# Energy-Efficient Execution of Data-Parallel Applications on Heterogeneous Mobile Platforms

Alok Prakash[1], Siqi Wang[2], Alexandru Eugen Irimiea[3] and Tulika Mitra[4]
School of Computing, National University of Singapore, Singapore, 117417
Email: (alok[1],wangsq[2],tulika[4])@comp.nus.edu.sg, alexandru.irim@gmail.com[3]

***Abstract*—State-of-the-art mobile system-on-chips (SoC) include heterogeneity in various forms for accelerated and energy-efficient execution of diverse range of applications. The modern SoCs now include programmable cores such as CPU and GPU with very different functionality. The SoCs also integrate performance heterogeneous cores with different power-performance characteristics but the same instruction-set architecture such as ARM big.LITTLE. In this paper, we first explore and establish the combined benefits of functional heterogeneity and performance heterogeneity in improving power-performance behavior of data parallel applications. Next, given an application specified in OpenCL, we present a static partitioning strategy to execute the application kernel across CPU and GPU cores along with voltage-frequency setting for individual cores so as to obtain the best power-performance tradeoff. We achieve over 19% runtime improvement by exploiting the functional and performance heterogeneities concurrently. In addition, energy saving of 36% is achieved by using appropriate voltage-frequency setting without significantly degrading the runtime improvement from concurrent execution.**

## I. Introduction

Over the past decade, desktop, laptops and mobile devices have all witnessed the irreversible transition towards multi-core and eventually many-core architectures (multiple processing cores on the same die) due to power/thermal constraints. In the beginning, the multi-core landscape has been dominated by homogeneous architectures consisting of a collection of identical (and possible simple) cores. These homogeneous multi-cores are simple to design, offer easy silicon implementation, and regular software environments. Unfortunately, general-purpose emerging workloads from diverse application domains have very different resource requirements that are hard to satisfy with a set of identical cores. In contrast, there exist many evidences that heterogeneous multi-core solutions consisting of different core types can offer significant advantage in terms performance, power, area, and delay. Thus heterogeneity has emerged as a promising solution in the face of complex, dynamic, and diverse applications.

We can broadly classify heterogeneous multi-cores into *performance heterogeneity*, where cores with the same functionality (same instruction-set architecture or ISA) but different power-performance characteristics are integrated together and *functional heterogeneity*, where cores with very different functionality (different ISA) are interspersed on the same die. In performance heterogeneous multi-core architectures, the difference stems from distinct micro-architectural features such as in-order core versus out-of-order core. The

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115

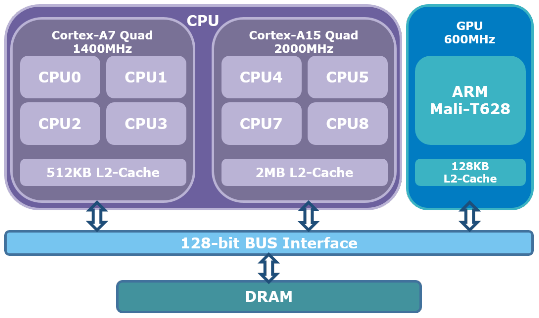

Fig. 1: Block Diagram of Exynos 5422 SoC featuring both performance and functional heterogeneity.

complex cores can provide better performance at the cost of higher power consumption, while the simpler cores exhibit low-power behavior alongside lower performance. An example of commercial multi-core featuring performance heterogeneity is ARM big.LITTLE [11] architecture integrating high-performance out-of-order ARM Cortex-A15 cores (big cores) with low-power in-order ARM Cortex-A7 cores (small cores). Functionally heterogeneous multi-cores comprise of cores with different functionality. This is fairly common in the embedded space where a multiprocessor system-on-chip (MPSoC) consists of general-purpose CPU cores, GPU cores, DSP blocks, and various hardware accelerators or IP blocks (e.g., video encoder/decoder, imaging, etc.). The heterogeneity is introduced here to meet the performance demand under stringent power budget as certain workloads are more suitable for GPUs, DSPs, or fixed-function accelerators. In this work, we focus on programmable heterogeneous cores in mobile platforms, namely CPU and GPU cores. Unlike performance heterogeneity, which is transparent to the software developer, functional heterogeneity is harder to exploit due to differences in ISA. Fortunately, new framework such as OpenCL [6] has been developed for writing programs that can execute across heterogeneous processing elements including CPU, GPU, DSP, and FPGAs. Previous works have explored the possibility of executing OpenCL programs across CPUs and GPUs but mostly from performance perspective [15][12][13]. Moreover, all these works have exclusively focused only on functional heterogeneity.

Recently, the mobile platforms (specially smartphones, tablets, and wearables) are adopting both performance and functional heterogeneity in the same chip. Figure 1, for instance, shows a simplified block diagram for the Samsung Exynos 5422 SoC [2] that powers the popular Galaxy S5 smartphone manufactured by Samsung. The SoC contains a high performance multi-core ARM Cortex-A15 CPU cluster, a low power multi-core ARM Cortex-A7 CPU cluster alongside a six-core ARM Mali T628 MP6 GPU. Moreover, similar to their desktop counterparts, the GPUs in these devices are also gradually becoming fully capable of performing general purpose computation with the help of OpenCL. The presence of such high performance components in a portable device enables users to run sophisticated applications such as video editing, immersive 3D games, etc. that could not have been possible only a few years ago. While the multi-core CPU takes care of task/thread-level parallelism, the GPU handles data-level parallelism. But most applications still have substantial sequential fraction needing to be accelerated through architectural approaches such as multi-issue out-of-order execution (as in ARM Cortex-A15) that can extract instruction-level parallelism from serial code transparently. The question remains is how best to partition and execute an OpenCL program across small CPU cores, big CPU cores, and the GPU so as to provide best power-performance tradeoff. Also, both the CPU and the GPU cores in state-of-the-art mobile platforms include dynamic voltage-frequency scaling (DVFS) capability that can further improve the power-performance behavior of the application.

