Exploiting Heterogeneous Mobile Architectures Through a Unified Runtime Framework

Chenying Hsieh, Ardalan Amiri Sani, Nikil Dutt

Exploiting Heterogeneous Mobile Architectures through a Unified Runtime Framework

Chenying Hsieh, Ardalan Amiri Sani, Nikil Dutt

Department of Computer Science,
University of California, Irvine, CA 92697, USA
{chenyinh, ardalan, dutt}@uci.edu

Abstract. Modern mobile SoCs are typically integrated with multiple heterogeneous hardware accelerators such as GPU and DSP. Resource heavy applications such as object detection and image recognition based on convolutional neural networks are accelerated by offloading these computation-intensive algorithms to the accelerators to meet their stringent performance constraints. Conventionally there are device-specific runtime and programming languages supported for programming each accelerator, and these offloading tasks are typically pre-mapped to a specific compute unit at compile time, missing the opportunity to exploit other underutilized compute resources to gain better performance. To address this shortcoming, we present SURF: a Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures. SURF supports several heterogeneous parallel programming languages (including OpenMP and OpenCL), and enables dynamic task-mapping to heterogeneous resources based on runtime measurement and prediction. The measurement and monitoring loop enables self-aware adaptation of run-time mapping to exploit the best available resource dynamically. Our SURF framework has been implemented on a Qualcomm Snapdragon 835 development board and evaluated on a mix of image recognition (CNN), image filtering applications and synthetic benchmarks to demonstrate the versatility and efficacy of our unified runtime framework.

1 Introduction

Mobile computing has benefited from a virtuous cycle of powerful computational platforms enabling new mobile applications, which in turn create the demand for ever more powerful computational platforms. In particular contemporary mobile platforms are increasingly integrating a diverse set of heterogeneous computing units [1] that can be used to accelerate newer mobile applications (e.g., augmented reality, image recognition, inferencing, 3-D gaming, etc.) that are computationally demanding. The privacy and security needs of these mobile applications (i.e., safely compute on the mobile platform, rather than suffer the vulnerability of sending to the cloud for processing) place further computational stress

[1] In this article we use the terms "compute unit" and "device" interchangeably

on emerging mobile platforms. Consequently, as shown in Table 1, contemporary mobile platforms typically include a diverse set of compute units such as multiple heterogeneous multi-processors (HMPs), and programmable accelerators such as GPUs, DSPs, NPUs, as well as other custom application-specific hardware accelerators.

Table 1: Contemporary Mobile SoCs [10]

Vendor	SoC	CPU	GPU	Other IPs
Qualcomm	Snapdragon	HMP	Adreno	Hexagon DSP
TI	OMAP	HMP	PowerVR	Tesla DSP
NVIDIA	Tegra	HMP	NVIDIA	-
Samsung	Exynos	HMP	Mali	Neural Processor
Apple	A series	HMP	Apple	Neural Processor

However, current mobile platforms and their supporting software infrastructures are unable to fully exploit these heterogeneous compute units for two reasons: 1) existing runtime systems are typically designed for one or a few compute units, thus unable to exploit other heterogeneous compute units that are left idle, and 2) conventional wisdom dictates that certain application codes are best accelerated by specific compute units (e.g., embarassingly parallel codes by GPUs, and filtering/signal processing by DSPs). Consequently, some compute units (e.g., GPUs) can get heavily overloaded with high resource contention resulting in overall poor performance. Indeed, in our recent study [10], we made the case for exploiting underutilized resources in heterogeneous mobile architectures to gain better performance and power; and even counterintuitively using a slower/less efficient but underused compute unit to gain overall performance and power benefits when the platform is saturated. To fully exploit such situations, we believe there is a need for a unified runtime framework for parallel programs that can accept applications and dynamically map them to fully utilize the available heterogeneous architectures.

Towards that end, this article motivates the need for, and presents the software architecture and preliminary evaluation of **SURF**, our Self-aware Unified Runtime Framework for parallel programs, that exploits the range of mobile heterogeneous compute units. SURF is a unified framework built on top of existing parallel programming interfaces to provide resource management and task schedulability for heterogeneous mobile platforms. Using SURF application interfaces, application designers can accelerate application blocks by creating schedulable SURF tasks. The SURF runtime system includes a self-aware task mapping module that considers resource contention, the platform's native scheduling scheme, and hardware architecture to perform performance-centric task mapping. We have implemented SURF in Android on a Qualcomm Snapdragon 835 development board, supporting OpenMP, OpenCL and Hexagon SDK as the programming interfaces to program CPU, GPU and DSP respec-

tively. Our initial experimental results – using a naive, but self-aware scheduling scheme – shows that SURF achieves average performance improvements of 24% over contemporary runtime systems, when the system is saturated with mutliple applications. We believe this demonstrates the potential upside of even larger performance improvements when more sophisticated scheduling algorithms are deployed within SURF.

The rest of this article is organized as follows. Section 2 presents background on existing mobile programming frameworks. Section 3 outlines opportunities to exploit heterogeneous compute units for mobile parallel workloads, and motivates the need for the SURF framework through a case study. Section 4 presents SURF's software architecture. Section 5 presents early experimental results using SURF to execute sample mobile workloads. Section 6 discusses related work, and Section 7 concludes the article.

2 Background

Modern mobile heterogeneous system-on-Chip (SoC) platforms are typically shipped with supporting software packages to program the integrated heterogeneous hardware accelerators. However, there is no unified programming framework. Open Computing Language (OpenCL) was designed to serve this purpose but it ends up being mostly limited to GPU only among mobile platforms. Other compute units such as DSP or FPGA need their own software supporting packages instead of relying on OpenCL. As a consequence, existing infrastructures require a static mapping of the workload to compute units at compile time. severe resource contention for one unit (e.g., the GPU) while underutilizing other units (e.g., DSP). Besides, there is no information sharing between individual device runtimes, which makes it difficult to make intelligent task-mapping decisions even if the schedulability is provided. Hence, existing software infrastructures are unable to exploit the full heterogeneity of compute units. In our previous case study [10], we showed how underutilized heterogeneous resources can be exploited to boost performance and gain power saving when the platform is saturated with workloads – an increasingly common scenario for mobile platforms where users are multi-tasking between mobile games, image/photo manipulation, video streaming, AR, etc. Our study highlighted the need for a new runtime that can dynamically manage and map applications to heterogeneous resources at runtime. To address these challenges, we have built SURF, a unified framework that sits on top of existing parallel programming interfaces to provide resource management and task schedulability for mobile heterogeneous platforms. Using SURF application interfaces, application designers can accelerate application blocks by creating schedulable SURF tasks. Next we analyze the performance of several popular mobile data parallel workloads on heterogeneous compute units to illustrate the potential for SURF to map these computations across these units.

