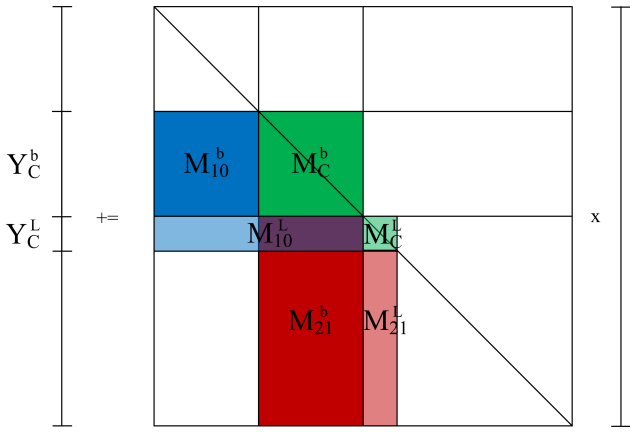

Figure 3.22: Dynamic workload distribution of Loop 1 in SYMV between one big core and one LITTLE core.

A key issue that explains the poor scalability of SYMV is the low ratio between the number of flops ($2m^2$) and memory accesses ($(m^2)/2 + 2m$ at best) which turns SYMV into a memory-bound kernel that, on current architectures, proceeds at the speed dictated by the bandwidth of the memory layer where M is stored. (The same observation with slightly different memory count applies to GEMV.) As a consequence, the performance that can be achieved by SYMV in particular,

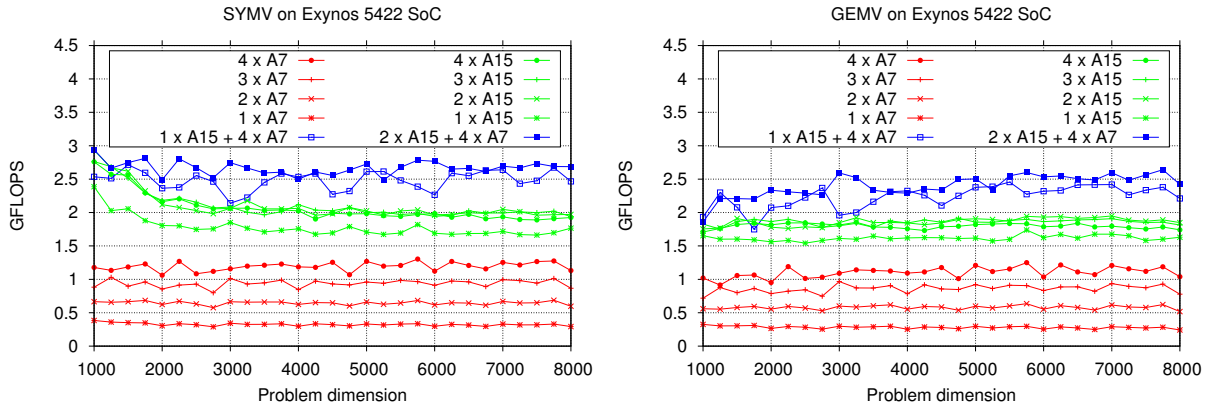

Figure 3.23: Parallel performance of SYMV (left) and GEMV (right) on the ARM big.LITTLE AMP embedded in the Exynos E5422 SoC.

and by any other Level-2 BLAS in general, greatly depends on the memory bandwidth of the target platform. Figure 3.23 offers graphical analyses of the scalability of SYMV and GEMV on the Exynos 5422 SoC. An important observation from this experiment is that SYMV and GEMV hardly scale when increasing the number of big cores beyond 1, but they do with the LITTLE cores. The reason is that a single big core almost saturates the memory bandwidth of the Cortex-A15 cluster so that minor performance increments can be expected by adding more cores of this type. On the other hand, the memory bandwidth available for the LITTLE cluster is able to feed all four LITTLE cores. The plot in the figure also reports the GFLOPS rates for asymmetry-aware configurations of SYMV and GEMV that employ either 1 or 2 Cortex-A15 cores plus the full Cortex-A7 cluster (four cores), showing little benefit from adding the second Cortex-A15 core.

3.7.3 Cache optimization for the big and LITTLE cores

Once specific micro-kernels are implemented for the architecture and the multi-thread is available for the operations, the next step to attain high performance is, given a target precision (single in this case), determine the configuration parameters n_c, k_c, m_c, n_r, and m_r for a single ARM core of each type, Cortex-A15 and Cortex-A7, that fit the cache organization. However, given that the BLIS-2 kernels are memory-bound operations and, consequently, the reuse of data is much lower than it is in BLIS-3, the relevance of these values is now much lower, leaving us the option to work with the optimal cache configuration parameters found for BLIS-3.

In order to optimize cache and register configuration parameters for BLIS-3 using single precision, we have to take into account the same considerations as those done for double precision. As pointed out previously, when optimizing cache configuration parameters for BLIS-3, n_c plays a minor role in this architecture, so it is set to 4,096, a value provided by experts for single precision on ARMv7 architectures. Moreover, the micro-kernels for these core architectures and precision are tuned with $m_r = 4$ and $n_r = 4$. Therefore, the optimization of GEMV and SYMV depends on determining the optimal values of m_c and k_c. However, given that only m_c affects the parallelization of BLIS-2 kernels in the proposed approach, since its value will be used to distribute the workload between the different types of cores, the original value for $k_c = 368$ is maintained. Table 5.1 provides the summary of the cache optimization parameter values that were calculated as the outcome of an independent experiment using a single Cortex-A7 and a single Cortex-A15 core.

	m_r	n_r	m_c	k_c	n_c
ARM Cortex-A15	4	4	400	368	4,096
ARM Cortex-A7	4	4	88	368	4,096

Table 3.3: Parameters for optimal performance of the Level-3 kernels in BLIS on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC using real single-precision IEEE arithmetic.

3.8 Evaluation of Asymmetric BLIS-2

In this section, a detailed analysis of the performance of GEMV and SYMV is carried out to verify the gains due to the asymmetry-aware implementation. In order to extend the insights gained from the previous tests, different shapes for the operands are considered and performance results are compared against an "Ideal", calculated as the accumulation of the performance rate for one Cortex-A15 core and four Cortex-A7 cores. We also include, as a reference, the results for one Cortex-A15 core and for four Cortex-A7 cores.

In all the tests the frequency is set to 1.4GHz for the Cortex-A7 cores and to 1.5GHz for the Cortex-A15 cores. In addition, cache configuration parameters are set to their optimal values (Table 5.1).

3.8.1 SYMV and GEMV with square operands

As concluded in the previous study, the best option in order to increase performance in BLIS-2 kernels employs the whole Cortex-A7 cluster in combination with one Cortex-A15 core. Thus this is the configuration used in the following. Figure 3.24 reports performance results for SYMV and GEMV when using square matrices ($m = n = k$). The results show that SYMV attains slightly higher performance than GEMV, reaching up to 90% of the "Ideal" performance (against 88% in the case of GEMV). Moreover, in both cases, the asymmetry-aware version of the operations outperforms the results obtained when using a single Cortex-A15 and when employing the Cortex-A7 cluster in isolation.

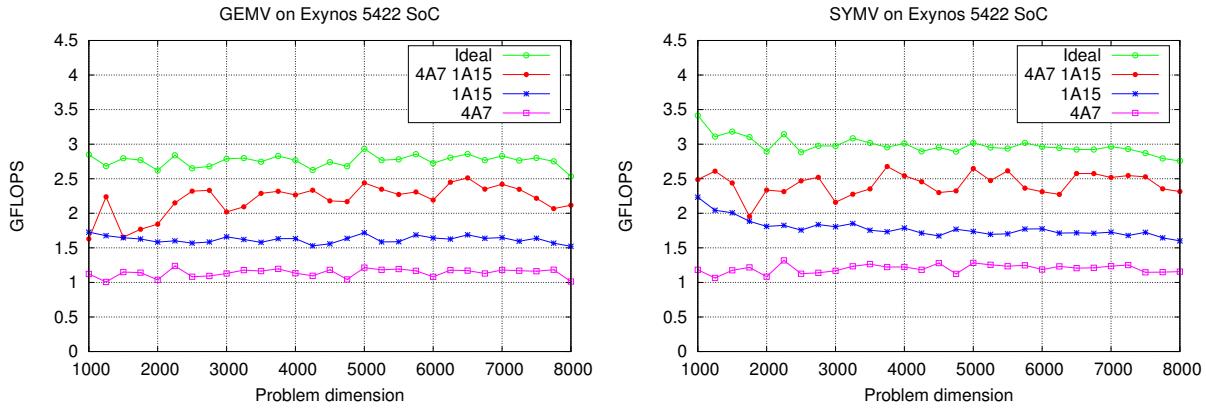

Figure 3.24: Parallel performance of GEMV (left) and SYMV (right) with square operands on the ARM big.LITTLE AMP embedded in the Exynos E5422 SoC.

3.8.2 GEMV with rectangular operands

Figure 3.25 presents performance results in terms of GFLOPS, when GEMV is applied to a "rectangular" matrix. On the left-hand side plot we show the results when the number of rows of the input matrix, dimension m, is fixed; while on the right-hand side plot the fixed dimension is n. When comparing both plots, we see that the performance when the dimension m is fixed is lower than when the dimension that is fixed corresponds to n. This behavior is due to the parallelization approach, since the workload distribution is done at Loop 1, which means that higher parallelism degree can be applied when the m-dimension is greater. This situation is shown on the right-hand side plot. On the same vein, when fixing the m-dimension to a small value (left plot) the performance decreases because it cannot take advantage of the multi-threaded implementation. Moreover, the same reason explains why, using a small m keeps performance at 67% of the "Ideal" performance (and even below the results obtained with one Cortex-A15, especially for small matrix sizes), while increasing m (and fixing $n = 256$) delivers 75% of the "Ideal" performance rate.

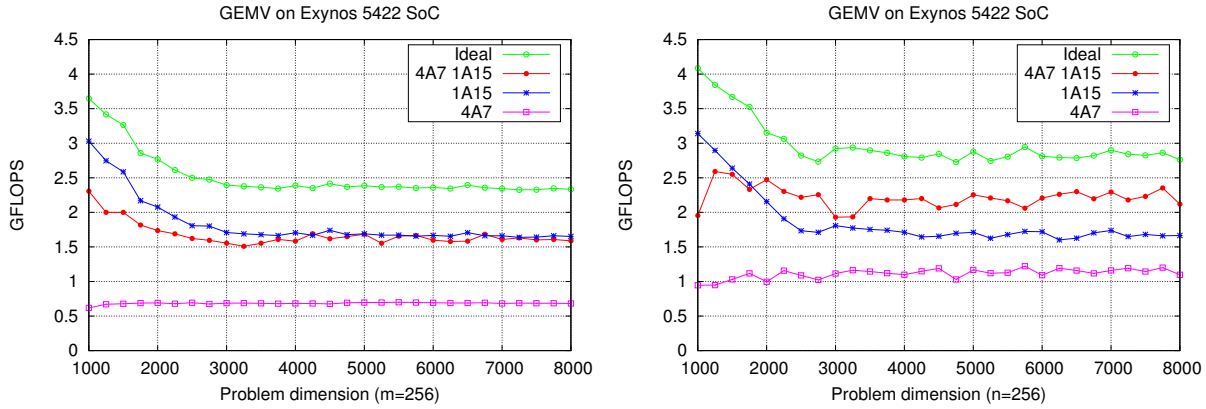

Figure 3.25: Parallel performance of GEMV with rectangular operands on the ARM big.LITTLE AMP embedded in the Exynos E5422 SoC.

3.9 Summary

After an overview of BLIS-3, in this chapter we have presented an optimized asymmetry-aware GEMM. We described the changes applied to the control-tree structure that governs the multi-threaded parallelization of BLIS GEMM in order to accommodate cache-aware configurations of the loop strides for each type of core architecture that match the organization of its cache hierarchy. In addition, two alternative scheduling strategies were introduced, asymmetric–static and dynamic, to produce a 1-D partitioning of (the iteration space for) one of the outer loops of BLIS GEMM between the two clusters that yields a balanced distribution of the micro-kernels. Furthermore, an orthogonal symmetric–static schedule was applied to map the workload of one of the inner loops across the cores of the same cluster. The practical benefits of the cache-aware configurations and asymmetry-aware scheduling strategies were demonstrated on the Exynos 5422, showing that the performance attained by the optimized GEMM on this platform is well beyond that of an architecture-oblivious multi-threaded implementation and close to that of an ideal scenario. Moreover, an analysis of the energy efficiency of the asymmetric architecture when running our optimized GEMM was included,

using the GFLOPS/W (billions of floating-point arithmetic operations, or FLOPS, per second and Watt) metric, which assesses the energy cost of FLOPS.

We leveraged the flexibility of the BLIS framework in order to extend the asymmetry-aware high performance implementation of GEMM to the whole BLAS-3 for AMPs, taking into consideration the operands' dimensions and shape. The key of our development was 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 showed considerable performance acceleration for the BLAS-3 kernels, and more moderate for the triangular system solve.

Finally, the insights gained from the optimization of the BLIS-3 were applied to two key BLIS-2 kernels, namely GEMV and SYMV. We developed architecture-aware micro-kernels for both operations that exploit data locality in the access to the data in the registers. Furthermore, we proposed a strategy to parallelize SYMV and GEMV using only one type of cluster that carries over to the full asymmetric architecture via the introduction of dynamic scheduling. Performance benefits of this asymmetry-aware implementation were demonstrated through an experimental analysis that reported the gains due to the implementation of specific micro-kernels as well as the parallelization of the kernels.

CHAPTER 4

Factorizations

The main objective of this chapter is to examine how well LAPACK performs on an AMP when linked to our asymmetry-aware parallel version of BLIS, since the tests performed with BLIS kernels in isolation provided good results in the previous chapter.

In this Chapter, we first analyze the computational performance and energy efficiency of LAPACK on the Exynos 5422 SoC when linked to our asymmetry-aware implementation of BLIS. 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. The impact of limiting the optimizations to this layer will be exposed via two crucial dense linear algebra operations [56], illustrative of quite different outcomes:

- The Cholesky factorization for the solution of symmetric positive definite (s.p.d.) linear systems (routine POTRF from LAPACK).
- The LU factorization (with partial row pivoting) for the solution of general linear systems (routine GETRF from LAPACK).

The performance of these two LAPACK routines on top of the asymmetry-aware BLIS (for the double precision case) is analyzed in terms of floating-point arithmetic throughput (GFLOPS) and energy efficiency (GFLOPS/W which determines the energy cost per flop). These results are later used in this chapter as a baseline to propose several techniques to improve performance. More specifically, we apply the look-ahead technique in order to enhance the LU factorization and we also propose malleability at thread level as a new approach that, in combination with look-ahead, increases the performance.

4.1 Cholesky factorization

Given a dense s.p.d. matrix $A \in \mathbb{R}^{n \times n}$, its *Cholesky factorization* is given by $A = U^T U$, where the Cholesky factor $U \in \mathbb{R}^{n \times n}$ is upper triangular [56]. Listing 4.1 displays a simplified C code that computes the Cholesky factorization of an `n`×`n` matrix stored starting at address `A` with column leading dimension `Alda`. For simplicity, we assume hereafter that the matrix size is an

integer multiple of the algorithmic block size `b`. The code overwrites the corresponding entries of the original matrix with the Cholesky factor, leveraging the numerical *kernels* (or building blocks) TRSM, SYRK from the Level-3 BLAS and a sort of recursive call to the Cholesky factorization to handle a small diagonal block POTRF.

```
#define A_ref(i,j) A[(j)*Alda+(i)]

for (k=0; k<n; k+=b_o) {

  // Factor current diagonal block
  POTRF( ..., b_i, &A_ref(k,k), Alda, &info );
  if ( k+b<n ) {
    // Triangular solve
    TRSM( ..., &A_ref( k, k ), Alda, &A_ref( k, j ), Alda );
    // Update trailing submatrix
    SYRK( ..., &A_ref( k, i ), Alda, &A_ref( i, i ), Alda );
  }
}
```

Listing 4.1: Blocked routine for the Cholesky factorization.

We next illustrate the impact of leveraging our platform-specific BLIS-3 from LAPACK using the Cholesky factorization.

Figure 4.1 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 strategy for comparison already applied to the BLIS-3, in the performance plot we include the GFLOPS rate for the ideal configuration calculated as the addition of the performance rate for the Cortex-A7 and Cortex-A15 clusters (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). All tests in this section were performed with the processor frequency set to 1.6 GHz for the Cortex-A15 cores and 1.4 GHz to the Cortex-A7 cores.

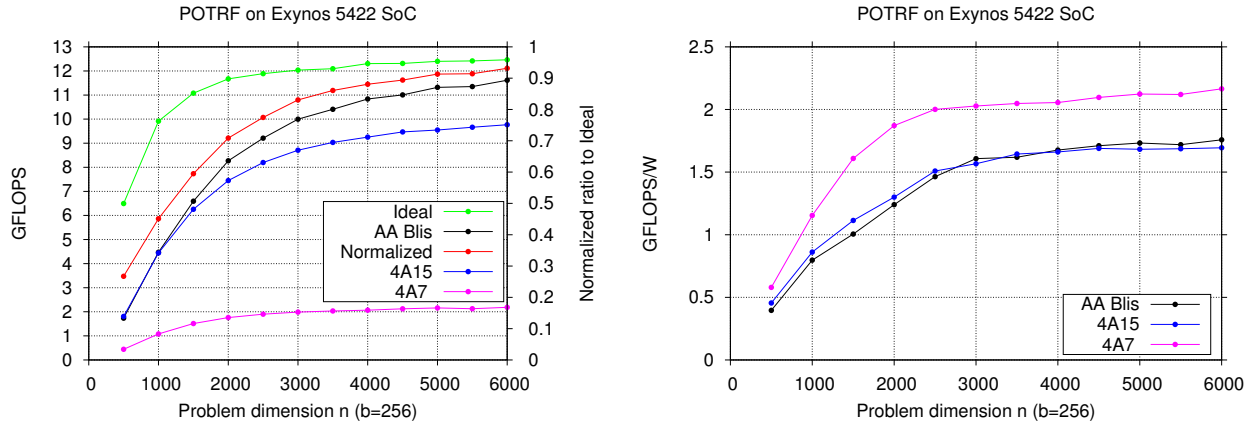

Figure 4.1: Performance and energy efficiency of POTRF for the solution s.p.d. linear systems.

For this particular factorization, as the problem dimension grows, the gap between the ideal peak performance and the sustained GFLOPS rate rapidly shrinks. This is quantified in the column

labeled as Normalized GFLOPS in Table 4.1, which reflects the numerical values represented by the normalized curve (in red) in Figure 4.1. Here, for example, the implementation obtains over 70% and 83% of the ideal peak performance for $n = 2{,}000$ and $n = 3{,}000$, respectively.

n	POTRF	
	Normalized GFLOPS (%)	Normalized flops of SYRK (%)
500	26.73	36.67
1000	45.12	64.97
1500	59.49	75.90
2000	70.85	81.65
2500	77.46	85.18
3000	83.06	87.58
3500	86.05	89.31
4000	88.06	90.61
4500	89.39	91.63
5000	91.29	92.46
5500	91.42	93.15
6000	93.16	93.69

Table 4.1: Performance of matrix factorizations for the solution of s.p.d. and general linear systems (POTRF) normalized with respect to the ideal peak performance (in %); and corresponding theoretical costs of the underlying basic building block SYRK normalized with respect to the total factorization cost (in %).

This appealing behavior 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, where one of them is a symmetric rank-k update (SYRK) that involves a row panel of $k = b$ rows [56]. Furthermore, as n grows, the cost of this update rapidly dominates the total cost of the decomposition, as shown in the column "Normalized flops" in Table 4.1, which represents the percentage of flops performed by SYRK in the Cholesky factorization depending on the size of the input matrix. As a result, the performance of this variant of the Cholesky factorization approaches that of SYRK, reported in Figure 3.17. Indeed, it is quite remarkable that, for $n = 6{,}000$, 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 $n = 6{,}000, k = 256$. As exposed in Chapter 3, our asymmetry-aware implementation of BLIS shows poorer performance for small matrix sizes due to the dynamic workload distribution and, for the same reason, the performance attained decays for the Cholesky factorization for matrix sizes below n=2000.

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 that of the alternative which relies on the Cortex-A15 cores only, since the raise in power dissipation is compensated by the increment in performance.

4.2 LU

Given a matrix $A \in \mathbb{R}^{m \times n}$, its LU factorization produces lower and upper triangular factors, $L \in \mathbb{R}^{m \times n}$ and $U \in \mathbb{R}^{n \times n}$ respectively, such that $PA = LU$, where $P \in \mathbb{R}^{m \times m}$ defines a permutation that is introduced for numerical stability [56]. Listing 4.2 displays a simplified C code that computes the LU factorization of an `m`×`n` matrix stored starting at address `A` with column leading dimension `Alda`. For simplicity, we assume hereafter that the matrix is square $m = n$ and this dimension also an integer multiple of the algorithmic size `b`. The code overwrites the corresponding entries of the original matrix with the LU factors, leveraging the numerical *kernels* (or building blocks) TRSM, GEMM from the Level-3 BLAS, and GETRF for the panel factorization.

```
#define A_ref(i,j) A[(j)*Alda+(i)]

for (k=0; k<n; k+=b_o) {

  // Factor current diagonal block
  GETRF( ..., b_i, &A_ref(k,k), Alda, piv, &info );
  // Interchanges to the panel
  SWAP(..., &A_ref(k,k), Alda, piv);
  if ( k+b<n ) {
    // Interchanges to the trailing submatrix
    SWAP(..., &A_ref(0,j), Alda, piv);
    // Triangular solve
    TRSM( ..., &A_ref( k, k ), Alda, &A_ref( k, j ), Alda );
    if ( k+b<m ) {
      // Update trailing submatrix
      GEMM( ..., &A_ref( j, k ), Alda, &A_ref( k, j ), Alda, &A_ref( j, j ), Alda);
    }
  }
}
```

Listing 4.2: Blocked routine for the LU factorization.

Figure 4.2 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 $m = n$. All the experiments in this section were performed employing IEEE double precision arithmetic and the chip frequency was set to 1.3 GHz for both types of cores.

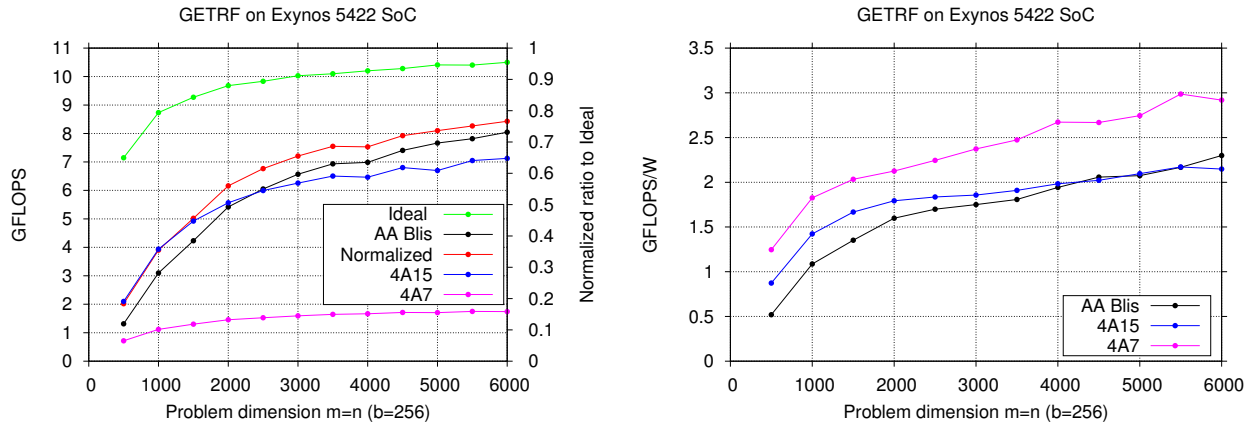

Figure 4.2: Performance and energy efficiency of GETRF for the solution general linear systems.

The actual performance and energy efficiency of the LU factorization follows the same general trends observed for the Cholesky factorization, although there are some differences that are worth discussing. 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 55.98% and 65.51% of the ideal peak performance for $n = 2{,}000$ and $n = 3{,}000$, 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 Tables 4.1 and 4.2.

n	GETRF	
	Normalized GFLOPS (%)	Normalized flops of GEMM (%)
500	18.36	36.67
1000	35.47	64.97
1500	45.64	75.90
2000	55.98	81.65
2500	61.52	85.18
3000	65.51	87.58
3500	68.63	89.31
4000	68.45	90.61
4500	72.01	91.63
5000	73.63	92.46
5500	75.14	93.15
6000	76.61	93.69

Table 4.2: Performance of matrix factorizations for the solution of general linear systems (GETRF) normalized with respect to the ideal peak performance (in %); and corresponding theoretical costs of the underlying basic building block GEMM with panel input operands normalized with respect to the total factorization cost (in %).

Let us discuss this further. Like POTRF, routine GETRF casts most flops in terms of efficient BLAS-3 kernels, in this case the matrix-matrix multiplication GEMM with panel input operands (previously analyzed as GEPP in Chapter 3). Nonetheless, the moderate performance behavior of GETRF lies in the high practical cost (i.e., execution time) of the 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 $n = 3{,}000$. 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 [92]. 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 the development of asymmetry-aware runtimes is still immature.

4.3 Look-ahead in the LU factorization

In order to explore the impact of the *look-ahead* in the LU factorization, a new implementation of the legacy code is needed to accommodate the new approach. To this end, we start from a basic algorithm that is successively modified to accommodate different strategies of *look-ahead* to the code. In addition, given that the implementation of the operation with *look-ahead* is not straightforward and in order to isolate the tests from the asymmetry of the targeted AMP, the initial experiments of the *look-ahead* algorithms were carried out on an Intel Xeon E5-2603 v3 processor (6 cores at a nominal frequency 1.6 GHz). Once the impact of *look-ahead* is tested on a symmetric platform, the AMP will be our next target.

As shown earlier in Chapter 3, the multi-threaded instances of the BLAS for current multi-core processor architectures take advantage of the simple data dependencies featured by the operations to exploit loop/data-parallelism at the block level (hereafter referred to as block-data parallelism or BDP). However, for more complex DLA operations, like those supported by LAPACK [11] and libflame [93], exploiting task-parallelism with dependencies (TP) is especially efficient when performed by a runtime that semi-automatically decomposes the computation into tasks and orchestrates their dependency-aware scheduling [48, 91, 5, 52]. For the BLAS kernels though, exploiting BDP is still the preferred choice, because it allows tighter control on the data movements across the memory hierarchy and avoids the overhead of a runtime that is unnecessary due to the (mostly) nonexistent data dependencies in the BLAS kernels. Exploiting both BDP and TP, in a sort of nested parallelism, can yield more efficient solutions as the number of cores in processor architectures continues to grow, and this combination will be our objective.

In this section we first review the conventional unblocked and blocked algorithms for the LU factorization, and then describe how BDP is exploited from within them. Next, we introduce a re-organized version of the algorithm that integrates look-ahead in order to enhance performance in a nested TP+BDP execution. The implementations presented were linked with BLIS version 0.1.8 or a tailored version of this library especially developed for this work and, unless otherwise stated, block-data parallelism (BDP) is extracted only from Loop 4 of the BLIS kernels. In addition, since BLIS kernels are applied to different shapes of input matrices depending on the implemented algorithm, we always refer to them with the generic name (i.e. GEMM, TRSM, etc.) independently of the real shape of the operands.

4.3.1 Basic algorithms and Block-Data Parallelism (BDP)

There exist a number of algorithmic variants of the LU factorization that can accommodate partial pivoting [56]. Among these, Figure 4.3 (left) shows an unblocked algorithm for the so-called *right-looking* (RL) variant, expressed using the FLAME notation [60]. The RL variant performs computations on the current panel traversing the matrix from left to right. Its main characteristic is that, at each iteration, after factorizing the current panel, columns on the right of that panel are updated immediately. For simplicity, we do not include pivoting in the following description of the algorithms, although all our actual implementations, (and in particular those employed in our experimental evaluation,) integrate standard partial pivoting. The cost of computing the LU factorization of an $m \times n$ matrix, via any of the algorithms presented in this chapter, is $mn^2 - n^3/3$ flops. Hereafter, we will consider square matrices of order n, for which the cost simplifies to $2n^3/3$ flops. For the RL variants, the major part of these operations are concentrated in the initial iterations of the algorithm(s). For example, the first 25% iterations account for almost 58% of the flops; the first half for 87.5%; and the first 75% for more than 98%. Thus, the key to high performance mostly lies in the initial stages of the factorization.

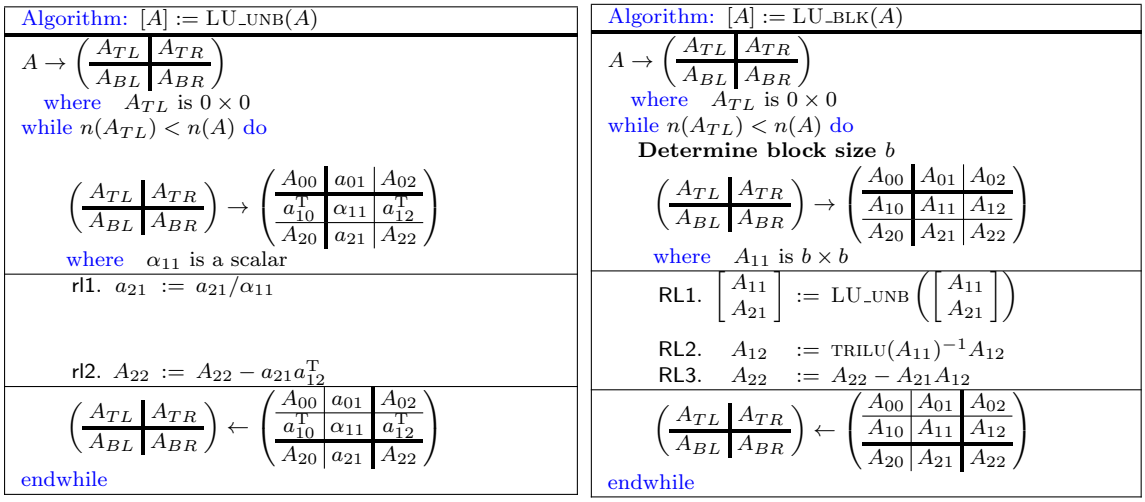

Figure 4.3: Unblocked and blocked RL algorithms for the LU factorization (left and right, respectively). In the notation, $n(\cdot)$ returns the number of columns of its argument, and TRILU$(\cdot)$ returns the strictly lower triangular part of its matrix argument, setting the diagonal entries of the result to ones.