*In this work, we explore, for the first time, the benefits of functional and performance heterogeneity along with DVFS in mobile platforms for general-purpose computing workloads.* Given an application, specified in OpenCL, we statically partition the application to run across CPU and GPU clusters on mobile application processors in conjunction with appropriate voltage-frequency setting for each core cluster (small core, big core and GPU). Our objective is to maximize the power-performance tradeoff of the application. The concrete contributions of this work are as follows:

- To the best of our knowledge, this is the first study that explores both functional and performance heterogeneity for a single application and shows concrete advantages of both kinds of heterogeneity depending on the application.
- We believe, this is also the first work that investigates partitioning of OpenCL applications across CPU and GPU in a mobile application processor. In mobile SoC, the CPU and the GPU need to share resources such as the memory bandwidth that has significant impact on the performance of the individual components. We carefully model the impact of resource sharing. Moreover, embedded GPUs are not as powerful as desktop discrete GPUs and hence the tradeoffs are very different.
- We develop a static partitioning algorithm that maximizes the power-performance tradeoff by taking into consideration not only the characteristics of the different cores but also their appropriate frequency scaling point. As energy is the most important metric in mobile platforms, DVFS in conjunction with partitioning of the application is imperative, but has not been examined before.
- We implement our approach on a state-of-the-art mobile application processor (Samsung Exynos 5422) prevalent in smartphone/tablet devices and we report power-performance results obtained directly from this real platform rather than relying on simulations.

## II. Related Work

Performance and power optimization of general purpose computing workloads on functionally heterogeneous computing systems has been extensively studied [13][15][16][21]. Traditionally, such heterogeneous systems contain multi-core CPUs for general purpose tasks, while an integrated or discrete GPU is used to accelerate data-parallel tasks in the applications [15]. In the context of discrete GPU platforms, several works [15][12][13][21][16] have been proposed to exploit the parallelism in tasks to concurrently execute on both CPU and GPUs, to improve throughput [15][12][13] and energy efficiency[15].

Qilin[15] employs an offline profiling step to create a regression model that predicts the execution time for applications at runtime with arbitarary input size on CPU or GPU. However, it requires extensive profiling, which is not required in our work. Grewe et al.[12] proposed a static task partitioning approach based on predictive modeling and program features. The SVM based model requires only static program features that do not need extensive profiling. A follow-up work[13] by the same authors also incorporate the effect of GPU contention. However, this paper focuses on executing multiple applications on GPU simultaneously. Current embedded platforms do not allow concurrent executions to the best of our knowledge.

In [21], the authors proposed an SVM based prediction model according to static code structures. The applications are then dispatched to CPU or GPU according to the predicted speedup on the platform. However, they considered multiple applications, unlike our work that splits a single application to run on both CPU and GPU. FluidiCL[16] proposed an OpenCL runtime that uses both CPU and GPU to execute a single application and performs data transfers and merging automatically. The whole kernel is launched on GPU, while sending small chunks of workload to CPU adaptively according to system load. Unlike [16], our approach predicts the runtime and decides the CPU-GPU partition statically at compile time while also exploiting DVFS to achieve energy-efficiency.

Unlike discrete GPU platforms, general purpose computing workloads executing on integrated GPU platforms also suffer due to shared resources[19][17][20]. The coordination of CPU and GPU therefore needs more consideration. Wang et al.[19] considered the total chip power budget of an AMD Trinity single chip heterogeneous platform and proposed a runtime algorithm for workload and power budget partitioning between GPU and CPU to improve throughput. [20] shows that in the CPU-GPU coordinated executions on a similar AMD platform, there is higher possibility of CPU and GPU accessing the same bank due to the dissimilarity of memory access patterns, therefore resulting in memory contention. [17] addressed the problem of shared resources for integrated GPUs in AMD platforms and used DVFS to improve energy efficiency.

In the context executing general purpose computing applications on mobile platforms, [9] examined the Mali GPU performance for HPC workloads, and improved its energy efficiency. This work mainly focused on GPU, without considering the possible collaboration with CPU. [8] proposed a workload partitioning algorithm for heterogeneous MPSoCs with the consideration of shared resources and synchronization. However they do not use GPU for OpenCL kernel execution.

The authors in [14] described their OpenCL framework

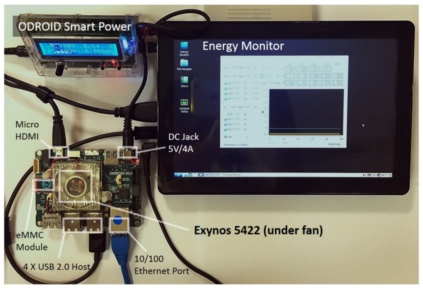

Fig. 2: Experimental Setup with Odroid-XU3: A state-of-the-art mobile platform with Samsung Exynos 5422 SoC.

to support ARM processors. We use a similar open source framework, FreeOCL [3] for the ARM CPU that acts as both the host processor and an OpenCL device.

A technique for automatic work-group size selection for OpenCL kernels for multicore CPUs [18] is proposed to improve cache utilization and load balancing. Similar idea is used in this work to not only improve performance for the GPU, but also to ease the shared resource contention.

