Revisiting Conventional Task Schedulers to Exploit Asymmetry in ARM big.LITTLE Architectures for Dense Linear Algebra

Luis Costero[a], Francisco D. Igual[a,*], Katzalin Olcoz[a], Enrique S. Quintana-Ortí[b]

[a] *Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Madrid, Spain*

[b] *Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain*

Abstract

Dealing with asymmetry in the architecture opens a plethora of questions from the perspective of scheduling task-parallel applications, and there exist early attempts to address this problem via *ad-hoc* strategies embedded into a runtime framework. In this paper we take a different path, which consists in addressing 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 is not only possible but 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.

Keywords: Dense linear algebra, Task parallelism, Runtime task schedulers, Asymmetric architectures

1. Introduction

The end of Dennard scaling has promoted heterogeneous systems into a mainstream approach to leverage the steady growth of transistors on chip dictated by Moore's law [1, 2]. ARM® big.LITTLE™ processors are a particular class of heterogeneous architectures that combine two types of multicore clusters, consisting of a few high performance (big) cores and a collection of low power (LITTLE) cores. These *asymmetric multicore processors* (AMPs) can in theory deliver much higher performance for the same power budget. Furthermore, compared with multicore servers equipped with graphics processing units (GPUs), NVIDIA's Tegra chips and AMD's APUs, ARM big.LITTLE processors differ in that the cores in these systems-on-chip (SoC) share the same instruction set architecture and a strongly coupled memory subsystem.

*Corresponding author

Email addresses: lcostero@ucm.es (Luis Costero), figual@ucm.es (Francisco D. Igual), katzalin@ucm.es (Katzalin Olcoz), quintana@ucm.es (Enrique S. Quintana-Ortí)

Task parallelism has been reported as an efficient means to tackle the considerable number of cores in current processors. Several efforts, pioneered by Cilk [3], aim to ease the development and improve the performance of task-parallel programs by embedding task scheduling inside a *runtime* (framework). The benefits of this approach for complex dense linear algebra (DLA) operations have been demonstrated, among others, by OmpSs [4], StarPU [5], PLASMA/MAGMA [6, 7], Kaapi [8], and `libflame` [9]. In general, the runtimes underlying these tools decompose DLA routines into a collection of numerical kernels (or tasks), and then take into account the dependencies between the tasks in order to correctly issue their execution to the system cores. The tasks are therefore the "indivisible" scheduling unit while the cores constitute the basic computational resources.

In this paper we introduce an efficient approach to execute task parallel DLA routines on AMPs via conventional asymmetry-oblivious schedulers. Our conceptual solution aggregates the cores of the AMP into a number of *symmetric virtual cores* (VCs) which become the only basic computational resources that are visible to the runtime scheduler. In addition, an specialized implementation of each type of task, from an asymmetry-aware DLA library, partitions each numerical kernel into a series of finer-grain computations, which are efficiently executed by the asymmetric aggregation of cores of a single VC. Our work thus makes the following specific contributions:

- We target in our experiments the Cholesky factorization [10], a complex operation for the solution of symmetric positive definite linear systems that is representative of many other DLA computations. Therefore, we are confident that our approach and results carry beyond a significant fraction of LAPACK (*Linear Algebra PACKage*) [11].

- For this particular factorization, we describe how to leverage the asymmetry-oblivious task-parallel runtime scheduler in OmpSs, in combination with a data-parallel instance of the BLAS-3 (*basic linear algebra subprograms*) [12] in the BLIS library specifically designed for ARM big.LITTLE AMPs [13, 14].

- We provide practical evidence that, compared with an *ad-hoc* asymmetry-conscious scheduler recently developed for OmpSs [15], our solution yields higher performance for the multi-threaded execution of the Cholesky factorization on an Exynos 5422 SoC comprising two quad-core clusters, with ARM Cortex-A15 and Cortex-7 technology.

- In conclusion, compared with previous work [13, 15], this paper demonstrates that, for the particular domain of DLA, it is possible to hide the difficulties intrinsic to dealing with an asymmetric architecture (e.g., workload balancing for performance, energy-aware mapping of tasks to cores, and criticality-aware scheduling) inside an asymmetry-aware implementation of the BLAS-3. As a consequence, our solution can refactor any conventional (asymmetry-agnostic) scheduler to exploit the task parallelism present in complex DLA operations.

The rest of the paper is structured as follows. Section 2 presents the target ARM big.LITTLE AMP, together with the main execution paradigms it exposes. Section 3 reviews the state-of-the-art in runtime-based task scheduling and DLA libraries for (heterogeneous and) asymmetric architectures. Section 4 introduces the approach to reuse

conventional runtime task schedulers on AMPs by relying on an asymmetric DLA library. Section 5 reports the performance results attained by the proposed approach; and Section 6 closes the paper with the final remarks.

2. Software Execution Models for ARM big.LITTLE SoCs

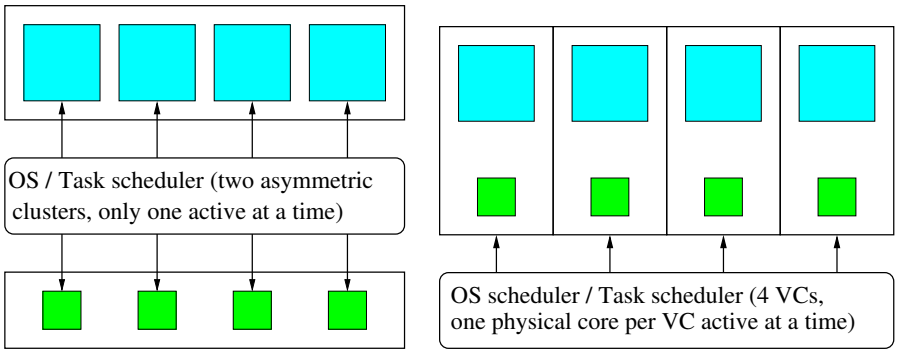
The target architecture for our design and evaluation is an ODROID-XU3 board comprising a Samsung Exynos 5422 SoC with an ARM Cortex-A15 quad-core processing cluster (running at 1.3 GHz) and a Cortex-A7 quad-core processing cluster (also operating at 1.3 GHz). Both clusters access a shared DDR3 RAM (2 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.

Modern big.LITTLE SoCs, such as the Exynos 5422, offer a number of software execution models with support from the operating system (OS):

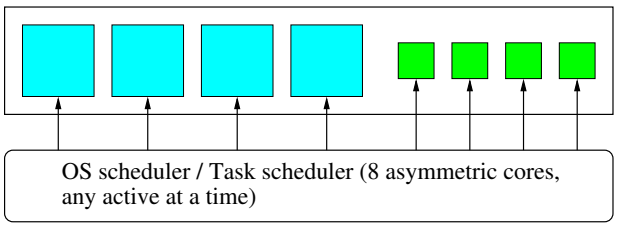
1. *Cluster Switching Mode (CSM)*: The processor is logically divided into two clusters, one containing the big cores and the other with the LITTLE cores, but only one cluster is usable at any given time. The OS transparently activates/deactivates the clusters depending on the workload in order to balance performance and energy efficiency.
2. *CPU migration (CPUM)*: The physical cores are grouped into pairs, each consisting of a fast core and a slow core, building VCs to which the OS maps threads. At any given moment, only one physical core is active per VC, depending on the requirements of the workload. In big.LITTLE specifications where the number of fast and slow cores do not match, the VC can be assembled from a different number of cores of each type. The *In-Kernel Switcher* (IKS) is Linaro's solution for this model.
3. *Global Task Scheduling (GTS)*. This is the most flexible model. All 8 cores are available for thread scheduling, and the OS maps the threads to any of them depending on the specific nature of the workload and core availability. ARM's implementation of GTS is referred to as big.LITTLE MP.

