OPTiC: Optimizing Collaborative CPU-GPU Computing on Mobile Devices with Thermal Constraints

Siqi Wang, Gayathri Ananthanarayanan, and Tulika Mitra, *National University of Singapore*

***Abstract*—The CPU-GPU co-execution of computation kernels on heterogeneous MPSoC can significantly boost performance compared to the execution on either the CPU or the GPU alone. However, engaging multiple on-chip compute elements concurrently at the highest frequency may not provide the optimal performance in a mobile system with stringent thermal constraints. The system may repeatedly exceed the temperature threshold necessitating frequency throttling and hence performance degradation. We present OPTiC, an analytical framework that given a computation kernel can automatically select the partitioning point and the operating frequencies for optimal CPU-GPU co-execution under thermal constraints. OPTiC estimates, through modeling, CPU and GPU power, performance at different frequency points as well as the performance impact of thermal throttling and memory contention. Experimental evaluation on a commercial mobile platform shows that OPTiC achieves an average 13.68% performance improvement over existing schemes that enable co-execution without thermal considerations.**

***Index Terms*—Co-execution, mobile platforms, thermal, GPU**

I. Introduction

MODERN heterogeneous multi-processor system-on-chip (MPSoC) architectures are equipped with multiple compute elements including multi-core CPUs, accelerators capable of general-purpose computing including graphic processing units (GPUs), digital signal processors (DSPs) and Field Programmable gate arrays (FPGAs). The execution of an application is therefore no longer constrained to only the CPU cores. The powerful CPU cores achieve good performance at the cost of high power consumption. With the inclusion of accelerators, some applications can reach the same or even better performance with lower power. In particular, for applications with high data parallelism, porting the computation onto GPUs largely improves performance and power efficiency. To facilitate this trend of collaboration among multiple compute elements, Khronos Group defined OpenCL [1], a cross-platform programming model that enables the exploitation of data parallelism. Multiple works [2], [3], [4] in addition show that the partitioned CPU-GPU co-execution of computation kernels can significantly boost performance compared to the execution on either CPU or GPU alone.

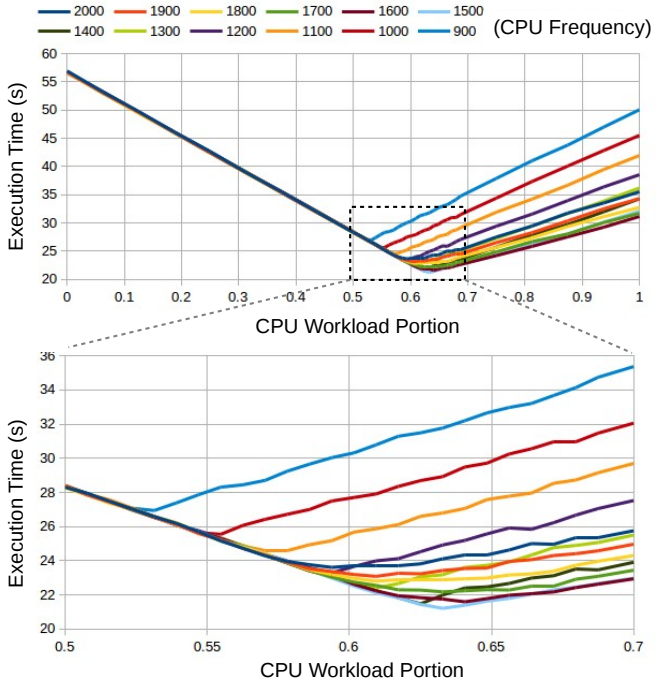

Fig. 1: Execution time of *SYRK* kernel under varying CPU-GPU workload partitioning and frequency.

On a mobile platform, the thermal constraints are more stringent in the absence of active cooling measures. As we engage multiple compute elements simultaneously, the total power increases and the chip temperature may repeatedly exceed the temperature threshold. This necessitates frequency throttling by the underlying operating system and leads to performance degradation.

Figure 1 illustrates the effect of thermal throttling on CPU-GPU co-execution on a mobile platform with *SYRK* kernel from the Polybench benchmark suite [5]. The experimental setup is detailed in Section V. We perform a design space exploration that sweeps through all the frequency levels of the CPU and the GPU with all possible workload partitions. Each line in the graph represents the execution time corresponding to a fixed CPU frequency, ranging from 900MHz to 2000MHz. In this example, the GPU never gets throttled even when running at the highest frequency and hence the GPU frequency is not plotted. The workload portion is the fraction allocated to the CPU, where 0 (or 1) corresponds to the execution

Manuscript received July 24, 2018; accepted September 3, 2018.
The first author and second author contributed equally to this work.
This work was partially funded by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2015-T2-2-088. This paper was recommended by Associate Professor Aviral Shrivastava and Associate Editor Sudeep Pasricha. *(Corresponding author: Tulika Mitra.)*
Siqi Wang, Gayathri Ananthanarayanan and Tulika Mitra are with Department of Computer Science, School of computing, the National University of Singapore, Singapore 117417.
Digital Object Identifier 10.1109/TCAD.xxxx.xxxx

of the kernel on the GPU (or CPU) alone. As reported in previous works [2], [3], [4], CPU-GPU co-execution provides lower runtime than either the CPU or the GPU execution alone. For SYRK, the best possible CPU alone, GPU alone execution time are 31.62s and 58.61s, respectively, while the best possible co-execution runtime is 21.43s. The optimal configuration for this runtime is 1500MHz CPU frequency with 64% of the workload allocated to CPU. Identifying the optimal configuration is challenging because it depends on (a) relative execution time of the kernel on CPU and GPU separately, (b) thermal throttling at higher frequency, and (c) contention for off-chip shared memory between CPU and GPU. But OPTiC successfully solves this problem through analytical modeling of all these factor individually and collectively.

We can make several important observations from Figure 1. (1) Thermal throttling impacts the co-execution performance significantly. Higher frequency does not necessarily provide lower runtime. A lower frequency (1500MHz for *SYRK*) may induce less thermal throttling and lead to lower runtime (21.43s) for co-execution compared to 24.61s at maximum frequency of 2000MHz. (2) Even if two frequency settings have the same single-device runtime, they may not behave similarly during co-execution due to different thermal behavior. For example, the CPU-alone execution at 1900MHz and 1400MHz for *SYRK* have the same runtime (34.3s); but the optimal workload partition and runtime for co-execution is different, as shown in the zoomed-in plot.

(3) The optimal workload partition is different for each CPU operating frequency depending on the individual runtime of CPU and GPU, thermal throttling behaviour, and memory contention. The optimal CPU workload partition for *SYRK* varies from 0.53 to 0.645 across different frequencies. (4) When CPU workload portion is less than 50% for *SYRK*, the co-execution runtime is not impacted by CPU frequency. This is because with lower CPU workload, the GPU runtime dominates even at the highest CPU frequency. But the GPU is not thermally throttled in our platform with increased CPU frequency and temperature; hence GPU runtime and co-execution runtime remain unchanged.

The optimal workload partition and CPU-GPU frequency depend on the application characteristics. One may conduct an exhaustive design space exploration to identify the optimal configuration. With M possible CPU frequencies, N possible GPU frequencies and S partition points, the design space has $M * N * S$ points. On our platform, it takes almost 1.5 days to execute all the design points for a single kernel, especially because the chip needs to be cooled down to the same ambient temperature before each execution. In contrast, our analytical modelling takes only 5 mins to come up with the optimal configuration. Developing an analysis framework that can predict the optimal configuration involves many challenges such as modelling the throttling behaviour of the kernel based on its power-performance characteristics, the underlying system behaviour, and the shared resource contention due to concurrent execution. In addition, developing our analytical model on a real hardware platform has its own challenges. We have limited information regarding the details of the hardware architecture and the supported hardware performance counters due to insufficient documentation. We adopt targeted micro-benchmarking to reverse engineer micro-architectural details, memory controller policies, precise definition of the performance counters, and the latency of the operations. Also we employ analytical modelling in conjunction with statistical regression using the available hardware performance counters to estimate power-performance-thermal behaviour.

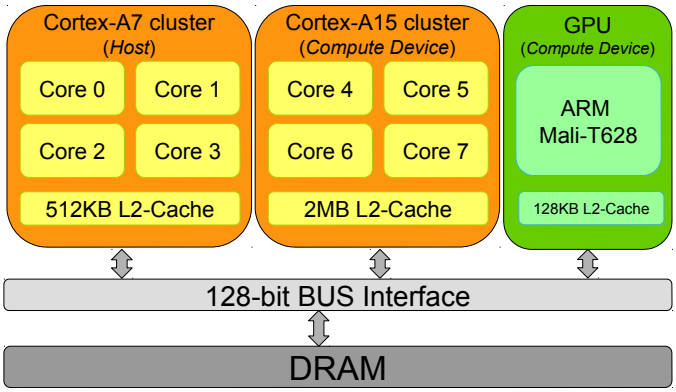

Fig. 2: Block Diagram of Exynos 5422 SoC with CPU and GPU on Odroid XU3.

We present OPTiC, an analytical framework that automatically selects the workload partition and the operating frequencies for optimal CPU-GPU co-execution. OPTiC estimates, through modeling, the CPU and GPU performance (Section IV-D1), power (Section IV-D2) at different frequencies, the impact of thermal throttling on performance (Section IV-B, IV-A2), as well as the performance penalty due to CPU-GPU memory contention (Section IV-C, IV-A3). OPTiC achieves substantial performance improvement over existing schemes that attempt co-execution without thermal considerations. Furthermore, OPTiC predicts the exact optimal configuration choice for most benchmarks kernels. The runtime of the OPTiC selected configuration is, on an average, only 5.1% higher compared to the optimal runtime obtained through exhaustive design space exploration.

II. Background

To explore CPU-GPU co-execution behaviour, we use a mobile platform with heterogeneous MPSoC architecture, Odroid XU3 [6]. We detail the platform characteristics, the programming model and the execution behaviours to facilitate understanding of OPTiC.

