Collaborative Adaptation for Energy-Efficient Heterogeneous Mobile SoCs

Amit Kumar Singh, *Member, IEEE,* Karunakar Reddy Basireddy, *Member, IEEE,* Alok Prakash, *Member, IEEE,* Geoff V. Merrett, *Member, IEEE,* Bashir M Al-Hashimi, *Fellow, IEEE*

Abstract—Heterogeneous Mobile System-on-Chips (SoCs) containing CPU and GPU cores are becoming prevalent in embedded computing, and they need to execute applications concurrently. However, existing run-time management approaches do not perform adaptive mapping and thread-partitioning of applications while exploiting both CPU and GPU cores at the same time. In this paper, we propose an adaptive mapping and thread-partitioning approach for energy-efficient execution of concurrent OpenCL applications on both CPU and GPU cores while satisfying performance requirements. To start execution of concurrent applications, the approach makes mapping (number of cores and operating frequencies) and partitioning (distribution of threads between CPU and GPU) decisions to satisfy performance requirements for each application. The mapping and partitioning decisions are made by having a collaboration between the CPU and GPU cores' processing capabilities such that balanced execution can be performed. During execution, adaptation is triggered when new application(s) arrive, or an executing one finishes, that frees cores. The adaptation process identifies a new mapping and thread-partitioning in a similar collaborative manner for remaining applications provided it leads to an improvement in energy efficiency. The proposed approach is experimentally validated on the Odroid-XU3 hardware platform with varying set of applications. Results show an average energy saving of 37%, compared to existing approaches while satisfying the performance requirements.

Index Terms—SoC, Heterogeneous computing, Adaptation, Energy-efficiency, concurrent execution.

1 Introduction

The reliance of embedded computing systems is increasing to heterogeneous mobile SoCs consisting of different types of cores such as CPU and GPU. An example includes the Samsung Exynos 5422 SoC containing a CPU with four ARM cortex-A15 (big) and four ARM Cortex-A7 (LITTLE) cores, and a GPU with six ARM Mali-T628 cores [1]. These architectures provide opportunities to exploit distinct features of CPU and GPU cores to meet performance and energy consumption requirements. Further, the cores in these SoCs support dynamic voltage and frequency scaling (DVFS), which can be used to reduce dynamic power consumption ($P \propto V^2 f$). This helps to reduce energy consumption if the extra time taken to run the workload at a lower voltage and frequency can be accounted by a sufficient reduction in the power consumption.

Open Computing Language (OpenCL) [2] provides an opportunity to write applications that can execute across heterogeneous cores including CPUs and GPUs [3]–[7]. However, since CPU and GPU cores typically handle task/thread and data level parallelism, respectively, the performance and energy consumption of an application varies when executed on only CPU, only GPU, or both CPU and GPU cores [8]. The variation depends upon the kind of parallelism dominating the application. Existing studies have shown significant reduction in execution time and energy consumption when an application simultaneously exploits both the CPU and GPU cores with an appropriate partitioning of threads between them [7].

For concurrently executing applications in a heterogeneous mobile SoC, e.g., image and MP3 decoding with respective frames per second (fps) requirements, energy-efficient run-time management of applications on SoC resources is desired. Recently, several efforts have been made towards this [9]–[15], but at an application completion and/or arrival, they lack to perform adaptations while using both the CPU and GPU cores for concurrently executing applications. Thus, cores freed by an application cannot be used by already running applications or to start execution of a waiting application, which could help in improving resource utilisation and/or energy efficiency. For executing and arrived applications, the adaptation process needs to identify new mapping (defined as the number of used cores, their types (e.g., big, LITTLE) and operating frequencies) and thread-partitioning (defined as the portion of threads to be executed on CPU cores). However, performing energy-efficient adaptation while satisfying applications' performance requirements possess several key challenges. First, since OpenCL enqueues all application threads between the CPU and GPU before starting execution, with no track of executed threads over time, online thread-partitioning is not possible. To enable this, threads can be grouped into several equal size chunks, and enqueuing and partitioning done at the chunk level. However, this requires to identify the best number of chunks (each chunk contains the same

A. K. Singh is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO43SQ, United Kingdom (email: a.k.singh@essex.ac.uk).
B. K. Reddy, G. V. Merrett, and B. M. Al-Hashimi are with the School of Electronics and Computer Science, University of Southampton, United Kingdom (e-mail: krb1g15@ecs.soton.ac.uk; gvm@ecs.soton.ac.uk; bmah@ecs.soton.ac.uk)
A. Prakash is with the School of Computer Engineering, Nanyang Technological University, Singapore (e-mail: alok@ntu.edu.sg)
Manuscript received August xx, 2018; revised November xx, 2018.

number of threads) so that the cores are neither 'starved' nor 'overfed' from threads. Second, upon an application's completion, energy saving benefits of adaptation (to a higher number of cores) for each executing application needs to be identified by considering timing and energy overheads. Third, upon an application's arrival, energy-efficient and performance satisfying adaptation (to a lower number of cores) of executing applications needs to be performed to facilitate performance satisfying execution of the arrived application. Finally, collaboration between CPU and GPU is required by identifying their processing capabilities at various adaptation points such that balanced and energy-efficient execution can be performed.

To address the above-mentioned challenges, this paper is the first study on energy-efficient collaborative adaptation for concurrently executing performance constrained applications on CPU and GPU cores of a mobile SoC, and makes the following contributions:

1) A collaborative adaptation approach that performs an energy-efficient mapping and thread-partitioning between CPU and GPU cores for concurrent applications, and applies energy optimizing adaptations (remapping and thread-repartitioning) at application completion and arrival through CPU and GPU collaboration.
2) To reduce adaptation overhead, for each application, energy-efficient mapping and thread-partitioning when using a different number of CPU core are explored at design-time and stored for using at run-time. Additionally, the best number of chunks for each application is also explored at design-time and execution starts by feeding chunks to CPU and GPU continuously by collaborating on their processing capabilities.
3) Implementation and experimental validity of the proposed approach on an Odroid-XU3 platform [16] (contains a state-of-the-art mobile SoC prevalent in tablet/smart-phone devices).

The structure of this paper is organized as follows. Section 2 provides a motivational case-study for performing adaptation in CPU-GPU mobile SoC. Section 3 presents related works. The proposed adaptive approach while highlighting the target problem is presented in Section 4. Section 5 presents the experimental set-up and results while including details of the considered hardware and software infrastructure. The concluding remarks are provided in Section 6.

2 Motivational Case Study: Adaptation in CPU-GPU Mobile SoC

We present an experimental case study to illustrate the potential of adaptation *i)* when an application completes execution and frees cores, and *ii)* when an application arrives, but there is no free core.

2.1 Adaptation at Application Completion

Fig. 1 illustrates the mapping and thread-partitioning process of applications Symmetric rank 2k (SYR2K) and Symmetric rank k (SYRK) from the Polybench benchmark [17]

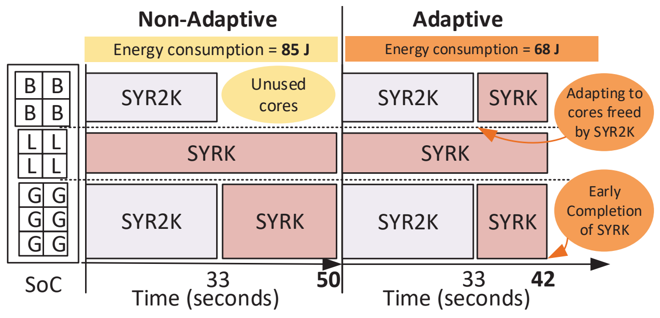

Figure 1. Adaptation at application completion: Non-adaptive vs. adaptive mapping and thread-partitioning on big (B), LITTLE (L) and GPU (G) cores of Samsung Exynos 5422 mobile SoC.

on the Samsung Exynos 5422 SoC when employing non-adaptive (e.g., [7], [15]) and our proposed collaborative adaptation approaches. To begin execution, the threads of SYR2K are enqueued to big (B) and GPU (G) cores, and to LITTLE (L) and GPU (G) cores for SYRK. Since embedded GPU drivers do not support spatially isolated and time multiplexed execution of multiple applications, SYR2K uses all the GPU cores until its completion. Thereafter, SYRK starts using the GPU cores. Both the applications are started with appropriate thread-partitioning between CPU and GPU cores by taking CPU and GPU processing capabilities into account, i.e., by having proper collaboration between CPU and GPU. It can be seen in Fig. 1 that the non-adaptive approach leaves cores unused when SYR2K completes execution. This has resulted in a higher execution time of 50 seconds for SYRK and a total energy consumption of 85 joule (J) for both the applications. In contrast, upon completion of SYR2K, the adaptive approach performs adaptation (through collaboration between CPU and GPU cores), i.e. remapping SYRK by taking the freed cores into account, and execution is started with a new thread-partitioning that enqueues more threads to CPU cores as it has increased processing capability. This leads to early completion of SYRK at 42 seconds. As a result of reduced execution time, total energy consumption is reduced to 68 J, i.e. by 20%.

