Heterogeneous Mobile Platform Characterization and Accelerator Design

by

Cao Gao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2017

Cao Gao
caogao@umich.edu
ORCID iD: 0000-0001-6858-1655

To my family.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor Professor Trevor Mudge. It was very fortunate for me to have him as an advisor who taught me so much on so many things, both about research and life. If I have to pick one of his many advises that had influenced me, it would be to focus the limited time on topics that are really important and meaningful. In addition, his laid back style and connections allowed me to explore a wide range of research projects and collaborate with different people. I am also really grateful for his support, guidance and patience during those hard times in my PhD. I also would like to thank my advisor Professor Ronald Dreslinski. From the first the day I joined the TRON (Trev + Ron) group, he has been very helpful in getting me started on research topics, designing experiments, organizing results and stories, and many other things. Those nights of paper editing together in CSE are very memorable to me. This thesis would not be possible with them.

I would like to thank several other faculty members that I have worked with during my PhD. It was a great experience collaborating with Professor Jason Mars and Professor Lingjia Tang. Their enthusiasm towards research was always an inspiration for me. In addition, it was Jason who introduced me to Professor David Blaauw and Professor Hun-Seok Kim, which led to a great collaboration during the last one and half years of my PhD. I learned great deal from David and Hun-Seok from our inter-group weekly meetings with Trev and Ron. Professor Carole-Jean Wu, Professor Valeria Bertacco, and Professor Todd Austin have all helped me with their valuable suggestions and feedback.

To my lab-mates in TRON lab: Yajing Chen (and Shengshuo Lv), Nilmini Aberaytne, Yiping Kang, Johann Hauswald, Tony Gutierrez, Joseph Pusdesris, Kuangyuan Chen, Byoungchan Oh, Jonathan Beaumont, Dong-Heyon Park, and Qi (Chi-master) Zheng. Knowing them is the best treasure I found here at Michigan. Not only did I get meaningful and insightful discussion about research, but also antidotes about all the other countries and cultures, continuous encouragement and support, and most importantly, a lot of fun time together. I would also like to thank Chris Emmons and Dam Sunwoo, who helped me had a good summer internship at ARM research.

Outside my group, there are many friends who helped me enjoy my life in Ann Arbor, for which I am very grateful. I would like to thank all my friends in the CSE department, especially Chang-Hong Hsu and Doowon Lee, for exchanging interesting information and ideas on research and graduate student life in general. I also am grateful to my Chinese friends outside the department, especially Xiang Yin and Zhaojian Li, for all the fun times together. It was wonderful to have them here far away from our home country.

Finally, I would like to thank my family. My mother have always been supporting and encouraging me these years, and special thanks to her. Together with my late father, they shaped me to become the person I am today. My extended family has been very supportive to me as well, and I cannot thank them enough.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Due to power constraints, growing sections of a chip need to remain passive. This is often referred to as the "dark silicon" problem. This problem is more severe on mobile platforms, especially on emerging IoT and wearable devices, due to their strict power envelope and energy budget. Heterogeneous architecture with specialized hardware is an effective way to utilize the limited active transistor count. To guide heterogeneous mobile platform design, analyzing current and near-future mobile workloads and building specialized hardware based on the analysis is essential. To this end, this dissertation starts with a quantitative analysis of current mobile applications on smartphones, followed by an accelerator-based solution for efficient execution of wearable workloads, with an emphasis on machine learning and signal processing kernels.

The first part of the thesis focuses on a study of mobile device utilization, where the results argue against using many (more than 4) cores in mobile devices, and suggest a flexible heterogeneous architecture to accommodate utilization variations. To meet the increasing computational need of mobile devices, many vendors try to increase the number of CPU cores in mobile device SoCs. However, it does not translate proportionally into performance gain and power reduction. This work analyzes the behavior of a wide range of commonly used mobile applications, using *Thread Level Parallelism* (TLP) as a major metric. The results demonstrate that mobile applications have TLP less than 2 on average, and we observe a small return on utilization when the number of cores is increased. Further analysis shows that TLP tends to appear in peaks and valleys, which gives opportunity for using flexible, heterogeneous mobile architecture for better energy efficiency.

The second part of the thesis consists of a low-power accelerator design for always-on applications used in wearable devices. A large class of these always-on applications, including many deep learning and signal processing applications, execute in a deterministic and repeatable fashion. This design takes advantage of this determinism by replacing the traditional cache based architecture with a non-uniform scratchpad architecture (NUSA). A framework is developed, which determines the most energy efficient NUSA scratchpad design within area constraints, and identifies the most energy efficient data assignment and runtime schedule given the target application. A fabricated prototype accelerator is also presented as an illustration of the technique. On average, a $2.1\times$ reduction in energy can be achieved by using a NUSA architecture compared to a uniform scratchpad architecture, and a $10\times$ to $36\times$ reduction compared to general purpose ARM M-class cores and DSP.

CHAPTER 1

Introduction

1.1 Heterogeneous Mobile Platforms

The scaling of CMOS technology had been enabling faster switching and lower power consumption for the past decades. The traditional Dennard scaling [5] states that as transistors get smaller, their power density stays constant, so the power consumption stays in proportion with area. Therefore, as we shrink the transistors down in size in conjunction to Moore's law [6], we would proportionally achieve lower power consumption per transistor, maintaining the overall power density.

However, Dennard scaling is reaching its physical limits to an extent that voltage cannot be scaled down as much as transistor gate length. Along with a rise in leakage current, this results in increased power density, rather than a constant power density. To ensure a safe operation, it is essential for the chip to perform within a fixed power budget [7], usually called the thermal design power (TDP) constraint. Increased power density under the same power budget means in order to avoid too high power dissipation, a certain part of computation, communication, and memory resources of an on-chip system cannot simultaneously be powered-on (at the peak performance level), and thus must stay "dark" [8, 9, 10]. The problem, therefore, is how to best utilize the abundance of transistors in the dark silicon era to continue to reap the rewards of Moore's Law down to the sub-10 nm node [11].

Mobile platforms with their even more strict power envelope and energy budget, suffers

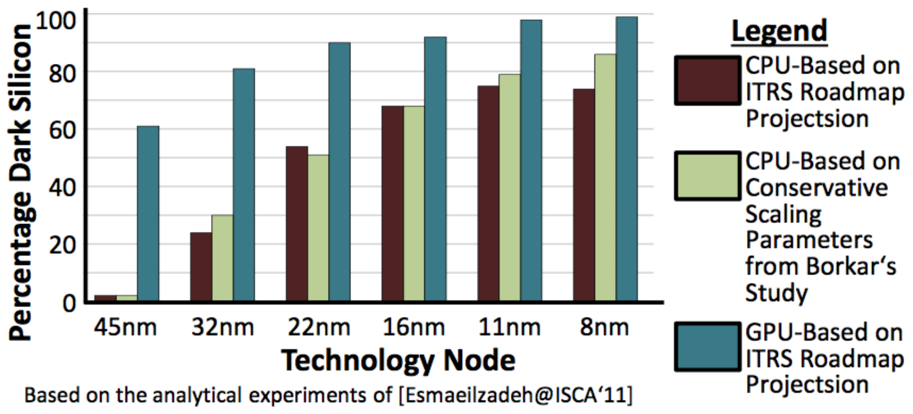


Figure 1.1: Dark silicon trends for different technology nodes [1]

more from the dark-silicon problem. For instance, the TDP for a mobile SoC is 2.5 to 3W for smartphones, 5W for tablets [12], while the TDP for desktop CPU alone is around 100W [13].

An effective way to utilize the limited active transistor count is to user heterogeneous architecture with specialized hardware. In dark silicon era, there is a choice about which parts of the chip should be powered on, so optimal performance can be achieved under peak power and temperature constraints. Specialized hardware usually have much superior performance and energy efficiency than general purpose cores, which can be powered on when executing its target task.

1.2 Contributions

In order to guide heterogeneous mobile platform design, this thesis first analyzes current mobile applications on a smartphone platform. Mobile devices are becoming more powerful and versatile than ever, calling for better embedded processors. Following the trend in desktop CPUs, microprocessor vendors are trying to meet such needs by increasing the number of cores in mobile device SoCs. However, increasing the number does not translate proportionally into performance gain and power reduction. In the past, studies have shown that there exists little parallelism to be exploited by a multi-core processor in desktop plat-

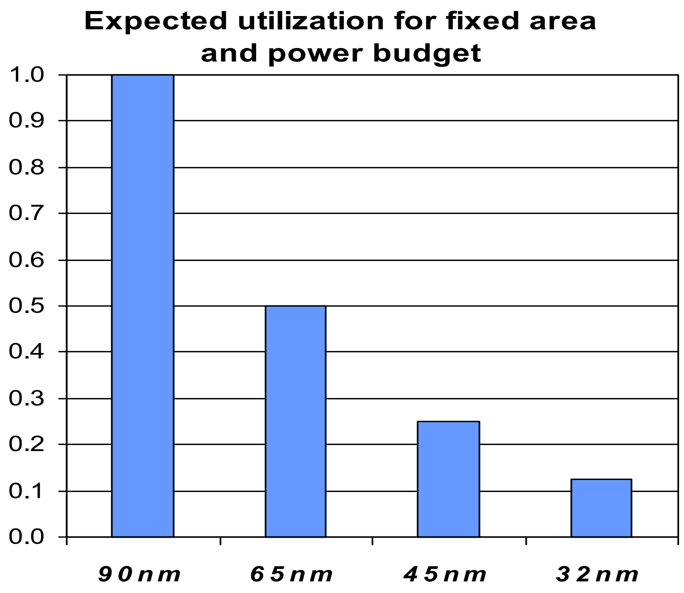


Figure 1.2: Utilization wall [2]

form applications, and many cores sit idle during runtime. We investigate whether the same is true for current mobile applications, and ask the question if quad- and octa- core designs are beneficial. We analyze the behavior of a wide range of commonly used mobile applications. We measure their *Thread Level Parallelism* (TLP), which is the machine utilization over the non-idle runtime. Our results demonstrate that mobile applications have TLP less than 2 on average, and we observe a small return on utilization when the number of cores is increased. Test on high load scenarios suggest that this is due to the nature of mobile devices and utilization cannot be increased easily. However, further analysis shows that TLP tends to appear in peaks, which gives opportunity for multi-core devices to gain better power efficiency. In all, we suggest that increasing cores aggressively may not be the right choice, and looking into the interactive nature of mobile applications can a good direction.

Next, this thesis discuss a design of an ultra low-power accelerator for always-on applications in wearable devices. These applications, such as keyword detection or heart rate monitoring, pose a significant challenge in energy-efficient design. A large class of always-on applications execute in a deterministic and repeatable fashion. Determinism and repeatability mean that the optimal memory access pattern can be pre-computed statically, removing the need for a cache and providing an opportunity to tailor the memory layout directly to the application. We present a non-uniform scratchpad architecture (NUSA) accelerator designed for this class of applications. To fully utilized the proposed NUSA, we develops a framework, which when given the target applications can: a) determines the most energy efficient NUSA scratchpad design within area constraints; and b) determines the most energy efficient data arrangement and runtime schedule given the target application. A fabricated prototype accelerator is presented as an illustration of our technique. We then generalize this prototype into an architecture that is used to evaluate a wider range of applications, and compare against traditional approaches. We show that, on average, a $2.1\times$ reduction in energy can be achieved by using a NUSA architecture compared to a uniform scratchpad architecture, and a $30\times$ reduction compared to a general purpose ARM M-class core.

In summary, this dissertation makes the following contributions:

- We construct a suite containing representative Android applications from a variety of categories, as well as their corresponding test actions.
- We measure the Thread Level Parallelism (TLP) of mobile applications on current mobile device platforms and show it is less than 2 on average.
- We observe diminishing returns of TLP when increasing the number of cores. Heavy-load test cases also show low TLPs, which suggests there is not a lack of hardware resources. Both demonstrate that having many powerful cores is over-provisioning.
- We make the case for the need of a flexible system that can accommodate both high performance and good energy-efficiency for different program phases.
- We suggest that a heterogeneous system is an adequate and energy-efficient solution for mobile devices.

- Identify the opportunity of utilizing the deterministic behavior of always-on applications in wearable devices, and the benefit of utilizing a non-uniform scratchpad architecture (NUSA).
- Design a framework which can generate a NUSA design for a wide set of applications, as well as tailor and map the applications to the target NUSA for the best energy-efficiency.
- Design a low power programmable accelerator for always-on wearable applications, which has been fabricated as a prototype chip.
- Evaluate the framework by generalizing the accelerator design and comparing to general purpose core baselines, as well as cache and USA based accelerators.

1.3 Organization

This dissertation is organized as follows. In Chapter 2, background information about mobile and wearable devices is provided. Chapter 3 discuss an analysis of mobile device utilization. It demonstrates diminishing returns of TLP with increasing the number of cores, motivates a heterogeneous design for energy efficiency and accommodating Chapter 4 introduce an low-power accelerator design for always-on application in wearable devices. Finally, Chapter 5 provides concluding remarks and future research directions.

CHAPTER 2

Background

This chapter outlines some of the background information regarding mobile platform and common applications. First, a discussion regarding the utilization of mobile devices and are discussed. Next, some common applications. Specifically, applications on deep learning algorithm are introduced.

2.1 Mobile Platforms

2.1.1 Mobile Device Utilization

Quite a number of mobile processors are made with four or even more cores today, which offer impressive computational potential to users. However, there are still high-end mobile SoCs that include only a dual-core CPU, and that still successfully provides desirable performance for its host mobile device. In order to make design choices, it is beneficial to analyze how much of the quad-core potential is being utilized.

We perform two preliminary experiments on an Origen board, a current mobile device platform with a quad-core 1.4GHz ARM Cortex-A9 CPU. First, we measure how many cores are actually activated by the OS when running an application. The Android system employs a CPU governor which turns individual cores on or off and changes their frequencies based on CPU loads. When it finds that a core sits idle for most of the time, it will turn it off to save power. We run a suite of commonly used applications with the default

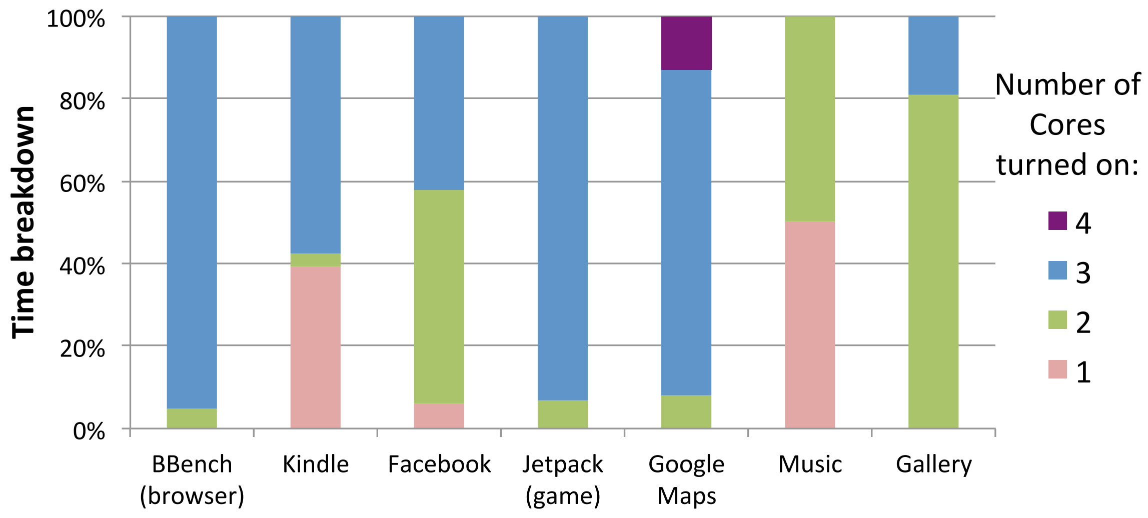


Figure 2.1: Time breakdown of number of cores activated by the Android OS

governor—*ondemand*. We find that the fourth core is only activated in ***2*** out of the 21 apps we tested. For nearly half of the apps, the OS also shuts off the third core for most of the time. We show part of our results in Fig. 2.1. We plot a breakdown of the time percentage that each system configuration spends for these applications. Different colors represent system configurations with different numbers of cores activated. For most of the applications, the fourth core is always shut down, For some of them, namely Email, Facebook, Music and Gallery in this graph, most of the time the third core is not activated as well. The only app in this graph that activates the fourth core is Google Maps. Browser and Jetpack (a game) do activate three cores for most of the time, but Facebook does so only for half of the time, and Email never. Note that activation does not mean utilization; it only means the OS thinks that this core *might* be utilized. The core could still sit activated and idle at the same time. We will show the core utilization in the results section.

In the second experiment, we override system setup and manually set the number of cores activated in the system. Since web-browsing is among the most commonly used features on mobile devices [14], we run a browser benchmark, BBench [15] on two browsers, and compare the performance of different CPU configurations. We plot the results in Fig. 2.2. The score is the time taken to render the complete set of webpages in BBench,

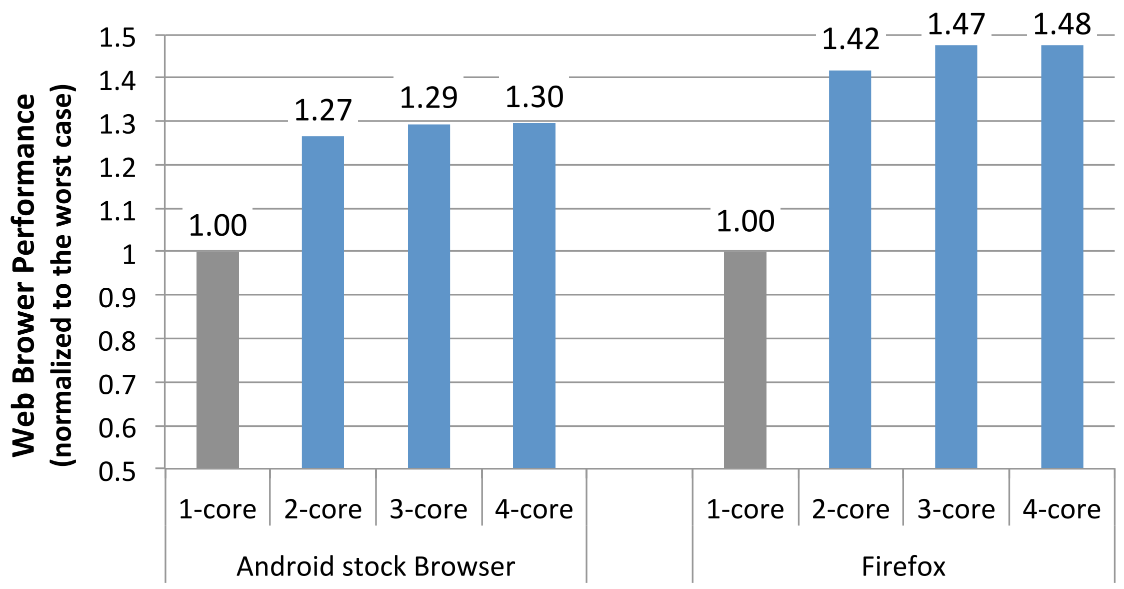


Figure 2.2: BBench score for different number of cores.

and lower score means better performance. It is clear from the graph that a single-core system suffers from poor performance. However, the performance gain is negligible from dual-core to triple-core and quad-core. It may seem confusing why the OS activates 3 cores for BBench when there is such little performance improvement. Actually it again proves that activation does not mean utilization; the *ondemand* CPU governor in Android OS is performance-aware and will keep the core activated unless it is very unlikely to be utilized [16].

Both of these tests suggest a more thorough quantitative investigation of quad-core CPU utilization.

2.1.1.1 Metrics

To evaluate the utilization of a multi-core system, we need a good metric for system profiling. A commonly used and accessible metric would be CPU utilization, which is simply the overall average CPU usage during runtime. However, it would underestimate the parallelism in mobile applications. Most of these apps are interactive, and there is a large portion of idle time for interactive applications, as shown in Fig. 2.3. The program itself

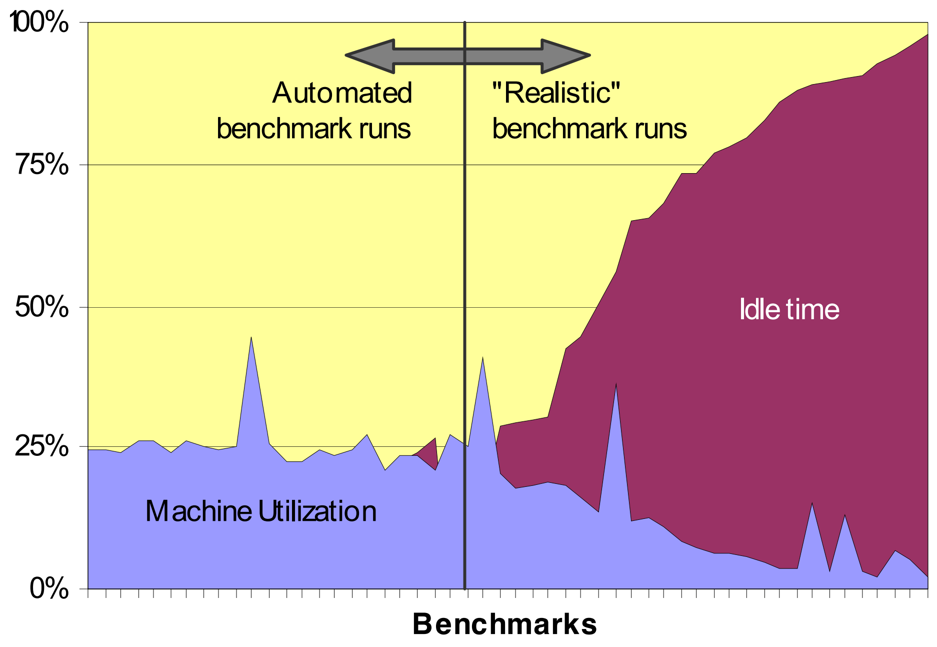


Figure 2.3: Figure qualitatively describing the utilization differences between automated desktop workloads and "realistic" runs performed by a user [3].

could be well parallelized and have a high machine utilization number during busy time. However, it sits on idle and waits for user input for most of the total running time, which drags down the average utilization number. To avoid that, we use *Thread Level Parallelism (TLP)* [17, 3]. TLP is defined as the machine utilization over the non-idle portions of the benchmark's execution. The formula for TLP is given by Equation. 2.1:

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{2.1}$$

where c_i is the fraction of time that *i* cores are concurrently running different threads, and n is the number of cores. Specifically, c_0 represents idle time fraction, which is excluded because it does not count towards the program's parallelism. Note that TLP is not a performance metric; the software could still spawn threads that do not perform useful work. Nevertheless, it is the natural metric to measure multi-core utilization, especially for interactive applications like the ones on a smartphone. The TLP serves as a good indicator of the number of processors needed to support the execution of a parallelized workload.

2.1.1.2 Early Studies on TLP

Flautner et al. [3] proposed the definition of TLP in 2000. At that time, multi-core was mostly exploited in research labs and appeared only in workstations and servers. They performed a study of TLP on desktop applications and found that a dual-core system improves the responsiveness of interactive programs. However, they also showed that desktop applications leveraged TLP very sparingly. This result was echoed 10 years later by Blake et al. [17] with a similar study of TLP of contemporary software and hardware, when multicore had become the norm rather than the exception in home and office desktops. They reported that 2-3 cores were more than adequate for almost all but a few domain specific applications like Video Authoring. After observing low single-thread performance could have a small impact on the TLP, they claimed that software is lagging behind and is the main limiting factor in TLP.

Smartphones were already becoming popular during the time when Blake et al. presented their results, and they have continued to supplant desktops for many applications. To reflect this it is important to analyze TLP behavior on mobile devices because the original studies did not. Besides exploring a different hardware platform, we are also using a slightly different set of benchmarks from the original work. Some categories of desktop applications are rarely seen on mobile devices, such as Video Authoring and professional Image Authoring. We also make some investigations into frequency scaling, which is especially interesting for smartphones and tablets due to their tight power budget.

2.1.2 The status quo for wearables

Battery life is still a major constraint for current wearable devices (Table. 4.1.) One typical method of reducing power consumption in smartphones is to reduce the brightness or resolution of the screen, which takes up a large portion of the total power consumption. However, the display in wearable devices already consumes a much lower portion of total energy than those in a smartphone. It can be as much as 20-40$\times$ fewer pixels and has only a relatively simple color scheme [18, 19]. Another major power consumer in wearables is the wireless connectivity module, which is usually implemented in very specific communication ASICs and is difficult to further optimize [20]. Therefore, optimizing the power consumption of the compute stack becomes critical.

2.2 Wearable Applications

In this section we discuss some of the common mobile applications, mostly used in Chapter 4. First, we introduce some recent deep learning algorithms used in near-future applications, namely Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN). Then, we introduce some common always-on applications used in wearable devices.