Data-Parallel Workload Characterization Data-parallel computations are common in several mobile application domains such as image recognition (using

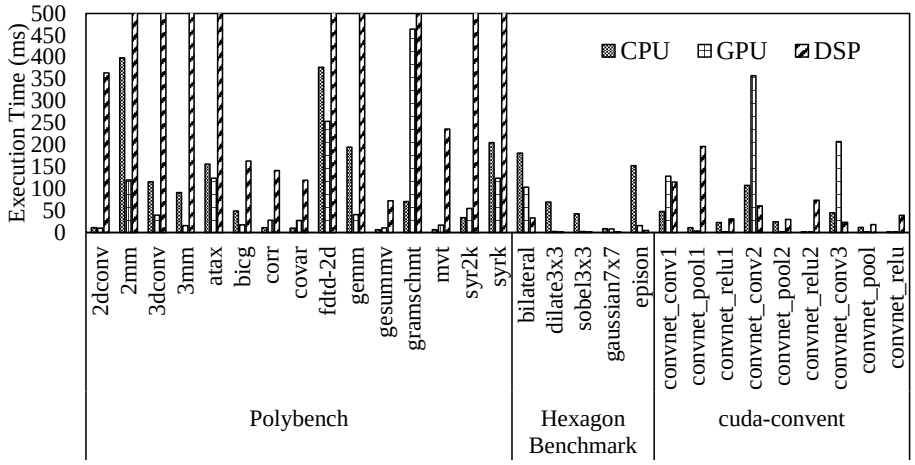


Fig. 1: Execution time of benchmarks on different compute units

CNNs) and image/video processing/manipulation where the same function is applied to a huge amount of data. Due to the simplicity of this programming pattern, they can be easily offloaded to hardware accelerators such as GPUs without substantial programming effort. In order to highlight the opportunity for gaining performance improvement through task mapping/schedulability across heterogeneous compute units, we measured the execution time of two benchmark suites (Polybench benchmark suite [7] and Hexagon SDK benchmark suite [16]), as well as for the critical layers in a CNN (cuda-convnet) that contain several common data-parallel kernels across different domains. In addition to their original implementations, we added OpenMP/CPU, OpenCL/GPU or C/DSP implementations to execute them on different compute units (CPU, GPU, DSP).

Figure 1 shows the measurement results of running each benchmark on the CPU, GPU and DSP respectively. As expected, we typically see one "dominant" version for best performance on a specific compute unit, e.g., syrk and convnet_pool1 have the lowest execution time on GPU, whereas bilateral and convnet_conv2 runs best on the DSP. However, note that the non-dominant (slower) versions (e.g., syrk and convnet_pool1 on CPU or DSP; and bilateral and convnet_conv2 on CPU or GPU) – while seemingly inferior in performance – can be opportunistically exploited by our SURF runtime to improve overall system performance, especially as the mobile platform suffers from high contention when popular apps (e.g., image recognition, photo manipulation/filtering) compete for a specific compute unit (e.g., the GPU for data parallel computations).

3 Motivational Case Study

With abundant compute resources on a mobile chip, a developer typically partitions an application into task kernels to be executed on compute units and

accelerators (e.g., CPU, GPU, DSP) that correspondingly promise a boost in performance. For instance, a convolutional neural network (CNN) application with multiple layers can be partitioned into data-parallel tasks for each layer and mapped onto GPUs for boosting performance. Intuitively, this strict partitioning of tasks to execute them on the highest-performing compute units should result in overall better performance. Mobile platforms often face resource contention when executing multiple applications, saturating these high-performing compute units. In such scenarios – contrary to intuition – offloading of computational pressure to other underutilized and seemingly under-performing compute units (e.g., DSPs) can actually result in overall improvements in performance and energy. Indeed, in an earlier experimental case study [10], we observed an average improvement of 15-46% in performance and 18-80% in energy when executing multiple CNNs, computer vision and graphics applications on a mobile Snapdragon 835 platform by utilizing idle resources such as DSPs and considering all available resources holistically.

In this section, we present this motivational study executing a mix of popular data-parallel workloads and show that both performance and energy consumption of mobile platforms can be improved by synergistically deploying these underutilized compute resources. We select and run three classes of applications: image recognition, image processing and graphics rendering workload, to emulate when the system is heavily-exercised by high computation-demanding applications such as augmented reality and virtual reality applications.

3.1 Experimental Setup

Experiment	Description
CPU-float, CPU-8bit	Run the original or quantized version on the CPU
GPU-float, GPU-8bit	Run the original or quantized version on the GPU
DSP-float	Run the original version on the DSP
DSP-8bit	Run the quantized version on the DSP w/ batch processing
DSP-8bit-nob	DSP-8bit w/o batch processing
Hetero	Layers or stages are statically configured to run on highest-performing compute unit
Hetero-noGPU	Like Hetero but avoid using GPU

Table 2: Keywords used in Experiments [10]

Platform: We use a Snapdragon 835 development board with the Android 6 operating system (which uses the Linux 4.4.63 kernel). The board's SoC integrates custom CPUs with big-LITTLE configurations that conform to ARM's ISA. It also integrates a GPU with unified shaders, all capable of running compute and graphics workloads. The 835 board has two Hexagon DSPs: a cellular modem DSP dedicated to signal processing, and a compute DSP for audio, sensor, and general purpose processing. We target exploiting the compute DSP since it is typically idle.

Applications: For the CNN applications, we select two Caffe CNNs: *lenet-5* and *cuda-convnet* using datasets MNIST and CIFAR10, respectively. MNIST represents a lightweight network with a few layers and low memory footprint whereas CIFAR10 has more layers and high memory footprint. We also implemented a quantized version of Caffe, which supports quantized matrix multiplication using 8-bit fixed-point for convolutional and fully-connected layers. The other layers still perform floating-point computation. The experiments include floating-point and fixed-point versions of CNN models running on CPU, GPU and DSP. For the CED application, we modified Chai CED [9] to support all heterogeneous compute resources for each stage.

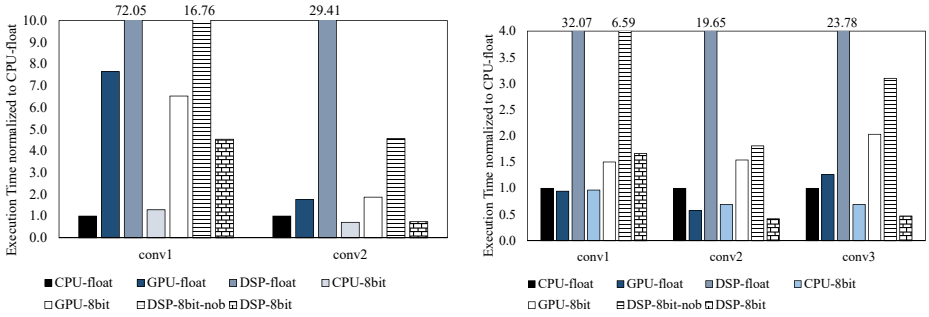
Table 2 summarizes the different experiments by executing the above applications on various compute units (CPU, GPU, DSP, and heterogeneous – including all compute units). In addition to the original floating-point version of CNNs, we also deploy 8-bit quantized versions to exploit the DSP effectively. The row *DSP-8-bit* represents a single function call for batch processing of 100 images to amortize the communication overhead, whereas the row *DSP-8bit-nob* represents no batch processing, i.e., separate function calls for each image.