2.2 Adaptation at Application Arrival

Fig. 2 illustrates the non-adaptive and adaptive mapping and thread-partitioning process when application CORRELATION (CR) arrives around 15 seconds after the start of SYR2K and SYRK. Initially, execution of SYR2K and SYRK starts as in Fig. 1. When CR arrives, it cannot be started immediately while following the non-adaptive approach as there is no available core. Rather, it starts execution when SYR2K releases cores after completion, which is not desired by the end users. Further, freed cores by SYRK are not used by CR. This has resulted in a higher execution time of 54 seconds for CR and a total energy consumption of 110 joule (J) for all the three applications. In contrast, by the adaptive approach, application CR is started immediately after its arrival by adapting running applications to lower number of cores. The approach chooses the running application that is over performing and releases its cores while updating the design point in terms of mapping and partitioning. Here, SYR2K is chosen to adapt to lower number of cores such

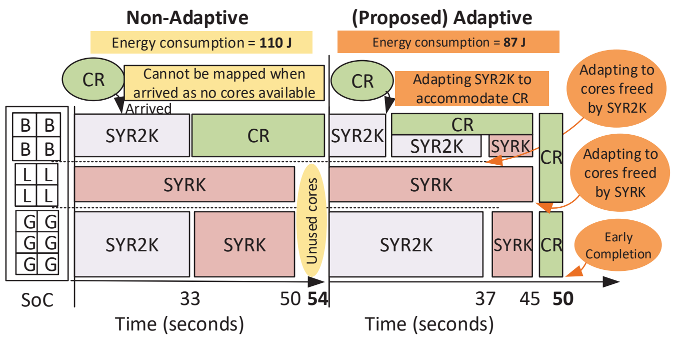
Non-Adaptive
Energy consumption = 110 J
(Proposed) Adaptive
Energy consumption = 87 J
CR
Cannot be mapped when arrived as no cores available
Arrived
Adapting SYR2K to accommodate CR
Adapting to cores freed by SYR2K
Adapting to cores freed by SYRK
Early Completion
B B B B
L L L L
G G G G G G
SoC
SYR2K
CR
SYRK
Unused cores
33
50
54
37
45
50
Time (seconds)
Time (seconds)

Figure 2. Adaptation at application arrival: Non-adaptive vs. adaptive mapping and thread-partitioning on big (B), LITTLE (L) and GPU (G) cores of Samsung Exynos 5422 mobile SoC.

that CR can be started when it arrived. The next collaborative adaptation is performed when application SYR2K completes execution at 37 seconds. The freed resources are used by SYRK. The last collaborative adaptation happens at the completion of SYRK at 45 seconds, where the freed resources are used by CR. The adaptation at each instance is performed through collaboration between CPU and GPU cores processing capabilities that are determined as the inverse of the time required to process the same number of threads on CPU and GPU cores, respectively.. This has resulted in early completion of CR at 50 seconds. The total energy consumption is also reduced to 87 J, i.e. by 21%, due to the reduction in execution time.

In summary, adaptation (remapping and repartitioning) at application arrival and completion brings several benefits such as the exploitation of freed cores, addition of a new application when no core is available and early completion of applications by exploiting freed cores, which also leads to lower energy consumption.

3 Related Works

To execute multiple applications concurrently in a heterogeneous mobile SoC, recently several efforts have been made, but they perform mapping and thread-partitioning of applications by assuming cores having the same instruction set architecture (ISA), e.g., big and/or LITTLE cores [9]–[14]. They have achieved energy efficiency by exploiting the appropriate cores. Usually, they employ Pthreads, which cannot be used to exploit cores of different ISAs such as CPU and GPU. Further, a close observation of these approaches indicates that most of them do not exploit more than one type of cores concurrently [9], [10], [12], [13]. Although the approaches in [11], [14] exploit big and LITTLE cores concurrently, exploitation of GPU cores is not possible as they handle instructions differently than the big and LITTLE cores.

For desktop SoCs containing CPU and GPU cores within the same chip, there have been some efforts to efficiently exploit the cores [18]–[22]. A run-time algorithm to partition the workload and power budget between CPU and GPU cores of an AMD Trinity SoC is proposed in [18]. Similar AMD SoC is used in [20] to perform coordinated CPU-GPU executions. However, due to access of the same memory bank in different patterns by the CPU and GPU, it results in memory contention. The problem of shared resources in AMD SoCs is addressed in [19]. In [21], Intel Haswell and Atom SoCs are considered to partition the workload between CPU and GPU cores for reducing energy consumption, but concurrent applications are not considered. In [22], criticality of GPU accesses is used to accelerate an application execution. However, these approaches do not comply with the limited power budget that is typical for embedded systems and coordination between CPU and GPU cores in such SoCs needs different kinds of consideration than mobile SoCs.

For desktop platforms, there has also been efforts to simultaneously exploit CPU and GPU cores, but the CPU and GPU cores are situated into different chips [4]–[6], [8], [23]. However, most of these consider static mapping [4], [5] and thus they cannot cater for run-time changing scenarios, and others suitable for run-time scenarios [6] do not concurrently exploit CPU and GPU cores. Further, CPU and GPU cores have separate memory in contrast to an embedded mobile SoC. Therefore, communication support between CPU and GPU cores is different and they cannot be efficiently applied to mobile SoCs.

For mobile SoCs containing CPU and GPU cores, some relevant efforts have been made with the use of OpenCL [7], [15], [24], [25]. To achieve energy efficiency, workloads are executed on Mali GPU cores in [24], but a potential collaboration with CPU cores is missing. In a similar way, GPU cores are not used for OpenCL kernel execution in [25]. However, threads representing the workload are partitioned by considering shared resources and synchronization. OpenCL framework for ARM processors was introduced in [26]. In [7], a similar open source framework, FreeOCL [27] is used for the ARM CPU that acts as both the host processor and an OpenCL device, enabling concurrent exploitation of CPU and GPU for an application threads. However, only a single application is considered at a time to perform static partitioning of threads between the CPU and GPU cores, and no adaptation happens. An adaptive approach for CPU-GPU SoC has recently been presented [28], but it considers only a single application at a time and only CPU or GPU for application execution. Other adaptive approaches have also been studied [29]–[32], but they consider only one type of core and mostly do not perform evaluation on a real mobile SoC [29], [30].

Mobile SoCs containing CPU and GPU cores are extensively exploited for mobile gaming and graphics workloads, [33]–[37]. However, these approaches have several drawbacks, e.g., consider a single application, no adaptation is employed and there is no control on the partitioning of application between CPU and GPU, i.e., it is fixed and well decided as all the graphics part of the application must be executed on the GPU while the remaining calculations have to be done on the CPU.

Our prior work considers execution of concurrent applications for mobile SoCs containing CPU and GPU [15], but the mapping and thread-partitioning for each application remain fixed, which is identified before starting applications' execution. This indicates no adaptation of the mapping and thread-partitioning during execution. Further, addition of a new application at run-time is not considered and thus run-time collaborative thread-partitioning is not performed. In

Table 1
Features of relevant existing and our approaches for executing concurrent applications on a CPU+GPU mobile SoC

Ref.	Concurrent Apps	CPU+GPU	Partition Control	Run-time App. Add.	Adapt.
[9]–[14]	✓	✗	✗	✗	✗
[33]–[37]	✗	✓	✗	✗	✗
[7]	✗	✓	✓	✗	✗
[15]	✓	✓	✓	✗	✗
Proposed	✓	✓	✓	✓	✓

contrast, the proposed approach performs energy-efficient adaptation (run-time remapping and thread-repartitioning) for concurrently executing applications at an application completion and at a new application arrival (addition) to start its execution as soon as possible in case no core is available. Table 1 shows a comparison of the features of the relevant existing approaches with our approach. It can be observed that only our approach considers adaptations for concurrently executing applications (Apps) on mobile SoCs containing CPU+GPU with the flexibility of run-time collaborative thread-partitioning between CPU and GPU (applied based on changes in execution scenarios) and run-time application addition (App. Add.).

4 Proposed Adaptive Approach

The proposed adaptive approach solves the following problem.

Given performance constrained applications to be executed at different moments of time on a heterogeneous CPU-GPU SoC supporting DVFS.

Optimize energy consumption (EC, computed by Equation 1) by performing efficient adaptation upon an application completion and start (arrival).

$$EC = EC_{CPU} + EC_{GPU} + EC_{MEM} \tag{1}$$

where, EC_{CPU}, EC_{GPU} and EC_{MEM} are the energy consumptions of CPU cores, GPU cores and memory, respectively, which can be computed as the product of respective power consumption and execution time. The power values are measured with the help of power sensors present for CPU, GPU and memory.