2.2.1 Deep Learning Algorithms

Neural networks have been around for many decades[21, 22]. However, until 2006, deep, fully connected neural networks were commonly outperformed by shallow architectures that used feature engineering[23]. Three additional reasons have recently helped deep architectures obtain state of the art performance: large datasets, faster, parallel computers and new machine learning insights. In this time of "big data", huge datasets can be collected rather easily and cheaply by institutions, which can be used to train deep models with many parameters. Improvement of multi-core CPU and GPU computing architectures gives opportunity to execute a large number of matrix multiplications, which are required by these algorithms. All these leads to an increase of application using deep learning algorithms.

2.2.1.1 General Structures

Even though Deep and Convolutional Neural Networks come in various forms, they share enough properties that a generic formulation can be defined. In general, these algorithms are made of a (possibly large) number of layers; these layers are executed in sequence so

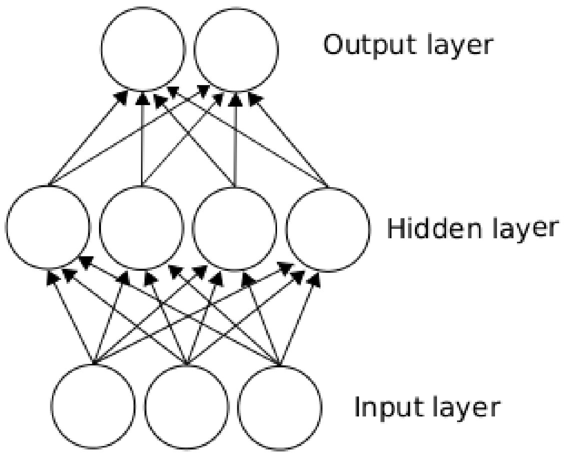


Figure 2.4: Multilayer Perceptron

they can be considered (and optimized) independently.

2.2.1.2 MLP (Multilayer Perceptron)

The Multilayer perceptron (MLP) is a neural network consisting of multiple mutually interconnected layers of neurons. Every neuron in one layer is connected to every neuron in the following layer. A non-linear function is applied to the neurons output of each layer.

2.2.1.3 CNN (Convolutional Neural Network)

Convolutional neural networks (CNNs) are mostly designed to recognize features in 2-dimensional image data, but are used for various other purposes as well. CNNs are primarily used for 2D image recognition, so we will illustrate their architecture on a 2D rectangular image consisting of pixels. Each pixel generally carries colour information. Colour can be represented by multiple channels (e.g. 3 RGB channels). For the sake of simplicity, we will consider only one single channel (shades of gray) while explaining the model. The neurons in CNNs work by considering a small portion of the image, let us call it subimage. The subimages are then inspected for features that can be recognized by the network. As a simple example, a feature may be a verticalline, an arch, or a circle. These features are then captured by the respective feature maps of the network. A combination of features is

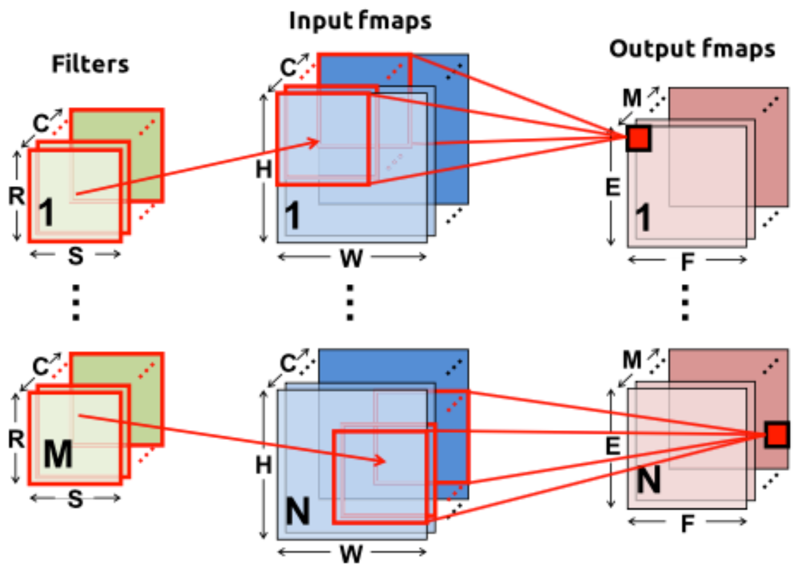


Figure 2.5: Convolutional Neural Network

then used to classify the image. Furthermore, multiple different feature maps are used to make the network robust to varying levels of contrast, brightness, colour saturation levels, noise, etc.

2.2.2 Always-on applications for wearables

We examine several example categories of always-on applications, which are summarized in the following subsections.

2.2.2.1 Keyword spotting

Keyword spotting is a detection task to identify the presence of specific spoken words in a stream of speech signals. It is often used to trigger automatic speech recognition and spoken dialog systems. It is common in wearable devices when hands-free activation is desired. Examples include detecting ”OK Google” when wearing a Google Glass or ”Hey Siri” when wearing an Apple watch. Keyword spotting can be implemented using classification techniques, such as hidden Markov models (HMM) [27] to identify different keywords. More recently machine learning algorithms based on Deep Neural Networks [28] and Recurrent Neural Networks [29] are being deployed because their better detection accuracy.

2.2.2.2 Seizure detection

Epileptic seizure detection refers to the use of algorithms to recognize the occurrence of a seizure. Typically these algorithms are based on the analysis of biological signals from a patient with epilepsy. They can be deployed in smartwatches or health wristbands for quick, always-on detection. The algorithms are typically based on a nearest-neighbor classifier of EEG features [30, 31], or the RNN technology mentioned above [32, 33].

2.2.2.3 Face detection

Face detection is the detection of a face in a scene. It is commonly used as a front-end to trigger facial recognition. It appears in Google Glass and smart home detection devices. Due to its wide range of application, face detection algorithms have been studied heavily. Recent implementations include DNNs [34] or Convolution Neural Networks (CNN) [35, 36]. The former is easier in terms of memory access patterns but the latter is usually more suitable for image processing.

2.2.2.4 Wake on user gesture

Wake-on user gesture is the feature of some wearable devices which can be activated using certain user-specified gestures. This enables the device to recognize who is interacting. For an entertainment device, it can recognize the user and load the right game profile or music play list. For a home climate control, it can adjust the environment to the wearer's preference. This feature can be implemented using one-vs-all classification [37] or margin classifiers [38].

CHAPTER 3

An Analysis of Mobile Device Utilization

Mobile devices are becoming more powerful and versatile than ever, calling for better embedded processors. Following the trend in desktop CPUs, microprocessor vendors are trying to meet such needs by increasing the number of cores in mobile device SoCs. However, increasing the number does not translate proportionally into performance gain and power reduction. In the past, studies have shown that there exists little parallelism to be exploited by a multi-core processor in desktop platform applications, and many cores sit idle during runtime. In this chapter, we investigate whether the same is true for current mobile applications, and ask the question if quad- and octa- core designs are beneficial.

We analyze the behavior of a wide range of commonly used mobile applications. We measure their *Thread Level Parallelism* (TLP), which is the machine utilization over the non-idle runtime. Our results demonstrate that mobile applications have TLP less than 2 on average, and we observe a small return on utilization when the number of cores is increased. Test on high load scenarios suggest that this is due to the nature of mobile devices and utilization cannot be increased easily. However, further analysis shows that TLP tends to appear in peaks, which gives opportunity for multi-core devices to gain better power efficiency. In all, we suggest that increasing cores aggressively may not be the right choice, and looking into the interactive nature of mobile applications can a good direction.

3.1 Introduction

Nowadays, mobile devices are gradually taking over the functions of traditional desktop applications. High-definition video playback, interactive games and web browsing are commonly supported by the latest smartphones and tablets. These performance-intensive tasks need powerful hardware support, which drives microprocessor vendors to continuously produce better mobile CPUs. Given the strict power budget of mobile devices, vendors reached the limits of frequency scaling quickly and turned to multi-processors. The first dual-core smartphones, such as Galaxy S II and HTC Sensation, came to market in 2011. Most of the high-end smartphones released in 2012 were dual-core or quad-core; in April 2013, the Samsung Galaxy S4 was released with the *Exynos 5 Octa*, which uses ARM's big.LITTLE architecture and has a total of eight cores. Mediatek shipped their octa-core SoC in late 2013, and Qualcomm announced their octa-core CPU with eight A53s in early 2014 as well.

However, some recent smartphones are still equipped with dual cores. Apple's new A8 Chip for the iPhone 6, released in September 2014, uses a dual-core CPU and still provides satisfactory performance. This leads to the question: How much of the computation potential residing in multi-core CPUs is actually being utilized? On the desktop end, Blake et al. [17] did a study on *Thread Level Parallelism* (TLP) on a suite of representative desktop applications. Their work was to measure the core utilization in modern multi-core CPUs, and they suggested that the number of cores that can be profitably used is less than 3 for most commonly used applications. It is possible that mobile device applications have similar characteristics and cannot effectively utilize a quad-core CPU, let alone hexa- and octa-core. Moreover, the GPU, DSPs, and ASICs in these systems already exploit much of the parallelism, leaving little for the CPU.

To make some observations about the benefit of multi-core, we performed two preliminary experiments on an up-to-date quad-core mobile device platform. First, we measured how many cores are actually activated by the Android OS when running an application.

We found that the fourth core was only activated in 2 out of 21 apps, and the OS also shut off the third core for nearly half of the apps. Note that activation does not mean the core is in use; it only means the OS thinks that the core *might* be used. Second, we overrode system setup and manually set the number of activated cores in the system. Then we ran a browser benchmark [15]; we saw a significant performance improvement from single-core to dual-core, but negligible improvements from dual-core to triple- and quad-core. Both of these results show rather modest gains from high numbers of cores (here more than 2). In all, to measure how much parallelism actually exists is helpful to: a) inform vendors and prevent them from over-provisioning hardware that cannot be effectively used, b) highlight the need to find more parallelism, c) provide suggestions for a better design.

In this work, we analyze a broad range of popular mobile applications to determine how the growing number of cores are utilized. We measure the Thread Level Parallelism (TLP) of these applications. The results show that mobile apps are utilizing less than 2 cores on average, which means multiple cores are used rather infrequently. A small TLP scalability is observed for most applications, and increasing the number of cores has diminishing return on TLP. Even in heavy-load real-world scenarios with background applications or multi-tab browsing, there is still not enough work to keep utilization high. Due to the physical constraint and interactive user pattern, mobile applications tend to have less parallelism to exploit than desktop applications. The GPU and mobile co-processors on chip also reduce CPU load. All these factors, and the history of the slow pace of exploiting parallelism in desktop and mobile software environments [17, 40], indicate that having many powerful cores is over-provisioning. Further analysis suggests that current mobile applications can benefit from a system with the flexbility to satisfy high performance and good energy-efficiency for different application phases. We find that TLP behavior exhibits short peaks and long valleys rather than remaining constant. Peaks require high performance, but not necessary good energy-efficiency because these peaks are usually short, meaning that power has less affect on overall energy consumption. Valleys, on the other hand, desire better energy-efficiency because they do not require high performance but usually dominate the application execution. There is also a number of other research opportunities that arise, such as building accelerators or customized hardware to further reduce the thread's TLP peaks with better energy efficiency, or building better OS infrastructure to utilize mobile heterogeneous systems.

To summarize, we make the following contributions:

- We construct a suite containing representative Android applications from a variety of categories, as well as their corresponding test actions.
- We measure the Thread Level Parallelism (TLP) of mobile applications on current mobile device platforms and show it is less than 2 on average.
- We observe diminishing returns of TLP when increasing the number of cores. Heavy-load test cases also show low TLPs, which suggests there is not a lack of hardware resources. Both demonstrate that having many powerful cores is over-provisioning.
- We make the case for the need of a flexible system that can accommodate both high performance and good energy-efficiency for different program phases.

The rest of the chapter is organized as follows: In Section 3.2 we describe the system setup, measurement method, and benchmarks used. Section 3.3 presents and analyzes the results. We discuss the related works in Section 3.5 and conclude the chapter in Section 3.6.

3.2 Methodology

3.2.1 System Setup

We use the Odroid XU+E board [41]. It has a Samsung Exynos 5410 SoC, which contains an ARM big.LITTLE octa core of four 1.6GHz A15s and four 1.2GHz A7s. Each core has its own 32KB/32KB L1 instruction and data cache; the four A15s share a 2MB L2 cache and the A7s share a 512KB L2 cache. Either four A15s or four A7s can be enabled at the same time, but not a mixture of them. The Odroid board has a PowerVR tri-core GPU running at 480MHz and with 2GB main memory. It also has an on-board current/power semiconductor sensor which measures the current/power consumption of CPUs, GPU and memory separately[1]. We run Android version 4.4.2 (Kitkat) and Linux kernel version 3.4.5. We choose the use the ART runtime instead of the older Dalvik. A web-cam is connected as smartphone camera.

3.2.2 Measurement

3.2.2.1 TLP

To get the TLP number, we track all the context switches that happen in the system, which reveals the information about the status of each core. For instance, a context switch from *SurfaceFlinger* to *swapper* on Core #0 indicates this core has turned from busy to idle. This information gives us the number of running cores at any time, which is sufficient to calculate TLP. Moreover, we can get information about which thread is running and filter out observation overhead threads. For example, we treat *adbd*, the Android Debug Bridge thread, as *swapper*. The core that is running *adbd* would then be treated as idle, preventing an overestimation of TLP. We use *ftrace* [42], a Linux kernel internal trace, to get context switches. The data we gathered contains task names, ids, CPU Number, and timestamp.

3.2.2.2 GPU utilization

For the PowerVR on the Odroid board, we directly read GPU utilization numbers from the sysfs interface provided. For the Mali GPU on the Origen board, we directly use a function from Mali GPU driver which monitors GPU utilization. This function is originally used for GPU dynamic voltage and frequency scaling.

[1]For CPU power, we measure the sum of power of big and little clusters.

3.2.3 Benchmarks

In this work we test a diverse range of real-world Android applications. We prefer applications that are: a) most widely used by users; b) from a broad range of diversified categories. Applications are also categorized in the Play Store, which make us easy to find applications in different categories. We choose not to use existing Android benchmark applications; the reason is that they mostly focus more on CPU or GPU performances and do not reflect the real scenarios. The tests are usually very CPU-intensive in order to test its peak performance; however, in real application this happens rather infrequently. We will show relevant result in Section. 3.3.1. Based on these requirements, we choose 18 top-pick applications from the Google Play Android App store, and 4 native ones in the Android OS. Most Android users download and install their applications using the Android Play Store, therefore its scores and ranking can be used as an credible indication of the popularity of Android applications. This means they are the applications commonly used in their category and are thus representative of current mobile software. They come from 10 different categories: browser, video player, music player, image viewer, communication, games, social networking, navigation, office, and file browser. They make use of important hardware resources on a mobile device (CPU, GPU, co-processors, etc).

We then perform test actions on each of the testing applications. Three applications (browser, Adobe reader and MX Player) are so widely used that they have already been included in some benchmarks [15, 43, 44], therefore we leverage the existing work and use their implementation directly. For other applications, we design a series of actions that cover most typical functions of the application under test. We also refer to the study in [45] on mobile applications usages, including what the popular applications are and how long each session (from opening to closing) normally last. Test actions on the Odroid board are automated using android adb commands and RERAN, a record and replay tool for Android OS [46]. These actions usually last for 30 seconds, and covers most typical functions of the application under test. We found 30 seconds is long and effective enough to cover all common actions for the benchmark applications we have chosen. Meanwhile, we keep them as simple as possible to maximize the operator's ability to produce a repeatable input. All experiments are repeated at least 5 times for more accurate results, and applications that require an Internet connection are repeated for at least 10 runs. We observe a low standard deviation of TLP results as shown in Section. 3.3.1.

It is also important to test TLP of scenarios with background applications, to reflect common daily usage. We also test three applications with a set of other applications running in the background. The three applications under test are Angry Birds, Adobe reader, and Chrome, while the background applications are Hangout, Spotify, and Email.

Except for multi-tasking tests, we kill all the running and background applications before testing to reduce experimental errors. We also never test the application in the same category consecutively; for instance, after a test run of Browser (no matter on stock Android Browser or Chrome), we will start testing applications in a new category like Gallery or Fruit Ninja but not testing either browser again immediately. We do this to avoid cache thrashes. In our suite, most of the applications does not require an Internet connection.

We briefly introduce each application, and its corresponding test actions in the following subsections. We summarize our benchmark in Table. 3.1.

3.2.3.1 Web browser

Web browsing has always been one of the most common usages since the introduction of smartphones. We use the Realistic General Web Browsing (R-GWB)[44], an automatic webpage rendering benchmark. It comprises offline pages of several most popular webpages, all of which utilize modern web technology such as CSS, HTML, ash, and multimedia. During the experiment, MobileBench uses JavaScript to load each webpage and then scroll over it with a pre-set speed. By doing so, it simulates an actual web browsing scenario. We run MobileBench on three popular browsers: the Android stock web browser, Firefox, and Chrome. For each test, we iterate through the MobileBench webpage set five

Category	**Application**
Browser (MobileBench)	Stock Browser Firefox Chrome
Video Player	MXPlayer Netflix
Music Player	Stock Music Spotify
Image Viewer	Stock Gallery Instagram
Communication	Skype Google Hangout
Games	Angry Bird Fruit Ninja Jetpack
SNS	Facebook Twitter
Navigation	Google Maps
Office	Adobe reader Stock Email Kindle
File Manager	Dropbox ES File Browser

Table 3.1: Applications

times, and profile the third one. We do not impose any other user input as MobileBench is automatic itself.

3.2.3.2 Video Player

Video playback is another commonly used application for smartphones. Specialized decoders and DSPs have made high-definition video easily available on current mobile devices. We use two applications: MXPlayer, a video playback application and Netflix, an online streaming application. We test both applications by playing a video for 30 seconds, with a 1 second pause at the 15th second of each of the tests.

3.2.3.3 Music Player

People are gradually replacing traditional music players with phones. We use the Android stock music player and Spotify for testing music players. Spotify is a music player that supports online music streamling. We test both of the two apps by running a series of actions including open new song, jump to an arbitrary position of the song (not in spotify), and open another song.

3.2.3.4 Image Viewer

One important function of modern smartphone is taking photos. Image viewers, in turn, become one of the commonly used application categories. We use Android stock image viewer (Gallery) and Instagram for testing image viewers. We test it by opening images, scrolling between images, opening another image and new folders, and playing a slideshow for a couple of seconds. Instagram is mobile photo-sharing service. Users take pictures share them on a variety of social networking platforms. We test it by scrolling new feeds, opening a picture, applying Amaro filter and changing brightness to 75, and then sharing the picture.

3.2.3.5 Communication

We use Google Hangout and Skype here. We test both of these applications by initialing a 1 minute video call, 30 seconds in foreground and 30 seconds in background (approximating an audio call).

3.2.3.6 Games

Mobile Games are becoming extremely popular these days. Many of them are actually very performance and graphic intensive, which makes them a good testing benchmark for CPU and GPU. The three games we choose are Angry Birds, Fruit Ninja, and Jetpack. Angry Bird is a puzzle video game; in the game, players use a slingshot to launch birds at pigs stationed on or within various structures, with the intent of killing all the pigs on the playing field. We play this game by entering the first stage, firing two birds with one miss and one hit. Fruit Ninja is an action game with lots of floating objects and high-frequency user input. It represents a more intensive mobile game comparing to Angry Birds. We play this game in the “zen mode”, where fruit keeps spawning for 90 seconds. During testing, the tester keeps sliding with a constant frequency horizontally in the upper middle of the screen. Jetpack Joyride is a game where the player tries to control the rider to avoid barriers and collect coins. We play this game by tapping the screen at a regular frequency for 45 seconds.

3.2.3.7 Social Networking

We choose the apps from two major Social networking service providers, Facebook and Twitter. When testing Facebook, we scroll feeds, open up the pictures in the feeds and browse the profile of the user. The action of testing Twitter includes clicks on tweets, checking out picture in the tweets and looking at the profile of the tweeter.

3.2.3.8 Navigation

GPS are ubiquitous on mobile devices. People use navigation apps to help them find places or guide routes. The most popular navigation application on Android platform is Google Maps. We test Google Maps by searching directions between airports in New York city: we search driving directions from Newark to JFK, then public transportation from JFK to LaGuardia. We also save an offline map of New York city to avoid fetching map data from the Internet during testing.

3.2.3.9 Office

The office category contains a broad range of commonly used apps. We test the following: 1) Android stock Email — we test it by writing an email and save that as a draft, sending it, checking and downloading new email. 2) Adobe reader — actions here include opening a pdf, zooming in and out, scrolling pages and searching for a keyword. 3) Amazon Kindle — actions here include opening a book, scrolling and jumping to arbitrary positions.

3.2.3.10 File Browser

The increasing storage capacity of mobile devices lead people to use that as a mass storage. File browsers are then helpful in organizing files. We test Dropbox and ES File Browser. For Dropbox, we open and change folders, sort the content in the folder, do searching and editing new text file. For ES File Browser, we open the folders on the SD cards, scroll the images in it, sort and change the view of the folder.

3.2.3.11 Background

The background applications we choose are Google Hangout (video chatting), Spotify (playing music), Email (checking emails)[2]. With those applications we test Fruit Ninja,

[2]During the test we automatically send an email to the account on board from the host machine every half minute

Maps, and Adobe Reader.

3.3 Results

In this section, we present our experimental results and analysis of mobile device utilization, specifically on CPU and GPU. First, we show that current mobile applications have a rather low average TLP on modern mobile device platforms. We show that increasing the number of cores has diminishing returns on TLP. Even some heavy-load real-world scenarios do not use many cores. High GPU utilization also indicates that some of the parallelism is already offloaded from the CPU to the GPU. All these factors, and the history of the slow pace of exploiting parallelism in desktop environments [17], suggests that having many powerful cores is likely to be over-provisioning.

3.3.1 Overall Results

We list a summary of the results in Table. 3.2. Each row in the table shows the TLP and standard deviation (σ) for an application type. The first line, "System", refers to the plain Android OS testing environment without any application running; the last line, "*Average*", is the average of the statistics of all tested applications. The standard deviation of the TLP over runs for each application is low. This indicates the tests are reproducible and insensitive to user input variation.

We make two observations based on these results:

1) All the applications demonstrate some, but quite limited TLP.

For Android, even in a case where a developer writes code with no awareness of multithreading, a number of threads are still created for external I/O, garbage collection, graphics rendering, etc. This means that even a programmer with no idea about multithreading could benefit from parallel processing on different CPU cores. Moreover, many software developers are aware of the multi-core hardware they are using and write applications explicitly

Category	**App**	**TLP**	σ **(TLP)**
System	[None]	1.03	0.00
Browser	Stock Browser	1.47	0.03
	Firefox	1.31	0.02
	Chrome	1.66	0.01
Video Player	MXPlayer	1.34	0.01
	Netflix	1.53	0.07
Music Player	Stock Music	1.29	0.03
	Spotify	1.23	0.05
Image Viewer	Stock Gallery	1.46	0.03
	Instagram	1.31	0.03
Communication	Google Hangout	1.82	0.15
	Skype	1.55	0.13
Games	Angry Birds	1.31	0.08
	Fruit Ninja	1.40	0.12
	Jetpack	1.54	0.09
Social Network	Facebook	1.43	0.04
	Twitter	1.32	0.04
Navigation	Google Maps	1.59	0.06
Office	Stock Email	1.52	0.04
	Adobe Reader	1.30	0.05
	Kindle	1.45	0.01
File Browser	Dropbox	1.33	0.02
	ES file browser	1.22	0.02
Background	Back_Fruit	1.65	0.12
	Back_Maps	1.91	0.11
	Back_Adobe	1.60	0.14
Average		1.46	0.06

Table 3.2: TLP results for the Odroid board — using ondemand governor.

