Architecture-Aware Configuration and Scheduling of Matrix Multiplication on Asymmetric Multicore Processors

Sandra Catalán[a], Francisco D. Igual[b], Rafael Mayo[a], Rafael Rodríguez-Sánchez[a], Enrique S. Quintana-Ortí[a],

[a]*Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain.*
[b]*Depto. de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Spain.*

Abstract

Asymmetric multicore processors (AMPs) have recently emerged as an appealing technology for severely energy-constrained environments, especially in mobile appliances where heterogeneity in applications is mainstream. In addition, given the growing interest for low-power high performance computing, this type of architectures is also being investigated as a means to improve the throughput-per-Watt of complex scientific applications.

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

Email addresses: catalans@uji.es (Sandra Catalán), figual@ucm.es (Francisco D. Igual), mayo@uji.es (Rafael Mayo), rarodrig@uji.es (Rafael Rodríguez-Sánchez), quintana@uji.es (Enrique S. Quintana-Ortí)

gains in performance with respect to its architecture-oblivious counterparts while exploiting all the resources of the AMP to deliver considerable energy efficiency.

Keywords: Matrix multiplication, asymmetric multicore processors, memory hierarchy, scheduling, multi-threading, high performance computing

1. Introduction

The decay of Dennard scaling [1] during the past decade marked the end of the "GHz race" and the shift towards multicore designs due to their more favorable performance-power ratio. In addition, the doubling of transistors on chip with each new semiconductor generation, dictated by Moore's law [2], has only exacerbated the *power wall* problem [3, 4, 5], leading to the arise of "dark silicon" [6] and the deployment of heterogeneous facilities for high performance computing [7, 8].

Asymmetric multicore processors (AMPs) are a particular class of heterogeneous architectures equipped with cores that share the same instruction set architecture[1] but differ in micro-architecture, and thus in complexity, performance, and power consumption. AMPs have received considerable attention in the last years as a means to improve the performance-power ratio of computing systems [9, 10, 11, 12] partly because, in theory, they can deliver much higher performance for the same power budget, mainly by exploiting the presence of serial and parallel phases within applications [11].

The *general matrix multiplication* (GEMM) is a crucial operation for the optimization of the Level-3 *Basic Linear Algebra Subprograms* (BLAS) [13], as portable and highly tuned versions of the remaining Level-3 kernels are in general built on top of GEMM [14]. In turn, the contents of BLAS conform a pivotal cornerstone upon which many sophisticated libraries to tackle complex scientific and engineering applications rely [15]. The importance of BLAS in general, and GEMM in particular, is illustrated by the prolonged efforts spent over the past decades to produce carefully tuned commercial libraries for almost any current architecture (e.g., Intel's MKL [16], AMD's

[1] According to this definition, servers equipped with one (or more) general-purpose multicore processor(s) and a PCIe-attached graphics accelerator, or systems-on-chip like the NVIDIA Tegra TK1, are excluded from this category.

ACML [17], IBM's ESSL [18], NVIDIA's CUBLAS [19], etc.) as well as the number of high quality open source solutions (e.g., GotoBLAS [20, 21], OpenBLAS [22], BLIS [23], and ATLAS [24]).

In this paper we propose efficient multi-threaded implementations of GEMM on an ARM big.LITTLE AMP consisting of a cluster composed of a few fast (big) cores and a complementary cluster with several slow (LITTLE) cores, shared main memory, and private L1/L2 caches per core/cluster, respectively. Our approach leverages the multi-threaded implementation of GEMM in the BLIS library, which decomposes the operation into a collection of nested loops around a *micro-kernel*. In this reference code, we modify the loop stride configuration and scheduling to distribute the micro-kernels comprised by certain loops among the big/LITTLE clusters and cores while taking into account the processor's computational power and cache organization. In more detail, this work makes the following specific contributions:

- Our optimized implementations modify 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.

- We integrate two alternative scheduling strategies, 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, we apply an orthogonal symmetric–static schedule to map the workload of one of the inner loops across the cores of the same cluster.

- We demonstrate the practical benefits of the cache-aware configurations and asymmetry-aware scheduling strategies on the Exynos 5422, a system-on-chip (SoC) consisting of an ARM Cortex-A15 quad core (big) cluster and an ARM Cortex-A7 quad core (LITTLE) cluster. Our experimental results show 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.

- We include an analysis of the energy efficiency of the asymmetric architecture when running our optimized GEMM, using the GFLOPS/W

(billions of floating-point arithmetic operations, or flops, per second and Watt) metric, which assesses the energy cost of flops.

To conclude, we emphasize that the scheduling approaches proposed in this paper are general and, in combination with the BLIS implementation of GEMM, can be ported with little effort to present and future instances of the ARM big.LITTLE architecture as well as to any other asymmetric design in general (e.g. the Intel QuickIA prototype [25]). Furthermore, we are confident that the principles underlying our scheduling decisions carry over to all Level-3 BLAS operations.

The rest of the paper is structured as follows. In Section 2, we compare our approach to optimize GEMM on AMPs with state-of-the-art works on similar architectures. In Section 3, we describe the mechanisms that underlie the original multi-threaded implementation of GEMM in the BLIS framework, and evaluate its performance and optimal cache parameter configuration for the Cortex-A15 and Cortex-A7 clusters. In Section 4, we investigate the effect of using standard, architecture-oblivious multi-threaded BLAS implementations on AMPs, and its negative impact on performance and energy efficiency. In Section 5, we introduce our strategies to adapt the original BLIS multi-threaded implementation to the asymmetric architecture, and report the performance and energy-efficiency results of the new codes. Finally, Section 6 closes the paper with a few concluding remarks and proposals for future work.