## III. Experimental Setup

The goal of this work is to explore the benefits of functional and performance heterogeneity in mobile application processors towards improved power-performance tradeoff. We first present our evaluation platform and the software environment.

### A. *Mobile Application Processor*

In this work, we perform all the experiments on a real state-of-the-art mobile application development platform — Odroid-XU3 [5] from Hardkernel — shown in Figure 2. This platform contains Samsung Exynos 5422 SoC (shown in Figure 1) featuring both functional and performance heterogeneity where all the programmable cores support OpenCL.

As mentioned earlier, the SoC implements ARM big.LITTLE technology with a cluster of four ARM Cortex A15 cores (big cores) and a cluster of four ARM Cortex A7 cores (small cores). All the cores implement ARM v7A ISA. The Cortex-A15 is complex out-of-order superscalar core that can execute high intensity workloads, while Cortex-A7 is a power efficient in-order core meant for low intensity workloads. While each core has private L1 instruction and data caches, the L2 cache is shared across all the cores within a cluster. The L2 caches (2MB for A15 and 512KB for A7) across clusters are kept seamlessly coherent. The architecture provides DVFS feature per cluster. 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. Similarly, A7 cluster frequency can be set between 200MHz to 1400MHz at an interval of 100MHz. The voltage at each frequency level is automatically set by the hardware. An idle cluster can be powered down if necessary. In our current setup, the development platform runs popular Ubuntu 14.04 LTS operating system. This OS version supports Heterogeneous Multi-Processing that allows all the CPU cores (both big and small cores) to be active simultaneously as well as task migration to and from any of the CPU cores [4].

The SoC also includes ARM Mali T628 MP6 GPU implementing “Midgard” architecture with six shader cores that can execute both the graphics and general-purpose computing workloads. Only four shader cores are used by OpenCL runtime. The shader cores share L2 cache (128 KB). But the CPU L2 cache is not kept coherent with the GPU L2 cache even though the GPU is allowed to read from the CPU cache. The main component of the shader core is a programmble massively multi-threaded “tri pipe” processing engine. The tri pipe contains one load-store pipeline, two arithmetic pipelines, and one texture pipeline (unused by OpenCL). The arithmetic pipeline is a VLIW design with SIMD vector characteristics operating on 128-wide registers. That is, the arithmetic pipeline has a mix of scalar and vector (SIMD) ALUs that can be fed with a single long instruction word. The load-store pipeline of each shader core has 16KB L1 data cache. Hundreds of hardware threads can run concurrently in the tri pipe. If some threads are waiting for memory, other threads can execute in the arithmetic pipeline thereby hiding the memory latency. A significant difference of this architecture from other GPU architectures is that the arithmetic pipelines are independent and can execute threads that are different, for example, in case of divergent branches and memory stalls. The GPU can be clocked at seven different voltage-frequency settings between 177MHz–600MHz as shown in Table I.

TABLE I: The available voltage-frequency settings for GPU

| Frequency (MHz) | 600 | 543 | 480 | 420 | 350 | 266 | 177 |
|---|---|---|---|---|---|---|---|
| Voltage (mV) | 975 | 962.5 | 912.5 | 875 | 850 | 975 | 762.5 |

The platform provides current sensors, one each for the A7 cluster, the A15 cluster, the GPU, and the DRAM main memory. The current sensors can be sampled at 4Hz to obtain continuous power readings for the different on-chip components. The implemented Linux Kernel (v3.10.69) also allows us to change the frequency of the different components by editing the appropriate virtual files for the corresponding devices in the Linux *sysfs* directory.

### B. *Software Environment*

In this work, we execute an application across the CPU and GPU cores through OpenCL. We now provide a short background on OpenCL followed by our runtime environment and how we partition an OpenCL kernel to execute on both CPU and GPU cores.

**OpenCL Background:** The Open Computing Language (OpenCL) [6] is an open standard for developing parallel applications in heterogeneous multi-core architectures including CPU, GPU, DSP, and FPGAs. Vendors who support OpenCL for their devices are responsible for the OpenCL runtime software and compilation tools that facilitate development and execution of OpenCL programs on the device. Most importantly, OpenCL allows runtime software from different vendors to co-exist so that a single program can exploit multiple devices.

In OpenCL model of computation, a *host* code segment running on the CPU controls one or more *compute devices* that perform the computations called the *kernels*. A compute device can be a CPU, GPU, DSP, or even FPGAs. The host code is responsible for setting up the devices and schedule kernels for execution on them using OpenCL API functions. It also needs to send and receive the data from the compute devices before and after the execution, respectively. The kernel or device code on the other hand is created and built on the host using OpenCL APIs during runtime and finally scheduled on the OpenCL device for execution.

Each compute device (e.g., GPU) consists of *compute units* (e.g., shader cores in Mali) and each compute unit consists of *processing elements* (e.g., arithmetic pipelines). Each kernel instance is called a *work-item* that operates on a single data point and executes on a processing element. A group of work-items constitutes a *work-group* that execute concurrently on the processing elements of a single compute unit. The work-items of an OpenCL program operate along the index space, termed as *NDRange*, of the input data for data-parallel applications. All the work-items execute the same code but may follow different control paths. In order to relate OpenCL execution model to the popular CUDA model, it is often easier to visualize the OpenCL work-item as CUDA thread, work-group as thread-block and NDRange as grid. Each work-item has private memory, while each work-group has a local memory shared across all the work-items in the work-group. All work-groups have access to a global memory that is also accessible by the host. The OpenCL memory model demands memory consistency across work-items within a work-group but not among different work-groups. This enables different work-groups to be launched on different compute devices (e.g., CPU and GPU) without worrying about maintaining memory consistency among the devices.