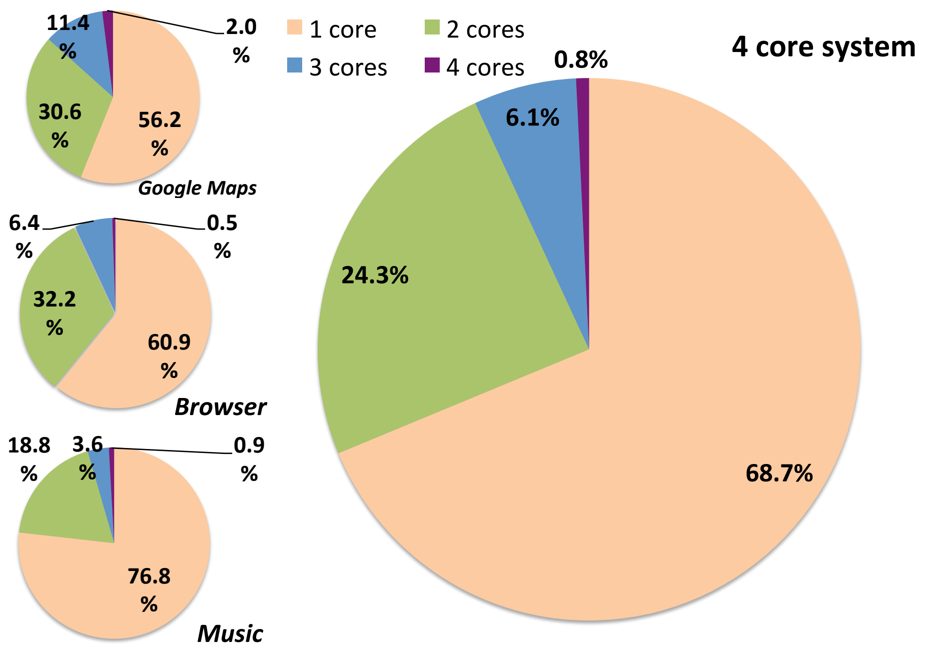


Figure 3.1: Time breakdown of how the multi-core is utilized. The big pie chart on the right shows an average result of all application categories we tested. The smaller pie charts show the breakdown of three representative kinds of apps: Google Maps with the highest TLP, stock Browser, and stock Music with a low TLP.

with multiple threads.

However, the parallelism we observed is generally still quite low. On average, we see a TLP of 1.46. The application with the highest TLP, Google Hangout, has a TLP of just 1.8. Applications like Music and File Browser have rather low TLP, around 1.2 to 1.3. This result shows, on average, the system is using less than 2 cores.

2) Multi-core is utilized infrequently.

We present a time breakdown of how the multi-core system is utilized in Fig. 3.1. The big pie chart on the right shows an average result of all application categories we tested. The smaller pie charts show the breakdown of three representative application categories. We observe a very low 4-core and 3-core utilization; on average only 0.68% of all the non-idle time is the system fully utilized (all four cores running), and 5.81% of the time when three cores are used — this means there is only a small amount of time that four or three cores are being utilized at the same time.

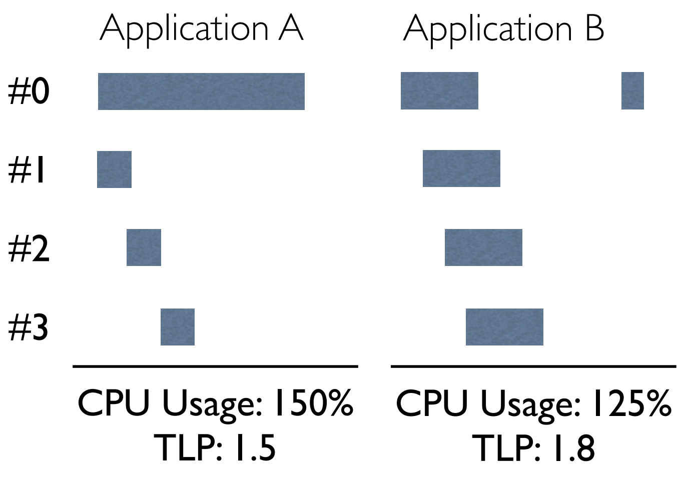


Figure 3.2: Application *B* exploits better parallelism than Application *A*: A uses more CPU time but has less TLP. Note the number in the graph is merely symbolic and do not reflect actual results.

3) Performance-intensive tasks do not necessarily mean high TLP.

Intuitively, applications with high CPU usage should utilize more cores and thus have a high TLP. However, we find several applications have both high TLP and relatively low CPU utilization. For instance, Android native browser has better TLP but less CPU usage than Firefox.

We believe it shows that how software is written has critical affect on TLP. We illustrate a possible scenario in Fig. 3.2. Compared to Application *A* where the single thread on Core #0 takes up much more work than other threads, Application *B* parallelize its work better so it utilizes more benefit from multi-cores. Another pair that shows similar result is Google Maps and Fruit Ninja. Google Maps has a higher TLP of 1.590 than that of Fruit Ninja (1.391) but only has half the CPU utilization. Comparatively, Google Maps has more content that gets parallelized: graphic rendering for map, Internet connection for search and route inquiry, user interface, etc. On the other hand, Fruit Ninja has less threads; we observe one core keeps near 100% usage all the time running one single main thread of Fruit Ninja, while some of the other cores sit idle. This result shows that rather than the inherent function of the application itself, the way software is written also has a very large impact on exploiting the TLP.

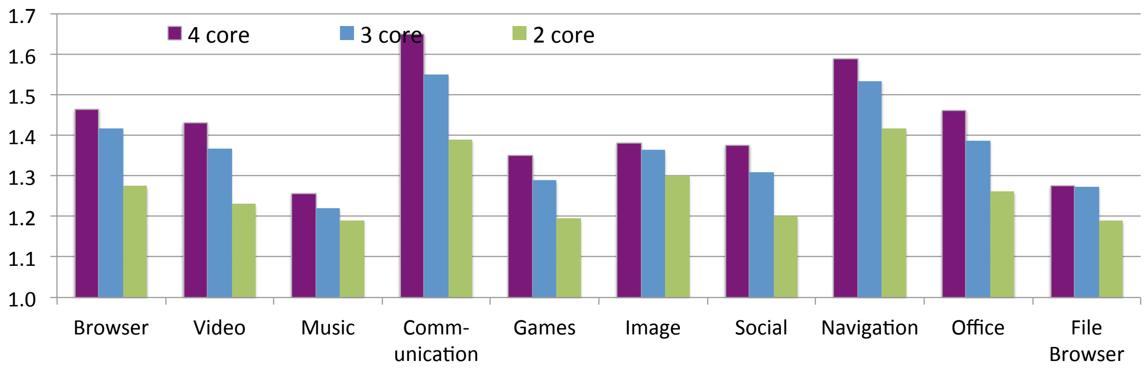


Figure 3.3: Overall TLP result for different number of cores

3.3.2 Core Scaling

Microprocessor vendors put more cores on a chip to exploit more parallelism in the system. Therefore, it is important to consider the change of TLP as the system scales from 2 to 3 to 4 cores.

We show our TLP results in Fig. 3.3 and 3.4. We change the number of active cores in the system and repeat the same experiments for TLP. For each category, the leftmost bar shows the TLP when 4 cores are kept activated. The middle one shows the TLP when the fourth core is shut down and the remaining 3 cores are kept activated. Similarly, the rightmost bar shows the system configuration with 2 cores. The results demonstrate:

1) Increasing the number of cores has little impact on TLP.

On average, TLP increased by 7.03% when we switch from a 2-core system to a 3-core system, and only 3.47% from a 3-core system to a 4-core system.

2) Most applications show some scalability, but not much.

Particularly, Games, Navigation, Office and Social apps show over a 10% increase of TLP from a 2-core to a 4-core system. File manager only shows a 5.6% increase. Most apps show small increases in TLP from 3-core to 4-core which are below 4%. This indicates that the software does not generate many concurrently parallel threads during its execution.

Fig. 3.5 presents the average multi-core utilization for a system with only 3 cores and 2 cores enabled. We observe similar results to that of the 4-core system: the third core is

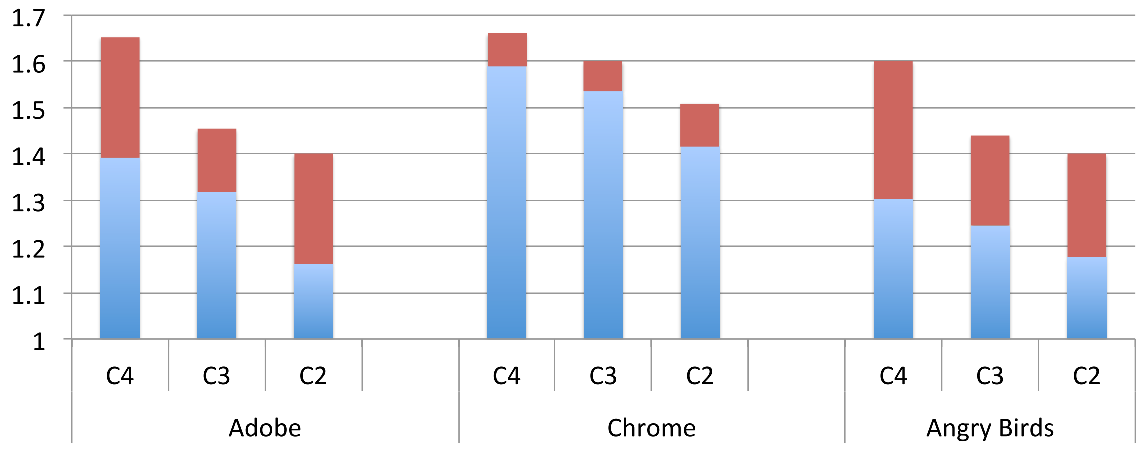


Figure 3.4: Increase of TLP with background apps (shown as red regions)

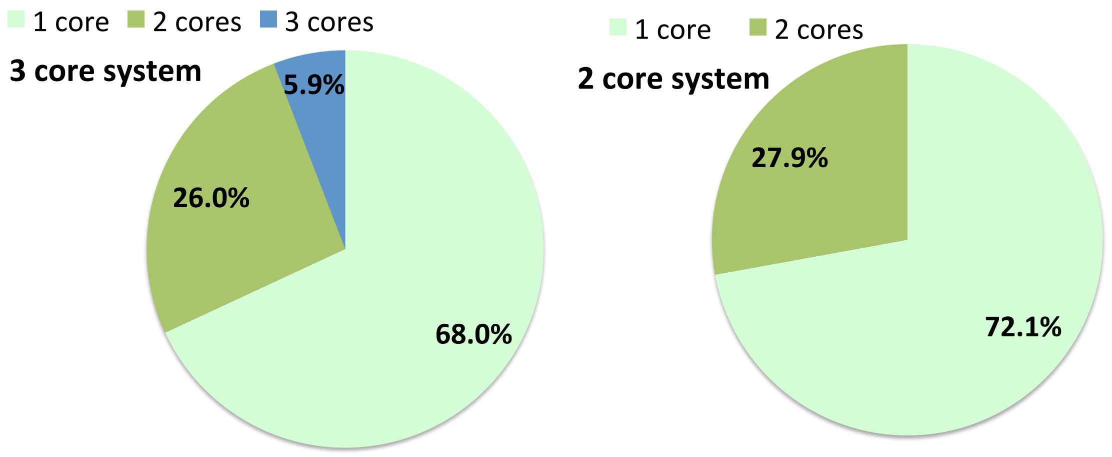


Figure 3.5: CPU breakdown for 3-core and 2-core system. This result is the average of all applications.

utilized very infrequently and the majority of time only the first core is used.

3.3.3 Heavy Load Scenarios

Intuitively, a multi-core system is beneficial when the CPU load is high. In this section we test the TLP of a couple of heavy load scenarios.

3.3.3.1 Background Applications

It is now common to have several applications like music or email checking running in background concurrently with a foreground application. One argument that favors having more cores is that they can boost performance of such scenarios. We measure the TLP of several applications with a set of background applications running concurrently (described in Section. 3.2.3), and present our result in Table .3.2 and Fig. 3.4. The results demonstrate that background applications only lead to limited increase in TLP, and we are still not fully utilizing all four cores with background activities.

3.3.3.2 Multi-tab Web Browsing

We test multi-tab scenarios to see how mobile browser applications exploit parallelism under high load circumstances. We measure the TLP and performance of MobileBench on different CPU configurations. We run one, two and three MobileBench tabs concurrently and measure the average of the metrics. We manually switch between tabs constantly to make them appear in the foreground for similar amount of time. Fig. 3.6, 3.7, 3.8, 3.9 show the results. The maximum TLP is still below 2 for big cores and 2.2 for little cores, and there is significant performance degradation when increasing the number of tabs.

The reason for low TLP here is that even for these two cases, we do not see a real “multi-tasking” scenario; instead, we see a main task and several light-load tasks, and that does not exhibit a high TLP. For instance, for multi-tab browsing, only the visible web pages will be loaded at regular speed, and all background pages will be given much less

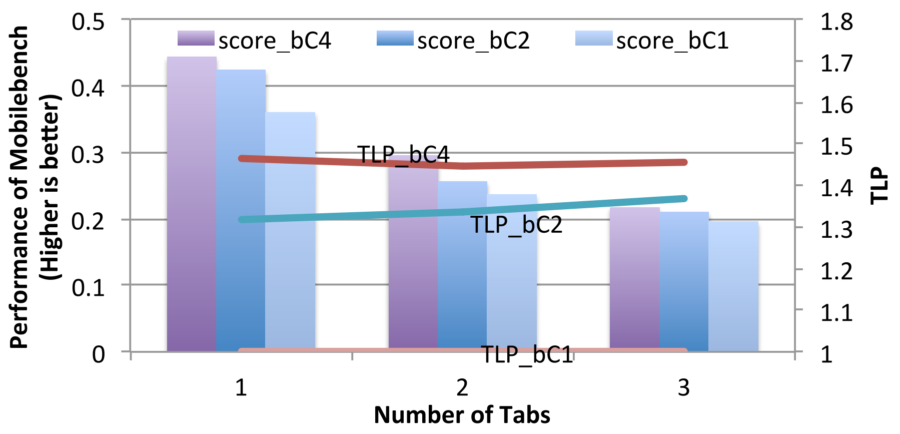


Figure 3.6: Performance and TLP results for browser. Performance are shown in columns and TLP in lines. Performance scores are calculated by taking the inverse of MobileBench rendering time then normalize against the worst score in each graphs. For instance, score_bC4 stands for performance of four big cores, lC2 for two little cores, etc. For each CPU configuration, we test three scenarios: 1, 2 and 3 tabs running MobileBench on Chrome. This graph shows big cores with stock Browser.

priority and use less CPU. Energy and thermal constraints are tight in mobile devices. It might be that the developers realize there is not enough need for implementing a fast but energy-hungry multi-tab browser for mobile phones. Reasons may include small display size or typical user behavior. In other words, the physical constraints and use pattern could reduce the amount of parallelizable work of mobile applications. In the future, we may have phones with bigger screens and higher resolutions, but human user perception will not change. On the other hand, the desktop TLP study by Blake et al. [17] showed the TLP of desktop applications remained relatively low, even after 10 years of effort writing parallelized software. Similarly, we have not seen a significant increase in TLP compared work from one year ago [40]. Clearly, parallelizing software is an extremely challenging problem, particularly for desktop/mobile applications.

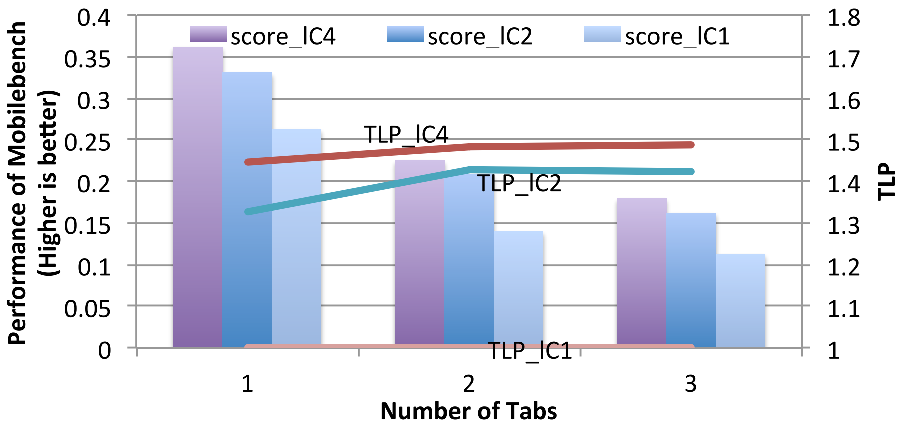


Figure 3.7: Performance and TLP: little cores with stock Browser

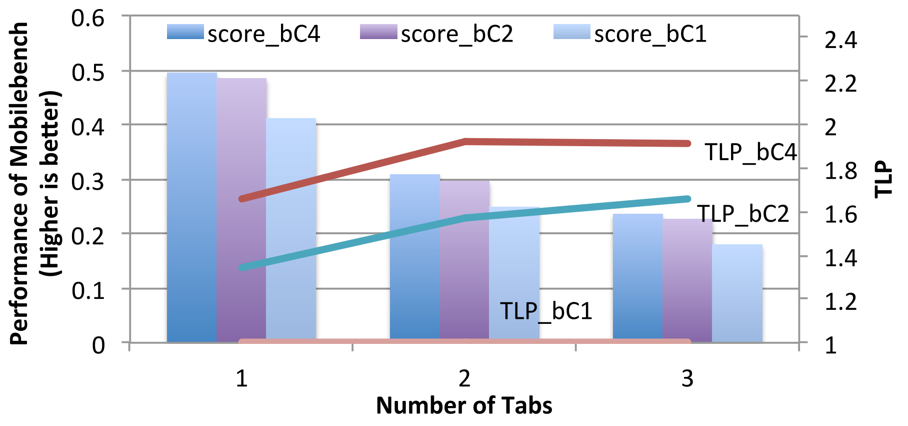


Figure 3.8: Performance and TLP: big cores with Chrome

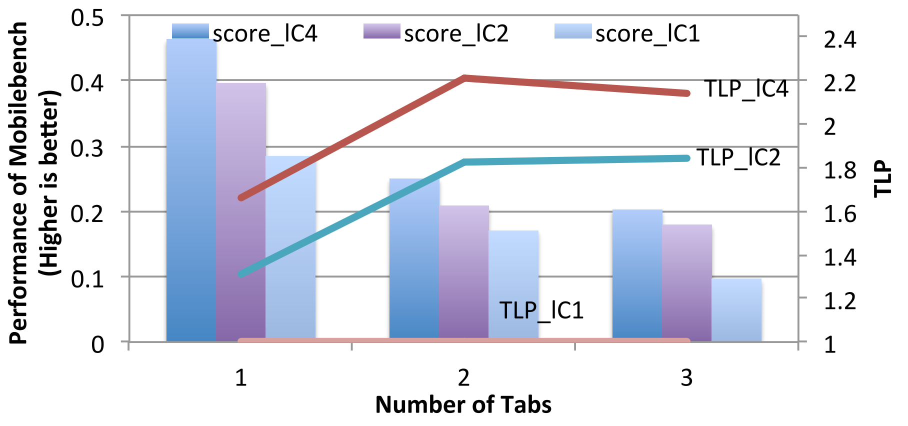


Figure 3.9: Performance and TLP: little cores with Chrome

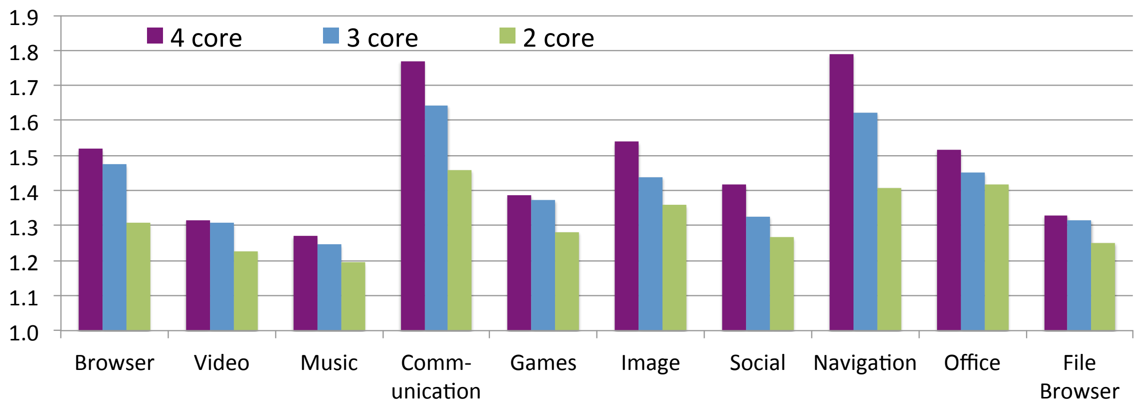


Figure 3.10: TLP result for the little cores

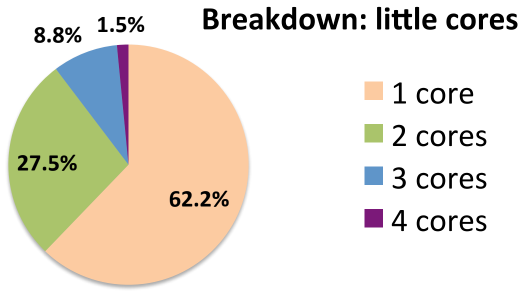


Figure 3.11: Average CPU Time breakdown: little cores

3.3.4 Alternative Architecture

3.3.4.1 Little Cores

We also perform the same set of TLP tests on the little cores in order to see how a less powerful CPU would affect TLP. We present our results in Fig. 3.11. Both the TLP and the average percentage of time that four or three cores are utilized has increased when using little cores compared to big cores. One reason is that tasks on more powerful cores run faster and finish earlier, reducing the overlap between them. This result suggests that as CPU architecture designs improve, exploiting TLP will be harder. The CPUs will be less utilized if software developers fail to produce better parallelized program.

Nevertheless, we still have TLP less than 2. Further suggesting that software is still the main limiting factor in exploiting TLP.

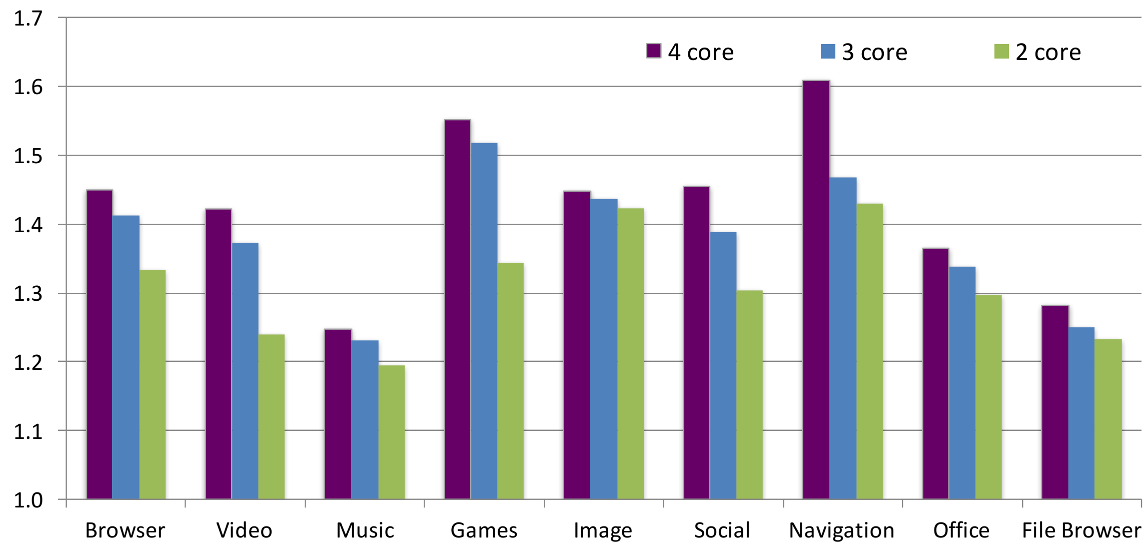


Figure 3.12: TLP result for the Origenboard (Quad-core 1.4 GHz A9)

3.3.4.2 A9 and Krait CPUs

We also perform the same set of TLP tests on two other development boards: the Origen board and Dragon board. The results are shown in Fig. 3.12, 3.13 and Fig. 3.14, 3.15. There are two observations to make: a.) both boards still show a limited amount of TLP; b.) the Origen board has higher TLP and 4-core and 3-core time than that of the Dragonboard. This echoes the observation in 3.3.2 that more powerful CPU tends to have less TLP.

In all, this result implies that as CPU architecture designs continue to evolve, exploiting TLP will be harder in more powerful cores. The CPUs will be less utilized if software developers fail to produce better parallelized program.