Subject to meeting performance requirement of each application within limited SoC resources.

For each application to be executed, the adaptation process needs to find energy-efficient mapping (used CPU cores and their frequency) and thread (referred to as work-item in OpenCL terminology) partitioning between CPU and GPU cores while enqueuing the chunks (a group of threads) continuously.

The proposed adaptive approach consists of two steps: *i)* offline profiling and *ii)* online mapping and partitioning for initial applications and adaptation by utilizing the profiled results.

4.1 Offline Profiling

Fig. 3 provides an overview of the offline profiling. It assumes that the applications are already installed into the system, for example in a mobile phone, and thus they can be profiled. For each available application (App_1 to App_m) and considered CPU-GPU mobile SoC (Samsung Exynos 5422), it provides *number of chunks*, *thread-partitioning*, *performance (1 / execution time)* and *energy consumption* at various combinations of used big (B), LITTLE (L) and GPU (G) cores and their operating frequencies. These combinations representing total design points (TDP) can be computed as follows:

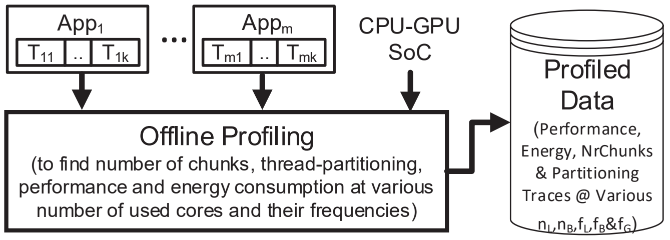
App1
T11
..
T1k
Appm
Tm1
..
Tmk
CPU-GPU SoC
Offline Profiling
(to find number of chunks, thread-partitioning, performance and energy consumption at various number of used cores and their frequencies)
Profiled Data
(Performance, Energy, NrChunks & Partitioning Traces @ Various nL,nB,fL,fB&fG)

Figure 3. Offline profiling.

$$TDP = \{(N_b \times F_b) + (N_L \times F_L) + (N_b \times F_b \times N_L \times F_L)\} \times \{1 \times F_g\} \tag{2}$$

where, N_b and N_L are the number of big and LITTLE cores, respectively. F_b, F_L and F_g are the number of voltage-frequency levels for big, LITTLE and GPU cluster, respectively. In Equation 2, $\{(N_b \times F_b) + (N_L \times F_L) + (N_b \times F_b \times N_L \times F_L)\}$ is the number of design points using big and LITTLE CPU cores and $\{1 \times F_g\}$ for GPU cores. For GPU cores, since all the cores are used by the application, the number of design points depends on the number of available voltage-frequency levels.

For each D_p ($\in TDP$) of an application, the computations of *number of chunks*, *thread-partitioning*, *performance (1 / execution time)* and *energy consumption* are performed as follows.

Number of chunks: The best number of chunks (NC) is identified by grouping all the threads of the application into several equal size chunks such that the used cores are neither starved nor overfed by enqueuing next chunk after the completion of the currently executed chunk and application execution time remains the same as that of enqueuing all the threads (or chunks) at the beginning. The starving situation occurs when the cores do not have enough threads to fully utilize them, i.e. at 100%. The cores are always fully utilized (100%) in overfed situations and there are more threads enqueued than can be processed by fully utilizing the cores. Fig. 4 demonstrates the process of finding the best number of chunks for SYR2K application when executed on the Samsung Exynos SoC with varying number of equal size chunks that contain all the application threads. The variation in the number of chunks defines the number of iterations and continues until the execution time in the current iteration exceeds the previous one. This helps to avoid unnecessary iterations. The maximum number of chunks (each contain a set of threads) leading to the same execution time (ET = 33 seconds) as that of enqueuing all the threads at the same time is 8 and chosen as the number of chunks, as highlighted in the figure. Such identification of the chunks avoids over and under feeding of threads for the considering design point. In Fig. 4, when the number of chunks is less than 8, it represents the case of overfeeding

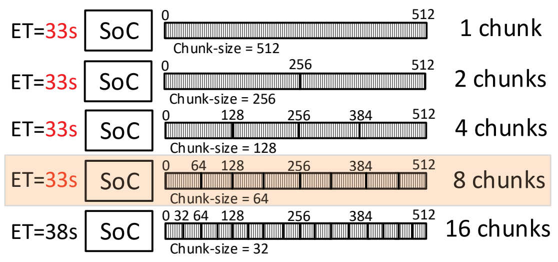

Figure 4. Finding number of chunks.

as more threads are enqueued than the SoC can process to achieve the same ET. With 16 chunks, the SoC is underfed, i.e. starves for the threads as chunk does not contain enough threads to fully utilize the cores and thus the execution is extended to 38 seconds. This indicates that 8 chunks represent the case of just enough feeding to achieve an execution time of 33 seconds.

This facilitates to achieve the same execution time while providing opportunity for required adaptation in terms of partitioning threads within a chunk between the CPU and GPU. Further, it helps to keep a track of the number of chunks or threads completed and remained for execution. To avoid memory inconsistency issues between work-items, each chunk contains many work-groups (a group of work-items) and thus the chunk-size CS in Fig. 4 represents the number of work-groups.

Thread-partitioning: To achieve minimum execution time for each design point D_p ($\in TDP$), the partitioning can be performed by having collaboration between CPU and GPU, which can be based on their individual processing capacities, and measured as the inverse of time taken to complete execution of all the work-groups on the CPU-only and GPU-only, respectively. Let WG be the total number of work-groups in the application, $ET_{CPUO_{D_p}}$ and $ET_{GPUO_{D_p}}$ are the execution times for CPU-only and GPU-only executions, respectively. Then, the portion K_{D_p} of the work-groups that should be executed on the CPU can be computed by assuming the completion on both CPU and GPU at the same time [7], [15], i.e.:

$$K_{D_p} \times ET_{CPUO_{D_p}} = (WG - K_{D_p}) \times ET_{GPUO_{D_p}}$$
$$i.e., K_{D_p} = \frac{WG}{1 + \frac{ET_{CPUO_{D_p}}}{ET_{GPUO_{D_p}}}} \tag{3}$$

To facilitate adaptations, enqueuing of work-groups takes place at chunk level and thus the thread-partitioning between CPU and GPU for a chunk needs to be defined. Since each chunk contains the same number of work-groups, the portion of the chunk CK_{D_p} to be executed on the CPU can be computed as:

$$CK_{D_p} = \frac{K_{D_p}}{NC} \tag{4}$$

Thus, the portion on the GPU would be $= CS - \frac{K_{D_p}}{NC}$. For the identified number of chunks of application SYR2K (demonstrated in Fig. 4) as 8, Fig. 5 provides example demonstration of thread-partitioning. Each chunk containing a total of 64 work-groups is partitioned between the CPU and GPU equally, i.e. both CPU and GPU are enqueued with 32 work-groups for each chuck.

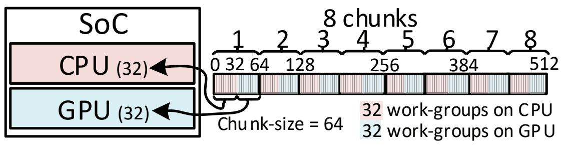

Figure 5. Thread-partitioning of chunks.

Performance (1 / execution time): The execution time by enqueuing the work-groups based on the identified number of chunks as earlier and by enqueuing all the work-groups as one chunk is the same as demonstrated in Fig. 4. Therefore, to maintain simplicity, we have computed it based on one chunk. To take contentions between CPU and GPU into account, it is computed as follows:

$$ET_{D_p} = max\{K_{D_p} \times ET_{CPUO_{D_p}}, (WG - K_{D_p}) \times ET_{GPUO_{D_p}}\} \tag{5}$$

This indicates that ET_{D_p} is governed by the device taking longer time to complete the execution of assigned work-groups.

Energy consumption: Application energy consumption for D_p, i.e. EC_{D_p}, is computed by following Equation 1.