2. Related Work

Heterogeneous (and asymmetric) architectures are an active research topic, with a vast design space that needs careful consideration in terms of power, performance, programmability, and flexibility [26]. Many of these works can be grouped into *i)* efforts to experimentally evaluate the computational performance and/or power-energy efficiency of AMPs using multi-threaded benchmarks and applications; and *ii)* contributions related to workload-partitioning strategies for the execution of GEMM on heterogeneous platforms. In the first group, Winter et al. [12] discuss power management techniques and thread scheduling for AMPs; and scheduling on AMP architectures is explored in a number of works; see, among others, [27, 28, 29, 12] and the references therein.

In the second group, mapping GEMM in an heterogeneous cluster is analyzed in [30], while a theoretical study of dynamic scheduling applied to GEMM in a similar scenario is introduced in [31].

Compared with previous work, our investigation aims to deliver an implementation of GEMM, based on an open source BLAS library (BLIS), that is highly optimized for asymmetric ARM big.LITTLE architectures. All previous efforts to implement and evaluate GEMM on AMPs employ simple codes, at best tuned via very basic tiling techniques, and therefore yield suboptimal codes. The research with heterogeneous clusters targets a more general and complex problem, and in practice can hardly be expected to produce an optimal solution for AMPs.

3. Multi-Threaded Portable Implementation of BLIS GEMM

Modern high-performance implementations of GEMM for general-purpose architectures follow the design pioneered by GotoBLAS [20]. BLIS in particular implements the GEMM $C \mathrel{+}= A \cdot B$, 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 1). The macro-kernel is then implemented in terms of two additional loops around a *micro-kernel* (Loops 4 and 5 in Figure 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; see [23] for further details.

Figure 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, m_c, n_r and m_r (which dictate the strides of the five outermost loops) are adjusted taking into account the latency of the floating-point units (FPUs), number of vector registers, and size/associativity degree of the cache levels. The goal is that 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 are streamed 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). By appropriately choosing the configuration parameters, these transfers are fully amortized with enough computation from within the micro-kernel; see [32].

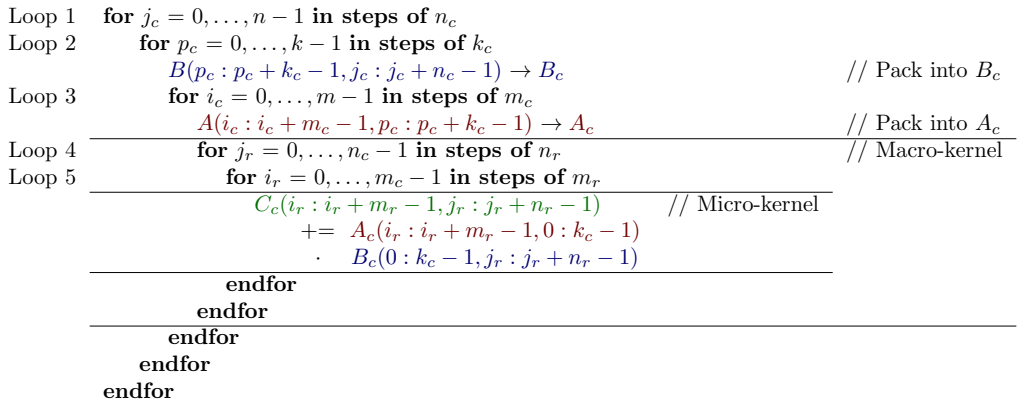

Figure 1: High performance implementation of GEMM in BLIS. In the code, $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is just a notation artifact, introduced to ease the presentation of the algorithm, while A_c, B_c correspond to actual buffers that are involved in data copies.

3.1. Multi-threaded GEMM *in BLIS*

BLIS allows to select, at execution time, which of the five loops of GEMM are parallelized. Several loops can be simultaneously executed in parallel in order to adapt the execution to specific properties of the target architecture. By default, when one of the loops is parallelized, a static partitioning and mapping of iteration chunks to threads is performed prior to the execution of the loop.

The multi-threaded version of GEMM integrated in BLIS has been previously analyzed for conventional symmetric multicore processors (SMPs) [33] and modern many-threaded architectures [34]. In both "types" of architectures, the parallel implementations exploit the concurrency available in the nested five–loop organization of GEMM 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 into race conditions as well as loops with options that exhibit too-fine granularity. The insights gained from these analyses [33, 34] about the loop(s) to parallelize 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 parallelism in this