For performance reasons, DLA libraries compute the LU factorization via a blocked algorithm that casts most computations in terms of GEMM, in contrast to the unblocked algorithm, which relies on BLAS-2 operations. Figure 4.3 (right) presents the blocked RL algorithm, which was previously used in Section 4.2. For each iteration, the algorithm processes panels of b columns, where b is the algorithmic block size. The three operations in the loop body factorize the "current" panel $A_p = \left[\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right]$, via the unblocked algorithm (LU_UNB, RL1); and next update the trailing submatrix, consisting of A_{12} and A_{22}, via a triangular system solve (TRSM, RL2) and a matrix multiplication with panel operands (GEMM, RL3), respectively. In practice, the block size b is chosen so that the successive invocations to the GEMM kernel deliver high FLOPS rates. If b is too small, the performance of GEMM will suffer, and so will that of the LU factorization. On the other hand, reducing b is appealing as this choice decreases the number of flops that are performed in terms of the panel factorization, an operation that can be expected to offer significantly lower throughput (FLOPS) than GEMM. (Concretely, provided $n \gg b$, the cost required for all panel factorizations is about $n^2b/2$ flops.) Thus, there is the tension between these two requisites, as it will be analyzed in Section 4.3.4.1.

When the target platform is a multi-core processor, the conventional parallelization of the LU factorization simply relies on multi-threaded instances of TRSM and GEMM to exploit BDP only. Compared with this, the panel factorization of A_p, which lies in the critical path of the blocked RL factorization algorithm, exhibits a reduced degree of concurrency. Thus, depending on the selected block size b and certain hardware features of the target architecture (number of cores, floating-point performance, memory bandwidth, etc.), this operation may easily become a performance bottleneck, as shown in Figure 4.4 for a given iteration k.

To illustrate the performance relevance of the panel factorization, Figure 4.5 displays a fragment of a trace corresponding to the LU factorization of a $10{,}000 \times 10{,}000$ matrix, using the blocked RL algorithm in Figure 4.3, with partial pivoting and "outer" block size $b = b_o = 256$. (All traces were obtained using `Extrae` version 3.3.0 [87].) The code is linked with multi-threaded versions of the

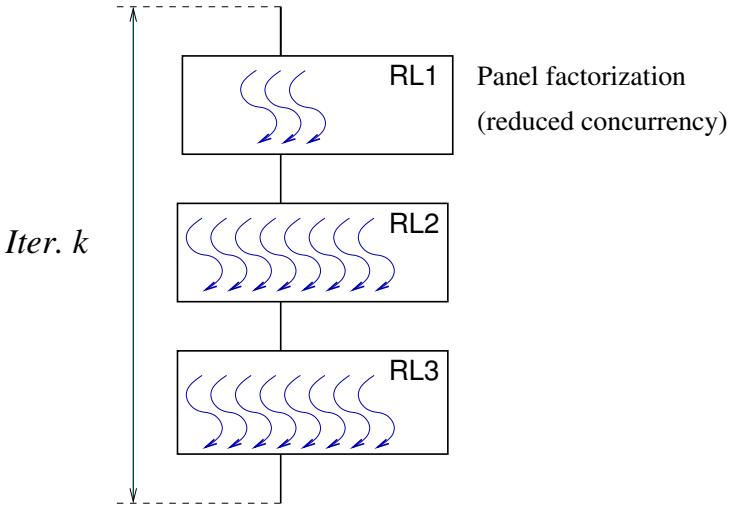

Figure 4.4: Exploitation of BDP in the blocked RL LU parallelization. A single thread team executes all the operations, with less active threads for RL1 due to the reduced concurrency of this kernel. In this algorithm, RL1 stands in the critical path.

BLIS kernels for GEMM and TRSM, using 6 threads in both cases. The panel factorization (PANEL) is performed via a call to the same blocked algorithm, with "inner" block size $b_i = 32$, and also extracts BDP from the same two kernels. With this configuration, the panel factorization represents less than 2% of the flops performed by the algorithm. However, the trace of the first four iterations reveals that its practical cost is much higher than could be expected. (The cost of factorizing a panel relative to the cost of an iteration becomes even larger as the iteration progresses.) Here we note also the significant cost of the row permutations, which are performed via the sequential legacy code for this routine in LAPACK (LASWP). However, this second operation is embarrassingly parallel and its execution time can be expected to decrease linearly with the number of cores, although scalability may be limited because it is a memory-bound operation.

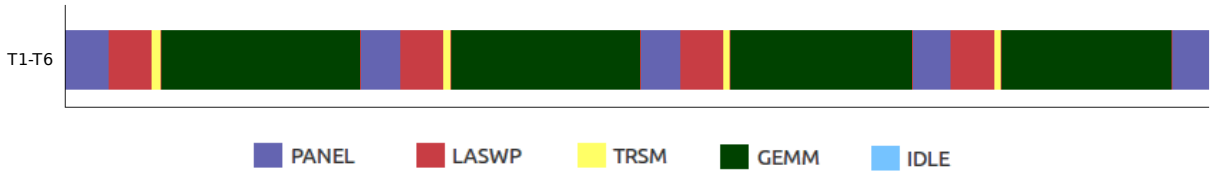

Figure 4.5: Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$.

At this point, we note that the operations inside the loop body of the blocked algorithm in Figure 4.3 (right) present strict dependencies that enforce their computation in the order RL1 $\Rightarrow$ RL2 $\Rightarrow$ RL3. Therefore, there seems to be no efficient manner to formulate a task-parallel (TP) version of the blocked algorithm in that figure.

4.3.2 Static look-ahead and nested Task-Parallelism+Block-Data Parallelism (TP+BDP)

A strategy to tackle the hurdle represented by the panel factorization in a parallel execution consists in the introduction of look-ahead [92] into the algorithm. Concretely, during each iteration of the decomposition this technique aims to overlap the factorization of the "next" panel with the update of the "current" trailing submatrix, in practice enabling a TP version of the algorithm with two separate branches of execution, as discussed next.

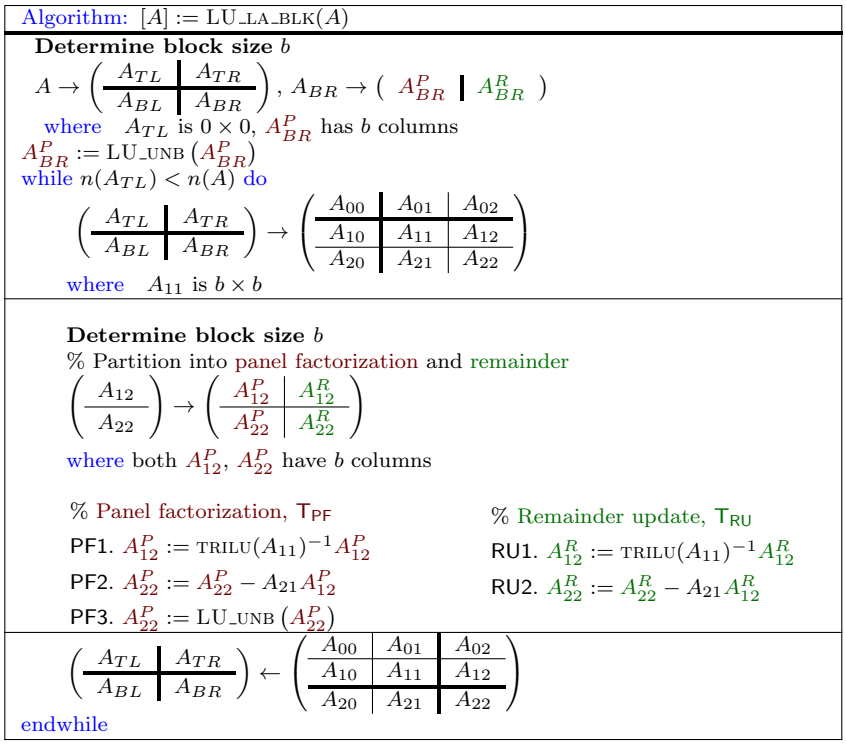

Figure 4.6: Blocked RL algorithm enhanced with look-ahead for the LU factorization.

Figure 4.6 illustrates a version of the blocked RL algorithm for the LU factorization re-organized to expose look-ahead. The key is to partition the trailing submatrix into two block column panels:

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) \rightarrow \left(\begin{array}{c|c} A_{12}^P & A_{12}^R \\ \hline A_{22}^P & A_{22}^R \end{array}\right) \tag{4.1}$$

where A_{22}^P corresponds to the block that, in the conventional version of the algorithm (i.e., without look-ahead,) would be factorized during the next iteration. This effectively separates the blocks that are modified as part of the next panel factorization from the remainder updates, left and right of the 2×2 partitioning in (4.1), respectively. Proceeding in this manner creates two coarse-grain independent tasks (groups of operations in separate branches of execution): $\mathsf{T_{PF}}$, consisting of PF1, PF2, PF3; and $\mathsf{T_{RU}}$, composed of RU1 and RU2 (Figure 4.6). The "decoupling" of these block panels thus facilitates that, in a TP execution of an iteration of the loop body of the look-ahead version, the updates on A_{12}^P, A_{22}^P and the factorization of the latter (operations on the next panel, in $\mathsf{T_{PF}}$) proceed concurrently with the updates of A_{12}^R, A_{22}^R (remainder operations, in $\mathsf{T_{RU}}$), as there are no dependencies between $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$.

By carefully tuning the block size b and adjusting the amount of computational resources (threads) dedicated to each of the two independent tasks, $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$, a nested TP+BDP execution of the algorithm enhanced with this static look-ahead can partially or totally overcome the bottleneck represented by the panel factorization, as shown in Figure 4.7 fo a given iteration k.

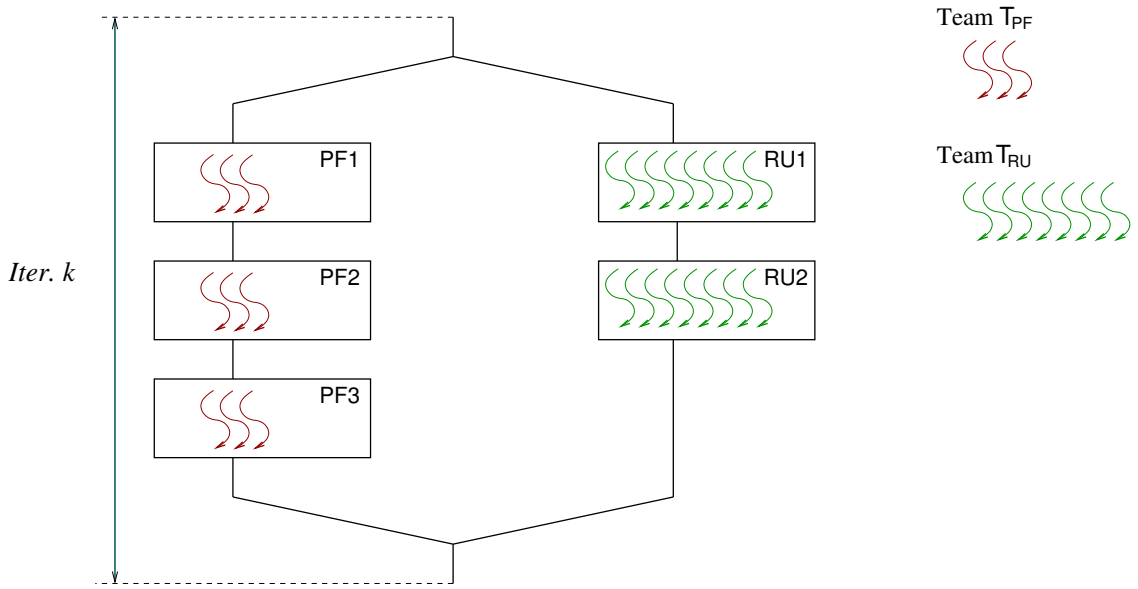

Figure 4.7: Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead. The execution is performed by teams PF and RU, consisting of $t_{\mathsf{pf}} = 3$ and $t_{\mathsf{ru}} = 8$ threads, respectively. In this algorithm, the operations on the $(k+1)$-th panel, including its factorization (PF3), are overlapped with the updates on the remainder of the trailing submatrix (RU1 and RU2).

Figure 4.8 illustrates a complete overlap of $\mathsf{T_{RU}}$ with $\mathsf{T_{PF}}$ attained by the look-ahead technique. The results in that figure correspond to a fragment of a trace obtained for the LU factorization of a 10,000 $\times$ 10,000 matrix, using the blocked RL algorithm in Figure 4.6, with partial pivoting, and outer block size $b = b_o = 256$. For this experiment, the $t = 6$ threads are partitioned into two teams: PF with $t_{\mathsf{pf}} = 1$ thread in charge of $\mathsf{T_{PF}}$, and RU with $t_{\mathsf{ru}} = 5$ threads responsible for $\mathsf{T_{RU}}$. The panel factorization (PANEL) is performed via a call to the same algorithm, with $b_i = 32$, and this operation proceeds sequentially (as PF consists of a single thread). The application of the row permutations is distributed between all 6 cores. As argued earlier, the net effect of the look-ahead is that the cost of the panel factorization no longer has a practical impact on the execution time of the (first four iterations of) the factorization algorithm, which is now basically determined by the cost of the remaining operations.

Given a static mapping of threads to tasks, b should balance the time spent in the two tasks as, if the operations in $\mathsf{T_{PF}}$ take longer than those in $\mathsf{T_{RU}}$, or vice-versa, the threads in charge of the less expensive part will become idle, causing a performance degradation. This was already visible in Figure 4.8, which shows that, during the first four iterations, the operations in $\mathsf{T_{PF}}$ are considerably less expensive than the updates performed as part of the remainder $\mathsf{T_{RU}}$. The complementary case, where $\mathsf{T_{PF}}$ requires longer than $\mathsf{T_{RU}}$, is illustrated using the same configuration, for a matrix of dimension 2,000 $\times$ 2,000, in Figure 4.9. Unfortunately, as the factorization proceeds, the theoretical costs and execution times of $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$ vary, making it difficult to determine the optimal value of b, which will need to be adapted during the factorization process.

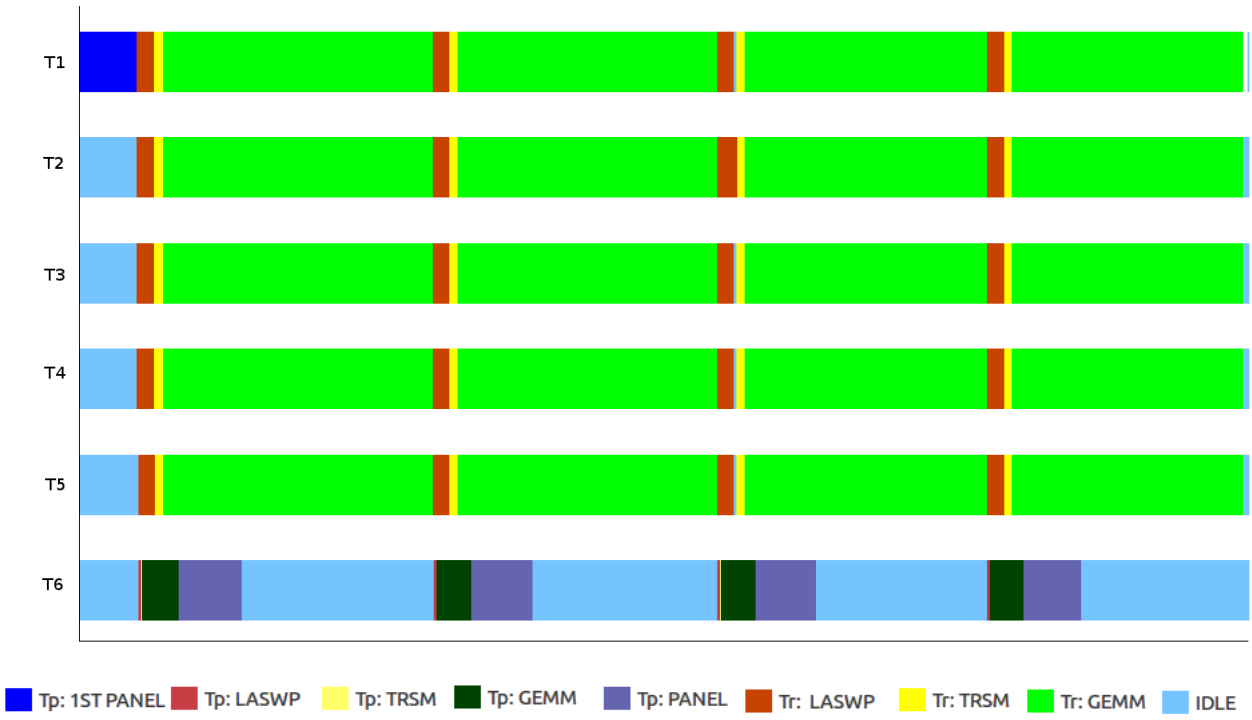

Figure 4.8: Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$.

To close this section, note that there exist strict dependencies that serialize the operations within each task: PF1 $\Rightarrow$ PF2 $\Rightarrow$ PF3 and RU1 $\Rightarrow$ RU2. Therefore, there is no further TP in the loop-body of this re-organized version. However, the basic look-ahead mechanism of level/depth 1 described in this subsection can be refined to accommodate further levels of TP, by "advancing" to the current iteration the panel factorization of the following d iterations, in a look-ahead of level/depth d. This considerably complicates the code of the algorithm, but can be seamlessly achieved by a runtime system enhanced with priorities.

4.3.3 Advocating for Malleable Thread-Level Linear Algebra Libraries

Let us assume, for simplicity, that $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$ consist only of the panel factorization involving A_{22}^P (PF3) and the update of A_{22}^R (RU2), respectively. Furthermore, let us consider a nested TP+BDP execution using $t = t_{\mathsf{pf}} + t_{\mathsf{ru}}$ threads, *initially* with a team PF of t_{pf} threads mapped to the execution of PF3 and a team RU of t_{ru} threads computing RU2.

Ideally, for the LU factorization with look-ahead, we would like to perform a *flexible sharing* of the computational resources so that, as soon as the threads in team PF complete PF3, they join team RU to help in the execution of RU2 or vice-versa. However, in practice, if team RU joins team PF performance will not be increased due to the low degree of parallelism of $\mathsf{T_{PF}}$. In this case we propose a slightly different approach referred to as early termination. We next discuss these two cases in detail.

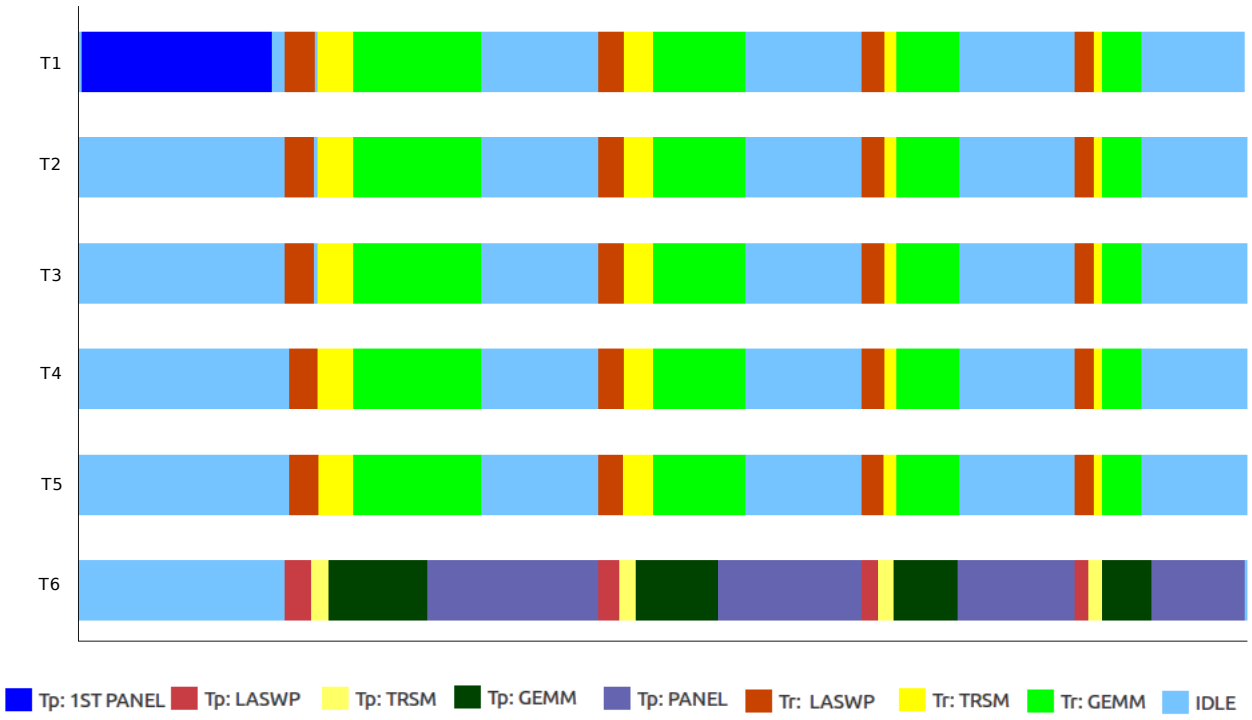

Figure 4.9: Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead, using 6 threads, applied to a square matrix of order 2,000, with $b_o = 256$, $b_i = 32$.

4.3.3.1 Worker sharing (WS): Panel factorization less expensive than update

Our goal is to enable that, at each iteration of the algorithm for the LU factorization with look-ahead, the threads in team PF that complete the panel factorization join the thread team RU working on the update. In this scenario, the problem is that, if the multiplication to update A_{22}^R was initiated via an invocation to a traditional GEMM, this is not possible as none of the existing high performance implementations of BLAS allows a modification of the number of threads working on a kernel that is already in execution to be modified. The migration of threads from one task to another is referred to as worker sharing (WS).

Suboptimal solution: Static re-partitioning

A simple workaround for this problem is to split A_{22}^R into multiple column blocks, for example, $A_{22}^R \rightarrow \left(A_1 \mid A_2 \mid \ldots \mid A_q \right)$, and to perform a separate call to BLAS GEMM in order to exploit BDP during the update of each block. Then, just before each invocation, the kernel's code queries whether the execution of the panel factorization is completed and, if that is the case, executes the suboperation with the threads from both teams (or only those of RU otherwise). Unfortunately, this approach presents several drawbacks:

- Replacing a single invocation to a coarse GEMM by multiple calls to smaller GEMM may offer lower throughput because the operands passed to GEMM are smaller and/or suboptimally "shaped". The consequence is that calling GEMM multiple times will internally incur re-

packing and data movement overheads, which are more difficult to amortize because of the smaller problem dimensions.

- The decision of which loops to partition for parallelism (note that A_{22}^R could have alternatively been split by rows, or into blocks), and the granularity of this partitioning is then placed upon the programmer's shoulders, who may lack the information that is necessary to make a good choice. For example, if the granularity is too coarse, this will have negative effect because the integration of the single thread in the update will likely be delayed. A granularity that is too fine, on the other hand, may reduce the parallelism within the BLAS operation or result in the use of cache blocking parameters that are too small.

Malleable thread-level BLAS (MTL)

The alternative that we propose exploits BDP inside RU2, *but allows to change the number of threads that participate in this computation even if the task is already in execution.* In other words, threads are viewed as a resource pool of workers that can be shared between different tasks and reassigned to the execution of a (BLAS) kernel that is already in progress. Those BLAS libraries that include this approach are hereafter referred to as malleable thread level (MTL) BLAS.

The key to our approach lies in the explicit exposition of the GEMM internals (and other BLAS-3 kernels) in BLIS. Concretely, assume that RU2 is computed via a single invocation to BLIS GEMM, and consider that this operation is parallelized by distributing the iteration space of Loop 4 among the threads in team PF (Figures 3.1 and 4.10). Then, just before Loop 4, we force the system to check if the execution of the panel factorization is completed and, based on this information, decides whether this loop is executed using either the union of the threads from both teams or only those in RU (Figure 4.11).

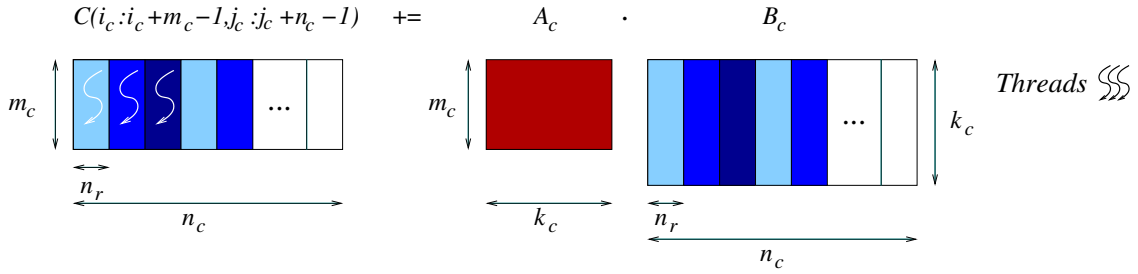

Figure 4.10: Distribution of the workload among $t = 3$ threads when Loop 4 of BLIS GEMM is parallelized. Different colors in the output C distinguish the panels of this matrix that are computed by each thread as the product of A_c and corresponding panels of the input B_c.

Let us re-analyze the problems listed for the work-around solution that statically partitioned the update of A_{22}^R, and compare them with our solution that implicitly embeds this partitioning inside BLIS:

- The partitioning of GEMM into multiple calls to smaller matrix multiplications does not occur. Our solution performs a single call to GEMM only, so that there is no additional re-packing nor data movements. For example, in the case just discussed, B_c is already packed and re-used independently of whether t or t_{ru} threads participate in the GEMM. The buffer A_c is packed only once per iteration of Loop 3 (in parallel by both teams or only RU).

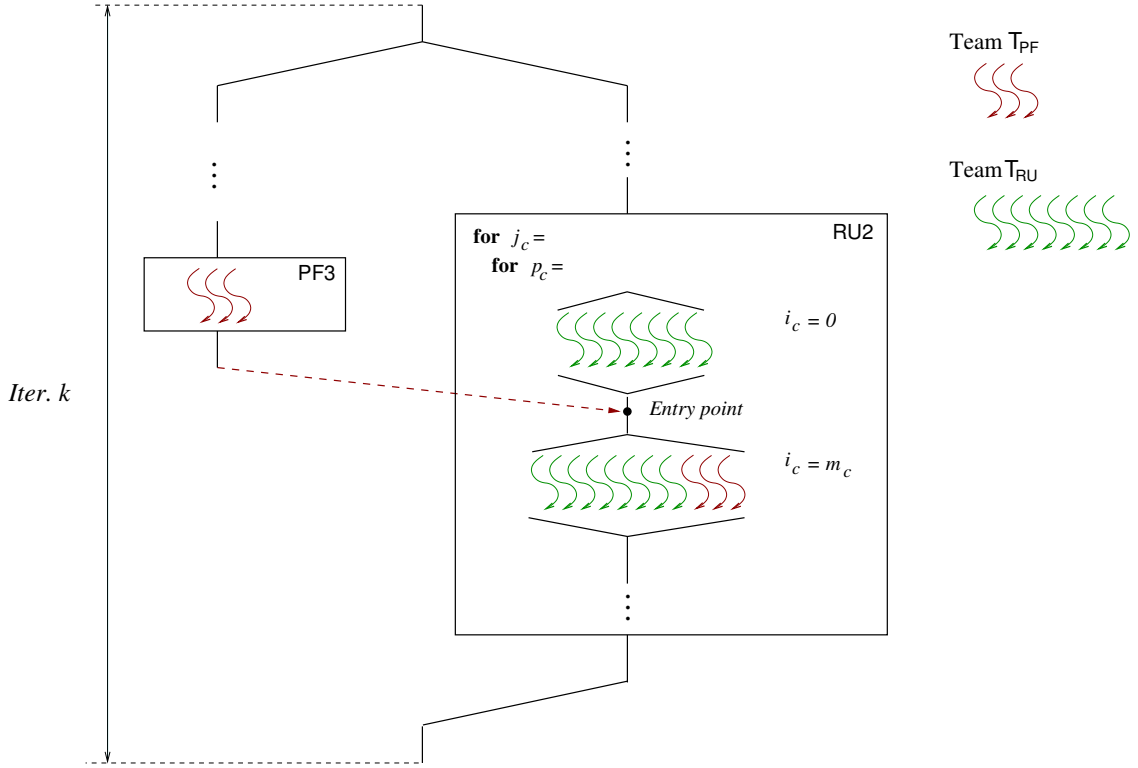

Figure 4.11: Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead and WS. The execution is performed by teams PF and RU, consisting of $t_{\mathsf{pf}} = 3$ and $t_{\mathsf{ru}} = 8$ threads, respectively. In this example, team PF completes the factorization PF3 while team RU is executing the first iteration of Loop 3 that corresponds to RU2/GEMM ($i_c = 0$). Both teams then merge and jointly continue the update of the remaining iterations of that loop ($i_c = m_c, 2m_c, \ldots$). With the parallelization of GEMM Loop 4, one such "entry point" enables the merge at the beginning of each iteration of loop i_c.

- The decision of the best partitioning/granularity is left in the hands of BLIS, which likely has more information to do a better job than the programmer.

Importantly, the partitioning occurs dynamically and is transparent to the programmer.

Figure 4.12 validates the effect of integrating a malleable version of BLIS into the same configuration that produced the results of Figure 4.8. A comparison of both figures shows that, with a malleable version of BLIS, the thread executing the operations in T_{PF}, after completing this task, rapidly joins the team that computes the remainder updates, thus avoiding the idle wait.

Compared with BLIS, the same approach cannot be integrated into GotoBLAS because the implementation of GEMM in this library only exposes the three outermost loops of Figure 3.1, while the remaining loops are encoded in assembly. The BLAS available as part of commercial libraries is not an option either because hardware vendors offer black-box implementations which do not permit the migration of threads.