3.3.5 GPU

Applications like games and web browsers require large amount of graphical computation. On the hardware side, almost all mobile device SoCs now contain their own GPU units. On the OS side, the Android 2D rendering pipeline has started to support hardware acceleration in Android 3.0 (*Honeycomb*). Hardware acceleration is enabled by default from Android 4.0 (*Ice Cream Sandwich*). Therefore, it is important to analyze the actual utilization of

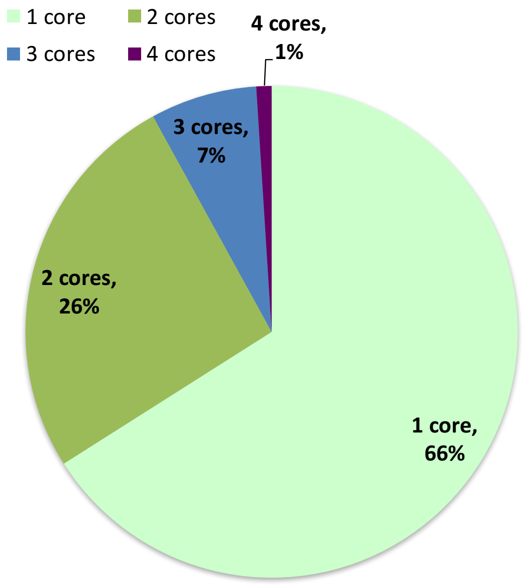


Figure 3.13: Average CPU Time breakdown: Origenboard

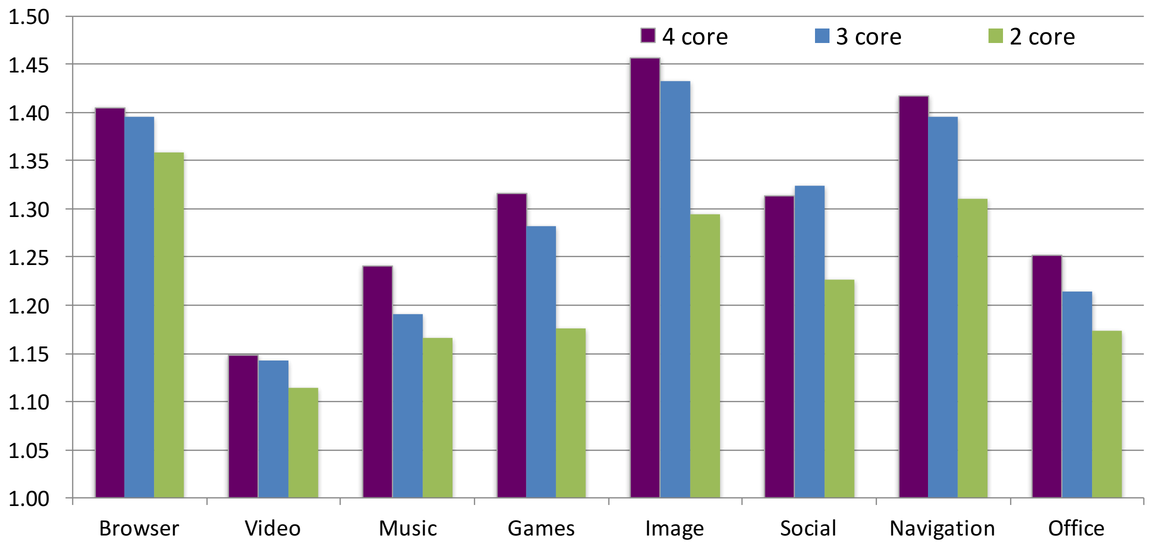


Figure 3.14: TLP result for the Dragonboard (Quad-core 2.15 GHz Krait 300)

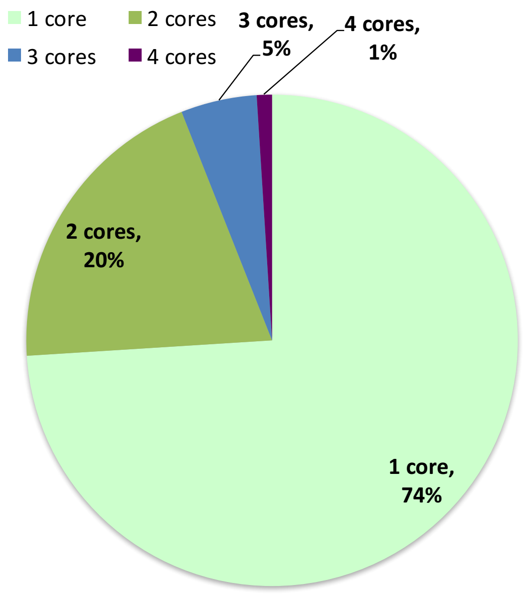


Figure 3.15: Average CPU Time breakdown: Dragonboard

mobile device GPUs.

We measure the GPU utilization of the same suite of applications on the Odroid board. We show our experimental result of GPU usage in Fig. 3.16. For each category, the leftmost bar shows the average GPU utilization when 4 big cores are kept activated, then 3 big cores, 2 big cores, and little cores. We have not found much variance when we change the number and type of CPU cores. Average GPU utilization is 24.1%, and some specific applications such as games, communcation (chats in the graph) and navigation utilize a considerable amount of the GPU. This is an indication that a part of the parallelism is already offloaded from the CPU to the GPU, which reduces the amount of parallelism that the CPU can exploit.

Given the availability of programmable GPUs, an increasing amount of general-purpose, non-graphics work can be offloaded from the application cores to the GPU or other accelerators for performance and energy efficiency. This led us to examine the energy efficiency of computation offloading for mobile platforms. We analyze several applications on both the CPU and the GPU and present the results in Fig. 3.17. We run the OpenMP and OpenCL

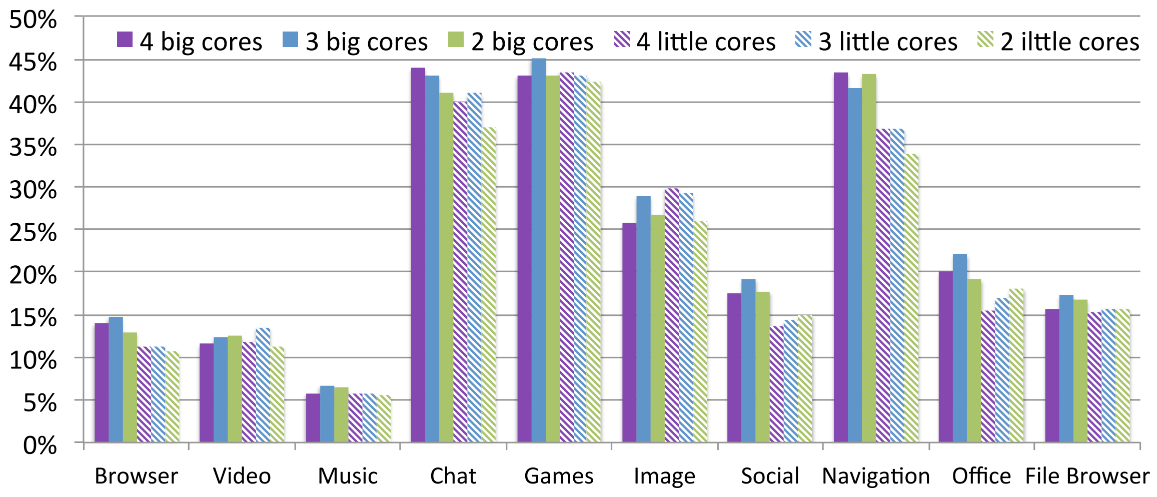


Figure 3.16: GPU utilization of different category of apps. Columns with different colors represent system configuration with different number and kind of cores activated.

versions of three machine learning algorithms – kmeans, backpropagation (BP), and nearest neighbor (NN) as well as a streaming algorithm Daxpy on a Qualcomm Snapdragon board, one of the few development boards which support offloading for general-purpose GPU applications. We evaluate the machine learning algorithms because these are the important building blocks of application domains such as audio recognition, image recognition/processing, and recommendation algorithms.

The programs running on the Krait CPU are written in OpenMP whereas the programs running on the Adreno GPU use OpenCL to exploit the heterogeneous GPU compute unit. We find that for Daxpy, the Krait CPU achieves higher energy efficiency than the GPU (less than 1). This means that when the parallelism can be well exploited by the software, Daxpy, and when the instructions are simple enough for the CPU, the energy efficiency of the multicore CPU can be equivalent or slightly higher than that of the GPU. On the other hand, for the machine learning algorithms, GPU offers higher energy efficiency of varying degrees. This suggests that, for software where there is an ample amount of thread- and data-level parallelism, e.g., the machine learning algorithms, and where there are instructions that can be accelerated by the GPU, e.g., the multiplication, sqaure root mathematical functions, it is more beneficial to offload the computation to the GPU or other accelerators

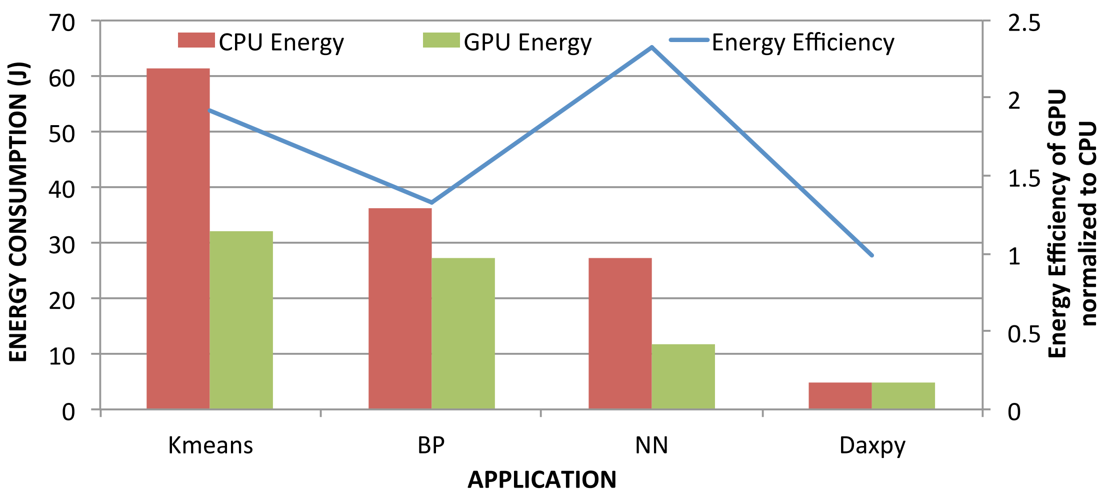


Figure 3.17: Energy consumption and energy efficiency (defined as performance per watt) comparison for Krait CPU vs. Adreno GPU.

for performance and energy efficiency. For workloads with more branch-divergence, or with a small amount of parallelism, it is more efficient to run them on the CPU. In short, the variety of mobile workloads suggests that we should look deeper into building a suitable heterogeneous system to take advantage of the different types and the varying degree of parallelism and better utilizing the existing hardware real estate.

3.4 Suggestions

In the previous section, we demonstrate that current mobile applications are not fully utilizing mobile devices, and simply adding more cores can be over-provisioning. However, it is not clear yet what kind of system is more desired for mobile devices. In this section, we try to shed light on this question by further analyzing the TLP behavior and energy efficiency of current CPUs. We make the following two observations:

a) TLP behavior exhibits short peaks and long valleys rather than staying constant. This suggests that during peak TLP times, higher performance is desired: the system needs to be kept responsive for better user experience, also these peaks are short so the extra computation power will not have a big impact on total energy consumption. During low

TLP times when the performance requirement is low, better energy-efficiency is required as any extra power will be a waste and will affect battery life.

b) For current systems, there is a distinct energy-efficiency difference between the big and little cores.

Based on these observations, we argue that current mobile applications can benefit from a system that has flexibility to accommodate both high performance and good energy-efficiency under different applications as well as different program phases. Architectures including heterogenous multi-cores. [47, 48] and flexible core architectures[49, 50, 51] might be among the possible solutions.

3.4.1 TLP vs. Time

We record the TLP over time for applications and present the results. We choose the 20 seconds[3]of the test starting from launching the applications. For Browser, pronounced peaks can be seen in TLP (Fig. 3.18). In MobileBench these occur when the application is launched and new browser webpages are opened (these actions are labeled in circled numbers in Fig. 3.18). We also see a shift of peaks towards the right from a 4 core system to a 2 core system corresponding with the actions, which reflects a quicker webpage load time in a 4 core system. For MXPlayer, there are more significant peaks during application launch and when starting, pausing and resuming a video. For Angry Birds, there are less pronounced peaks during runtime but it still shows one when the application launches as well as few others during the game.

These results show that the interactive nature of most mobile application can cause TLP to fluctuating above 2, but the average TLP still remains low. The peaks do suggest a need for multiple cores for quicker response time, which is critical for better user experience. However, multiple cores are used only during brief bursts, mostly at the application launch time. Moreover, even during these peaks, we do not observe a constant high peak TLP

[3]20 seconds is enough for apps to reach steady state.

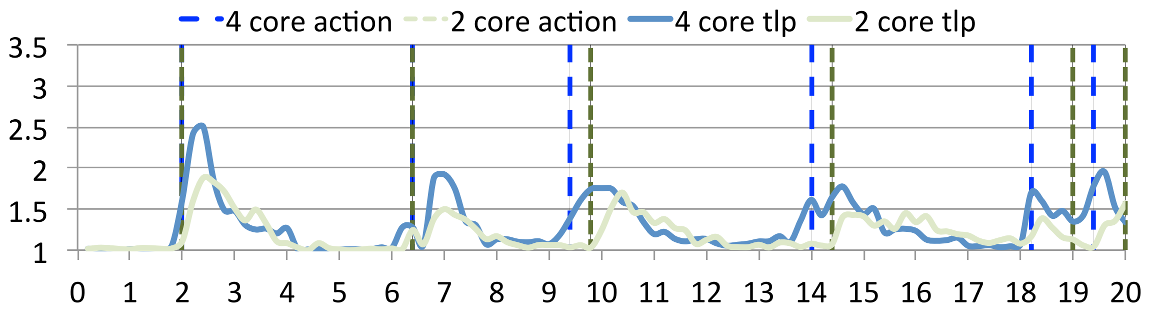


Figure 3.18: TLP vs. time in seconds for mobile applications. For Browser, the solid lines represent TLP, and the dashed vertical lines show when there is an action, such as application startup or opening a new webpage. Actions are labeled by circled numbers. This graph is for stock browser (using 4 cores).

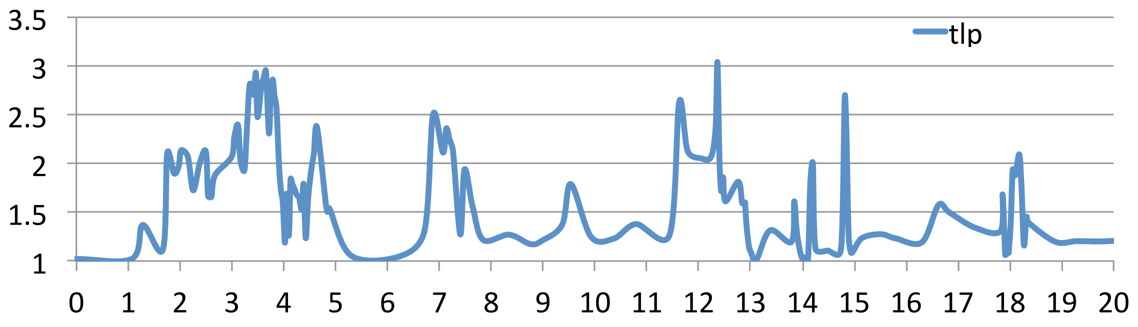


Figure 3.19: TLP vs. time: chrome

(above 3). This suggests that the idea of keeping many big cores (four or even more) for short bursts may not be a good choice for mobile devices. Instead, a system that can provide high performance during peaks and good energy-efficiency fits better with the fluctuate TLP pattern of mobile applications.

3.4.2 Energy Efficiency of Big and Little Cores

The processor takes up a substantial portion of power consumption in mobile devices [52]. It is meaningful to analyze the energy efficiency between big cores and little cores. We run MobileBench on the Odroid board for different types of cores, different number of cores, and three different frequencies[4] on each cluster (big and little). We show the results

[4]We use 1.6, 1.2 and 0.8GHz for big cores, and 1.2, 0.8, 0.5GHz for little cores.

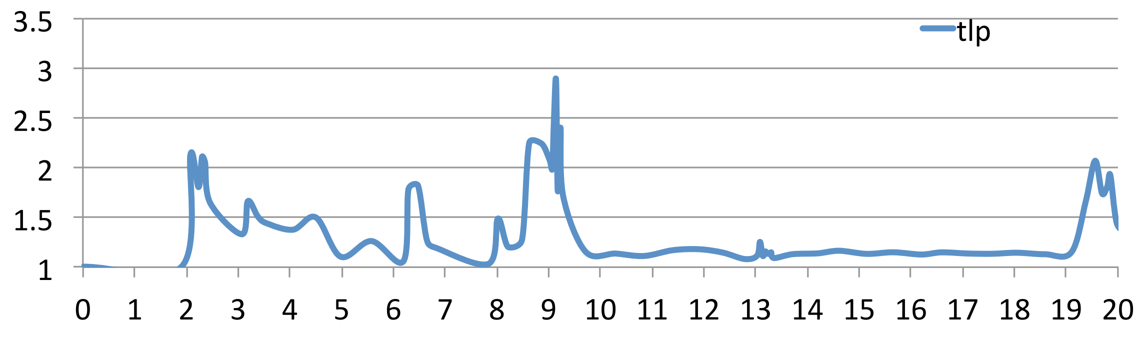


Figure 3.20: TLP vs. time: MXPlayer

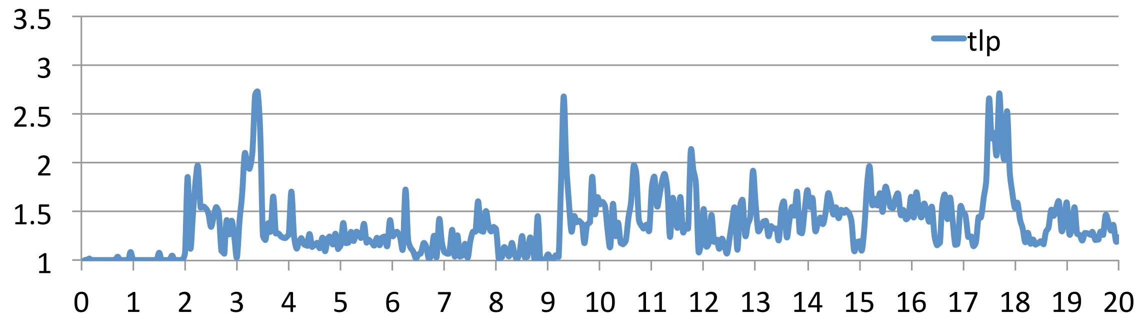


Figure 3.21: TLP vs. time: Angry Birds

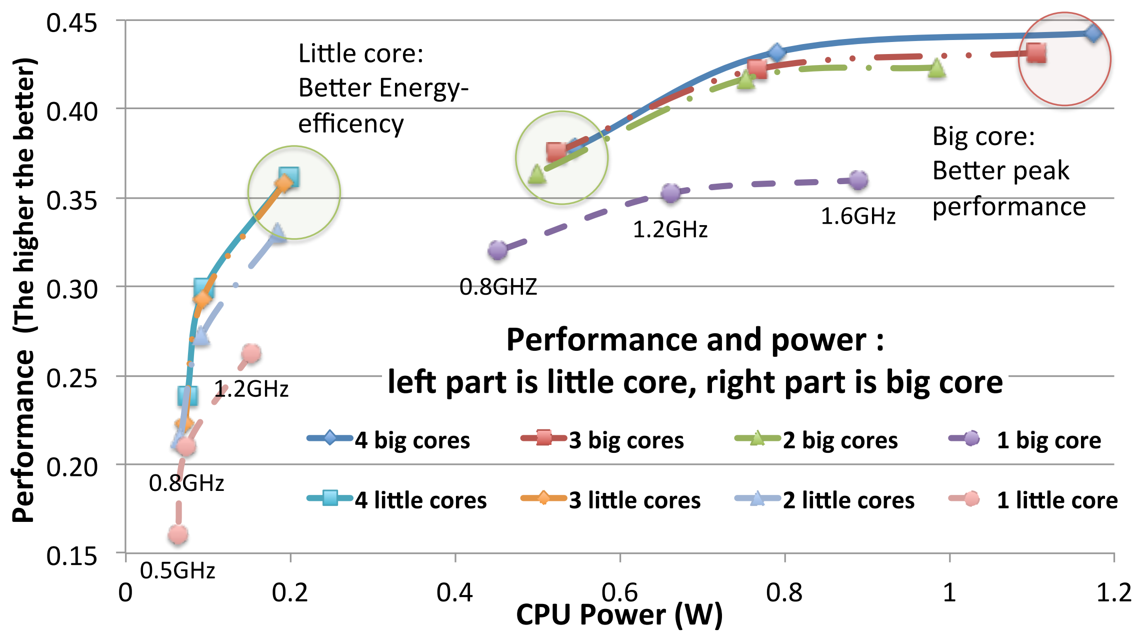


Figure 3.22: Performance and power under different frequency and cores (tested using MobileBench). Lines represent different cores configurations; dots on lines represent different frequencies, with lower frequencies (thus poor performance) on the left. Error bars are drawn to a show range of scores for each dot.

on two different clusters in Fig. 3.22. For every core/frequency combination, we did four repeated runs. The results show a distinct energy-efficiency difference between the big and little cores: big cores have better performance, but little cores only use roughly a quarter of the power consumption than big cores. Though big cores have approximately 25% less execution time, their power consumption is 3× more than little cores so they consume more total energy. In a system with flexbility, we can use little cores as much as possible, and big cores in situations that are both computational intensive.

Additionally, the importance of user experience makes such architecture more desirable. Human users want to deliver a response within a user acceptable timeframe rather than finishing the task as fast as possible. For instance, literature shows that a latency less than 0.1s is not perceivable by a human [53]. Any extra resources that are used in accelerating the program to finish faster than 0.1s is unnecessary. In the case of browser performance, as shown in Fig. 3.22, we may or may not need to switch to a big core depending on the workload of the webpage and the quality of experience demanded by the user. More flexibility in such scenarios will be beneficial for both performance and energy-efficiency.

3.4.3 Thread CPU Time Distribution

We gather some statistics about the threads created and present the result in Fig. 3.23. We see that actually there are a large number of threads being generated. Most of the threads are system threads generated by the Android OS, which are very short-lived and only take up a little amount of total work. The major part of the program is still not well parallelized.

A heterogeneous system may take advantage of this situation. Many of the short-living threads are *helper threads* which are not on the critical path of program execution. If we run the *main threads* on the big cores and these *helper threads* on the little core, we may save a considerable amount of power without performance degradation. However, whether we can effectively and accurately prioritize the threads appears to be difficult, and suggests an interesting line of research for the future.

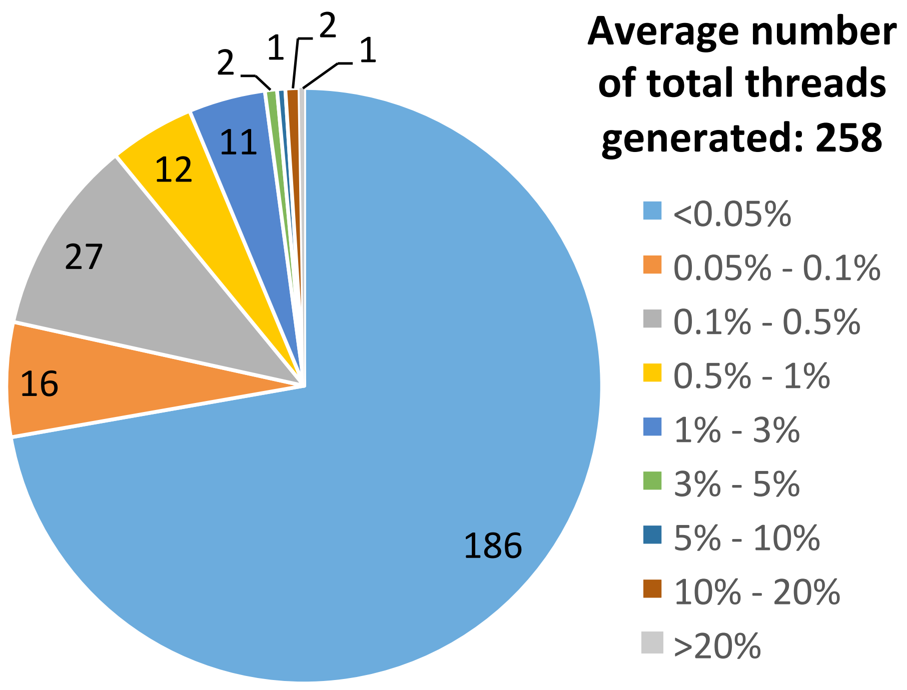


Figure 3.23: Thread CPU time distribution. We break down the threads by their CPU time into 8 categories, which are shown in the legend. For instance, "<0.05%" means this thread only occupies less than 0.05% of all CPU time, ">5%-10%" means larger than 5% but less than 10%. For the applications we tested, the average number of total threads generated is 258. Each area for the pie chart shows the average number of threads that fall into this category. It shows that most threads are short living.