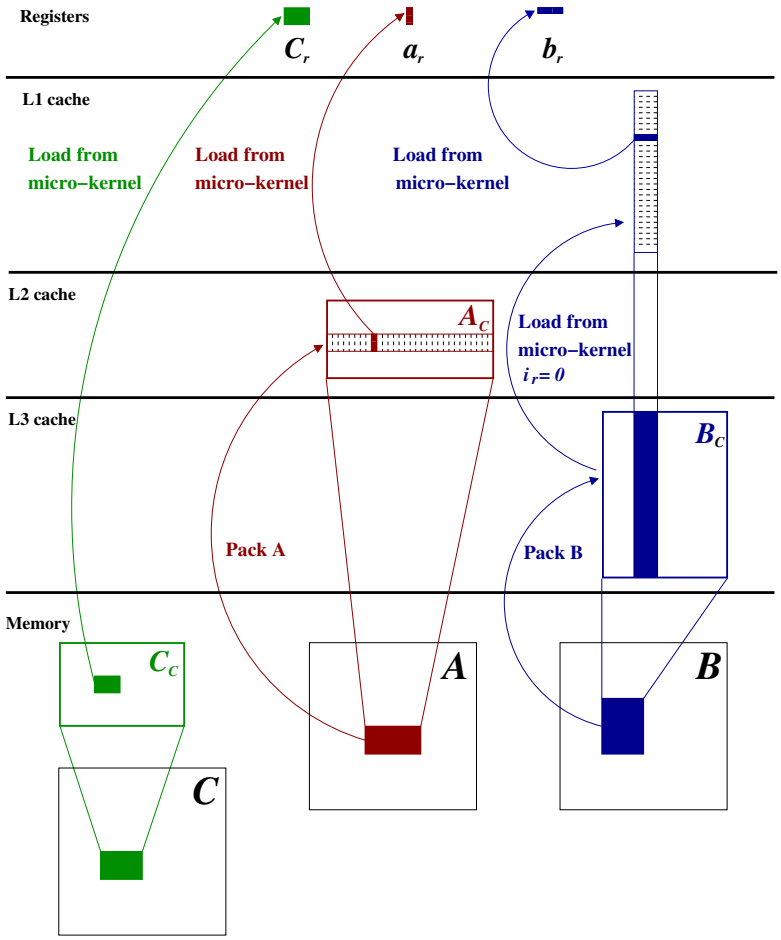

Figure 2: Data movement involved in the BLIS implementation of GEMM.

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.

- 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.

- 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. 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 sum up, these are general guidelines to decide which loops are theoretically 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. Experimental setup

The ODROID-XU3 board employed in our experiments contains a Samsung Exynos 5422 SoC with an ARM Cortex-A15 quad-core processing cluster (running at 1.6 GHz in our setup) and a Cortex-A7 quad-core processing cluster (running at 1.4 GHz). Both clusters access a shared DDR3 RAM (2 Gbytes) via 128-bit coherent bus interfaces. Each ARM core (either Cortex-A15 or Cortex-A7) has a 32+32-Kbyte L1 (instruction+data) cache. The four ARM Cortex-A15 cores share a 2-Mbyte L2 cache, while the four ARM Cortex-A7 cores share a smaller 512-Kbyte L2 cache; see Figure 3. All

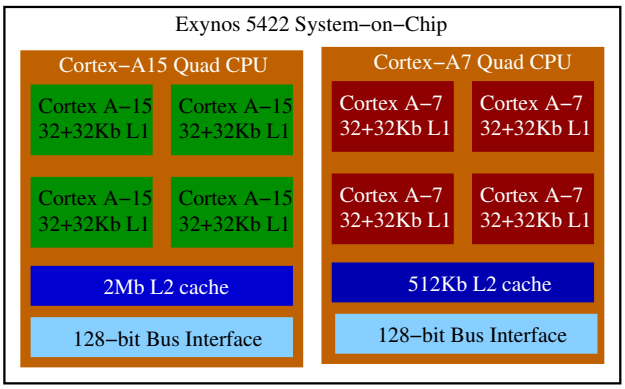

Figure 3: Exynos 5422 block diagram.

our tests hereafter employ IEEE double-precision arithmetic and square matrices of order $r = m = n = k$. We ensure that the cores run at their highest frequency by setting the Linux *performance* governor with the appropriate frequency limits. Codes are instrumented with the `pmlib` [35] framework, which collects power consumption data corresponding to instantaneous power readings from four independent sensors in the board (for the Cortex-A7 cores, Cortex-A15 cores, DRAM and GPU), with a sampling rate of 250 ms.

3.3. Cache optimization for the big and LITTLE cores

An initial step in order to attain high performance with the implementation of BLIS GEMM is, given a target precision (single or double), 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. We next describe our experimental effort towards this goal. A recent study [36] shows that, in principle, this optimization is also possible via analytic derivation.

The first aspect to note is that, in this architecture, 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, we performed independent empirical searches using

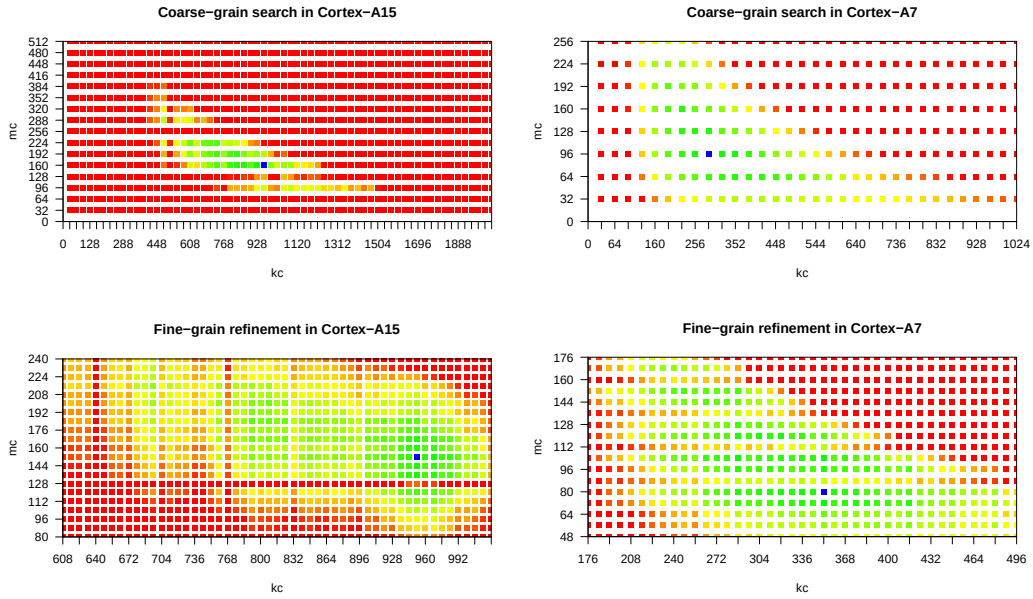

Figure 4: BLIS optimal cache configuration parameters m_c and k_c for the ARM Cortex-A15 (left) and Cortex-A7 (right) in the Samsung Exynos 5422 SoC. The performance ranges from red (lowest GFLOPS) to green (highest GFLOPS); the optimal (m_c, k_c) pair is marked as a blue dot.

a single Cortex-A15 core and a single Cortex-A7 core. In both cases, we initially applied a coarse-grain search to detect potential optimal regions, 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 4, where the top and bottom plots correspond to the coarse search and the fine-grain refinement respectively. Performance is measured hereafter in terms of GFLOPS.