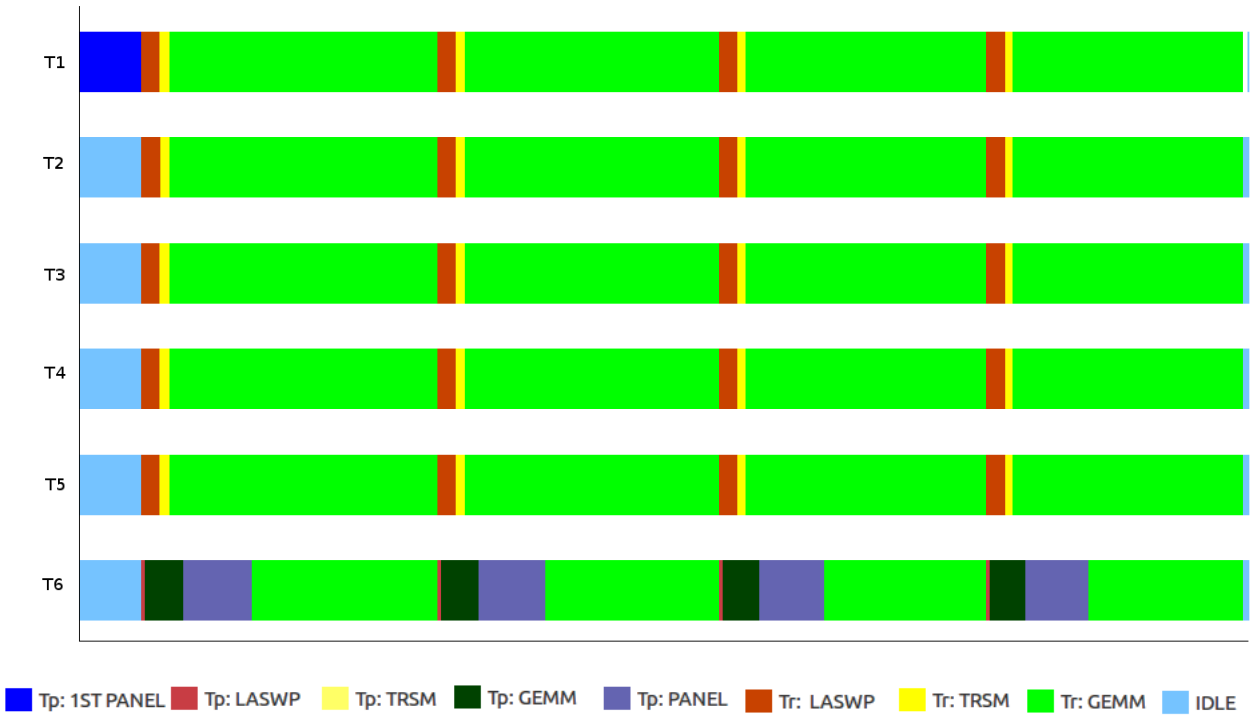

Figure 4.12: Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead and *malleable BLIS*, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$.

4.3.3.2 Early termination: Panel factorization more expensive than update

The analysis of this case will reveal some important insights. In order to discuss them, let us consider that, in the LU factorization with look-ahead, the panel factorization (PF3) is performed via a call to the blocked routine in Figure 4.3 (right). We assume two blocking parameters: $b = b_o$ for the outer routine that computes the LU factorization of the complete matrix using look-ahead, and b_i for the inner routine that factorizes each panel. (Note that, if $b_i = b_o$ or b_i=1, the panel factorization is then simply done via the unblocked algorithm.) Furthermore, we will distinguish these two levels by referring to them as the *outer LU* (factorization with look-ahead) and the *inner LU* (factorization of the panel via the blocked algorithm without look-ahead). Thus, at each iteration of the outer LU, a panel of b_o columns is factorized via a call to LU_BLK (inner LU), and this second decomposition proceeds to factorize the panel using a blocked algorithm with block size b_i (Figure 4.13).

From Figure 4.3 (right), the loop body for the inner LU consists of a call to the unblocked version of the algorithm (RL1), followed by the invocations to TRSM and GEMM that update A_{12} and A_{22}, respectively (RL2 and RL3). Now, let us assume that the update RU2 by the thread team RU is completed while the threads of team PF are in the middle of the computations corresponding to an iteration of the loop body of the inner LU. Then, provided the versions of the BDP versions TRSM and GEMM kernels that are invoked from the inner LU are malleable, inside them the system will perform the actions that are necessary to integrate the thread team RU, which is now idle, into the corresponding (and subsequent) computation(s). Unfortunately, the updates in the loop body of the inner LU involve small-grained computations (A_{12} and A_{22} have at most $b_o - b_i$ columns,

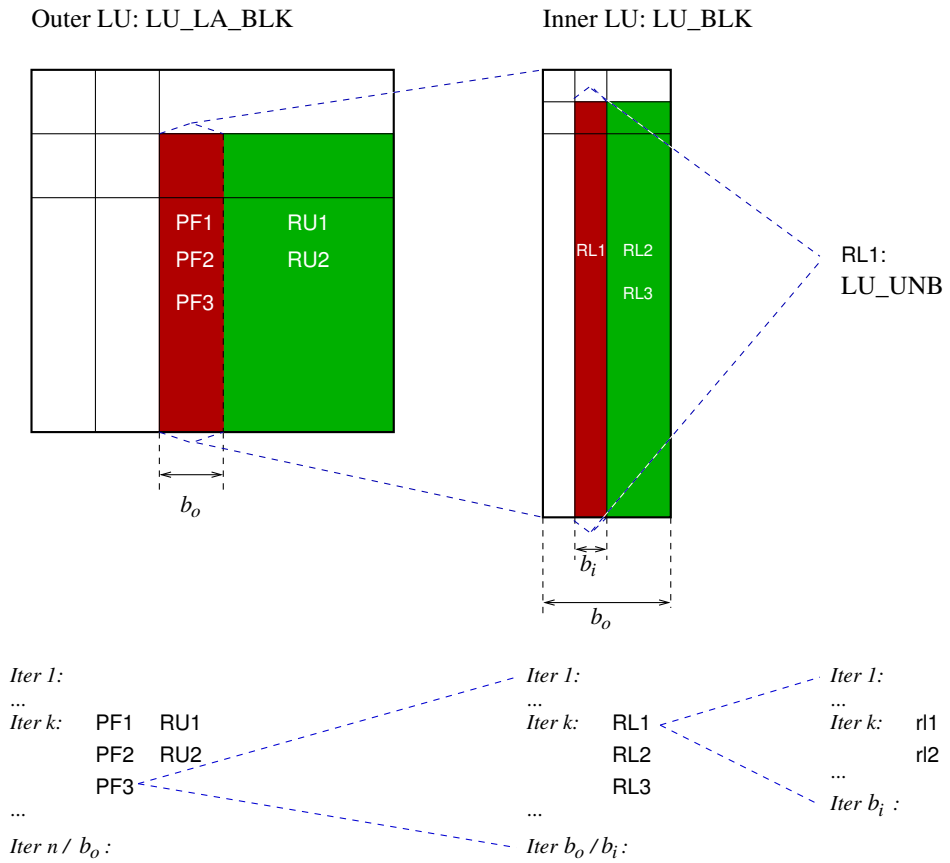

Figure 4.13: Outer vs inner LU and use of algorithmic block sizes.

decreasing by b_i columns at each iteration), and little parallel performance can be expected from it, especially because of partial pivoting.

In order to deal with this scenario, a different option is to force the inner LU to stop at the end of the current iteration, to then rapidly proceed to the next iteration of the outer LU. We refer to this strategy as *early termination* (ET). In order to do this though, the transformations computed to factorize the current inner-panel must be propagated first to the remaining columns outside this panel, introducing a certain delay in this version of the ET strategy due to the implementation of the RL version of the LU factorization.

An alternative is to rely on a left-looking (LL) version of the LU factorization for the inner LU, as discussed next. The blocked LL algorithm for the LU factorization differs from the blocked RL variant (algorithm in the right-hand side of Figure 4.3) in the operations performed inside the loop-body, which are replaced by

$$\begin{aligned}
\text{LL1.} \quad & A_{01} && := \quad \text{TRILU}(A_{00})^{-1} A_{01}, \\
\text{LL2.} \quad & \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} && := \quad \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} - \begin{bmatrix} A_{10} \\ A_{20} \end{bmatrix} A_{01}, \\
\text{LL3.} \quad & \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} && := \quad \text{LU_UNB}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \right).
\end{aligned}$$

Thus, at the end of a certain iteration, this variant has updated only the current column of the inner-panel and those to its left. In other words, no transformations are propagated beyond that point

(i.e., to the right of the current column/inner-panel), and ET can be implemented in a straightforward manner, with no delay compared with an inner LU factorization via the RL variant.

A definitive advantage of the LL variant compared with its RL counterpart is that the former implements a lazy algorithm, which delays the operations towards the end of the panel factorization, while the second corresponds to an eager algorithm that advances as much computations as possible to the initial iterations. Therefore, in case the panel factorization has to be stopped early, it is more likely that the LL variant has progressed in the factorization further. For example, consider the factorization of an $m \times n$ matrix that is stopped at iteration $k < n$. The LL algorithm will have performed $m^2k - m^3/3$ flops at that point while, for the RL algorithm, the flop count raises to that of the LL algorithm plus $2(n-k)(mk - k^2/2)$. The appealing consequence of using the LL algorithm is that it enables the use of larger block sizes for the following updates in the LL variant.

From an implementation point of view, the synchronization between the two teams of threads is easy to handle. For example, at the beginning of each iteration of the outer LU, a boolean flag is reset to indicate that the remainder update is incomplete. The thread team RU then sets this value as soon as this task is complete. In the mean time, the flag is queried by the thread team PF, at the end of every iteration of the inner LU, aborting its execution when a change is detected. With this operation mode, there is no need to protect the flag from race conditions. This solution also provides an adaptive (automatic) configuration of the block size as, if chosen too large, it will be adjusted for the current (and, possibly, subsequent) iterations by the early termination of the inner LU. The process is illustrated in Figure 4.14.

Figure 4.15 shows the effect of integrating the ET mechanism into the same configuration that produced the results in Figure 4.9. A comparison between both figures shows that, with the ET, as soon as the RU team is done with the update, this situation is notified to the PF team and the execution of the panel factorization is aborted in order to make all the existing threads move on to the next iteration of the factorization. Note that the time scale is different in both figures, and that the first GEMM performed by the $\mathsf{T_{PF}}$ team looks much larger now; in fact, the duration of this GEMM is exactly the same as before (b_o is still 256 in the first iteration) and the three remaining GEMM are much shorter due to the automatic adaption of the algorithmic block size performed by the ET mechanism.

4.3.3.3 Relation to adaptive look-ahead via a runtime

Compared with our approach, which only applies look-ahead at one level, a TP execution that relies on a run-time for adaptive-depth look-ahead exposes a higher degree of parallelism from "future iterations", which can amortize the cost of the panel factorization over a considerably larger number of flops. This can be beneficial for architectures with a large number of cores, but can be partially compensated by increasing the number of threads dedicated to the panel factorization, combined with a careful fine-grain exploitation of the concurrency [20], in our approach. On the other hand, adaptive-depth look-ahead via a runtime suffers from re-packing and data movement overheads due to multiple calls to GEMM. Moreover, it couples the algorithmic block size that fixes the granularity of the tasks to that of the suboperands in GEMM. Finally, the runtime-based solutions rarely exploit nested TP+BDP parallelism and, even if they do so, taking advantage of a MTL BLAS from within them may be difficult.

4.3.4 Experimental Evaluation

In this section we analyze in detail the performance behavior of several multi-threaded implementations of the algorithms for the LU factorization:

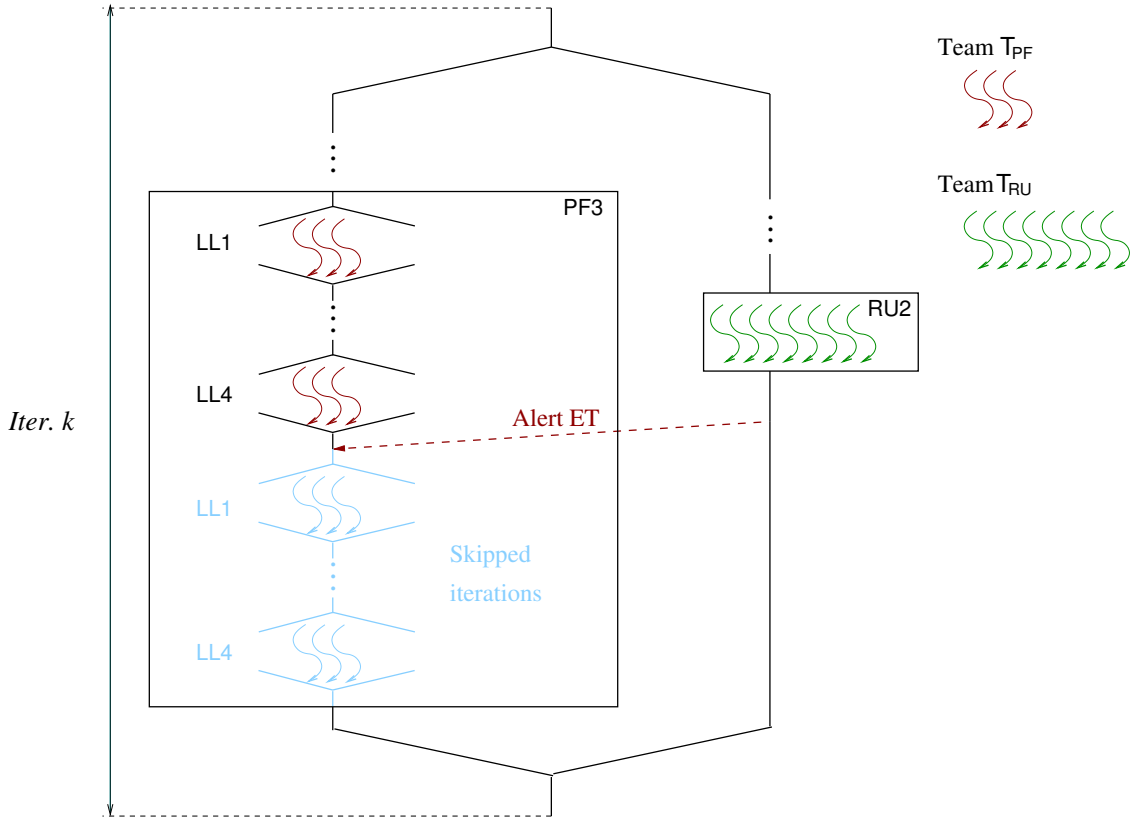

Figure 4.14: Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead and ET. The execution is performed by teams $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$, consisting of $t_{\mathsf{pf}} = 3$ and $t_{\mathsf{ru}} = 8$ threads, respectively. In this example, team $\mathsf{T_{RU}}$ completes the update RU2 while team $\mathsf{T_{PF}}$ is executing an iteration of the panel factorization PF3. $\mathsf{T_{RU}}$ then notifies of this event to $\mathsf{T_{PF}}$, which then skips the remaining iterations of the loop that processes the panel.

- LU: Blocked RL (Figure 4.3). This code only exploits BDP, via calls to the (non-malleable) multi-threaded BLIS (version 0.1.8).
- Variants enhanced with look-ahead (Figure 4.6). The following three implementations take advantage of nested TP+BDP, with n threads to the operations on the panel (team PF) and $t - n$ to the remainder updates (team RU).
  - LU_LA (subsection 4.3.2): Blocked RL with look-ahead.
  - LU_MB (subsection 4.3.3.1): Blocked RL with look-ahead and malleable BLIS.
  - LU_ET (subsection 4.3.3.2): Blocked RL with look-ahead, malleable BLIS, and early termination of the panel factorization.
- LU_OS: Blocked RL with adaptive look-ahead extracted via the OmpSs runtime (version 16.06). LU_OS decomposes the factorization into a large collection of tasks connected via data dependencies, and then exploits TP only, via calls to a sequential instance of BLIS. In more detail, the OmpSs parallel version divides the matrix into a collection of panels of fixed width b_o. All operations performed during an iteration of the algorithm on the same panel (row permutation, triangular system solve, matrix multiplication and, possibly, panel

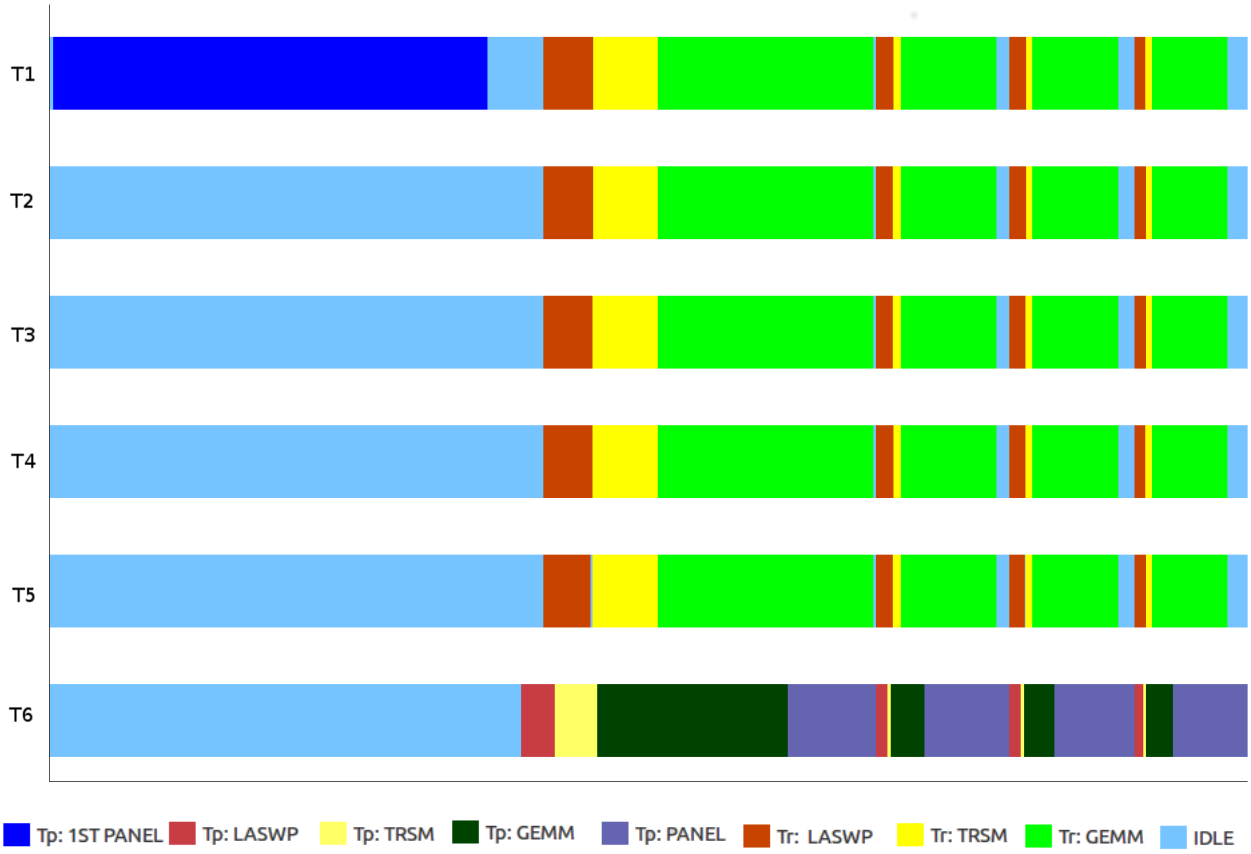

Figure 4.15: Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead and early termination, using 6 threads, applied to a square matrix of order 2,000, with initial $b_o = 256$ and $b_i = 32$.

factorization) are then part of the same task. This implementation includes priorities to advance the schedule of tasks involving panel factorizations.

All codes include standard partial pivoting and compute the same factorization. Also, all solutions perform the panel factorization via the blocked RL algorithm, except for LU_ET and LU_OS, which employ the blocked LL variant. The performance differences between the LL and RL variants, when applied solely to the panel factorization, were small. Nonetheless, for LU_ET, employing the LL variant improves the ET mechanism and unleashes a faster execution of the global factorization. For LU_OS we integrated the LL variant as well to favor a fair comparison between this implementation and our LU_ET. The block size is fixed to b_o during the complete iteration in all cases, except for LU_ET which initially employs b_o, but then adjusts this value during the factorization as consequence of the ET mechanism.

In the experiments, we considered the factorization of square matrices, with random entries uniformly distributed in (0,1), and dimension $n = 500$ to 12,000 in steps of 500. The block size for the outer LU was tested for values $b_o = 32$ to 512 in steps of 32. The block size for the inner LU was evaluated for $b_i = 16$ and 32. We employed one thread per core (i.e., $t = 6$) in all executions.

4.3.4.1 Optimal block size

The performance of the blocked LU algorithms is strongly influenced by the outer block size b_o. As discussed in subsection 4.3.1, this parameter should balance two criteria:

- Deliver high performance for the GEMM kernel. Concretely, in the algorithms in Figures 4.3 and 4.6, a value of b_o that is too small turns A_{21} and A_{12}/A_{12}^R into narrow column and row panels respectively, transforming the matrix multiplication involving these blocks (RL3 in Figure 4.3 and RU2 in Figure 4.6) into a memory-bound kernel that will generally deliver low performance. Note that, for GEMM $m \approx n \gg k$ and $k = b_o$.
- Reduce the amount of operations performed in the panel factorization (about $n^2 b_o/2$ flops, provided $n \gg b_o$), in order to avoid the negative impact of this sequential stage.

Figure 4.16 sheds further light on the roles played by these two factors. The plot in the left-hand side reports the performance of GEMM, in terms of GFLOPS, showing that the implementation of this kernel in BLIS achieves an asymptotic performance peak for $k(= b_o)$ around 144. (The performance drop observed for k slightly above 256 is due to the optimal value of k_c being equal to that number in this architecture, as discussed in Chapter 3.) The right-hand side plot reports the ratio of flops performed in the panel factorizations with respect to those of the LU factorization.

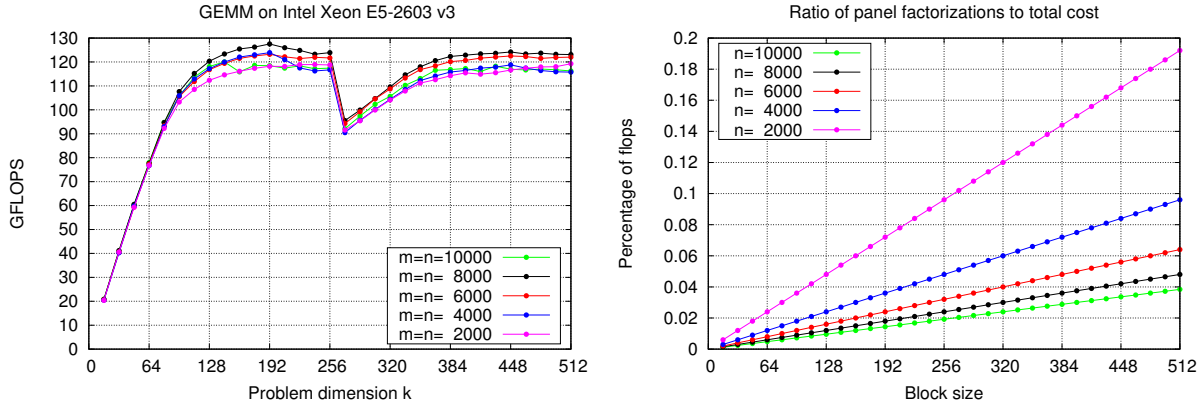

Figure 4.16: GFLOPS attained with GEMM (left) and ratio of flops performed in the panel factorizations normalized to the total cost (right).

The combined effect of these criteria seems to point in the direction of choosing the smallest b_o that attains the asymptotic GFLOPS rate for GEMM. However, Figure 4.17 illustrates the experimental optimal block size b_o for the distinct LU factorization algorithms, exposing that this is not the case. We next discuss the behavior for LU, LU_LA and LU_MB, which show different trends. (LU_ET and LU_OS will be analyzed latter.) In particular, LU benefits from the use of larger values of b_o than the other two codes for all problem dimensions. The reason is that a large block size operates on wide panels, which turns their factorization into a BLAS-3 operation with a mild degree of parallelism, and reduces the impact of this computation on the critical path of the factorization. LU_LA exhibits a similar behavior for large problems, but favors smaller block sizes for small to moderate problems. The reason is that, for LU_LA, it is important to balance the panel factorization ($\mathsf{T_{PF}}$) and remainder update ($\mathsf{T_{RU}}$) so that their execution approximately requires the same time.

Compared with the previous two implementations, LU_MB promotes the use of small block sizes, up to $b_o = 192$, for the largest problems. (Interestingly, this corresponds to the optimal

value of k for GEMM.) One reason for this behavior is that, when the malleable version of BLIS is integrated into LU_MB, the practical costs of the two branches/tasks do not need to be balanced. Let us elaborate this case further, by considering the effect of reducing the block size, for example, from b_o to $b'_o = b_o/2$. For simplicity, in the following discussion we will use approximations for the block dimensions and their costs; furthermore, we will assume that $n \gg b_o$. The first and most straight-forward consequence of halving the block size is that the number of iterations is doubled. Inside each iteration with the original block size b_o, the loop body invokes, among others kernels, a GEMM of dimensions $m \times (m - b_o) \times b_o$ (with m the number of rows in the trailing submatrix A_{22}^R), for a cost of $2m^2b_o$ flops; in parallel, the factorization involves a panel of dimension $m \times b_o$, for a cost of $mb_o^2 - b_o^3/3 \approx mb_o^2$ flops. When the block size is halved to b'_o, the same work is basically computed in two consecutive iterations. However, this reduces the amount of flops performed in terms of panel factorizations to about $2m(b'_o)^2 = mb_o^2/2$ while it has a minor impact on the number of flops that are cast as GEMM (two of these products, at a cost of $2m^2b'_o = 2m^2b_o/2$ flops each). The conclusion is that, by reducing the block size, we decrease the time that the single thread spends in the panel factorization $\mathsf{T_{PF}}$, favoring its rapid merge with the thread team that performs the remainder update $\mathsf{T_{RU}}$. Thus, in case the execution time of the LU is dominated by $\mathsf{T_{RU}}$, adding one more thread to perform this task (in this scenario, in the critical path) as soon as possible will reduce the global execution time of the algorithm.

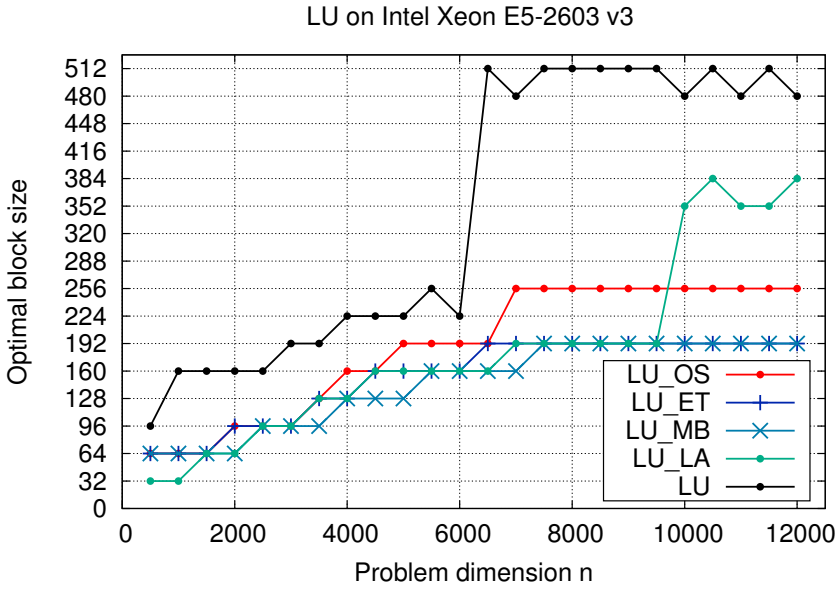

Figure 4.17: Optimal block size of the blocked RL algorithms for the LU factorization.

4.3.4.2 Performance comparison of the variants with static look-ahead

The previous analysis on the effect of the block size exposes that choosing the optimal block size is a difficult task. Either we need a model that can accurately predict the performance of each building block appearing in the LU factorization, or we perform an extensive experimental analysis to select the best value. The problem is even more complex if we consider that, in practice, an optimal selection would have to vary the block size as the factorization progresses. Concretely, for the factorization of a square matrix of order n via a blocked algorithm, note that the problem is decomposed into multiple subproblems that involve the factorization of matrices of orders $n - b_o$,

$n-2\cdot b_o$, $n-3\cdot b_o$, etc. From Figure 4.17, it is clear that the optimal value of b_o will be different for several of these subproblems. In the end, the value that we show in Figure 4.17 for each problem has to be considered as a compromise that attains fair performance for a wide range of the subproblems appearing in that case.

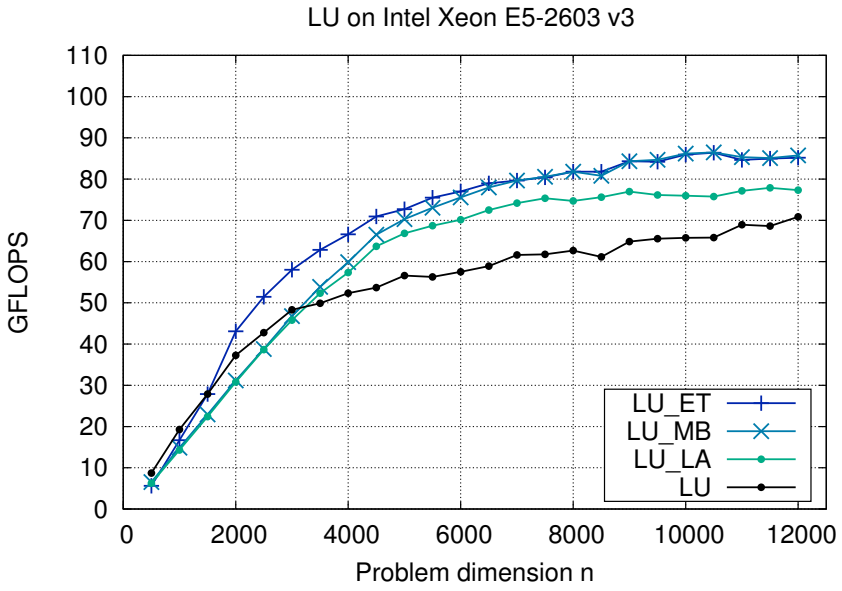

Figure 4.18: Performance comparison of the blocked RL algorithms for the LU factorization (except LU_OS) with a fixed block size $b_o = 256$.

Figure 4.18 reports the GFLOPS rates attained by the distinct implementations to compute the plain LU factorization and the variants equipped with static look-ahead (i.e., all except LU_OS), using $b_o = 256$ as a compromise value for all of them. Although this value is optimal for only a few cases, the purpose of this experiment is to show the improvements attained by gradually introducing the techniques enhancing look-ahead. The figure reveals some relevant trends:

- Except for the smallest problems, integrating the look-ahead techniques clearly improves the performance of the plain LU factorization implemented in LU.