3.5 Related works

3.5.1 Mobile Device Workload Characterization

Gutierrez et al. [15] measure the microarchitectural behavior of a set of mobile applications. Their result show that real-world interactive smartphone applications differ significantly from the SPEC suite, which is a desktop benchmark. Hayenga et al. [54] present a workload characterization and memory-level analysis of internet and media-centric applications for an embedded system used for mobile applications. Berkel [55] shows a workload characterization including application, radio and media processing. Ma et al. [56] characterize performance and power consumption of 3D mobile games on three mainstream mobile SoC architectures. Canali et al. [57] make an analysis on web-based services. Sunwoo et al. [58] propose a methodology to tractably explore the processor design space and to characterize applications in a full-system simulation environment.

3.5.2 Mobile Benchmarks

To accurately characterize mobile device applications, a set of representative benchmarks is necessary. Traditional desktop benchmarks such as PARSEC or SPEC are not suited for mobile devices [15]. Recently, a few number of mobile device benchmarks has been proposed. Mobile bench [44] is a collection of applications, including a revised version of BBench, Adobe pdf reader, Photo viewer and Video playback. Mobybench [43] is also comprised of popular applications, which has already been ported to the gem5 simulator [59]. AM-Bench [60] an open source based mobile multimedia benchmark for Android platform. Some more application-specific benchmarks have also been proposed [61, 62]. Most of these suites have not cover many other commonly used applications such as social apps or games.

3.5.3 Parallelism in Programs

Flautner et al. [3] propose the definition of Thread Level Parallelism. Both they and Blake et al. [17] measure the TLP on desktop applications. The latter work, which is more recent, argues that 2-3 cores were enough for desktop applications. To the best of our knowledge, there is no such study about TLP on mobile platforms.

3.5.4 Mobile Device Power Consumption

Due to the limited capacity of battery in hand-carry devices, a number of works have been done on mobile device power consumptions. Caroll et al. [52] make an anlysis of smartphone power consumption by providing a detailed breakdown of the device's main hardware components. The work by Simunic et al. [63] is a early work on dynamic voltage scaling for portable systems. Yang et al. [16] propose an adaptive user-and-application-aware dynamic CPU frequency scaling technique, which saves power by scaling the system down to a minimum user-acceptable frequency.

3.6 Conclusion and Discussion

In this chapter, we considered how multi-core processors in mobile devices are being used. We have shown that current mobile applications cannot effectively use a large number of cores. Instead, we suggest that a flexible system that can accommodate both high performance and good energy-efficiency is a more preferable choice for current mobile applications.

We have analyzed a wide range of common mobile applications, and calculated the *Thread Level Parallelism* (TLP) of these applications. The average TLP across all categories is 1.46, which shows that mobile apps are utilizing less than 2 cores on average. The applications with the highest TLP, Google Hangout, only has a TLP of just 1.8. We have also evaluated a number of different CPU configurations, including different numbers of cores, core frequencies, and CPU types. We observe a diminishing return on TLP when the number of cores increases. Even in those heavy-load real-world scenarios with background applications or multi-tab browsing, there is still not enough work to keep utilization high. Both these results suggests that having many powerful cores is over-provisioning. Due to physical constraint and interactive user patterns, mobile applications tend to have less parallelism to exploit than desktop applications. The GPU and mobile co-processors on chip also takes off work from CPU. Historically, the desktop TLP study by Blake et al. [17] showed the TLP of desktop applications remained relatively low, even after a 10 year gap. It indicates that parallelizing software is an extremely challenging problem. All these contribute to the low TLP for current mobile applications.

On the other hand, we find out that TLP behavior exhibits peaks and valleys rather than remaining constant. User experience, which is critical for mobile applications, also varies by different application scenarios and different users. A system with the flexiblity to satisfy both high performance and good energy-efficiency for different program phases is a good choice for mobile devices.

We believe this work can motivate new research directions. TLP is a utilization metric rather than a performance metric. We have only used web browser benchmarks [15, 44] as performance metrics in this chapter; the research community can benefit from having benchmarks that quantitatively measure the performance for popular applications of various categories such as games, social, office, etc. Responsiveness is another metric worth noting, especially for user experience of interactive mobile applications [16]. This is also where the peak TLP mainly comes from. Building an accelerator or specific processor architecture tailored for such phases may be an interesting avenue of research into. Last but not least, we analyzed TLPs of different CPUs. As future work, it would also be interesting to make a further analysis of the impact of GPU and co-processors on TLP.

CHAPTER 4

A Low Power Accelerator for Always-On Applications in Wearable Devices

Always-on applications in wearable devices, such as keyword detection or heart rate monitoring, pose a significant challenge in energy-efficient design. A large class of these always-on applications execute in a deterministic and repeatable fashion, including many deep learning and signal processing applications. Determinism and repeatability mean that it is know a priori which memory elements are accessed when and how often. We take advantage of this determinism by replacing the traditional cache based architecture with a non-uniform scratchpad architecture (NUSA) where certain address ranges are intentionally optimized for low access energy while others are optimized for high area density. An optimal memory access pattern is then pre-computed statically, taking advantage of this non-uniform memory architecture, placing frequently access elements in low energy address ranges thereby reducing overall energy consumption.

In this chapter, we present a NUSA based accelerator designed for this class of applications. To fully utilize the proposed NUSA, this chapter develops a framework, which: a) determines the most energy efficient NUSA scratchpad design within area constraints; and b) identifies the most energy efficient data assignment and runtime schedule given the target application. We evaluate a wide range of applications, and compare against traditional approaches. A fabricated prototype accelerator is also presented as an illustration of our technique. We show that, on average, a 2.1$\times$ reduction in energy can be achieved by using

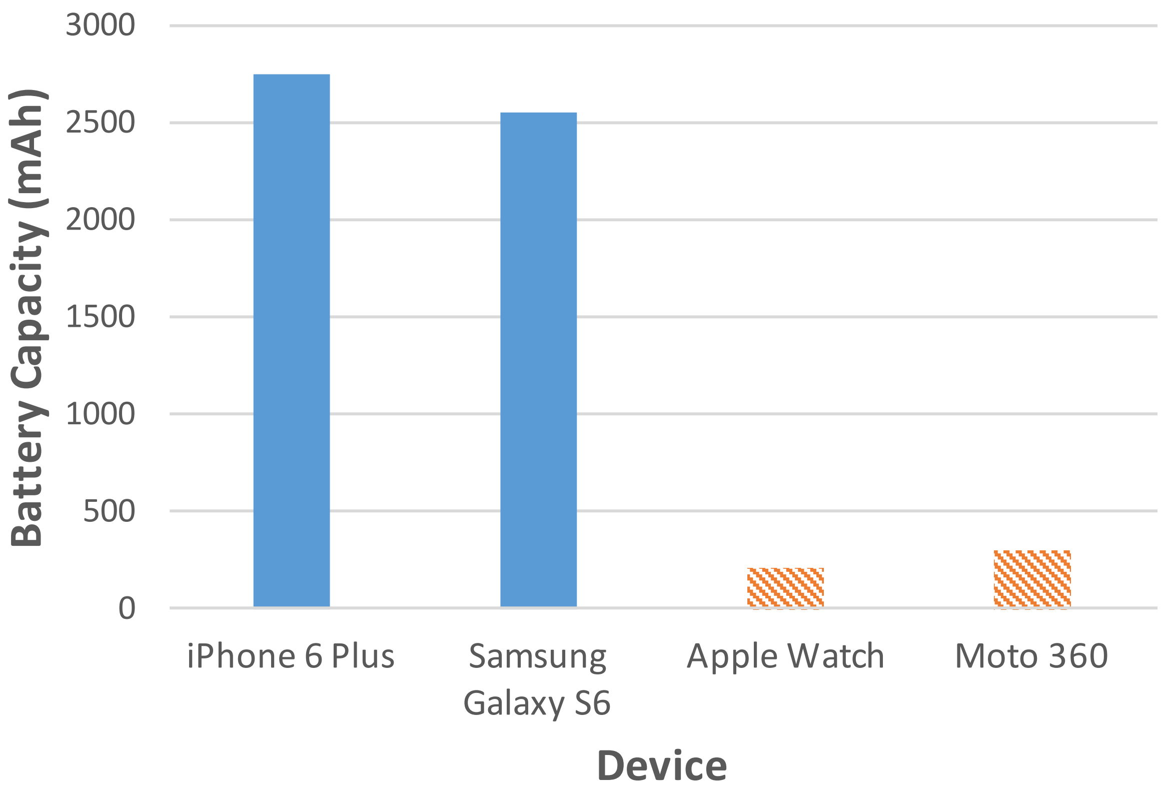


Figure 4.1: Battery constraint in wearable devices

a NUSA architecture compared to a uniform scratchpad architecture, and a 10× to 36× reduction compared to general purpose ARM M-class cores and DSP.

4.1 Introduction

Wearable devices, being in close proximity to users, are critical parts in the Internet of Things (IoT) eco-system. A flourishing group of these devices, including smart watches, smart glasses, and activity trackers, are rapidly emerging, defining new man-machine interfaces and compelling user experiences. One key aspect of this new interface is the continuous, ”always-on” access to its user; examples include voice activation for smart watches and user gesture identification on activity trackers. Wearables also serve as an important filter on the very edge of the cloud, triggering more powerful compute resources only when the always-on applications find that the information from the sensors is meaningful.

Application	Application Processor	TI C55 DSP	**Smartwatch target**
Voice activation	20mA	4.5mA	**0.6mA**
Wake-on-gesture	40mA	1.5mA	**1.1mA**
Bluetooth Low Energy	0.03mA	0.03mA	**0.03mA**
Battery Life	**5 hours**	**50 hours = 2 days**	**173 hours = 7 days**

Table 4.1: Expected battery life target for wearable devices [4]

Energy consumption is the key design factor for these always-on applications. By definition, these applications need to be running continuously; this is very different from what is usually observed in mobile devices [18, 64], where much of the silicon can be kept ”dark”, or off, as long as possible. In addition, wearables have extremely tight battery constraints due to their form factor. Battery capacity can be about an order of magnitude lower than the already constrained budget of smartphones, as shown in Fig. 4.1.

Meeting such strict energy consumption proposes a challenge to traditional designs. The display in wearable devices consumes a much lower portion of the total energy compared to one in a smartphone [18, 19], which in turn makes optimizing the processing power even more critical. Using powerful general-purpose processors with a high-level OS is an untenable solution. Even with a DSP, it is hard to meet user expectations of charging the wearable devices no more than once a week. Emerging applications, which demand more computation, will make it even more difficult to meet this tight energy budget.

One key observation about always-on applications is that they run in a *deterministic and repeatable fashion*. They are usually tailored to the constraints of the wearable device they are running on, such as the type/number of sensors available or the use case of the device. For instance, a smart watch with a microphone and a gyro can execute keyword spotting and user gesture detection, but it is not designed to run face detection or heart rate seizure detection due to the lack of appropriate sensors. Though the data being used, such as the voice pattern of the user, can change over time, the inherent algorithm and computation will remain the same. Due to their specific functionality, always-on functions are often implemented as accelerators. Their deterministic and repeatable computation makes it possible to determine a priori the memory access patterns and design the accelerator to be more energy efficient.

With this observation, we present a low power multi-application accelerator design for always-on applications in wearables. Using the deterministic characteristics of always-on applications, we make several design choices to avoid unnecessary overhead while providing flexibility. First, we propose to use an on-chip, software-managed scratchpad memory instead of a cache based design. Second, the energy consumption is further reduced by utilizing a non-uniform scratchpad architecture (NUSA): frequently accessed data are assigned to nearer, smaller scratchpad banks with low access energy, and infrequently accessed data to further, larger banks with higher density. Third, to address the challenge of identifying the optimal NUSA architecture and data layout, a two-phase design framework is proposed: 1) in the **pre-silicon** accelerator design phase, the framework determines the best NUSA hierarchy, including the number of levels in the scratchpad and the size of each level; 2) in the **post-silicon** scheduling phase, the framework chooses the number of active processing elements and determines the data assignment and optimal access pattern for the NUSA scratchpad, pursuing the best energy-efficiency within the application latency requirement. By solving a formal optimization problem, the proposed framework identifies the energy-efficient data placement and corresponding optimal access pattern that are NUSA architecture specific and also application dependent.

To demonstrate these ideas, we design a multi-application accelerator targeted towards always-on applications using deep learning algorithms, which has been fabricated as a prototype chip. Based on the computationally intensive kernels from the applications, a CISC instruction set is formed for the accelerator with the goal of minimizing design and runtime programming complexity. Finally, we further generalize the architecture by evaluating the latency and energy of different designs, and evaluate the framework compared to both uniform scratchpad architectures (USA) and cache based architectures. On average, the proposed approach exhibits a 2.1× reduction in energy by using NUSA compared to USA, and a 10× to 36× reduction comparing NUSA with general purpose ARM M-class cores and DSP.

To summarize, we make the following contributions:

- Identify the opportunity of utilizing the deterministic behavior of always-on applications in wearable devices, and the benefit of utilizing a non-uniform scratchpad architecture (NUSA).

- Design a framework which can generate a NUSA design for a wide set of applications, as well as tailor and map the applications to the target NUSA for the best energy-efficiency.

- Design a low power programmable accelerator for always-on wearable applications, which has been fabricated as a prototype chip.

- Evaluate the framework by generalizing the accelerator design and comparing to general purpose core baselines, as well as cache and USA based accelerators.

The rest of the chapter is organized as follows. Section 4.2 describes background information. In Section 4.3 we describe the accelerator design framework. With the framework, we first present an example accelerator design, which has been fabricated in a prototype chip, in Section 4.4. Then we generalize the design, present and analyze the results in Section 4.5 and Section 4.6. We discuss the related works in Section 4.7, and, conclude the chapter in Section 4.8.

4.2 Background

4.2.1 Always-on applications

We examine several example categories of always-on applications, which are summarized in the following subsections.

4.2.1.1 Keyword spotting

Keyword spotting is a detection task to identify the presence of specific spoken words in a stream of speech signals. It is often used to trigger automatic speech recognition and spoken dialog systems. It is common in wearable devices when hands-free activation is desired. Examples include detecting ”OK Google” when wearing a Google Glass or ”Hey Siri” when wearing an Apple watch. Keyword spotting can be implemented using classification techniques, such as hidden Markov models (HMM) [27] to identify different keywords. More recently machine learning algorithms based on Deep Neural Networks [28] and Recurrent Neural Networks [29] are being deployed because their better detection accuracy.

4.2.1.2 Seizure detection

Epileptic seizure detection refers to the use of algorithms to recognize the occurrence of a seizure. Typically these algorithms are based on the analysis of biological signals from a patient with epilepsy. They can be deployed in smartwatches or health wristbands for quick, always-on detection. The algorithms are typically based on a nearest-neighbor classifier of EEG features [30, 31], or the RNN technology mentioned above [32, 33].

4.2.1.3 Face detection

Face detection is the detection of a face in a scene. It is commonly used as a front-end to trigger facial recognition. It appears in Google Glass and smart home detection devices. Due to its wide range of application, face detection algorithms have been studied heavily.

Recent implementations include DNNs [34] or Convolution Neural Networks (CNN) [35, 36]. The former is easier in terms of memory access patterns but the latter is usually more suitable for image processing.

4.2.1.4 Wake on user gesture

Wake-on user gesture is the feature of some wearable devices which can be activated using certain user-specified gestures. This enables the device to recognize who is interacting. For an entertainment device, it can recognize the user and load the right game profile or music play list. For a home climate control, it can adjust the environment to the wearer's preference. This feature can be implemented using one-vs-all classification [37] or margin classifiers [38].

4.2.2 The status quo for wearables

Battery life is still a major constraint for current wearable devices (Table. 4.1.) One typical method of reducing power consumption in smartphones is to reduce the brightness or resolution of the screen, which takes up a large portion of the total power consumption. However, the display in wearable devices already consumes a much lower portion of total energy than those in a smartphone. It can be as much as 20-40$\times$ fewer pixels and has only a relatively simple color scheme [18, 19]. Another major power consumer in wearables is the wireless connectivity module, which is usually implemented in very specific communication ASICs and is difficult to further optimize [20]. Therefore, optimizing the power consumption of the compute stack becomes critical.

4.2.3 Non-uniform scratchpad architecture

The idea of non-uniform memory architectures has been widely adopted in memory and cache design for multi-core systems. In this work, we utilize non-uniform memory on embedded systems in the form of a scracthpad—NUSA. The strategy is to include both

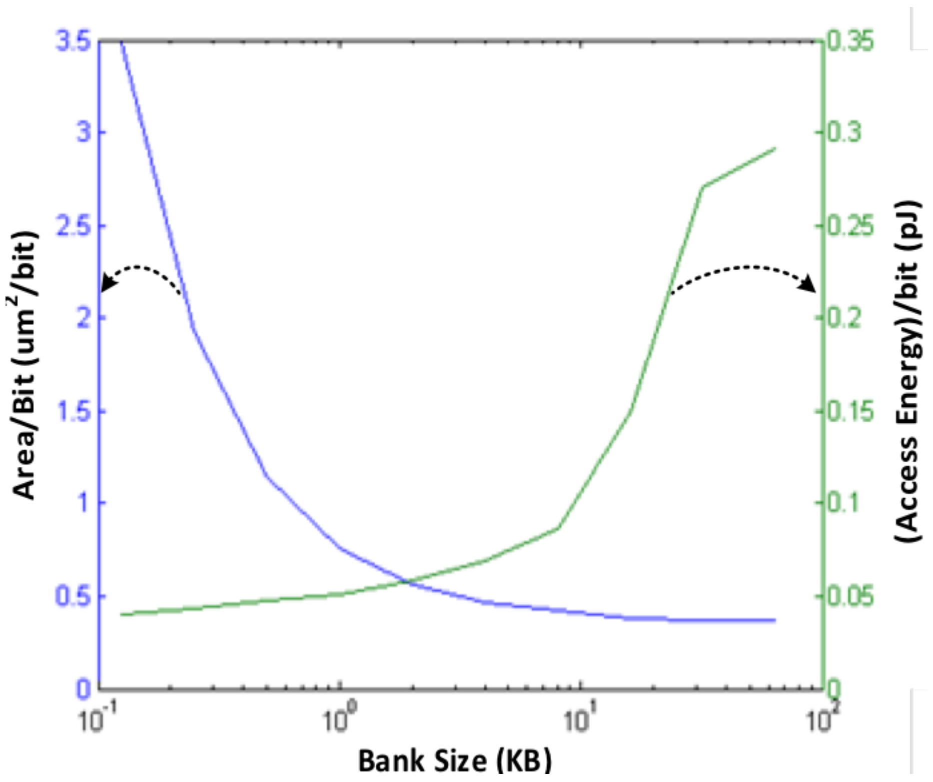


Figure 4.2: Trade-off between area and energy (data from SRAM compiler)

small/nearby and large/far-away banks in order to differentiate the characteristics of scratch-pad banks. Fig. 4.2 is generated by a typical SRAM compiler using a commercial 40nm CMOS process with a 6T cell. As shown in the figure, smaller banks can provide lower access energy but at the cost of lower density. Larger banks, which have the opposite characteristic, can be included to compensate for the low density. In addition, smaller banks can be placed nearer to the processing element, further reducing the energy spent on the interconnect.

To utilize NUSA, the programmer should assign frequently accessed data to the small banks for energy efficiency, and infrequently accessed data to the large banks for density. For instance, in matrix vector multiplication, the elements in the vector, which are accessed many times, should be placed in the small/nearby banks. Meanwhile the elements in the matrix, accessed only once, should be placed in the large/far-away banks. When the application becomes complicated, it can be difficult to find the optimal NUSA structure. We will address this in Section. 4.3.

4.3 Accelerator design framework

Fig.4.3 illustrates the overall procedure of the proposed framework. It consists of two phases: pre-silicon and post-silicon. The pre-silicon phase provides the accelerator chip design: it determines the best NUSA layout, including the number of different scratchpad levels and sizes. After the chip design has been set, the post-silicon phase acts like a compiler: it determines which kernels should be running on the accelerator, how many processing elements to use, generates the best data partitioning, and the operation schedule. That is, the post-silicon phase identifies the optimal memory access and processing patterns for each target application given the NUSA design determined in the pre-silicon phase.

4.3.1 Pre-silicon

The pre-silicon phase decides on the optimal NUSA architecture considering a wide set of target applications that are anticipated to run on the multi-application accelerator. To do so, it is given the set of kernels supported by the accelerator, the area constraints, and the characteristics of density and access energy for the type of memory element being used in the scratch-pads. The pre-silicon design phase is performed in four steps, which are discussed in the following subsections.

4.3.1.1 Determine the kernels for acceleration

First, the applications are analyzed, and all kernels to be supported on the accelerator are identified. Typical kernels include matrix-vector multiplication, signal transform (FFT, DCT), and non-linear activation funcation. An example set of kernels supported by an accelerator design will be described in the Sec. 4.4.3. All other kernels that are not supported by the accelerator will be handled by the general purpose microprocessor (i.e., the control

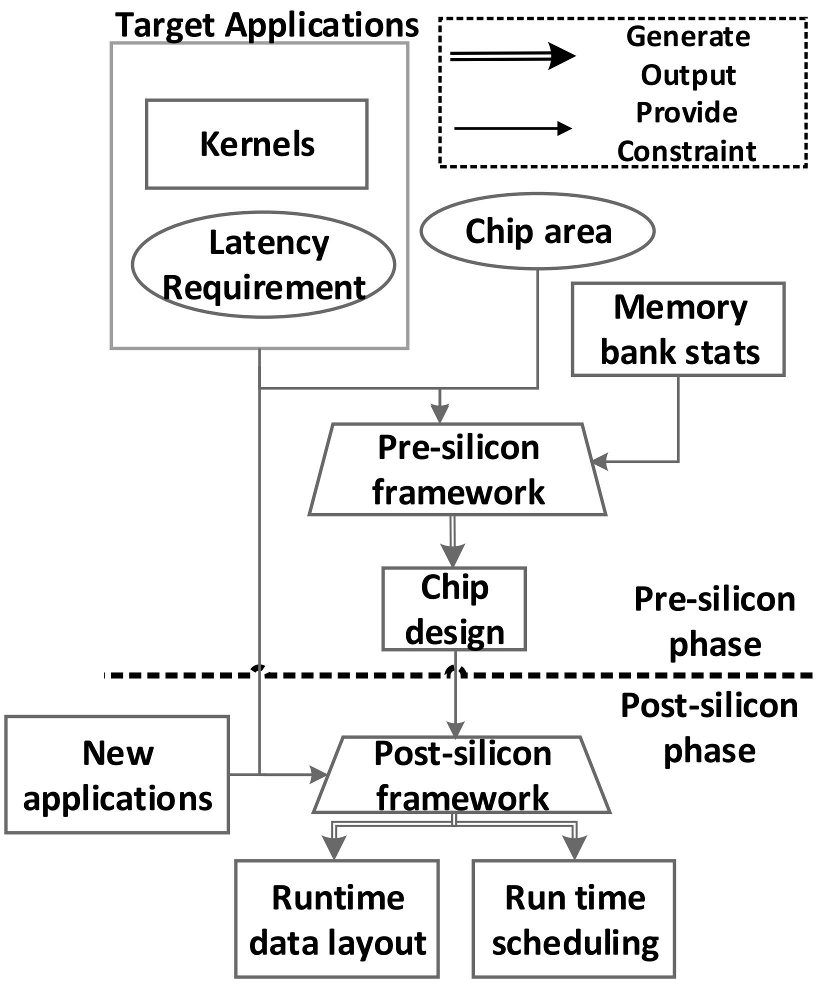


Figure 4.3: Overall description of framework

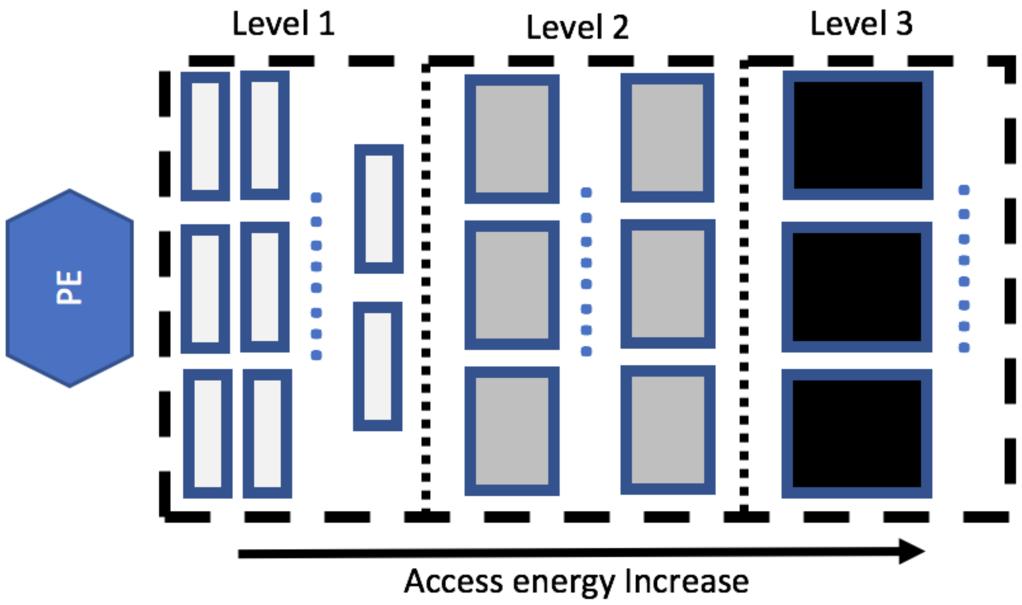


Figure 4.4: Example memory layout. Banks are stacked on one side of the processing element (PE), starting from smaller banks. The height of the stack is determined by the framework, striking a balance between average wire distance and total area.

core).