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 associativy-degree of the caches allow 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.

3.4. Multi-threaded BLIS performance 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 summarized in Section 3.1, 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. We note that, although the two types of clusters are evaluated in isolation in this section, the performance GFLOPS figures will be of interest for the asymmetric-aware versions of GEMM that will be presented in Sections 4 and 5, as their aggregation can be considered as an *ideal scenario* for the peak performance that can be extracted from the complete asymmetric SoC.

The plots in Figure 5 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. We note that, when calculating the energy efficiency of one type of cluster, the energy consumed by the complementary (idle) cluster is also accounted for, so that we are reporting the energy efficiency of the complete SoC.

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, though the utilization of the fourth core yields a smaller increase, of an additional 1.4 GFLOPS. 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-A15 offers the best results in terms of GFLOPS/W. However, the benefits of increasing the number of threads are less significant when compared with those obtained with the Cortex-A7 cores. Concretely, the energy efficiency attained with the complete Cortex-A7 cluster is twice that observed with a single core of the same type. In contrast, the best energy efficiency for the Cortex-A15 is only 33% higher than that measured with a single Cortex-A15 core. Moreover, due to the non-linear increase in performance when adding the fourth Cortex-A15 core, the most energy-efficient solution is obtained with three cores instead

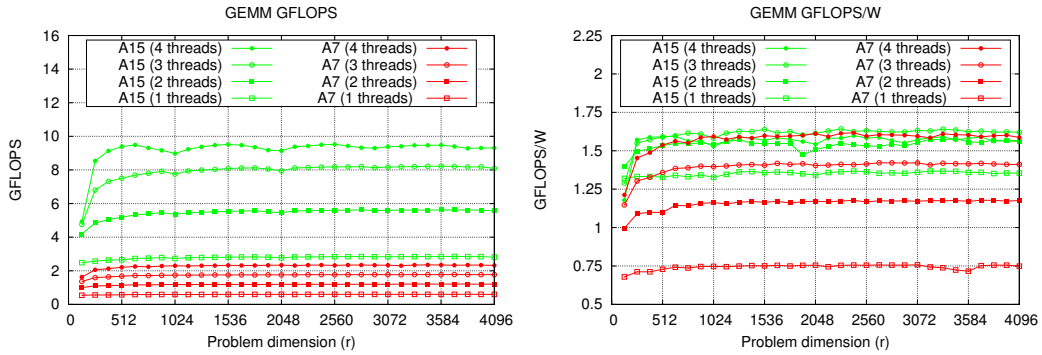

Figure 5: Performance (left) and energy efficiency (right) of the BLIS GEMM using exclusively one type of core, for a varying number of threads.

of the complete cluster. It is also worth emphasizing that the exploitation of four Cortex-A7 cores delivers significantly higher energy efficiency than an alternative that leverages a single Cortex-A15 core, though the overall performance of the former option is slightly worse.

In general, these graphs reveal that the performance achieved by the complete Cortex-A15 cluster is roughly four times that of the Cortex-A7 cluster but their energy efficiency is similar. This last observation is interesting since, a priori, one could expect that the Cortex-A7 cluster was more energy-efficient than the Cortex-A15 cluster. However, we would like to remark that all our experiments report the energy efficiency of the complete SoC, and that the Cortex-A15 cluster in idle state already dissipates more power than a single Cortex-A7 core in execution.

4. Architecture-Oblivious BLIS GEMM on the big.LITTLE SoC

The default approach adopted by BLIS to map GEMM on a multi-threaded CPU (see Section 3.1) presents two main drawbacks when applied to simultaneously 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 we face a system with two different architectures (Cortex-A15 and Cortex-A7), and thus different optimal cache parameters, we would ideally need to use different loop strides/configuration parameters for each type of core.

The following experiment is designed to expose the negative impact of these two mismatches between the BLIS approach and the Exynos 5422 SoC on the performance and energy behavior of GEMM. For the evaluation, given the guidelines in Section 3.1 and the lack of an L3 cache in this chip, we adopt the following two-level parallelization strategy:

- 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 up to *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. We 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 6 illustrates the implications of the previous scheduling strategy in terms of loop partitioning and assignment to threads. In total, eight threads are created and binded to the cores so that we are extracting in overall *8-way parallelism* within BLIS. Note how the iteration space for all loops is homogeneously distributed across the cores (i.e., without taking into account the core type).