Figure 1 offers an schematic view of these three execution models for modern big.LITTLE architectures. GTS is the most flexible solution, allowing the OS scheduler to map threads to any available core or group of cores. GTS exposes the complete pool of 8 (fast and slow) cores in the Exynos 5422 SoC to the OS. This allows a straight-forward port of existing threaded application, including runtime task schedulers, to exploit all the computational resources in this AMP, provided the multi-threading technology underlying the software is based on conventional tools such as, e.g., POSIX threads or OpenMP. Attaining high performance in asymmetric architectures, even with a GTS configuration, is not as trivial, and is one of the goals of this paper.

Alternatively, CPUM proposes a pseudo-symmetric view of the Exynos 5422, transforming this 8-core asymmetric SoC into 4 symmetric multicore processors (SMPs), which are logically exposed to the OS scheduler. (In fact, as this model only allows one active core per VC, but the type of the specific core that is in operation can differ from one VC to another, the CPUM is still asymmetric.)



(a) CSM (b) CPUM



(c) GTS

Figure 1: Operation modes for modern big.LITTLE architectures.

In practice, runtime task schedulers can mimic or approximate any of these OS operation modes. A straight-forward model is simply obtained by following the principles governing GTS to map ready tasks to any available core. With this solution, load unbalance can be tackled via *ad-hoc* (i.e., asymmetry-aware) scheduling policies embedded into the runtime that map tasks to the most "appropriate" resource.

3. Parallel Execution of DLA Operations on Multi-threaded Architectures

In this section we briefly review several software efforts, in the form of task-parallel runtimes and libraries, that were specifically designed for DLA, or have been successfully applied in this domain, *when the target is (an heterogeneous system or) an AMP*.

3.1. Runtime task scheduling of complex DLA operations

3.1.1. Task scheduling for the Cholesky factorization

We start by describing how to extract task parallelism during the execution of a DLA operation, using the Cholesky factorization as a workhorse example. This particular operation, which is representative of several other factorizations for the solution of linear

systems, decomposes an $n \times n$ symmetric positive definite matrix A into the product $A = U^T U$, where the $n \times n$ Cholesky factor U is upper triangular [10].

```
1  void cholesky (double *A[s][s], int b, int s)
2  {
3     for (int k = 0; k < s; k++) {
4
5        po_cholesky (A[k][k], b, b);              // Cholesky factorization
6                                                  // (diagonal block)
7        for (int j = k + 1; j < s; j++)
8           tr_solve (A[k][k], A[k][j], b, b);     // Triangular solve
9
10       for (int i = k + 1; i < s; i++) {
11          for (int j = i + 1; j < s; j++)
12             ge_multiply (A[k][i], A[k][j],
13                          A[i][j], b, b);        // Matrix multiplication
14          sy_update (A[k][i], A[i][i], b, b);    // Symmetric rank-b update
15       }
16
17    }
18 }
```

Listing 1: C implementation of the blocked Cholesky factorization.

Listing 1 displays a simplified C code for the factorization of an `n`×`n` matrix `A`, stored as `s`×`s` (data) sub-matrices of dimension `b`×`b` each. This blocked routine decomposes the operation into a collection of building *kernels*: `po_cholesky` (Cholesky factorization), `tr_solve` (triangular solve), `ge_multiply` (matrix multiplication), and `sy_update` (symmetric rank-`b` update). The order in which these kernels are invoked during the execution of the routine, and the sub-matrices that each kernel read/writes, result in a direct acyclic graph (DAG) that reflects the dependencies between tasks (i.e., instances of the kernels) and, therefore, the task parallelism of the operation. For example, Figure 2 shows the DAG with the tasks (nodes) and data dependencies (arcs) intrinsic to the execution of Listing 1, when applied to a matrix composed of 4×4 sub-matrices (i.e., `s`=4).

The DAG associated with an algorithm/routine is a graphical representation of the task parallelism of the corresponding operation, and a runtime system can exploit this information to determine the task schedules that satisfy the DAG dependencies. For this purpose, in OmpSs the programmer employs OpenMP-like directives (`pragma`s) to annotate routines appearing in the code as tasks, indicating the directionality of their operands (input, output or input/output) by means of clauses. The OmpSs runtime then decomposes the code (transformed by Mercurium source-to-source compiler) into a number of tasks at run time, dynamically identifying dependencies among these, and issuing *ready tasks* (those with all dependencies satisfied) for their execution to the processor cores of the system.

Listing 2 shows the annotations a programmer needs to add in order to exploit task parallelism using OmpSs; see in particular the lines labelled with '`#pragma omp`". The clauses `in`, `out` and `inout` denote data directionality, and help the task scheduler to keep track of data dependencies between tasks during the execution. We note that, in this implementation, the four kernels simply boil down to calls to four *fundamental computational kernels* for DLA from LAPACK (`dpotrf`) and the BLAS-3 (`dtrsm`, `dgemm` and `dsyrk`).

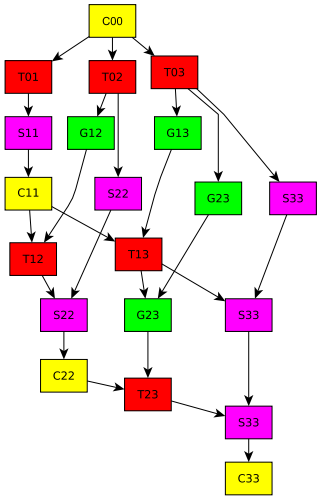


Figure 2: DAG with the tasks and data dependencies resulting from the application of the code in Listing 1 to a 4×4 blocked matrix (`s`=4). The labels specify the type of kernel/task with the following correspondence: "C" for the Cholesky factorization, "T" for the triangular solve, "G" for the matrix multiplication, and "S" for the symmetric rank-`b` update. The subindices (starting at 0) specify the sub-matrix that the corresponding task updates, and the color is used to distinguish between different values of the iteration index `k`.

3.1.2. Task scheduling in heterogeneous and asymmetric architectures

For servers equipped with one or more graphics accelerators, (specialized versions of) the schedulers underlying OmpSs, StarPU, MAGMA, Kaapi and `libflame` distinguish between the execution target being either a general-purpose core (CPU) or a GPU, assigning tasks to each type of resource depending on their properties, and applying techniques such as data caching or locality-aware task mapping; see, among many others, [16, 17, 18, 19, 20].