Experiment Platform: Odroid XU3 platform is powered by Exynos 5422 SoC [7] shown in Figure 2. The SoC features ARM big.LITTLE architecture consisting of heterogeneous processing with a cluster of four ARM Cortex A15 cores designed for performance, and a cluster of four ARM Cortex A7 cores designed for power efficiency. The MPSoC also includes ARM Mali T628 MP6 GPU implementing "Midgard" architecture with six shader cores. The platform supports OpenCL 1.1 facilitating general purpose computation on the GPU. The integrated CPU-GPU platform with shared DRAM through the bus interface avoids the overheads of transferring data between the CPU and the GPU during collaborative execution.

OpenCL Programming Model: The Open Computing Language is an open standard for developing parallel applications in heterogeneous multi-core architectures including CPU, GPU, DSP, and FPGAs [1]. It allows runtime software from different vendors to co-exist and the same program can exploit multiple different devices. An OpenCL *host* (CPU) controls the computation by launching a *kernel* on the *compute devices* (CPU, GPU, etc.). The compute device consists of one or more compute units (e.g., a GPU Mali core). The host is responsible for setting up the execution including run-time compiling of OpenCL code, mapping/transferring data to the devices, initiating the execution and finally reading the data from the devices for further execution. Each kernel instance, called a *work-item*, runs the kernel code on a single data point. A user-defined number of work-items form a *work-group*. All the work-items in a work-group share a local memory and execute concurrently on a single compute unit. Memory consistency is required for work-items within a work-group but not across different work-groups, which makes a work-group the unit for scheduling, and enables different work-groups to be launched on different compute devices without concern about memory consistency.

OpenCL Runtime: The OpenCL runtime software for the Mali GPU is supplied by the vendor. But the current mobile SoCs typically do not include OpenCL support for the ARM CPU cores. We install an open-source OpenCL runtime, FreeOCL [8] to utilize the CPU cores as OpenCL compute devices. As different OpenCL runtime softwares are used for the CPU and the GPU, the execution cannot be automatically partitioned across the devices at runtime. Instead, co-execution is achieved by launching the kernel on both devices concurrently with pre-defined workload partitions, defined in the granularity of a work-group. The number of work-groups to be executed are decided statically through OPTiC.

For an OpenCL application executing on the CPU, the role of the CPU cores toggles between host and compute devices continuously. The CPU first acts as a host to launch the kernel on the devices. The role of the CPU then transforms into an OpenCL device. The host program waits for the CPU to be discharged from the kernel execution. The host cannot launch another kernel on the GPU at this point even if the GPU has completed its earlier workload. The GPU has to wait till the CPU cores have completed the current kernel execution. This effect introduces additional overhead to GPU execution and is a limitation of the OpenCL runtime that we are using. To evaluate the CPU cores only as an OpenCL device, we modified the FreeOCL runtime to pin the OpenCL host functions to the A7-cluster and the device functions to the A15-cluster. The A7-cluster is also reserved for other system functions to reduce the interference with OpenCL device functions. Also, OpenCL runtime considers the big.LITTLE clusters (A15 and A7 clusters) as a single homogeneous device and splits the workload equally to run on all A15 and A7 cores. Therefore, limiting the OpenCL workloads to only high-performance A15 cores eliminates the sub-optimal performance introduced by OpenCL runtime and ensures optimal prediction. In the paper, we will use CPU and A15 interchangeably to represent the OpenCL execution on A15 cores as OpenCL devices.

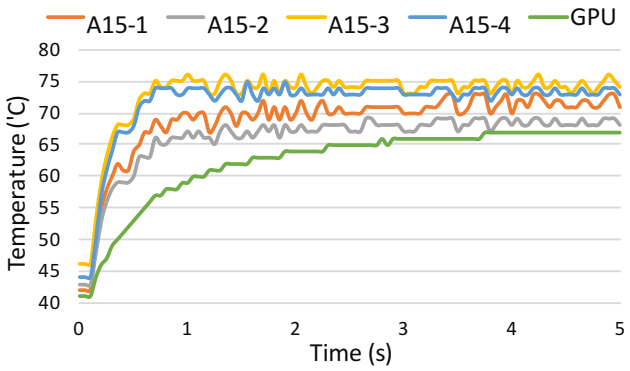

Fig. 3: Thermal Behaviour of A15, GPU in co-execution.

Dynamic Voltage Frequency Scaling (DVFS): DVFS is the most commonly used technique for the trade-off between power/thermal and performance. Odroid XU3 provides DVFS feature per cluster for CPU and GPU cores. Note that all the cores within a cluster should run at the same frequency level. The A15 cluster can be clocked between 200MHz to 2000MHz at an interval of 100MHz and the GPU can be clocked at seven different voltage-frequency settings between 177MHz and 600MHz. The voltage at each frequency level is automatically set by the hardware, as shown in Table I and II.

TABLE I: A15 CPU Frequency and Voltage Settings.

f (MHz)	900	1000	1100	1200	1300	1400
V (mV)	932.5	958.8	985	1012.5	1038.8	1052.5
f (MHz)	1500	1600	1700	1800	1900	2000
V (mV)	1080	1108.7	1147.5	1186.2	1237.5	1315

TABLE II: GPU Frequency and Voltage Settings.

f (MHz)	600	543	480	420	350	266	177
V (mV)	975	962.5	912.5	875	850	975	762.5

Thermal Behaviour: The Odroid XU3 platform has five temperature sensors that capture the temperature of the four cores in the A15 cluster and the GPU. During execution, the temperature change is significant. Figure 3 shows the thermal behaviour of the A15 cluster and the GPU when co-executing *SYRK* at the highest frequency settings for the initial 5 seconds. We start sampling the thermal sensors before the kernel execution to capture the initial temperature. The difference in initial temperatures is because of the core placement. The thermal response of each A15 core to the input power is different. The temperature of the A15 cluster rises when the kernel execution starts and the hottest core reaches 75°C (thermal threshold) very quickly (within 1s). This triggers the underlying OS thermal management. The frequency of the whole A15 cluster is throttled down to 900 MHz, causing temperature drop. The frequency changes back to 2000MHz once the temperature is lower than the thermal threshold, causing the temperature to rise again. The thermal throttling is triggered very frequently, resulting in a fluctuating A15 cluster temperature around the thermal threshold (74 - 76°C). It greatly reduces the processing speed, resulting in as high as 41% increase in CPU runtime. The temperature of the GPU is much cooler and stabilizes at around 67°C in this example without any thermal throttling.

III. Related Works

In recent years, researchers have focused towards developing scheduling techniques to appropriately map the applications onto heterogeneous MPSoCs comprising of functionally different computing cores (CPU and GPU) [9], [10]. Most of the techniques [11], [12], [13], [14] target towards maximizing performance and energy efficiency. These techniques map the entire application kernel on to a single device (CPU or GPU). Several other works [15], [2], [4], [16], [3] partition the OpenCL workloads to be executed on CPU and GPU concurrently.

Authors in [2] show that by exploiting both functional heterogeneity and performance heterogeneity concurrently, significant performance gains can be obtained. Additionally, [2] presents a static workload partitioning technique and provides the voltage frequency settings to achieve power efficiency. Recently, [4] proposes a mapping and thread partitioning scheme for executing multiple OpenCL applications concurrently. They execute each application across all the design points and collect the execution time on the individual devices (CPU/GPU) to compute the optimal workload partition at each design point. They also model the delay due to existing enqueued threads from another application in the GPU and repartition the workload for minimum energy consumption at runtime. The work in [3] also proposes scheduling approach for multiple kernels running concurrently on the GPU. OPTiC, in contrast, focuses on the co-execution of a single kernel and does not require extensive offline profiling. Work [15] automatically partitions threads to CPUs and GPUs by providing new APIs. The APIs require access locations of all threads to be analyzed statically and thus needs extensive profiling.

However, none of the works mentioned earlier consider the effect of thermal constraints on the mobile platforms. Dynamic voltage and frequency scaling (DVFS) is a common technique to address the power and thermal issues on the mobile platforms. Existing approaches to DVFS for power/thermal management include running at the lowest or highest frequency, as well as reacting to runtime workload to reduce or increase frequency based on the CPU usage. Many works[17], [18], [19] model the thermal behaviour of the device and dynamically adjust the frequencies of the processors to mitigate the effect of thermal violation on performance. Qscale[20] demonstrates a thermally efficient thread scheduling and DVFS policy for heterogeneous mobile platforms. The authors limit the use of the big CPU cores to execute only the QoS critical threads, which effectively slows down the heating of the big cores. The thread to core mapping of the QoS critical threads is based on the thermal signature of each of the big cores and the thermal coupling between the CPU and GPU. These works do not involve workload partitioning and consider different applications executing concurrently on CPU and GPU.

Another work [21] extends [2] and [4] to map and partition the application in CPU-GPU MPSoCs taking into account the temperature behaviour. It performs exhaustive profiling similar to [4] that measures the temperature at different design points to compute the new partition, in order to achieve minimum energy consumption and best temperature.

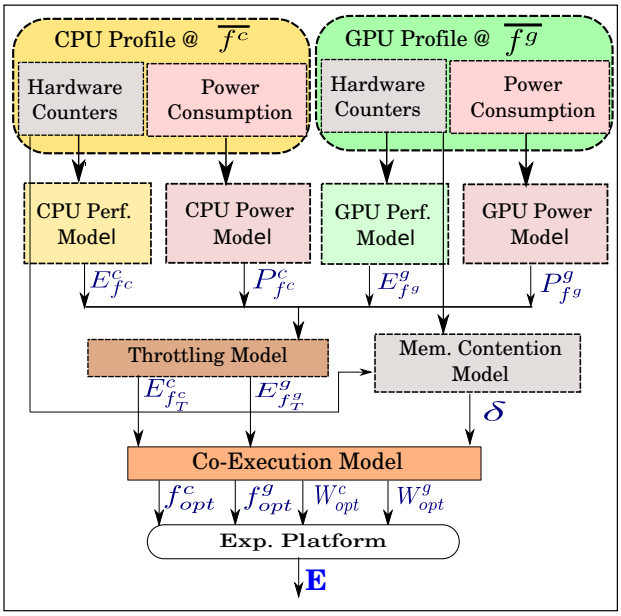

Fig. 4: OPTiC Framework.