**OpenCL Runtime:** The OpenCL runtime software for the Mali GPU is supplied by the vendor in an effort to promote the usage of the GPU for general-purpose computing applications. On the other hand, current mobile SoCs typically do not ship with OpenCL support for the ARM CPU cores [14]. In order to explore the concurrent execution of OpenCL applications on CPU alongside the GPU, we compiled and installed an open-source OpenCL runtime, called FreeOCL [3] on this platform. This enabled each of the eight CPU cores (four big and four small cores) to be used as an OpenCL compute unit. From the perspective of the OpenCL programmer, there is no difference between the big and LITTLE cores. Moreover, unlike other open-source OpenCL runtimes such as [7], FreeOCL also enabled us to schedule and launch OpenCL kernel concurrently on all CPU cores and GPU in this SoC.

```
1  /*Pseudocode for splitting on CPU and GPU */
2  splittingPoint = (global_work_size *
       split_fraction) / work_group_size;
3  //Parameters for workload on CPU
4  globalWorkSizeCPU = splittingPoint *
       work_group_size;
5  offsetCPU = 0;
6  //Parameters for workload on GPU
7  globalWorkSizeGPU = (global_work_size /
       work_group_size - splittingPoint) *
       work_group_size;
8  offsetGPU = splittingPoint * work_group_size;
9  //Enqueue OpenCL kernels
10 clEnqueueNDRangeKernel(clCPUCommandQue,
       clCPUKernel, dim, offsetCPU,
       globalWorkSizeCPU, work_group_size, 0,
       NULL, NULL);
11 clEnqueueNDRangeKernel(clGPUCommandQue,
       clGPUKernel, dim, offsetGPU,
       globalWorkSizeGPU, work_group_size, 0,
       NULL, NULL);
```

**OpenCL code partitioning across CPU-GPU:** The pseudo-code shown above describes how an OpenCL kernel computation is partitioned across the CPU and the GPU cores. Given an application, we statically determine the fraction of work-items to be executed on each device. In order to facilitate the execution of the OpenCL kernels, the work load (global work_size) on both CPU and GPU should be multiples of work-groups size (Line 2). Therefore, *splittingPoint*, which is used for the actual splitting, is calculated to be the number of work-groups that is nearest to the desired fraction of CPU work load (split_fraction). Next, the global work-size and offset values for the CPU and GPU are calculated based on the splittingPoint as shown in the pseudo-code (Lines 4-8). The partitioned application is subsequently executed by enqueuing kernels on both devices with the new global work-size and offset values (Lines 10-11).

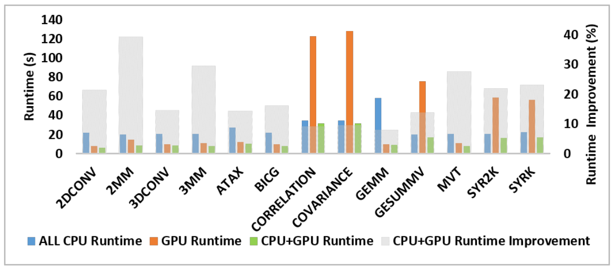

Fig. 3: Runtime improvement with CPU and GPU Partitioning.

### *C. Benchmark Applications*

We use the GPU version of the popular Polybench benchmark suite [10]. This suite contains data-parallel applications written in OpenCL. Each application includes a comparison code in the end that greatly helps in verifying the output from the parallel OpenCL kernel execution. The application codes are minimally modified to launch on CPU-alone, GPU-alone or on both CPU and GPU based on an input argument. Amongst the 15 applications in this suite, only the FDTD-2D application can not be executed on our platform due to a bug in the original code. Also, the GRAMSCHMIDT application can not be split for concurrent CPU-GPU execution. Therefore, we use the remaining 13 applications for all the experiments in this work. We select the appropriate work-group size for each benchmark as discussed in Section V-A.

## IV. Advantages of Heterogeneity

We first examine the benefit of partitioning an OpenCL application across CPU and GPU cores in a mobile application processor. Figure 3 shows on Y-axis the runtime for execution of OpenCL kernel on all CPU cores (4 A7 + 4 A15), GPU core, and optimally partitioned for performance across CPU and GPU cores. For this experiment, we set the maximum possible frequency for each cluster. We reinforce additional environmental cooling to ensure that the chip never hits its thermal design power threshold. Some applications have better runtime on CPU, while others run faster on GPU. But most applications benefit significantly from utilizing both the CPU and the GPU. The secondary Y-axis shows the percentage runtime improvement for CPU+GPU execution over the best of CPU only and GPU only executions. The improvement can be as high as 40% and on average 19% across all the benchmarks. This experiment clearly establishes the advantages of harnessing the power of both CPU and GPU cores for OpenCL execution on mobile platforms.

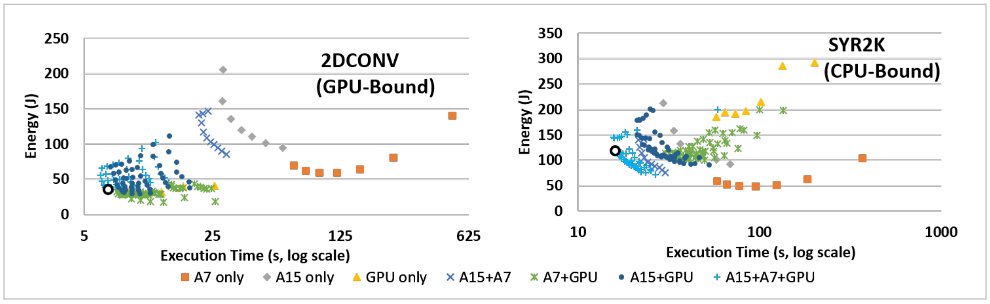

Fig. 4: Design Space for 2DCONV and SYR2K Application. Black circles represent optimal $ED^2$ point

We now delve deeper to understand the impact of heterogeneity and the energy-performance tradeoffs. We present the complex design space of partitioning and frequency selection for running an OpenCL application across small cores, big cores, and the GPU on Exynos SoC. We choose two different benchmark applications to illustrate the tradeoff in the design space: *2DCONV* is an application that is more suitable for the GPU, while *SYR2K* is more suitable for the CPU. For both the applications, we run the OpenCL program using different compute devices.