The designers/developers of the OmpSs programming model and the Nanos++ runtime task scheduler recently introduced a new version of their framework, hereafter referred to as Botlev-OmpSs, specifically tailored for AMPs [15]. This asymmetry-conscious runtime embeds a scheduling policy CATS (Criticality-Aware Task Scheduler) that relies on bottom-level longest-path priorities, keeps track of the criticality of the individual tasks, and leverages this information, at execution time, to assign a ready task to either a critical or a non-critical queue. In this solution, tasks enqueued in the critical queue can only be executed by the fast cores. In addition, the enhanced scheduler integrates uni- or bi-directional work stealing between fast and slow cores. According to the authors, this sophisticated *ad-hoc* scheduling strategy for heterogeneous/asymmetric processors attains remarkable performance improvements in a number of target applications; see [15] for further details.

When applied to a task-parallel DLA routine, the asymmetry-aware scheduler in Botlev-OmpSs maps each task to a single (big or LITTLE) core, and simply invokes a sequential DLA library to conduct the actual work. On the other hand, we note that this

```
#pragma omp task inout([b][b]A)
void po_cholesky (double *A, int b, int ld)
{
 static int         INFO = 0;
 static const char UP   = 'U';
 dpotrf (&UP, &b, A, &ld, &INFO);  // LAPACK Cholesky factorization
}

#pragma omp task in([b][b]A) inout([b][b]B)
void tr_solve (double *A, double *B, int b, int ld)
{
 static double      DONE = 1.0;
 static const char LE   = 'L', UP = 'U', TR = 'T', NU = 'N';
 dtrsm (&LE, &UP, &TR, &NU, &b, &b,
        &DONE, A, &ld, B, &ld);    // BLAS-3 triangular solve
}

#pragma omp task in([b][b]A, [b][b]B) inout([b][b]C)
void ge_multiply (double *A, double *B, double *C, int b, int ld)
{
 static double      DONE = 1.0, DMONE = -1.0;
 static const char TR   = 'T', NT    = 'N';
 dgemm (&TR, &NT, &b, &b, &b,
        &DMONE, A, &ld, B, &ld,
        &DONE,  C, &ld);           // BLAS-3 matrix multiplication
}

#pragma omp task in([b][b]A) inout([b][b]C)
void sy_update (double *A, double *C, int b, int ld)
{
 static double      DONE = 1.0, DMONE = -1.0;
 static const char UP   = 'U', TR    = 'T';
 dsyrk (&UP, &TR, &b, &b,
        &DMONE, A, &ld,
        &DONE,  C, &ld);           // BLAS-3 symmetric rank-b update
}
```

Listing 2: Labeled tasks involved in the blocked Cholesky factorization.

approach required an important redesign of the underlying scheduling policy (and thus, a considerable programming effort for the runtime developer), in order to exploit the heterogeneous architecture. In particular, detecting the criticality of a task at execution time is a nontrivial question.

3.2. Data-parallel libraries of fundamental (BLAS-3) DLA kernels

3.2.1. Multi-threaded implementation of the BLAS-3

An alternative to the runtime-based (i.e., task-parallel) approach to execute DLA operations on multi-threaded architectures consists in relying on a library of specialized kernels that statically partitions the work among the computational resources, or leverages a simple schedule mechanism such as those available, e.g., in OpenMP. For DLA operations with few and/or simple data dependencies, as is the case of the BLAS-3, and/or when the number of cores in the target architecture is small, this option can avoid the costly overhead of a sophisticated task scheduler, providing a more efficient solution. Currently this is preferred option for all high performance implementations of the BLAS for multicore processors, being adopted in both commercial and open source packages such as, e.g., AMD ACML, IBM ESSL, Intel MKL, GotoBLAS [21], OpenBLAS [22] and BLIS.

BLIS in particular mimics GotoBLAS to orchestrate all BLAS-3 kernels (including the matrix multiplication, GEMM) as three nested loops around two packing routines, which accommodate the data in the higher levels of the cache hierarchy, and a *macro-kernel* in charge of performing the actual computations. Internally, BLIS implements the macro-kernel as two additional loops around a *micro-kernel* that, in turn, boils down to a loop around a symmetric rank-1 update. For the purpose of the following discussion, we will only consider the three outermost loops in the BLIS implementation of GEMM for the multiplication $C := C + A \cdot B$, where A, B, C are respectively $m \times k$, $k \times n$ and $m \times n$ matrices, stored in arrays `A`, `B` and `C`; see Listing 3. In the code, `mc`, `nc`, `kc` are cache configuration parameters that need to be adjusted for performance taking into account, among others, the latency of the floating-point units, number of vector registers, and size/associativity degree of the cache levels.

```
1  void gemm (double A[m][k], double B[k][n], double C[m][n],
2             int m,  int n,  int k, int mc, int nc, int kc)
3  {
4     double *Ac = malloc (mc * kc * sizeof (double)),
5            *Bc = malloc (kc * nc * sizeof (double));
6
7     for (int jc = 0; jc < n; jc+=nc) {              // Loop 1
8        int jb = min(n-jc+1, nc);
9
10       for (int pc = 0; pc < k; jc+=kc) {           // Loop 2
11          int pb = min(k-pc+1, kc);
12
13          pack_buffB (B[pc][jc], Bc, kb, nb);       // Pack A->Ac
14
15          for (int ic = 0; ic < m; ic+=mc) {        // Loop 3
16             int ib = min(m-ic+1, mc);
17
18             pack_buffA (A[ic][pc], Ac, mb, kb);    // Pack A->Ac
19
20             gemm_kernel (Ac, Bc, C[ic][jc],
21                          mb, nb, kb, mc, nc, kc);  // Macro-kernel
22          }
23       }
24    }
25 }
```

Listing 3: High performance implementation of GEMM in BLIS.

3.2.2. Data-parallel libraries for asymmetric architectures

The implementation of GEMM in BLIS has been demonstrated to deliver high performance on a wide range of multicore and many-core SMPs [23, 24]. These studies offered a few relevant insights that guided the parallelization of GEMM (and also other Level-3 BLAS) on the ARM big.LITTLE architecture under the GTS software execution model. Concretely, the architecture-aware multi-threaded parallelization of GEMM in [13] integrates the following three techniques:

- A dynamic 1-D partitioning of the iteration space to distribute the workload in either Loop 1 or Loop 3 of BLIS GEMM between the two clusters.
- A static 1-D partitioning of the iteration space that distributes the workload of one of the loops internal to the macro-kernel between the cores of the same cluster.

- A modification of the control tree that governs the multi-threaded parallelization of BLIS GEMM in order to accommodate different loop strides for each type of core architecture.

The strategy is general and can be applied to a generic AMP, consisting of any combination of fast/slow cores sharing the main memory, as well as to all other Level-3 BLAS operations.

To close this section, we emphasize that our work differs from [13, 15] in that we address sophisticated DLA operations, with a rich hierarchy of task dependencies, by leveraging a conventional runtime scheduler in combination with a data-parallel asymmetry-conscious implementation of the BLAS-3.

4. Retargeting Existing Task Schedulers to Asymmetric Architectures

In this section, we initially perform an evaluation of the task-parallel Cholesky routine in Listings 1–2, executed on top of the conventional (i.e., default) scheduler in OmpSs linked to a sequential instance of BLIS, on the target Exynos 5422 SoC. The outcome from this study motivates the development effort and experiments presented in the remainder of the paper.