On the other hand, authors in [22] demonstrate a compile-time CPU frequency selection approach for affine programs. The approach categorizes a certain program region by approximating the operational intensity and parallelism features through static analysis. The frequency and core settings of the program category are chosen from profiling the processor with representative micro-benchmarks. The authors further implement a lightweight runtime DVFS approach based on energy efficiency, and show that the compile-time approach to CPU frequency selection can significantly outperform runtime-based approaches. The experimental evaluation of our work confirms these findings.

OPTiC, with comprehensive analytical model for CPU, GPU performance, power and temperature, is able to predict the impact of thermal behaviour on the execution time without extensive profiling, and recommends the best execution configuration before hand.

IV. OPTiC Framework

We present OPTiC: a framework for **OP**timizing collaborative CPU-GPU computing with **T**hermal **C**onstraints, as shown in Figure 4. The variables in blue denote the output of the respective models. The high-level operation of OPTiC is as follows: For each kernel, two profile runs are performed to obtain the CPU and GPU parameters from CPU-alone and GPU-alone executions, respectively. The profile run for CPU-alone execution is performed at frequency $\overline{f^c}$ while GPU-alone execution is performed at $\overline{f^g}$. OPTiC uses the hardware counters and power values from the profile runs, pass them through a performance model and power model to obtain the respective predictions for all possible target frequency configurations(f^c and f^g). The predicted performance and power information are then passed to a thermal throttling model to capture the effect of thermal constraints. OPTiC extracts memory subsystem related hardware counters from the profile runs and uses them in its memory contention model to

model the interferences between devices. These models finally add up to a co-execution model that calculates the performance of the CPU-GPU co-execution for all the frequency pair combinations. The configuration with the minimum predicted runtime is the optimal configuration identified by OPTiC.

OPTiC does not attempt to predict a throttle-free configuration; rather the model is able to analyse the thermal behaviour and predict the impact of thermal throttling on the runtime and thus provide optimal performance achievable under the current thermal condition. We will next explain how we construct the co-execution model (Section IV-A), followed by the details of the throttling model (Section IV-B), memory contention model (Section IV-C), and finally the independent CPU and GPU performance models (Section IV-D).

A. CPU-GPU Co-Execution Model

In OpenCL programming model as discussed in Section II, the unit for scheduling is a work-group. All the work-items in the same work-group execute concurrently. The work-groups are independent of each other. Thus we set the granularity of the workload partition to be a work-group. The execution time is observed to be linearly proportional to the workload partition (number of work-groups) allocated when executing CPU or GPU alone. When executing CPU and GPU concurrently, as both devices are working in their address spaces, we can safely assume the same linear relation with the workload partition. A non-linear model can be applied if the linear relationship does not hold for other platforms. The co-execution time is the maximum of the time taken by the individual devices with the allocated workload partition. However, the effect of thermal throttling may drastically increase the execution time of the devices. In addition, the execution of both devices concurrently will cause contention among shared resources, such as the main memory. The effects of thermal throttling and memory contention are included to complete the co-execution model.

1) Co-Execution of CPU and GPU: Figure 5 shows a diagram representing the execution time of CPU and GPU for increasing number of work-groups. The primary and secondary Y-axes are the percentage of work-groups allocated to the CPU (increasing) and GPU (decreasing), respectively. At a given frequency combination (f^c, f^g) for CPU and GPU, the CPU takes $E^c_{f^c}$ time to complete all the work-groups, while the GPU takes $E^g_{f^g}$ time. The rate of work-group processing is denoted as r_c and r_g for the CPU and the GPU respectively, calculated as total workload (which is considered as 1) over runtime: $r_c = 1/E^c_{f^c}$ and $r_g = 1/E^g_{f^g}$ The point where the two execution time meet (E) is therefore the shortest execution time possible to complete all the work-groups using co-execution ($r_c \times E + r_g \times E = 1$). The co-execution time is calculated as shown in Eqn. (1). The workload partition is calculated using Eqn. (2).

$$E = \frac{1}{(r_c + r_g)} = \frac{1}{(1/E^c_{f^c} + 1/E^g_{f^g})} \tag{1}$$

$$\begin{aligned} &\text{CPU work portion} = E/E^c_{f^c} \\ &\text{GPU work portion} = 1 - E/E^c_{f^c} \end{aligned} \tag{2}$$

2) Effect of Thermal Throttling: Thermal throttling greatly affects the runtime by drastically reducing the processing speed. It prolongs the execution, resulting in a longer execution time which we denote as the throttled execution time as $E^c_{f^c_T}$. The workload partition calculated in Section IV-A1 is therefore not optimal. Note that the throttling of frequency starts only when the processor temperature reaches a certain threshold. We denote the time taken to reach the threshold to be t_{TL} (time to reach thermal limit). The details of how we estimate $E^c_{f^c_T}$ and t_{TL} are discussed in Section IV-B.

Here we consider the common case where CPU frequency is throttled and GPU is not affected for simpler illustration. Also note that in our platform, the GPU temperature is always lower than the thermal threshold and its frequency does not get throttled. The rate of execution will be reduced after time t_{TL}, as shown in Figure 5 (b). We approximate the behaviour by estimating an average execution rate r'_c that is lower than the original r_c. r'_c is calculated with throttled time as shown in Eqn. (3). The numerator is the remaining workload when thermal throttling starts while the denominator is the time for processing the remaining workload.

$$r'_c = \frac{1 - r_c \times t_{TL}}{E_{f^c_T} - t_{TL}} \tag{3}$$

As shown in Figure 5(b), the yellow and green lines represent scenarios where the system starts the frequency throttling at different time t_{TL}. In the first scenario (yellow dotted line, CPU_throttle_1), the temperature of CPU rises quickly resulting in frequency throttling at $t_{TL} = 2$. The final intersection of the CPU and GPU execution line is therefore different from the non-throttling scenario. We again equate the workload processed by CPU and GPU in time E to 1 as $r_c t_{TL} + r'_c(E - t_{TL}) + r_g \times E = 1$. The execution time can then be calculated by Eqn. (4).

$$E = (1 - r_c \times t_{TL} + r'_c \times t_{TL})/(r_g + r'_c), \tag{4}$$

The equation is valid only when kernel execution time is larger than t_{TL}, i.e., thermal throttling happens before the kernel completes execution. As the CPU runtime is prolonged by throttling, the workload portion allocated to CPU is less and the final execution time is longer.

In the second scenario (green solid line, CPU_throttle_2), the throttling happens later at $t_{TL} = 6$. When executing CPU and GPU together, the execution is finished before CPU temperature exceeds the thermal threshold. So the optimal configuration remains unchanged as in the non-throttling scenario (Eqn. (1)). The workload partition calculation remains the same as shown in Eqn. (2).

3) Effect of Memory Contention: In our experimental platform, the CPU cores and GPU share the DRAM through a bus interface. When CPU and GPU are executing concurrently, both the devices compete for the shared memory subsystem. This memory contention has different impacts on devices. We illustrate the case where CPU execution performance is affected whereas GPU performance remains unchanged. As shown in Figure 5 (c), the green solid line shows the CPU performance curve after considering contention. The effect of memory contention is small in the beginning and becomes

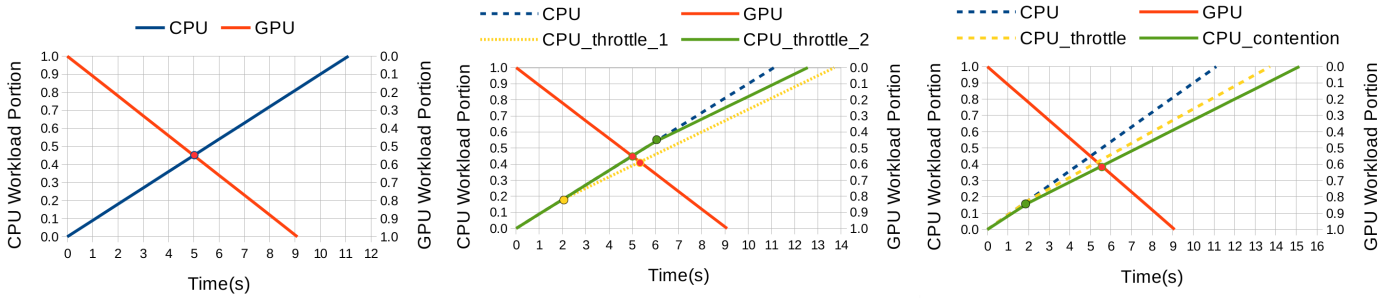

Fig. 5: (a) CPU GPU co-execution model. (b) Co-execution with CPU frequency throttling. (c) Co-execution with contention.

more prominent as time passes. Such effect is captured by discounting execution rate by δ. The execution time after including the effects of memory contention is calculated as

$$E = 1/(\frac{r_c}{1+\delta} + r_g) \tag{5}$$

The workload partition calculation remains the same as shown in Eqn. (2). The workload portion to allocate on CPU is decreased with increased overall execution time. The details of the calculation for the discount parameter δ is provided in Section IV-C.

Now when we combine the effects of both the thermal throttling and memory contention, the execution time can be calculated with Eqn. (4) with updated processing rate $r_c = \frac{r_c}{(1+\delta)}$ and $r'_c = \frac{r'_c}{(1+\delta)}$.

4) The Design Space: We have so far presented how we can predict the co-execution performance from the isolated CPU and GPU execution times, considering the effect of the thermal behaviour and the memory contention. The predicted performance $E(f^c, f^g)$ is based on the frequency and isolated execution time of the devices at a particular frequency combination(f^c and f^g). The change of frequency not only changes the isolated execution time, but also changes the thermal throttling behaviour as well as the memory contention. In order to find the optimal configuration for a kernel, we go through all the possible frequency combinations of the devices (all possible combinations of f^c and f^g) and predict the co-execution performance. The frequency configuration which gives the minimum execution time is therefore the optimal configuration located by OPTiC.