Similarly, computations (NC, CK_{D_p}, ET_{D_p}, and EC_{D_p}) are performed for other design points ($\in TDP$) for each application. This results in generation of the profiling data. Considering all the used terms, each design point D_p of an application is represented and stored as an 11-tuple:
$D_p = (Prf, EC_{CPU}, EC_{GPU}, EC_{MEM}, NC, CK, n_b, f_b, n_L, f_L, f_g)$

where, Prf is performance, EC_{CPU}, EC_{GPU} and EC_{MEM} are energy consumptions of CPU, GPU and memory, respectively, NC is the number of chunks, CK is the chunk portion on CPU, n_b and n_L are the number of used big and LITTLE cores, respectively, and f_b, f_L and f_g are frequencies of big, LITTLE and GPU cores, respectively. Since the number of design points can be huge, they are distilled using the techniques of [15] so that only efficient points are stored and storage overhead is reduced. Towards this, if execution time and energy consumption of a point using higher number of cores are the same or smaller than that of a point using lower number of cores, then the former point is discarded. All the design points for each application (App_1 to App_m) are stored in descending order of Prf and shown as Profiled Data (Performance, Energy, NrChunks & Partitioning Traces @ various n_L, n_b, f_L, f_b & f_G) in the storage database of Fig. 3. For further illustration, Table 2 shows some real samples of the stored profiling results for SYR2K application. When a core type is not used, e.g. n_b equals to 0, the corresponding frequency is denoted as X. Storing the profiling results sorted by Prf leads to low complexity search of points meeting a certain level of performance. The offline profiling and storing can be avoided by employing the best effort or online learning heuristics [10], [11], [38] to find a design point at an adaptation instance, but online complexity will increase and achieved results might not be efficient.

Table 2
Profiling results for SYR2K

$Prf(1/seconds)$	$EC_{CPU}(J)$	$EC_{GPU}(J)$	$EC_{MEM}(J)$	NC	CK(work-groups)	n_b	$f_b(MHz)$	n_L	$f_L(MHz)$	$f_g(MHz)$
0.056	85.742	5.266	3.340	8	32	4	2000	4	1400	177
0.054	86.642	5.462	3.324	8	32	3	2000	2	800	177
0.050	95.093	5.142	3.558	8	32	4	2000	3	1400	177
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.007	11.809	43.915	8.770	256	32	0	X	1	800	600

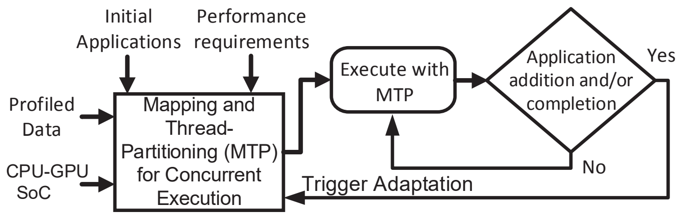

Figure 6. Adaptive mapping and thread-partitioning.

4.2 Online Adaptation

Fig. 6 provides an overview of the online (run-time) mapping and thread-partitioning with adaptation (remapping and thread-repartitioning) while utilizing the profiling data. First, it performs mapping and thread-partitioning of applications that arrived at time t = 0, i.e. initial applications that a user can start with. The adaptation process triggers at an application addition and/or completion. It could lead to a new mapping and thread-partitioning (MTP), computed by remapping and thread-repartitioning, of executing applications if beneficial in terms of energy, otherwise execution is continued with earlier MTP. The detailed steps are presented in Algorithm 1, where mapping/thread-partitioning of initial (line 1) and newly added applications (line 9) and adaptation (lines 16 and 31) is performed as follows.

4.2.1 *Mapping and Partitioning for Initial Applications*

The mapping of applications' threads to cores and thread-partitioning (MTP) is identified by taking the performance requirements and the design points (profiled data sample shown in Table 2) generated in the previous step into account, as shown in Fig. 6. As mentioned earlier, the mapping is defined as the number of used cores, their types (e.g., big, LITTLE) and operating frequencies, i.e. n_b, f_b, n_L, f_L and f_g, whereas thread-partitioning defines portion of work-groups (CK) of each chunk ($\in NC$) to be executed on CPU. To identify the MTP, each initial application's stretched execution due to co-scheduling needs to be considered. First, performance requirement satisfying points is chosen for each application from its storage design space, e.g., from Table 2 for SYR2K application. Then, for each combination of design points, CDP, formed by considering one point from each initial application such that the total number of used CPU cores is less than or equal to the number of available CPU cores and individual frequencies of used big, LITLLE and GPU cores are the same, each application's stretched execution time SET due to co-scheduling, new partitioned workload (work-groups) K', total energy consumption ($EC^{K'}$) and execution time ($ET^{K'}$) are computed as follows.

SET: Since spatially isolated CPU cores are chosen for each application, the execution time on the CPU side remains almost the same. However, on the GPU side, it is stretched by the time taken to complete earlier enqueued application threads as existing embedded GPU driver doesn't support spatial and time-multiplexed execution of applications. Considering i initial applications, App_1 to App_i, are enqueued sequentially, the stretched execution time of App_i on the GPU cores is computed as follows [15].

$$SET_{App_i} = \sum_{a=App_1}^{App_i} ET_{GPU_a} \tag{6}$$

where,

$$ET_{GPU_a} = ET_{CPU_a} = \frac{1}{Prf_{a_{D_p}}} \tag{7}$$

It indicates that balanced execution was performed between CPU and GPU during offline profiling through collaboration.

K': To balance the execution between CPU and GPU cores while considering SET, K' is computed by having collaboration between CPU and GPU, which takes their processing capabilities into account, as follows [15].

$$\begin{aligned} K' \times ET_{CPU_{owg}} &= (WG - K') \times ET_{GPU_{owg}} \\ i.e., K' &= \frac{WG}{1 + \frac{ET_{CPU_{owg}}}{ET_{GPU_{owg}}}} \end{aligned} \tag{8}$$

where, $ET_{CPU_{owg}}$ and $ET_{GPU_{owg}}$ are the time required to process one work group (owg) on the CPU and GPU cores, respectively, and can be computed as follows.

$$\begin{aligned} ET_{CPU_{owg}} &= \frac{ET_{CPU_{App_i}}}{K} \\ ET_{GPU_{owg}} &= \frac{SET_{App_i}}{WG - K} \end{aligned} \tag{9}$$

$ET_{CPU_{App_i}}$ can be computed by Equation 7 as CPU side execution is not stretched.

As mentioned earlier, since enqueuing is performed at chunk level to enable adaptations, the portion of each chunk to be executed on the CPU is computed as:

$$CK' = \frac{K'}{NC} \tag{10}$$

$EC^{K'}$ is computed as:

$$EC^{K'} = K' \times \frac{EC_{CPU}}{K} + (WG - K') \times \frac{EC_{GPU}}{WG - K} + EC_{MEM} \tag{11}$$

where, EC_{CPU}, EC_{GPU} and EC_{MEM} are obtained from the chosen design point of the application.

Algorithm 1: Run-time Mapping/Thread-partitioning and Adaptation

```
Map/Partition Initial Applications (Section 4.2.1) ;
while 1 do
    if any executing_application(s) have completed then
        $Adapt1$ = true; Update SoC resources;
    end
    if any application(s) A arrived (need to be added) then
        $A.Adapt2$ = true; Put applications(s) in AppQueue;
    end
    if AppQueue.size() > 0 then
        for each application $a \in$ AppQueue do
            Find $required\_cores$, $M$ and $K$ of $a$;
            if $required\_cores <= available\_cores$ then
                // Perform Allocation
                Execute $a$ based on $M$ and $K$ ;
                Update SoC resources;
            end
            else if a.Adapt2 == true then
                for each executing application ea do
                    while $ea$ is over-performing do
                        Consider releasing one core of $ea$
                          and update its $M''$ and $K''$ ;
                        if $released\_cores ==$
                          $required\_cores$ then
                            Execute $a$ based on $M$ and $K$;
                            Update SoC resources;
                            break;
                        end
                    end
                end
                $a.Adapt2$ = false;
            end
        end
    end
    if Adapt1 == true then // Try Adaptation
        for each executing application ea do
            Find $RC$, $RT$ and $REC$;
            Find $M'''$ and $K'''$ based on fair sharing of
              resources with other executing applications
              and $REC'/REC$;
            For $M'''$ and $K'''$, estimate $RT'$;
            if
              $RT' + M_{o\_t} < RT$ && $REC' + M_{o\_e} < REC$
              then
                Execute $ea$ by following $M'''$ and $K'''$;
                Update SoC resources;
            end
        end
        $Adapt1$ = false;
    end
end
```

$ET^{K'}$ is computed as:

$$ET^{K'} = \max\{K' \times ET_{CPU_{owg}}, (WG - K') \times ET_{GPU_{owg}}\} + \mu \quad (12)$$

where, μ is memory contention overhead due to concurrent execution.

After performing above computations, total energy consumption for all the concurrent applications is computed by adding individual application's energy consumption. Then, these computations are repeated for all the combination of design points. Finally, the combination point having minimum energy consumption and satisfying the performance requirement of each application is chosen. For the chosen point of each application, the number of used cores, their types and operating frequencies are returned as the thread-to-core mapping and CK' as the chunk partition to start the execution of initial applications by using *sched_setaffinity* interface in the Linux scheduler while enqueuing applications' chunks continuously. In addition to energy efficiency, choosing such a point also results in performance improvement of the applications (shown in Section 5.2.2), i.e. $Prf(1/Execution\ Time)$ is greater than the performance requirement.