4.1. Evaluation of conventional runtimes on AMPs

Figure 3 reports the performance, in terms of GFLOPS (billions of flops per second), attained with the conventional OmpSs runtime, when the number of worker threads varies from 1 to 8, and the mapping of worker threads to cores is delegated to the OS. We evaluated a range of block sizes (`b` in Listing 1), but for simplicity we report only the results obtained with the value `b` that optimized the GFLOPS rate for each problem dimension. All the experiments hereafter employed IEEE double precision. Furthermore, we ensured that the cores operate at the highest possible frequency by setting the appropriate *cpufreq* governor. The conventional runtime of OmpSs corresponds to release 15.06 of the Nanos++ runtime task scheduler. For this experiment, it is lined with the "sequential" implementation of BLIS in release 0.1.5. (For the experiments with the multi-threaded/asymmetric version of BLIS in the later sections, we will use specialized versions of the codes in [13] for slow+fast VCs.)

The results in the Figure reveal the increase in performance as the number of worker threads is raised from 1 to 4, which the OS maps to the (big) Cortex-A15 cores. However, when the number of threads exceeds the amount of fast cores, the OS starts binding the threads to the slower Cortex-A7 cores, and the improvement rate is drastically reduced, in some cases even showing a performance drop. This is due to load imbalance, as tasks of uniform granularity, possibly laying in the critical path, are assigned to slow cores.

Stimulated by this first experiment, we recognize that an obvious solution to this problem consists in adapting the runtime task scheduler (more specifically, the scheduling policy) to exploit the SoC asymmetry [15]. Nevertheless, we part ways with this solution, exploring an architecture-aware alternative that leverages a(ny) conventional runtime task scheduler combined with an underlying asymmetric library. We discuss this option in more detail in the following section.

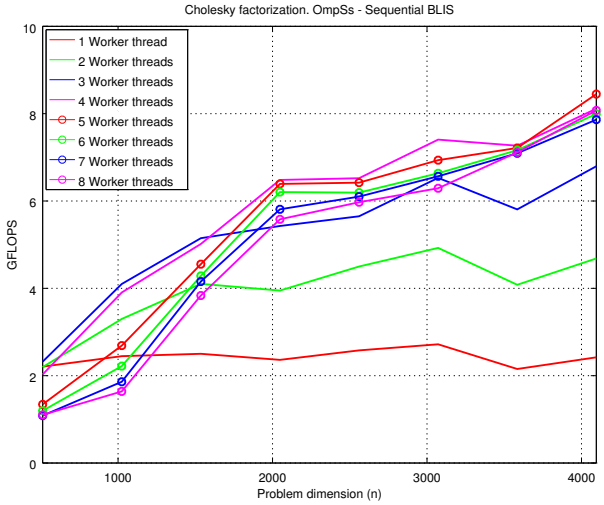


Figure 3: Performance of the Cholesky factorization using the conventional OmpSs runtime and a sequential implementation of BLIS on the Exynos 5422 SoC.

4.2. Combining conventional runtimes with asymmetric libraries

4.2.1. General view

Our proposal operates under the GTS model but is inspired in CPUM. Concretely, our task scheduler regards the computational resources as four *truly* symmetric VCs, each composed of a fast and a slow core. For this purpose, unlike CPUM, *both* physical cores within each VC remain active and collaborate to execute a given task. Furthermore, our approach exploits concurrency at two levels: *task-level parallelism* is extracted by the runtime in order to schedule tasks to the four symmetric VCs; and each task/kernel is internally divided to expose *data-level parallelism*, distributing its workload between the two asymmetric physical cores within the VC in charge of its execution.

Our solution thus only requires a conventional (and thus asymmetry-agnostic) runtime task scheduler, e.g. the conventional version of OmpSs, where instead of spawning one worker thread per core in the system, we adhere to the CPUM model, creating only one worker thread per VC. Internally, whenever a ready task is selected to be executed by a worker thread, the corresponding routine from BLIS internally spawns two threads, binds them to the appropriate pair of Cortex-A15+Cortex-A7 cores, and asymmetrically divides the work between the fast and the slow physical cores in the VC. Following this idea, the architecture exposed to the runtime is *symmetric*, and the kernels in the BLIS library configure a "black box" that abstracts the architecture asymmetry from the runtime scheduler.

In summary, in a conventional setup, the core is the basic computational resource for the task scheduler, and the "sequential" tasks are the minimum work unit to be assigned to these resources. Compared with this, in our approach the VC is the smallest (basic) computational resource from the point of view of the scheduler; and tasks are further divided into smaller units, and executed in parallel by the physical cores inside the VCs.

4.2.2. Comparison with other approaches

Our approach features a number of advantages for the developer:

- The runtime is not aware of asymmetry, and thus a conventional task scheduler will work transparently with no special modifications.
- Any existing scheduling policy (e.g. cache-aware mapping or work stealing) in an asymmetry-agnostic runtime, or any enhancement technique, will directly impact the performance attained on an AMP.
- Any improvement in the asymmetry-aware BLIS implementation will directly impact the performance on an AMP. This applies to different ratios of big/LITTLE cores within a VC, operating frequency, or even to the introduction further levels of asymmetry (e.g. cores with a capacity between fast and slow).

Obviously, there is also a drawback in our proposal as a tuned asymmetry-aware DLA library must exist in order to reuse conventional runtimes. In the scope of DLA, this drawback is easily tackled with BLIS. We recognize that, in more general domains, an ad-hoc implementation of the application's fundamental kernels becomes mandatory in order to fully exploit the underlying architecture.

4.2.3. Requisites on the BLAS-3

We finally note that certain requirements are imposed on a multi-threaded implementation of BLIS that operates under the CPUM mode. To illustrate this, consider the GEMM kernel and the high-level description of its implementation in Listing 3. For our objective, we still have to distribute the iteration space between the Cortex-A15 and the Cortex-A7 but, since there is only one resource of each type per VC, there is no need to partition the loops internal to the macro-kernel. Furthermore, we note that the optimal strides for Loop 1 are in practice quite large (`nc` is in the order of a few thousands for ARM big.LITTLE cores), while the optimal values for Loop 3 are much more reduced (`mc` is 32 for the Cortex-A7 and 156 for the Cortex-A15). Therefore, we target Loop 3 in our data-parallel implementation of BLIS for VCs, which we can expect to easily yield a proper workload balancing.

5. Experimental results

5.1. Performance evaluation for asymmetric BLIS

Let us start by reminding that, at execution time, OmpSs decomposes the routine for the Cholesky factorization into a collection of tasks that operate on sub-matrices (blocks) with a granularity dictated by the block size `b`; see Listing 1. These tasks typically perform invocations to a fundamental kernel of the BLAS-3, in our case provided by BLIS, or LAPACK; see Listing 2.

The first step in our evaluation aims to provide a realistic estimation of the potential performance benefits of our approach (if any) on the target Exynos 5422 SoC. A critical factor from this perspective is the range of block sizes, say b^{opt}, that are optimal for the conventional OmpSs runtime. In particular, the efficiency of our hybrid task/data-parallel approach is strongly determined by the performance attained with the asymmetric BLIS implementation when compared against that of its sequential counterpart, for problem dimensions `n` that are in the order of b^{opt}.

Table 1: Optimal block sizes for the Cholesky factorization using the conventional OmpSs runtime and a sequential implementation of BLIS on the Exynos 5422 SoC.