Having presented the high-level operation of the framework, we now introduce the individual components in detail.

B. Thermal Throttling Model

As active cooling is not often presented on mobile platforms, the vendors often set a safe operating thermal limit (T_{TL} = 75°C). The processor cores adjust their operating frequency based on the set thermal limit to prevent catastrophic failure of the chip. Once the CPU hits the thermal limit, the OS performs frequency throttling. In the Linux kernel 3.10.72 present on our platform, the throttling mechanism oscillates the CPU frequency between the set frequency (f^c) and 900 MHz (f^c_{throt}). When the CPU temperature rises above T_{TL}, its operating frequency is adjusted to f^c_{throt}. When the chip temperature goes below T_{TL}, the frequency is set back to f^c.

The performance impact due to throttling is quantified in two steps. We first find the time taken (t_{TL}) by the core to reach T_{TL}, and then predict the number of throttles after t_{TL} till the end of the kernel execution. We adopt the well-known RC equivalent circuit model for the CPU temperature as shown in Eqn. (6).

$$T = P^c_{f^c} \times R_c + (T_{init} - P^c_{f^c} \times R_c)e^{\frac{-t}{\tau}} \tag{6}$$

Here T is the CPU temperature, $P^c_{f^c}$ is the power consumption on CPU, R_c is the core thermal resistance, τ is the chip time constant, t is the elapsed time and T_{init} is the initial temperature. The time taken to reach the thermal limit (t_{TL}) can therefore be calculated given the initial temperature and power by Eqn. (7).

$$t_{TL} = \tau ln\left(\frac{R_c P^c_{f^c} - T_{init}}{R_c P^c_{f^c} - T_{TL}}\right) \tag{7}$$

The time periods (t_L and t_H), are the periods for which the CPU will stay at f^c_{throt} and f^c, respectively. Eqn. (7) is used to compute these time periods with different initial temperature, end temperature and power values. The temperature drops to T_{new} after the throttling before it rises again and the drop in temperature is approximated by κ times the power difference at f^c and f^c_{throt} where κ is a platform specific parameter. To calculate t_L, we plug in T_{TL} as the initial temperature and T_{new} as the end temperature with $P^c{}_{f^c_{throt}}$, and the reverse with $P^c_{f^c}$ for t_H. The power values are predicted from the models discussed in Section IV-D2.

In this model, we assume the workload to be in terms of cycles. We obtain the workload cycles from the non-throttled CPU runtime $E^c_{f^c}$ of the workload at f^c. As t_{TL} denotes the time at which throttling sets in, the remaining workload that needs to be executed after throttling $L^c_R = (E^c_{f^c} - t_{TL}) \times f^c$. As the CPU frequency oscillates between the set frequency f^c and f^c_{throt}, we can approximate the workload cycles executed in one throttle period $TC = t_L \times f^c_{throt} + t_H \times f^c$. The number of throttles $N_{TC} = L^c_R/TC$. The remainder workload in the last throttle period $L_{TC} = (N_{TC} - \lfloor N_{TC} \rfloor) \times TC$. The new CPU runtime $E^c_{f^c_T}$ incorporating the effects of throttling is calculated using Eqn. (8).

$$E^c_{f^c_T} = \begin{cases} t_{TL} + \lfloor N_{TC} \rfloor \times (t_L + t_H) + \frac{L_{TC}}{f^c_{throt}} \\ \qquad\qquad \text{, if } L_{TC} \leq t_L \times f^c_{throt} \\ t_{TL} + \lfloor N_{TC} \rfloor \times (t_L + t_H) + t_L + \frac{(L_{TC} - (t_L \times f^c_{throt}))}{f^c} \\ \qquad\qquad \text{, otherwise} \end{cases} \tag{8}$$

In our experimental platform, the GPU does not hit 75°C with general-purpose workload at runtime. However, we can follow a similar approach to model the throttling behaviour of GPU and obtain the throttled performance $E^g_{f^g_T}$.

C. CPU-GPU Memory Contention Model

As mentioned in Section IV-A3, the co-execution of CPU and GPU together causes memory contention. Through various experiments on our evaluation platform, we found that the GPU memory accesses are prioritized over CPU accesses. Thus there is a neglible impact on the performance of GPU due to contention for memory. Therefore, we model a resource contention that causes an increase in execution time only for the CPU. We define this overhead as the contention overhead ($Cont$). $Cont$ is proportional to the execution time where the CPU and the GPU are concurrently executing.

$$Cont = T_{CPU} * \delta \tag{9}$$

δ represents the increase in runtime due to co-execution over its isolated runtime. T_{CPU} is the isolated execution time of the kernel.

Parameter δ is obtained through a two-step process. For a given kernel, we use the following hardware counters `Cache:L2 data refill`, `Cache:L2 data write` and `Cache:L2 data victim` for the CPU and `Mali L2 Cache: External read bytes` and `Mali L2 Cache: External write bytes` for the GPU from the profile run at $\overline{f^c}$ and $\overline{f^g}$ and obtain the average and maximum memory request rate in (bytes/sec) of CPU and GPU at frequency $\overline{f^c}$ and $\overline{f^g}$, respectively. The memory request rate varies based on the memory intensity of the benchmark and the operating frequency. We obtain the average and maximum memory request rate at other frequencies (f^c and f^g) through a multi-variate additive regression model. The request rate at other frequencies is modeled as a function of the variables α, ξ and λ. α denotes the ratio of memory access time over computation. ξ is the ratio of target frequency over the profile frequency and $\lambda_{\overline{f}}$ is the memory request rate at profile frequency.

$$\lambda_f = func1(\alpha, \xi, \lambda_{\overline{f}}) \tag{10}$$

The second step is to model δ as a function of average and maximum memory request rate (λ^A_f, λ^M_f) of both the CPU and GPU.

$$\delta_f = func2(\lambda^{A^c}_{f^c}, \lambda^{M^c}_{f^c}, \lambda^{A^g}_{f^g}, \lambda^{M^g}_{f^g}) \tag{11}$$

We use OpenCL-based STREAM benchmarks from [23] to generate memory requests of various intensities from CPU and GPU and measure the execution times of isolated and co-execution runs. DS-5 Streamline [24] with gator is used to obtain the required hardware counters. `func1` is generated using generalized additive models of R [25] while `func2` is generated using first-order linear regression model of R including all the interaction terms.

D. Power-Performance Estimation across Frequency Settings

The optimal configuration for a given kernel is obtained by going through the design space of different frequency combinations of the devices (shown in Table I,II). The respective throttling and memory contention for the frequency pair of CPU and GPU are predicted as discussed in Section IV-B and IV-C. This information is then plugged into the co-execution model as discussed in Section IV-A to obtain the prediction of the co-execution time. Here in this section we describe how the performance and power values for other frequency configurations are obtained from a profile frequency.

1) Performance Model: With the performance counter values collected from only one execution at a certain frequency, our performance models are able to predict the execution time of the application at all frequency points. Note that the OpenCL execution incurs overheads on the host device such as API overheads and the overhead caused by JIT compilation [26]. These overheads do not need to be considered in our performance model as the host functions are pinned to the A7 cores.

*a) **CPU Performance Model**:* Performance prediction for traditional CPU processors is a well researched topic. Researchers use simulations, analytical models, hardware counter based models and machine learning models [27], [28], [29], [30], [31] to achieve good predictions. These approaches usually require detailed knowledge of the micro-architecture or hardware counters that reveals the property of the hardware and applications. The parameters to be used in the model are micro-architecture specific and need to be carefully selected. The OpenCL applications make the CPU behave similar to the GPUs [26]. Thus different frequency levels may result in different thread scheduling and thus different performance. In addition, the execution of OpenCL on CPU platforms is not a well-researched topic; existing works are mostly focusing on the API overheads rather than the actual execution time [26], [32]. Therefore in this work, we adopt a simple analytical model of GPU [33] with hardware counter information, to scale the execution performance to other frequencies for CPU. The CPU execution time (E^c) is modeled as comprising of the computational time, memory access time and latency hiding, as shown in Eqn.(12). Parameter α is the ratio of memory access time over computation time and represents the compute intensity. β represents the latency hiding fraction over the total computation time and is linearly proportional to α.

$$\begin{aligned} E^c &= E^c_{comp} + E^c_{mem} - E^c_{hide} \\ &= E^c_{comp} + \alpha \times E^c_{comp} - \beta \times E^c_{comp} \\ &= (1 + \alpha - \beta) \times E^c_{comp} \end{aligned} \tag{12}$$

From the CPU profile run at one frequency ($\overline{f^c}$), we collect the number of memory accesses through L2 cache miss counter: `Cache:L2 data refill` (N_mem) and the computational information through number of computation instructions executed: `Instruction: Executed` (I_ex) - `Instruction: Load/Store` (I_mem). Thus α is calculated as the number of memory access over computation instructions: $\alpha = N\_mem/(I\_ex - I\_mem) \times C_1$, $\beta = C_2 \times \alpha$. The constants C_1 and C_2 are application agnostic and computed by analysing micro-benchmarks with different compute intensity.

When we employ frequency scaling of A15, the CPU frequency is changed to f^c while the memory frequency does not change. Indeed, the memory controller frequency cannot be changed in our platform. Therefore, the computation time

will be scaled with frequency, keeping the memory access time the same. The execution time at f^c, denoted as $E^c_{f^c}$, is estimated by Eqn.(13).

$$\begin{aligned} E^c_{f^c} &= \frac{\overline{f^c}}{f^c} \times (E^c_{comp} - E^c_{hide}) + E^c_{mem} \\ &= \Big[\frac{\overline{f^c}}{f^c} \times (1-\beta) + \alpha\Big] \times E^c_{comp} \end{aligned} \tag{13}$$

*b) **GPU Performance Model**:* There is a rich literature on analytical performance models for conventional high-end GPUs [33], [34], [35], [36]. But ARM Mali GPU Midgard architecture differs in many-aspects from conventional desktop-class GPUs. Midgard GPUs are based on Very Long Instruction Word (VLIW) architecture and each instruction word contains multiple operations. Single Instruction Multiple Data (SIMD) is adopted so that most arithmetic instructions operate on multiple data elements simultaneously. Instead of extracting thread level parallelism, all threads in Midgard GPUs have individual program counters and the warp size is 1. Threads on a Mali GPU are independent and can diverge without any performance impact. In addition, Mali GPUs use caches instead of local memories and OpenCL local and private memories are mapped into main memory. To the best of our knowledge, there are no published works on the performance models for embedded GPUs that have architectures similar to that of ARM-Mali GPUs.