4.3.1.2 Determine the data element characteristics

The framework analyzes all the kernels, and distinguishes the type of data access along three axes—a) lifetime, b) access frequency, and c) size: a) The lifetime of the data can be categorized as either temporary or static. Temporary data means it will be used only once in one iteration of the application and discarded; static data means it will be used many times or seldom changes across different iterations of the runtime. For instance, in keyword spotting, the speech signal sent by the sensor to the accelerator is considered temporary data, while the neural network classifier parameters are static data. Since always-on applications will usually run many iterations, these two data types are quite different. b) The access frequency of the data can be used to determine data placement—more frequently accessed data assigned to small/nearby banks, and less frequently accessed data to large/far-away banks; c) The sizes of data is necessary to determine the memory size of each level in the NUSA design.

4.3.1.3 Determine the NUSA configuration

In this step, the most energy efficient NUSA configuration to enable the optimal trade-offs in access energy and bit density is determined. Access ratios between different NUSA levels are also specified in this phase. This problem can be formalized as an optimization problem described below:

We have n different types of data with different access frequencies. For every type of data i we have its size T_i, number of elements N_i, and access frequency F_i. For the technology node and memory used, we have a bank access energy function $E(x)$ and density function $D(x)$, where x is the unit of bank size. The access energy also includes energy for the wires $E_w(i)$ and routing peripherals $E_p(i)$, which are NUSA layout dependent. To estimate these quantities, the framework automatically generates a layout, such as the one shown in Fig. 4.4. Given an area constraint A, we can determine the optimal unit bank size x_i and number of banks B_i for the n different types of data, which can be written as:

$$\begin{aligned} minimize \sum(E(x_i) + E_w(i) + E_p(i))N_iF_i \\ such\ that\ (\sum D(x_i)B_i) < A \\ and\ B_ix_i > T_i, \forall i \end{aligned} \tag{4.1}$$

The first term is the total access energy, the second term specifies the area constraint, and the third term species the data size requirement. Notice that this optimization problem is defined with highly discretized variables. As the optimization is performed off-line, we solve it by enumerating all possible combinations of memory bank sizes.

4.3.1.4 Determine the number of processing elements

The number of processing elements (PE) can be determined from the latency requirement of the application and computation capability of each PE. Though the scratchpad will usually occupy more space than the PEs, area constraints may also apply for PEs.

With these four steps, the multi-application accelerator design can be tailored for the set of target applications specified by the designer. Extra scratchpad space or additional PEs can also be added for potential future applications which might require more scratchpad capacity or computation demand.

4.3.2 Post-silicon

The post-silicon framework is used to map each application onto the NUSA accelerator identified in the pre-silicon phase. It contains four steps, which are outlined in the following subsections.

4.3.2.1 Determine the number of active PEs

The framework determines the number of active PEs for each kernel, with the goal of minimizing total energy consumption within the application latency constraint. The final number of active PEs could be 0, which means that the kernel is better off running on the control core than offloading it to the accelerator. This usually happens because the kernel is so small that the overhead of invoking the accelerator offsets the benefit of using it. This process can be formalized as a new optimization problem described below:

We are given n different kernels, X available PEs, and the total latency constraint for the application, T_c. For each kernel, we have a latency function $T_i(x)$ and energy function $E_i(x)$, where x is the number of PEs activated.

Latency $T_i(x)$ consists of two parts: the computation latency $T_{pi}(x)$ and communication latency $T_{mi}(x)$. When $x! = 0$, computation latency $T_{pi}(x)$ is the latency for the accelerator to finish the kernel, otherwise it is the latency to run on the control core. Communication latency $T_{mi}(x)$ is the time spent for the control core to program the PEs and gather results from them.

Energy $E_i(x)$ consists of three parts: a) control core energy $E_{ci}(x)$, b) PE energy $E_{pi}(x)$, and c) memory energy $E_{mi}(x)$. All of them can be broken down into a static

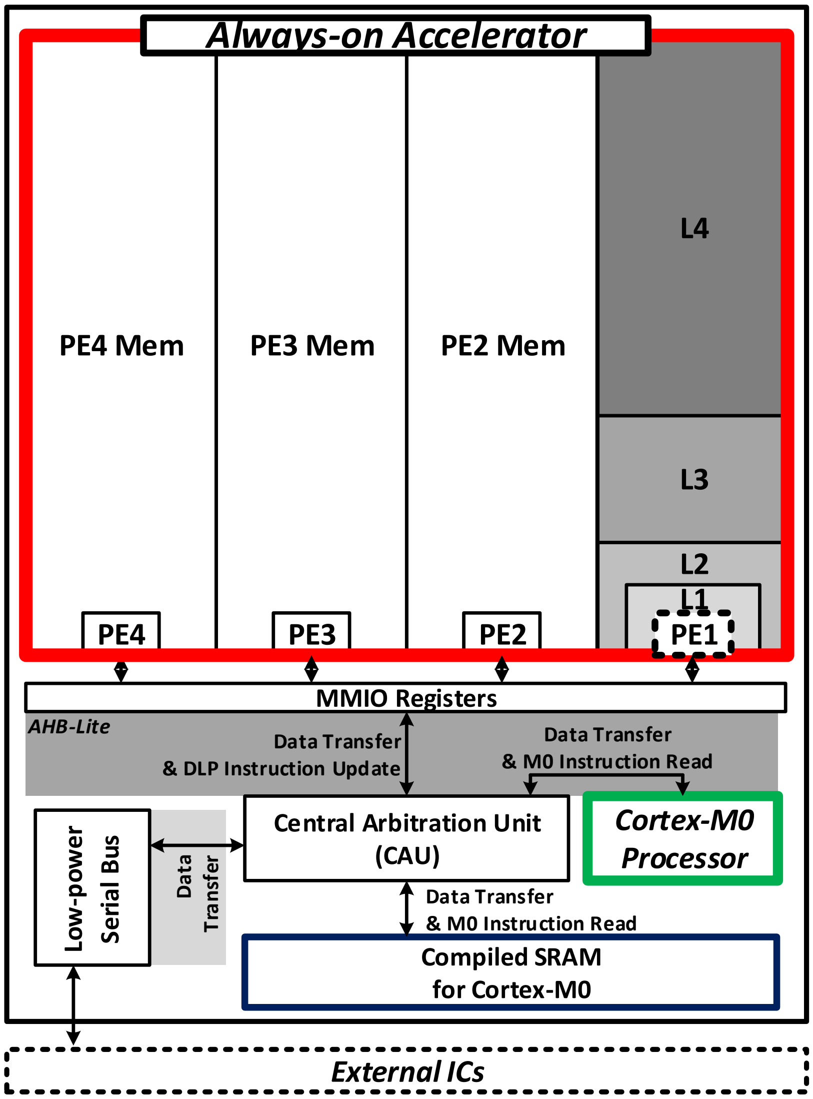
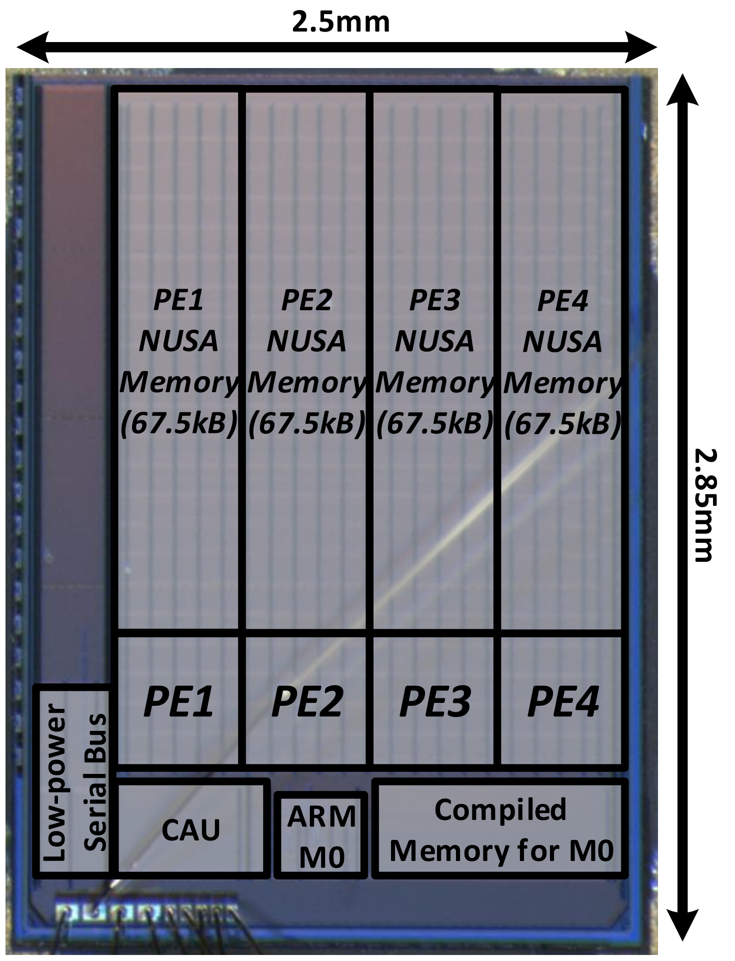

Always-on Accelerator
PE4 Mem
PE3 Mem
PE2 Mem
L4
L3
L2
L1
PE4
PE3
PE2
PE1
MMIO Registers
AHB-Lite
Data Transfer & DLP Instruction Update
Data Transfer & M0 Instruction Read
Central Arbitration Unit (CAU)
Cortex-M0 Processor
Low-power Serial Bus
Data Transfer
Data Transfer & M0 Instruction Read
Compiled SRAM for Cortex-M0
External ICs

Figure 4.5: Overall architecture of the accelerator



Process	40nm
Chip Area	7.1mm^2
Operating Frequency	1.9 – 19.3MHz
Operating Power	0.288mW
Total NUSA SRAM Size	270kB

Figure 4.6: Die photo and specifications of the accelerator chip

portion and a dynamic portion. Both the static and dynamic depend on the active time of each component, as well as the total latency. For instance, if the static power consumption per PE is Ps_{pe} and dynamic power consumption is Pd_{pe}, PE energy $E_{pi}(x)$ is then roughly $XT_{mi}(x)Ps_{ee} + xT_{pi}(x)Pd_{pe}$. More accurate calculation of energy can be used if necessary.

The problem then can be written as:

$$\begin{aligned} minimize \sum E_{ci}(x_i) + E_{pi}(x_i) + E_{mi}(x_i) \\ such\ that (\sum T_{pi}(x_i) + T_{mi}(x_i)) < Tc \\ and\ x_i < X, \forall i \end{aligned} \tag{4.2}$$

where the first term is the total energy consumption, the second term the total latency, and the third term the number of PEs. It is necessary to iterate all possible combinations, because the latency and energy consumption may not change monotonically with the number of PEs. For instance, when increasing the number of PEs, computation latency on the accelerator may decrease, but communication latency may increase since there are more PEs to orchestrate. Sec. 4.6 will investigate this in detail.

4.3.2.2 Tailor applications towards NUSA

Before mapping the kernel on the NUSA acceleartor, we first analyze opportunities to modify the kernel to make its data access pattern benefits more from the target NUSA. For instance, in the case of matrix multiplication, the concept of matrix tiling can be applied where a large matrix is divided into smaller size submatrices to be merged in the final stage. The tiling scheme assigns some portion of the NUSA memory as an access energy efficient scratchpad space which holds temporary inputs and outputs. The tiling scheme incurs overheads to merge the final output from different submatrices but the overall energy consumption is minimized because each submatrix processing exhibits significantly lower NUSA memory access energy by the optimal scheduling that aggressively utilizes nearby memories. Similarly, a large size FFT can be partitioned into multiple smaller size FFTs that can be sufficiently performed using only nearby memory banks. The framework analyzes the overhead of tiling and merging considering the memory access energy consumption associated with each NUSA layer. Note this step only modifies the data access pattern of the kernels; the output should be mathematically equivalent without affecting the algorithm functionality.

4.3.2.3 Assign the data layout for the scratch-pads

In this step we assign data to their location in the NUSA to achieve maximum energy-efficiency. This step is similar to the pre-silicon phase described in Section 4.3.1.3, except the range of the density function $D(x)$ and energy function $E(x)$ will be limited; only the bank sizes x that are actually implemented on the chip can be selected. In fact, if the application does not change, the data layout should be exactly the same as planned in the pre-silicon phase. One special case is when the application needs more scratchpad space than is available on the chip. Off-chip data access energy can be prohibitive, given the limited energy budget in wearables. Therefore, for the scope of this work, we assume data is always stored on-chip for always-on applications. The scenario in which data does not fit on chip is left for future work. This scenario can be avoided by using the pre-silicon framework to ensure there is enough space on a new accelerator.

4.3.2.4 Generate the code

Finally, after the PE scheduling and NUSA layout is determined, the framework generates the code for runtime execution. For all the kernels to be executed on the PE, their code in the original program will be replaced by the generated code, which includes the memory-mapped addresses for the data, and the code to control the PEs.

4.4 Accelerator design example

In this section we describe the prototype accelerator chip designed and fabricated in a 40nm low power (LP) process. Details on its architecture and instruction set are discussed in this section. Analysis of a more generalized architecture is given in Section 4.5 and Section 4.6.

4.4.1 Architecture

The overall architecture of the prototype chip is shown in Fig.5a. The accelerator is composed of 4 processing elements (PE), 4 levels of customized NUSA SRAMs for each PE, an ARM Cortex-M0 as the control core, a 32kB compiled SRAM for the M0, a low-power serial bus [65] for external I/O, and a central arbitration unit serving as the hub to connect these components together.

The NUSA SRAM occupies most of the chip area. Each PE has 4 levels of NUSA SRAM, with sizes of 1.5kB, 6kB, 12kB, and 48kB, respectively. In addition, every PE has access to the other three PE's NUSA SRAM through the AHB-lite bus. From the perspective of a PE, other PE scratchpads essentially becomes an extra level of far-away NUSA. This is useful in supporting applications that have data sizes larger than the capacity of the local scratchpad memory, and applications with data of many different access frequencies.

The PEs are the main computation units in the accelerator. Section 4.4.3 will discuss the computation supported by the PEs. The M0, which serves as a control core, programs the PEs by writing instructions to their instruction buffers and activates them. It also handles operations not supported in the PE or that are not energy-efficient enough to be executed on the PEs. It has access to the NUSA memory at every PE, as well as a dedicated compiled SRAM. Section 4.4.2 will introduce the operation in detail.

Internal communication between components is done via the central arbitration unit and the AHB-Lite bus. External communication, for instance with the sensors and main application cores, is handled by the low-power serial bus [65], enabling integration into a complete system. Both the NUSA SRAM and the M0 compiled SRAM are in the same address space and can be fully accessed externally. Since the optimal data placement in the NUSA is known a priori, input data from the sensors can be placed into the optimal location directly.

The architecture has been implemented as a functional prototype chip that has been fabricated, tested, and evaluated. The die photo and specification are shown in Fig. 4.6.

4.4.2 Operation

The operation of the accelerator is diagrammed in Fig. 4.7. In general, always-on wearable devices will run for long periods of time before powering-down. This means that the initialization is negligible compared to continuous operation. The initialization procedure includes loading the PE NUSA SRAMs with static data (data that will not change across different iterations, such as parameters or constants), and the M0 SRAM with code.

After the initialization procedure, one iteration of an always-on application consists of a combination of the six stages as labeled in Fig. 4.7:

1. *External I/O writes the inputs to the accelerator.* The inputs typically come from sensors, such as voice waveforms for keyword spotting or heart rate monitors for seizure detection. The NUSA location where the inputs are stored is determined beforehand by the framework.

2. *Control core pre-processing.* External I/O will activate the control core (M0 in this implementation), which may perform some pre-processing. It includes operations that are either not supported in the PE, or are too small that the overhead of invoking the accelerator offsets the benefit of using it. The amount of computation depends on the applications and is decided by the framework; it is sometimes the case that no pre-processing is necessary.

3. *Programming the PEs.* The control core will write the PE instructions to their respective instruction buffers, then activate the PEs. The number and timing of the PEs activated by the control core is configurable and determined by the framework.

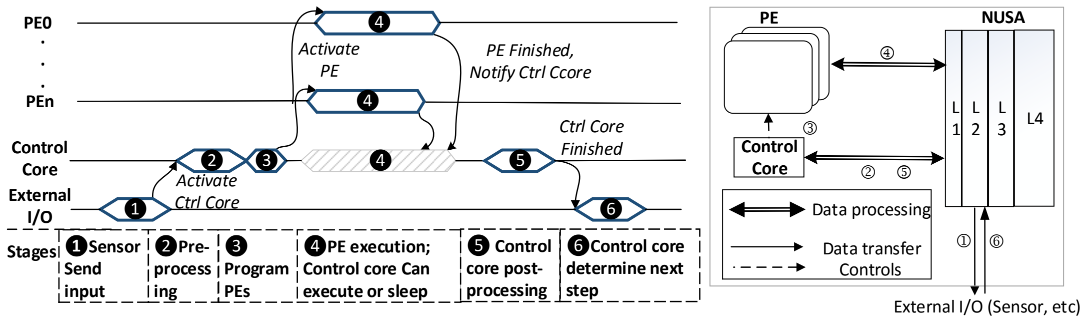


Figure 4.7: Timing and block diagram of accelerator operation

4. *PE execution.* This phase is where the PE executes on the data. The control core may be put to sleep, or perform other work, depending on the need of the application.

5. *Control core post-processing.* Once the PEs finish their execution they notify the control core. Similar to the pre-processing step, the control core then performs some optional post-processing on the results from the PEs.

6. *Final result.* Depending on the result, the control core may decide to wake up the main application core and let it take over, activate some sensors or alarms, or go back to sleep until the next iteration starts.

During one iteration of inputs, stages 2 through 5 can be repeated zero or more times, and can be optionally run in a pipeline fashion. These decisions are decided statically by the post-silicon part of the framework. The use of ping-pong CISC instruction buffers are to overlap PE execution with the programming of the PE for the next CISC macro instruction, further reducing the time that the control core needs to be on.

4.4.3 Instruction set

We target our instruction set towards memory access intensive linear algebra kernels which are common in neural network classifiers and signal processing algorithms.

To accommodate the form factor and energy budget of a wearable device, we lay out three guidelines for our ISA design:

Instructions	Example purposes	Configurable fields
MAC	Matrix-vector / matrix multiplication, dot product	Matrix size, vector size, add offset (Y/N), apply NLI (Y/N)
FFT	FFT, DFT, elementwise operation	FFT points, data size
NLI	Non-linear function using a piecewise linear interpolation	Data size
MOV	Memory copy	Data size

Table 4.2: Accelerator instruction set. For "configurable fields", common fields such as data location or precision are not included.

A) Provide flexible fixed point operation

For these applications, fixed point is enough to provide good precision [28, 34]. Avoiding expensive floating point operations can save ALU area, memory space, and computation power. In addition, variable data precisions (6 choices from 6 to 32 bits) are supported for the user to exploit trade-offs between precision and energy efficiency.

B) Minimize the accelerator logic

The complexity of the decoding and issue logic in the accelerator should be minimized for energy efficiency. This can be done by providing extra information in the instruction directly. For instance, in one stage of FFT, the number of FFT instances can be calculated as (FFT size / FFT unit size). Instead of implementing the calculation in the accelerator, for example, this number is set as a field in the instruction.

C) Reduce programming effort for the control core

Writing instructions to PEs is the job of the control core (M0 in this chip). If many instructions are required to complete one application, the M0 would be waken up by PEs to be reprogrammed many times. It is beneficial to reduce the number of instructions in order to keep the M0 asleep as long as possible.

The above three guidelines motivate a CISC instruction design with many fields in the instruction. A control FSM in the PE will drive the operation by decomposing the instruction into micro-codes. We extract the important kernels in the target applications and implement them as 192b long CISC instructions for the accelerator, as shown in Table. 4.2.

The **MAC** instruction performs a series of vector dot products, where the size of the vector and number of total vectors are configurable. The most typical use case for a MAC instruction is matrix vector multiplication, which is among the major computations in deep neural networks (DNN). The MAC instruction supports adding an offset vector to the result vector and/or applying a non-linear function (NLI) instruction.

The **FFT** instruction executes a stage of an FFT. A stage means a set of butterfly operations through the entire input vector. The entire FFT operations require $log_2(N)$ stages to complete, N being the size of the input. The data size and the number of stages are specified in the instruction. It operates on complex numbers. By setting the imaginary part to 0, it can also be used for DCT or vector elementwise operations.

The **NLI** instruction executes piecewise linear interpolation[1] for each element in a vector. The parameters, a_i and b_i, for each segment are too long to be specified by the instruction; instead they are stored in a dedicated space in the accelerator and can be programmed by the control core. The NLI instruction can be used to approximate non-linear functions, for instance the activation functions in a DNN.

The **MOV** instruction performs data copy in the NUSA memory. It is more efficient than the control core when copying a large chunk of data, such as merging partial results for a parallel matrix vector multiplication. ALUs in the PEs are not involved in the MOV instruction.

4.4.3.1 Support for convolution

Convolution is an important kernel in the IoT space, being widely used in signal processing and image processing. Two methods on the accelerator are supported to perform convolution:

***A**.) using MAC: convolution as a dot product*

Fig. 4.8 describes this method. We name the two functions to be convoluted as $f(t)$ and

[1]Use $f(x) = a_i x + b_i$ when x belongs to the i-th segment ($x_i < x < x_{i+1}$) to approximate a nonlinear function $f(x)$.

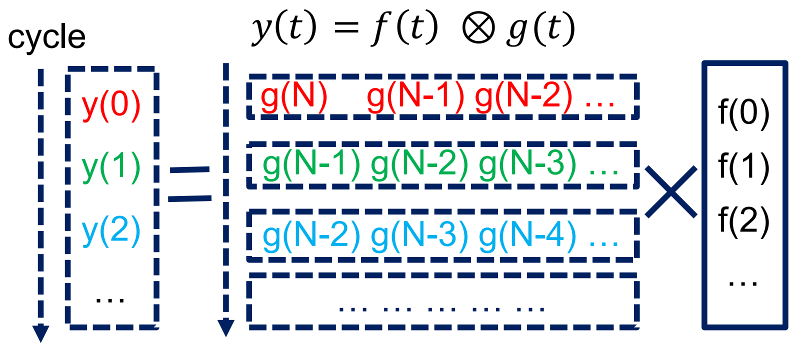


Figure 4.8: Convolution using MAC instruction

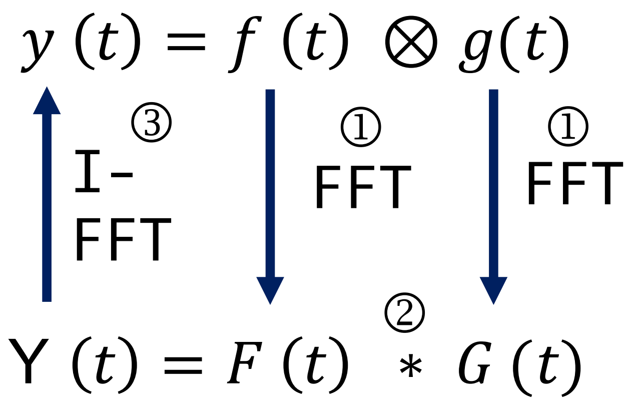


Figure 4.9: Convolution using FFT instruction

$g(t)$, and the result as $y(t)$. We first reverse $g(t)$, and start computing the dot product with $f(t)$ and the reversed version of $g(t)$. The result of the dot product will be the first element of the result $y(t)$. Then we move $g(t)$ by one element, and compute the next dot product along the way. The MAC instruction is used in the actual implementation.

***B**.) using FFT: convolution in the frequency domain*

This is done by utilizing the *convolution theorem*: the Fourier transform of a convolution is the element-wise product of Fourier transforms. Therefore, we can transform the two functions we want to convolve, $f(t)$ and $g(t)$ by using the FFT instruction to transform them to frequency domain. Then an element-wise product of the two functions is performed. Finally, an inverse FFT is used to obtain the result. As for implementation, the FFT instruction can support the FFT and element-wise product, and inverse FFT can be done using forward FFT and data swapping. The process is shown in Fig. 4.9.