	Problem dimension (n)														
	512	1,024	1,536	2,048	2,560	3,072	3,584	4,096	4,608	5,120	5,632	6,144	6,656	7,168	7,680
1 WT	192	384	320	448	448	448	384	320	320	448	448	448	448	384	448
2 WT	192	192	320	192	448	448	384	320	320	448	448	448	448	384	448
3 WT	128	192	320	192	384	448	320	320	320	448	448	448	448	384	448
4 WT	128	128	192	192	192	320	320	320	320	448	320	448	448	384	448

Table 1 reports the optimal block sizes b^{opt} for the Cholesky factorization, with problems of increasing matrix dimension, using the conventional OmpSs runtime linked with the sequential BLIS, and 1 to 4 worker threads. Note that, except for smallest problems, the observed optimal block sizes are between 192 and 448. These dimensions offer a fair compromise, exposing enough task-level parallelism while delivering high "sequential" performance for the execution of each individual task via the sequential implementation of BLIS.

The key insight to take away from this experiments is that, in order to extract good performance from a combination of the conventional OmpSs runtime task scheduler with a multi-threaded asymmetric version of BLIS, the kernels in this instance of the asymmetric library must outperform their sequential counterparts, for matrix dimensions in the order of the block sizes in Table 1. Figure 4 shows the performance attained with the three BLAS-3 tasks involved in the Cholesky factorization (GEMM, SYRK and TRSM) for our range of dimensions of interest. There, the multi-threaded asymmetry-aware kernels run concurrently on one Cortex-A15 plus one Cortex-A7 core, while the sequential kernels operate exclusively on a single Cortex-A15 core. In general, the three BLAS-3 routines exhibit a similar trend: the kernels from the sequential BLIS outperform their asymmetric counterparts for small problems (up to approximately $\mathtt{m}$, $\mathtt{n}$, $\mathtt{k} = 128$); but, from that dimension, the use of the slow core starts paying off. The interesting aspect here is that the cross-over threshold between both performance curves is in the range, (usually at an early point,) of b^{opt}; see Table 1. This implies that the asymmetric BLIS can potentially improve the performance of the overall computation. Moreover, the gap in performance grows with the problem size, stabilizing at problem sizes around $\mathtt{m}$, $\mathtt{n}$, $\mathtt{k} \approx 400$. Given that this value is in the range of the optimal block size for the task-parallel Cholesky implementation, we can expect a performance increment in the order of 0.3–0.5 GFLOPS per added slow core, mimicking the behavior of the underlying BLIS.

5.2. Integration of asymmetric BLIS in a conventional task scheduler

In order to analyze the actual benefits of our proposal, we next evaluate the conventional OmpSs task scheduler linked with either the sequential implementation of BLIS or its multi-threaded asymmetry-aware version. Hereafter, the BLIS kernels from first configuration always run using one Cortex-A15 core while, in the second case, they exploit one Cortex-A15 plus one Cortex-A7 core. Figure 5 reports the results for both setups, using an increasing number of worker threads from 1 to 4. For simplicity, we only report the results obtained with the optimal block size. In all cases, the solution based on the multi-threaded asymmetric library outperforms the sequential implementation for relatively large matrices (usually for dimensions $\mathtt{n} > 2{,}048$) while, for smaller problems, the

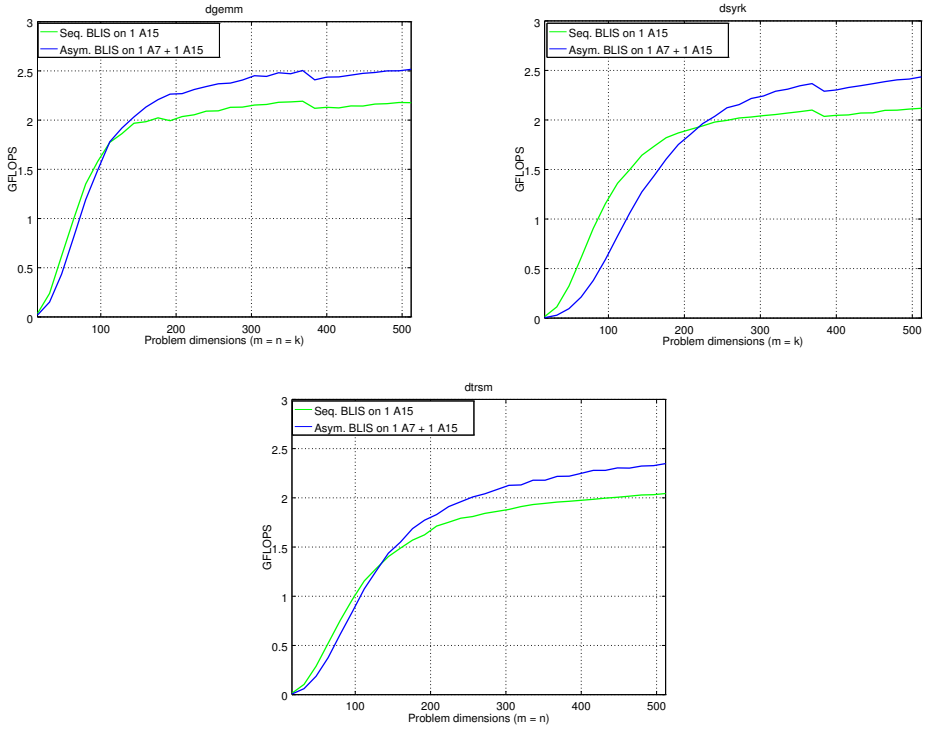


Figure 4: Performance of the BLAS-3 kernels in the sequential and the multi-threaded/asymmetric implementations of BLIS, using respectively one Cortex-A15 core and one Cortex-A15 plus one Cortex-A7 core of the Exynos 5422 SoC.

GFLOPS rates are similar. The reason for this behavior can be derived from the optimal block sizes reported in Table 1 and the performance of BLIS reported in Figure 4: for that range of problem dimensions, the optimal block size is significantly smaller, and both BLIS implementations attain similar performance rates.

The quantitative difference in performance between both approaches is reported in Tables 2 and 3. The first table illustrates the raw (i.e., absolute) gap, while the second one shows the difference per Cortex-A7 core introduced in the experiment. Let us consider, for example, the problem size $\mathtt{n} = 6{,}144$. In that case, the performance roughly improves by 0.975 GFLOPS when the 4 slow cores are added to help the base 4 Cortex-A15 cores. This translates into a performance raise of 0.243 GFLOPS per slow core, which is slightly under the improvement that could be expected from results experiments in the previous section. Note, however, that the performance per Cortex-A7 core is reduced from 0.340 GFLOPS, when adding just one core, to 0.243 GFLOPS, when simultaneously using all four slow cores.