The GPU performance model is based on the execution cycles of the workload on the underlying GPU architecture, frequency and the workload characteristics. The ARM Mali-T628 GPU consists of six shader cores (SC) and an inter-core task management unit (TMU). The TMU is responsible for distributing the tasks to shader cores for execution. As depicted in Figure 6, Each SC in the Midgard architecture is made up of three different pipelines: *Arithmetic pipeline*, *Load/Store (L/S) pipeline* and *Texture pipeline*. The three pipelines are collectively known as the $tripipe$. Each thread uses only one of the Arithmetic or Load-Store execution pipes at any point in time. Two instructions from the same thread execute in sequence. The cycles for which the *tripipe* is active is determined by the slowest among the three pipelines. General purpose OpenCL applications typically only use the Arithmetic or Load-Store pipelines. The texture pipeline used for reading image data types is not often engaged. Thus, *the tripipe active cycles* is usually determined by the slowest of the Arithmetic and the L/S pipelines, as shown in Eqn. (14). OH_{TP} and OH_{TM} denote the overheads associated with driving the *tripipe* and *TMU*.

$$\textit{Tripipe Cycles} = \texttt{MAX}(L^g_{comp}, L^g_{mem}) + OH_{TP} \tag{14}$$

$$\textit{GPU Active Cycles} = \textit{Tripipe Cycles} + OH_{TM} \tag{15}$$

From the GPU profile run of the kernel at one frequency $\overline{f^g}$, we collect the following hardware performance counters : `GPU Active cycles`, `Tripipe cycles`, `LS instructions`, `LS instruction issues`, `Arith. instructions`, `L2 cache counters`. In Midgard architecture, arithmetic instructions are single cycle throughput, thus we use *Arith. instructions* counter to approximate the Arithmetic cycles (L^g_{comp}). Similar to arithmetic instructions, a single load/store can be executed in a single cycle under ideal circumstances. *LS instruction issues* counter is used to approximate the L/S cycles (L^g_{mem}). The overhead values OH_{TP} and OH_{TM} at the profiled frequency are therefore calculated using Eqn. (14) and (15). We assume the overhead values to be the same across all frequencies.

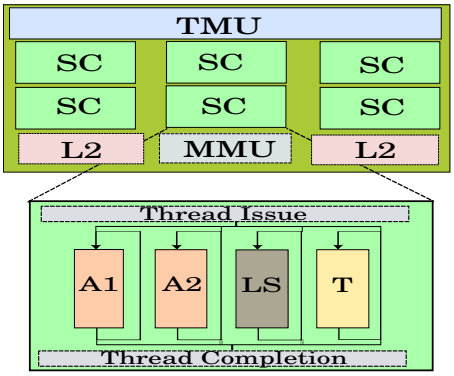

Fig. 6: Mali GPU Architecture.

In order to predict for other GPU frequencies (f^g), L^g_{mem} needs to be scaled. The L^g_{mem} cycles varies because of the changes in memory access cycles due to cache misses. In ARM Midgard architecture, cache misses does not stall the pipeline while waiting for the misses to be resolved, instead, it results in an instruction restart/reissue. *LS instruction issues* counter counts the retry cycles due to misses in the data cache. Thus we use the difference between *L/S instructions* and *LS instruction issues* counter to approximate the cycles incurred due to cache misses ($Miss^g$). The memory access latency cycles (ML_{f^g}) varies with GPU frequency(f^g). We use global latency benchmark similar to [37] to obtain ML_{f^g} of our hardware platform. For a given application kernel, the impact of variation of ML_{f^g} on $Miss^g$ is dependent on the intensity and distribution of L2 misses. We use L2 miss rate ($L2_{MR}$) to quantify the miss intensity and we assume uniform distribution of the L2 misses over the all *tripipe* cycles. $L2_{TP}$ represents L2 misses per *Tripipe* Cycle. W in Eqn.(16) is modeled as function of $L2_{MR}$ and $L2_{TP}$.

$$L^g_{mem}(f^g) = \textit{L/S instructions} + ML_{f^g} \times W \times Miss^g \tag{16}$$

With the corresponding L^g_{comp} and L^g_{mem} at f^g, we can calculate the *GPU Active Cycles* by Eqn. (14) and (15). The GPU runtime $E^g_{f^g}$ can therefore be calculated by Eqn. (17).

$$E^g_{f^g} = \textit{GPU Active Cycles}/f^g \tag{17}$$

2) Power Model: The power values of the devices are required for an accurate thermal behavior and therefore better throttling prediction as discussed in Section IV-B. Research on processor power modeling is usually based on instructions [38], hardware-counters [39], [40] and utilization [41]. Instruction-level models require massive information including the power of each instructions and overheads of all instruction pairs, making it impractical to be implemented. There have been substantial are focused around the hardware-counter-based models, where the counters are selected manually [39] or automatically by machine learning [40] for specific platforms. The identification of such counters is largely dependent on the processor architecture and benchmark characteristics.

In addition, the number of counters to be collected concurrently are usually restricted, resulting in a long profiling time. Utilization-based models, on the other hand, use one profile run to capture the activity in order to approximately predict the power. In this work, we implement an utilization-based power model to predict the power of all other frequency levels from one profile run. Few hardware counters are collected to achieve better prediction.

The total power comprises of dynamic power and static power and is defined as $P = A_c * V^2 f + P_{stat}$. Here V and f are the operating voltage and frequency respectively, A_c is the activity factor times capacitance factor. As A_c value remains constant across frequency levels, we combine these two factors into one for easier notation. The static power is application agnostic and depends on the voltage/frequency settings. Thus it can be obtained through profiling at each frequency level $(P_{stat})_{f^c}$. We note that in our mobile system, the stringent temperature constraints (75°C) ensure that additional leakage power due to high temperature is negligible.

We profile the kernel at frequency $\overline{f^c}$ and collect the power value ($P^c_{\overline{f^c}}$) from the on-chip sensors. The A_c value remains unchanged across frequency levels because in our co-execution the CPU is never idle. The power for another frequency level ${P^c}_{f^c}$ is estimated as shown in Eqn. (18).

$$ {P^c}_{f^c} = A_c * V_f^2 * f^c + (P_{stat})_{f^c} \tag{18} $$

For GPU, a similar utilization based model is implemented. GPU power ${P^g}_{f^g}$ is calculated similar to Eqn. (18).

3) Portability of OPTiC: The analytical models in OPTiC are parameterizable but need to be instantiated with platform-specific values. To port OPTiC onto a new platform, profiling and micro-benchmarking are required to identify the parameter values for thermal throttling, memory contention, and power-performance models. The models can then be applied with the new parameter values as discussed in Section IV-A to obtain co-execution performance.

V. Experimental Evaluation

We conduct the evaluation of the OPTiC framework on the Odroid XU3 platform [6]. The Polybench benchmark suite [5] is used, from which twelve benchmarks that are suitable to be partitioned to run on CPU and GPU concurrently are chosen. The benchmarks are configured with appropriate input sizes as shown in Table III, in order to minimize the interferences of OpenCL API overheads and reveal the pure kernel execution portion.

In addition, to minimize the measurement error as well we emphasize the effect of thermal throttling, multiple iterations are used for benchmarks with short execution time. We later show in Section V-G the evaluation of OPTiC with end-to-end performance results and thermal throttling behavior for real-world applications using some of these kernels.

A. Performance evaluation of Co-execution

To quantify the effectiveness of kernel co-execution, we first obtain the performance of the OpenCL kernel on a single device. We execute the OpenCL kernel on 4 A15 cores, denoted as `CPU-alone` execution. Similarly, we execute the same kernel on the 4 shader cores of Mali GPU, denoted as `GPU-alone` execution. These executions are performed for all possible A15 and GPU frequency configurations. Table IV shows the best execution time achievable for the kernel on a single device. We observe that some kernels benefit greatly from GPU executions while others do not.

TABLE III: Benchmark Configurations for Polybench.

Name	2DCONV	2MM	3MM	ATAX
Input Size	4096x4096	512x512	512x512	4096x4096
Iterations	100	30	15	75
Name	BICG	CORR	COVAR	GEMM
Input Size	4096x4096	2048x2048	2048x2048	1024x1024
Iterations	60	1	1	5
Name	GESUMMV	MVT	SYR2K	SYRK
Input Size	4096	4096	2048x2048	1024x1024
Iterations	250	50	1	30

This behavior encourages us to explore the benefits of co-execution that utilize both the CPU and the GPU concurrently. We use the configuration (frequency and workload partition) provided by our OPTiC framework for each kernel and execute the kernels on our experimental platform. The execution time for the benchmarks are listed in comparison to the minimum of the standalone execution time, as shown in Table IV. We can see that by co-executing the kernel on CPU and GPU with the configuration provided by our OPTiC framework, we are able to achieve better performance than executing the kernel on a single computing device (CPU/GPU). The performance benefit can be as high as 35% when compared to the best possible single device execution. We obtain on an average 13.8% performance improvement over standalone execution.

B. Importance of Thermal Consideration

We then evaluate the importance of considering the thermal constraints in co-execution. We compare OPTiC with [2], which enables co-execution of the kernel and partitions the workload using load balancing strategy for each kernel based on its runtime for CPU-alone and GPU-alone executions. The model in [2] does not consider the effects of thermal constraints and always runs at the highest possible CPU and GPU frequency for the co-execution.