3.2 Opportunities for Exploiting Underutilized Resources

Figure 2 presents the performance of the convolutional layers of MNIST and CIFAR10. Since the Hexagon DSP is fixed-point optimized, the quantized version (*DSP-8bit*) of the conventional layers are able to outperform some of the other versions. Therefore – following intuition – the performance of a single application can be boosted by allocating the workload to the corresponding highest-performing compute unit. *However – counterintuitively – we may be able to exploit seemingly slower compute units to gain overall performance and energy improvements.* Figure 3 illustrates this scenario, showing the execution time of running one to three instances of CIFAR10 in parallel. When executing only one CIFAR10 instance, the *GPU-only* version yields the best result compared to *GPU-CPU* and *GPU-DSP* versions (as expected). However, when we execute multiple instances of CIFAR10 (i.e., panels showing *CIFAR10*2* and *CIFAR10*3*), we observe that offloading to the other seemingly inferior compute units (e.g., CPU & DSP) yields overall better performance. Indeed, when executing 3 instances of CIFAR10 (*CIFAR10*3*), we see that the performance of *GPU-CPU* and *GPU-DSP* significantly outperform the *GPU-only* version, since the GPU is saturated. This simple example motivates the opportunity to exploit underutilized resources such as DSPs as outlined in Sections 3.3 and 3.4.

3.3 Optimization for Single Application Class

Intuitively, the performance and energy consumption of an application (e.g., CNN) can be improved by partitioning and executing on specific accelerators (e.g., GPUs). But frameworks such as Tensorflow and Caffe run the CNN model on the same GPU, saturating that compute unit while missing the opportunity to improve performance and energy consumption by exploiting other underutilized



(a) MNIST (b) CIFAR10

Fig. 2: Performance of Convolutional Layers [10]

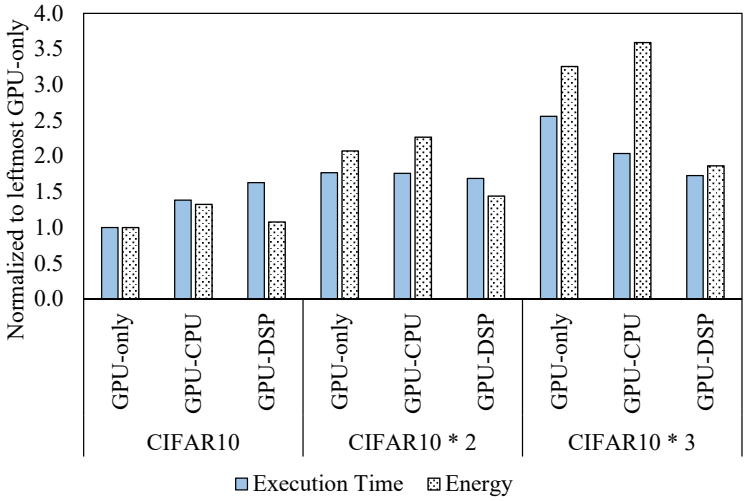
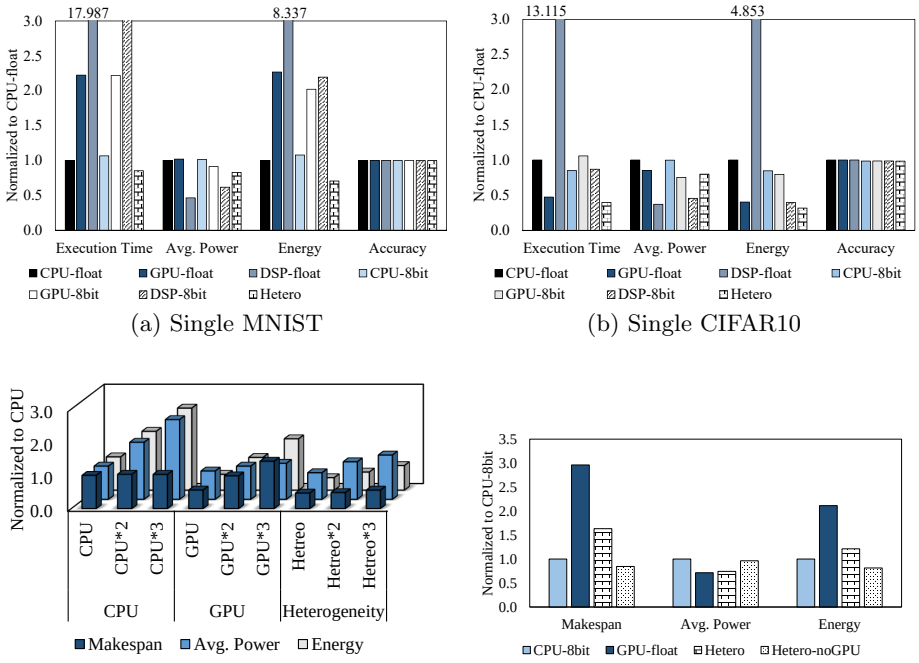


Fig. 3: Performance of executing multiple CIFAR10 instances on different compute units [10]

compute units (e.g., CPU and DSP). Therefore, we partition the neural network at the layer level so each layer can be executed as a task running on a different compute unit to exploit heterogeneity. Figure 4a, 4b shows the execution time, average power and energy consumption of running different versions of MNIST and CIFAR10. For MNIST, *conv2* runs on DSP and the others run on CPU. For CIFAR10, *conv2*, and *conv3* run on DSP, and the others run on GPU. Although *DSP-8bit* has better performance over convolution layers in general as shown in Figure 2, it performs worse due to the floating-point computation in other layers such as the Pooling and ReLu layers. For all quantized models, the accuracy drops 1.4% on average. *Hetero* represents the results of utilizing diverse compute units to gain performance and energy improvements. Indeed, the *Hetero* results show a 15.6% performance boost and a 25.4% energy saving



(a) Single MNIST

(b) Single CIFAR10

(c) Running multiple CIFAR10 instances

(d) Running CIFAR10 with graphics applications

Fig. 4: Performance, power, and energy consumption for single or multiple CNNs/CED with different task mapping [10]

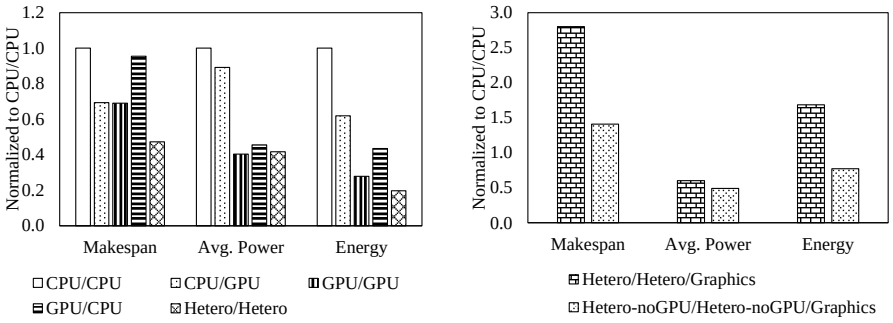
on average compared to *CPU-float* and *GPU-float* (which respectively perform best for MNIST and CIFAR10).