As a demonstration, for starting the execution of initial applications SYR2K and SYRK based on the chosen design points as in Fig. 1 (4 Big and GPU cores for SYR2K, and 4 LITTLE and GPU cores for SYRK), NC for both SYR2K and SYRK is 8 (demonstrated in Fig. 4), and CK is 32 (Fig. 5) and 16 respectively. Due to their co-scheduling, SET of SYRK is 71 (33+38 based on Equation 6). This results in K' of 384 and thus CK' of 48, which is the number of work-groups of each chunk to be enqueued and executed on CPU cores for SYRK. However, since SYR2K occupies the GPU first, its partitioning remains the same as earlier. Without any adaptation, the overall energy consumption is 85J and execution time of SYR2K and SYRK is 33s and 50s, respectively, as shown in Fig. 1.

4.2.2 Run-time Adaptation

For adaptation, Algorithm 1 checks for the following two events: *i)* any application(s) has completed execution (line 3) to set $Adapt1$ flag as true and update cores availability/unavailability (line 4) to maintain accurate status of resources, *ii)* any application(s) needs to be added (line 6) to put application in $AppQueue$ (line 7). Therefore, there are two possible instances of adaptation in the following order: *i)* at application addition (arrival) and it needs more cores than available ones (line 16) and *ii)* at application completion (line 31). The adaptation process at application addition (line 16) is carried out first so that arrived applications can be started as soon as possible for a better user experience. The adaptation at application completion (line 31) is performed to further explore the energy saving opportunities for executing applications. Details of adaptation steps are as follows.

Adaptation at application addition (arrival)

For each queued job (line 10), first allocation is tried on the available cores so that unnecessary adaptations are avoided (line 12). Therefore, if there are enough available cores to satisfy the requirements, the execution is continued by selecting the cores and their frequency (M) and thread-partitioning (K) leading to minimum energy consumption while following the profiled data (line 13) and SoC resources are updated. Otherwise, adaptation of executing applications is tried (line 17) to allocate the queued application(s). However, it is tried only when queued application is just added so that overheads of trying for unnecessary multiple adaptations for the same application can be avoided. To perform adaptation, performance of each executing application is checked to find if it is over-performing (line 18), i.e. its Prf $(1/ET)$ is greater than the performance requirement. If so, one of its core is released while updating the mapping M'' and partitioning K'' (line 19). M'' contains one core less compared to earlier mapping, but at the same frequencies. Based on M'' (defines used cores, their types and frequencies), CK is chosen from the profiling data as K''. The same process continues while the executing application over-performs and sufficient cores to support the arrived application are released (line 20). Then, the arrived application is executed on the required number of cores by following M and K from the profiled data while ensuring that the cores will have the same frequency as in M'' in case they are used by an executing application. Thereafter, the SoC resources are updated. In case sufficient cores cannot be made available, the arrived application resides in the queue and allocated on the freed cores (as available cores) by the executing applications.

Adaptation at application completion

At application completion, the freed cores are tried to be exploited by executing applications. For each application (line 32), the following parameters are computed.

Remaining chunks (RC): Each executing application writes the number of RC as $(NC - NrProcessesChunks)$ to a shared memory location. Therefore, NC is read from the respective shared location.

Remaining time (RT):

$$RT = \frac{RC}{ATOC} \tag{13}$$

where, $ATOC$ is the average time to process one chunk and is computed as inverse of number of processed chunks divided by processing time.

Energy consumption to process RC (REC):

$$REC = RC \times EC_{oc} \tag{14}$$

where, EC_{oc} is average energy consumption to process one chunk (oc) based on current mapping and partitioning, and can be computed by observing energy consumption of earlier processed chunks.

Mapping (M''') and thread-partitioning (K'''): M''' determines the number of used cores such that other executing applications also get the same number of cores from the freed pool to enable adaptation opportunity for each application. In case not enough cores are available for equal share, energy gain (REC'/REC) of each application due to adaptation to a higher core number is computed and the applications achieving higher energy gains are given the freed cores. M''' may also include frequency of used cores such that REC'/REC is maximized while considering an appropriate collaborative partitioning K''' computed as follows (like Equation 8).

$$K''' = \frac{RC}{1 + \frac{ET_{CPU_{oc}}}{ET_{GPU_{oc}}}} \tag{15}$$

where, $ET_{CPU_{oc}}$ and $ET_{GPU_{oc}}$ are the executions time to process one chunk (*oc*) on CPU and GPU, respectively, and can be computed as in Equation 9.

Remaining energy consumption (REC'): Based on M''' and K''', it can be computed as in Equation 11.

New Remained Time (RT'): can be computed as:

$$RT' = \frac{RC}{ATOC'} \tag{16}$$

where, $ATOC'$ is the average time to process one chunk and can be chosen from the profiled data based on M''' and K'''.

After above computations, it is checked if adaptation is beneficial or not while taking the migration overhead in terms of time (M_{o_t}) and energy (M_{o_e}) into account. The migration overheads for all the possible migrations, e.g. migrating from two to four cores, are computed offline to keep a low run-time overhead. If adaptation is beneficial (line 36), the application is executed by following M''' and K'''. Otherwise, it is continued with earlier mapping and partitioning. It can be noticed that mapping and partitioning is always achieved by having collaboration between CPU and GPU processing capabilities. Further, our approach is generic, but one-time profiling is required when the application or platform changes. This implies that when a new application is installed on a new mobile device, one-time profiling is desired.

5 EXPERIMENTAL RESULTS

5.1 Experimental Set-up

5.1.1 Heterogeneous CPU-GPU Mobile SoC Hardware

In modern CPU-GPU mobile SoCs, usually, cores of the same type are situated within a cluster and their number could vary, e.g., big (B), LITTLE (L) and GPU (G) cores shown in Fig. 1. In particular, we consider the Samsung Exynos 5422 SoC [1], which is present on the Odroid-XU3 board [16]. The SoC is based on ARM's big.LITTLE technology [39] and contains three clusters: one with four ARM Cortex-A15 (big) CPU cores, one with four ARM Cortex-A7 (LITTLE) CPU cores and another with six ARM Mali-T628 GPU shader cores. The SoC provides DVFS per cluster, where the big cluster can operate from 200 MHz to 2000 MHz and the LITTLE cluster between 200 MHz and 1400 MHz, with a step-size of 100 MHz. The GPU cluster can operate at 177 MHz, 266 MHz, 350 MHz, 420 MHz, 480 MHz, 543 MHz and 600 MHz. It should be noted that, with variation in frequency, the firmware automatically adjusts the voltage based on preset pairs of voltage-frequency values.

5.1.2 Software for CPU-GPU Mobile SoC

Operating System

We used Ubuntu 14.04 LTS that enables use of all the CPU cores (big and LITTLE) simultaneously by Heterogeneous Multi-Processing (HMP). Additionally, it supports DVFS by editing relevant virtual files of devices in the *sysfs* directory, and core disabling of CPU cores to use selective big and/or LITTLE cores for an application.

OpenCL and FreeOCL

To exploit heterogeneous multi-core architectures containing cores of two different ISAs such as CPU and GPU, data-parallel applications are potential candidates and they can be developed using the open standard of Open Computing Language (OpenCL) [2] [40], [41]. It also supports exploitation of multiple devices, e.g., CPU and GPU, by a single program. In OpenCL context, a computing system consists of many devices attached to a host (controller) processor that is usually a CPU. The threads (computations) are referred to as kernels, where a kernel instance is called as work-item that operates on a single data point. A group of work-items form a work-group. The OpenCL memory model does not demand memory consistency among work-groups, therefore, they can be launched on different devices (e.g., CPU and GPU). In additional to data-parallel applications, other application models, e.g., data-flow graphs [42] can also be considered, but they need to be translated to OpenCL to exploit both CPU and GPU.

FreeOCL [27] is an open-source runtime library that provides OpenCL support for the ARM CPU cores. Such support is typically not available in current mobile SoCs as OpenCL runtime software is supplied only for the Mali GPU to promote its usage for general purpose computing or acceleration. It enabled ARM CPU to act as both host processor and an OpenCL device, and thus, both CPU and GPU cores can be concurrently exploited for executing an application.

OpenCL Applications

We used the GPU version of the popular Polybench benchmark suite [17], which contains several data-parallel applications from various application domains. Specifically, we considered the following.

- Data mining applications: CORRELATION (CR) and COVARIANCE (CV).
- Linear algebra kernels: 2 matrix multiplications (2M) and Matrix vector multiplication (MV).
- Basic linear algebra subprograms (BLAS) Routines: Generalized matrix multiply (GE), Symmetric rank k (SR) and Symmetric rank 2k (S2).
- Deep learning application: two-dimensional convolution (2D).