5.3. Performance comparison versus asymmetry-aware task scheduler

Our last round of experiments aims to assess the performance advantages of different task-parallel executions of the Cholesky factorization via OmpSs. Concretely, we consider

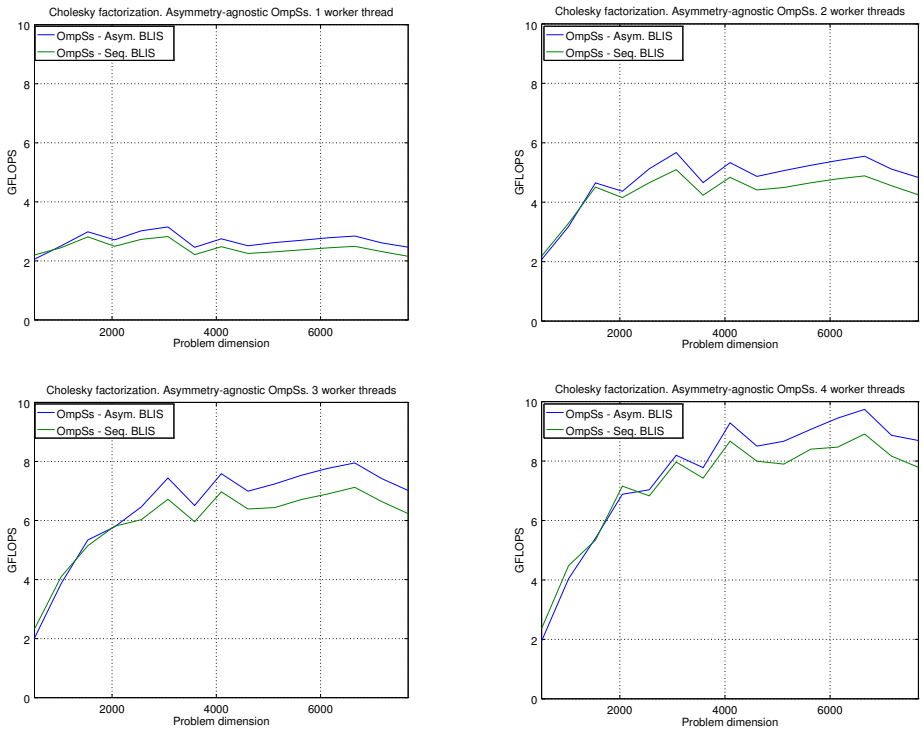


Figure 5: Performance of the Cholesky factorization using the conventional OmpSs runtime linked with either the sequential or the multi-threaded/asymmetric implementations of BLIS in the Exynos 5422 SoC.

(1) the conventional task scheduler linked with the sequential BLIS ("OmpSs - Seq. BLIS"); (2) the conventional task scheduler linked with our multi-threaded asymmetric BLIS that views the SoC as four symmetric *virtual cores* ("OmpSs - Asym. BLIS"); and (3) the criticality-aware task scheduler in Botlev-OmpSs linked with the sequential BLIS ("Botlev-OmpS - Seq. BLIS"). In the executions, we use all four Cortex-A15 cores and evaluate the impact of adding an increasing number of Cortex-A7 cores, from 1 to 4, for Botlev-OmpSs.

Figure 6 shows the performance attained by the aforementioned alternatives on the Exynos 5422 SoC. The results can be divided into groups along three problem dimensions:

- For small matrices ($\mathtt{n} = 512, 1{,}024$), the conventional runtime using exclusively the four big cores (that is, linked with a sequential BLIS library for task execution) attains the best results in terms of performance. This was expected and was already observed in Figure 5; the main reason is the small optimal block size, enforced by the reduced problem size, that is necessary in order to expose enough task-level parallelism. This invalidates the use of our asymmetric BLIS implementation due to the low performance for very small matrices; see Figure 4. We note that the ad-hoc Botlev-OmpSs does not attain remarkable performances either for this dimension range, regardless the amount of Cortex-A7 cores used.

Table 2: Absolute performance improvement (in GFLOPS) for the Cholesky factorization using the conventional OmpSs runtime linked with the multi-threaded/asymmetric BLIS with respect to the same runtime linked with the sequential BLIS in the Exynos 5422 SoC.

	Problem dimension (n)											
	512	1,024	2,048	2,560	3,072	4,096	4,608	5,120	5,632	6,144	6,656	7,680
1 WT	-0.143	0.061	0.218	0.289	0.326	0.267	0.259	0.313	0.324	0.340	0.348	0.300
2 WT	-0.116	-0.109	0.213	0.469	0.573	0.495	0.454	0.568	0.588	0.617	0.660	0.582
3 WT	-0.308	-0.233	-0.020	0.432	0.720	0.614	0.603	0.800	0.820	0.866	0.825	0.780
4 WT	-0.421	-0.440	-0.274	0.204	0.227	0.614	0.506	0.769	0.666	0.975	0.829	0.902

Table 3: Performance improvement per slow core (in GFLOPS) for the Cholesky factorization using the conventional OmpSs runtime linked with the multi-threaded/asymmetric BLIS with respect to the same runtime linked with the sequential BLIS in the Exynos 5422 SoC.

	Problem dimension (n)											
	512	1,024	2,048	2,560	3,072	4,096	4,608	5,120	5,632	6,144	6,656	7,680
1 WT	-0.143	0.061	0.218	0.289	0.326	0.267	0.259	0.313	0.324	0.340	0.348	0.300
2 WT	-0.058	-0.054	0.106	0.234	0.286	0.247	0.227	0.284	0.294	0.308	0.330	0.291
3 WT	-0.102	-0.077	-0.006	0.144	0.240	0.204	0.201	0.266	0.273	0.288	0.275	0.261
4 WT	-0.105	-0.110	-0.068	0.051	0.056	0.153	0.126	0.192	0.166	0.243	0.207	0.225

- For medium-sized matrices (n = 2,048, 4,096), the gap in performance between the different approaches is reduced. The variant that relies on the asymmetric BLIS implementation commences to outperform the alternative implementations for n=4,096 by a short margin. For this problem range, Botlev-OmpSs is competitive, and also outperforms the conventional setup.

- For large matrices (n = 6,144, 7,680) this trend is consolidated, and both asymmetry-aware approaches deliver remarkable performance gains with respect to the conventional setup. Comparing both asymmetry-aware solutions, our mechanism attains better performance rate, even when considering the usage of all available cores for the Botlev-OmpSs runtime version.

To summarize, our proposal to exploit asymmetry improves portability and programmability by avoiding a revamp of the runtime task scheduler for AMPs. In addition, our approach renders performance gains which are, for all problems cases, comparable with those of ad-hoc asymmetry-conscious schedulers; for medium to large matrices, it clearly outperforms the efficiency attained with a conventional asymmetry-oblivious scheduler.

5.4. Extended performance analysis

We next provide further details on the performance behavior of each one of the aforementioned runtime configurations. The execution traces in this section have all been extracted with the `Extrae` instrumentation tool and analyzed with the visualization package `Paraver` [25]. The results correspond to the Cholesky factorization of a single problem with matrix dimension n = 6,144 and block size b = 448.

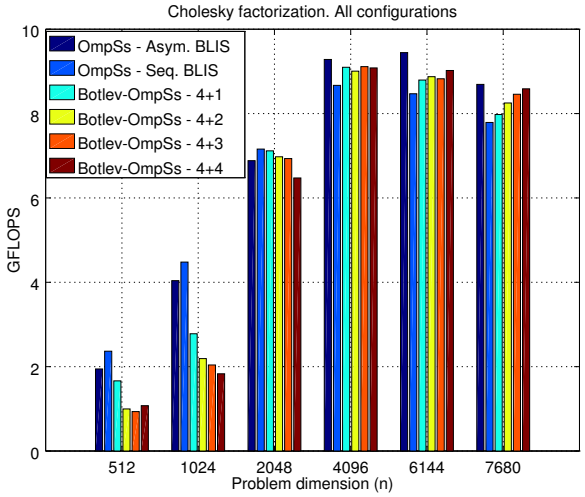


Figure 6: Performance (in GFLOPS) for the Cholesky factorization using the conventional OmpSs runtime linked with either the sequential BLIS or the multi-threaded/asymmetric BLIS, and the *ad-hoc* asymmetry-aware version of the OmpSs runtime (Botlev-OmpSs) linked with the sequential BLIS in the Exynos 5422 SoC. The labels of the form "4+x" refer to an execution with 4 Cortex-A15 cores and x Cortex-A7 cores.