- **A7 only:** Execute OpenCL kernel on four cores in A7 cluster. We vary the frequency setting for A7 cluster to obtain different design points. To keep the number of design points manageable without losing any insights, we change the frequency at an interval of 200MHz instead of 100MHz.
- **A15 only:** Execute OpenCL kernel on four cores in A15 cluster. We vary the frequency setting for A15 cluster. Again, to keep the number of experiments feasible, we set frequency from 800–2000MHz at an interval of 200 MHz.
- **GPU only:** Execute OpenCL kernel on GPU only. We vary the frequency setting of the GPU.
- **A7+A15:** Execute OpenCL kernel on 8 cores (4 A7 and 4 A15). In this case, the OpenCL runtime on ARM automatically allocates the work-groups to the eight cores depending of the relative speed of execution, that is, we do not pre-select the partitioning of workload between the small and big cluster. We vary the frequency for A7 and A15 clusters individually.
- **A7+GPU:** Execute OpenCL kernel on both A7 cluster and the GPU. In this case, we need to partition the work-items between A7 cluster and GPU. To keep the design space plot relatively uncluttered, we only plot the results for the Pareto-optimal partitioning points. Both A7 cluster and GPU frequency are varied individually.
- **A15+GPU:** Similar to the previous case except that we split the kernel across A15 cluster and GPU.
- **A15+A7+GPU:** Execute OpenCL kernel on A15 cluster, A7 cluster and the GPU. In this case, we need to partition the work-items between CPU and GPU. Note that as in the case of *A7+A15* execution, the partitioning between A7 and A15 cluster is taken care of by the OpenCL runtime. We will statically partition between CPU and GPU. Again, to keep the design space plot relatively uncluttered, we only plot the results for the Pareto-optimal partitioning points. A7 cluster, A15 cluster, and GPU frequency are all varied independently.

Figure 4 illustrates the energy-performance tradeoff for *2DCONV* and *SYR2K* applications. The X-axis plots the execution time in seconds in log scale, while the Y-axis plots the energy in Joules for various design points. It is clear from the plots that the two benchmarks behave very differently. As *2DCONV* benefits from GPU significantly, we can see that the *GPU only* design points are close to the Pareto-front with very low-energy design points, but *A7 only* and *A15 only* executions are very far from the Pareto-optimal design points. On the other hand, for *SYR2K*, *A7-only* executions lead to low-energy Pareto-optimal points (though very inefficient in terms of execution time); but *A15 only* and *GPU only* executions have very high energy consumption without proportionate improvement in execution time. Indeed, it is interesting to note that simple low-power A7 cluster is significantly better compared to more powerful GPU core for this application. These plots clearly illustrate the benefits of performance and functional heterogeneity individually. In terms of performance heterogeneity, as expected, A15 cluster provides better execution time at the cost of higher energy compared to A7 cluster for both benchmarks. Similarly, *A7+A15* executions, as expected, improve the execution time significantly compared to *A7 only* and *A15 only* (because more cores are used) at the cost of higher energy consumption. For functional heterogeneity, the relative benefit of GPU or CPU depends on the application characteristics.

Now let us examine the impact of combined functional and performance heterogeneity, which is the focus of this work. It is evident that utilizing both functional and performance heterogeneity leads to Pareto-optimal design points that are superior in terms of execution time with minimal impact on energy consumption. For *2DCONV* benchmark, engaging both A7 and A15 cores with GPU leads to performance-optimal design points, while engaging only A7 cluster with GPU creates energy-optimal choices. As *SYR2k* does not benefit much from GPU execution, only performance heterogeneity with A7 and A15 clusters brings us close to Pareto-optimal design front. Including the GPU with A7 and A15 clusters reduces the execution time further to reach the Pareto-optimal front. In this case, the most energy-efficient points are simply contributed by A7 cluster with high runtime. Moreover, looking at the points closer to the Pareto front, *2DCONV* works well with functional heterogeneity (A15+GPU), while *SYR2K* is more suitable for performance heterogeneity (A7+A15).

Our static partitioning plus frequency selection approach introduced in the next section can generate the Pareto-optimal design points. However, to simplify the quantitative evaluation of our solution, we use $ED^2$ metric (energy $\times$ delay $\times$ delay) that encapsulates the energy-performance tradeoff. The design space for both applications include the optimal $ED^2$ point shown as a black circle on the Pareto front. The CPU-GPU (A7, A15-GPU) frequency(in MHz) combination and the CPU workload partition for the optimal $ED^2$ point for 2DCONV and SYR2K applications is (1400, 1000-600, 21%) and (1400, 1600-600, 71%) respectively.

## V. Design Space Exploration

In the previous section, we established that executing an application concurrently after partitioning between CPU and GPU, not only helps in reducing the execution time, but also energy with appropriate DVFS settings. In this section, we will discuss the proposed techniques to obtain the appropriate partitions and DVFS settings for the optimal $ED^2$ value.