Figure 7 reports the performance and energy efficiency using the (two-level) symmetric-static scheduling (SSS) that parallelizes loops 1 and 4. For

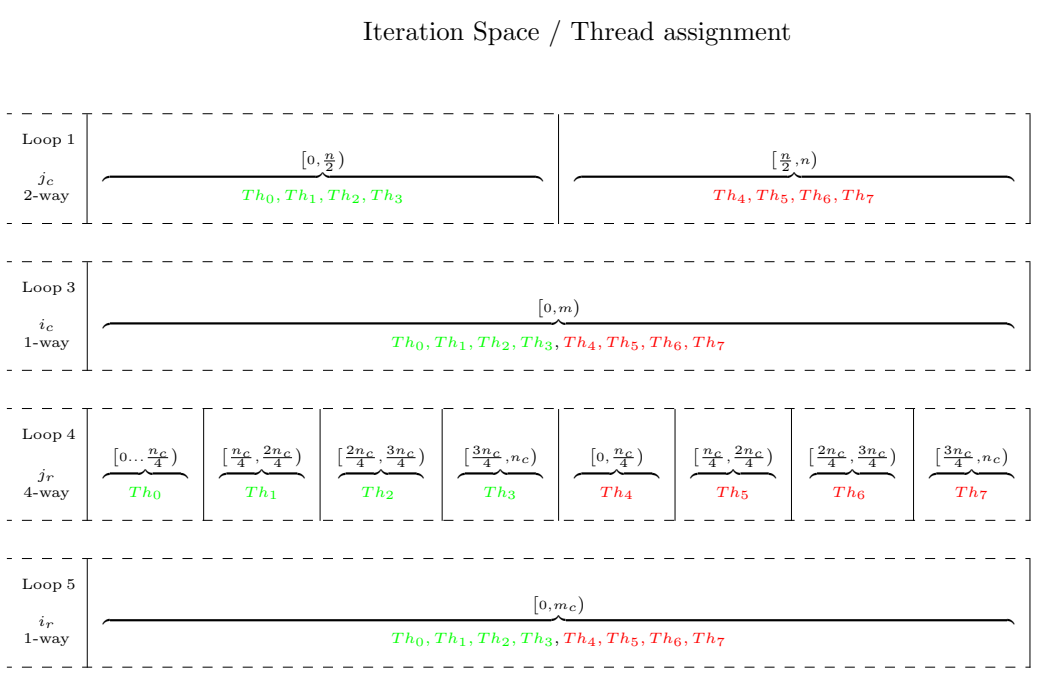

Figure 6: Partitioning of the iteration space and assignment to threads/cores for a multi-threaded BLIS implementation with 8-way parallelism that combines 2-way parallelism from Loop 1 and 4-way parallelism from Loop 4. Threads in green and red are respectively mapped to big and LITTLE cores.

reference, we also include the results from the 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 (see Section 3). 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). This is a theoretical upper bound for the performance 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 either the differences in the cache hierarchy 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

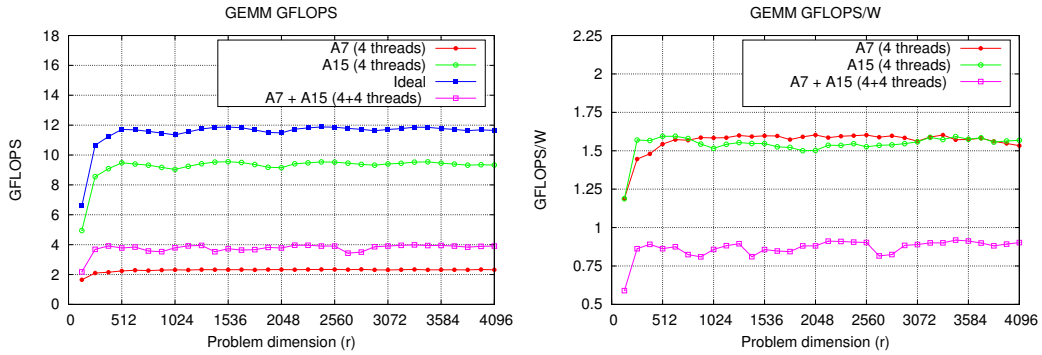

Figure 7: Performance (left) and energy efficiency (right) of the reference BLIS GEMM using exclusively one type of core in isolation, and the SSS version with a coarse-grain parallelization of Loop 1 and the fine-grain parallelization of Loop 4 using 4 threads per cluster.

space to the processing cores in a homogeneous manner. This causes a severe workload imbalance, as the threads running 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 achieves the worst energy results. In conclusion, this experiment naturally motivates the need of an efficient alternative to the homogeneous SSS partitioning of the iteration space integrated in the original multi-threaded implementation of BLIS GEMM.

5. Architecture-Aware Optimization of BLIS GEMM on the big.LITTLE SoC

In this section, we briefly review the control mechanism that governs the parallelization of BLIS GEMM. Next, we propose and integrate two asymmetry-aware strategies for workload scheduling of the BLIS GEMM micro-kernels as well as a cache-aware configuration for AMPs; and we evaluate the impact of these techniques on performance and energy efficiency. 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 based on the performance difference 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.

5.1. BLIS internals

The execution of all BLIS routines, including GEMM, is commanded by a *control-tree*. This is a recursive data structure that encodes all the information necessary to combine the basic building blocks offered by the BLIS framework in order to implement high-performance algorithms for virtually any BLAS-like operation. The control tree for a given BLAS-3 operation governs, among others, which combination of loops are to be executed to complete the operation (that is, the exact *algorithmic variant* to execute at each level of the general algorithm), the loop stride for each loop (specific to each target architecture), and the exact points at which packing must occur. In addition, for multi-threaded BLIS implementations, the control tree defines which loops need to be parallelized and the level of concurrency to extract at each point of the algorithm.

A key property of the *control trees* is that they can be leveraged to modify these parameters without affecting the rest of the BLAS implementation, boosting programmer's productivity and enhancing flexibility. In our modifications of the BLIS framework, we have exploited this abstraction mechanism in order to encode the differences between the original framework and our versions adapted for AMPs. In particular, we next focus on the necessary modifications and requirements to implement an asymmetric scheduling of the loop iteration space to fast and slow cores, and the modification of the loop strides in order to develop a cache-aware configuration for BLIS GEMM.