The post-silicon framework can decide to use A.) or B.), based on the characteristics of the convolution itself and memory available. In general, B.) will only be beneficial for problems of large data sizes and when abundant scratchpad space is available. Note that though in our evaluation both are supported, currently only B.) is possible on the fabricated chip. This is because the NUSA SRAM on the chip works at a 96-bit word granularity. For A.), access to individual elements in a word is required, which is not currently implemented on chip for the MAC instruction.

4.5 Design evaluation

In addition to the prototype chip, we generalize the architecture by simulating the latency and power consumption with different number of PEs, NUSA memory layouts, and optimal memory access patterns. We compare the performance of traditional CPUs as well as cache, USA, and NUSA based accelerator designs.

4.5.1 Accelerator

In order to obtain a breakdown of the energy consumption of individual components, the design is simulated using the final RTL of the fabricated chip in Synopsys VCS tool, and the Value Change Dump (VCD) files generated from the VCS simulation are then fed into Synopsys Prime-Time PX for energy consumption estimation. The accelerator, M0 core, and AHB bus all run at 10 MHz. When the number of PEs simulated is larger than 4, the PE energy is extrapolated from the energy measured for 0-4 active PEs. For scratchpad SRAM memory, access energy is obtained from SPICE simulation; for caches it is obtained from CACTI [66].

4.5.2 Baseline

We compare the accelerator against Cortex M0/M3/M4 cores and a TI TMS320VC5509A low-power fixed-point DSP [67]. The latency of M-class cores are simulated using Keil uVision 5 microcontroller development kit [68]. CMSIS math library is used for computational intensive kernels. The latency of TI DSP is simulated using Code Composer Studio (CCS) integrated development environment (version5) with built-in simulator for the DSP under test [69], and C55x DSP Library [70] is used for kernels. All ARM and DSP code is compiled with -O2, -Otime flag, optimized for execution time. The frequencies for all the three cores are set at 10MHz, the same as the accelerator; the frequency for DSP is set at 108MHz, the lowest as shown in the datasheet, but the power number is scaled to 10MHz in the results for a fair comparison. Energy consumption for the baseline cores is estimated by multiplying latency with power consumption numbers, which is obtained from datasheets and simulation using final chip RTL.

Applications	Example Device	Kernels
Keyword spotting	Smart Watch, Smart Glass	FFT, matrix vector multiplication, DNN
Face detection	Smart Glass	CNN or DNN
Seizure detection	Activity Tracker, Fitness Tracker	Quadratic classifier
Wake-on gesture (User Identification)	Activity Tracker, Smart Watch	Support vector machine (SVM)

Table 4.3: Always-on applications evaluated in this work

4.5.3 Benchmarks

We evaluate both micro-benchmarks, which are one or a combination of several instructions, and full applications. The full applications we evaluated in this work is presented in Table. 4.3.

Keyword spotting [28]: this benchmark consists of two phases: a) *pre-processing phase*, where the MFCC (Mel-frequency coefficients) for a 25ms voice sample frame is calculated. The major computations involved are a 512-pt FFT and a 400*24 matrix vector multiplication. b) *classification phase*, where a set of the MFCC results are pass to a DNN with 3 fully-connected layers. The size of the layers are 408×384, 384×384, and 384×13. The DNN identifies the voice sample as one of the 10 keywords, OOV (out-of-vocabulary), or silence. The classification results for each voice sample frame is then smoothed with a moving average window to generate a final result.

Face detection: A small image is passed to a neural net to determine if it contains a face. Two versions of face detection are evaluated in this chapter: a) *DNN* [34]: it uses a DNN consisting of 2 fully-connected layers. The sizes of the layers are 16×1032 and 2×16. b) *CNN* [35]: a CNN is used, including convolutional layers, subsampling layers, and fully-connected layers; 2 for each type. We implement it by laying out the 2-D patch to a 1-D array, then treat it as a typical convolution .

Seizure detection [30]: The features of an electroencephalogram (EEG) signal is passed to two quadratic classifiers to determine if the EEG is normal, interictal, or ictal (epileptic). The major computations include a dimension reduction, and one or two quadratic classifiers, which is implemented as matrix-matrix multiplication with the size of [25,2] and [300,25], and a matrix-matrix-matrix multiplication with the size of [2,300], [2,2], and [300,2].

User identification [71]: The users wrist movement will be sampled and passed to a SVM classifier to determine if it is the movement pre-defined by the user. The SVM is implemented as matrix multiplication with the size of [48:12] and [1:48], and a Vector element-wise multiplication with a vector size of 12.

4.6 Results

In this section, we present the results on a set of micro-benchmarks to demonstrate the benefit of using NUSA and the proposed framework. We also simulate several end-to-end applications, and compare the results of the different accelerator designs with that of general purpose embedded cores and DSP.

4.6.1 Pre-silicon NUSA design

We compare 6 memory layout scenarios in three categories: cache, USA, and NUSA. Associativity of 2 is chosen for cache design.

USA is selected as the baseline. It represents the case where accesses, on average, have the same access energy. If the requests are evenly distributed to the memory system, half the accesses will be below the average and half above. For NUSA, designs of 1-level, 2-level, and 4-level are used as three representative memory layout scenarios. The level number represents the maximum number of different bank sizes in the design. Banks with smaller size are placed closer to the processing elements, and more frequently accessed data are assigned to those nearby banks. In the end, we also evaluate the design with the memory layout chosen by the pre-silicon framework.

Every layout scenario is set to have the same area constraint, determined by the area to hold the data needed for the particular micro-benchmark being evaluated in the baseline USA design. The configuration of USA is chosen as the cross-point on the SRAM trade-off graph (Fig. 4.2), representing a reasonable access energy and size. The unit bank size is 2kB with a density of $0.613\mu m^2$/bit and access energy of 0.061 bit/pJ. For the NUSA designs we equally divide the area for each level to represent a naïve approach to determining how the design should be configured. In the NUSA framework approach, we allow the framework described in Section 4.3 to determine the optimal division of the area for each level, which may not be an equal division of space.

The two micro-benchmarks under test are the MAC instruction and the FFT instruction. The dimension for the matrices in the MAC instructions are chosen as 800, 400 and 150, and are combined as shown on the X-axis of Fig. 4.10. FFT sizes are chosen as 512, 256, and 128, and are combined with the MAC instructions as shown in Fig. 4.11.

In Fig. 4.10,Fig. 4.11 we show the results. Each set of bars represent a set of micro-benchmarks, and the bars within a set represent the different designs. First of all, the cache clearly has significant overhead, on average around 40% over a scratchpad memory architecture, even assuming a 100% hit rate. This is because the tag array and comparison logic are relatively large for small bank sizes. Second, NUSA demonstrates advantages over USA designs in all scenarios. As shown in the graphs, the framework can find an optimal NUSA design given the constraints. On average the naïvely allocated NUSA designs achieve a $1.67\times$ improvement compared to USA. The framework further improves the NUSA design by $1.26\times$, netting a total improvement of $2.1\times$.

4.6.2 Multi-application scheduling

It is common for wearable devices to have multiple always-on applications. Therefore, in addition to single applications in the last section, it is meaningful to explore how the framework would work in terms of energy and area for multiple applications.

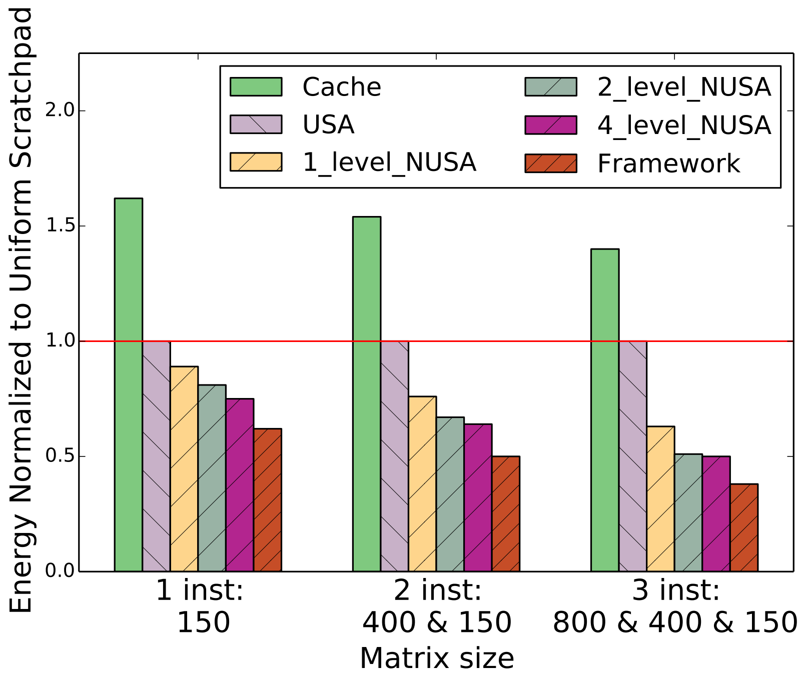


Figure 4.10: Energy reduction with NUSA framework: MAC instructions

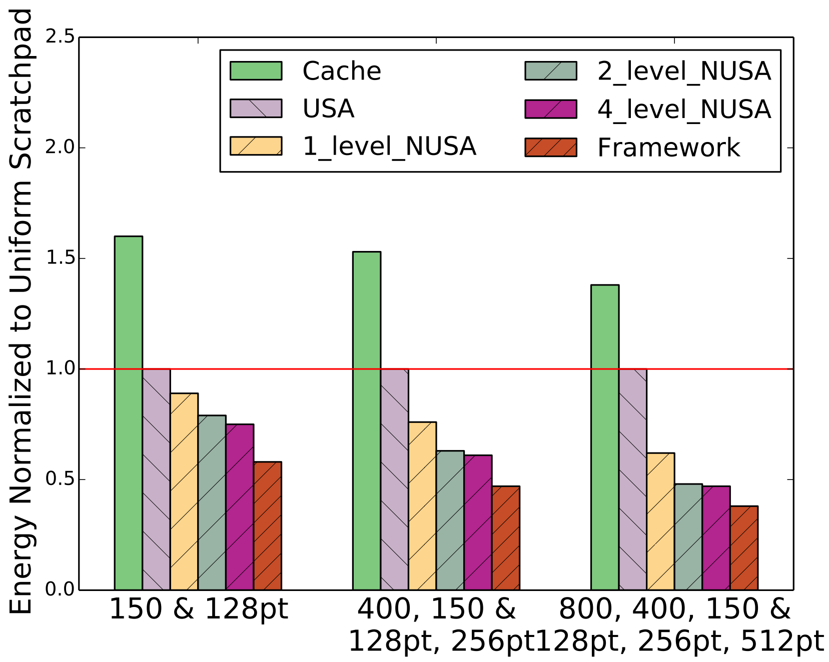


Figure 4.11: Energy reduction with NUSA framework: mixture of MAC and FFT instructions

Two wearable device targets are evaluated, both of which contains a mixture of two always-on applications. Fig. 4.13 shows the result for a smartwatch, with Key Word Spotting (KWS) and Seizure Detection (SD); Fig. 4.12 for a smartglass, with Key Word Spotting (KWS) and Face Detection (Face). For each device target, we evaluate five different mixture of the two workloads, as represented by the first 5 sets of bars. The 6th set of bars represent the average energy consumption of the five different workload scenarios, and the 7th set of bars represent the area of different accelerator architectures, which are shown in different colors. Six different accelerator architectures are evaluated. Uniform memory and generic NUSA represent designs that are agnostic to applications. Dedicated accelerators (KWS, SD, and Face accelerator) are tailored towards one specific application. Framework targets for both applications. A 50%/50% ratio is chosen here, but it can be changed based on the actual usage of the wearable device.

The results demonstrate that the Framework works well under all workload scenarios within the area constraint. On average, it achieves 29% energy reduction over the baseline. On the contrary, using two dedicated accelerators only achieved 2% more energy reduction at the expense of exceeding the area constraint. Using only one of the accelerator will lead to sub-optimal energy consumption, especially if the accelerator cannot holds the full content of the application and needs to fetch data from outside of the accelerator. For instance, for the case of Smartwatch, the accelerator for Seizure Detection cannot hold all the weights for KWS and leads to extra energy due to inefficient memory access. [2]

4.6.3 Post-silicon NUSA assignment

In this section, we present an example of how applications are mapped to NUSA memory banks in Fig. 4.14 and Fig. 4.15. The example application is Keyword Spotting (KWS). The main kernels in KWS include: FFT operation, Mel-scale frequency filter (MSFF) for

[2]We conservatively assume all the data can still be hold in on-chip generic SRAMs; off-chip memory access can further increase energy significantly.

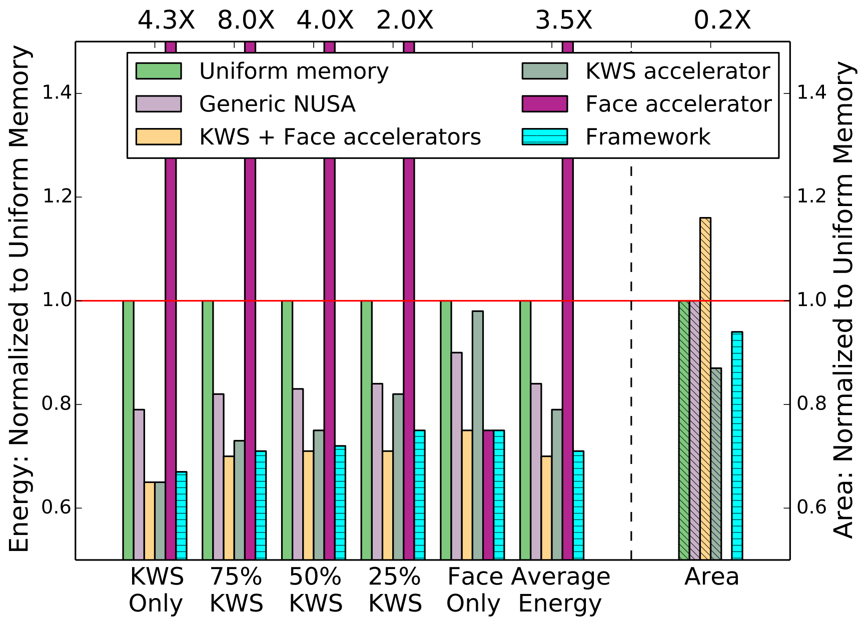


Figure 4.12: Multi-application scheduling. Smartglass: Key Word Spotting (KWS) and Face Detection (Face)

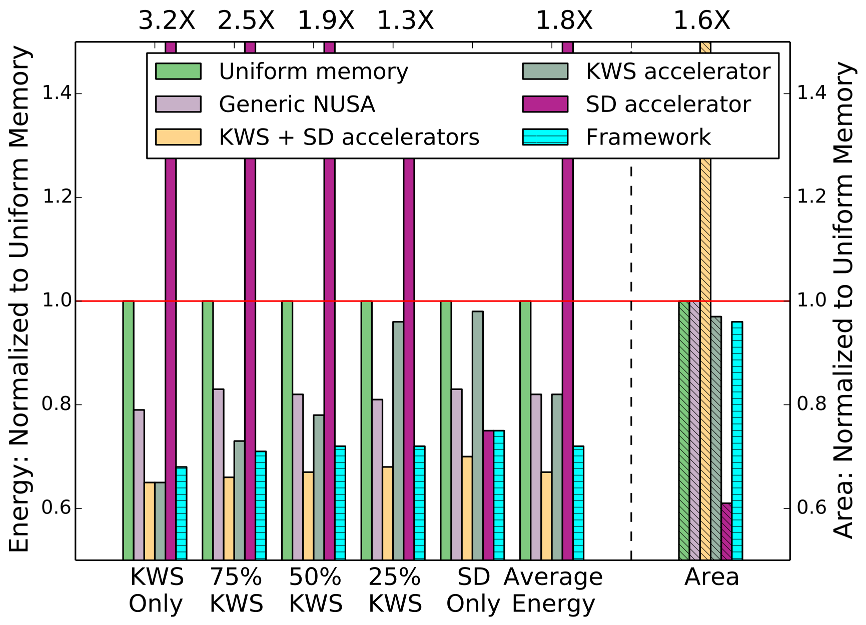
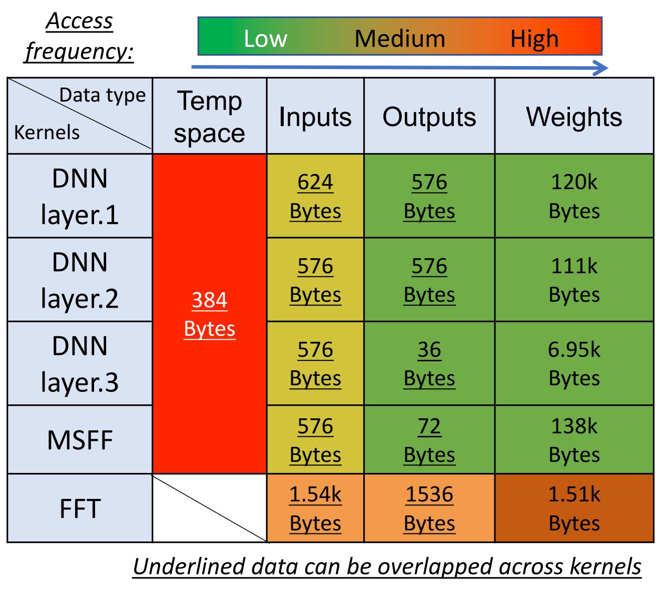


Figure 4.13: Multi-application scheduling. Smartwatch: Key Word Spotting (KWS) and Seizure Detection (SD)

pre-processing, and a Deep Neural Network that consists of 3 fully connected layers (FCL) for classification. Both MSFF and FCL are implemented using the MAC instruction, while the FFT has a dedicated CISC instruction. These kernels run sequentially.

First, the characteristics of the application is analyzed by the post-silicon framework. The first step determines the number of active PEs for each kernel: all 4 PEs are active for DNN FCL 1 and 2; 2 PEs are active for MSFF; and only 1 PE is active for the DNN FCL 3 and FFT. Then, the second step reserves a temporary space for MAC instructions, with a size equals to the smallest banks in the accelerator NUSA (L1 SRAM). With that, we analyze the data elements for the application, as shown in Fig. 4.14. The matrix vector multiplication kernels have four data elements: temp space (for the tiling approach described in Sec. 3.2.2), input vector, output vector, and weights (the matrix). For FFT, there are inputs, outputs, and weights (twiddle factor for FFT butterfly operation). Weights and twiddle factor are the same across different iterations, so they should not be overwritten across different kernels. Except these two, all other data elements can be overwritten and their memory space can be shared among kernels.

Next, based on the usage characteristics, data are mapped to NUSA with the target of lowest total access energy. Fig. 4.15 demonstrates the mapping for FFT, MSFF, and DNN FCL 3. The mapping of DNN FCL 1 and 2 is simpler and omitted due to space limitation. Data that are accessed most frequently, including temp space and inputs, are generally assigned to closer SRAM banks. The temp space, inputs and outputs space for FFT in this mapping is shared with DNN FCL 1 and 2. The gap in PE2, PE3, PE4 exists because FCL 1 and 2 inputs and outputs occupy larger space than FCL 3 and MSFF. Far-away banks are used to store infrequently accessed data elements, mostly to store matrices for matrix vector multiplication kernels.

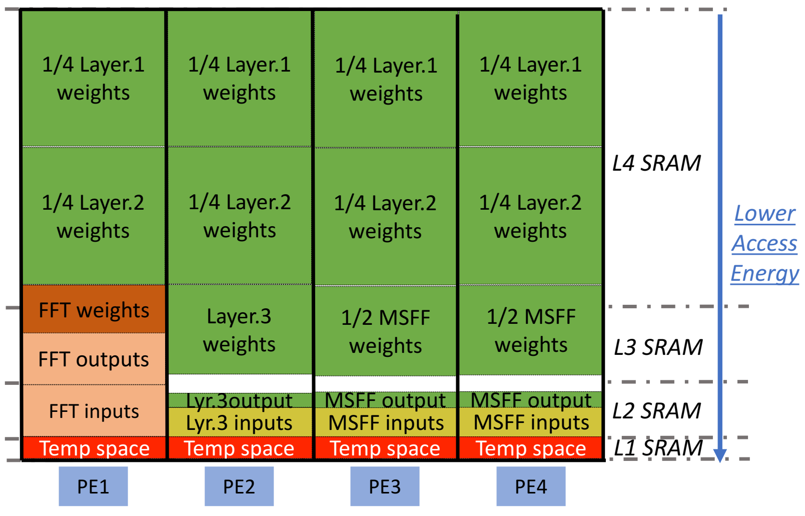


Access frequency: Low Medium High

Kernels \ Data type	Temp space	Inputs	Outputs	Weights
DNN layer.1	384 Bytes	624 Bytes	576 Bytes	120k Bytes
DNN layer.2		576 Bytes	576 Bytes	111k Bytes
DNN layer.3		576 Bytes	36 Bytes	6.95k Bytes
MSFF		576 Bytes	72 Bytes	138k Bytes
FFT		1.54k Bytes	1536 Bytes	1.51k Bytes

Underlined data can be overlapped across kernels

Figure 4.14: NUSA assignment for keyword spotting: data analysis



Memory layout not exactly to scale

Figure 4.15: NUSA assignment for keyword spotting: data assignment

4.6.4 Post-silicon runtime scheduling

In Fig. 4.16 and 4.17, we present the results of the runtime scheduling (post-silicon) part of the framework. Specifically, we look at an accelerator running MAC instructions with 16 PEs.

Fig. 4.16 presents the execution latency. Each set of bars represent the latencies of a certain matrix size for a matrix vector multiplication in a MAC instruction, while the individual bars represent activating different numbers of PEs within the same matrix size. The execution latency is further broken down into two parts: the bottom part shows time spent on the control core and the top part shows time in PE computation. In general, computation time (latency on the PE) reduces with more active PEs; on the other hand, communication time (latency on control core) increases because a) more PEs need to be programmed and b) after PE computation finishes, more time needs to be spent on gathering all the partial results. Activating more PEs is beneficial for larger sizes of matrices, as shown in the graph. This is because communication latency takes up a relatively less portion for these sizes.

Energy consumption for the same set of benchmarks is presented in Fig. 4.17. Each bar is broken down into three parts: the energy spent on the control core, including the core itself and its compiled SRAM; the energy spent on PE computation; and the energy spent on accessing the NUSA SRAM. NUSA SRAM access energy increases with the problem size, but mostly stays the same regardless of the number of PEs active. The energy consumption of both the control core and the PEs are more complicated; it does not grow monotonically with the number of PEs. This is because static power consumes a considerable portion of total energy spent for the control core and PEs, and static power depends on the total latency. For instance, though the control core latency increases with the number of PEs, the energy spent on the control core may still decrease because the static power decreased along with the total runtime.

In all, these graphs show that the latency and energy can be affected by many factors, and both NUSA and a framework are necessary to reach the optimal design.