Figure 4c shows the results of running multiple CIFAR10 instances. The results are grouped by CPU, GPU and heterogeneous resources and the values are normalized to *CPU-8bit*. For *CPU-8bit*, the performance is scalable but the power and energy consumption increases drastically with more instances because more cores are exercised. The performance of *GPU-8bit* downgrades along with the increase of instances because they contend for the GPU. *Hetero* shows more stability than the others due to the distribution of the workload over all compute resources. We also simulate the scenario when the GPU is saturated by rendering high-quality graphics. We use the GPU Performance Analyzer benchmark to produce a high quality graphics workload. As Figure 4d shows, the performance of GPU-float and *Hetero* decreased significantly because the GPU is fully-saturated by the above-mentioned graphics workload. *Hetero-noGPU* is statically configured to offload the *conv2*, *conv3* and *relu* layers to DSP while the other layers run on CPU. As *Hetero-noGPU* specifically avoided using the GPU, its performance and energy consumption outperforms the others.

3.4 Optimization for Multiple application Classes

When executing multiple application classes on a system, both the task partitioning and the exploitation of heterogeneous resources help for better distribution of workload, which in turn leads to better performance and energy consumption. Figure 5a presents the results of running different combinations of CED and CIFAR10. *CPU/CPU* represents the static task mapping policy that CED runs only on the CPU and CIFAR10 also runs on the CPU. The other terms in the figure follow the same convention. Makespan is from when we execute all the applications in parallel to when the last application terminates. By exploiting all heterogeneous (including underutilized) resources efficiently, we can achieve better results: the fully heterogeneous with *Hetero* mapping outperforms CPU-only and GPU-only up to 51% for performance and 55% for energy consumption.

Figure 5b presents the results of running all three workload including CED, CIFAR10 and the graphics benchmark. The mapping policy Hetero/Hetero/-Grahpics contends for GPU and therefore fail to achieve better outcome. However, the Hetero-noGPU/Hetero-noGPU/Grahpics policy where we adjust the CED and CIFAR10 to map only on CPU and DSP outshadows the previously policy since GPU becomes the bottleneck due to severe contention.



(a) Mixed workload of image processing and recognition

(b) Mixed workload of image processing, recognition and graphics rendering

Fig. 5: Performance of mixed workload

This scenario highlights the need for runtime decision making for pairing workload from different applications with compute unit according to the system status – something not possible in existing runtimes. Hence, we proposed our runtime model, SURF, to deal with the problem which will be detailed in the next section.

4 SURF: Self-aware Unified Runtime Framework

SURF [11] is a unified runtime framework built on top of existing programming interfaces and device runtime to provide adaptive, opportunistic resource management and task schedulability that exploits underutilized compute resources. Figure 6 shows the architectural overview of SURF. In a nutshell, mobile applications create SURF tasks through SURF APIs. When a SURF task is submitted, a self-aware task mapping algorithm is invoked referencing runtime information of compute units provided by SURF service. After the task mapping decision is made, the corresponding parallel runtime stub executes that task.

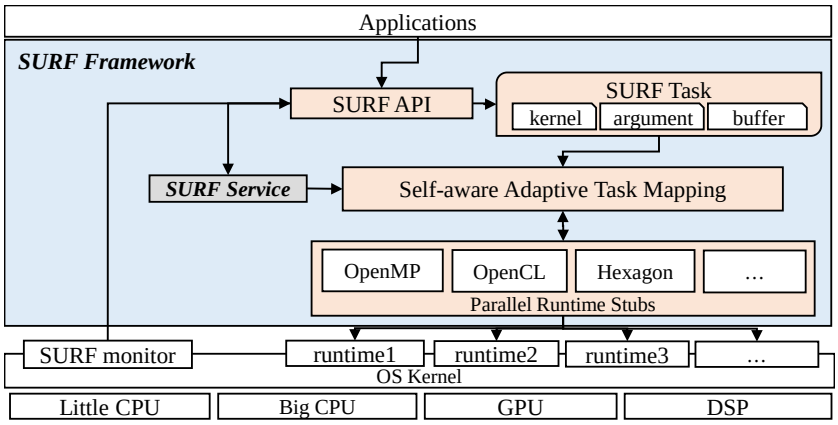


Fig. 6: SURF Architecture

4.1 Application and Task Model

Figure 7 shows the hierarchy of SURF's application model. At the highest level, the mobile platform admits new applications at any time. A newly entering application (e.g., CNN in Figure 7) can create and submit tasks to SURF dynamically. A task (e.g., conv1, pool and relu1 in Figure 7's CNN application) represents a computational chunk (parallel algorithm or application block) that could be a candidate for acceleration. A kernel residing in a task represents the programming-interface-specific implementation artifact to program one compute unit (e.g., OpenMP, OpenCL and Hexagon DSP kernels as shown on the right side of Figure 7). SURF opportunistically maps each task (encapsulating multiple kernels) for scheduling execution on a specific compute unit. All kernels in a task share a set of common inputs and outputs,

The code block in Figure 8 demonstrates an example of how to use the application interfaces to create and execute a 2-dimensional convolution task with three kernels including an OpenMP, a OpenCL and a Hexagon DSP kernel.