- The version with malleable BLAS (LU_MB) improves the performance of the basic version of look-ahead (LU_LA) for the larger problems. This is a consequence of the cost of the panel factorization relative to that of the global factorization. Concretely, for fixed b_o, as the problem size grows, the global flop-cost varies cubically in n, as $2n^3/3$, while the flop-cost of the panel factorizations grows quadratically, with $n^2 b_o/2$. Thus, we can expect that, for large n, the remainder update $\mathsf{T_{RU}}$ becomes more expensive than the panel factorization $\mathsf{T_{PF}}$. This represents the actual scenario that was targeted by the variant with malleable BLIS.

- The version that combines the malleable BLAS with ET (LU_ET) delivers the same performance of LU_MB for large problems, but outperforms all other variants with static look-ahead for the smaller problems. Again, this could be expected by considering the relative cost of the panel factorization for small n.

4.3.4.3 Performance comparison with OmpSs

We next extend the experimental analysis by providing a comparison of the best variant with static look-ahead, LU_ET, with the implementation that extracts parallelism via the OmpSs runtime, LU_OS. In this experiment we depart from the previous case, performing an extensive evaluation in order to report the performance for the optimal block size for each problem dimension and algorithm. The actual optimal values employed in the experiment are those extracted from Figure 4.17. For LU_OS, we select a value for b_o that is then fixed for the complete factorization. As this variant overlaps the execution of tasks from different iterations in time, it is difficult to vary the block size as the factorization progresses. For LU_ET, the selected value of b_o only applies to the first iteration. After that, the ET mechanism automatically adjusts this value during the factorization process.

Figure 4.19 shows the results for this comparison in the lines labelled as "(b_opt)". LU_ET is very competitive, clearly outperforming the runtime-based solution for most problems and offering competitive performance for the largest four.

Manually tuning the block size to each problem dimension is in general impractical. For this reason, the figure also shows the performance curves when the block size is fixed to $b_o = 192$ for LU_ET and $b_o = 256$ for LU_OS. These values were selected because they offered high performance for a wide range of problem dimensions, especially, the largest ones, as reported in Figure 4.17. Interestingly, the performance lines corresponding to this configuration, labelled with "(b=192)"/"(b=256)", show that choosing a suboptimal value for b_o has a minor impact on the performance of our solution LU_ET, because the ET mechanism adjust this value on-the-fly (for the smaller problem sizes). Compared with this, the negative effect of a suboptimal selection on LU_OS is clearly more visible. This effect is observed for small problem dimensions where the gap between the lines for the optimal block size and the fixed block size is larger.

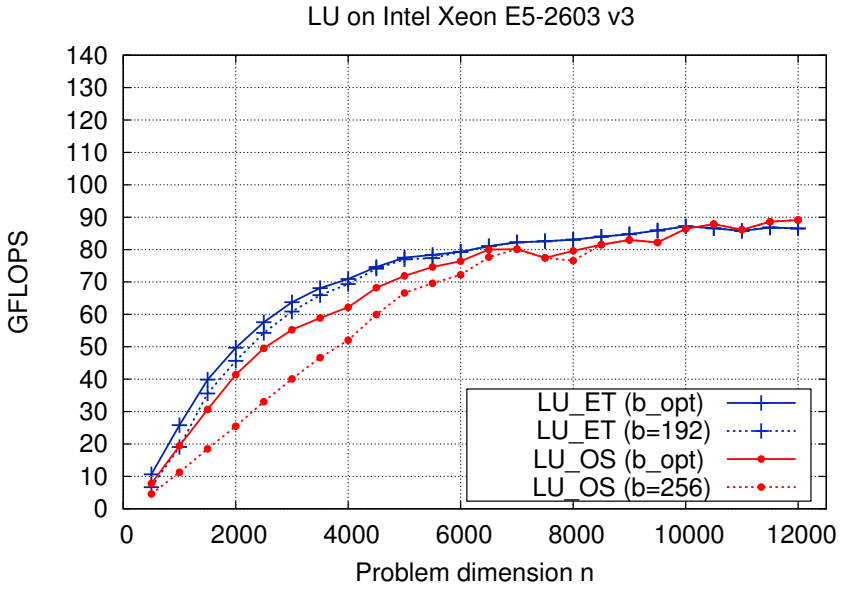

Figure 4.19: Performance comparison between the OmpSs implementation and the blocked RL algorithm for the LU factorization with look-ahead, malleable BLIS and ET. Two configurations are chosen for each algorithm: optimal block size for each problem size; and fixed block sizes $b_o = 192$ for LU_ET and $b_o = 256$ for LU_OS.

A comparison with other parallel versions of the LU factorization with partial pivoting is possible, but we do not expect substantial changes in our results. In particular, Intel MKL includes a highly-tuned routine for this factorization that relies in their own implementation of the BLAS and some type of look-ahead. Therefore, whether the advantages of one implementation over the other come simply from the use of a different version of the BLAS, or from the positive effects of our WS and ET mechanism, will be really difficult to infer. The PLASMA library [5] also provides a routine for the LU factorization with partial pivoting supported by a runtime that implements dynamic look-ahead. The techniques integrated in PLASMA's routine are not different from those in the OmpSs implementation evaluated in this work. Therefore, when linked with BLIS, we do not expect a different behavior between PLASMA's routine and LU_OS.

4.3.4.4 Multi-socket performance comparison with OmpSs

We conclude the experimental analysis by including a multi-socket experiment that compares different configurations of the best variant with static look-ahead, LU_ET, with the runtime approach via OmpSs, LU_OS.

In this last experiment we consider the two sockets present in the platform, using 12 threads in our tests, and report, as in the previous section, the performance for the optimal block size for each problem dimension and algorithm. The optimal values employed in this case are displayed in Figure 4.20.

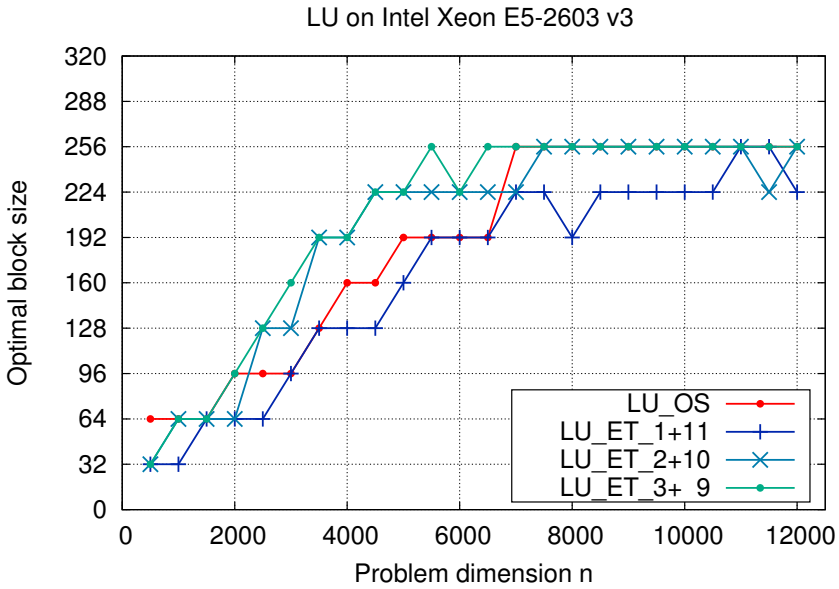

Figure 4.20: Optimal block size of the blocked ET and OmpSs algorithms for the LU factorization.

Figure 4.21 shows the results for this comparison in the lines labelled as "(b_opt)". LU_ET is very competitive, clearly outperforming the runtime-based solution for most problems and offering competitive performance for the largest five, except for the case that maps one thread in $\mathsf{T_{PF}}$ and the rest of resources in $\mathsf{T_{RU}}$.

As in the case where only one socket is employed, the performance curves are obtained for a fixed block size and the optimal block size for each problem dimension. The block size is fixed to $b_o = 256$ for all cases except for LU_ET when mapping one thread to $\mathsf{T_{PF}}$ and eleven to $\mathsf{T_{RU}}$. For LU_ET, according to Figure 4.20, the value that offers high performance for a wide range of problem

dimensions is $b_o = 224$. As in the previous study where only one socket was considered, the performance lines corresponding to the fixed block size configuration, labelled with "(b=256)"/"(b=224)", show how the ET mechanism is less affected by the use of a suboptimal block size value. Note that for the case where only one thread is in charge of $\mathsf{T_{PF}}$, the difference between the optimal block size and the fixed block size is larger than in the other cases. This behavior is due to the reduced number of threads in charge of $\mathsf{T_{PF}}$ which makes the first iteration of the factorization costly. Consequently, adjusting the block size for the next iterations is not enough to overcome the effects of the suboptimal initial block size election.

In addition, the impact of different thread mapping is shown in Figure 4.21. While in the previous section using one thread in the $\mathsf{T_{PF}}$ and the rest in the $\mathsf{T_{RU}}$ was enough, here we observe the benefits of adding more threads to $\mathsf{T_{PF}}$. Since more resources are available (2 socket vs. 1 sockets), increasing the number of threads in $\mathsf{T_{PF}}$ makes this task finish earlier and, consequently, all the threads can join faster to the execution of $\mathsf{T_{RU}}$. Interestingly, when all cores are in use, employing only one thread in $\mathsf{T_{PF}}$ harms performance, since the time spent in the execution of $\mathsf{T_{PF}}$ is increased (if compared to the other cases) and we are missing resources in $\mathsf{T_{RU}}$ for longer time.

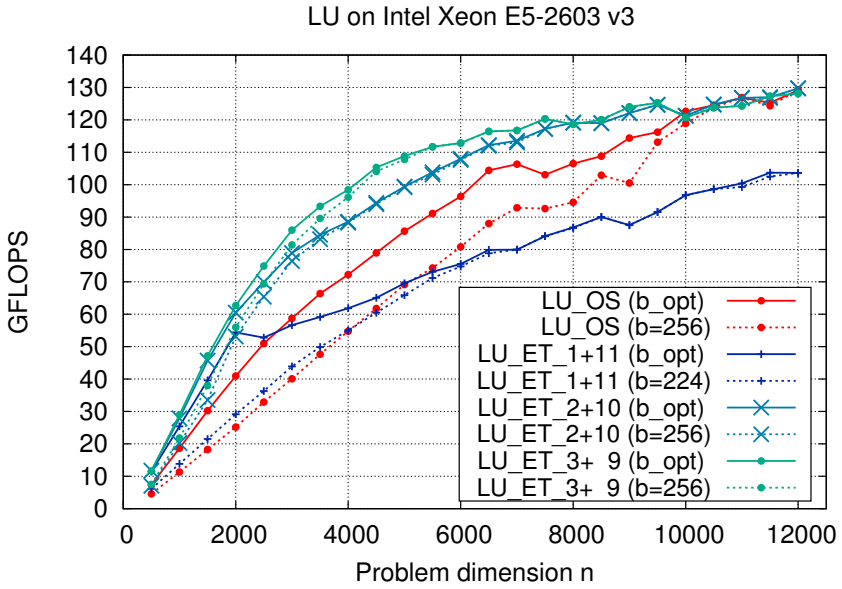

Figure 4.21: Performance comparison between the OmpSs implementation and the blocked RL algorithm for the LU factorization with look-ahead, malleable BLIS and ET. Two configurations are chosen for each algorithm: optimal block size for each problem size; and fixed block size $b_o = 256$ for all cases, except for LU_ET when mapping one thread to $\mathsf{T_{PF}}$ and eleven to $\mathsf{T_{RU}}$.

Again, a comparison with other parallel versions of the LU factorization is possible but substantial changes in our results are not expected for the same reason stated in the previous section.

4.4 Malleable LU on big.LITTLE

Following the static parallelization schemes described in Section 4.3.3 for the MTL BLAS for the Intel Xeon E5-2603 v3, in this section, we adapt and evaluate them when the target system is an AMP. Given that there exist a dynamic alternative in order to apply look-ahead (via a

runtime), we review different implementations for both cases and study their performance results. We start this section with the analysis of the static look-ahead implementations enhanced with malleability and early termination, where the operations that constitute the main building blocks are thoroughly examined; then the performance results are analyzed and compared with dynamic look-ahead versions that rely on a runtime. All the experiments in this section were performed employing IEEE double precision arithmetic and the chip frequency was set to 1.3 GHz for both Cortex-A15 and Cortex-A7 cores. Furthermore, all variants of the LU include standard partial pivoting and compute the same factorization. They also compute the factorization of the panel (kernels RL1 and PF3) via the blocked RL algorithm, except for the variants with static look-ahead, which integrate the blocked LL algorithm due to the application of ET. In the experiments we considered 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 factorization of the panel was $b_i = 32$. The value $b_i = 16$ was discarded because, as a part of an independent experiment, it was checked that using $b_i = 16$ resulted in no differences on performance when compared with $b_i = 32$.

4.4.1 Mapping of threads for the variant with static look-ahead

A first step to migrate the parallel version of LU with static look-ahead presented in Section 4.3.3 to an AMP can simply replace the calls to TRSM and GEMM with their asymmetry-aware counterparts. However, the strategy to map the cores of the Exynos 5422 to the tasks now offers a richer collection of possibilities as, in addition to the number of threads that are assigned to each task, we also need to decide the type of cores.

We note that all the configurations explored in this section are enhanced with WS, ET and MTL so that the team in charge of $\mathsf{T_{PF}}$ becomes involved in the execution of $\mathsf{T_{RU}}$ once the former task is completed and the $\mathsf{T_{PF}}$ thread team stops its execution if the $\mathsf{T_{RU}}$ team happens to finish first.

4.4.1.1 Thread mapping of GEMM

In order to analyze the effect of thread mapping in the LU factorization, four different strategies are tested in this section. In all cases, the number of threads computing the TRSM (in both $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$) is the number of big cores that are available in that task; if no big cores are assigned to a task, all available LITTLE cores will perform the TRSM. Regarding LASWP, the whole set of (slow + fast) threads will perform it, except for the small pivoting interchanges included in $\mathsf{T_{PF}}$ that are single-threaded.

Each configuration is identified using a naming scheme of the form "GEMM(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 GEMM 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 GEMM kernels appearing in $\mathsf{T_{RU}}$. Specifically, the following configurations were tested in our experiments:

- GEMM(2+0|0+4): two Cortex-A7 cores execute $\mathsf{T_{PF}}$ and four Cortex-A15 cores are mapped to $\mathsf{T_{RU}}$. With this configuration, two Cortex-A7 cores remain idle.
- GEMM(4+0|0+4): the whole Cortex-A7 cluster executes $\mathsf{T_{PF}}$ and the Cortex-A15 cluster is in charge of $\mathsf{T_{RU}}$.
- GEMM(1+1|3+3): one Cortex-A7 core and one Cortex-A15 core are mapped to $\mathsf{T_{PF}}$ while the remaining cores (three Cortex-A7 cores plus three Cortex-A15 cores) are mapped to $\mathsf{T_{RU}}$.

- GEMM(0+1|4+3): only one Cortex-A15 core is mapped to $\mathsf{T_{PF}}$; the remaining cores (four Cortex-A7 cores plus three Cortex-A15 cores) execute $\mathsf{T_{RU}}$.

For the first two parallel configurations, GEMM(2+0|0+4) and GEMM(4+0|0+4), at the beginning of each iteration, the execution employs symmetric kernels since $\mathsf{T_{PF}}$ is mapped to (either two or four) Cortex-A7 cores only, while $\mathsf{T_{RU}}$ is executed by the four Cortex-A15 cores only. Now, when $\mathsf{T_{PF}}$ is completed, the active Cortex-A7 cores become part of the team in charge of $\mathsf{T_{RU}}$, and the execution relies on asymmetric kernels from that point. With these tests we aim to verify that the optimal number of threads mapped to $\mathsf{T_{PF}}$ depends on the problem size due to the low parallelism of this task. Figure 4.22 shows that, for the smaller problems ($n < 3{,}500$), higher performance is attained by the configuration that uses two Cortex-A7 cores only, but this situation is reversed for the larger problem dimensions, for which the exploitation of the four Cortex-A7 delivers higher GFLOPS rates. The reason is that the kernels appearing in $\mathsf{T_{PF}}$ involve operands of small dimension, so that increasing the number of threads devoted to this task does not necessarily improve the performance of the global algorithm. Furthermore, when the Cortex-A7 cores are moved to $\mathsf{T_{RU}}$, the asymmetric kernels require large problem dimensions to reach its peak performance (see [28] for details).

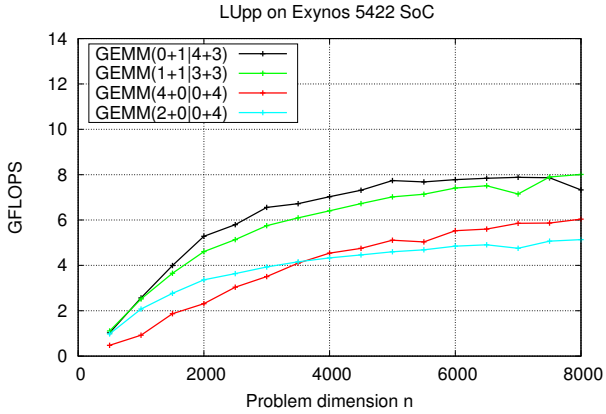

Figure 4.22: Performance of the conventional parallelization of the blocked RL algorithm with static look-ahead enhanced with MTL and WS/ET for different GEMM configurations.

The last two "task-hybrid" configurations differ from the previous configurations in the combination of a few Cortex-A7 and Cortex-A15 in at least one of the GEMM operations. In the first case, GEMM(1+1|3+3), the same number of big and LITTLE cores are used in each task, using less resources in $\mathsf{T_{PF}}$. As shown in Figure 4.22, combining two different kind of cores in both tasks increases the performance of the LU factorization by up to 40%. On the other hand, GEMM(0+1|4+3) features an uneven number of cores in the GEMM kernel. While in the previous strategy the number of Cortex-A7 and Cortex-A15 cores is the same when computing a GEMM, here only one Cortex-A15 is employed in $\mathsf{T_{PF}}$ and the whole Cortex-A7 cluster plus three Cortex-A15 cores are used. This mapping favors the execution of $\mathsf{T_{RU}}$ because more cores execute it since the beginning and, as pointed out in Section 3.3, the number of threads in execution has a great impact in the performance of GEMM.

4.4.1.2 Configuration of TRSM

From the analysis of the best thread mapping for GEMM (GEMM(0+1|4+3)) found in the previous section, now we explore three different configurations for TRSM. The main difference among these three cases is the strategy applied to parallelize the TRSM kernel:

- TRSM(L1L4;0+1|4+3): the asymmetric distribution of the workload between clusters takes place in Loop 1; then, in Loop 4, each core is assigned an even part of the workload according to the cluster it belongs to.

- TRSM(L4;0+1|4+3): the asymmetric distribution between the clusters is performed only at one level in Loop 4.

- TRSM(L4;0+1|0+3): leverages an asymmetry-oblivious implementation of TRSM that takes advantage of the Cortex-A15 cores only, yielding slightly higher performance for large matrices ($n > 6{,}000$).

Figure 4.23 reports the results for the different configurations of TRSM. Clearly, the worst option is to leverage an asymmetric distribution only in Loop 4. Curiously, although this reduces the number of buffers for A_c and B_c and diminishes the volume of cache misses, it offers lower performance. This behavior 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. However, the overhead introduced by the dynamic scheduling precludes an increase of performance by exploiting the asymmetry of the platform, 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 TRSM is configured to apply the dynamic distribution at a higher level (Loop 1), the overhead of this mechanism is significantly reduced and the performance improves considerably. Finally, we check the impact of employing only the Cortex-A15 cores to compute TRSM. In view of the results, 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 TRSM; in the best situation TRSM operates on a triangular matrix of dimension 256×256 making the addition of extra resources worthless.

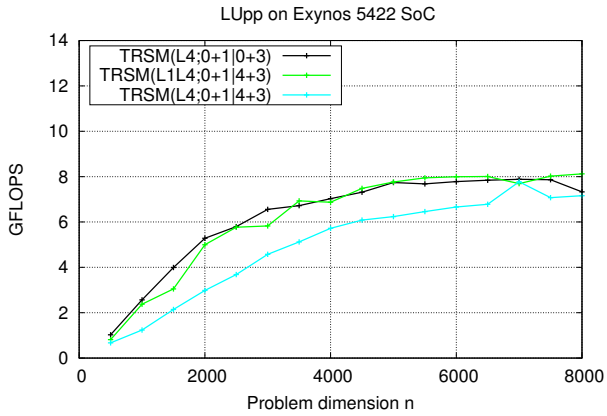

Figure 4.23: Performance of the conventional parallelization of the blocked RL algorithm with static look-ahead enhanced with MTL and WS/ET for different TRSM configurations.

4.4.1.3 The relevance of the partial pivoting

So far, the analysis of the LASWP routine was not considered because TRSM and GEMM are the most costly building blocks of the LU factorization with partial pivoting. However, the reorganization of the code in order to apply look-ahead precludes the overlap of all the row permutations in $\mathsf{T_{PF}}$ with the $\mathsf{T_{RU}}$ tasks due to data dependencies. For this reason, a deeper analysis on the impact of the parallelization of this operation is performed. To this end, we adopt the best configurations found in the previous sections (GEMM(0+1|4+3)) and (TRSM(L4; 0+1|0+3)), that is the one that employs the Cortex-A15 cluster when performing TRSM and a combination of Cortex-A7 and Cortex-A15 cores for GEMM (note that in the $\mathsf{T_{PF}}$ task only one thread is used, but seven are available for $\mathsf{T_{RU}}$).

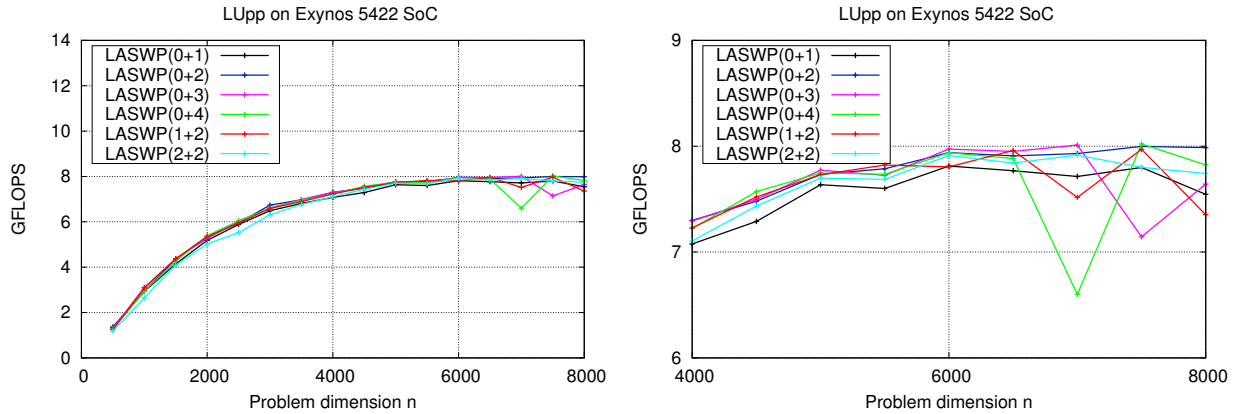

Figure 4.24: Performance of the conventional parallelization of the blocked RL algorithm with different LASWP parallelizations and static look-ahead enhanced with MTL and WS/ET.

In Figure 4.24 (left) we report the performance when changing the parallelization degree and the mapping of the threads that execute the pivoting. These results are presented for different combinations of Cortex-A15 and Cortex-A7 cores. In view of them, we can conclude that, for small matrices, those parallelizations that only use Cortex-A15 cores in general provide better results. This is not surprising since the workload is too small to take advantage of the Cortex-A7 cores. On the other hand, for large size matrices it is hard to determine if there is any difference in performance. When zooming in the results (right-hand side plot in Figure 4.24) although we appreciate that adding one or more Cortex-A7 cores reduces performance. Therefore, we conclude that, in order to attain high performance, the best option when parallalizing LASWP should employ two Cortex-A15 cores, yielding an improvement that is on average around 7.7%.

4.4.2 Dynamic look-ahead with OmpSs

There are different approaches that rely on a runtime in order to introduce a dynamic look-ahead in the implementation of the LU factorization. Concretely, in a conventional runtime-based approach with task priorities (i.e., OmpSs), a dynamic look-ahead is orchestrated by the runtime which is in charge of parallelizing the algorithm. A second option is the use of an asymmetry-aware BLAS library in combination with a traditional runtime [36]. This choice exposes the AMP as a conventional symmetric multi-core processor to the OmpSs runtime, hiding the asymmetry inside 4 symmetric virtual cores (VCs), composed each of a single Cortex-A15 core plus a single Cortex-A7

core. An alternative that we also explored relies on an asymmetry-aware version of OmpSs called Botlev [35].

Figure 4.25 illustrates the performance of the runtime-based parallel configurations of LU described in the previous paragraph:

- LU_OS_VC: OmpSs parallelization of the basic algorithm with asymmetry-oblivious version of the runtime, linked with the asymmetry-aware implementation of TRSM and GEMM, and executed using all cores of the Exynos 5422 SoC grouped into 4 VCs.
- LU_OS_BO: OmpSs parallelization of the basic algorithm, parallelized using the Botlev asymmetry-aware version of the runtime, linked with the sequential implementation of TRSM and GEMM, and executed using all cores of the Exynos 5422 SoC.
- LU_OS: OmpSs parallelization of the basic algorithm with an asymmetry-oblivious runtime, linked with the sequential implementation of TRSM and GEMM, and executed using all cores of the Exynos 5422 SoC.

As expected, this experiment reveals that considerably lower performance rates are obtained when no attempt is made to take into account the asymmetry of the architecture. This corresponds to the configuration LU_OS, which combines an asymmetry-oblivious OmpSs scheduler with implementations of the GEMM and TRSM kernels that are not aware of the asymmetry. In contrast, the highest performance rates are observed for the configuration that views the Exynos 5422 SoC as 4 VCs, using asymmetry-aware versions of the GEMM and TRSM kernels, but an asymmetry-oblivious scheduler. The opposite solution, combining an asymmetry-aware scheduler with asymmetry-unconscious kernels, delivers a GFLOPS rate that is in between the other two. These results agree with the observations made for the Cholesky factorization in [36].

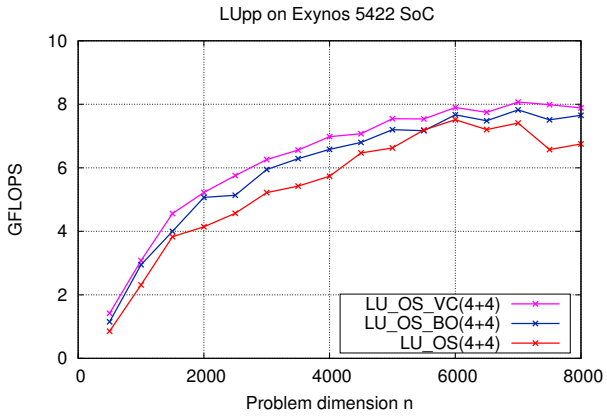

Figure 4.25: Performance of the OmpSs parallelization of the blocked RL algorithm for LU (dynamic look-ahead).

4.4.3 Global comparison

Figure 4.26 compares the best parallel configuration for each of the versions of the LU factorization evaluated in this section:

- LU_AS: blocked RL algorithm for LU (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: LU with look-ahead with the optimal configurations of the basic kernels determined in subsection 4.4.1.1: GEMM(0+1|4+3), TRSM(L4;0+1|0+3), and two Cortex-A15 for the execution of LASWP.
- LU_OS_VC: OmpSs parallelization of the basic algorithm with asymmetry-oblivious version of the runtime, linked with the asymmetry-aware implementation of TRSM and GEMM, and executed using all cores of the Exynos 5422 SoC grouped into 4 VCs.

In this plot it can observed 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 MTL and WS/ET, yielding an implementation that consistently delivers up to 2 GFLOPS more than the conventional algorithm for LU. Finally, the runtime-based implementation, which applies a dymamic look-ahead strategy, proves that the deeper look-ahead is especially beneficial for small problems ($n < 2{,}000$). For mid-large problem dimensions though, our implementation matches or slightly outperforms the runtime-based approach.

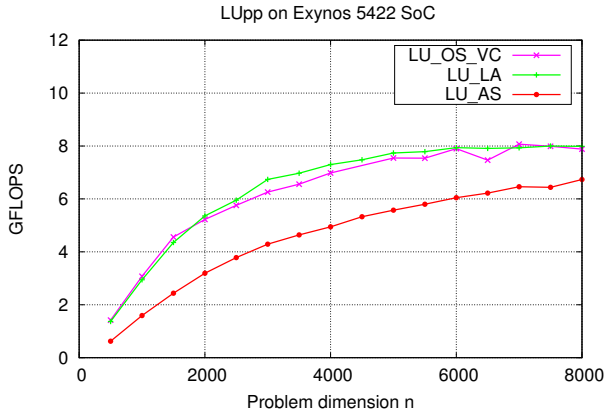

Figure 4.26: Performance of the best parallel configuration for each variant: conventional blocked RL algorithm, static look-ahead version enhanced with MTL and WS/ET, and runtime-based implementation.

4.5 Summary

In this chapter, 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 two major factorization from LAPACK illustrates two 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.

To tackle the low performance attained by the LU factorization, we have introduced WS and ET as two novel techniques to avoid workload imbalance during the execution of matrix factorizations, enhanced with look-ahead, for the solution of linear systems. The WS mechanism especially benefits from the design of a MTL instance of BLIS, which allows the thread team in charge of the panel

factorization, upon completion of this task, to be reallocated to the execution of the trailing update. The ET mechanism tackles the opposite situation, with a panel factorization that is costlier than the trailing update. In such scenario, the team that performed the update communicates to the second team that it should terminate the panel factorization, advancing the factorization process into the next iteration.

Our results on an Intel Xeon E5-2603 v3 showed the performance benefits of our version enhanced with WS+ET and malleable BLIS compared with a plain LU factorization as well as a version with look-ahead. The experiments also reported competitive performance compared with an LU factorization that is parallelized by means of a sophisticated runtime, such as OmpSs, that introduces look-ahead of dynamic (variable) depth. Compared with the OmpSs solution, our approach offers higher performance for most problem dimensions, seamlessly tunes the algorithmic block size, and features a considerably smaller memory footprint as it does not require a sophisticated runtime support.