The applications CR, CV, 2M, MV, GE, SR, S2 and 2D have 2048, 2048, 128, 4096, 512, 512, 512 and 2048 work-groups, respectively. They form a diverse set of representative applications, containing varying number of kernels and have varying execution times. The application codes are slightly modified to launch them only on CPU cores, only on GPU cores, or on both CPU and GPU cores.

C/C++ Mobile benchmarks

To diversify evaluations, we also considered mobile benchmarks, specifically FFmpeg (FF) that is used for video/audio processing [43] and VideoLAN Client (VLC) that is a cross-platform media player and streaming media server [44]. Most part of the code for these applications is implemented in C/C++. These applications use only CPU cores. However, they can use GPU cores with required translation to an embedded GPU programming language such as OpenCL.

Each application has a randomly assigned performance requirement that represents required completion time of the application. For frame based application like audio/video processing, the timing requirement can be translated to throughput requirement, where throughput is expressed as a frame rate to guarantee a good user experience. Similarly, it can also be translated to instructions per second (IPS) requirement considering the total number of instructions in an application is known.

The applications are considered in various random mixes to make a representative set of concurrent applications and any of them can be added to the SoC at run-time. Such scenarios can be observed in a mobile phone where user tries to run more applications at the same time, e.g., internet browser and mp3 player, and jpeg decoding at the next moment. The contention overhead (μ in Equation 12) due to concurrent execution is computed as deviation in applications' performance when run individually and concurrently, is experimentally found to be 3.8% in the worst-case.

5.2 Comparison Results

The comparison candidates based on their closeness to our approach are:

1) *CPU or GPU for Mapping [8] (CoGM)*: The device taking lower time is chosen for mapping an application. A comparison with it shows the potential of jointly using both the CPU and GPU. Since it might end up mapping all the applications on one device, to make a fair comparison, the applications are distributed between CPU and GPU to achieve lower overall execution time.
2) *CPU for Mapping and Adaptation [31] (CMA)*: Big and LITTLE CPU cores are used for mapping and adaptation (remapping). To make a fair comparison, adaptation happens only at application completion and arrival.
3) *Individual Application based Mapping and Partitioning [7] (IAMP)*: It follows the same mapping and partitioning that is achieved by considering individual applications.
4) *Concurrent Application based Mapping and Partitioning [15] (CAMP)*: The mapping and partitioning is performed for initial concurrent applications based on stretching on GPU (Section 4.2.1).

None of these approaches consider adaptation by using both CPU and GPU cores, unlike our collaborative adaptation approach, which has been referred to as adaptive mapping and thread-partitioning (*AdaptMTP*). For each approach, the runtime management thread is pinned to one of the big (Cortex-A15) cores. Further, the offline profiling information is made available to all the approaches for a fair comparison.

5.2.1 Energy consumption

Energy consumption is computed by measuring the power consumption from on-board power sensors of Odroid-XU3 every 100ms. A power measurement circuitry present in the SoC estimates the power as: voltage $\times$ current, where four real time current/voltage sensors are used for four power domains: big, LITTLE, GPU and DRAM. This allows us to compute energy consumption of all the software components, e.g., adaptive algorithm, profiled data, OS, drivers, applications, etc., running in the SoC.

The energy consumption benefits due to adaptation is evaluated for several run-time scenarios under the following two cases.

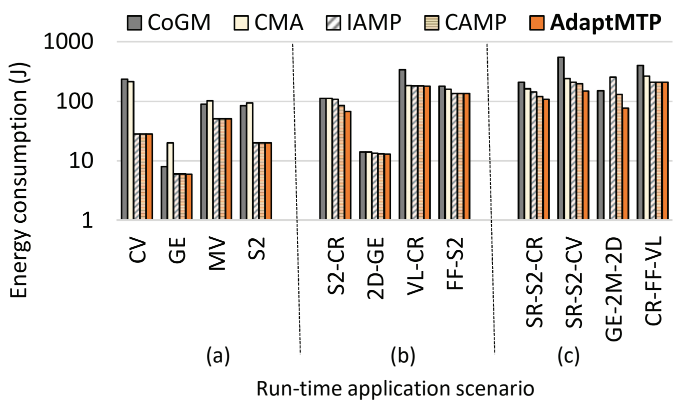

Figure 7. Energy consumption by various approaches for different run-time scenarios: (a) single, (b) two and (c) three concurrent applications, without runtime application addition.

Without application addition

This evaluation shows energy saving benefits of our AdaptMTP approach when adaptation happens only for initial application(s), i.e., no application is added during the execution of initial applications. Fig. 7 provides the energy consumption results when various approaches are employed for different run-time scenarios: (a) single, (b) two and (c) three initial concurrent applications. For two and three concurrent application scenarios in (b) and (c), respectively, it can be observed that our approach achieves lower energy consumption compared to existing approaches. This is due to beneficial adaptations at an application completion, where freed cores are allocated to the currently running application, if beneficial. However, for single application scenarios in (a), energy consumption is not reduced by AdaptMTP over IAMP and CAMP as there is no opportunity of adaptation. Since IAMP, CAMP and AdaptMTP achieve the same mapping and partitioning for each application, the overall energy consumption remains the same. Further, AdaptMTP leads to similar results as that of IAMP and CAMP when there is no opportunity of adaptation due to completion of applications around the same time, e.g., in scenario CR-FF-VL. It can also be observed that energy consumption is higher when running higher number of applications. Our approach also satisfies the performance requirement of each application in all the considered scenarios in Fig. 7. However, performance requirements are not satisfied by existing approaches in some cases and bar is made absent for them, e.g., in run-time scenario GE-2M-2D when CMA is employed.

With application addition

This evaluation shows energy saving benefits of adaptation for both initial and newly added application(s) at different moments of time. Fig. 9 provides energy consumption results when various approaches are employed for different run-time scenarios. The execution starts with the initial applications as in Fig. 7 and a random number of applications between one (+X) to three (+X*3) are added at different moments of time during the execution of initial applications. The added applications are amongst those mentioned in Section 5.1.2. A couple of observations can be made from Fig. 9. First, AdaptMTP outperforms all the existing approaches for all the run-time scenarios. As compared to

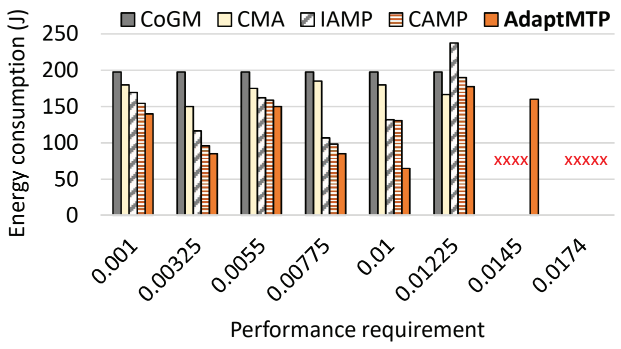

Figure 8. Energy consumption at varying performance constraints

Fig. 7, the energy savings by AdaptMTP are more significant as adaptation opportunities are exploited for more number of applications. Second, for some scenarios (marked by red X), the additional application cannot be allocated either due to unavailability of resources or performance satisfying mapping and partitioning is not found. Further, performance requirements are not satisfied by existing approaches in some cases and bar is made absent for them, e.g., for 2D-GE+X*2 scenario when CMA is employed. In such cases, the additional application is placed in the application queue for later execution when resources become available. These scenarios typically arise due to a higher number of initial concurrent applications, e.g. Fig. 7 (c), where resources are fully occupied by the applications while satisfying their performance requirements. This also indicates that a large number of applications cannot be executed concurrently without violating the performance requirements due to limited SoC resources.

On an average for the above two cases (Fig. 7 and Fig. 9), AdaptMTP achieves energy savings of 37% when compared to the most promising existing approach CAMP.

Effect of Varying Performance Constraints

We also analysed the effect of varying performance constraints on energy consumption when employing AdaptMTP and existing approaches. Figure 8 shows energy consumption results when performance constraints of concurrent applications SR, S2 and CR are varied from 0.001 to 0.0145. With a higher value of performance constraint, e.g., 0.0145, all the approaches fail to satisfy the constraint (xxxx in Figure 8), but AdaptMTP satisfies the constraint. However, with further increase in the value of the performance constraint, e.g., 0.0174 and beyond, none of the approaches satisfy the constraint. It can also be observed that AdaptMTP always provides energy savings over existing approaches, i.e., it has lower energy consumption. This indicates that AdaptMTP outperforms existing approaches for any chosen performance constraint.