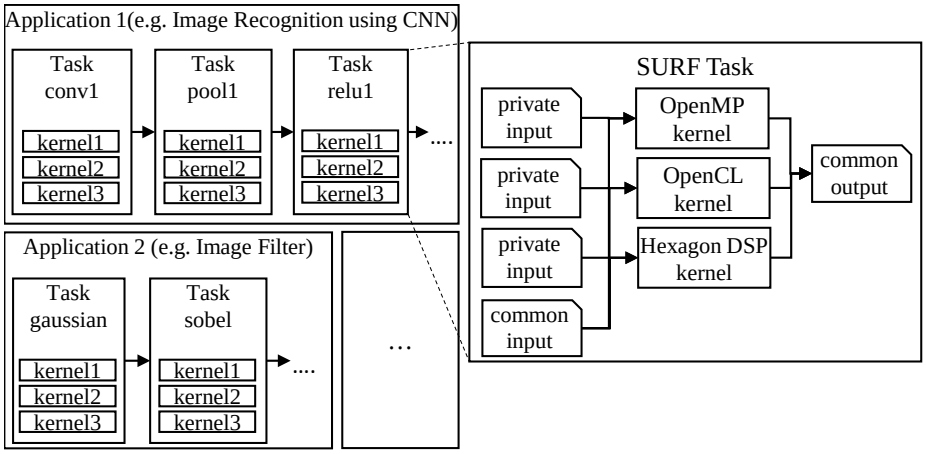


Fig. 7: Application and Task Model

Lines 1-2 create the input and output SURF buffer; Line 4 creates a task; Lines 5-8 add common arguments for all kernels; Lines 9-11 create three kernels to run on CPU, GPU and DSP with user-provided OpenMP binary, OpenCL source code, Hexagon DSP binary respectively, and associate the kernels with the task; and Lines 11-12 execute and destroy the task.

```
1  surf_buffer_t in = surf_buffer_create(size_in);
2  surf_buffer_t out = surf_buffer_create(size_out);
3  /* fill in input buffer */
4  surf_task_t task = surf_task_create(3);
5  surf_task_add_args(task, 0, in, size_in, SURF_MEM_READ | SURF_MEM_BUFFER);
6  surf_task_add_args(task, 1, out, size_out, SURF_MEM_WRITE | SURF_MEM_BUFFER);
7  surf_task_add_args(task, 2, &ni, sizeof(int), 0);
8  surf_task_add_args(task, 3, &nj, sizeof(int), 0);
9  surf_task_create_kernel(task, "conv2D_cpu", SURF_DEV_CPU, SURF_KERNEL_OPENMP |
       SURF_KERNEL_USE_BINARY, "res/libpb.so", 0);
10 surf_task_create_kernel(task, "conv2D_gpu", SURF_DEV_GPU, SURF_KERNEL_OPENCL |
       SURF_KERNEL_USE_SOURCE, "res/2dconv.cl", 0);
11 surf_task_create_kernel(task, "conv2D_dsp", SURF_DEV_DSP, SURF_KERNEL_HEXAGON |
       SURF_KERNEL_USE_BINARY, "res/libconv.so", 0);
12 surf_task_enqueue(task);
13 surf_task_destroy(task);
```

Fig. 8: Sample code of SURF application interfaces including SURF buffer, task and kernel creation as well as SURF task execution and termination.

4.2 Memory Management and Synchronization

SURF assumes compute units are sharing the system memory which is also the dominant architecture in mobile SoCs. Hence, the expensive data movement

between device memory can be ignored if the memory is mapped to all the devices correctly. The SURF buffer object is a memory region mapped to all the device address space through device-specific programming interfaces e.g., OpenCL Qualcomm extension and Hexagon SDK APIs for Qualcomm SoCs. Memory synchronization is still necessary when the buffer is used among different devices to ensure the running device can see the most recent update of data. SURF automatically synchronizes memory objects when the memory object is going to be used by a different device; this memory overhead is included in SURF's task mapping decision.

4.3 Self-aware adaptive task mapping

SURF employs a self-aware adaptive task mapping strategy. SURF exhibits self-awareness [5] by creating a model of the underlying heterogeneous resources, assessing current system state via the SURF monitor, and using predictive models to guide mapping decisions. This enables SURF to act in a self-aware manner, combining both *reactive* (e.g., as new applications arrive or when active applications exit), as well as *proactive* (e.g., through the use of predictive models to enable evaluation of opportunistic mapping to underutilized compute units) strategies to enable efficient, adaptive runtime mapping.

SURF's current implementation deploys a variant of the heterogeneous earliest finish time (HEFT) [17] task mapping algorithm, enhanced to incorporate the cost of runtime resource contention. We consider two types of contention:

intra-compute-unit the contention happens when multiple tasks are submitted to a compute unit. The cost of the contention depends on the device runtime and the hardware architecture. For compute unit accelerators such as GPU and DSP, the task execution is usually exclusive due to costly context switch overheads. A FIFO task queue is implemented for each compute unit, so we include the wait time in the queue when calculating the finish time for a task. We also consider device concurrency (i.e., how many tasks can run concurrently on a device) in the analysis. Contemporary mobile GPUs can only accommodate one task execution at a time. Other devices such as DSPs may have more than one concurrent task execution (e.g. Qualcomm Hexagon DSP supports up to 2 when setting to 128-byte vector context mode [16]). And of course for the CPU cluster we can have multiple, concurrent tasks executing across the big.LITTLE cores, that typically employs an existing sophisticated scheduler such as the Linux Completely Fair Scheduler (CFS) [14].

inter-compute-unit Typically memory contention is the major bottleneck when there are concurrent memory-intensive task executions in different compute units, resulting in the execution makespan of a task increasing significantly.

Figure 9 shows SURF's dynamic task mapping scheme. SURF proposes a heuristic-based scheme to estimate the finish time for a task running on different compute units considering both intra- and inter-compute-unit contention. First, to determine which compute unit has the fastest execution time, a new task

starts within a profile phase to measure the execution time for all kernels in the task. Mapping phase comes after the profile phase is finished where it begins to find the earliest finish time based on the runtime information. Policy determines how to perform the task mapping according to the task profile and device load. Equation 1 shows how we estimate the finish time. T_{task}^{cu} is the finish time when

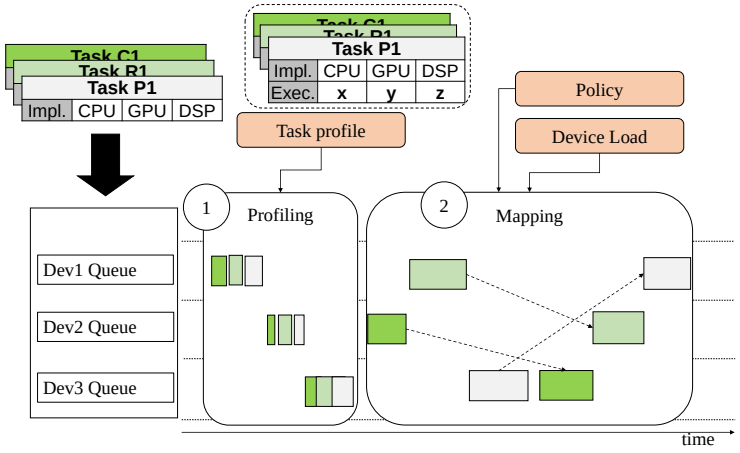


Fig. 9: SURF Task Mapping Scheme

executing task t on compute unit cu. T_{inter} is the execution time considering inter-compute-unit contention. The influence of memory contention to execution time is difficult to estimate at runtime because the micro-architecture metrics for hardware accelerators are usually not feasible; hence we use a history-based method to model that effect. A history buffer is introduced to track execution time of the latest n runs. T_{inter} is the average of the history buffer. T_{intra} is the execution time considering intra-compute-unit contention. For GPU/DSP, T_{intra} is the sum of execution time of earlier submitted tasks. For CPU, T_{intra} is complicated to estimate if left unbounded. So we estimate the worst execution time based on OpenMP programming model and assume the active CPU threads have the same priority under CFS policy (each thread is allocated with the same time slice). SURF configures an OpenMP kernel to execute on a CPU cluster with a thread on each core. Hence, we approximate the worst execution time by Equation 2. TPC is the number of concurrent OpenMP tasks in the CPU cluster. T_o represent the overhead of deploying the task to the compute units and the memory synchronization if it is necessary (e.g., memory buffer is written by GPU and CPU is going to use the results). SURF finds the kernel with the minimum T_{task}^{cu} and submits it to the SURF device queue for execution.