5.2. Static-asymmetric scheduling (SAS)

Taking into account the experiment in Section 4, we have revamped the original multi-threaded implementation of BLIS GEMM 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 one of the loops 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 assigned according to the capabilities of each type of core.

Our reimplementation also comprises, as an configuration knob, an interface to specify the *ratio* of performance between big and LITTLE cores. For the specific loop that is selected as 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 specific big.LITTLE setups (for example, to changes in the core frequency that affect 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.

5.2.1. Mapping the iteration space to clusters and cores

Given the memory organization of the Exynos 5422 SoC, and the guidelines given for the parallelization of BLIS GEMM in section 3, we evaluated the following parallelization options for SAS:

- Coarse-grain: the micro-kernels of the complete multiplication are distributed among the Cortex-A15 and Cortex-A7 clusters by parallelizing either Loop 1 or Loop 3, with a different number of iterations of the parallelized loop assigned to each cluster (*2-way parallelism*).

- Fine-grain: the execution of each macro-kernel is partitioned among the cores of the same type by parallelizing Loop 4, Loop 5, or both (*4-way parallelism*).

To illustrate this, Figure 8 depicts the distribution of the iteration space across fast and slow threads for an scenario in which the iteration space of Loop 1 is asymmetrically distributed across fast and slow threads, using a

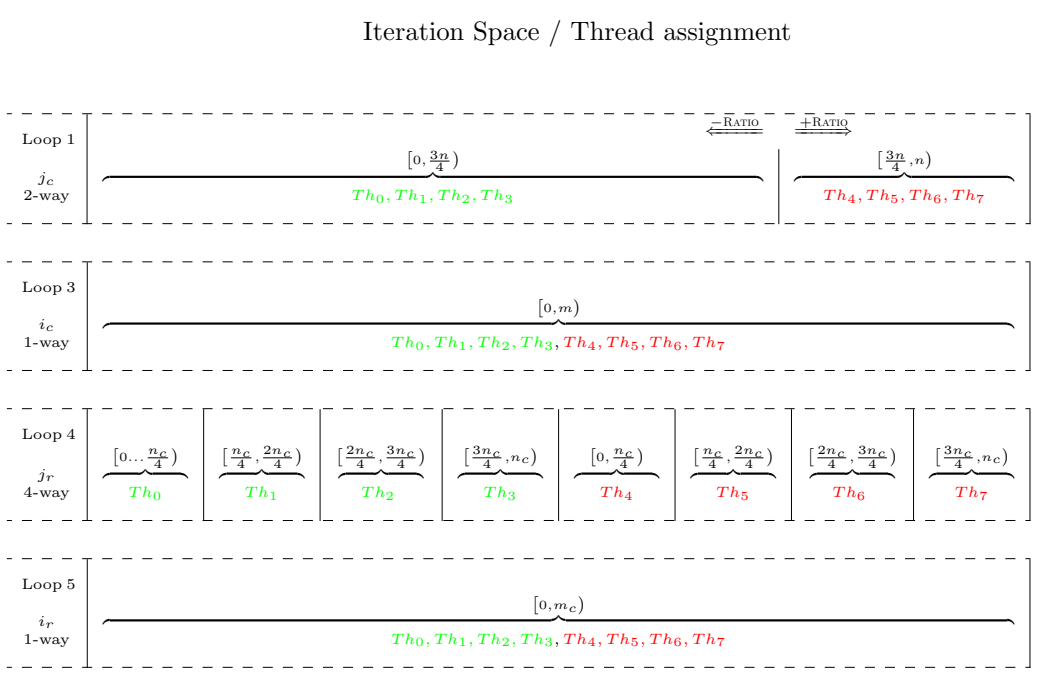

Figure 8: Partitioning of the iteration space and assignment to threads/cores for a multi-threaded BLIS implementation with 8-way parallelism that asymmetrically combines 2-way parallelism from Loop 1 (using a ratio between fast and slow cores of 3) and 4-way parallelism from Loop 4.

ratio 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 among the cores of the same cluster.

5.2.2. Evaluation of SAS

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. For brevity, because the qualitative conclusions that can be extracted from these parallelization strategies are very similar, we only report 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, using (distribution) ratios for the coarse-grain parallelization that range from 1 to 7.

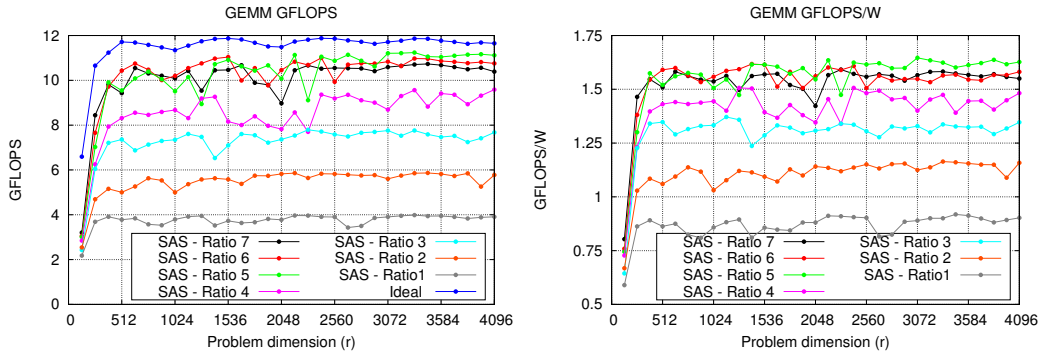

Figure 9: Performance (left) and energy-efficiency (right) of the SAS version of BLIS GEMM with a coarse-grain parallelization of Loop 1 and a fine-grain parallelization of Loop 4 using 4 threads per cluster.