Table IV in addition shows the actual execution times on the platform for the predicted configuration without thermal consideration [2]. OPTiC shows an average of 13.7% (and as high as 25%) improvement in performance compared to [2] by taking into account thermal behavior. Furthermore, for benchmarks $2DCONV$, $ATAX$, $BICG$, $SYR2K$, [2] performs worse than the corresponding best possible single device execution. For $2DCONV$ kernel, there is a 33% increase in the co-execution runtime compared to the single device runtime. This observation clearly emphasizes the need for considering the thermal constraints while enabling co-execution.

C. Optimal Configuration Prediction

We perform exhaustive runtime measurement of all design points by executing each kernel at all frequency settings and workload partitions. 12 CPU frequencies, 7 GPU frequencies

TABLE IV: Execution times with single-device, OPTiC predicted co-execution configuration, and predicted co-execution config from [2] with no thermal consideration.

Benchmark Names	Execution Time (Sec)					Benefits of OPTiC due to Co-execution (%)	Benefits of OPTiC due to Thermal Con. (%)
	CPU-alone Execution	GPU-alone Execution	Min of CPU- and GPU-alone	OPTiC	[2] (no thermal consideration)		
2DCONV	100.92	7.98	7.98	7.95	10.61	0.38	25.07
2MM	22.41	14.68	14.68	9.52	11.76	35.15	19.05
3MM	30.72	11.05	11.05	9.03	9.16	18.28	1.42
ATAX	41.57	12.48	12.48	12.27	15.32	1.68	19.91
BICG	32.87	9.95	9.95	9.65	11.72	3.02	17.66
CORR	53.18	129.01	53.18	41.93	50.64	21.15	17.20
COVAR	47.63	99.57	47.63	38.12	45.51	19.97	16.24
GEMM	86.19	9.97	9.97	9.13	9.75	8.43	6.36
GESUMMV	21.53	73.84	21.53	21.32	21.95	0.98	2.87
MVT	29.29	11.26	11.26	9.42	9.6	16.34	1.88
SYR2K	29.75	58.63	29.75	26.92	30.9	9.51	12.88
SYRK	31.63	56.81	31.63	21.98	28.8	30.51	23.68
					Average	13.78%	13.68%

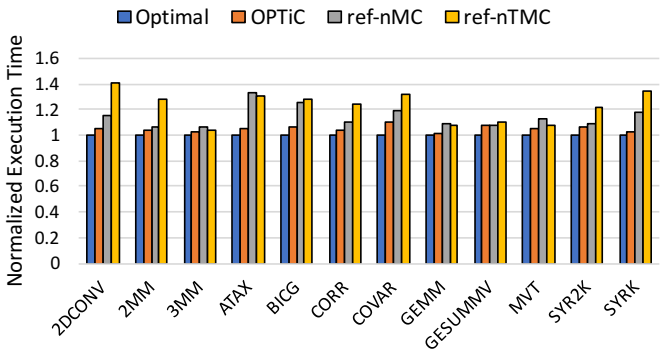

Fig. 7: Execution time of exhaustively searched optimal config with OPTiC, `ref-nMC` and `ref-nTMC` predicted configs.

and 20 different workload partition point make up the design space with 1680 points. The experiments take on average 1.5 days per benchmark, mainly because after each execution, one-minute cooling down period is applied to ensure a consistent ambient thermal condition. In contrast, with only two profile runs, OPTiC is able to perform the prediction of all design points within 5 minutes on an average for a kernel. This leads to 432X speedup in the design space exploration time with the OPTiC framework compared to an exhaustive search.

The configuration with the minimum execution time from exhaustive execution of all design points is the *actual* optimal configuration. This optimal runtime is compared against the runtime of the configuration obtained from OPTiC. We pick the configuration suggested by OPTiC and execute it on the platform to obtain the runtime. Table V shows the comparison. The Frequency shown in the table are the CPU frequencies. The GPU frequency choices are at 600MHz for all the benchmarks and therefore omitted. For benchmarks $2MM$, $3MM$, $CORR$, $GESUMMV$ and $SYRK$, OPTiC chooses accurately the best frequency setting with minimal workload-partition difference. For other benchmarks, the OPTiC predicted configurations still provide near-optimal runtime. The runtime of the predicted configuration is on average 5.1% higher than the optimal achievable runtime.

D. Influence of the Co-execution Model

We use two simplified co-execution models to evaluate the effectiveness of OPTiC's memory contention and thermal throttling models. Starting with OPTiC as reference, the model `ref-nMC` considers thermal throttling effects but removes the memory contention model as discussed in Section IV-A3. The model `ref-nTMC` removes the thermal throttling consideration as discussed in Section IV-A2 in addition to the memory contention model. We have already shown in Section V-B, the need for thermal consideration. Model `ref-nTMC` is included here for comprehensive comparison.

Figure 7 shows the runtime comparison of the predicted optimal configuration by OPTiC, `ref-nMC` and `ref-nTMC` normalized to the actual optimal execution time. The execution time of the predicted configurations is on an average 14.33% higher for `ref-nMC` and 22.74% higher for `ref-nTMC` than the optimal. When compared to OPTiC, the execution time is on an average 8.85% higher for `ref-nMC` and 16.83% higher for `ref-nTMC`.

For most of the kernels, `ref-nMC` predicts the same frequency configuration as OPTiC as shown in Table V but with more workload allocated to the CPU. As memory contention only affects the CPU runtime, the CPU time is stretched while the GPU time remains the same. Thus for benchmarks like *ATAX* and *BICG*, ignoring memory contention results in suboptimal workload allocation. OPTiC is able to capture the effect and predict the delay in CPU runtime, and therefore allocates less workload to the CPU and achieves better performance. `ref-nTMC` ignores the effect of thermal throttling and thus predicts the optimal frequency configurations to be at the highest frequency settings with corresponding workload partition. The predicted configurations are near optimal for benchmarks that do not suffer much from thermal throttling, such as *3MM*, *GEMM* and *MVT*. But for other benchmarks like *2DCONV*, such configuration can yield 40% more runtime compared to the optimal.

E. Accuracy of Individual Models

1) Performance and Power Models: Figure 8 and 9 shows the estimation error of the CPU, GPU power-performance estimations in Section IV-D1 and IV-D2 for all the benchmarks

TABLE V: Execution time for exhaustively searched optimal configuration and OPTiC predicted optimal configuration.

Benchmark Names	Optimal Configuration through Exhaustive Search			OPTiC Predicted Optimal Configuration			Error (%)
	Frequency (MHz)	CPU Portion	Execution Time (Sec)	Frequency (MHz)	CPU Portion	Execution Time (Sec)	
2DCONV	1600	6.15%	7.54	1500	6.82%	7.95	5.46
2MM	1700	39.06%	9.16	1700	37.60%	9.52	3.97
3MM	1900	26.56%	8.80	1900	24.68%	9.03	2.13
ATAX	1800	15.23%	11.69	1900	16.05%	12.27	4.94
BICG	1900	14.06%	9.10	2000	15.22%	9.65	6.02
CORR	1700	66.92%	40.58	1700	66.21%	41.93	3.32
COVAR	1700	66.92%	34.48	1800	66.61%	38.12	10.57
GEMM	1900	10.35%	9.00	1600	10.38%	9.13	1.48
GESUMMV	1700	78.10%	19.88	1700	72.97%	21.32	7.28
MVT	1700	25.10%	8.91	2000	26.72%	9.42	5.78
SYR2K	1600	66.02%	25.25	1700	62.63%	26.92	6.65
SYRK	1500	63.67%	21.43	1500	63.12%	21.98	2.57
						Average	5.06

averaged across all frequency settings. Note that the power-performance estimation models (Section IV-D1 and IV-D2) evaluated in this subsection individually do not consider the increase in execution time or change in power consumption due to thermal throttling. The effect of thermal throttling on CPU and GPU independently as well as on CPU-GPU co-execution are modeled separately (Section IV-B) starting with non-disrupted power and execution time estimations obtained from individual models and we present the evaluation of the model in the next subsection. Therefore, in this set of experiments, we apply additional active fan cooling to minimize the effects of thermal throttling when measuring the execution time/power at different frequencies and obtain accurate evaluation of the estimated execution time/power. However, the executions of some benchmarks at the highest CPU frequency setting (2.0 GHz) still suffer from thermal throttling. The non-throttled execution times at the highest CPU frequency setting are not obtainable and thus omitted for the evaluation.

The bars in Figure 8 and 9 show the average prediction error of the performance and power model across all frequency settings (with the exclusion of 2.0 GHz for some benchmarks where non-throttled performance could not be measured as discussed in the previous paragraph). The blue bars represent average prediction errors for the CPU and the green bars represent average prediction error for the GPU. The maximum and minimum prediction error are shown as the vertical line and the median error is denoted as yellow dashes on the line for each benchmark across all possible frequency settings. On an average, the individual models are able to predict within 3.55% for performance and 2.15% for power. The inaccuracy in performance and power prediction can affect the overall prediction of co-execution performance, but such effects are not directly correlated. For example, for benchmarks 2DCONV and 2MM, although the individual prediction errors of 2MM are higher than 2DCONV, OPTiC is still able to find the optimal configuration for 2MM and achieves less prediction error, as shown in Table V.

2) Thermal Throttling Model: The individual A15 performance model predicts runtime at different frequency without thermal consideration. The actual runtime at higher frequency is stretched due to thermal throttling. The first column in

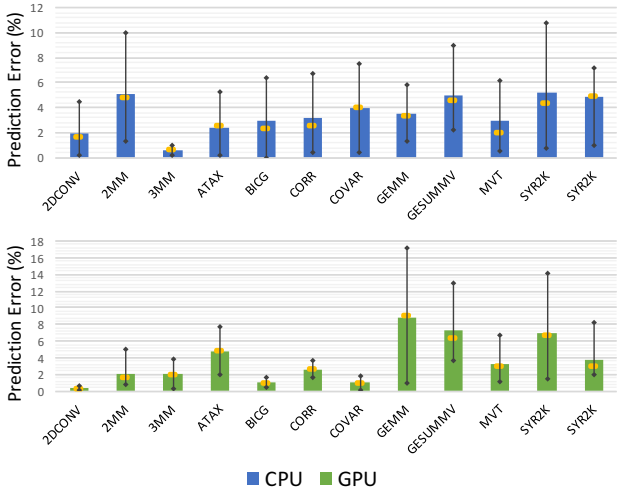

Fig. 8: Accuracy of performance predictions.