Finally, we applied our WS+ET solution to the LU factorization on an Exynos 5422 SoC equipped with an ARM Cortex-A7/Cortex-A15 multi-core processor in order to analyze the potential benefits, already observed for the Intel Xeon platform, that can be obtained on an AMP. In this case, we compared our strategy that applies a static look-ahead with the runtime-based approach (via OmpSs) that integrates a dynamic look-ahead. The experimental results show that the use of a thread-level malleable library, combined with the WS+ET strategies, outperforms the runtime-based implementation for small and medium matrices and matches the performance for large problem dimensions. The reason for this behavior lies in that, compared with the classical blocked algorithm, the static look-ahead partially removes the scalability bottleneck imposed by the panel factorization. In addition, compared with the runtime solution, the static look-ahead produces a more cache-friendly execution.

To conclude, this work does not intend to propose an alternative to runtime-based solutions. Instead, the message implicitly carried in these experiments aims to emphasize the benefits of malleable thread-level libraries, which we expect to be crucial in order to exploit the massive thread parallelism of future architectures. This work opens a plethora of interesting questions for future research. In particular, how to generalize the ideas to a multi-task scenario, what kind of interfaces may ease thread-level malleability, and which kind of support is necessary in the runtime for this purpose.

CHAPTER 5

Reduction to Condensed Forms

In this chapter we extend the study of LAPACK routines in combination with the asymmetry-aware implementation of BLIS in order to complete the study about the performance benefits when targeting an AMP. To this end, we analyze three routines that perform two-sided orthogonal reductions (TSOR) of a dense matrix to a condensed form. Efficient and numerically reliable algorithms for the computation of the eigenvalues/singular values of a dense matrix consist of two stages [56]; first, the matrix is transformed to a condensed matrix through a TSOR and then a specific solver is applied to the condensed matrix in order to accurately compute the eigenvalues/singular values.

We study the three main routines provided by LAPACK [11] for TSOR to distinct condensed forms:

- SYTRD [75] reduces a symmetric matrix to tridiagonal form via similarity (i.e., eigenvalue-preserving) transformations;
- GEBRD [55] transforms a general matrix to bidiagonal form; and
- GEHRD [76] reduces a general matrix to Hessenberg form via similarity transformations.

The first and third routines are applied as an initial stage to compute the eigenvalues of a square matrix, while the second routine is the first step for the computation of the singular values of a non-necessarily square matrix.

In the following sections, we first describe the operations for the TSOR. Next, we present several optimizations to these procedures specifically designed for an ARM big.LITTLE AMP. Then, we present a detailed experimental analysis to illustrate the performance benefits of our architecture- and asymmetry-aware variants of SYTRD, GEBRD and GEHRD on the big.LITTLE architecture. Finally, we propose a performance model that guides the selection of the optimal algorithmic block size and core configuration for the TSOR stage.

5.1 General Structure of the Reduction to Condensed Forms

Given a square matrix A, of order n, the associated eigenvalue problem is formally defined by

$$AX = X\Lambda, \tag{5.1}$$

where the $n \times n$ diagonal matrix $\Lambda = \text{diag}(\lambda_1,\lambda_2,\ldots,\lambda_n)$ contains the eigenvalues of A, and the columns of the $n \times n$ matrix X contain the corresponding eigenvectors [56]. On the other hand, the singular value decomposition (SVD) of an $m \times n$ matrix A is defined as

$$A = U\Sigma V^T, \tag{5.2}$$

where $\Sigma = \text{diag}(\sigma_1,\sigma_2,\ldots,\sigma_r)$ is a square matrix of order $r = \min(m,n)$ that contains the singular values of A in decreasing order of magnitude (i.e., $\sigma_i \geq \sigma_{i+1}$); and U,V^T, of respective dimensions $m \times r$ and $r \times n$, are orthogonal and their columns comprise the left and right singular vectors of the matrix.

The routines in LAPACK for the solution of (symmetric and general) eigenproblems as well as the computation of the singular values tackle dense instances of these problems by first reducing A to a condensed matrix C, of dimension $m \times n$ (with $m = n$ for eigenproblems), via a collection of Householder (orthogonal) reflectors [56]. For performance reasons, at each iteration of these TSOR procedures (usually known as blocked algorithms), several orthogonal reflectors are aggregated into a single block reflector, which is then applied via calls to efficient Level-3 BLAS. It is important to note that there exists a multi-stage approach that performs the TSOR in two or more steps, by first reducing A to a band matrix and then successively refining this to the sought-after condensed form [19]. Nevertheless, this alternative approach will not be considered as it often requires a higher number of flops. We next describe the processes in some detail.

Let us denote the algorithmic block size as b and, for simplicity, assume hereafter that m, n are both integer multiples of b. Consider that we have progressed up to an iteration $j \in \{1,2,\ldots,\min(m,n)/b\}$, applying the necessary transformations (from the left and right) to the matrix in order to obtain:

$$A \quad \rightarrow \quad \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & A_{11} & A_{12} \\ \hline 0 & A_{21} & A_{22} \end{array}\right),$$

where C_{00} is $(j-1)b \times (j-1)b$; A_{11} is $b \times b$; and the blocks C_{00}, C_{10}, C_{01} (and C_{02}), contain the corresponding entries of the sought-after condensed form C. The following operations are then computed during the current iteration of the TSOR routines SYTRD, GEHRD and GEBRD:

1. PANEL FACTORIZATION (PF): The "current" column-panel $\left(\frac{A_{11}}{A_{21}}\right)$ and row-panel $(A_{11} \mid A_{12})$ are reduced to the target condensed form using a sequence of orthogonal transformations. Simultaneously, these transformations are aggregated in the form of matrices V, X, both of dimension $m - jb \times b$, and U, Y, both of size $n - jb \times b$, such that the application of these transformations yields

$$\left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline 0 & C_{21} & A_{22} - VY^T - XU^T \end{array}\right),$$

   implying that, upon completion of this operation, the computation of the condensed form has progressed by b columns/rows.

2. TRAILING UPDATE (TU): The submatrix A_{22} is updated as $A_{22} := A_{22} - VY^T - XU^T$.

This generic TSOR procedure implements a blocked algorithm that processes the $m \times n$ matrix A, from top-left to bottom-right, in blocks of b-column/row panels starting at columns/rows $\hat{\jmath} =$

$(j-1)b = 0, b, 2b, \ldots$. The bulk of the computation in PF corresponds to the formation of matrices X, Y (U, V are obtained as part of the panel factorization). In particular, for each reduced column in the panel, this may require several matrix-vector multiplications.

This generic TSOR procedure requires some specialization depending on the type of condensed form to be computed:

- For SYTRD, $m = n$, A is symmetric, $X = Y$, $U = V$ and, in order to exploit the symmetry, only the lower (or upper) half of A_{22} is updated in TU. Note that only in this reduction C_{02}=0, given that A is symmetric.
- The reduction to bidiagonal form via GEBRD can be re-organized to reduce the computational cost in case $m \gg n$ (or vice-versa), but the previous procedure is the preferred choice if $m \approx n$.
- The reduction to Hessenberg form via GEHRD slightly differs from the generic TSOR procedure in the specific blocks that are updated (and annihilated) at each iteration [86].

5.2 General Optimization of the TSOR Routines

There are four optimization keys that have to be addressed to ensure high performance for the execution of the TSOR routines on the target AMP:

- Development of tuned micro-kernels for the Level-2 and Level-3 BLAS and each type of core.
- Asymmetry-aware parallelization of the Level-2 and Level-3 BLAS.
- Selection of the algorithmic block size for the TSOR procedure.
- Configuration of the number and type of cores to utilize for each type of Level-2 and Level-3 BLAS kernel invoked from the TSOR procedures.

The first two factors are tackled through the use of our asymmetry-aware implementation of BLIS, presented in Chapter 3. Therefore, this chapter focuses on performing a general evaluation of the impact of the last two factors on performance. The experiments in the following sections were performed with the Cortex-A7 cores operating at 1.4 GHz and the Cortex-A15 at 1.5 GHz, using real single-precision IEEE arithmetic. In addition, all our experiments employ the sequential Level-1 kernels from BLIS (version 0.1.8), in combination with the multi-threaded asymmetry-aware instances of the Level-3 and Level-2 kernels. Cache and register configuration parameters are set to the values reported in Tables 5.1 and 5.2, respectively.

	m_r	n_r	m_c	k_c	n_c
ARM Cortex-A15	4	4	400	368	4,096
ARM Cortex-A7	4	4	88	368	4,096

Table 5.1: Parameters for optimal performance of the Level-3 kernels in BLIS on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC using real single-precision IEEE arithmetic.

In general, the algorithmic block size b selected for the TSOR routines has an important performance effect. This impact will be analyzed in detail for each TSOR routine included in this chapter. In addition, a general analysis about the relevance of selecting the most appropriate core configuration is presented in this section.

	n_r	m_c	n_c
ARM Cortex-A15	4	832	2,560
ARM Cortex-A7	4	144	2,560

Table 5.2: Parameters for optimal performance of the Level-2 kernels in BLIS on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC using real single-precision IEEE arithmetic.

5.2.1 Selection of the core configuration

A complementary factor that dictates the performance of the TSOR procedures, when executed on an AMP, is the number/type of cores (configuration) that are employed for the execution of each building block. Here we consider the BLAS-2 kernels (GEMV and SYMV) and BLAS-3 kernels (GEMM and SYR2K) that appear in the TSOR procedures as building blocks. Although, the TRMM and TRMV kernels are also found in some of the TSOR procedures tackled in this chapter, we do not consider them due to the short time that is spent on them during the execution of the reductions. In this section, we present the performance for each building block when using different core configurations, namely 4 Cortex-A7 cores, 1 Cortex-A15 core, 1 Cortex-A15 core + 4 Cortex-A7 cores (only for BLAS-2 routines) and 4 Cortex-A15 cores + 4 Cortex-A7 cores (only for BLAS-3 routines). In addition, we perform our tests for two different block sizes in order to analyze the effect of this parameter.

The two plots in the top row of Figure 5.1 report the performance rate attained by GEMV, using matrix operands of two practical shapes encountered in the TSOR routines. These two graphs reveal that the threshold dimension from which it is more convenient to use a Cortex-A15 core plus the full Cortex-A7 cluster depends on the iteration step (m-dimension) and the algorithmic block size (n-dimension). For small block sizes, large values in the m-dimension favor the use of the Cortex-A15 core plus the full Cortex-A7 cluster. In contrast, for large block sizes, small values in the m-dimension are to be preferred. In addition, the block size dictates the highest sustainable performance observed for GEMV. For small block sizes, this kernel attains 2.5 GFLOPS due to data re-use in the caches, but this value decreases to only 2 GFLOPS for large block sizes.

The four plots in the bottom two rows of Figure 5.1 show the results for an analogous experiment using the Level-3 BLAS routines and two matrix shapes that appear during the TSOR routines. The conclusions inferred from this analysis of the Level-3 BLAS are similar to those presented for GEMV. In summary, the point from which it is more beneficial to use the entire SoC or the Cortex-A15 cluster only depends on the iteration step and the block size. However, for the Level-3 BLAS, large block sizes tend to render higher performance, as they allow to select closer-to-optimal loop strides while extracting an ampler level of concurrency within the kernels.

To complete the analysis of the main building blocks present in the TSOR routines, Figure 5.2 shows the performance of the SYMV routine. In contrast with the previous kernels, an optimal configuration of this building block always exploits a Cortex-A15 core plus the full Cortex-A7 cluster.

In summary, the algorithmic block size directly affects the shapes of the matrix operands passed to the BLAS kernels invoked from the TSOR procedures. Furthermore, the experiments in this section illustrate that the block size changes the threshold from which it is more beneficial to use a certain configuration for the execution of a certain building block (kernel). Therefore, the effect of the block size has to be analyzed simultaneously with the core configuration.

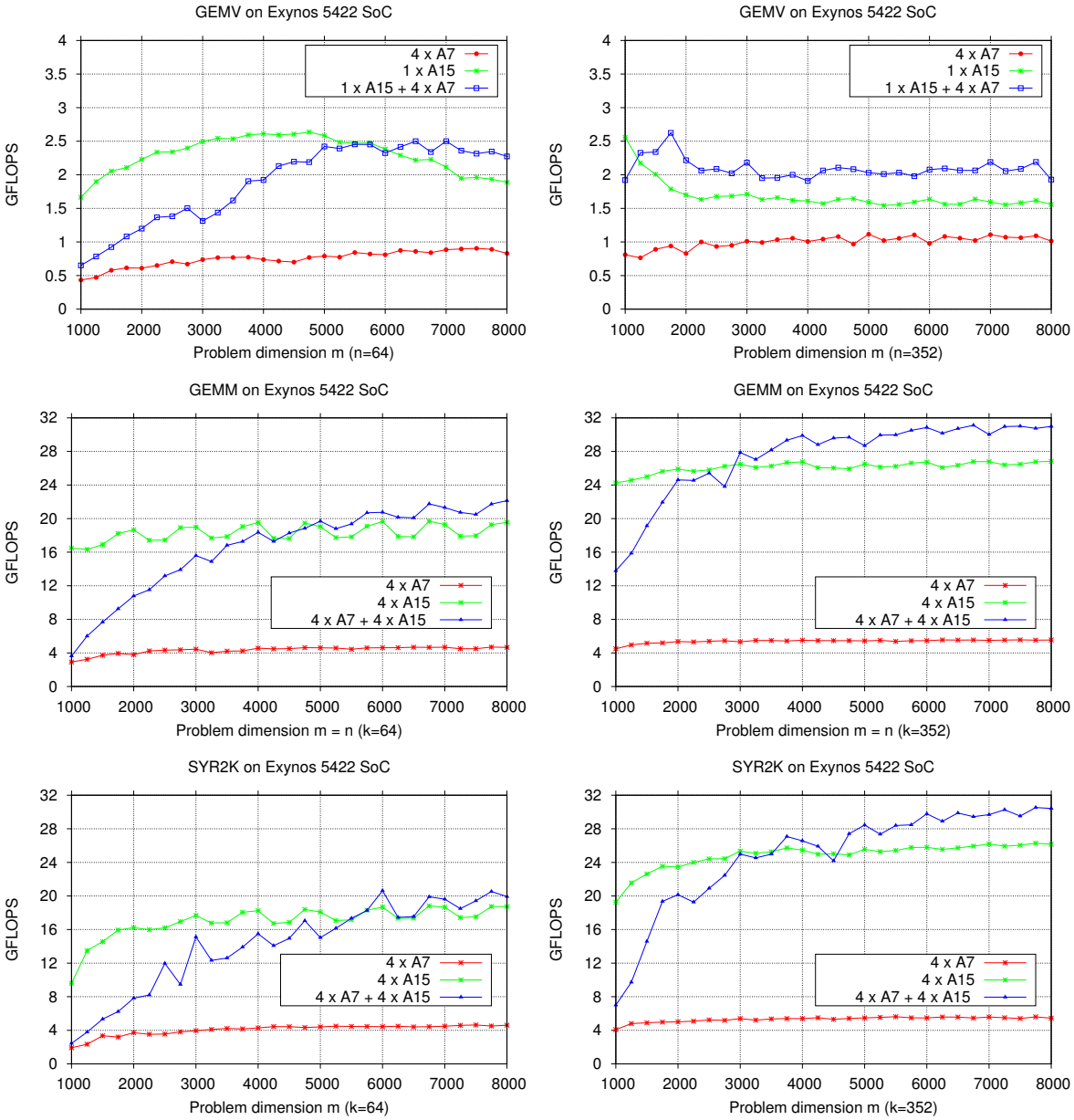

Figure 5.1: Performance of Level-2 and Level-3 BLAS on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC.

To close this section, we point out that employing a parallel configuration for the execution of small-size matrix-vector products is counterproductive. To avoid this negative effect, we designed an adaptive strategy that, according to problem dimension and block size, selects the best core configuration at runtime. This strategy ensures that the performance of the asymmetry-aware configuration matches that attained with the Cortex-A15 cluster for small- to medium-size problems. Compared with that, for larger problems, our asymmetry-aware algorithms add the Cortex-A7 cluster to the computation, raising the GFLOPS rate by a factor that is close to 30%.

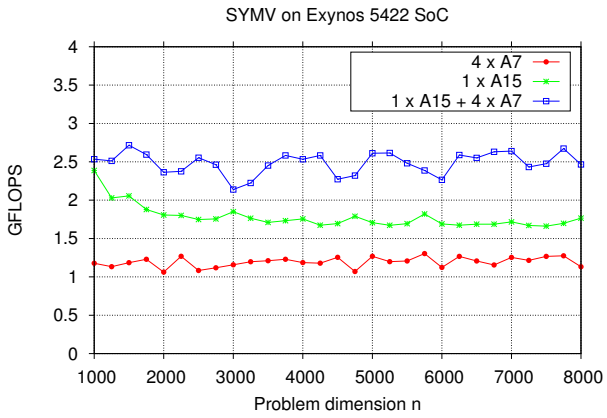

Figure 5.2: Performance of Level-2 and Level-3 BLAS on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC

5.3 Reduction to Tridiagonal Form (SYTRD)

Given a dense matrix $A \in \mathbb{R}^{n \times n}$, its *reduction to tridiagonal form* T *by an orthogonal similarity transformation* is given by $T = Q^T A Q$. Listing 5.1 displays a simplified C code that computes this type of reduction for of an `n`×`n` matrix stored starting at address `A` with column leading dimension `Alda`. For simplicity, hereafter we assume that the matrix size is an integer multiple of the block size b. The code overwrites the corresponding entries of the original matrix with the corresponding elements of the tridiagonal matrix T, leveraging the numerical *kernels* (or building blocks) LATRD, SYR2K and SYTD2 for this purpose. The LATRD subroutine from LAPACK reduces the first b rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal similarity transformation; the main BLAS-2 building blocks of this routine are GEMV and SYMV. Routine SYTD2 implements an unblocked algorithm that reduces a symmetric matrix to real symmetric tridiagonal form by orthogonal similarity transformations; it relies on the BLAS-2 SYMV kernel.

```
#define A_ref(i,j) A[(j)*Alda+(i)]

for (k=0; k<n; k+=b) {

  // Reduce current diagonal block
  LATRD( ..., &A_ref( k, k ), Alda, work, ldwork );

  // Update trailing submatrix
  SYR2K( ..., &A_ref( k+b, k ), Alda, work( b+1 ), ldwork, &A_ref( k+b, k+b ), Alda );

}
// Reduce the rest of the matrix
SYTD2( ..., &A_ref( k, k ), Alda, &info);
```

Listing 5.1: Blocked routine for the reduction to tridiagonal form.

The overall cost of routine SYTRD is $4n^3/3$ flops, with $2n^3/3$ flops performed via calls to SYR2K, and the rest corresponding to the Level-2 BLAS GEMV and SYMV.

5.3.1 The impact of the algorithmic block size in SYTRD

In order to analyze the effect of the algorithmic block size on SYTRD, we perform an initial test using different block sizes b for distinct problem dimensions. Figure 5.3 reports the results in GFLOPS when the experiment is run using a single Cortex-A15 core. Those results show a performance gap between the lowest and highest GFLOPS rates of about 0.5 GFLOPS for the smallest problem size. However, for the largest problem the difference is around 0.25 GFLOPS. With this experiment we expose that the block size exerts a relevant impact on performance along the problem size range.

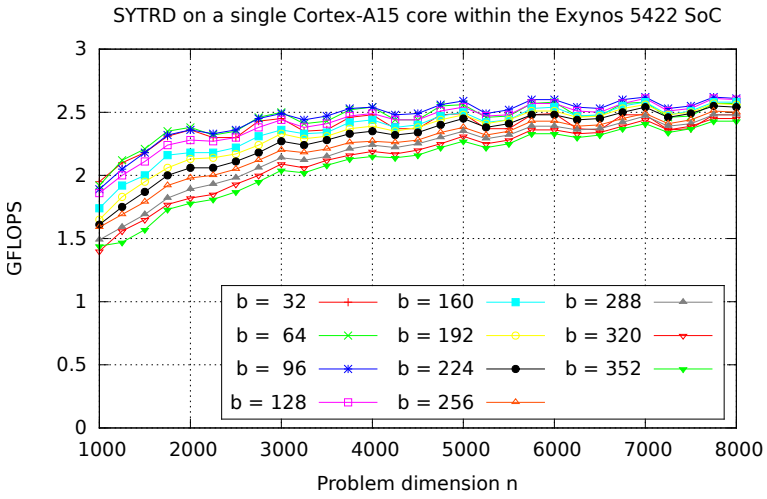

Figure 5.3: Performance of SYTRD on a single Cortex-A15 core within the ARM big.LITTLE AMP embedded in the Exynos 5422 SoC using different algorithmic block sizes.

To better understand the role of the block size b, Figure 5.4 profiles the influence of this parameter on the distinct building blocks appearing in SYTRD routine. As reported in the figure, the symmetric matrix-vector product (SYMV, green lines) accounts for a major part of the global execution time, with this fraction of the practical cost growing with the problem size. This implies that any optimization of this particular kernel, via either an architecture-aware implementation or an asymmetry-aware parallelization, can be expected to yield important gains on the performance of the reduction routine. In addition, the execution time of SYMV is basically independent of the block size. Therefore, the optimization of this parameter for SYTRD can be pursued by taking into account only the other two components of the reduction, namely GEMV and SYR2K.

The execution time of the general matrix-vector products (carried out via GEMV, dark blue lines) grows with the block size, while that of the symmetric rank-$2k$ update (SYR2K, red lines) has the opposite behavior. The reason for these opposite trends lies in that an increase of the block size shifts part of the computational cost of the reduction (in the order of n^2b flops) from the symmetric rank-$2k$ update to the general matrix-vector product. This has a minor impact on the theoretical cost/execution time of the SYR2K kernel, as the volume of computations performed in terms of this type of operations is $2n^3/3$ flops; indeed, the reduction in the execution time of this component is basically due to the use of a larger block size, which delivers a higher GFLOPS rate. However, increasing the amount of flops that are cast in terms of GEMV has a major effect on the practical cost of GEMV, as this is a memory-bound operation that proceeds at a much lower GFLOPS rate. In other words, although increasing b produces a small raise in the amount of flops that are cast in

terms of GEMV (when compared to the total flops of the reduction routine), the practical cost (i.e., execution time) becomes much larger due to the low performance of this kernel.

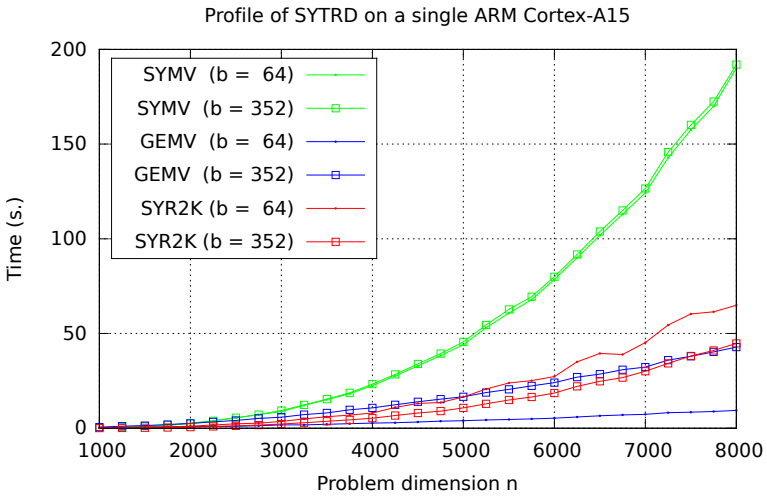

Figure 5.4: Profile of execution time spent by SYTRD on a single ARM Cortex-A15 core embedded in the Exynos 5422 SoC. This experiment sets the block size to either 64 and 352 for comparison.

5.3.2 Performance of the optimized SYTRD

This section shows the performance benefits of a tuned selection of the block size and core configuration, together with the integration of architecture-aware micro-kernels and asymmetry-aware parallel version of the building blocks for SYTRD.

During the execution of the routine, we dynamically adjust the number/type of cores independently for the main building blocks in order to tune the performance depending on the dimensions of the operands that are involved in each call to a building block. This dynamic optimization for SYTRD was applied to GEMV and SYR2K. Note that, for SYMV, no dynamic optimization is necessary, since using the Cortex-A7 cluster and one Cortex-A15 core is always the best option (see Section 5.2.1).

Figure 5.5 illustrates the performance of the SYTRD routine, using the model-driven optimal algorithmic block size ($b = 64$) that will be presented in Section 5.6. The architecture-aware micro-kernels for the Level-2 BLAS kernels and the asymmetry-aware parallelizations of the Level-2/3 kernels correspond to the implementations presented in Chapter 3. The plots include a configuration with no optimizations applied to the Level-2 BLAS (labeled as "Initial") as well as one where all optimizations are present (labeled as "Asymmetry-aware"). Additionally, for comparison purposes, the plot includes four additional reference configurations:

- 4 x A15: execution on the Cortex-A15 cluster only (4 threads).
- 4 x A7: execution on the Cortex-A7 cluster only (4 threads).
- Ideal - 4: theoretical performance rate obtained by adding the GFLOPS rates of the isolated Cortex-A15 cluster and the isolated Cortex-A7 cluster.

- Ideal - 1: theoretical performance rate obtained by adding the GFLOPS rate of a single Cortex-A15 core multiplied by 4 (number of cores in the cluster) plus that of a single Cortex-A7 multiplied by 4.

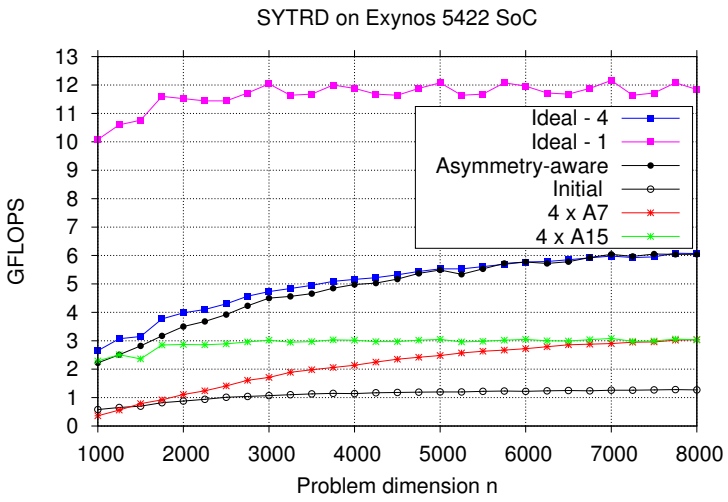

Figure 5.5: Performance of SYTRD on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC. This experiment sets the block size to $b = 64$.

Let us discuss in some detail the results in Figure 5.5. The use of the Cortex-A15 cluster only shows a performance rate that is almost flat, close to 3 GFLOPS. The reason is that the Level-2 BLAS, dominates the execution time of the routine, and adding more than one Cortex-A15 core does not contribute any performance benefit. In contrast, the trend observed for the line that employs the Cortex-A7 cluster only shows an asymptotic performance that is close to that of the Cortex-A15 cluster, as the Level-2 BLAS do scale with the problem dimension for this type of cores. These results can be related back to the analysis of the building blocks SYMV, GEMV and SYR2K in Section 5.2.

The asymmetry-aware configuration shows a consistent performance advantage over its homogeneous (i.e., symmetric or single-cluster) counterparts as the former takes advantage of the computational power of the Cortex-A15 cores for the execution of the Level-3 BLAS SYR2K and the scalability of Level-2 BLAS SYMV/GEMV on the Cortex-A7 cluster. Concretely, for the largest problem size, the speed-up of this solution grows to be above 2× with respect to the execution using any of the two clusters in isolation. In addition, the asymmetry-aware configuration benefits from the multi-threaded Level-2 BLAS and the optimized micro-kernels to deliver a performance rate that is up to 6× higher than the initial configuration. Focusing on the ideal (theoretical) configurations, our solution attains a performance rate that lies close to that of the Ideal-4 case, showing a fair distribution of the workload between the two clusters and no significant performance leaks. A less pleasant scenario appears in the comparison against the Ideal-1 case. This is explained by the actual lack of scalability of the Level-2 BLAS when executed on the Cortex-A15 (Chapter 3), in contrast with the unrealistic assumption of perfect scalability for the Ideal-1 line.

The major cause for the acceleration of SYTRD, which allows to narrow the gap between the performances of the asymmetry-aware configuration and Ideal-4, is the large amount of symmetric matrix-vector products (SYMV) which are, in turn, large and independent of the block size. Large matrix-vector products result in appealing opportunities to exploit a parallel configuration.

5.4 Reduction to Bidiagonal Form (GEBRD)

Given a dense matrix $A \in \mathbb{R}^{m \times n}$, its *reduction to bidiagonal form* B *by an orthogonal similarity transformation* is given by $B = Q^T AP$. Listing 5.2 displays a simplified C code that computes this type of reduction for an `m`×`n` matrix stored starting at address `A` with column leading dimension `Alda`. For simplicity, we assume hereafter that the matrix size is $m = n$ and this parameter is an integer multiple of the block size `b`. Furthermore we assume that the code overwrites the corresponding entries of the original matrix with those of the upper bidiagonal matrix B. For this purpose, it leverages the numerical *kernels* (or building blocks) LABRD, GEMM and GEBD2. Routine LABRD from LAPACK reduces the first b rows and columns of a general matrix to a bidiagonal form using the BLAS-2 GEMV kernel. The GEBD2 kernel reduces a general matrix to bidiagonal form using an unblocked algorithm.

```
#define A_ref(i,j) A[(j)*Alda+(i)]

for (k=0; k<n; k+=b) {

  // Reduce current diagonal block
  LABRD( ..., b, &A_ref(k,k), Alda, work(n*b), n );

    // Update trailing submatrix
    GEMM( ..., &A_ref( k+b, k ), Alda, work(n*b+b), n, &A_ref( k+b, k+b ), Alda );
    GEMM( ..., work(b), n, &A_ref( k, k+b ), Alda, &A_ref( k+b, k+b ), Alda );

}
// Reduce the rest of the matrix
GEBD2( ..., &A_ref( k, k ), Alda, work, &info);
```

Listing 5.2: Blocked routine for the reduction to bidiagonal form.

For GEBRD, the effect of the Level-2 BLAS is more prominent than it was in SYTRD. The total cost of this reduction, $8n^3/3$ flops, is split into $2n^3/3$ flops performed in terms of the Level-3 BLAS for the general matrix-matrix multiplication (GEMM) and $2n^3$ flops as calls to GEMV (invoked from LABRD and GEBD2 routines).