Figure 9 displays the results for this experiment. The performance results show that, when the appropriate workload distribution is applied, the asymmetric-aware SAS outperforms the peak performance of all other configurations, being close to that of the ideal case. In particular, the left-hand side graph reveals that the worst performance is achieved when the ratio is 1 (i.e., an homogeneous inter-cluster prallelization). Also, the performance grows until a ratio of 5–6 is used, and above 6, in general declines with a lower limit existing at the performance line delivered by the Cortex-A15 cluster in isolation (not included in the figure for clarity). These results indicate that ratios below 5 map that too much workload to the Cortex-A7 cluster, and ratios above 6 assign an excessive workload to the Cortex-A15 cluster.

For the largest tested problem, 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 the same flops per Joule as the setup that exclusively employs the Cortex-A15 cluster. 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, till the slow threads complete their work.

5.3. 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 using 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 (see Section 3). 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, the performance of GEMM is quite independent of n_c, since there is not a 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, 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 the actual Cortex-A7-specific optimal parameters. However, it is still higher than that obtained with the cache parameters for the Cortex-A15 as, with those much larger values, the memory buffer A_c does not fit into the small L2 cache of the Cortex-A7.

5.3.1. Comparison of SAS and CA-SAS

The combination of the coarse-grain and fine-grain parallelization strategies described in Section 5.2.1 yields the same parallelization options for CA-SAS. For the same reasons, we only report next the results corresponding

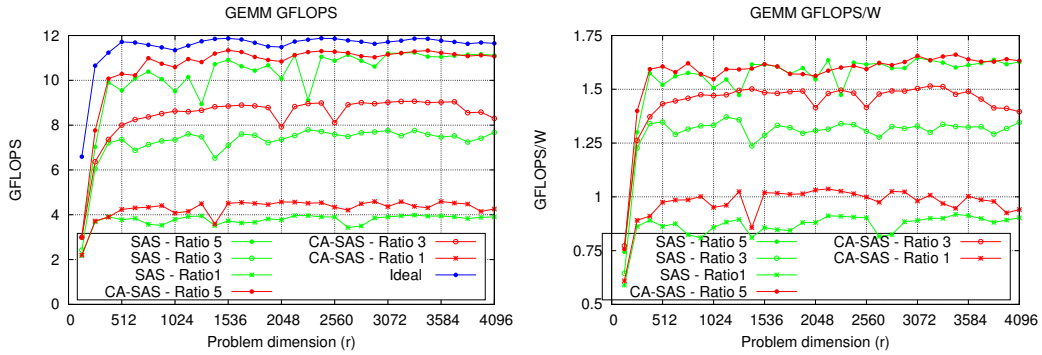

Figure 10: Performance (left) and energy-efficiency (right) of the SAS and CA-SAS versions of BLIS GEMM with a coarse-grain parallelization of Loop 1 and a fine-grain parallelization of Loop 4 using 4 threads per cluster.

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 (distribution) 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 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. We notice 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 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,

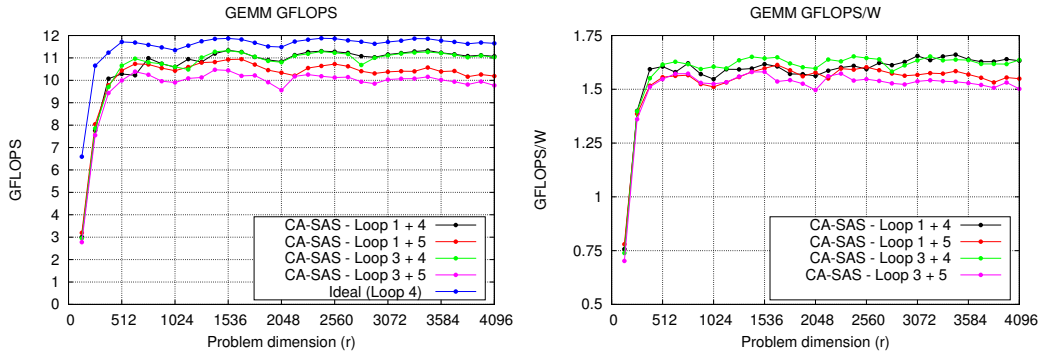

Figure 11: Performance (left) and energy-efficiency (right) of the CA-SAS version of BLIS GEMM with a coarse-grain parallelization of either Loop 1 or Loop 3; combined with a fine-grain parallelization of either Loop 4 or Loop 5, using a ratio 5 in both cases and 4 threads per cluster.

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 set 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.

5.4. Cache-aware dynamic-asymmetric scheduling (CA-DAS)

Our 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, though the target loop this feature is applied to still needs to be chosen with care.

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 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 binded 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.

5.4.1. Evaluation of CA-DAS

This last round of experiments presents a more reduced number of options, since Loop 1 was identified as a poor choice to dynamically distributing 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 Loop 4 or in Loop 5, using either two control-trees (one for "fast" and one for "slow" threads, CA-DAS) or a single control-tree for both types of threads (DAS). 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 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. If the fine-grain parallelization 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 is lower for Loop 5 than for Loop 4 (see Figure 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.