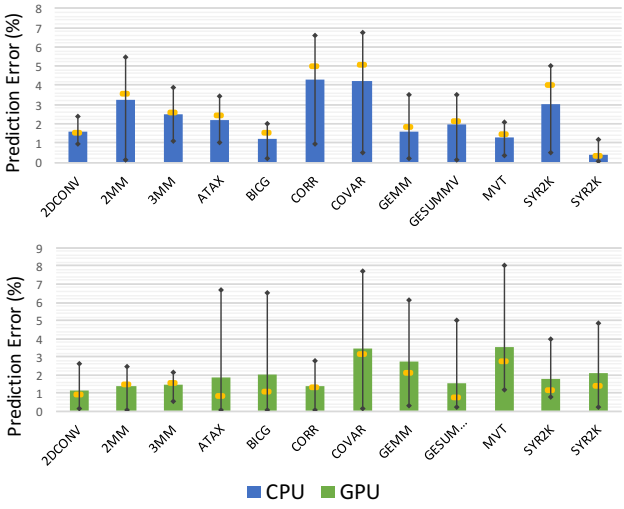

Fig. 9: Accuracy of power predictions.

Figure 10 shows the estimated runtime without thermal consideration, the second column shows the revised estimation with the thermal consideration as discussed in Section IV-B, and the third column shows the actual runtime measured on the system. Our throttling model brings down the prediction error for SYRK from 9.18% to 5.80%. At lower frequencies (below

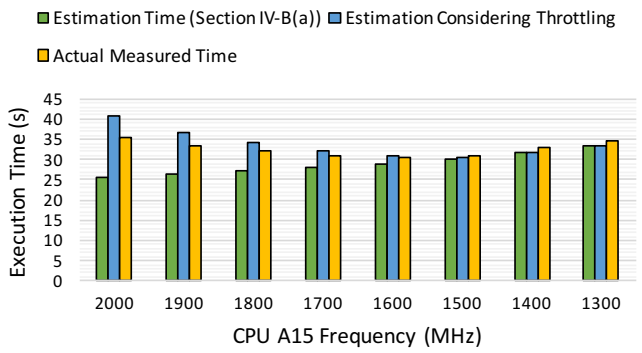

Fig. 10: CPU runtime estimation with thermal throttling consideration for SYRK.

1400 MHz), as there are few or no throttling, the estimation error is dominated by the CPU performance model.

F. Influence of DVFS

The aim of OPTiC framework is to minimize the co-execution time of the kernel and hence both the CPU and the GPU run at close to 100% utilization. This leads to thermal emergencies and we quantify its effect on performance in OPTiC. As OPTiC carefully chooses the CPU and GPU frequencies by taking care of the thermal constraints, dynamic voltage and frequency scaling (DVFS) is therefore not enabled during kernel execution except for thermal throttling and hence we use Linux *performance* governor. However, one may wonder whether other existing Linux power governors (*interactive, ondemand, conservative*) and other IPC (Instructions Per Cycle) based DVFS schemes would have been equally effective as OPTiC. Here we choose one workload based DVFS scheme (Linux *interactive* governor) and an IPC based scheme (CPU-miser) [42] for comparison purposes. CPU-miser [42] is a runtime DVFS scheme that aims to reduce power consumption given a specific constraint on the performance degradation. It specifies an index of CPU-intensiveness from the measured instructions per cycles achieved by the running workload in the current time interval, predicts the index for the next time interval and adjusts the frequency. We implemented CPU-miser for our platform. In this set of experiment, CPU-miser is invoked with 100 ms time interval and the degradation factor is set to 10%. Considering the slow varying nature for Polybench benchmarks, the index for the current interval is used as the prediction for the next time interval [43].

Table VI shows the performance improvement of OPTiC when compared to the best of CPU-alone and GPU-alone configurations as well as the co-execution configuration suggested by [2] under the two DVFS schemes. In general, OPTiC (with fixed frequency settings) is more effective than the existing power governors and the IPC-based DVFS scheme in the presence of thermal throttling during co-execution across all the benchmarks. The average performance improvement of OPTiC compared to CPU-GPU co-execution + dynamic DVFS with Linux *interactive* governor is 10.56% and CPU-GPU co-execution + dynamic DVFS CPU-miser is 5.44% respectively. Note that we use Linux kernel 3.10.106 to enable the Linux power governors, which is different from the linux kernel 3.10.72 used in the previous experiments. Hence the runtime reported are different from Table IV.

For most of the benchmarks, GPU has better performance than CPU execution. Thus dynamic DVFS on CPU does not have much impact on the minimum time of the single device execution (Min(C,G)) as the GPU execution time is not affected by the CPU frequency settings. As CPU-miser [42] introduces degradation in performance for CPU, increase in Min(C,G) are observed for $GESUMMV$, $SYR2K$ and $SYRK$, where minimum single device executions are on CPU. However, the degradation in performance is not as high as 10% as we configured, because CPU-miser helped to some extent with the thermal throttling by adjusting the CPU to a lower frequency. For a naive co-execution configuration [2], co-execution of CPU and GPU achieves better performance compared to single device execution for most of the benchmarks with dynamic DVFS. CPU-miser helps with the thermal throttling problem by reducing the CPU frequency based on the IPC achievable, gives better performance than the workload based *interactive* performance governor. However, CPU-miser is not designed to be thermal-aware, causing unpredictable performance drops for some benchmarks. On the contrary, OPTiC, which is thermal aware, is designed to predict the runtime thermal behaviours of the benchmark and further improves the co-execution performance.

G. Application Case Studies

To illustrate the effectiveness of OPTiC, we use two of the benchmarks kernels, matrix multiplication (*GEMM*) and convolution (*2DCONV*), in two full-fledged real-world applications. These two applications are Tiny YOLOv3, an object detection model in Darknet [44] neural network and a simple 128 labels, image-classification convolutional neural network (CNN) from ARM Compute Library [45].

A continuous stream of images are used in both applications. We profile the dominant kernels of the application and use the profiling results to obtain the parameters necessary for OPTiC to determine the optimal partitioning and frequency settings for these dominant kernels. The rest of the application (non-kernel portions) executes on the CPU only. There are 13 convolution layers out of the 23 layers in Tiny YOLOv3. Darknet transforms the convolution operations into matrix multiplications (*GEMM*), which comprises 76.35% of the runtime of the application. The OPTiC-predicted partitioning and frequency selection are then applied to the *GEMM* kernels for co-execution. Similarly, convolution (*2DCONV*) is the dominant computation (74.21% of runtime) in the image-classification CNN, and we apply OPTiC-based partitioning, frequency selection for co-execution of the *2DCONV* kernels.

GPU-accelerated (*GPU-acc*) execution where *2DCONV* and *GEMM* kernels are executed on GPU with the *performance* DVFS governor is used as the baseline. Furthermore, the co-execution of CPU and GPU with the partition selected by [2] which neglects the thermal behaviour are presented for comparison. The non-kernels are executed on CPU in all cases. The CPU-alone, GPU-acc and Model [2] executions are

TABLE VI: Influence of dynamic DVFS.

Benchmark Names	**Execution time of OPTiC** (s) (with Linux *performance* governor)	Execution time (s) of the execution with dynamic DVFS Linux *interactive* governor +		Performance improvement of OPTiC over the execution with dynamic DVFS Linux *interactive* governor +		Execution time (s) of the execution with dynamic DVFS CPU-miser [42] +		Performance improvement of OPTiC over the execution with dynamic DVFS CPU-miser [42] +	
		Min(C,G)	[2]	Min(C,G)	[2]	Min(C,G)	[2]	Min(C,G)	[2]
2MM	**11.09**	17.78	12.44	37.61%	10.83%	17.65	11.51	37.14%	3.60%
3MM	**10.81**	13.73	12.03	21.31%	10.19%	13.70	11.81	21.11%	8.51%
ATAX	**12.09**	12.33	12.85	1.93%	5.87%	11.85	12.97	-2.04%	6.76%
BICG	**9.40**	9.71	10.75	3.17%	12.48%	9.43	12.65	0.29%	25.63%
GEMM	**11.39**	12.69	11.35	10.22%	-0.42%	12.55	11.53	9.19%	1.21%
GESUMMV	**22.23**	20.86	24.99	-6.58%	11.05%	22.61	22.95	1.69%	3.11%
MVT	**9.58**	11.48	10.86	16.58%	11.78%	11.42	10.18	16.11%	5.90%
SYR2K	**25.54**	33.76	32.35	24.34%	21.06%	33.50	23.19	23.75%	-10.16%
SYRK	**23.90**	31.95	27.23	25.19%	12.24%	32.67	25.00	26.85%	4.41%
Average	-	-	-	**14.86%**	**10.56%**	-	-	**14.90%**	**5.44%**

TABLE VII: Image classification case study evaluation.

Configuration	Normalized Execution Time		Avg. CPU Temp		Avg. GPU Temp		% CPU Throttling	
	Performance	*Interactive*	*Performance*	*Interactive*	*Performance*	*Interactive*	*Performance*	*Interactive*
GPU-acc	1	1.02	69.16 °C	67.80 °C	62.40 °C	59.86 °C	22.29 %	18.44 %
CPU	3.9	3.97	72.65 °C	72.85 °C	60.69 °C	61.04 °C	77.51 %	79.55 %
[2]	1.05	1.06	71.68 °C	72.10 °C	63.50 °C	63.29 °C	52.39 %	56.07 %
OPTiC	0.95	-	63.56 °C	-	54.77 °C	-	0	-

TABLE VIII: Tiny YOLOv3 case study evaluation.