5.4.1 The impact of the algorithmic block size in the Reduction to Bidiagonal Form (GEBRD)

Following the same idea as for SYTRD, in this section we study the configuration of the type and number of cores that execute the main building blocks of GEBRD in order to test the potential performance increase. In order to analyze the effect of varying the block size, we perform an initial experiment with different algorithmic block sizes and problem dimensions. The results of this test in GFLOPS are reported in Figure 5.7. As for the SYTRD case, the difference between the highest and lowest GFLOPS rates for the smallest problem size is around 0.5 GFLOPS. However, for the largest problem, the difference is smaller than for SYTRD in this case being around 0.17 GFLOPS.

To gain a deeper insight on the behavior of GEBRD depending on the block size b, we obtain a profile of the main building blocks that compose GEBRD. The profile is shown in Figure 5.7 and proves that the execution time of this reduction is clearly split into two components: the general matrix-vector products basically found in PF (GEMV, dark blue lines), and (two) general matrix-matrix multiplications for TU (GEMM, light blue lines). Increasing the block size here shifts part of the flops from TU to PF, with an effect on performance similar to that already discussed for SYTRD.

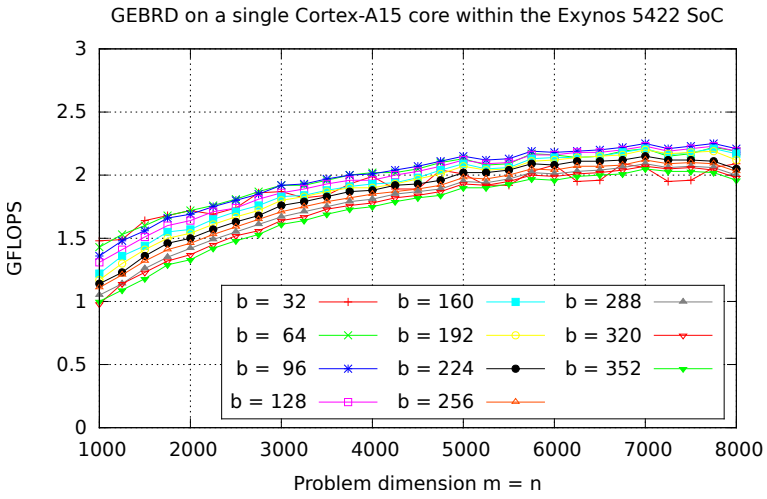

Figure 5.6: Performance of GEBRD on a single Cortex-A15 core within the ARM big.LITTLE AMP embedded in the Exynos 5422 SoC using different algorithmic block sizes.

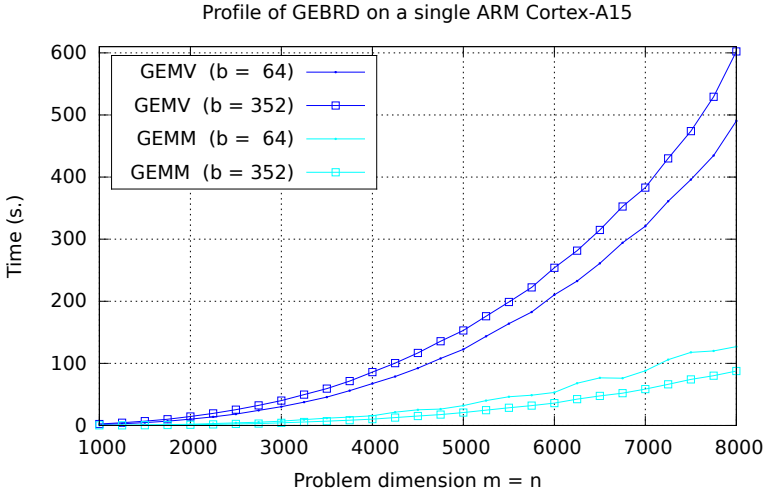

Figure 5.7: Profile of execution time spent by GEBRD on a single ARM Cortex-A15 core embedded in the Exynos 5422 SoC. This experiment sets the block size to 64 and 352, respectively, for comparison.

5.4.2 Performance of the optimized GEBRD

We next build upon the architecture-aware micro-kernels and asymmetry-aware parallel version of the building blocks of GEBRD presented in Chapter 3, in order to study the effect of selecting an optimal block size and core configuration. As already done for SYTRD, we dynamically adjust the number and type of cores for the main kernels of this reduction, namely GEMV and GEMM.

Figure 5.8 reports the performance of GEBRD ($b = 96$). The "Initial"line refers to a configuration with no optimizations applied to BLAS-2, while the "Asymmetry-aware" line integrates all the proposed optimizations. Again, for comparison purposes, we include the performance for the

Cortex-A15 cluster (4 x A15) and Cortex-A7 cluster (4 x A7) in isolation, the theoretical GFLOPS rate as the addition of the performance of both clusters in isolation (Ideal - 4) and the theoretical performance rate as the addition of the performance of a single core of each type multiplied by 4 (Ideal - 1).

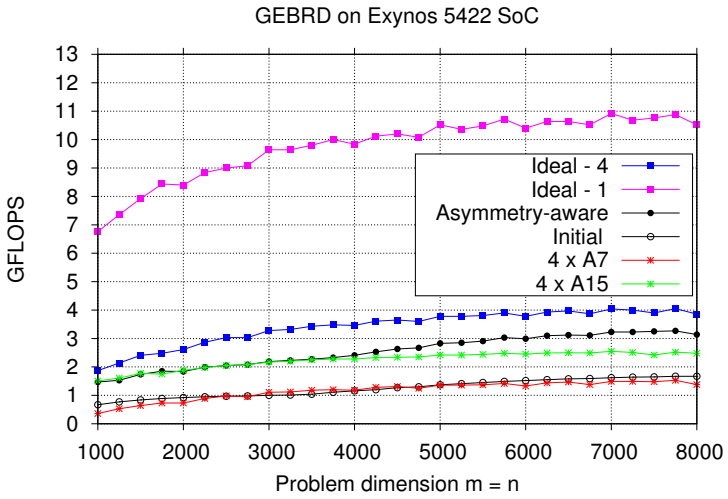

Figure 5.8: Performance of GEBRD on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC. This experiment sets the block size to $b = 96$.

In contrast to the results obtained for SYTRD, Figure 5.8 shows that, for GEBRD, the asymmetry-aware configuration steadily approaches but does not reach the performance of the Ideal-4 curve, providing a 2× higher performance rate than the initial configuration. Although there exists a large amount of calls to GEMV in GEBRD, and the aggregated time spent on this kernel is considerably large compared with the total execution time, only a small fraction of the GEMV kernels involve a matrix operand that is large enough to benefit from a parallel configuration. This is also the case even for large problem sizes, as the matrix operand passed to the matrix-vector multiplications decreases in size at each iteration step. Thus, using Ideal-4 as a theoretical reference function here is, to a certain extent, unrealistic.

5.5 Reduction to Upper Hessenberg Form (GEHRD)

Given a dense matrix $A \in \mathbb{R}^{n \times n}$, its *reduction to upper Hessenberg form H by an orthogonal similarity transformation* is given by $H = Q^T AQ$. Listing 5.3 displays a simplified C code that computes this decomposition for an `n`×`n` matrix stored starting at address `A` with column leading dimension `Alda`. For simplicity, we assume hereafter that the matrix size is an integer multiple of the block size b. The code overwrites the corresponding entries of the original matrix with the upper Hessenberg matrix H, leveraging the numerical *kernels* (or building blocks) LAHR2, GEMM, TRMM, LARFB and GEHD2 for this purpose. Routine LAHR2 mainly employs GEMV, TRMV, GEMM and TRMM to reduce the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero. Furthermore, it returns the auxiliary matrices which are needed to apply the transformation to the unreduced part of A. Routine LARFB applies a block reflector or its transpose to a general rectangular matrix using GEMM and TRMM as main

building blocks. Finally, routine GEHD2 implements an unblocked algorithm that reduces a general square matrix to upper Hessenberg.

```
#define A_ref(i,j) A[(j)*Alda+(i)]

for (k=0; k<n; k+=b) {

  // Reduce current diagonal block
  LAHR2( ..., &A_ref( 1, k ), Alda, work, ldwork );

    // Apply block reflector
    GEMM( ..., work, ldwork, &A_ref( k+b, k ), Alda, &A_ref( 1, k+b ), Alda );
    TRMM( ..., &A_ref( k+1, k ), Alda, work, ldwork );

    for(j=0; j<b-2; j++)
      AXPY( ..., work( ldwork*j+1 ), &A_ref( 1, k+j+1 )

    LARFB( ..., &A_ref( k, k+1 ), Alda, &A_ref( k+1, k ), Alda, &A_ref( k+1, k+b ), Alda, work, ldwork);

}
// Reduce the rest of the matrix
GEHD2( ..., &A_ref( k, k ), Alda, work,  &info);
```

Listing 5.3: Blocked routine for the reduction to upper Hessenberg form.

The cost of this reduction is $10n^3/3$ flops, with 20% cast in terms of distinct types of BLAS-2 matrix-vector products (called from LAHR2 and GEHD2) and the remaining 80% in efficient Level-3 BLAS [86].

5.5.1 The impact of the algorithmic block size in the Reduction to Upper Hessenberg Form (GEHRD)

As in the previous reductions, we first analyze how the type and number of cores that execute the main building blocks of GEHRD affect performance. As an initial step we run GEHRD while varying the block size b for different problem dimensions with the aim of exposing the impact of this parameter. Figure 5.9 reports the performance obtained in GFLOPS for this test. According to the results, a difference of 0.5 GFLOPS exists between the highest and lowest performance rates for the smallest problem size. In contrast with the previous reductions, for GEHRD the performance gap increases with the problem dimension for small block sizes. This is due to the higher percentage of flops performed in kernels of BLAS-3, which favors the use of larger block sizes. The main conclusion from this preliminary experiment is that the block size exerts a relevant and consistent impact on performance across the problem size range.

As shown in Figure 5.10, the execution time is mainly due to two components, namely GEMV and BLAS-3 (GEMM and TRMM). The execution time of GEMV (dark blue lines) grows with the block size, while BLAS-3 (light blue lines) has the opposite behaviur. However, since about 80% of the flops are executed in BLAS-3 calls, the execution time corresponding to GEMV, for this TSOR, amounts to about 50% of the total. This favors the use of larger block sizes as the performance decreases for GEMV due to the adoption of a larger block size (dark blue lines) is outweighed by the gains obtained in BLAS-3 due to the use of a larger block size (light blue lines).

5.5.2 Performance of the GEHRD routine

In this section we analyze the performance benefits of a tuned selection of the block size and core configuration when GEHRD is run on top of the asymmetry-aware version of BLIS. In the following test the number and type of cores of the main building blocks of GEHRD (GEMV and GEMM) are

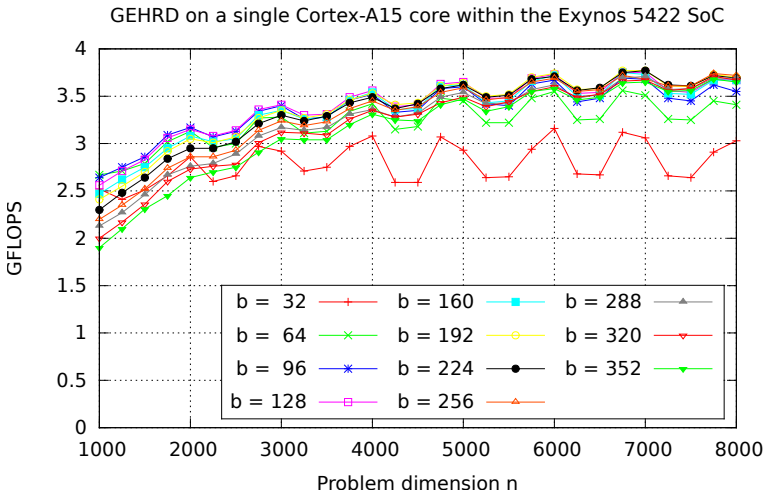

Figure 5.9: Performance of GEHRD on a single Cortex-A15 core within the ARM big.LITTLE AMP embedded in the Exynos 5422 SoC using different algorithmic block sizes.

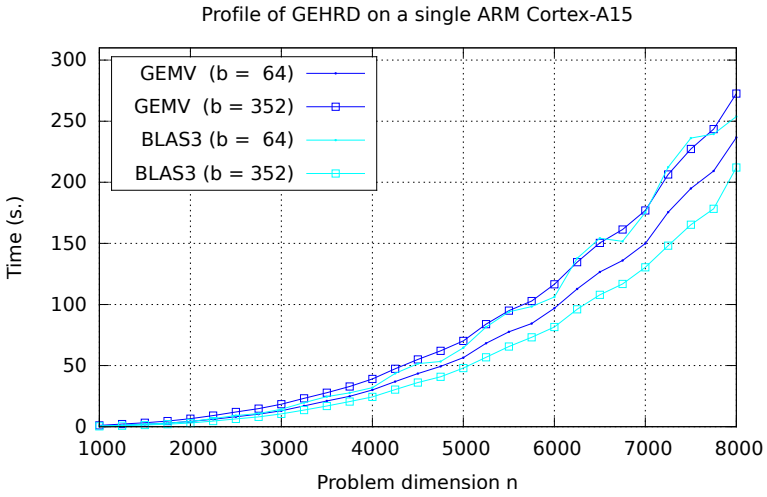

Figure 5.10: Profile of execution time spent by GEHRD on a single ARM Cortex-A15 core embedded in the Exynos 5422 SoC. This experiment sets the block size to 64 and 352, respectively, for comparison.

dynamically adjusted in order to improve the performance depending on the dimensions of the operands of each call to a building block.

Figure 5.11 illustrates the performance of routine GEBRD using the optimal algorithmic block size $b = 128$ (chosen experimentally). As for the previous reductions, the plots include the configuration with no optimizations applied to BLAS-2 (labeled as "Initial") as well as an alternative where all optimizations are present (labeled as "Asymmetry-aware"). We also include, for comparison purposes, the performance for both clusters in isolation (4 x A15 and 4 x A7); and two ideals, one

adding the performance of the clusters in isolation (Ideal - 4) and other adding the performance of one core of each type multiplied by 4 (Ideal - 1).

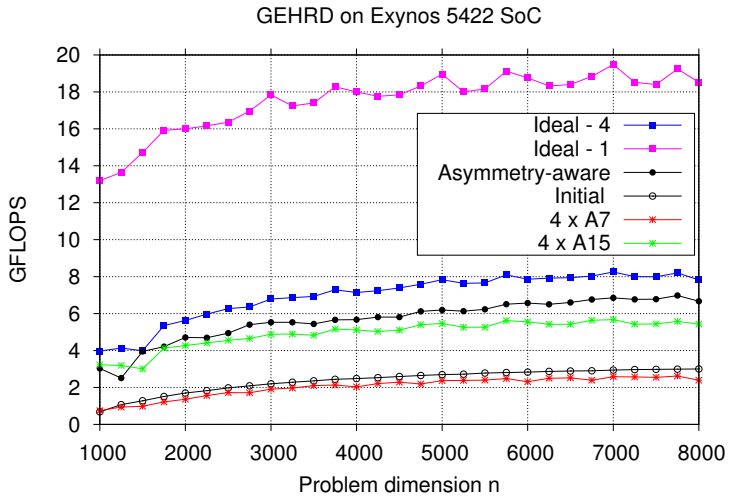

Figure 5.11: Performance of GEHRD on the ARMv7 big.LITTLE embedded in the Exynos 5422 SoC. This experiment sets the block size to $b = 128$.

The plot in Figure 5.11 shows that, for this TSOR routine, the performance of the Ideal-4 curve is never reached, but steadily approached by the asymmetry-aware configuration. In addition, the performance rate of GEHRD is 3× higher than that of the initial consideration, and slightly higher than the increment reached for GEBRD due to the larger amount of time spent on BLAS-3 operations in this case. As for GEBRD, the size of the matrix operand involved in GEMV is large enough to benefit from a parallel configuration for this kernel in a very small fraction of the invocations. For this reason, even though there exists a large amount of calls to GEMV in GEHRD and a long time is spent on them, using Ideal-4 as a theoretical reference is overly optimistic.

5.6 Modeling the Performance of the TSOR Routines

In the previous sections we illustrated the relevance of selecting the optimal block size and core configuration for the three building blocks appearing in SYTRD (namely, SYMV, GEMV and SYR2K). In this section, we propose a methodical approach to model performance; see also [83, 10, 84, 85]. Here we select the optimal block size for the TSOR procedures, based on the experimental performance observed for their building blocks and the theoretical flop count for each type of building block. In order to illustrate this, we employ the specific case of the reduction to tridiagonal form of a symmetric $n \times n$ matrix A via routine SYTRD. The same method carries over to the remaining two TSOR procedures.

At this point, we remind some observations connecting them to the experiments in Sections 5.2 and 5.3:

- Consider, for simplicity, that n is an integer multiple of the block size: $n = r \cdot b$, for a given integer r. The blocked single-step reduction to tridiagonal form processes the $n \times n$ matrix A, from top-left to bottom-right, in a set of iterations $j \in \{1,2,\ldots,n/b\}$, in blocks of b-column panels starting at rows/columns $\hat{\jmath} = (j-1)b = 0,\ b,\ 2b,\ldots,(r-1)b$. In general n may not

be an integer multiple of the block size and, in such case, the last panel receives a special treatment using an unblocked code.

- At iteration j, the assembly of V requires b symmetric matrix-vector multiplications, of decreasing dimensions $n-(\hat{\jmath}+1), n-(\hat{\jmath}+2), n-(\hat{\jmath}+3), \ldots, n-(\hat{\jmath}+b)$. Overall, the reduction performs $n-2$ calls to SYMV, involving matrices of dimensions $n-1, n-2, \ldots, 2$. Thus, the block size b has no effect on the number of flops nor the shapes of the operands passed to the sequence of calls to SYMV. We can conclude, hence, that b should exert no impact on the performance of SYTRD. This observation is confirmed by the results in Figure 5.4.

- The experimental analysis in Chapter 3 revealed that a close-to-optimal configuration for the parallel execution of SYMV, on the Exynos 522 SoC, employs a single Cortex-A15 core plus the full quad-core Cortex-A7 cluster. Slightly higher performance can be attained by activating a second Cortex-A15 core, but this will come at a non-negligible energy cost, which may be relevant for a low-power architecture. In consequence, we prefer the configuration with a single Cortex-A15 core. We use hand-coded micro-kernels for both types of core architectures, and a dynamic distribution of the iteration space of Loop 1 among the system cores, with cache-aware granularity m_c that depends on the core type (see Tables 5.1 and 5.2).

- The assembly of V at iteration j requires $6 \cdot b$ general matrix-vector multiplications of dimensions that depend on the algorithmic block size. Concretely, the dimensions of GEMV vary (linearly in both dimensions) from $n-(\hat{\jmath}+1) \times 1$ to $n-(\hat{\jmath}+b) \times b$. As a consequence, the overall number of flops performed by GEMV directly depends on the algorithmic block size of SYTRD, and we can conclude that b plays some role on performance. This is confirmed by the results in Figure 5.4.

- The experimental analysis in Chapter 3 hinted similar conclusions for GEMV to those exposed for SYMV in the sense that close-to-optimal performance is obtained by using a single Cortex-A15 core plus the full quad-core Cortex-A7 cluster. However, for small problem dimensions, it is more beneficial to use a single Cortex-A15 core.

- At the end of each iteration j, routine SYTRD invokes the SYR2K kernel to update the trailing submatrix in A of order $n-(\hat{\jmath}+b)+1$, using two panels of dimension $n-(\hat{\jmath}+b)+1 \times b$ each ($A_{22} := A_{22} - UV^T - VU^T$). Therefore, increasing the block size b accelerates the decay of the trailing submatrix dimensions as the iteration progresses, but augments the number of columns in the panels. In conclusion, we can expect that b has a certain effect on the performance of the sequence of calls to SYR2K, because it affects the operands' dimensions and shapes. This is reflected by the results in Figure 5.4.

- The algorithmic block size directly affects the matrix shapes involved in SYR2K and changes the threshold value for which it is more beneficial to use only the Cortex-A15 cluster or the full SoC. In conclusion, we should employ either the Cortex-A15 cluster or the full SoC when the dimension of A_{22} is respectively or larger than the threshold for a given algorithmic block size.

To sum up, at iteration $j \in \{1,2,\ldots,n/b\}$, routine SYTRD invokes the following Level-2 and Level-3 BLAS routines:

1. $b-1$ calls to SYMV, each involving a square matrix of order $r-k$, with $r = n-(j-1)b$ and $k = 1,2,\ldots,b-1$.
2. $6b$ calls to GEMV, each of the b involving a matrix of dimension $(r-k) \times k$, with $k = 1,2,\ldots,b$.

3. A single call to SYR2K to perform two updates of the form $\hat{C} += \hat{A} \cdot \hat{A}^T$, on a triangular part of a square result matrix $\hat{C}$ of order $s = n - jb$, and with the input operand $\hat{A}$ of dimension $s \times b$.

Therefore, the total cost of the routine, $4n^3/3$ flops, can be distributed among the three building blocks as follows:

1. SYMV: $\sum_{j=1}^{n/b} \sum_{k=1}^{b-1} 2(r-k)^2 = 2n^3/3$ flops.
2. GEMV: $\sum_{j=1}^{n/b} 12(rb^2/2 - b^3/3) = 3n^2b$ flops.
3. SYR2K: $\sum_{j=1}^{n/b} 2s^2b = 2n^3/3$ flops.

Note that the number of calls to SYMV and the dimension of the matrix operand for this kernel are independent of the algorithmic block size. Therefore, this type of kernel does not play a role in the optimization of b, and our target can be simplified to the minimization of the execution time for GEMV and SYR2K only. This can be formulated as:

$$\min_b \ \{T_{gemv} + T_{syr2k}\},$$

where the execution time due to the flops performed via GEMV and SYR2K are given by

$$\begin{aligned} T_{gemv} &= \sum_{j=1}^{n/b} \sum_{k=1}^{b} \frac{12(r-k)k}{G_{gemv}(r-k,k,\mathcal{C})} \quad \text{and} \\ T_{syr2k} &= \sum_{j=1}^{n/b} \frac{2s^2b}{G_{syr2k}(s,b,\mathcal{C})}, \end{aligned}$$

respectively. In the last expressions, $G_{gemv}(p,q,\mathcal{C})$ and $G_{syr2k}(p,q,\mathcal{C})$ stand for the GFLOP rates delivered by the corresponding routines when operating on a problem of dimension (p,q) using a core configuration $\mathcal{C}$. Note that in our model we distinguish the GFLOP rates of GEMV for the transposed and non transposed case. At this point, we remind that, for GEMV, the optimal configuration employs either a single Cortex-A15 core or 1 Cortex-A15 + 4 Cortex-A7 cores. In contrast, for SYR2K the optimization procedure has to select between 4 Cortex-A15 cores or the full Exynos 5422 SoC; see Figure 5.1.

This optimization model guides the search for the optimal block size and core configuration for SYTRD using the data for the experimental GFLOPS rates observed for SYR2K and GEMV. As we are only interested in a qualitative comparison of the execution time for different values of b and core configurations, we do not need to perform an exhaustive evaluation of the building blocks. Instead, we can select some representative values and interpolate the GFLOPS for the missing performance rates. Moreover, we note that the building blocks GEMM and GEMV appear also in the remaining two TSOR procedures, GEHRD and GEBRD. Therefore, we can reuse most of the experimental evaluation of the building blocks to tune the block size and core configuration for all three TSOR routines.

Figure 5.12 shows the evaluation of the performance determined via the model in comparison with the practical results obtained from an exhaustive execution of SYTRD using different algorithmic block sizes. For each problem dimension, the top plot in that figure reports model-driven estimates of the time increment with respect to the execution time obtained when using the optimal block size for that problem size. Concretely, for the problem of dimension $n = 1{,}000$, the variation of time is normalized with respect to the execution time using an algorithmic block size $b = 32$ (which corresponds to the optimal value of b for that problem size); for n ranging from 1,250 to 2,500 the results are normalized with respect to the execution time using $b = 64$; and for $n > 2{,}500$, they are normalized with respect to the execution time using $b = 96$. In order to offer quantitative

variations of the execution time, the model should have also taken into account the execution time of SYMV. However, as we are only interested in a qualitative detection of the optimal algorithmic block size, we can simplify the search by neglecting the impact of SYMV in the model. Overall, the model estimates that the optimal block size is either 64 or 96, with the differences between these two algorithmic block sizes being below 1%. In addition, the model exposes that the execution time grows with the algorithmic block size.

The model-driven search of the optimal algorithmic block size is validated with the exhaustive evaluation of the performance of SYTRD in the bottom plot in Figure 5.12. The practical results confirm that the actual algorithmic block sizes yielding the highest performance are also 64 and 96, with the performance declining when the algorithmic size exceeds the largest of these values.

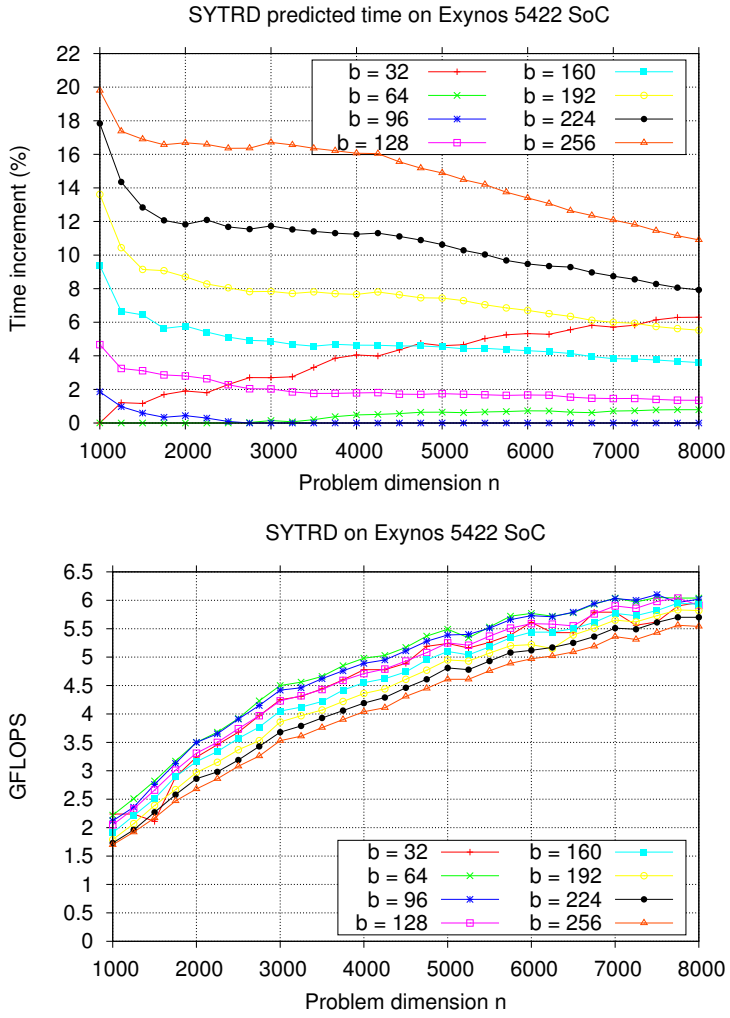

Figure 5.12: Model-driven estimation of the relative execution time (top) and actual performance (bottom).

The previous experiment shows that the model can be used to perform a search of the optimal algorithmic block size without testing the factorization itself. However, as the model predicts performance differences below 1% between the two close-to-optimal algorithmic block sizes, (though similar to those observed in practice,) this search methodology may introduce small deviations in

the value selected for b. Table 5.3 quantifies the impact of a suboptimal choice of b, comparing the execution time for executions that employ the algorithmic block size predicted by the model against those with the optimal algorithmic block size obtained from the experimentation. The results in the table reveal that the relative error is consistently below 2% (except in one case), being smaller than 1% for most problem dimensions.

Problem dimension	Optimal (b) Model	Optimal (b) Real	Difference (%)	Problem dimension	Optimal (b) Model	Optimal (b) Real	Difference (%)
1,000	32	32	–	4,750	96	64	1.66
1,250	64	64	–	5,000	96	64	1.82
1,500	64	64	–	5,250	96	96	–
1,750	64	64	–	5,500	96	64	0.36
2,000	64	64	–	5,750	96	64	1.05
2,250	64	64	–	6,000	96	64	0.69
2,500	64	64	–	6,250	96	64	0.17
2,750	96	64	1.89	6,500	96	96	–
3,000	96	64	1.78	6,750	96	96	–
3,250	96	64	2.19	7,000	96	64	0.16
3,500	96	64	0.86	7,250	96	96	–
3,750	96	64	1.85	7,500	96	96	–
4,000	96	64	1.81	7,750	96	64	1.32
4,250	96	64	1.59	8,000	96	64	0.33
4,500	96	64	1.16	Average			0.71

Table 5.3: Relative differences of time for SYTRD between executions using the optimal block size determined by the model and the real optimal value detected via exhaustive experimental tests.

5.7 Summary

In this chapter, we have presented architecture- and asymmetry-aware realizations of the TSOR procedures for the solution of general and symmetric dense eigenvalue problems as well as singular value problems for ARM big.LITTLE multi-core architectures. Our experiments with tuned versions of these routines, specifically optimized for the ARM Cortex-A15 and Cortex-A7 cores present in the big.LITTLE target, show a significant acceleration of the execution time compared with a simple execution of LAPACK's legacy codes for this purpose.

Our theoretical and practical analyses reveal the large impact of the Level-2 BLAS kernels on the performance of the TSOR procedures and the critical roles of the algorithmic block size and the core configuration. Concretely, the block size has to be finely adjusted to distribute the workload between the Level-2 and Level-3 kernels, taking into account that the memory-bound nature of the former often places this type of operations on the critical path of the algorithm. In addition, an optimal execution also depends on the number and type of cores employed for each type and dimension of the building blocks, with these two parameters determining when it becomes convenient to add the LITTLE cores to the execution. As a spin off of this study, we propose a model to predict the optimal algorithmic block size for a TSOR routine based on the input dimensions, the main building blocks of the reduction, and the flops performed by them.

Our experiments reveal that the model is highly accurate, providing an algorithmic block size that deviates by less than 1% from the best GFLOPS rate.

Overall, we believe that the approach applied to optimize the routines for the TSOR to condensed forms on the target platform presented in this chapter carries over to other asymmetric and heterogeneous architectures, including hybrid CPU-GPU systems, as well as multisocket/multi-core servers where distinct CPUs/cores operate at different frequencies.