5.2.2 *Performance*

Adaptation also leads to performance (1 / execution time) improvement over existing approaches as demonstrated earlier in Fig. 1 and Fig. 2. Fig. 10 shows normalized average performance improvement of AdaptMTP over existing approaches when applied to various run-time scenarios. AdaptMTP achieves better performance by performing beneficial adaptations. The achieved performance values are

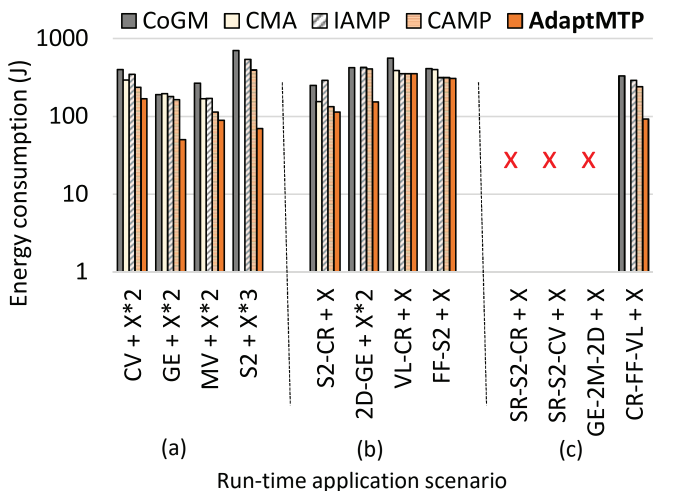

Figure 9. Energy consumption by various approaches for different run-time scenarios: (a) single, (b) two, and (c) three initial concurrent applications, and random number (X) of applications added at different moments of time in each scenario.

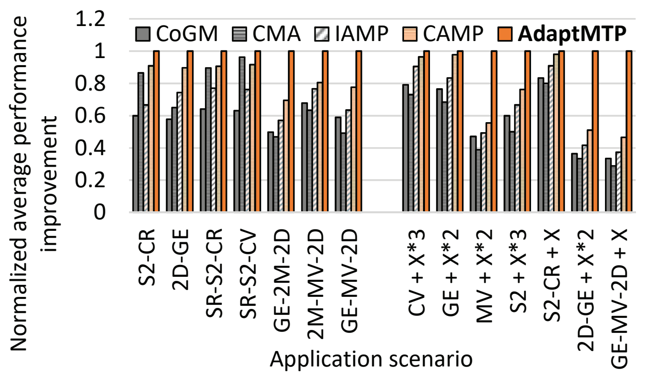

Figure 10. Performance improvement in different run-time scenarios.

normalized with respect to (w.r.t.) the AdaptMTP approach. A better performance by AdaptMTP also helps to achieve low energy consumption, which is computed as the product of average power consumption and execution time. However, in some scenarios, lower energy consumption is achieved by performing execution at lower frequencies, leading to low average power consumption.

5.3 Offline Profiling

The profiling time for each application depends on the total number of design points (TDP) to be evaluated. TDP can be computed by Equation 2 provided N_b, N_L, F_b, F_L and F_g are known. For the considered mobile SoC, N_b, N_L, F_b, F_L and F_g are 4, 4, 19, 13 and 7, respectively. Therefore, $TDP = \{4\times19 + 4\times13 + 4\times19\times4\times13\} \times 7 = 28560$. For each design point of an application, the profiling consists of finding the best number of chunks (NC), thread-partitioning (CK), performance (Prf) and energy consumption (EC) as described in Section 4.1. The time to find the NC depends upon the number of iterations taken and execution time of the application in each iteration, as explained in Section 4.1. The value of CK for NC is computed by equation 4 in the order of μs. When the application completes execution with the identified NC and CK, Prf is computed as the inverse of execution time (ET) and EC by Equation 1, in the order of μs. Out of all these times, time to find the NC dominates as whole application execution needs to be completed in each iteration. Since the number of design points are huge (28560) and such computations have to be performed for each design point of each application, we considered step-size of frequency as 200MHz instead of 100MHz, i.e. values of F_b and F_L as 10 and 7, respectively, which resulted in only 8316 total design points (computed by Equation 2). This includes usage of only CPU, only GPU and both CPU and GPU. For all the applications, whole profiling took around 90 hours. Since it is one time process and its usage leads to energy savings and performance improvement as shown earlier, it is suitable to employ.

The profiled data (design points) for each application is stored for using at run-time, as shown in Table 2 for SYR2K. As described in Section 4.1, only the efficient design points are stored by discarding the points using higher number of cores and leading to lower performance and higher energy consumption as compared to a design point using lower number of cores. The storage overhead for each application is 11.5 kB, which is quite low and thus suitable for a mobile SoC. Further, the energy required to store the profiled data is included in the energy consumption evaluation in Section 5.2 as it includes energy consumption of the memory. This indicates it is worth investing in such small overheads to achieve energy efficiency.

In case the architecture is large, the number of design points and their storage overhead might become huge. In such cases, regression model can be derived by using some of the design points and rest can be achieved by using the model [10], [11]. The storage overhead to store the model will be low, but results will not be as accurate as that of storing the design points.

5.4 Online Overheads

5.4.1 *Initial applications mapping and thread-partitioning*

The run-time overhead for mapping and thread-partitioning of initial applications depends on the number of CDP considered for the concurrent initial applications and time taken to perform various computations (e.g., SET, K', CK', $EC^{K'}$ and $ET^{K'}$) for each application. These computations are done in the order of μs. The number of CDP depends on the number of applications nAa, number of big and LITTLE cores (N_b and N_L) and frequency levels of big, LITTLE and GPU cores (F_b, F_L and F_g). Therefore, overall complexity is $O(nAp.N_b.N_L.F_b.F_L.F_g)$. The average timing overhead over various run-time scenarios appears to be approximately 20 ms, which is quite small. The average energy overhead computed as the product of average power consumption of the core executing AdaptMTP to find the mapping/thread-partitioning and timing overhead is 13mJ (0.65×20), which is significantly low compared to the average overall energy consumption of around 178J, shown in Fig. 7 and 9.

5.4.2 *Adaptation*

Fig. 11 shows overall adaptation overhead (%) in terms of timing and energy for various run-time scenarios w.r.t. total execution time and energy consumption, respectively. The average total execution time and energy consumption for

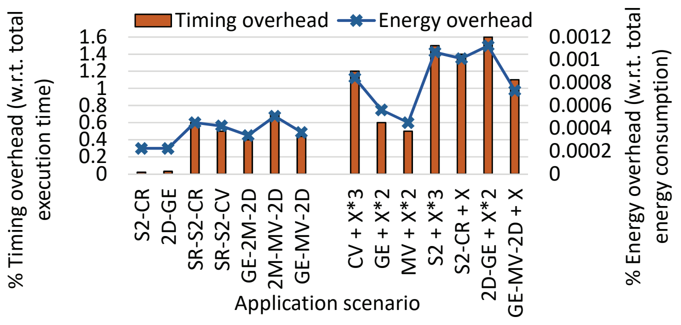

Figure 11. Adaptation overhead for different run-time scenarios.

various scenarios are 160 seconds and 178J, respectively. The absolute average timing and energy overhead of adaptations are 1.2 seconds and 0.001J, respectively, where each adaptation is performed in the order of milliseconds. The overhead depends mainly on the number of adaptations (nAd) performed over the whole execution and time and energy taken by each adaptation. Further, for each adaptation, *sched_setaffinity* interface of Linux changes the threads to cores affinity based on the identified mapping. The overhead associated with changing the core affinity varies with number of used cores and their operating frequencies, but it is quite small [45]. Considering these factors, the worst-case complexity for adaptation is $O(nAd.N_b.N_L.F_b.F_L.F_g)$. This indicates that adaptation complexity increases when the platform has more core types and frequency points (levels). However, since the number of adaptations is minimized while having reduced adaptation overheads facilitated by offline profiling information, the approach is scalable to higher size platforms containing more core types and frequency levels. For the considered scenarios and platform in Fig. 11, the average timing and energy overhead of adaptations is 0.75% and 0.00059% w.r.t. total execution time and energy consumption, respectively. These overheads are very minimal and thus proposed approach can be efficiently applied to achieve energy savings.

6 Conclusions

We proposed a run-time management approach that performs energy-efficient mapping and thread-partitioning of initial and newly added performance constrained applications and adaptation when any application(s) completes execution and/or there is no available resource at an application arrival. The mapping and partitioning is always identified through collaboration between CPU and GPU cores, and exploitation of profiling results. The adaptation process recomputes the mapping and partitioning, and is applied when beneficial in terms of energy consumption. Comparative evaluations on real hardware platform showing energy savings by the proposed adaptive approach over existing approaches indicate that it can be used to develop future energy-efficient embedded systems with CPU-GPU mobile SoCs.