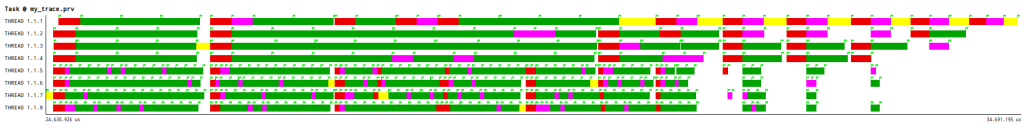
5.4.1. General task execution overview.

Figure 7 reports complete execution traces for each runtime configuration. At a glance, a number of coarse remarks can be extracted from the trace:

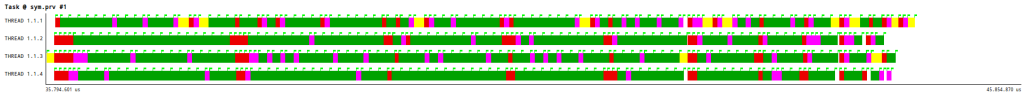
- From the perspective of total execution time (i.e., *time-to-solution*), the conventional OmpSs runtime combined with the asymmetric BLIS implementation attains the best results, followed by the Botlev-OmpSs runtime configuration. It is worth pointing out that an asymmetry-oblivious runtime which spawns 8 worker threads, with no further considerations, yields the worst performance by far. In this case, the load imbalance and long idle periods, especially as the amount of concurrency decreases in the final part of the trace, entail a huge performance penalty.

- The flag marks indicating task initialization/completion reveal that the asymmetric BLIS implementation (which employs the combined resources from a VC) requires less time per task than the two alternatives based on a sequential BLIS. An effect to note specifically in the Botlev-OmpSs configuration is the difference in performance between tasks of the same type, when executed by a big core (worker threads 5 to 8) or a LITTLE one (worker threads 1 to 4).

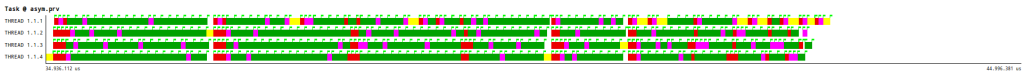
- The Botlev-OmpSs task scheduler embeds a (complex) scheduling strategy that includes priorities, advancing the execution of tasks in the critical path and, whenever possible, assigning them to fast cores (see, for example, the tasks for the factorization of diagonal blocks, colored in yellow). This yields an execution timeline that is more compact during the first stages of the parallel execution, at the cost of longer idle times when the degree of concurrency decreases (last iterations of the



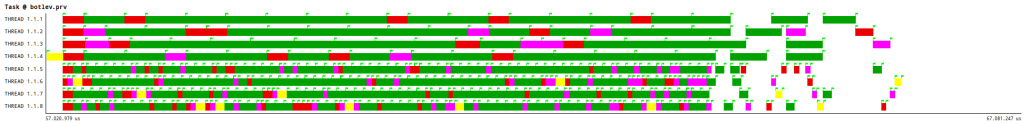
(a) OmpSs - Sequential BLIS (8 worker threads)



(b) OmpSs - Sequential BLIS (4 worker threads)



(c) OmpSs - Asymmetric BLIS (4 worker threads)



(d) Botlev-OmpSs - Sequential BLIS (8 worker threads, 4+4)

Figure 7: Execution traces of the three runtime configurations for the Cholesky factorization (`n` = 6,144, `b` = 448). The timeline in each row collects the tasks executed by a single worker thread. Tasks are colored following the convention in Figure 2; phases colored in white between tasks represent idle times. The green flags mark task initialization/completion.

factorization). Although possible, a priority-aware technique has not been applied in our experiments with the conventional OmpSs setups and remains part of future work.

We next provide a quantitative analysis on the task duration and a more detailed study of the scheduling strategy integrated in each configuration.

5.4.2. Task duration.

Table 4 reports the average execution time per type of task for each worker thread. These results show that the execution time per individual type of task is considerably shorter for our multithreaded/asymmetric BLIS implementation than for the alternatives based on a sequential version of BLIS. The only exception is the factorization of the diagonal block (`dpotrf`) as this is an LAPACK-level routine, and therefore it is not available in BLIS. Inspecting the task execution time of the Botlev-OmpSs configuration,

Table 4: Average time (in ms) per task and worker thread in the Cholesky factorization (`n` = 6,144, `b` = 448), for the three runtime configurations.

	OmpSs - Seq. BLIS (4 worker threads)				OmpSs - Asym. BLIS (4 worker threads)				Botlev-OmpSs - Seq. BLIS (8 worker threads, 4+4)			
	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`
WT 0	89.62	48.12	47.14	101.77	79.82	42.77	44.42	105.77	406.25	216.70	–	–
WT 1	88.96	48.10	47.14	–	78.65	42.97	44.56	76.35	408.90	207.41	212.55	–
WT 2	89.02	48.36	47.18	87.22	79.14	43.14	44.60	85.98	415.31	230.07	212.56	–
WT 3	90.11	48.51	47.42	–	79.28	43.10	44.59	67.73	410.84	216.95	216.82	137.65
WT 4	–	–	–	–	–	–	–	–	90.97	48.97	48.36	–
WT 5	–	–	–	–	–	–	–	–	90.61	48.86	48.16	90.78
WT 6	–	–	–	–	–	–	–	–	91.28	49.43	47.97	89.58
WT 7	–	–	–	–	–	–	–	–	91.60	49.49	48.62	95.43
AVG.	89.43	48.27	47.22	94.49	79.22	42.99	44.54	83.96	250.72	133.49	119.29	103.36

we observe a remarkable difference depending on the type of core tasks are mapped to. For example, the average execution times for `dgemm` range from more than 400 ms on a LITTLE core, to roughly 90 ms on a big core. This behavior is reproduced for all types of tasks.

5.4.3. Task scheduling policies and idle times.

Figure 8 illustrates the task execution order determined by the Nanos++ task scheduler. Here tasks are depicted using a color gradient, attending to the order in which they are encountered in the sequential code, from the earliest to the latest.

At runtime, the task scheduler in Botlev-OmpSs issues tasks to execution out-of-order depending on their criticality. The main idea behind this scheduling policy is to track the criticality of each task and, when possible, advance the execution of critical tasks assigning them to the fast Cortex-A15 cores. Conformally with this strategy, an out-of-order execution reveals itself more frequently in the timelines for the big cores than in those for the LITTLE cores. With the conventional runtime, the out-of-order execution is only dictated by the order in which data dependencies for tasks are satisfied.