CHAPTER 6

Conclusions

6.1 Conclusions and Main Contributions

The main goal of the dissertation was *to study, design, develop and analyze experimental solutions (models, programs, tools and techniques) that are energy-aware for scientific and engineering applications on low-power architectures.*

At the conclusion of this work, the main contributions of this dissertation are the following:

- The adaptation of the SYMV and GEMV BLAS-2 kernels in BLIS to an AMP.
- The adaptation of the BLAS-3 kernels in BLIS and an experimental study reflecting the performance gains attained by the asymmetry-aware version of the library.
- The performance and energy efficiency study of the Cholesky and LU factorizations when relying on the BLAS-3 asymmetry-aware version.
- The proposal of thread-level malleability technique to share computational resources in the task parallel execution of matrix factorizations.
- The performance and energy efficiency study of the Two-Sided Orthogonal Reductions (TSOR) routines when relying on the BLAS-3 and BLAS-2 asymmetry-aware versions. This contribution includes the creation of a model in order to determine the optimal block size and number of cores for the SYMV and GEMV kernels at each step of the TSOR routines.

The main contribution of this dissertation is the adaptation of BLAS-2 and BLAS-3 kernels from BLIS to create an asymmetry-aware version of the library that exploits the resources of an AMP, which has been the basis for the remaining parts of the dissertation. Using this library as the starting point, we have developed solutions for DLA operations that belong to the LAPACK level, completing the creation of asymmetry-aware DLA operations at all levels.

As a part of the thesis, and as a consequence of the complete study carried out for the LAPACK kernels, we have proposed a new technique to share execution resources among running tasks that we named as thread-level malleability. This technique can be applied either on symmetric or asymmetric platforms, reporting performance gains thanks to the better utilization of the existing

resources. In our case, the technique has been applied to DLA operations. In principle, the same idea can be applied to libraries of different nature, since it is a general strategy to share or reuse threads in a code.

An additional contribution of this thesis is the development of models to estimate performance and energy efficiency for different DLA operations on both, symmetric and asymmetric platforms. Those models gave us a better understanding of the operations and their behavior, especially on AMPs. In addition, they may be used in different scenarios, for example, they can be applied to any system running DLA operations in order to make scheduling decisions.

The following sections discuss the contributions and summarize the corresponding conclusions in more detail.

6.1.1 BLAS-3 kernels

An asymmetric-aware version of the BLAS-3 kernels included in BLIS was developed in order to exploit the available resources on an AMP. Given that all BLAS-3 kernels (except the TRSM operation) can be implemented following the same structure as the matrix multiplication, this specific kernel was used in order to explore different strategies to distribute the workload between the two type of cores present in the ARMv7 big.LITTLE (quad-core cortex A15 + quad-core cortex-A7) SoC. The key to our development was the integration of a coarse-grain scheduling policy, which dynamically distributes the workload between the two core types present in those architectures, combined with a complementary static schedule that repartitions this work among the cores of the same type.

With the promising performance results obtained for the matrix multiplication, the same strategy was applied to the remaining BLAS-3 operations for distinct operand shapes. Our results revealed excellent improvements in performance compared with the homogeneous implementations that operate exclusively on one type of core (either A15 or A7), and also with respect to multi-threaded implementations that simply apply a symmetric workload distribution and do not take into account the different cache organization of the cores and performance capabilities. In general, the results show considerable performance acceleration for the BLAS-3 kernels, and more moderate for the triangular system solve.

The complete asymmetry-aware BLAS-3 was tested also on a 64-bit ARMv8 architecture, with different number of big/LITTLE cores (four LITTLE cores + two big cores), delivering similar results as observed for the 32-bit ARM architecture. This experimental study was conclusive to demonstrate flexibility of our solution, as the second platform features different amount of big/LITTLE cores, clock frequency and precision.

6.1.2 BLAS-2 kernels

Following the same idea applied on BLAS-3, two operations of BLAS-2 were made asymmetry-aware. The changes in this case included using the cache optimization parameters, integrating the appropriate scheduling mechanism, developing hand-tuned micro-kernels, parallelizing the codes, and finding the cache optimization parameters for the multi-threaded kernels.

The hand-tuned micro-kernels exploit data locality when accessing data in the registers, providing gains in performance thanks to the better use of the resources. Additionally, the SYMV and GEMV routines were parallelized in order to distribute the workload between the distinct types of cores. This modification turned out in a study about the appropriate number and type of cores that must be used in each case.

All the changes made in order to make SYMV and GEMV asymmetry-aware led to important performance gains that complete the work already performed for BLAS-3 and shows the importance of employing full asymmetry-aware DLA libraries.

6.1.3 TSOR routines on AMPs and models

A contribution of this dissertation consisted in the realization of architecture- and asymmetry-aware versions of the TSOR procedures for the solution of general and symmetric dense eigenvalue problems as well as the computation of the SVD on ARM big.LITTLE multi-core architectures. The experiments with tuned versions of these routines, specifically optimized for the ARM Cortex-A15 and Cortex-A7 cores present in the ODROID-XU3, showed a significant acceleration of the execution time compared with a simple execution of LAPACK' legacy codes for this purpose.

The theoretical and practical analyses revealed the large impact of the BLAS-2 kernels on the performance of the TSOR procedures and the critical roles of the algorithmic block size and the core configuration. Concretely, the block size has to be finely adjusted to distribute the workload between the BLAS-2 and BLAS-3 kernels, taking into account that the memory-bound nature of the former often places this type of operations on the critical path of the algorithm. In addition, an optimal execution also depends on the number and type of cores employed for each type and dimension of the building blocks, with these two parameters determining when it becomes convenient to add the LITTLE cores to the execution. As a spin off of this insight, a simple model was proposed to predict the optimal algorithmic block size for a TSOR routine based on the input dimensions, the main building blocks of the reduction, and the flops performed by them.

6.1.4 LU and Cholesky factorization on big.LITTLE

Based on the asymmetry-aware BLIS-3, a legacy implementation of LAPACK was migrated to run on the target AMP. In doing so, the benefits and drawbacks of conducting a simple (plain) migration which does not perform any major optimizations in LAPACK were explored. The experimentation with two major routines from LAPACK, the LU and the Cholesky factorization, illustrates two distinct scenarios (cases), ranging from a compute-bound operation/routine (Cholesky factorization) where high performance is 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.

More specifically, the plain migration of the LU factorization provides good results for large matrix sizes. However, for small and medium sizes the panel factorization is a bottleneck that reduces performance significantly. As a mean to palliate the effect of the panel factorization, we adopted a look-ahead. The analysis of its impact exposed a significant performance improvement for medium and large matrix sizes.

The study of these two LAPACK routines exposed two important insights: taking advantage of an asymmetry-aware DLA library of basic operations (BLAS) is essential in order to exploit asymmetric platforms and, given that many scientific applications use more sophisticated DLA operations, new strategies should be considered when adapting the whole DLA stack (BLAS+LAPACK) to AMPs.

6.1.5 Thread-level malleability

A contribution of this dissertation was the introduction of Work Stealing (WS) and Early Termination (ET) as two novel techniques to avoid workload imbalance during the execution of matrix factorizations, enhanced with look-ahead, for the solution of linear systems. These techniques may

be applied when having more than one task in the code, and they will be beneficial if workload imbalance is present. Using the LU as an example, we could show that the WS mechanism especially benefits from the adoption of a malleable thread-level instance of BLIS, which allows the thread team in charge of the panel factorization, upon completion of this task, to be reallocated to the execution of the trailing update. The ET mechanism tackles the opposite situation, with a panel factorization that is costlier than the trailing update. In such scenario, the team that performed the update communicates to the second team that it should terminate the panel factorization, advancing the factorization process into the next iteration.

The results on an Intel Xeon E5-2603 v3 showed the performance benefits of the LU version enhanced with malleable BLIS and ET compared with a plain LU factorization as well as a version with look-ahead. The experiments also reported competitive performance compared with an LU factorization that was parallelized by means of a sophisticated runtime, such as OmpSs, that introduces look-ahead of dynamic (variable) depth. Compared with the OmpSs solution, our approach offered higher performance for most problem dimensions, seamlessly tuned the algorithmic block size, and featured a considerably smaller memory footprint as it does not require a sophisticated runtime support.

To conclude, and thanks to the promising results obtained in our tests, we can expect thread-level malleability to be crucial in order to exploit the massive thread parallelism of future architectures.

6.2 Related Publications

The contributions of the dissertation are supported by the publication of the content in different peer-reviewed national and international conferences and journals. In this section, the publications related to each contribution are listed and classified as directly-related to the content of the dissertation, indirectly-related or unrelated.

6.2.1 Directly related publications

6.2.1.1 Chapter 3. Basic Linear Algebra Subprograms (BLAS)

The first step in making DLA libraries asymmetry-aware was the analysis of GEMM [28], the BLAS-3 model operation, in order to identify the best mechanisms to adapt it to an AMP. The work presented in this paper studies different scheduling schemas that improve GEMM performance: static-asymmetric scheduling, cache-aware static-asymmetric scheduling and cache-aware dynamic-asymmetric scheduling. That study is extended in [24], where the best scheduling option is applied not only to GEMM, but for all the remaining BLAS-3 operations (except for TRSM), for different loop parallelizations and operand shapes. The second paper completes the work started with the GEMM kernel, proving the feasibility of an asymmetry-aware BLAS-3 and the performance gains of that adaption. Furthermore, the asymmetry-aware BLAS-3 version was ported to a different big.LITTLE architecture with a distinct ratio of big/LITTLE cores, consolidating the initial results.

The adaption of SYMV and GEMV BLAS-2 operations, following the same idea as for GEMM, are included as part of the work developed in the construction of asymmetry-aware versions of TSOR in [8], which will be analyzed in Section 6.2.1.3.

Journal [28] Catalán, S., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R. Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multi-core processors. *Journal of Cluster Computing* (2016), Vol. 19(3), pp. 1037–1051.

In this paper, we design and embed several architecture-aware optimizations into a multi-threaded general matrix multiplication (GEMM), a key operation of the BLAS, in order to obtain a high performance implementation for ARM big.LITTLE AMPs. Our solution is based on the reference implementation of GEMM in the BLIS library, and integrates a cache-aware configuration as well as asymmetric–static and dynamic scheduling strategies that carefully tune and distribute the operation's micro-kernels among the big and LITTLE cores of the target processor. The experimental results on a Samsung Exynos 5422, a system-on-chip with ARM Cortex-A15 and Cortex-A7 clusters that implements the big.LITTLE model, expose that our cache-aware versions of GEMM with asymmetric scheduling attain important gains in performance with respect to its architecture-oblivious counterparts while exploiting all the resources of the AMP to deliver considerable energy efficiency.

Catalán, S., Herrero, J. R., Igual, F. D., Rodríguez-Sánchez, R., Quintana-Ortí, E. S., Adeniyi-Jones, C. Multi-threaded dense linear algebra libraries for low-power asymmetric multi-core processors. *Journal of Computational Science* (2016), To appear. Journal [24]

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 multi-core 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.

6.2.1.2 Chapter 4. Factorizations

Upon the adaption of BLAS-3 the natural step was creating an asymmetry-aware version of the LAPACK operations. To do so, we focused on matrix factorizations for linear systems, selecting the Cholesky and LU factorizations as examples of different nature included in the library. The migration of LAPACK routines to an AMP started in [24], where a plain migration of the legacy version of LAPACK was performed to experimentally assess the benefits, limitations, and potential of this approach from the perspectives of both throughput and energy efficiency. That work demonstrated the need of new approaches in order to improve the performance for factorizations, especially for small/medium matrix sizes. In [36], an exhaustive analysis of different approaches (combining runtimes, asymmetry-aware DLA libraries and asymmetry-oblivious DLA libraries) for the Cholesky factorization was performed, showing the benefits of the combination of runtimes and asymmetry-aware libraries; this work was extended in [37], doing a similar study for the LU factorization and getting comparable conclusions.

A different strategy to increase performance is proposed in [26], advocating for thread-level malleable DLA libraries in order to share the execution resources among the tasks in a specific code. This idea comes as an alternative that does not require a runtime to maximize the use of the available resources. The application of that idea to AMPs is analyzed in [31], where the benefits

and drawbacks of using a static (through the proposed malleability approach) and dynamic (relying on a runtime) scheduling are thoroughly discussed.

The following is a detailed list of the main publications related to this topic:

JOURNAL [37] COSTERO, L., IGUAL, F. D., OLCOZ, K., CATALÁN, S., RODRÍGUEZ-SÁNCHEZ, R., QUINTANA-ORTÍ, E. S. Revisiting conventional task schedulers to exploit asymmetry in multi-core architectures for dense linear algebra operations. *Parallel Computing* (2017), To appear.

> Dealing with asymmetry in the architecture opens a plethora of questions related with the performance- and energy-efficient scheduling of task-parallel applications. While there exist early attempts to tackle this problem, for example via ad-hoc strategies embedded in a runtime framework, in this paper we take a different path, which consists in addressing the asymmetry at the library-level by developing a few asymmetry-aware fundamental kernels. The appealing consequence is that the architecture heterogeneity remains then hidden from the task scheduler.
>
> In order to illustrate the advantage of our approach, we employ two well-known matrix factorizations, key to the solution of dense linear systems of equations. From the perspective of the architecture, we consider two low-power processors, one of them equipped with ARM big.LITTLE technology; furthermore, we include in the study a different scenario, in which the asymmetry arises when the cores of an Intel Xeon server operate at two distinct frequencies. For the specific domain of dense linear algebra, we show that dealing with asymmetry at the library-level is not only possible but delivers higher performance than a naive approach based on an asymmetry-oblivious scheduler. Furthermore, this solution is also competitive in terms of performance compared with an ad-hoc asymmetry-aware scheduler furnished with sophisticated scheduling techniques.

CONFERENCE PROCEEDINGS [36] COSTERO, L., IGUAL, F. D., OLCOZ, K., CATALÁN, S., RODRÍGUEZ-SÁNCHEZ, R., AND QUINTANA-ORTÍ, E. S. Refactoring conventional task schedulers to exploit asymmetric ARM big.LITTLE architectures in dense linear algebra. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (2016), pp. 692–701.

> Dealing with asymmetry in the architecture opens a plethora of questions from the perspective of scheduling task-parallel applications for which there exist early ad-hoc strategies embedded into an asymmetry-conscious runtimes. In this paper we take a different path that addresses the complexity of the problem at the library level, via a few asymmetry-aware fundamental kernels, hiding the architecture heterogeneity from the task scheduler. For the specific domain of dense linear algebra, we show that this elegant solution delivers much higher performance than a naive approach based on an asymmetry-oblivious scheduler. Furthermore, this solution also outperforms an ad-hoc asymmetry-aware scheduler furnished with sophisticated scheduling techniques.

ARXIV [26] CATALÁN, S., HERRERO, J. R., QUINTANA-ORTÍ, E. S., RODRÍGUEZ-SÁNCHEZ, R. , AND VAN DE GEIJN, R. A. A case for malleable thread-level linear algebra libraries: The LU Factorization with Partial Pivoting. In *Applied Mathematics and Computation*, In review.

> We propose two novel techniques for overcoming load- imbalance encountered when implementing so-called look-ahead mechanisms in relevant dense matrix factorizations for

the solution of linear systems. Both techniques target the scenario where two thread teams are created/activated during the factorization, with each team in charge of performing an independent task/branch of execution. The first technique promotes worker sharing (WS) between the two tasks, allowing the threads of the task that completes first to be reallocated for use by the costlier task. The second technique allows a fast task to alert the slower task of completion, enforcing the early termination (ET) of the second task, and a smooth transition of the factorization procedure into the next iteration. The two mechanisms are instantiated via a new malleable thread-level implementation of the Basic Linear Algebra Subprograms (BLAS), and their benefits are illustrated via an implementation of the LU factorization with partial pivoting enhanced with look-ahead. Concretely, our experimental results on a six core Intel-Xeon processor show the benefits of combining WS+ET, reporting competitive performance in comparison with a task-parallel runtime-based solution.

Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., and Rodríguez-Sánchez, R. Static versus dynamic task scheduling of the LU factorization on ARM big.LITTLE architectures. In *International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, (2017), pp. 733–742. Conference Proceedings [31]

We investigate several parallel algorithmic variants of the LU factorization with partial pivoting (LUpp) that trade off the exploitation of increasing levels of task-parallelism in exchange for a more cache-oblivious execution. In particular, our first variant corresponds to the classical implementation of LUpp in the legacy version of LAPACK, which constrains the concurrency exploited to that intrinsic to the basic linear algebra kernels that appear during the factorization, but exerts an strict control of the cache memory and a static mapping of kernels to cores. A second variant relaxes this task-constrained scenario by introducing a look-ahead of depth one to increase task-parallelism, increasing the pressure on the cache system in terms of cache misses. Finally, the third variant orchestrates an execution where the degree of concurrency is only limited by the actual data dependencies in LUpp, potentially yielding to a higher volume of conflicts due to competition for the cache memory resources. The target platform for our implementations and experiments is a specific asymmetric multi-core processor (AMP) from ARM, which introduces the additional scheduling complexity of having to deal with two distinct types of cores; and an L2-shared cache per cluster of the AMP, which results in more conflictivity in the access to this key cache level.

6.2.1.3 Chapter 5. Reductions

This chapter tackles the TSOR used in the solution of dense eigenvalue and singular-value problems, using as starting point the asymmetry-aware BLAS-2 and BLAS-3 versions. This work is presented in [8], showing that a plain migration of these routines (using the SYTRD kernel as an example) relying only on the asymmetry-aware version of BLAS-3 is not enough when the target architecture is an AMP. The analysis of the impact when parallelizing BLAS-2 for AMPs and how the block size of these reductions affect the performance are included in this paper, as well as a model that helps in predicting the best combination of cores and the algorithmic block size depending on the matrix size.

Conference Proceedings [8]

Alonso, P., Catalán, S., Herrero, J. R., and Quintana-Ortí, E. S. Reduction to tridiagonal form for symmetric eigenproblems on asymmetric multi-core processors. In *International Workshop on Programming Models and Applications for Multi-cores and Manycores (PMAM)*, (2017), pp. 39–47.

> We investigate how to leverage the heterogeneous resources of an Asymmetric Multi-core Processor (AMP) in order to deliver high performance in the reduction to condensed forms for the solution of dense eigenvalue and singular-value problems. The routines that realize this type of two-sided orthogonal reductions (TSOR) in LAPACK are especially challenging, since a significant fraction of their floating-point operations are cast in terms of memory-bound kernels while the remaining part corresponds to efficient compute-bound kernels. To deal with this scenario: 1) we leverage implementations of memory-bound and compute-bound kernels specifically tuned for AMPs; 2) we select the algorithmic block size for the TSOR procedures via a practical model; and 3) we adjust the type and number of cores to use at each step of the reduction. Our experiments validate the model and assess the performance of our asymmetry-aware TSOR routines, using an ARMv7 big.LITTLE AMP, for three key operations: the reduction to tridiagonal form for symmetric eigenvalue problems, the reduction to Hessenberg form for general eigenvalue problems, and the reduction to bidiagonal form for singular-value problems.

6.2.2 Indirectly related publications

A parallel research was performed into time and energy modeling of DLA operations and DLA fault tolerance on AMPs. Models were important in the characterization of the operations in order to gain insights of the potential bottlenecks of each one. Those models were built for BLAS kernels [27, 9] and LAPACK operations [29, 8] on both symmetric and asymmetric platforms. Moreover, analyses about fault tolerance and their cost were performed as a critical issue on very low-power platforms [33, 7, 25].

Journal [9] Alonso, P., Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Time and energy modeling of high-performance level-3 BLAS on x86 architectures. *Simulation Modelling Practice and Theory* (2015), Vol. 55, pp. 77–94.

Journal [33] Chalios, C., Nikolopoulos, D., Catalán, S., Quintana-Ortí, E. S. Evaluating asymmetric multi-core systems-on-chip and the cost of fault tolerance using iso-metrics. *IET Computers & Digital Techniques* (2016), Vol. 10 (2), pp. 85–92.

Journal [29] Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Time and energy modeling of a high-performance multi-threaded Cholesky factorization. *Journal of Supercomputing* (2017), Vol. 73(1), pp. 139–151.

Journal [25] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R. Energy balance between voltage-frequency scaling and resilience for linear algebra routines on low-power multi-core architectures. *Parallel Computing* (2017), To appear.

Conference Proceedings [22] Catalán, S., Gónzalez-Domínguez, J., Mayo, R., and Quintana-Ortí, E. S. Analyzing the energy efficiency of the memory subsystem in multi-core processors In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, (2014) pp. 10–17.

Conference Proceedings [27]

Catalán, S., Igual. F. D., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Time and energy modeling of high performance multi-threaded matrix multiplication. In *15th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, (2015), Vol. 1, pp. 311–316.

Aliaga, J., Catalán, S., Chalios, C., Nikolopoulos, D., and Quintana-Ortí, E. S. Performance and fault tolerance of preconditioned iterative solvers on low-power ARM architectures. In *Parallel Computing (ParCo)*, (2016), Vol. 27, pp. 711–720. Conference Proceedings [7]

6.2.3 Other publications

The publications listed in this section refer mainly to the collaboration in the development of some modules of the PMLib library and the analysis of the framework. This section also includes publications about distinct approaches in order to make power measurements and alternatives that may optimize the collection of power samples.

The publications related to that parallel work are listed below:

Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Automatic Detection of Power Bottlenecks in Parallel Scientific Applications. *Computer Science - Research and Development* (2013), Vol 29 (3-4), pp. 221–229. Journal [16]

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. Quintana-Ortí, E. S. Assessing power monitoring approaches for energy and power analysis of computers. *Journal of Sustainable Computing, Informatics and Systems* (2014), Vol. 4 (2), pp. 68–82. Journal [42]

Castaño, M. A., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Reducing the cost of power monitoring with DC wattmeters. *Computer Science - Research and Development* (2014), Vol. 30(2), pp. 107–114. Journal [21]

Charles, J., Sawyer, W., Dolz, M. F., Catalán, S. Evaluating the performance and energy efficiency of the COSMO-ART model system. *Computer Science - Research and Development* (2014), Vol. 30(2), pp. 177–186. Journal [34]

Dolz, M. F., Kunkel, J., Chasapis, K., Catalán, S. An analytical methodology to derive power models based on hardware and software metrics. *Computer Science - Research and Development* (2016), Vol. 31(4), pp. 165–174. Journal [43]

Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Tracing the power and energy consumption of the QR factorization on multi-core processors. In *12th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)* (2012), pp. 134–142. Conference Proceedings [15]

Barrachina, S., Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. S. An integrated framework for power-performance analysis of parallel scientific workloads. In *3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)* (2013), 114–119. Conference Proceedings [14]

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. Quintana-Ortí, E. S. Solving some mysteries in power monitoring of servers: Take care of your wattmeters! In *Energy Efficiency in Large Scale Distributed Systems (EE-LSDS)*, Lecture Notes in Computer Science, Vol. 8046. Springer-Verlag, 2013, pp. 3–18. Conference Proceedings [41]

Conference Proceedings [30]

Catalán, S., Malossi, A. C. I., Bekas, C., and Quintana-Ortí, E. S. The impact of voltage-frequency scaling for the matrix-vector product on the IBM Power8. In *European Conference on Parallel Processing (Euro-Par)*, (2016), pp. 103–116.

Conference Proceedings [23]

Catalán, S., Ezzatti, P., Quintana-Ortí, E. S, and Remón, A. The impact of panel factorization on the Gauss-Huard algorithm for the solution of linear systems on modern architectures. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, (2016), pp. 405–416.

6.3 Open Research Lines

The use of low-power systems in HPC is a relatively novel approach in computer science, and thus several research questions remain open after the conclusion of this thesis. Some of the open research lines are detailed next:

- Fully adaption of BLAS-2 operations to make them asymmetry-aware.
- Development of fully-functional implementations of the malleable BLAS kernels, based on BLIS, and further tailoring for AMP architectures.
- Creation of interfaces that may ease the use of thread-level malleability functionality.
- Extension of the thread-level malleability approach to a multi-task scenario through the integration of this technique into runtimes that exploit task-level parallelism, such as OmpSs [48].

CAPÍTULO 7

Conclusiones

7.1 Conclusiones y Contribuciones Principales

El principal objetivo de la tesis era *estudiar, diseñar, desarrollar y analizar soluciones experimetnales (modelos, programas, herramientas y técnicas) que son conscientes de la energía para aplicaciones científicas y de ingeniería en arquitecturas de bajo consumo.*

Tras la finalización de este trabajo, las contribuciones principales de esta tesis son las siguientes:

- La adaptación de los kernels BLAS-2 SYMV y GEMV para un multiprocesador asimétrico (AMP del inglés *Asymmetric Multicore Processor*).
- La adaptación de los kernels BLAS-3 de BLIS y un estudio experimental sobre las ganancias de rendimiento de la versión de la biblioteca consciente de la asimetría.
- El estudio de rendimiento y eficiencia energética de las factorizaciones Cholesky y LU al utilizar la versión BLAS-3 consciente de la asimetría.
- La propuesta de la técnica de maleabilidad a nivel de thread para compartición de recursos computacionales en una ejecución paralela de factorizaciones de matrices basada en tareas.
- El estudio de rendimiento y eficiencia energética de las rutinas TSOR al utilizar las versiones BLAS-2 y BLAS-3 conscientes de la asimetría. Esta contribución incluye la creación de un modelo para determinar el tamaño óptimo de bloque y el número de cores de los kernels SYMV y GEMV en cada paso de las rutinas TSOR.

  This contribution includes the creation of a model in order to determine the optimal block size and number of cores for the SYMV and GEMV kernels at each step of the TSOR routines.

La principal contribución de esta tesis es la adaptación de los kernels BLAS-2 y BLAS-3 de BLIS para crear una versión de la biblioteca consciente de la asimetría que explote los recursos de un AMP, lo que ha constituido la base para el trabajo restante de la tesis. Utilizando esta biblioteca como punto de partida, se han desarrollado soluciones para operaciones de álgebra lineal densa (o DLA del inglés *Dense Linear Algebra*) que pertenecen al nivel de LAPACK, completando la creación de operaciones DLA conscientes de la asimetría a todos los niveles.

Como parte de la tesis y como consecuencia del completo estudio llevado a cabo para los kernels de LAPACK, se ha propuesto una técnica de compartición de recursos de ejecución entre tareas en ejecución denominada maleabilidad a nivel de thread. Esta técnica puede ser aplicada tanto en sistemas simétricos como asimétricos, proporcionando mejoras en el rendimiento gracias a la mejor utilización de los recursos existentes. En este caso, la técnica se ha aplicado a operaciones DLA. En principio, la misma idea se puede aplicar a bibliotecas de distinta naturaleza, puesto que es una estrategia general para compartir o reutilizar threads en un código.

Una contribución adicional de esta tesis es el desarrollo de modelos para estimar el rendimiento y eficiencia energética para diferentes operaciones DLA tanto en sistemas simétricos como asimétricos. Estos modelos han permitido un mejor conocimiento de las operaciones y de su comportamiento, especialmente en los AMPs. Además, pueden ser usados en cualquier sistema que ejecute operaciones DLA para tomar decisiones de planificación

Las siguientes secciones explican las contribuciones y resumen las correspondientes coclusiones en más detalle.

7.1.1 Kernels BLAS-3

Una versión asimétrica de los kernels BLAS-3 incluidos en BLIS se ha desarrollado para explotar los recursos disponibles en un AMP. Puesto que todos los kernels BLAS-3 (excepto la operación TRSM) se pueden implementar siguiendo la misma estructura que la del producto de matrices, este kernel en concreto se ha utilizado para explorar distintas estrategias de distribución de trabajo entre los dos tipos de cores presentes en el SoC ARMv7 big.LITTLE (quad-core cortex A15 + quad-core cortex-A7). La clave de nuestra implementación es la integración de una política de planificación de grano grueso, que distribuye dinámicamente la carga de trabajo entre los dos tipos de cores presentes en estas arquitecturas, combinada con una planificación estática complementaria que reparte este trabajo entre los cores del mismo tipo.

Con los prometedores resultados de rendimiento obtenidos para el producto de matrices, la misma estrategia se ha aplicado al resto de operaciones BLAS-3 para distintas formas de operandos. Los resultados han demostrado mejoras importantes en rendimiento en comparación con las implementaciones homogéneas que trabajan exclusivamente en un tipo de core (bien A15 o A7), y también respecto a las implementaciones multi-hilo que simplemente aplican una distribución de trabajo simétrica y no tienen en cuenta la distinta organización de caches de los cores ni su capacidad de cómputo. En general, los resultados muestran una aceleración considerable del rendimiento para los kernels BLAS-3, siendo más moderada para la resolución de sistemas triangulares.

La versión completa de BLAS-3 se ha probado también en un arquitectura ARMv8 de 64 bits, con distinto número de cores big/LITTLE (cuatro cores LITTLE + dos cores big), proporcionando resultados similares a los observados para la arquitectura ARM de 32 bits. Este estudio experimental ha sido concluyente a la hora de demostrar la flexibilidad de nuestra solución, ya que la segunda plataforma dispone de una cantidad diferente de cores, distinta frecuencia de reloj y precisión.

7.1.2 Kernels BLAS-2

Siguiendo la misma idea aplicada en BLAS-3, dos operaciones BLAS-2 se han modificado para hacerlas conscientes de la asimetría. Los cambios en este caso incluyen el uso de parámetros de optimización de cache, la integración del mecanismo de planificación adecuado, el desarrollo de micro-kernels específicos, la paralelización de los códigos y la búsqueda de los parámetros óptimos de cache para los kernels multi-hilo.

Los micro-kernels específicos explotan la localidad de los datos al acceder a los registros, proporcionando ganancias en el rendimiento gracias a un mejor uso de los recursos. Además, las rutinas SYMV y GEMV se han paralelizado para distribuir la carga de trabajo entre distintos tipos de core. Esta modificación dio lugar a un estudio sobre el número y tipo de cores apropiado que debe usarse en cada caso.

Todos los cambios hechos para que SYMV y GEMV sean consciente de la asimetría han dado lugar a mejoras importantes en el rendimiento que completan el trabajo ya realizado para BLAS-3 y muestran la importancia de utilizar bibliotecas de álgebra lineal densa conscientes de la asimetría a todos los niveles.

7.1.3 Rutinas TSOR en AMPs y modelos

Una contribución de esta tesis ha consistido en la realización de versiones conscientes de la asimetría y de la arquitectura de los procedimientos TSOR para la resolución general y simétrica de problemas de valores propios densos así como la computación de la descomposición de valores singulares (SVD del inglés *Singular Value Decomposition*) en arquitecturas ARM big.LITTLE multicore. Los experimentos con las versiones apropiadas de estas rutinas, específicamente optimizadas para los cores ARM Cortex-A15 y Cortex-A7 presentes en el ODROID-XU3, han mostrado una aceleración significativa en el tiempo de ejecución comparado con una simple ejecución de los códigos LAPACK para este propósito.