$$T_t^{cu} = T_{inter}^{cu} + T_{intra}^{cu} + T_o, cu \in \{CPU, GPU, DSP\} \tag{1}$$

$$T_{intra}^{cpu} = TPC * T_{inter}^{cpu} \quad (2)$$

4.4 Parallel Runtime Stub

Parallel runtime stub is an abstract layer on top of the existing programming interfaces. This layer utilizes their interfaces to communication with the corresponding runtime. The corresponding stub provides the following features: a) Initialization of programming resources for different programming interfaces accordingly; b) Memory management and synchronization: while the shared system memory model between heterogeneous compute units is dominant in mobile SoCs, and saves expensive data movement, it still needs to perform memory synchronization between cache and system memory before another compute unit accesses the memory; and c) Computation kernel execution. SURF currently supports three programming interfaces: OpenMP, OpenCL and Hexagon SDK to program CPU, GPU and DSP respectively.

4.5 SURF Service and Monitor

The SURF service is a background process that synchronizes the system information with application processes. The SURF Monitor collects system status and profile results. For example, we collect execution time of OpenMP threads from the entity *sum_exec_runtime* through sysfs so to estimate how long an OpenMP kernel runs.

5 Experimental Results

5.1 Experimental Setup

Figure 10 shows our experimental setup. We have implemented the SURF framework using C/C++ in Android 7 running on Qualcomm Snapdragon 835 development board, which has two CPU clusters (big.LITTLE configuration), and integrated GPU and DSP. SURF considers the little CPU cluster, big CPU cluster, GPU and DSP as four compute units when making task mapping decisions where GPU and DSP are exclusive for 1 and 2 tasks respectively. SURF kernels can be created by the programming interfaces of OpenMP, OpenCL and Hexagon SDK to program CPU, GPU and DSP respectively. We deploy the Caffe convolutional neural network framework [12], Canny Edge Detector (CED), Polybench benchmark suite and Hexagon SDK benchmarks to run on SURF. We also use the Snapdragon Profiler [15] to measure the utilization for each compute unit. Power consumption is measured by averaging the product of voltage and current read from the power supply module through Linux *sysfs* interface (e.g. */sys/class/power_supply*). Energy consumption is the product of makespan and average power consumption. We also access Android Debug Bridge (adb) through WiFi instead of USB connection so the USB charging will not compromise the

Table 3: Details of applications and benchmarks used in our experimental sets

Name	Source	Category	#tasks	Dominant Device
CUDA-Convnet	Caffe	Image Recognition	9	mixed
Canny Edge Detector	Synthetic	Image Filter	4	mixed
syrk	Polybench	Linear Algebra	1	GPU
gemm	Polybench	Linear Algebra	1	GPU
bilateral	Hexagon SDK	Image Filter	1	DSP
epsilon	Hexagon SDK	Image Filter	1	DSP

Table 3: Details of applications and benchmarks used in our experimental sets (continuation)

Name	#Iteration	Workload		
		Heavy(H)	Medium(M)	Light(L)
CUDA-Convnet	150	batch=100, 32x32	n/a	batch=10
Canny Edge Detector	150	batch=100, 640x354	n/a	batch=1
syrk	200	512x512	384x384	256x256
gemm	200	768x768	512x512	256x256
bilateral	200	3840x2160	1920x1080	1280x960
epsilon	200	7680x4320	3840x2160	1920x1080

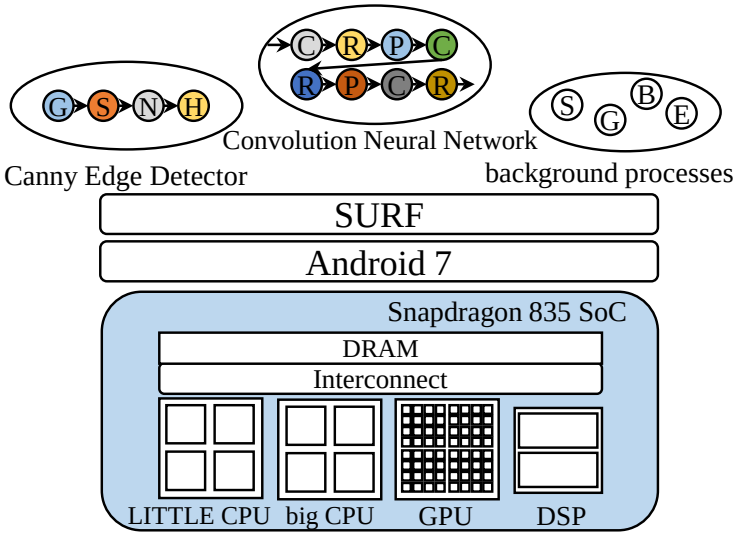


Fig. 10: Experimental Setup

results. The big.LITTLE processor governors are set to performance mode so as to not interfere with our performance-centric task mapping.

In our experimental sets, we run two applications: image recognition (cuda-convnet within Caffe and with Cifar10 dataset) and image filter (CED) representing foreground processes that have 9 and 4 SURF tasks respectively. We also run two GPU-dominant benchmarks (*syrk* and *gemm* from Polybench) and two DSP-dominant benchmarks (*bilateral* and *epsilon*) representing background processes and each of the benchmarks runs one SURF task. We characterize application workloads as heavy and light workloads by changing batch processing size (how many images are processed each iteration) and benchmark workloads as heavy, medium and light workload by changing their input size. Light workload is characterized as real-time workload which can be done within 20ms. Medium and heavy workload are the ones can be done within 20-100ms and above 100ms respectively. Table 3 summarizes the configurations of applications and benchmarks used in our experimental sets.