Configuration	Normalized Execution Time		Avg. CPU Temp		Avg. GPU Temp		% CPU Throttling	
	Performance	*Interactive*	*Performance*	*Interactive*	*Performance*	*Interactive*	*Performance*	*Interactive*
GPU-acc	1	1.01	65.09 °C	62.10 °C	59.61 °C	57.28 °C	0	0
CPU	5.3	5.53	70.97 °C	71.38 °C	61.52 °C	62.74 °C	32.8 %	42.88 %
[2]	1.10	1.12	68.33 °C	64.48 °C	60.79 °C	56.95 °C	12.32 %	7.57 %
OPTiC	0.93	-	53.15 °C	-	47.67 °C	-	0	-

configured to be at the highest CPU, GPU frequency with the *performance* and *interactive* governor, while OPTiC runs at the frequency selected by the framework without dynamic DVFS (with *performance* governor only).

Tables VII and VIII show the end-to-end latency of the entire applications including the sequential non-kernel computations under different DVFS governors, the average CPU and GPU temperature during the application execution and the percentage of time CPU is at lower frequency due to thermal throttling. The execution times reported are normalized to the baseline (*GPU-acc* with *performance* governor), i.e., lower value is better.

It can be clearly seen that OPTiC achieves performance gain: 5–7% improvement in end-to-end latency compared to GPU-only accelerated execution. Note that this performance gain of OPTiC is measured on a real platform with applications that have substantial sequential code (which becomes performance bottleneck with parallelization). Moreover, the performance gains are achieved w.r.t. a highly-optimized GPU-accelerated version. More importantly, OPTiC significantly lowers the average core temperature by 4 –9°C and in addition completely eliminates thermal throttling. Such temperature control exhibits the potential of OPTiC in extending system lifetime without compromising performance with the help of accelerators and co-execution.

A slight degradation ($< 2\%$) in performance can be observed for the execution with the *interactive* governor, comparing to the *performance* governor. This is because for CPU-only execution with high utilization rates, the *interactive* DVFS governor does not get a chance to lower the frequency and hence temperature/thermal throttling. However, for the GPU-accelerated executions, the *interactive* governor reduces the CPU frequency during low CPU utilization, resulting in an average of 2°C reduction in the CPU temperature. Note that the CPU gets throttled even in *GPU-acc* configuration (though much lower than CPU-only execution) for the image classification application as the non-convolutional layers executing on CPU are quite power-hungry

H. Limitation

As a static analysis framework, OPTiC estimates through profiling the performance of a computational kernel for CPU-GPU co-execution on platforms with stringent thermal constraints. For unknown kernels, OPTiC requires two profile runs (one execution on CPU and one execution on GPU) to identify the optimal settings. It is well suited for applications like Polybench as we demonstrated, for which the execution characteristics of the OpenCL kernels can be fully captured by the profile runs, enabling an accurate estimation of the run-time behaviour. However, for applications that present variability at runtime depending on the input, careful profiling needs to be performed to capture the dominant execution characteristics in order for OPTiC to locate the optimal execution configuration at compile time.

VI. Conclusion

Partitioned co-execution of computation kernels on CPU-GPU systems can significantly reduce the execution time. Identifying the optimal configuration that results in minimum

execution time is quite challenging due to thermal throttling and memory contention. We present OPTiC, a framework that predicts the co-execution performance through analytical modeling of CPU and GPU performance, power, impact of thermal throttling and memory contention. OPTiC achieves an average 13.68% performance gain over existing schemes attempting co-execution without thermal considerations.

REFERENCES

[1] "OpenCL: The open standard for parallel programming of heterogeneous systems." https://goo.gl/A9wXRJ.
[2] A. Prakash et al, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," in *ICCD*, 2015.
[3] Y. Wen and M. O'Boyle, "Merge or Separate?: Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms," in *GPGPU*, 2017.
[4] A. K. Singh, A. Prakash, K. R. Basireddy, and G. V. Merrett et al, "Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs," *TECS*, 2017.
[5] S. Grauer-Gray, L. Xu, R. Searles, and S. Ayalasomayajula et al, "Auto-tuning a High-level Language Targeted to GPU Codes," in *InPar*, 2012.
[6] "Odroid-XU3," http://goo.gl/Nn6z3O.
[7] "Exynos 5 Octa (5422)," www.samsung.com/exynos/.
[8] "FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting CPUs." https://goo.gl/qWL1Eg.
[9] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.*, 2015.
[10] S. Wang, A. Prakash, and T. Mitra, "Software Support for Heterogeneous Computing," in *SVLSI*, 2018.
[11] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, "An Efficient Scheduling Scheme Using Estimated Execution Time for Heterogeneous Computing Systems," *J. Supercomput.*, 2013.
[12] C. Bolchini et al, "A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures," in *ESWEEK*, 2016.
[13] K. Dev and S. Reda, "Scheduling Challenges and Opportunities in Integrated CPU+GPU Processors," in *ESTIMedia*, 2016.
[14] S. Wang, G. Zhong, and T. Mitra, "CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications," *TECS*, 2017.
[15] C. Luk and S. Hong et al, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *MICRO*, 2009.
[16] A. Vilches and R. Asenjo et al, "Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips," in *PCS*, 2015.
[17] F. Paterna and T. v. Rosing, "Modeling and Mitigation of Extra-SoC Thermal Coupling Effects and Heat Transfer Variations in Mobile Devices," in *ICCAD*, 2015.
[18] A. Prakash et al, "Improving Mobile Gaming Performance Through Cooperative CPU-GPU Thermal Management," in *DAC*, 2016.
[19] G. Bhat and G. S. et al, "Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile Platforms," *VLSI*, 2018.
[20] O. Sahin and A. Coskun, "QScale: Thermally-efficient QoS Management on Heterogeneous Mobile Platforms," in *ICCAD*, 2016.
[21] E. W. Wächter et al, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Reliable mapping and partitioning of performance-constrained openCL applications on CPU-GPU MPSoCs," in *ESTImedia*, 2017.
[22] W. Bao and C. Hong et al, "Static and Dynamic Frequency Scaling on Multicore CPUs," *ACM Trans. Archit. Code Optim.*, 2016.
[23] T. Deakin, J. Price, and M. Martineau et al, "GPU-STREAM v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *HiPC*, 2016.
[24] "ARM DS-5," http://ds.arm.com/ds-5/.
[25] "Generalized additive Model of R," https://stat.ethz.ch/R-manual/R-devel/library/mgcv/html/gam.html.
[26] J. H. Lee, N. Nigania, H. Kim, and K. Patel et al, "OpenCL Performance Evaluation on Modern Multicore CPUs," *Sci. Program.*, 2015.
[27] A. Butko et al, "Full-System Simulation of big.little Multicore Architecture for Performance and Energy Exploration," in *MCSoC*, 2016.
[28] K. Van Craeynest et al, "Understanding Fundamental Design Choices in single-ISA Heterogeneous Multicore Architectures," *TACO*, 2013.
[29] S. Williams and A. Waterman et al, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, 2009.
[30] M. Pricopi, T. S. Muthukaruppan, and V. Venkataramani et al, "Power-performance modeling on asymmetric multi-cores," in *CASES*, 2013.
[31] M. Walker, S. Bischoff, S. Diestelhorst, and G. Merrett et al, "Hardware-validated CPU performance and energy modelling," in *ISPASS*, 2018.
[32] J. Shen, J. Fang, H. Sips, and A. Varbanescu, "Performance Traps in OpenCL for CPUs," in *PDP*, 2013.
[33] S. Hong et al, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *ISCA*, 2009.
[34] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and Power Analysis of ATI GPU: A Statistical Approach," in *IEEE NAS*, 2011.
[35] J. Sim et al, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *PPoPP*, 2012.
[36] Q. Wang and X. Chu, "GPGPU Performance Estimation with Core and Memory Frequency Scaling," *CoRR*, 2017.
[37] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," *TPDS*, 2017.
[38] V. Tiwari and S. Malik et al, "Instruction level power analysis and optimization of software," in *Technologies for wireless computing*, 1996.
[39] W. Huang, C. Lefurgy, W. Kuk, A. Buyuktosunoglu, and M. Floyd et al, "Accurate Fine-Grained Processor Power Proxies," in *MICRO*, 2012.
[40] M. J. Walker and S. Diestelhorst et al, "Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs," *TCAD*, 2017.
[41] W. Dargie, "A Stochastic Model for Estimating the Power Consumption of a Processor," *TC*, 2015.
[42] R. Ge and X. Feng et al, "CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters," in *ICPP*, 2007.
[43] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Mobile Computing*, 1994.
[44] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.
[45] "ARM Compute Library," https://developer.arm.com/technologies/compute-library.

Siqi Wang received the B. Eng. (Hons) degree in Electrical Engineering from the National University of Singapore in 2014. She is currently a research assistant and is working toward the Ph.D. degree in the department of Computer Science at the School of Computing, the National University of Singapore. Her current research interests include performance and energy optimization, task scheduling and general-purpose GPUs on heterogeneous multi-processor embedded systems.

Gayathri Ananthanarayanan received the B.Eng. (Hons) degree in Electronics and Instrumentation Engineering and Masters degree in Embedded Systems from Birla Institute of Technology and Science, Pilani in 2008 and 2010. She obtained her Ph.D. degree from Indian Institute of Technology Delhi in 2017. She is a Post-doctoral researcher at the School of Computing, National University of Singapore. Her research interests lie in the broad areas of computer architecture and embedded systems, with specific focus on Power, Thermal and Performance modeling, characterization and management.

Tulika Mitra is a Professor of computer science at the School of Computing, National University of Singapore, Singapore. Her research interests include the design automation of embedded real-time systems with particular emphasis on software timing analysis/optimizations, application-specific processors, energy-efficient computing, and heterogeneous computing. She received a BE degree in computer science from Jadavpur University, Kolkata, India, in 1995, an ME degree in computer science from the Indian Institute of Science, Bengaluru, India, in 1997, and a PhD degree from the State University of New York, Stony Brook, NY, USA, in 2000.