Los análisis teóricos y prácticos han revelado el gran impacto de los kernels BLAS-2 en el rendimiento de los procedimientos TSOR y de los roles críticos del tamaño del bloque algorítmico y de la configuración de los cores. Concretametne, el tamaño de bloque tiene que ser ajustado con precisión para distribuir la carga de trabajo entre los kernels BLAS-2 y BLAS-3, teniendo en cuenta que en el caso de las primeros se tratan de operaciones limitadas por memoria que, debido a su naturaleza, a menudo se encuentran en el camino crítico del algoritmo. Además, una ejecución óptima también depende del número y tipo de cores empleados para cada tipo de dimensión de los bloques constituyentes, con estos dos parámetros determinando cuando conviene añadir cores LITTLE a la ejecución. Como contribución derivada de esta circunstancia, se ha propuesto un modelo sencillo para predecir el tamaño óptimo del bloque algorítmico para una rutina TSOR basado en las dimensiones de los operandos de entrada, los principales bloques constituyentes de la reducción y los flops obtenidos en por cada uno.

7.1.4 Factorizaciones LU y Cholesky en big.LITTLE

Basándonos en la versión BLIS-3 consciente de la asimetría, la implementación LAPACK de referencia se ha migrado para ejecutarla sobre un AMP. De esta manera, se han explorado los beneficios e inconvenientes de llevar a cabo una migración simple (plana) en la que no se realiza ninguna optimización relevante en LAPACK. Los experimentos con dos de las rutinas principales de LAPACK, las factorizaciones LU y Cholesky, ilustran dos escenarios (casos) distintos, yendo de una operación/rutina limitada por memoria (factorización Cholesky) en la que un alto rendimiento se obtiene fácilmente solo con la migración plana; a una operación limitada por cómputo (factorización LU) en la que para conseguir la misma mejora en los resultados es necesaria una reorganización significativa del código que introduce mecanismos avanzados de planificación.

Más concretamente, la migración plana de la factorización LU proporciona buenos resultados para matrices de tamaño grande. Sin embargo, para tamaños medianos y pequeños la factorización del panel es un cuello de botella que reduce el rendimiento significativamente. Como medio para paliar el efecto de la factorización del panel, se ha aplicado la técnica de look-ahead. El análisis

del impacto al aplicar esta estrategia ha demostrado mejoras importantes en el rendimiento para tamaños de matrices medianos y grandes.

El estudio de estas dos rutinas LAPACK ha mostrado dos aspectos importantes: es esencial utilizar bibliotecas de álgebra lineal densa conscientes de la asimétria para las operaciones básicas (BLAS) para así poder explotar las plataformas asimétricas y, dado que muchas aplicaciones científicas utilizan operaciones de álgebra lineal densa más sofisticadas, se deben considerar nuevas estrategias al adaptar la pila completa de álgebra lineal densa (BLAS+LAPACK) a los AMPs.

7.1.5 Maleabilidad a nivel de thread

Una contribución de esta tesis ha sido la introducción de dos nuevas técnicas, Work Stealing (WS) y Early Termination (ET), para evitar el desbalanceo de carga durante la ejecución de factorizaciones sobre matrices, mejorada con look-ahead, para la resolución de sistemas lineales. Estas técnicas se pueden aplicar cuando existe más de una tarea en el código y serán beneficiosas en aquellos casos en los que hay desbalanceo de carga. Utilizando la LU como ejemplo, se ha podido comprobar que el mecanismo de WS se beneficia especialmente de una versión de BLIS con maleabilidad a nivel de thread, lo que permite que el equipo de threads que se encargan de la factorización del panel, una vez terminada esta tarea, se puedan reasignar a la ejecución de la actualización de la matriz. El mecanismo ET se encarga de la situación contraria, con una factorización del panel mucho más costosa que la actualización de la matriz. En este escenario, el equipo de threads que ejecuta la actualización le comunica al segundo equipo que debe terminar la factorización del panel, avanzando el proceso de factorización a la siguiente iteración.

Los resultados en un Intel Xeon E5-2603 v3 han mostrado los beneficios en el rendimiento de la versión de la LU mejorada con BLIS maleable y ET en comparación con la factorización LU plana, así como con la versión con look-ahead. Los experimentos también muestran un rendimiento competitivo comparado con una factorización LU paralelizada mediante un sofisticado runtime, como OmpSs, que introduce un look-ahead de profundidad dinámica (variable). Comparado con la solución de OmpSs, nuestra aproximación ofrece mayor rendimiento para la mayoría de problemas, un ajuste óptimo del tamaño de bloque algorítmico y un uso de la memoria considerablemente menor ya que no requiere el soporte de un sofisticado runtime.

Para concluir, y gracias a los prometedores resultados de los tests realizados, se puede esperar que la maleabilidad a nivel de thread sea crucial para explotar el paralelismo masivo de threads en futuras arquitecturas.

7.2 Publicaciones relacionadas

Las contribuciones de esta tesis están respaldadas por la publicación de su contenido en disintas conferencias y revistas revisadas por pares tanto de carácter nacional como internacional. En esta sección, se listan las publicaciones relacionadas con cada contribución y se clasifican como directamente relacionadas con el contenido de la tesis, indirectamente relacionadas y no relacionadas.

7.2.1 Publicaciones directamente relacionadas

7.2.1.1 Chapter 3. Basic Linear Algebra Subprograms (BLAS)

El primer paso para hacer bibliotecas de álgebra lineal densa conscientes de la asimetría ha sido el análisis de GEMM [28], la operación modelo de BLAS-3, para identificar el mejor mecanismo para adaptarla a un AMP. El trabajo presentado en este artículo estudia distintos esquemas de

planificación que mejoran el rendimiento de GEMM: planificación asimétrica estática, planificación asimétrica estática consciente de la cache y planificiación asimétrica dinámica consciente de la cache. Este estudio se extiende en [24], donde la mejor opción de planificaión se aplica no solo a la GEMM, sino a todas las operaciones restantes de BLAS-3 (excepto para TRSM), para distintas opciones de paralelización de bucles y formas de operandos. El segundo artículo completa el trabajo empezado con el kernel GEMM, probando la viabilidad de una versión BLAS-3 completa consciente de la asimetría y analizando las mejoras en el rendimiento gracias a esta adaptación. Además, la versión BLAS-3 consciente de la asimétria se porta a una arquitectura big.LITTLE diferente con una proporción de cores big/LITTLE distinta, consolidando los resultados iniciales.

La adaptación de las operaciones BLAS-2 SYMV y GEMV, siguiendo la misma idea que para GEMM, se ha incluido como parte del trabajo desarrollado en la creación de versiones conscientes de la asimetría de TSOR en [8], lo que será analizado en la Sección 7.2.1.3.

Catalán, S., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R. Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multi-core processors. *Journal of Cluster Computing* (2016), Vol. 19(3), pp. 1037–1051. — Journal [28]

Catalán, S., Herrero, J. R., Igual, F. D., Rodríguez-Sánchez, R., Quintana-Ortí, E. S., Adeniyi-Jones, C. Multi-threaded dense linear algebra libraries for low-power asymmetric multi-core processors. *Journal of Computational Science* (2016), To appear. — Journal [24]

7.2.1.2 Chapter 4. Factorizaciones

Una vez finalizada la adaptación de BLAS-3 el siguiente paso natural ha sido la creación de una versión consciente de la asimetría de las operaciones LAPACK. Para ello, nos hemos centrado en la factorización de matrices para sistemas lineales, seleccionando las factorizaciones Cholesky y LU como ejemplos de distinta naturaleza incluidos en la biblioteca. La migración de las rutinas LAPACK a un AMP se empezó en [24], en el que una migración plana de LAPACK se realizó para comprobar experimentalmente los beneficios, limitaciones y el potencial de esta aproximación desde la perspectiva del rendimiento y la eficiencia energética. Este trabajo demostró la necesidad de nuevas aproximaciones para mejorar el rendimiento de las factorizaciones, especialmente para tamaños de matriz pequeños/medianos. En [36], un análisis exhaustivo de diferentes enfoques (combinando runtimes, bibliotecas de álgebra lineal densa conscientes de la asimetría y bibliotecas de álgebra lineal densa no conscientes de la misma) para la factorización Cholesky se llevó a cabo, mostrando los beneficios de la combinación de runtimes y bibliotecas conscientes de la asimetría; este trabajo se extendió en [37], llevando a cabo un estudio similar para la factorización LU y obteniendo conclusiones comparables.

Una estrategia diferente para incrementar el rendimiento se propuso en [26], abogando por la maleabilidad a nivel de thread en bibliotecas de álgebra lineal densa para la compartición de recursos de ejecución entre las tareas de un código dado. Esta idea surge como una alternativa que no requiere de un runtime para maximizar el uso de los recursos disponibles. La aplicación de esta idea a AMPs se analiza en [31], donde los beneficios e inconvenientes de usar una planificación estática (a través de la técnica de maleabilidad propuesta) y dinámica (basada en un runtime) se discuten detalladamente.

A continuación se presenta una lista detallada de las principales publicaciones relacionadas con este tema:

Costero, L., Igual, F. D., Olcoz, K., Catalán, S., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Revisiting conventional task schedulers to exploit asymmetry in multi-core architectures for dense linear algebra operations. *Parallel Computing* (2017), To appear. — Journal [37]

Conference Proceedings [36] Costero, L., Igual, F. D., Olcoz, K., Catalán, S., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Refactoring conventional task schedulers to exploit asymmetric ARM big.LITTLE architectures in dense linear algebra. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (2016), pp. 692–701.

Arxiv [26] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R. , and van de Geijn, R. A. A case for malleable thread-level linear algebra libraries: The LU Factorization with Partial Pivoting. In *Applied Mathematics and Computation*, In review.

Conference Proceedings [31] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., and Rodríguez-Sánchez, R. Static versus dynamic task scheduling of the LU factorization on ARM big.LITTLE architectures. In *International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, (2017), pp. 733–742.

7.2.1.3 Chapter 5. Reducciones

Este capítulo aborda las rutinas TSOR usadas en la resolución de problemas densos de valores propios y valores singulares, utilizando como base las versiones BLAS-2 y BLAS-3 conscientes de la asimetría. Este trabajo se presenta en [8], mostrando que la migración directa de estas rutinas (utilizando el kernel sytrd como ejemplo) basada únicamente en la versión BLAS-3 consciente de la asimetría no es suficiente cuando la arquitectura objetivo es un AMP. El análisis del impacto al paralelizar BLAS-2 para los AMPs y el efecto que tiene el tamaño de bloque sobre el rendimiento se incluyen en este artículo, así como un modelo que ayuda a predecir la mejor combinación de cores y de tamaño de bloque algorítmico dependiendo del tamaño de matriz.

Conference Proceedings [8] Alonso, P., Catalán, S., Herrero, J. R., and Quintana-Ortí, E. S. Reduction to tridiagonal form for symmetric eigenproblems on asymmetric multi-core processors. In *International Workshop on Programming Models and Applications for Multi-cores and Manycores (PMAM)*, (2017), pp. 39–47.

7.2.2 Publicaciones indirectamente relacionadas

Una investigación paralela se ha llevado a cabo sobre modelado de tiempo y energía para operaciones de álgebra lineal densa y tolerancia a fallos sobre AMPs. Los modelos han sido relevantes en la caracterización de las operaciones para obtener un mejor conocimiento de los potenciales cuellos de botella en cada una. Estos modelos se han construido para kernels BLAS [27, 9] y operaciones LAPACK [29, 8] en paltaformas simétricas y asimétricas. Además, se han realizado análisis sobre tolerancia a fallos y su coste como un tema crítico en plataformas de muy bajo consumo [33, 7, 25].

Journal [9] Alonso, P., Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Time and energy modeling of high-performance level-3 BLAS on x86 architectures. *Simulation Modelling Practice and Theory* (2015), Vol. 55, pp. 77–94.

Journal [33] Chalios, C., Nikolopoulos, D., Catalán, S., Quintana-Ortí, E. S. Evaluating asymmetric multi-core systems-on-chip and the cost of fault tolerance using iso-metrics. *IET Computers & Digital Techniques* (2016), Vol. 10 (2), pp. 85–92.

Journal [29] Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Time and energy modeling of a high-performance multi-threaded Cholesky factorization. *Journal of Supercomputing* (2017), Vol. 73(1), pp. 139–151.

Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R. Energy balance between voltage-frequency scaling and resilience for linear algebra routines on low-power multi-core architectures. *Parallel Computing* (2017), To appear. Journal [25]

Catalán, S., Gónzalez-Domínguez, J., Mayo, R., and Quintana-Ortí, E. S. Analyzing the energy efficiency of the memory subsystem in multi-core processors In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, (2014) pp. 10–17. Conference Proceedings [22]

Catalán, S., Igual. F. D., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Time and energy modeling of high performance multi-threaded matrix multiplication. In *15th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, (2015), Vol. 1, pp. 311–316. Conference Proceedings [27]

Aliaga, J., Catalán, S., Chalios, C., Nikolopoulos, D., and Quintana-Ortí, E. S. Performance and fault tolerance of preconditioned iterative solvers on low-power ARM architectures. In *Parallel Computing (ParCo)*, (2016), Vol. 27, pp. 711–720. Conference Proceedings [7]

7.2.3 Otras publicaciones

Las publicaciones listadas en esta sección se refieren en su mayoría a la colaboración en el desarrollo de algunos módulos de la biblioteca PMLib y el análisis de este framework. Esta sección también incluye publicaciones sobre distintas aproximaciones para realizar mediciones de consumo y diversas alternativas para optimizar la recolección de muestras de potencia.

Las publicaciones relacionadas con este trabajo paralelo se listan a continuación:

Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Automatic Detection of Power Bottlenecks in Parallel Scientific Applications. *Computer Science - Research and Development* (2013), Vol 29 (3-4), pp. 221–229. Journal [16]

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. Quintana-Ortí, E. S. Assessing power monitoring approaches for energy and power analysis of computers. *Journal of Sustainable Computing, Informatics and Systems* (2014), Vol. 4 (2), pp. 68–82. Journal [42]

Castaño, M. A., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Reducing the cost of power monitoring with DC wattmeters. *Computer Science - Research and Development* (2014), Vol. 30(2), pp. 107–114. Journal [21]

Charles, J., Sawyer, W., Dolz, M. F., Catalán, S. Evaluating the performance and energy efficiency of the COSMO-ART model system. *Computer Science - Research and Development* (2014), Vol. 30(2), pp. 177–186. Journal [34]

Dolz, M. F., Kunkel, J., Chasapis, K., Catalán, S. An analytical methodology to derive power models based on hardware and software metrics. *Computer Science - Research and Development* (2016), Vol. 31(4), pp. 165–174. Journal [43]

Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Tracing the power and energy consumption of the QR factorization on multi-core processors. In *12th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)* (2012), pp. 134–142. Conference Proceedings [15]

Barrachina, S., Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. S. An integrated framework for power-performance analysis of parallel Conference Proceedings [14]

scientific workloads. In *3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)* (2013), 114–119.

Conference Proceedings [41] Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. Quintana-Ortí, E. S. Solving some mysteries in power monitoring of servers: Take care of your wattmeters! In *Energy Efficiency in Large Scale Distributed Systems (EE-LSDS)*, Lecture Notes in Computer Science, Vol. 8046. Springer-Verlag, 2013, pp. 3–18.

Conference Proceedings [30] Catalán, S., Malossi, A. C. I., Bekas, C., and Quintana-Ortí, E. S. The impact of voltage-frequency scaling for the matrix-vector product on the IBM Power8. In *European Conference on Parallel Processing (Euro-Par)*, (2016), pp. 103–116.

Conference Proceedings [23] Catalán, S., Ezzatti, P., Quintana-Ortí, E. S, and Remón, A. The impact of panel factorization on the Gauss-Huard algorithm for the solution of linear systems on modern architectures. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, (2016), pp. 405–416.

7.3 Líneas abiertas de investigación

El uso de sistemas de bajo consumo en CAP es relativamente nuevo en computación y, por tanto, varias cuestiones permanecen abiertas tras la conclusión de esta tesis. Algunas de las líneas abiertas de investigación se detallan a continuación:

- Adaptación completa de operaciones BLAS-2 para hacerlas conscientes de la asimetría.
- Desarrollo de una implementación maleable completamente funcional de los kernels BLAS, basada en BLIS, y su correspondiente adaptación para arquitecutas AMP.
- Creación de interfaces que faciliten la utilización de la maleabilidad a nivel de thread.
- Extensión de la aproximación de maleabilidad a nivel de thread a un escenario muti-tarea a través de la integración de esta técnica en runtimes que permiten el paralelismo a nivel de tarea, como por ejemplo OmpSs [48].

Acronyms

ACML AMD Core Math Library. 5

AMP asymmetric multicore processor. ix, 9, 15, 20, 47, 52, 71, 75, 99–103, 105, 106, 109–114, 116

API Application Programming Interface. 12

BDP block-data parallelism. 52

BLAS Basic Linear Algebra Subprograms. 5, 6, 15, 17, 38, 71

BLIS BLAS-like Library Instantiation Software Framework. 7, 8, 15–17, 19–21, 23–27, 29, 30, 32, 34, 36–38, 40, 46, 47

CAP Computación de Altas Prestaciones. 116

CPU Central Processing Unit. xvii

DLA Dense Linear Algebra. 4, 5, 11, 13, 52, 53, 99–103, 106, 109, 110

ET Early Termination. 101, 112

FLOPS floating-point arithmetic operations per second. xvii

GFLOPS billions of floating-point arithmetic operations per second. 8, 21, 33, 45, 47, 85, 88, 91

GFLOPS/W GFLOPS per Watt. 21, 29, 47

GPU Graphics Processing Unit. 9

HPC High Performance Computing. xvii, xviii, 108

ISA instruction set architecture. 9

LAPACK Linear Algebra PACKage. vii, 4, 5, 8, 47, 48, 52, 79, 101, 111

LLS linear least squares. 8

MFLOPS/W MFLOPS per Watt. xvii

MTL malleable thread level. 59, 63, 71

s.p.d. symmetric positive definite. 47

SIMD Single Instruction Multiple Data. 10, 16, 23

SoC systems-on-chip. 9, 15, 26, 47

TP task-parallel. 54, 55

TSOR Two-Sided Orthogonal Reductions. 99, 102, 105, 109, 113, 114

WS worker sharing. 58, 101, 112

Bibliography

[1] Green500. https://www.top500.org/green500/.

[2] The international technology roadmap for semiconductors. http://www.itrs.net/reports.html.

[3] Mont-blanc project. http://www.montblanc-project.eu/.

[4] Top500. https://www.top500.org/.

[5] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series* (2009), vol. 180, IOP Publishing, p. 012037.

[6] Albers, S. Energy-efficient algorithms. *Communications of the ACM 53*, 5 (2010), 86–96.

[7] Aliaga, J. I., Catalán, S., Chalios, C., Nikolopoulos, D. S., and Quintana-Ortí, E. S. Performance and fault tolerance of preconditioned iterative solvers on low-power arm architectures. In *PARCO* (2015), pp. 711–720.

[8] Alonso, P., Catalan, S., Herrero, J. R., Quintana-Ortí, E. S., and Rodríguez-Sánchez, R. Reduction to tridiagonal form for symmetric eigenproblems on asymmetric multicore processors. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores* (2017), ACM, pp. 39–47.

[9] Alonso, P., Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Time and energy modeling of high–performance level-3 blas on x86 architectures. *Simulation Modelling Practice and Theory 55* (2015), 77–94.

[10] Alonso, P., Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Time and energy modeling of high-performance level-3 BLAS on x86 architectures. *Simulation Modelling Practice and Theory 55* (2015), 77 – 94.

[11] Anderson, E., Bai, Z., Bischof, C., Blackford, L., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, 1999.

[12] ARM. Neon. `https://developer.arm.com/technologies/neon`.

[13] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., et al. The landscape of parallel computing research: A view from berkeley. Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[14] Barrachina, S., Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. An integrated framework for power-performance analysis of parallel scientific workloads. *Energy* (2013), 114–119.

[15] Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Tracing the power and energy consumption of the QR factorization on multicore processors. In *12th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)* (2012), pp. 134–142.

[16] Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Automatic detection of power bottlenecks in parallel scientific applications. *Computer Science-Research and Development 29*, 3-4 (2014), 221–229.

[17] Barroso, L. A. The price of performance. *Queue 3*, 7 (Sept. 2005), 48–53.

[18] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep 15* (2008).

[19] Bischof, C. H., Lang, B., and Sun, X. A framework for symmetric band reduction. *ACM Trans. Math. Soft. 26*, 4 (2000), 581–601.

[20] Castaldo, A. M., Whaley, R. C., and Samuel, S. Scaling LAPACK panel operations using parallel cache assignment. *ACM Trans. Math. Soft. 39*, 4 (July 2013), 22:1–22:30.

[21] Castaño, M. A., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Reducing the cost of power monitoring with dc wattmeters. *Computer Science-Research and Development 30*, 2 (2015), 107–114.

[22] Catalán, S., Domínguez, J. G., Mayo, R., and Ortí, E. S. Q. Analyzing the energy efficiency of the memory subsystem in multicore processors. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on* (2014), IEEE, pp. 10–17.

[23] Catalán, S., Ezzatti, P., Quintana-Ortí, E. S., and Remón, A. The impact of panel factorization on the gauss-huard algorithm for the solution of linear systems on modern architectures. In *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2016, pp. 405–416.

[24] Catalán, S., Herrero, J. R., Igual, F. D., Rodríguez-Sánchez, R., Quintana-Ortí, E. S., and Adeniyi-Jones, C. Multi-threaded dense linear algebra libraries for low-power asymmetric multicore processors. *Journal of Computational Science* (2016).

[25] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., and Rodríguez-Sánchez, R. Energy balance between voltage-frequency scaling and resilience for linear algebra routines on low-power multicore architectures. *Parallel Computing* (2017).

[26] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R., and van de Geijn, R. A. A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting. *CoRR abs/1611.06365* (2016).

[27] Catalán, S., Igual, F., Rodríguez-Sánchez, R., and Quintana-Ortí, E. Time and energy modeling of high performance multi-threaded matrix multiplication. In *15th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)* (2015), pp. 311–316.

[28] Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. *Cluster Computing 19*, 3 (2016), 1037–1051.

[29] Catalán, S., Igual, F. D., Mayo, R., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Time and energy modeling of a high-performance multi-threaded cholesky factorization. *The Journal of Supercomputing 73*, 1 (2017), 139–151.

[30] Catalán, S., Malossi, A. C. I., Bekas, C., and Quintana-Ortí, E. S. The impact of voltage-frequency scaling for the matrix-vector product on the ibm power8. In *European Conference on Parallel Processing* (2016), Springer International Publishing, pp. 103–116.

[31] Catalán, S., Rodríguez-Sánchez, R., Quintana-Ortí, E. S., and Herrero, J. R. Static versus dynamic task scheduling of the lu factorization on arm big. little architectures. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International* (2017), IEEE, pp. 733–742.

[32] Center, B. S. Marenostrum 4 begins operation. `https://www.bsc.es/news/bsc-news/marenostrum-4-begins-operation`.

[33] Chalios, C., Catalán, S., Quintana-Orti, E., and Nikolopoulos, D. Evaluating asymmetric multicore systems-on-chip and the cost of fault tolerance using iso-metrics. *IET Computers & Digital Techniques 10*, 2.

[34] Charles, J., Sawyer, W., Dolz, M. F., and Catalán, S. Evaluating the performance and energy efficiency of the cosmo-art model system. *Computer Science-Research and Development 30*, 2 (2015), 177–186.

[35] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proc. 29th ACM Int. Conf. on Supercomputing* (2015), ICS'15, pp. 329–338.

[36] Costero, L., Igual, F. D., Olcoz, K., Catalán, S., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. 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.

[37] Costero, L., Igual, F. D., Olcoz, K., Catalán, S., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Revisiting conventional task schedulers to exploit asymmetry in multi-core architectures for dense linear algebra operations. *Parallel Computing* (2017).

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

[39] Demmel, J., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., and Sorensen, D. Prospectus for the development of a linear algebra library for high-performance computers. In *MATHEMATICS AND COMPUTER SCIENCE DIVISION REPORT ANL/MCS-TM-97, ARGONNE NATIONAL LABORATORY, ARGONNE, IL* (1987).

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

[41] Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Solving some mysteries in power monitoring of servers: Take care of your wattmeters! In *European Conference on Energy Efficiency in Large Scale Distributed Systems* (2013), Springer, Berlin, Heidelberg, pp. 3–18.

[42] Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Assessing power monitoring approaches for energy and power analysis of computers. *Sustainable Computing: Informatics and Systems 4*, 2 (2014), 68–82.

[43] Dolz, M. F., Kunkel, J., Chasapis, K., and Catalán, S. An analytical methodology to derive power models based on hardware and software metrics. *Computer Science-Research and Development 31*, 4 (2016), 165–174.

[44] Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., et al. The international exascale software project roadmap. *The international journal of high performance computing applications 25*, 1 (2011), 3–60.

[45] Dongarra, J., Moler, C., Bunch, J., and Stewart, G. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[46] Dongarra, J. J., Cruz, J. D., Hammarling, S., and Duff, I. S. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw. 16*, 1 (March 1990), 18–28.

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

[48] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 02 (2011), 173–193.

[49] Duranton, M., Yehia, S., De Sutter, B., De Bosschere, K., Cohen, A., Falsafi, B., Gaydadjiev, G., Katevenis, M., Maebe, J., Munk, H., et al. The hipeac vision. *Report, European Network of Excellence on High Performance and Embedded Architecture and Compilation 12* (2010).

[50] Earl C. Joseph, Steve Conway, C. I. G. C. C. M. N. M. A strategic agenda for european leadership in supercomputing: Hpc2020 - idc final report of the hpc study for the dg information society of the european commission. http://www.hpcuserforum.com/EU/downloads/SR03S10.15.2010.pdf.

[51] Feng, W.-c., Feng, X., and Ge, R. Green supercomputing comes of age. *IT professional 10*, 1 (2008).

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

[53] Fujitsu. Fujitsu hpc and the development of the post-k supercomputer.

[54] Garbow, B. S. Eispack – a package of matrix eigensystem routines. *Computer Physics Communications 7*, 4 (1974), 179 – 184.

[55] Golub, G., and Kahan, W. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis 2*, 2 (1965), 205–224.

[56] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, 1996.

[57] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (May 2008), 12:1–12:25.

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

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

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

[61] GWT-TUD GmbH. Vampir performance optimization. http://www.vampir.eu/.

[62] Hardkernel. Odroid-xu3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127.

[63] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Pub., San Francisco, 2012.

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

[65] Intel Corp. Intel 64 and ia-32 architectures software developer manual. volume 3b: System programming guide, part 2, 2015.

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

[67] Kabir, K., Haidar, A., Tomov, S., and Dongarra, J. On the design, development, and analysis of optimized matrix-vector multiplication routines for coprocessors. In *30th ISC High Performance* (2015), Springer, pp. 58–73.

[68] Kågström, B., Ling, P., and van Loan, C. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw. 24*, 3 (1998), 268–302.

[69] Koufaty, D., Reddy, D., and Hahn, S. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 125–138.

[70] Kunkel, J. HDTrace - a tracing and simulation environment of application and system interaction. Tech. Rep. 2, Department of Informatics, Scientific Computing. Universität Hamburg, 2011.

[71] Laub, A. Numerical linear algebra aspects of control design computations. *IEEE Transactions on Automatic Control 30*, 2 (1985), 97–108.

[72] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5*, 3 (Sept. 1979), 308–323.

[73] Low, T. M., Igual, F. D., Smith, T. M., , and Quintana-Ortí, E. S. Analytical modeling is enough for high performance BLIS. Tech. Rep. FLAWN #74, Department of Computer Sciences, The University of Texas at Austin, 2014. Available at `http://www.cs.utexas.edu/users/flame/`. Submitted to ACM Trans. Math. Softw.

[74] Ludwig, T. Editorial for the first international conference on energy-aware high performance computing. *Computer Science-Research and Development 25*, 3 (2010), 123–124.

[75] Martin, R. S., Reinsch, C., and Wilkinson, J. H. Householder's tridiagonalization of a symmetric matrix. *Numer. Math. 11*, 3 (March 1968), 181–195.

[76] Martin, R. S., and Wilkinson, J. H. Similarity reduction of a general matrix to hessenberg form. *Numer. Math. 12*, 5 (Dec. 1968), 349–368.

[77] Mittal, S. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv. 48*, 3 (Feb. 2016), 45:1–45:38.

[78] Moore, B. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE transactions on automatic control 26*, 1 (1981), 17–32.

[79] Netlib.org. `http://www.netlib.org/lapack`.

[80] NVIDIA. Nvidia reference manual, 2015. `http://docs.nvidia.com/deploy/nvml-api/index.html`.

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

[82] Paraver: the flexible analysis tool. `http://www.cepba.upc.es/paraver`.

[83] Peise, E., and Bientinesi, P. Performance modeling for dense linear algebra. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (Nov 2012), pp. 406–416.

[84] Peise, E., and Bientinesi, P. *A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels.* Springer International Publishing, Cham, 2015, pp. 245–258.

[85] Peise, E., Fabregat-Traver, D., and Bientinesi, P. *On the Performance Prediction of BLAS-based Tensor Contractions.* Springer International Publishing, Cham, 2015, pp. 193–212.

[86] Quintana-Ortí, G., and van de Geijn, R. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Softw. 32*, 2 (June 2006), 180–194.

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

[88] Shende, S. S., and Malony, A. D. The tau parallel performance system. *The International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

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

[90] Srinivasan, S., Zhao, L., Illikkal, R., and Iyer, R. Efficient interaction between os and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 62–72.

[91] StarPU project. `http://runtime.bordeaux.inria.fr/StarPU`.

[92] Strazdins, P. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[93] Van Zee, F. G. `libflame`*: The Complete Reference.* `www.lulu.com`, 2009.

[94] Van Zee, F. G., Smith, T. M., Marker, B., Low, T. M., van de Geijn, R. A., Igual, F. D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., and Killough, L. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft.* (2014). In print. Available at `http://www.cs.utexas.edu/users/flame`.

[95] Van Zee, F. G., and van de Geijn, R. A. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw. 41*, 3 (2015), 14:1–14:33.

[96] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *International Conference on Supercomputing (ICS)* (1998).

[97] Xianyi, Z., Qian, W., and Yunquan, Z. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems* (Dec 2012), pp. 684–691.