5.2 Experimental Results

As Table 4 shows, we run six test sets composed of combinations of heavy/-light applications and heavy/medium/light benchmarks. Figure 11 shows the

Table 4: Speedup for different test sets

	Foreground-Background Workload	Speedup	Makespan Difference (s)
Set1	H-H	1.33	19.07
Set2	H-M	1.17	7.15
Set3	H-L	1.04	1.43
Set4	L-H	1.34	12.60
Set5	L-M	1.34	5.07
Set6	L-L	1.22	1.72

execution makespan of running our six test sets with static best-performing task mapping and SURF dynamic task mapping. The static best-performing mapping configures each task to run on their best-performing compute unit according to the profiling results without SURF. SURF's dynamic task mapping outperforms static mapping by 24% on average. Table 4 also shows that the speedup increases with the level of the background benchmark workload because for heavy background benchmarks, a single run of them will occupy the compute resources for long time in GPU and DSP, which creates opportunities to map alternative kernels to exploit other underutilized compute units. The light applications have better speedup than heavy applications because the light application setup experiences more contention with background processes during the entire makespan and it's easy to find alternative kernels because the kernels in one task tend to have similar performance in light workload configuration. Figure 12 shows the sum of all device utilization of the makespan including little/big CPUs, GPU and DSP utilization when running each test set (max 400% across the 4 classes

of units). GPU and DSP utilization are similar (increased by 4.51% and 3.38% respectively) across all in general, since the GPU and DSP are heavily exercised. Here the Big CPU is better utilized (increased by 30.6%) by our dynamic scheme, and is the major contributor to the speedup.

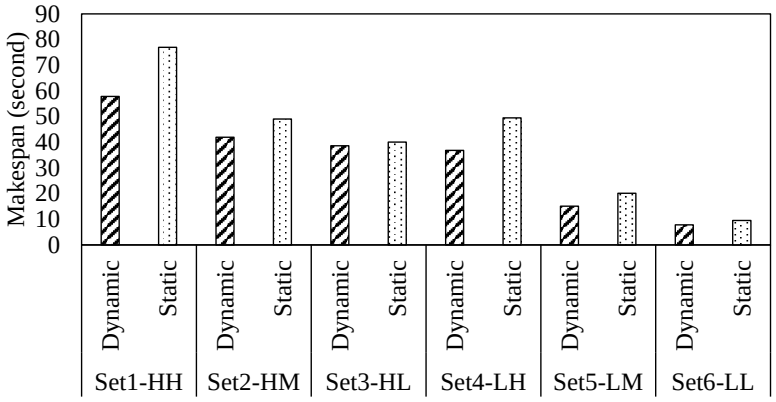


Fig. 11: Makespan of static and SURF dynamic task mapping

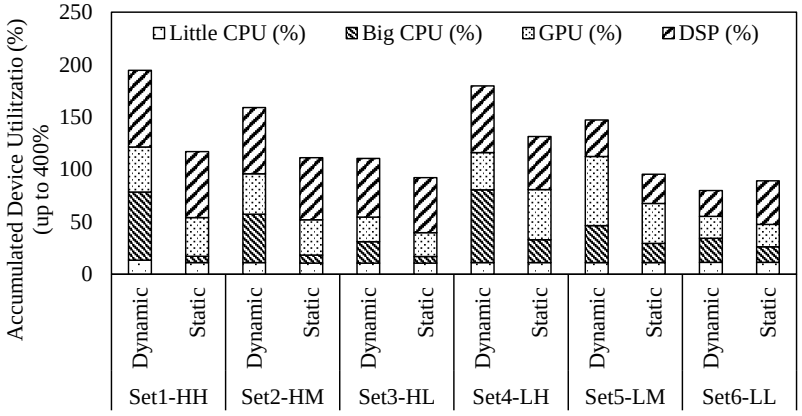


Fig. 12: Device utilization of makespan for each test set

Further experiments were conducted where DSP-dominant background processes (bilateral and epsilon) are not executed, but only foreground applications and GPU-dominant background processes. Figure 13 and 14 show the results of makespan and utilization respectively. SURF's dynamic scheme outperforms

static mapping by 27% in performance which is slightly better than the previous experiments because there are more available resources while GPU is saturated. The utilization for big CPU and DSP increased by 43.15% and 8.13% respectively which shows part of the computation are offloaded to them.

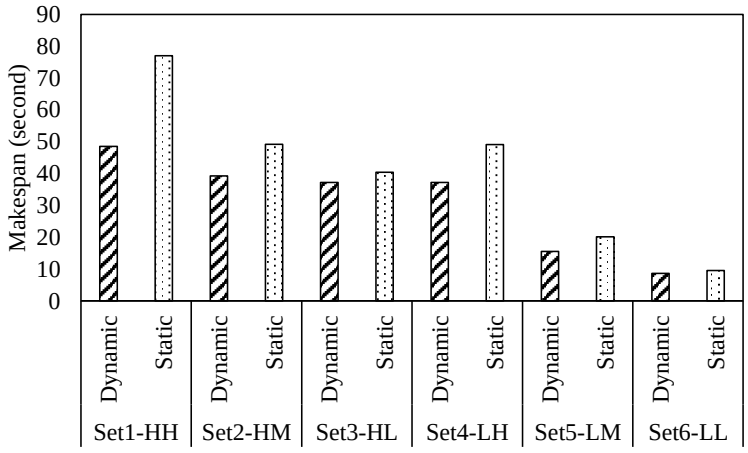


Fig. 13: Makespan of static and SURF dynamic task mapping w/o DSP-dominant background processes

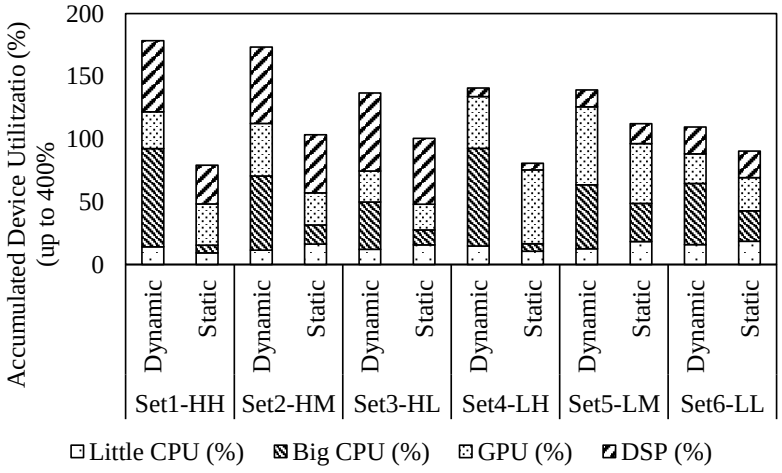


Fig. 14: Device utilization of makespan for each test set w/o DSP-dominant background processes

While these preliminary experimental results demonstrate SURF's efficacy in exploiting underutilized compute units for improving performance, the current policy which applies the HEFT algorithm introduced in Section 4.3 is not power- and energy-aware. As a result, the power and energy consumption increase by 62.8% and 31.6% on average shown in Figure 15. We speculate that the current implementation for the computational kernels make SURF infeasible to deploy energy-aware policy because they are not optimized according to the hardware architecture. Hence, the trade-off between performance and energy becomes trivial - either high performance and energy consumption or low performance and low energy consumption. We expect to see reductions in energy consumption once the kernels are optimized with an energy-aware policy. This development is currently ongoing.