### *A. Work-group Size Manipulation*

In OpenCL, each kernel instance is a work-item and a group of work-items constitute a work-group (see Section III). Before proceeding to partition the work-groups between CPU and GPU, we first need to select the appropriate work-group size. The challenges as well as the benefits of selecting the right work-group size for OpenCL applications executing on the CPU, especially to improve cache utilization, have been discussed in [18]. The same is also true for executing OpenCL applications on the GPU. This problem is further exacerbated in case of SoC-class GPUs, such as the one used in this work, due to the limited amount of cache available in these devices. For our platform, we observe that the work-group size does not impact the CPU performance due to sufficient cache capacity; but it changes the GPU performance drastically. Therefore, we use the optimal work-group size for the GPU as the work-group size for both CPU and GPU.

The maximum work-group size allowed for Mali GPU is 256 [1] but the maximum size might not be feasible for some applications. The OpenCL API can be used to obtain the maximum possible work-group size for a given kernel. But this work-group size may not be optimal. The OpenCL runtime can also be used to automatically select a work-group size for the kernel if there is no data sharing among work-items [1]. Again, as reported in [9], this does not always produce the best results.

We employ a simple approach to select the best work-group size for an application. We note that the preferred work-group size is in powers of 2 [1]. We exhaustively explore all work-group size in powers of 2 up to the maximum possible work-group size for the application and choose the one that provides the best performance. Figure 5 shows the improvement in execution time with selected work-group size compared to the default work-group size specified in the Polybench suite. Applications *2DCONV* and *3DCONV* do not show any improvement in performance as the default work-group size is itself the optimal value. Overall, we observe an average 40% improvement by selecting the best work-group size. The partitioning and frequency selection are performed with this best work-group size.

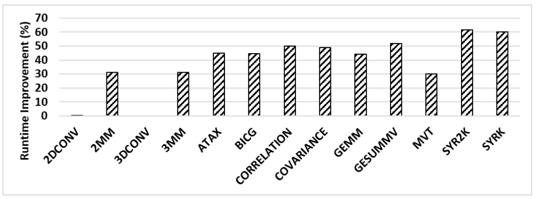

Fig. 5: Runtime improvement with best work-group size.

### *B. Execution Time and Power Estimation*

Selecting the appropriate DVFS point for the CPU clusters and GPU requires us to model the impact of frequency scaling on power-performance behavior of an OpenCL kernel. We can estimate the power-performance behavior of CPU and GPU independently and include the impact of memory contention between the two in the end.

*1) CPU Estimation:* In order to estimate the effect of DVFS for an application, we sample the execution time and power consumption of the OpenCL kernel at the minimum (200 MHz for A15, 200 MHz for A7) and maximum (2000 MHz for A15, 1400 MHz for A7) frequency for each cluster. We then predict the performance and power for the remaining frequency points. During the OpenCL kernel execution, all the CPU cores are utilized to the maximum 100% as the FreeOCL runtime schedules multiple work-groups to a single CPU core similar to [14]. Due to 100% CPU utilization during kernel execution, runtime can be safely modeled using a linear relationship, as shown in equation 1.

$$T = \alpha / f + \beta \tag{1}$$

where $T$ is the execution time, $f$ is the frequency, and $\alpha, \beta$ are constants obtained through interpolation from the runtime at two extreme points.

The total power ($P_{total}$) of the CPU core is estimated using Equation 2, where $A$ is the activity factor, $C$ is the capacitance, $V$ is voltage, $f$ is frequency and $P_i$ is the idle power at the corresponding frequency setting.

$$P_{total} = ACV^2 f + P_i \tag{2}$$

In the first term, since we have no knowledge on the value of $C$ and we have a constant activity of 100%, we grouped the two parameters together and regarded them as one constant $c$. The value of $c$ is determined by taking the average of the values calculated from the two extreme frequency settings, used in the execution time estimation part. The idle power $P_i$ is obtained through experiments performed once at each frequency setting.

To evaluate the accuracy, we run the kernel at all possible frequency values to obatin the actual runtime and power consumption. We then compute the average estimation error for all frequency settings for each application. Figure 6(a) and 6(b) show this average error in execution time and power estimation for A15 and A7 clusters, respectively. The linear model serves well in this case and results in an average estimation error of less than 2% in execution time and less than 6% in power consumption.

*2) GPU Estimation:* Similar to the estimation for the execution time and power consumption during CPU-only execution, we also obtain the model for GPU-only execution. Given an application, we sample its runtime and power at the minimum (177 MHz) and maximum (600 MHz) GPU frequency. Similar to the CPU-only execution, we use the linear model in equation 1 to estimate the execution time and equation 2 for power consumption at the other GPU frequency

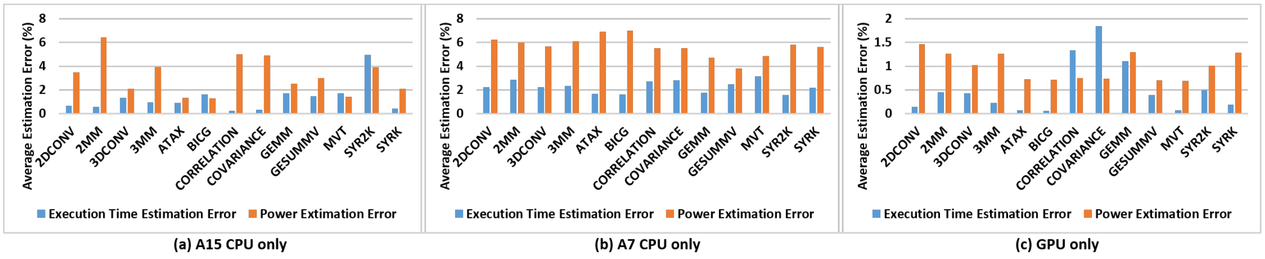

Fig. 6: Estimation error for A15 CPUs only, A7 CPUs only and GPU only execution averaged across frequencies

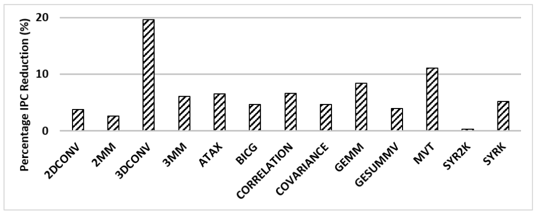

Fig. 7: IPC difference for GPU due to concurrent execution with CPU.