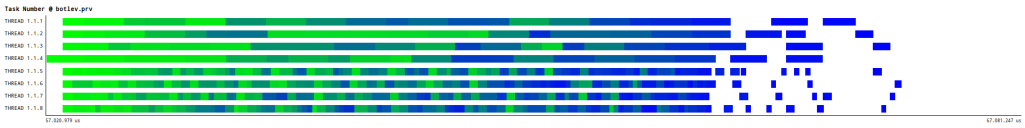
From the execution traces, we can observe that the Botlev-OmpSs alternative suffers a remarkable performance penalty due to the existence of idle periods in the final part of the factorization, when the concurrency in the factorization is greatly diminished. This problem is not present in the conventional scheduling policies. In the first stages of the factorization, however, the use of a priority-aware policy for the Botlev-OmpSs scheduler effectively reduces idle times. Table 5 reports the percentage of time each worker thread is in `running` or `idle` state. In general, the relative amount of time spent in idle state is much higher for Botlev-OmpSs than for the conventional implementations (17% vs. 5%, respectively). Note also the remarkable difference in the percentage of idle time between the big and LITTLE cores (20% and 13%, respectively), which drives to the conclusion that the fast cores stall waiting for completion of tasks executed on the LITTLE cores. This fact can be confirmed in the final stages of the Botlev-OmpSs trace.

The previous observations pave the road to a combination of scheduling policy and execution model for AMPs, in which asymmetry is exploited through *ad-hoc* scheduling policies during the first stages of the factorization –when the potential parallelism is massive–, and this is later replaced with the use of asymmetric-aware kernels and coarse-

(a) OmpSs - Sequential BLIS (8 worker threads)

(b) OmpSs - Sequential BLIS (4 worker threads)

(c) OmpSs - Asymmetric BLIS (4 worker threads)



(d) Botlev-OmpSs - Sequential BLIS (8 worker threads, 4+4)

Figure 8: Task execution order of the three studied runtime configurations for the Cholesky factorization ($\mathtt{n} = 6{,}144$, $\mathtt{b} = 448$). In the trace, tasks are ordered according to their appearance in the sequential code, and depicted using a color gradient, with light green indicating early tasks, and dark blue for the late tasks.

grain VCs in the final stages of the execution, when the concurrency is scarce. Both approaches are not mutually exclusive, but complementary depending on the level of task concurrency available at a given execution stage.

6. Conclusions

In this paper, we have addressed the problem of refactoring existing runtime task schedulers to exploit task-level parallelism in novel AMPs, focusing on ARM big.LITTLE systems-on-chip. We have demonstrated that, for the specific domain of DLA, an approach that delegates the burden of dealing with asymmetry to the library (in our case, using an asymmetry-aware BLIS implementation), does not require any revamp of existing task schedulers, and can deliver high performance. This proposal paves the road towards reusing conventional runtime schedulers for SMPs (and all the associated improvement techniques developed through the past few years), as the runtime only has a

Table 5: Percentage of time per worker thread in idle or running state for different runtime configurations for the Cholesky factorization (`n` = 6,144, `b` = 448). Note that WT 0 is the master thread, and thus is never idle; for it, the rest of the time till 100% percentage is devoted to `synchronization`, `scheduling` and `thread creation`. For the rest of threads, this amount of time is devoted to `runtime overhead`.

	OmpSs - Seq. BLIS (4 worker threads)		OmpSs - Asym. BLIS (4 worker threads)		Botlev-OmpSs - Seq. BLIS (8 worker threads, 4+4)	
	`idle`	`running`	`idle`	`running`	`idle`	`running`
WT 0	–	98.41	–	97.85	–	86.53
WT 1	5.59	94.22	5.51	94.29	13.63	86.28
WT 2	3.14	96.67	5.27	94.53	13.94	85.98
WT 3	5.77	94.07	5.17	94.62	13.43	86.47
WT 4	–	–	–	–	19.26	80.51
WT 5	–	–	–	–	21.12	78.69
WT 6	–	–	–	–	20.84	78.97
WT 7	–	–	–	–	20.09	79.70
AVG.	4.84	95.89	5.32	94.90	17.47	82.89

symmetric view of the hardware. Our experiments reveal that this solution is competitive and even improves the results obtained with an asymmetry-aware scheduler for DLA operations.

Acknowledgments

The researchers from Universidad Complutense de Madrid were supported by project CICYT TIN2012-32180. Enrique S. Quintana-Ortí was supported by projects CICYT TIN2011-23283 and TIN2014-53495-R as well as the EU project FP7 318793 "EXA2GREEN".

References

[1] 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.

[2] M. Duranton, D. Black-Schaffer, K. D. Bosschere, J. Maebe, The HiPEAC vision for advanced computing in Horizon 2020, `http://www.hipeac.net/roadmap` (2013).

[3] Cilk project home page, `http://supertech.csail.mit.edu/cilk/`, last visit: July 2015.

[4] OmpSs project home page, `http://pm.bsc.es/ompss`, last visit: July 2015.

[5] StarPU project home page, `http://runtime.bordeaux.inria.fr/StarPU/`, last visit: July 2015.

[6] PLASMA project home page, `http://icl.cs.utk.edu/plasma/`, last visit: July 2015.

[7] MAGMA project home page, `http://icl.cs.utk.edu/magma/`, last visit: July 2015.

[8] Kaapi project home page, `https://gforge.inria.fr/projects/kaapi`, last visit: July 2015.

[9] FLAME project home page, `http://www.cs.utexas.edu/users/flame/`, last visit: July 2015.

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

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

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Transactions on Mathematical Software 16 (1) (1990) 1–17.

[13] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors, ArXiv e-prints 1506.08988.
URL `http://arxiv.org/abs/1506.08988`

[14] F. G. Van Zee, R. A. van de Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM Transactions on Mathematical Software 41 (3) (2015) 14:1–14:33.
URL http://doi.acm.org/10.1145/2764454
[15] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, M. Valero, Criticality-aware dynamic task scheduling for heterogeneous architectures, in: Proceedings of ICS'15, 2015.
[16] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Transactions on Mathematical Software 36 (3) (2009) 14:1–14:26.
[17] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using SMPSs, Concurrency and Computation: Practice and Experience 21 (18) (2009) 2438–2456.
[18] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.
[19] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, Dense linear algebra solvers for multicore with GPU accelerators, in: Proc. IEEE Symp. on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–8.
[20] T. Gautier, J. V. F. Lima, N. Maillard, B. Raffin, XKaapi: A runtime system for data-flow task programming on heterogeneous architectures, in: Proc. IEEE 27th Int. Symp. on Parallel and Distributed Processing, IPDPS'13, 2013, pp. 1299–1308.
[21] K. Goto, R. A. van de Geijn, Anatomy of a high-performance matrix multiplication, ACM Transactions on Mathematical Software 34 (3) (2008) 12:1–12:25.
URL http://doi.acm.org/10.1145/1356052.1356053
[22] OpenBLAS, http://xianyi.github.com/OpenBLAS/, last visit: July 2015.
[23] 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.Accepted. Available at http://www.cs.utexas.edu/users/flame.
[24] 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. Symp. on Parallel and Distributed Processing, IPDPS'14, 2014, pp. 1049–1059.
[25] Paraver: the flexible analysis tool, http://www.cepba.upc.es/paraver.