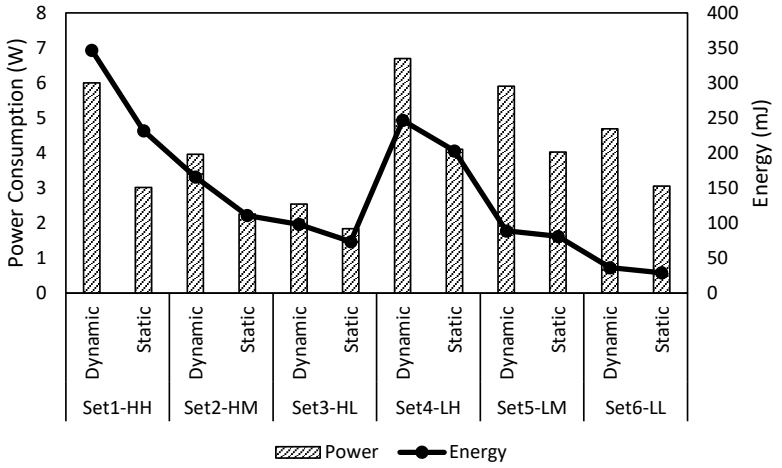


Fig. 15: Power and energy consumption for adaptive HEFT policy in SURF

6 Related Work

Heterogeneous resource management has been widely studied, with a large body of existing work on task scheduling/mapping algorithms [17] [8] [4] [13]. For instance, Topcuoglu et al. [17] proposes the heterogeneous earliest finish time (HEFT) algorithm that schedules tasks in a directed acyclic graph (DAG) onto a device to minimize execution time. Choi et al. [4] estimates the remaining execution time for tasks on CPU and GPU by using a history buffer and selects the most suitable device. The StarPU [2] framework targets high performance computing and enables dynamic scheduling between CPU and GPU based on static knowledge of the tasks. Zhou et al. [19] perform task mapping

onto heterogeneous platforms for fast completion time. Some recent efforts also address domain-specific platforms: Wen et al. [18] and Bolchini et al. [3] propose dynamic task mapping schemes specific for OpenCL; Georgiev et al. [6] proposes a memetic algorithm based task scheduler for mobile sensor workload; and Aldegheri et al. [1] presents a framework allowing multiple programming languages and exploit their different level of parallelism for computer vision applications which achieves better performance and energy consumption.

SURF distinguishes from these works in two directions. First, the SURF framework is composed of a runtime system for task mapping and APIs for mobile systems. SURF is built on top of existing programming interfaces and dynamically profiles task execution and perform task mapping without user-provided static knowledge. Second, SURF is *self-aware*: aware of the heterogeneous hardware architecture, existing scheduling scheme and the runtime system status. It takes care of resource contention of single compute units while other works make assumptions that all the compute unit are exclusive to a single task (e.g., CPU should not be exclusive). The device concurrency of hardware accelerators is also ignored in these previous works.

7 Conclusion

In this article, we presented the architecture of SURF, a self-aware unified runtime framework built on top of existing programming interfaces including OpenMP, OpenCL and Hexagon DSP SDK for mapping tasks onto CPU, GPU, and DSP respectively in mobile SoCs. We illustrated how to use SURF's application interfaces to create and execute a SURF task. SURF performs task mapping while being aware of existing scheduling schemes, intra- and inter-compute-unit contention and heterogeneous hardware architectures to select the compute unit with the earliest finish time for the given tasks without user-provided static information about the tasks. Our early experimental results show an average of 24% speedup by running mixed mobile workloads including two applications, image recognition by using convolution neural networks and an image filter with couple of background processes sharing workload on the compute units. Our ongoing work is incorporating more sophisticated mapping and prediction algorithms, and analyzing the performance as well as energy benefits of deploying SURF on emerging heterogeneous mobile platforms.

References

1. Aldegheri, S., Manzato, S., Bombieri, N.: Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming. In: IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018. pp. 119–124 (2018), https://doi.org/10.1109/VLSI-SoC.2018.8644937
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. : Pract. Exper. 23(2), 187–198 (Feb 2011), http://dx.doi.org/10.1002/cpe.1631

3. Bolchini, C., Cherubin, S., Durelli, G.C., Libutti, S., Miele, A., Santambrogio, M.D.: A runtime controller for opencl applications on heterogeneous system architectures. SIGBED Rev. 15(1), 29–35 (Mar 2018), http://doi.acm.org/10.1145/3199610.3199614
4. Choi, H.J., Son, D.O., Kang, S.G., Kim, J.M., Lee, H.H., Kim, C.H.: An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. The Journal of Supercomputing 65(2), 886–902 (Aug 2013), https://doi.org/10.1007/s11227-013-0870-6
5. Dutt, N.D., Jantsch, A., Sarma, S.: Toward smart embedded systems: A self-aware system-on-chip (soc) perspective. ACM Trans. Embedded Comput. Syst. 15(2), 22:1–22:27 (2016), https://doi.org/10.1145/2872936
6. Georgiev, P., Lane, N.D., Rachuri, K.K., Mascolo, C.: Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In: Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking. pp. 320–333. MobiCom '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2973750.2973777
7. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to gpu codes. In: 2012 Innovative Parallel Computing (InPar). pp. 1–10 (May 2012)
8. Gregg, C., Boyer, M., Hazelwood, K., Skadron, K.: Dynamic heterogeneous scheduling decisions using historical runtime data. In: Workshop on Applications for Multi- and Many-Core Processors (A4MMC) (2011)
9. Gómez-Luna, J., Hajj, I.E., Chang, L., García-Floreszx, V., de Gonzalo, S.G., Jablin, T.B., Peña, A.J., Hwu, W.: Chai: Collaborative heterogeneous applications for integrated-architectures. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2017)
10. Hsieh, C., Sani, A.A., Dutt, N.: The case for exploiting underutilized resources in heterogeneous mobile architectures. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE) (March 2019)
11. Hsieh, C., Sani, A.A., Dutt, N.: Surf: Self-aware unified runtime framework for parallel programs on heterogeneous mobile architectures. In: 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC). pp. 136–141 (Oct 2019)
12. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22Nd ACM International Conference on Multimedia. pp. 675–678. MM '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2647868.2654889
13. Kadjo, D., Ayoub, R., Kishinevsky, M., Gratz, P.V.: A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile platforms. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2015)
14. Kumar, A.: Multiprocessing with the completely fair scheduler (2008)
15. Qualcomm: Snapdrgon profiler (2013), https://developer.qualcomm.com/software/hexagon-dsp-sdk
16. Qualcomm: Hexagon dsp sdk (2017), https://developer.qualcomm.com/software/snapdragon-profiler
17. Topcuoglu, H., Hariri, S., Min-You Wu: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (March 2002)

18. Wen, Y., Wang, Z., O'Boyle, M.F.P.: Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In: 2014 21st International Conference on High Performance Computing (HiPC). pp. 1–10 (Dec 2014)
19. Zhou, H., Liu, C.: Task mapping in heterogeneous embedded systems for fast completion time. In: 2014 International Conference on Embedded Software (EMSOFT). pp. 1–10 (Oct 2014)