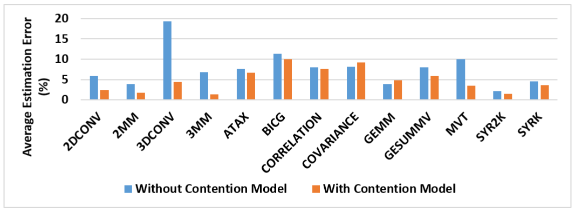

Fig. 8: Runtime Estimation error for concurrent execution averaged across different frequency settings.

settings between the two extremes. The applications are also executed at all the GPU frequency points to obtain the actual runtime, power consumptions and compute the estimation error. Figure 6(c) shows the estimation error in runtime and power estimation for GPU-only execution. The figure confirms the applicability of the linear model and results in an average estimation error of approximately 0.5% in runtime and 1% in power consumption.

*3) Concurrent Execution Estimation:* Let us assume that we choose to run fraction $N$ of the work-items on the CPU cores and rest on GPU at a particular DVFS setting. Now, we need to estimate the execution time and power for concurrent execution. Equation 3 estimates the runtime for the entire application after splitting between CPU and GPU where $T_{CPU}$ and $T_{GPU}$ are estimated runtime for CPU-only and GPU-only execution at the DVFS setting.

$$T_{concurrent} = max\left(T_{CPU} \times N, T_{GPU} \times (1 - N)\right) \quad (3)$$

The total power consumption is estimated by adding the estimated power consumption of individual devices at the DVFS setting.

**Modeling Memory Contention:** The blue (left) columns in Figure 8 show the estimation error in runtime averaged across various DVFS settings for concurrent execution. In this case, we identify the best partitioning for each frequency setting (we discuss how to derive it next), observe the power, runtime for concurrent execution and computed the error compared to estimated values. While the average error stays below 10%, few applications show relatively large estimation error. This is due to the contention for memory bandwidth between the CPU and GPU, especially for applications with frequent memory accesses. Therefore, this interference must be incorporated while estimating the concurrent execution time.

We model the impact of this contention by observing the drop in instructions per cycle (IPC) value of a compute device when executing concurrently with another compute device. We first execute the GPU at the lowest (177 MHz) and highest (600 MHz) frequencies, while keeping the CPU idle. Next, we execute the GPU at the two extreme frequencies while

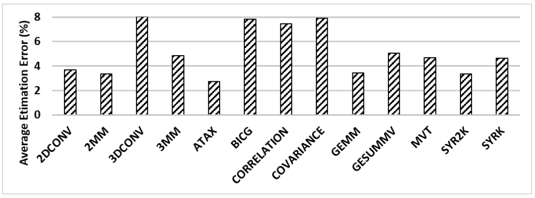

Fig. 9: Power Estimation error for concurrent execution averaged across different frequency settings.

running the CPU in parallel with its own partitioned workload. Figure 7 shows the reduction in GPU IPC due to the sharing of memory bandwidth with the CPU during concurrent execution when compared to the GPU-only execution at the highest GPU frequency. Similar results are also obtained at the lowest GPU frequency. The drop in IPC at the two extreme GPU frequencies is used in a linear model to account for memory contention at other GPU frequencies. We include this contention effect (drop in IPC) in estimating concurrent execution time, thereby reducing the execution time estimation error during concurrent execution. The orange (right) columns in Figure 8 show the estimation error in execution time averaged across various DVFS settings after incorporating this factor. It can be observed that not only does the average estimation error drops from 7.6% to 4.8%; but also there is a significant reduction in the maximum error.

Figure 9 plots the power estimation error averaged across all DVFS points. It can be observed that the power estimation is quite accurate with an average estimation error of 5.1%.

## C. CPU-GPU Partitioning Ratio

We now focus on judiciously partitioning an OpenCL kernel across the CPU and GPU based on their individual capabilities. We use load balancing strategy for each kernel based on its runtime for CPU-only and GPU-only executions. This strategy partitions the workload between CPU and GPU such that both compute devices take the same amount of time to execute the assigned workload portion. We partition the input data (*global_work_size*) of the OpenCL kernels for

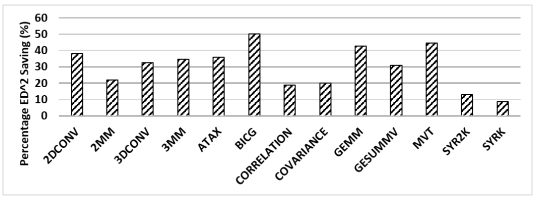

Fig. 10: $ED^2$ improvement

CPU and GPU before launching them concurrently on the two compute devices. Equation 4 gives the fraction (*split_fraction*) of the *global_work_size* of the application that should be executed on the CPU for the optimal load balancing.

$$N = 1/(1+m) \tag{4}$$

where $m$ is the ratio of the execution time between CPU-only and GPU-only executions. We call the partitioning point as the "splittingPoint" to denote the splitting of the original *global_work_size* into two, one each for CPU and GPU.