Acknowledgment

This work was supported in part by the Engineering and Physical Sciences Research Council under EPSRC Grant EP/L000563/1 and EP/K034448/1 the PRiME Programme Grant (www.prime-project.org). Experimental data used in this paper can be found at DOI: 10.5258/SOTON/D1077

References

[1] (2016) Exynos 5 Octa (5422). www.samsung.com/exynos/.
[2] (2016) The open standard for parallel programming of heterogeneous systems. https://goo.gl/A9wXRJ.
[3] P. Greenhalgh, “Big. little processing with arm cortex-a15 & cortex-a7,” *ARM White paper*, pp. 1–8, 2011.
[4] D. Grewe and M. F. O’Boyle, “A static task partitioning approach for heterogeneous systems using OpenCL,” in *International Conference on Compiler Construction*. Springer, 2011, pp. 286–305.
[5] D. Grewe, Z. Wang, and M. F. O’Boyle, “OpenCL task partitioning in the presence of GPU contention,” in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 87–101.
[6] C.-K. Luk, S. Hong, and H. Kim, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 45–55.
[7] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, “Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms,” in *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 208–215.
[8] Y. Wen, Z. Wang, and M. F. O’Boyle, “Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms,” in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.
[9] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling heterogeneous multi-cores through performance impact estimation (pie),” in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 213–224.
[10] J. Ma, G. Yan, Y. Han, and X. Li, “An analytical framework for estimating scale-out and scale-up power efficiency of heterogeneous manycores,” *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 367–381, 2016.
[11] A. Aalsaud, R. Shafik, A. Rafiev, F. Xia, S. Yang, and A. Yakovlev, “Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems,” in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2016, pp. 368–373.
[12] E. Del Sozzo, G. Durelli, E. Trainiti, A. Miele, M. Santambrogio, and C. Bolchini, “Workload-aware power optimization strategy for asymmetric multiprocessors,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 531–534.
[13] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, “Sparta: runtime task allocation for energy efficient heterogeneous many-cores,” in *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2016, p. 27.
[14] B. K. Reddy, A. Singh, D. Biswas, G. Merrett, and B. Al-Hashimi, “Inter-cluster thread-to-core mapping and dvfs on heterogeneous multi-cores,” *IEEE Transactions on Multiscale Computing Systems*, pp. 1–14, 2017.
[15] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, “Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 147, 2017.
[16] “Odroid-XU3,” http://www.hardkernel.com/, 2016.
[17] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–10.
[18] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, “Workload and power budget partitioning for single-chip heterogeneous processors,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 401–410.
[19] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili, “Coordinated energy management in heterogeneous processors,” *Scientific Programming*, vol. 22, no. 2, pp. 93–108, 2014.
[20] H. Wang, R. Singh, M. J. Schulte, and N. S. Kim, “Memory scheduling towards high-throughput cooperative heterogeneous computing,” in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 331–342.
[21] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, “A black-box approach to energy-aware scheduling on integrated cpu-gpu systems,” in *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 70–81.

[22] S. Rai and M. Chaudhuri, "Using criticality of gpu accesses in memory management for cpu-gpu heterogeneous multi-core processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 133, 2017.

[23] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 273.

[24] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez, "Energy efficient hpc on embedded socs: Optimization techniques for mali gpu," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 123–132.

[25] K. Chandramohan and M. F. O'Boyle, "Partitioning data-parallel programs for heterogeneous mpsocs: time and energy design space exploration," in *ACM SIGPLAN Notices*, vol. 49, no. 5. ACM, 2014, pp. 73–82.

[26] G. Jo, W. J. Jeon, W. Jung, G. Taft, and J. Lee, "Opencl framework for arm processors with neon support," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM, 2014, pp. 33–40.

[27] (2017) FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting cpus. [Online]. Available: https://github.com/zuzuf/freeocl

[28] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for opencl programs on embedded heterogeneous systems," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, 2017, pp. 11–20.

[29] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "Adaptive resource management for simultaneous multitasking in mixed-grained reconfigurable multi-core processors," in *Proceedings of the IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2011, pp. 365–374.

[30] W. Quan and A. D. Pimentel, "A hierarchical run-time adaptive resource allocation framework for large-scale mpsoc systems," *Design Automation for Embedded Systems*, vol. 20, no. 4, pp. 311–339, 2016.

[31] H. Kim and H. Yang, "An online self-adaptive system management technique for multi-core systems," *Procedia Computer Science*, vol. 83, pp. 417–424, 2016.

[32] K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett, "Adamd: Adaptive mapping and dvfs for energy-efficient heterogeneous multi-cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[33] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated cpu-gpu power management for 3d mobile games," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[34] W.-M. Chen, S.-W. Cheng, P.-C. Hsiu, and T.-W. Kuo, "A user-centric cpu-gpu governing framework for 3d games on mobile devices," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. IEEE, 2015, pp. 224–231.

[35] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz, "A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile platforms," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 2015, p. 62.

[36] U. Gupta, R. Ayoub, M. Kishinevsky, D. Kadjo, N. Soundararajan, U. Tursun, and U. Ogras, "Dynamic power budgeting for mobile systems running graphics workloads," *IEEE Trans. on Multi-Scale Comp. Sys*, 2017.

[37] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Synergistic cpu-gpu frequency capping for energy-efficient mobile games," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, p. 45, 2018.

[38] A. K. Singh, C. Leech, B. K. Reddy, B. M. Al-Hashimi, and G. V. Merrett, "Learning-based run-time power and energy management of multi/many-core systems: current and future trends," *Journal of Low Power Electronics*, vol. 13, no. 3, pp. 310–325, 2017.

[39] "ARM big.LITTLE Technology," http://www.arm.com/, 2014.

[40] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao, "Virtcl: a framework for opencl device abstraction and management," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 161–172.

[41] A. Karami, F. Khunjush, and S. A. Mirsoleimani, "A statistical performance analyzer framework for opencl kernels on nvidia gpus," *The Journal of Supercomputing*, vol. 71, no. 8, pp. 2900–2921, 2015.

[42] C. Im, S. Ha, and H. Kim, "Dynamic voltage scheduling with buffers in low-power multimedia applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 686–705, 2004.

[43] "FFmpeg," https://github.com/FFmpeg/FFmpeg, 2018.

[44] "VLC media player," https://github.com/videolan/vlc, 2018.

[45] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 2013, pp. 1–10.

Amit Kumar Singh (M'09) is a Lecturer at University of Essex, UK. He received the B.Tech. degree in Electronics Engineering from Indian Institute of Technology (Indian School of Mines), Dhanbad, India, in 2006, and the Ph.D. degree from the School of Computer Engineering, Nanyang Technological University (NTU), Singapore, in 2013. He was with HCL Technologies, India for year and half until 2008. He has a post-doctoral research experience for over five years at several reputed universities. His current research interests are system level design-time and runtime optimizations of 2D and 3D multi-core systems for performance, energy, temperature, reliability and security. He has published over 80 papers in reputed journals/conferences, and received several best paper awards.

Basireddy Karunakar Reddy received his M.Tech. degree in Microelectronics and VLSI from Indian Institute of Technology (IIT), Hyderabad, India in 2015. He is a Ph.D. student in Electronic and Electrical Engineering at the University of Southampton, UK. His current research interests include design-time and run-time optimization of performance and energy in multi-core heterogeneous systems.

Alok Prakash received his Ph.D. degree in 2014 from Nanyang Technological University (NTU), Singapore. He is currently a Senior Research Fellow in the School of Computer Science and Engineering, NTU, where he leads a team of researchers and Ph.D. students in developing low cost camera-based traffic law enforcement sensors under the TUMCreate project. His research interests include Intelligent Transportation Systems, especially focused on solving the first mile last mile issues as well as low-cost and low-power embedded systems design with particular emphasis in mapping complex computer vision applications on modern heterogeneous mobile SoCs. His papers have been nominated for best paper award in DAC 2016 and HEART 2018.

Geoff Merrett (GSM'06-M'09) is an Associate Professor in the School of Electronics and Computer Science at the University of Southampton, UK, and Head of its Centre for IoT and Pervasive systems. He received the B.Eng. and Ph.D. degrees from Southampton in 2004 and 2009, respectively. His research interests are in energy management of mobile and embedded systems, and has published over 175 articles in journals/conferences in these areas.

Bashir M. Al-Hashimi (M'99-SM'01-F'09) is an ARM Professor of Computer Engineering, Dean of the Faculty of Physical Sciences and Engineering, and the Co-Director of the ARM-ECS Research Centre, University of Southampton, Southampton, U.K. He has published over 380 technical papers. His current research interests include methods, algorithms, and design automation tools for low-power design and test of embedded computing systems. He has authored or co-authored five books and has graduated 35 Ph.D. students.