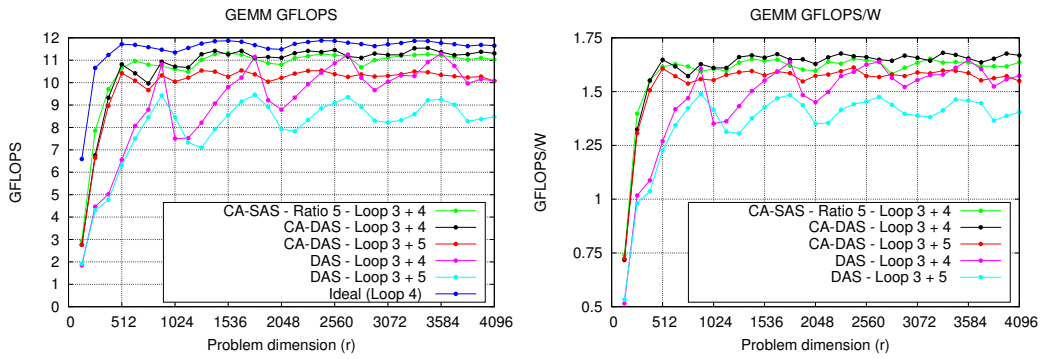

Figure 12: Performance (left) and energy-efficiency (right) of the CA-DAS and DAS versions of BLIS GEMM with a coarse-grain parallelization of Loop 3 and a fine-grain parallelization of either Loop 4 or Loop 5, using 4 threads per cluster in both cases.

6. Conclusions

We have proposed and evaluated several mechanisms to efficiently map the framework for matrix multiplication integrated in the BLIS library to an asymmetric ARM big.LITTLE (Cortex A15+A7) SoC. Our results reveal excellent improvements in performance compared with a homogeneous implementation that operates 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.

This is an important step towards a full BLAS implementation optimized for big.LITTLE architectures, which is a future goal in our research effort. While we believe that the approach applied to GEMM carries over to the rest of the BLAS, there are a number of issues that need to be addressed to further increase performance and adaption to other (present and future) asymmetric architectures. Among others, the most relevant factor is the adoption of different micro-kernels, tuned to each type of core, in order to extract the maximum performance for those asymmetric architectures. A

port to a 64-bit ARMv8 architecture, and an experimental study on architectures with different number of big/LITTLE cores are also key milestones in our roadmap.

Acknowledgments

The researchers from Universitat Jaume I were supported by project CICYT TIN2011-23283 of MINECO and FEDER, the EU project FP7 318793 "EXA2GREEN" and the FPU program of MECD. The researcher from Universidad Complutense de Madrid was supported by project CICYT TIN2012-32180.

References

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

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

[3] M. Duranton *et al*, The HiPEAC vision for advanced computing in horizon 2020, `http://www.hipeac.net/roadmap` (2013).

[4] J. F. Lavignon *et al*, ETP4HPC strategic research agenda achieving HPC leadership in Europe.

[5] R. Lucas *et al*, Top ten Exascale research challenges, `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf` (2014).

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in: Proc. 38th Annual Int. Symp. on Computer architecture, ISCA'11, 2011, pp. 365–376.

[7] The TOP500 list (2015).
URL `http://www.top500.org`

[8] The GREEN500 list (2015).
URL `http://www.green500.org`

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

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

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

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

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

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

[15] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, University of California at Berkeley, Electrical Engineering and Computer Sciences (2006).

[16] Intel Corp., Intel math kernel library (MKL) 11.0, `http://software.intel.com/en-us/intel-mkl` (2015).

[17] AMD, AMD Core Math Library, `http://developer.amd.com/tools/cpu/acml/pages/default.aspx` (2015).

[18] IBM, Engineering and Scientific Subroutine Library, `http://www.ibm.com/systems/software/essl/` (2015).

[19] NVIDIA, CUDA basic linear algebra subprograms, https://developer.nvidia.com/cuBLAS (2015).

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

[21] K. Goto, R. van de Geijn, High performance implementation of the level-3 BLAS, ACM Trans. Math. Softw. 35 (1) (2008) 4:1–4:14.
URL http://doi.acm.org/10.1145/1377603.1377607

[22] OpenBLAS, http://xianyi.github.com/OpenBLAS/ (2015).

[23] F. G. Van Zee, R. A. van de Geijn, BLIS: A framework for generating blas-like libraries, ACM Trans. Math. Soft.To appear.

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

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

[26] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, R. Iyer, Quickia: Exploring heterogeneous architectures on real prototypes, in: Proc. IEEE 18th Int. Symp. on High-Performance Computer Architecture, HPCA'12, 2012, pp. 1–8.

[27] J. Hourd, C. Fan, J. Zeng, Q. S. Zhang, M. J. Best, A. Fedorova, C. Mustard, Exploring practical benefits of asymmetric multicore processors, in: 2nd Workshop on Parallel Execution of Sequential Programs on Multicore Architectures, PESPMA 2009.

[28] N. B. Lakshminarayana, J. Lee, H. Kim, Age based scheduling for asymmetric multiprocessors, in: Proc. Conference on High Performance Computing Networking, Storage and Analysis, SC'09, 2009, pp. 25:1–25:12.

[29] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing, ACM Trans. Des. Autom. Electron. Syst. 18 (1) (2013) 5:1–5:23.

[30] D. Clarke, A. Lastovetsky, V. Rychkov, Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models, in: Euro-Par 2011: Parallel Processing Workshops, Vol. 7155 of LNCS, 2012, pp. 450–459.

[31] O. Beaumont, L. Marchal, Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms, in: Proc. 23rd Int. Symp. High-performance Parallel and Distributed Computing, HPDC'14, 2014, pp. 141–152.

[32] T. M. Low, F. D. Igual, T. M. Smith, E. S. Quintana-Ortí, Analytical modeling is enough for high performance BLIS, ACM Trans. Math. Soft.In review. Available at `http://www.cs.utexas.edu/users/flame`.

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

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

[35] P. Alonso, R. M. Badia, J. Labarta, M. Barreda, M. F. Dolz, R. Mayo, E. S. Quintana-Ortí, R. Reyes, Tools for power-energy modelling and analysis of parallel scientific applications, in: 41st Int. Conf. on Parallel Processing – ICPP, 2012, pp. 420–429.

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