To identify the best partitioning point as well as frequency setting, we first estimate the CPU-only, GPU-only performance and power as each frequency setting using Equation 1 and 2. Now using these individual power-performance estimations, we can choose the best splittingPoint at each frequency setting with Equation 4. We then estimate the runtime and power for concurrent execution with the selected splittingPoint. This gives us the runtime and power for every point in our design space shown in Figure 4 and hence the Pareto front.

To concretely evaluate the power-performance improvement due to our approach of exploiting heterogeneity, we select the point with the best energy-delay-squared product ($ED^2$) from the design space. The $ED^2$ metric gives more weight to the execution time that ensures minimal execution time degradation while still providing significant energy savings. Figure 10 shows the $ED^2$ savings of the best operating point compared to the best runtime performance scenario (best partitioning at maximum frequency settings for all devices as shown in Figure 3). Table II shows the DVFS settings (A15 CPU and GPU) and CPU workload fraction for the best $ED^2$ values. The frequency of A7 CPU is at 1400 MHz and not shown in the table due to lack of space. Lastly, Figure 11 shows the respective performance degradation and energy savings at optimal $ED^2$ point compared to maximum frequency setting. Most applications exhibit significant energy savings with negligible degradation in execution time (less than 10%).

| App. Name | Frequency (MHz) | | CPU Workload | App. Name | Frequency (MHz) | | CPU Workload |
|---|---|---|---|---|---|---|---|
| | CPU | GPU | Fraction | | CPU | GPU | Fraction |
| 2DCONV | 1000 | 600 | 0.21 | COVAR | 1800 | 600 | 0.52 |
| 2MM | 1600 | 600 | 0.37 | GEMM | 1000 | 600 | 0.12 |
| 3DCONV | 1400 | 600 | 0.29 | GESUM | 1600 | 600 | 0.77 |
| 3MM | 1400 | 600 | 0.28 | MVT | 1000 | 600 | 0.30 |
| ATAX | 1200 | 600 | 0.20 | SYR2K | 1600 | 600 | 0.71 |
| BICG | 1000 | 600 | 0.22 | SYRK | 1600 | 600 | 0.70 |
| CORR | 1800 | 600 | 0.52 | | | | |

TABLE II: Frequency setting and CPU workload fraction for optimal $ED^2$

## VI. Conclusion

We established the advantages of harnessing both functional and performance heterogeneity of mobile application processors from performance and energy perspective. We then presented an approach to statically partition an OpenCL kernel across CPU, GPU cores and select appropriate voltage-frequency settings for different core types so as to reach the optimal power-performance tradeoff. Our experiments on state-of-the-art mobile platform demonstrate an average 19% improvement in runtime across applications by employing both CPU, GPU cores as opposed to utilizing only CPU or GPU. Moreover, we can achieve an average of over 36% energy savings with marginal loss to the earlier runtime improvement by appropriately manipulating the frequency settings.

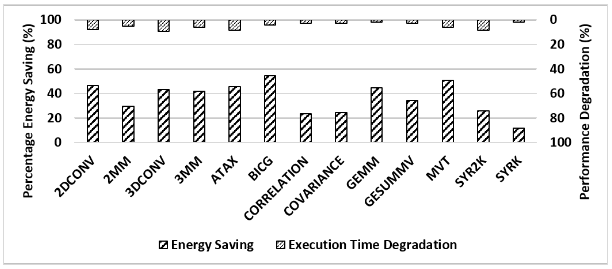

Fig. 11: Energy saving and performance degradation

## References

[1] ARM Mali-T600 Series GPU OpenCL, Version 2.0, Developer Guide. . http://goo.gl/R0FKs8.

[2] Exynos 5 Octa (5422). www.samsung.com/exynos/.

[3] FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting CPUs. https://goo.gl/qWL1Eg.

[4] Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology. (pdf). http://goo.gl/UVAXVS.

[5] Odroid-XU3. http://goo.gl/Nn6z3O.

[6] OpenCL: The open standard for parallel programming of heterogeneous systems. https://goo.gl/A9wXRJ.

[7] pocl: Portable Computing Language. http://portablecl.org/.

[8] K. Chandramohan et al. Partitioning Data-parallel Programs for Heterogeneous MPSoCs: Time and Energy Design Space Exploration. In *LCTES*, 2014.

[9] I. Grasso et al. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. In *PDPS*, 2014.

[10] S. Grauer-Gray et al. Auto-tuning a High-level Language Targeted to GPU Codes. In *InPar*, 2012.

[11] Peter Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[12] D. Grewe et al. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC*, volume 6601. 2011.

[13] D. Grewe et al. OpenCL Task Partitioning in the Presence of GPU Contention. In *LCPC*, volume 8664. 2014.

[14] G. Jo et al. OpenCL Framework for ARM Processors with NEON Support. In *WPMVP*, New York, NY, USA, 2014.

[15] C. Luk et al. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO*, 2009.

[16] P. Pandit et al. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *CGO*, 2014.

[17] I. Paul et al. Coordinated Energy Management in Heterogeneous Processors. In *SC*, 2013.

[18] S. Seo et al. Automatic OpenCL Work-group Size Selection for Multicore CPUs. In *PACT*, 2013.

[19] H. Wang et al. Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors. In *PACT*, 2012.

[20] H. Wang et al. Memory Scheduling Towards High-throughput Cooperative Heterogeneous Computing. In *PACT*, 2014.

[21] Y. Wen et al. Smart Multi-task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *HiPC*, 2014.