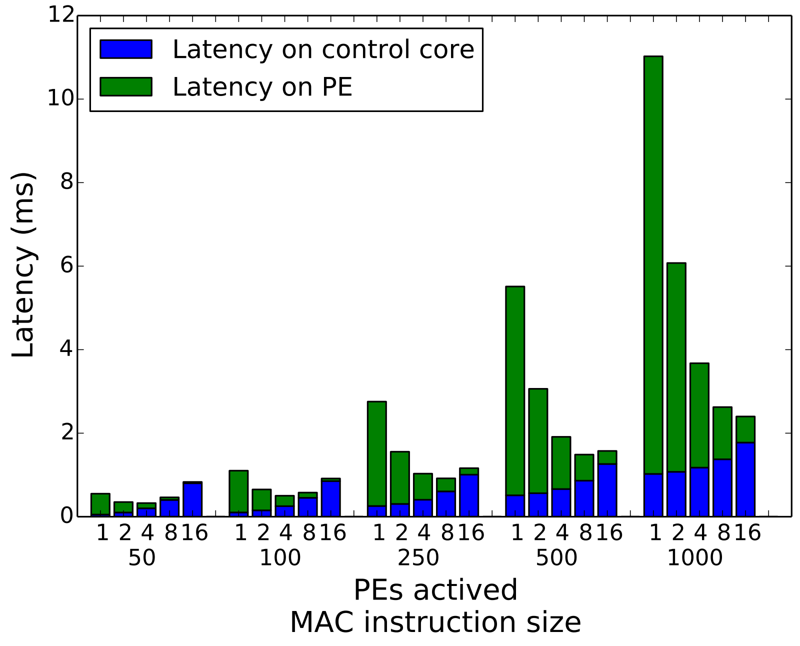


Figure 4.16: Runtime scheduling for MAC instruction: latency for different number of PEs activated

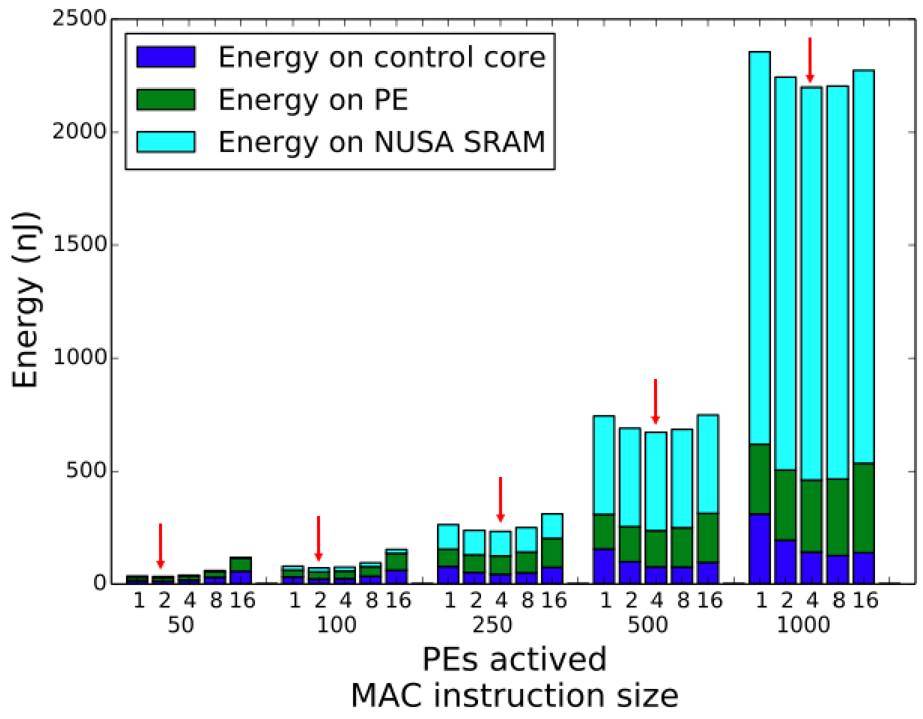


Figure 4.17: Runtime scheduling for MAC instruction: energy for different number of PEs activated

4.6.5 End-to-end applications

In Fig. 4.18 and Fig. 4.19, we demonstrate the result of running the end-to-end always-on applications in Table. 4.3. We compare the accelerator against general purpose embedded cores, specifically ARM Cortex M0, M3, and M4, and a TI low-power fixed point DSP. Experimental setup for these cores are described in Section. 4.5.1 and 4.5.2.

Latency: Fig. 4.18 show the improvement of latency against three M-class cores and DSP. First, applications involving more computation can generally benefit more from the accelerator, because the overhead in launching the accelerator gets amortized for bigger kernels. For instance, keyword spotting achieves higher latency improvement than face detection as it works on a larger DNN, and wake on gesture has little improvement because its data size is very small. Second, applications running mainly matrix-vector multiplication have larger improvement over applications running FFT and matrix-matrix multiplication. There are two reasons for that: 1.) there are more opportunities to optimize matrix-matrix multiplication and FFT for M-class cores and DSP, especially for the M3 and M4. Also, M4 has comparable performance to a low-power DSP because it has good signal processing support. 2.) FFT and matrix multiplication needs more PE instructions to run, which incurs more communication overhead between the control core and the accelerator.

Energy: Fig. 4.19 show the reduction of energy consumption. Applications running mainly matrix-vector multiplication and FFT achieve more energy reduction than matrix-matrix multiplication. This is because matrix-vector multiplication has a more diverse difference in terms of access frequency: vectors are accessed more times than matrices. Even DSP has better energy efficiency than general M-cores, the accelerator still use 10X less energy.

On average, the accelerator achieves an $8\times$ to $28\times$ improvement in latency, and a $10\times$ to $36\times$ reduction in energy-efficiency compared to different M-class cores and a TI DSP.

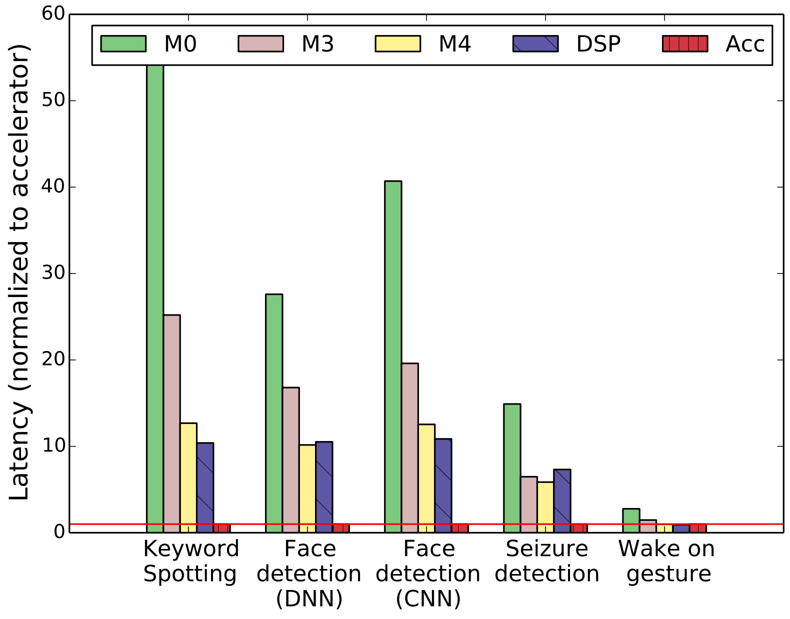


Figure 4.18: End-to-end application: latency

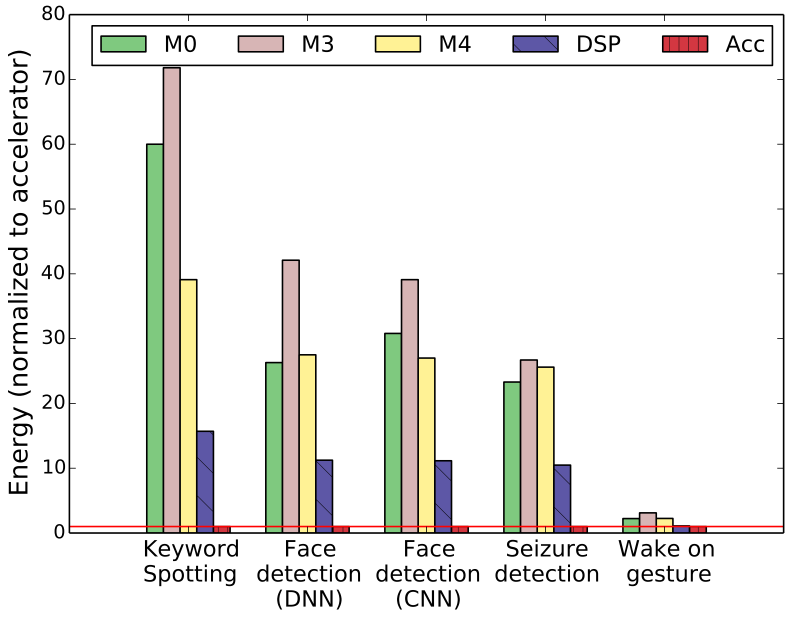


Figure 4.19: End-to-end application: energy

4.7 Related works

Accelerators in wearable devices. Due to the stringent constraint of energy, accelerators are common in smartphones and wearable devices SoCs. Most prevailing accelerators in wearable devices are for communication [81, 20] or computer vision [82]. Recently, deep learning algorithms have been shown to be highly effective in a wide variety of applications. As a result, there has been significant interest in developing accelerators for such algorithms [83, 84, 85]. They emphasize the need of optimizing memory access, which is echoed in our work. There are also approaches for providing a complete DSP [86, 4] or multi-core [87] solution towards wearable device kernels. Our work provides an accelerator design and framework specifically targeted to always-on applications in wearable devices.

Wearable device characterization Various studies have been done on measuring and analyzing wearable energy consumptions, including studies on Android Wear [18, 19, 88], and Google glass [89]. Our work uses these characterizations as a point of departure.

Non-uniform memory allocation Non-uniform memory allocation schemes have been proposed [90, 91, 92, 93, 94]. They are mostly targeted to multi-core server systems. In contrast, our work aims at embedded system with very limited energy budget and scratchpad memories. In particular, we leverage the deterministic behavior for optimal energy efficiency.

4.8 Conclusion

In this chapter, we observed that always-on applications in wearable devices execute in a *deterministic and repeatable fashion*. From this observation, a new accelerator design and framework was proposed that uses a non-uniform scratchpad architecture (NUSA). The framework was used to determine the structure of the NUSA scratchpad, as well as a post-silicon program schedule and data partition. Using the NUSA hierarchy and framework we then presented the design of an accelerator , which was fabricated as a prototype chip. In addition, the chapter further generalized the architecture by evaluating the latency and energy of different designs, comparing it to both a uniform scratchpad based design and general purpose CPU solutions. The results showed, on average, a 2.1× improvement in energy-efficiency for NUSA designs compared to their uniform counterpart. Further results showed that, on end-to-end applications, there was a 10× to 36× total reduction in energy on the NUSA accelerator compared to general purpose M-class cores and DSP.

CHAPTER 5

Conclusions

This thesis focus on characterizing and building heterogeneous mobile platforms in the age of “dark silicon”. With Dennard scaling is reaching its physical limits, technology scaling has enabled increasing on chip integration to the extent that, in the near future, a chip will have more transistors than can be simultaneously powered on within the peak power and temperature budgets. The problem, therefore, is how to best utilize the abundance of transistors in the dark silicon era. This problem is more severe on mobile platforms, especially emerging IoT and wearable devices, due to their strict power envelope and energy budget. Heterogeneous architecture with specialized hardware is an effective way to utilize the limited active transistor count. To guide heterogeneous mobile platform design, analyzing current and near-future mobile workloads and building specialized hardware based on the analysis is essential. To this end, this dissertation starts with a quantitative analysis of current mobile applications on smartphones, followed by an accelerator-based solution for efficient execution of wearable workloads, with an emphasis on machine learning and signal processing kernels.

5.1 Summary

The first part of the thesis focuses on a study of mobile device utilization.

We considered how multi-core processors in mobile devices are being used. We have shown that current mobile applications cannot effectively use a large number of cores, by analyzing a wide range of common mobile applications, and calculated the *Thread Level Parallelism* (TLP) of these applications. The average TLP across all categories is 1.46, which shows that mobile apps are utilizing less than 2 cores on average. Instead, we suggest that a flexible system that can accommodate both high performance and good energy-efficiency is a more preferable choice for current mobile applications. We find out that TLP behavior exhibits peaks and valleys rather than remaining constant. User experience, which is critical for mobile applications, also varies by different application scenarios and different users. A system with the flexiblity to satisfy both high performance and good energy-efficiency for different program phases is a good choice for mobile devices.

The second part of the thesis consists of a low-power accelerator design for always-on applications used in wearable devices.

We observed that always-on applications in wearable devices execute in a *deterministic and repeatable fashion*. This design takes advantage of this determinism by replacing the traditional cache based architecture with a non-uniform scratchpad architecture (NUSA). The framework was used to determine the structure of the NUSA scratchpad, as well as a post-silicon program schedule and data partition. Using the NUSA hierarchy and framework we then presented the design of an accelerator. A fabricated prototype accelerator is also presented as an illustration of the technique. In addition, the chapter further generalized the architecture by evaluating the latency and energy of different designs, comparing it to both a uniform scratchpad based design and general purpose CPU solutions. The results showed, on average, a 2.1$\times$ improvement in energy-efficiency for NUSA designs compared to their uniform counterpart. Further results showed that, on end-to-end applications, there was a 10$\times$ to 36$\times$ total reduction in energy on the NUSA accelerator compared to general purpose M-class cores and DSP.

5.2 Future Directions

This work has mostly focused on the characteristic and design of CPU and ASIC. There are many other important components on a heterogeneous mobile platform that are worth investigating. Mobile GPU would be an good example, as future mobile applications requires rich graphics experience as well as heavy computations. Sample use case can be running augment reality applications with image recognition, with speech recognition running in the background. This problem becomes more interesting with the fast moving trend of deep learning — running some computational kernels in deep learning are faster and more energy efficient on GPGPU (General-purpose computing on graphics processing). Meanwhile, graphics workloads, which directly affects user experience, should not be affected. To investigate and analyze existing and near-future graphics workloads (games, VR) and computational workloads (deep learning), then design hardware and software solutions to co-execute them may be one topic to explore [95, 96, 97, 98].

In this work, we analyzed the components individually, and mostly assume the component the workload will be executing on is pre-determined statically. To dynamically adjust workload assignment to different components, based on the characteristics of the workload, environment, and user need, will be an interesting problem. This issue is briefly touched in Section. 4.3.2, where the proposed framework can decided whether it is beneficial to run a certain workload on the general purpose embedded core, or on the accelerator. A more in depth and broad discussion can be a research topic to explore [99, 100, 101, 102].

BIBLIOGRAPHY

[1] Shafique, M., Garg, S., Henkel, J., and Marculescu, D., "The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives," *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, ACM, New York, NY, USA, 2014, pp. 185:1–185:6.

[2] Goulding-Hotta, N., Sampson, J., Venkatesh, G., Garcia, S., Auricchio, J., Huang, P. C., Arora, M., Nath, S., Bhatt, V., Babb, J., Swanson, S., and Taylor, M., "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro*, 2011, pp. 86–95.

[3] Flautner, K., Uhlig, R., Reinhardt, S., and Mudge, T., "Thread-level parallelism and interactive performance of desktop applications," *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[4] "Ultra Low Power Integrated Platform for Connectivity and Audio/Voice/Sensing," `http://www.chipex.co.il/_Uploads/dbsAttachedFiles/CEVA.pdf`.

[5] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R., "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, 1974, pp. 256–268.

[6] Schaller, R. R., "Moore's Law: Past, Present, and Future," *IEEE Spectr.*, Vol. 34, No. 6, June 1997, pp. 52–59.

[7] Haghbayan, M. H., Rahmani, A. M., Weldezion, A. Y., Liljeberg, P., Plosila, J., Jantsch, A., and Tenhunen, H., "Dark silicon aware power management for manycore systems under dynamic workloads," *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 509–512.

[8] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B., "Conservation cores: reducing the energy of mature computations," 2010.

[9] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A., "Toward Dark Silicon in Servers," *IEEE Micro*, Vol. 31, No. 4, July 2011, pp. 6–15.

[10] Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., and Burger, D., “Dark silicon and the end of multicore scaling,” *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, 2011.

[11] Kanduri, A., Rahmani, A. M., Liljeberg, P., Hemani, A., Jantsch, A., and Tenhunen, H., *A Perspective on Dark Silicon*, Springer International Publishing, Cham, 2017, pp. 3–20.

[12] Halpern, M., Zhu, Y., and Reddi, V. J., “Mobile CPU’s rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction,” *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 64–76.

[13] “Power Management in Intel Architecture Servers,” .

[14] Zhu, Y. and Reddi, V. J., “WebCore: architectural support for mobileweb browsing,” *Proceedings of the 41th annual International Symposium on Computer Architecture (ISCA)*, 2014.

[15] Gutierrez, A., Dreslinski, R. G., Wenisch, T. F., Mudge, T., Saidi, A., Emmons, C., and Paver, N., “Full-system analysis and characterization of interactive smartphone applications,” *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011.

[16] Yang, L., Dick, R., Memik, G., and Dinda, P., “HAPPE: Human and Application Driven Frequency Scaling for Processor Power Efficiency,” *Mobile Computing, IEEE Transactions on*, 2013.

[17] Blake, G., Dreslinski, R. G., Mudge, T., and Flautner, K., “Evolution of thread-level parallelism in desktop applications,” *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA)*, 2010.

[18] Liu, R. and Lin, F. X., “Understanding the Characteristics of Android Wear OS,” *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’16, ACM, New York, NY, USA, 2016, pp. 151–164.

[19] Liu, R., Jiang, L., Jiang, N., and Lin, F. X., “Anatomizing System Activities on Interactive Wearable Devices,” *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys ’15, ACM, New York, NY, USA, 2015, pp. 18:1–18:7.

[20] Chen, Y., Lu, S., Kim, H. S., Blaauw, D., Dreslinski, R. G., and Mudge, T., “A low power software-defined-radio baseband processor for the Internet of Things,” *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 40–51.

[21] Hinton, G. E., “Mapping Part-whole Hierarchies into Connectionist Networks,” *Artif. Intell.*, Vol. 46, No. 1-2, Nov. 1990, pp. 47–75.

[22] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, Vol. 86, No. 11, Nov 1998, pp. 2278–2324.

[23] Socher, R., *Recursive deep learning for natural language processing and computer vision*, Ph.D. thesis, Stanford, 2014.

[24] Hauswald, J., Kang, Y., Laurenzano, M. A., Chen, Q., Li, C., Mudge, T., Dreslinski, R. G., Mars, J., and Tang, L., "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," *ACM SIGARCH Computer Architecture News*, Vol. 43, ACM, 2015, pp. 27–40.

[25] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L., "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, ACM, New York, NY, USA, 2017, pp. 615–629.

[26] Hauswald, J., Laurenzano, M. A., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R. G., Mudge, T., Petrucci, V., Tang, L., and Mars, J., "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers," *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, ACM, New York, NY, USA, 2015, pp. 223–238.

[27] Wilpon, J. G., Rabiner, L. R., Lee, C.-H., and Goldman, E., "Automatic recognition of keywords in unconstrained speech using hidden Markov models," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1990.

[28] Shah, M., Wang, J., Blaauw, D., Sylvester, D., Kim, H.-S., and Chakrabarti, C., "A fixed-point neural network for keyword detection on resource constrained hardware," *Signal Processing Systems (SiPS), 2015 IEEE Workshop on*, 2015.

[29] Fernández, S., Graves, A., and Schmidhuber, J., "An Application of Recurrent Neural Networks to Discriminative Keyword Spotting," *Proceedings of the 17th International Conference on Artificial Neural Networks*, 2007.

[30] Gajic, D., Djurovic, Z., Di Gennaro, S., and Gustafsson, F., "Classification of EEG signals for detection of epileptic seizures based on wavelets and statistical pattern recognition," *Biomedical Engineering: Applications, Basis and Communications*, 2014.

[31] Qu, H. and Gotman, J., "A patient-specific algorithm for the detection of seizure onset in long-term EEG monitoring: possible use as a warning device," *IEEE Transactions on Biomedical Engineering*, 1997.

[32] Ghosh-Dastidar, S., Adeli, H., and Dadmehr, N., "Principal component analysis-enhanced cosine radial basis function neural network for robust epilepsy and seizure detection," *IEEE Transactions on Biomedical Engineering*, 2008.

[33] Güler, N. F., íbeyli, E. D., and Güler, I., "Recurrent Neural Networks Employing Lyapunov Exponents for EEG Signals Classification," *Expert Systems with Applications: An International Journal*, Oct. 2005.

[34] "The database of faces," `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`.

[35] Garcia, C. and Delakis, M., "Convolutional face finder: a neural architecture for fast and robust face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov 2004.

[36] Lawrence, S., Giles, C. L., Tsoi, A. C., and Back, A. D., "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, 1997.

[37] Rifkin, R. and Klautau, A., "In defense of one-vs-all classification," *Journal of machine learning research*, No. Jan, 2004, pp. 101–141.

[38] Boser, B. E., Guyon, I. M., and Vapnik, V. N., "A training algorithm for optimal margin classifiers," *Proceedings of the fifth annual workshop on Computational learning theory*, ACM, 1992, pp. 144–152.

[39] Gao, C., Gutierrez, A., Rajan, M., Dreslinski, R. G., Mudge, T., and Wu, C.-J., "A study of mobile device utilization," *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015, pp. 225–234.

[40] Gao, C., Gutierrez, A., Dreslinski, R., Mudge, T., Flautner, K., and Blake, G., "A study of Thread Level Parallelism on mobile devices," *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[41] "Odroid Wiki," .

[42] "ftrace - Function Tracer," .

[43] Huang, Y., Zha, Z., Chen, M., and Zhang, L., "Moby: A Mobile Benchmark Suite for Architectural Simulators," *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[44] Pandiyan, D., Lee, S.-Y., and Wu, C.-J., "Performance, Energy Characterizations and Architectural Implications of An Emerging Mobile Platform Benchmark Suite: MobileBench," *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[45] Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G., "Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage," *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[46] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T., "RERAN: Timing- and touch-sensitive record and replay for Android," *Software Engineering (ICSE), 2013 35th International Conference on*, 2013.

[47] "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," .

[48] "Variable SMP A Multi-Core CPU Architecture for Low Power and High Performance," .

[49] Khubaib, K., Suleman, M. A., Hashemi, M., Wilkerson, C., and Patt, Y. N., "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp," *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[50] Petrica, P., Izraelevitz, A. M., Albonesi, D. H., and Shoemaker, C. A., "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems," *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[51] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F., and Mahlke, S., "Composite cores: Pushing heterogeneity into a core," *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[52] Carroll, A. and Heiser, G., "An analysis of power consumption in a smartphone," *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[53] Miller, R. B., "Response Time in Man-computer Conversational Transactions," *Proceedings of Fall Joint Computer Conference, Part I*, 1968.

[54] Hayenga, M., Sudanthi, C., Ghosh, M., Ramrakhyani, P., and Paver, N., "Accurate system-level performance modeling and workload characterization for mobile internet devices," *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, 2008.

[55] Van Berkel, C., "Multi-core for mobile phones," *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2009.

[56] Ma, X., Deng, Z., Dong, M., and Zhong, L., "Characterizing the Performance and Power Consumption of 3D Mobile Games," *Computer, IEEE transaction on*, 2013.

[57] Canali, C., Colajanni, M., and Lancellotti, R., "Performance Evolution of Mobile Web-Based Services," *Internet Computing, IEEE*, 2009.

[58] Sunwoo, D., Wang, W., Ghosh, M., Sudanthi, C., Blake, G., Emmons, C. D., and Paver, N. C., "A structured approach to the simulation, analysis and characterization of smartphone applications," *Workload Characterization (IISWC), IEEE International Symposium on*, 2013.

[59] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A., "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[60] Chayong Lee, E. K. and Kim, H., "The AM-Bench: An Android Multimedia Benchmark Suite," *Technical Report, School of Computer Science, Georgia Institute of Technology*, 2012.

[61] "GFXBench: unified graphics benchmark based on DXBenchmark (DirectX) and GLBenchmark (OpenGL ES)," .

[62] "Aurora Softworks: Quadrant," .

[63] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and De Micheli, G., "Dynamic voltage scaling and power management for portable systems," *Proceedings of the 38th annual Design Automation Conference (DAC)*, 2001.

[64] Gao, C., Gutierrez, A., Rajan, M., Dreslinski, R. G., Mudge, T., and jean Wu, C., "A study of mobile device utilization," *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015.

[65] Pannuto, P., Lee, Y., Kuo, Y.-S., Foo, Z., Kempke, B., Kim, G., Dreslinski, R. G., Blaauw, D., and Dutta, P., "MBus: An Ultra-Low Power Interconnect Bus for Next Generation Nanopower Systems," *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, 2015.

[66] Thoziyoor, S., Muralimanohar, N., Ahn, J. H., and Jouppi, N. P., "CACTI 5.1," *Technical Report HPL-2008-20, HP Labs*, 2008.

[67] "TMS320VC5509A Fixed-Point Digital Signal Processor," http://www.ti.com/product/TMS320VC5509A.

[68] "μVision IDE," http://www2.keil.com/mdk5/uvision/.

[69] "Code Composer Studio (CCS) Integrated Development Environment (IDE)," http://www.ti.com/tool/ccstudio.

[70] "SPRC100 TMS320C55x DSP Library (DSPLIB)," http://www.ti.com/tool/sprc100.

[71] Cornelius, C., Peterson, R., Skinner, J., Halter, R., and Kotz, D., "A wearable system that knows who wears it," *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014.

[72] Lin, F. X., Wang, Z., and Zhong, L., "K2: A Mobile Operating System for Heterogeneous Coherence Domains," *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, ACM, New York, NY, USA, 2014, pp. 285–300.

[73] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M., "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, IEEE Computer Society, Washington, DC, USA, 2003, pp. 81–.

[74] Gordon, M. S., Hong, D. K., Chen, P. M., Flinn, J., Mahlke, S., and Mao, Z. M., "Accelerating Mobile Applications Through Flip-Flop Replication," *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, ACM, New York, NY, USA, 2015, pp. 137–150.

[75] Chen, Y., Chiotellis, N., Chuo, L. X., Pfeiffer, C., Shi, Y., Dreslinski, R. G., Grbic, A., Mudge, T., Wentzloff, D. D., Blaauw, D., and Kim, H. S., "Energy-Autonomous Wireless Communication for Millimeter-Scale Internet-of-Things Sensor Nodes," *IEEE Journal on Selected Areas in Communications*, Vol. 34, No. 12, Dec 2016, pp. 3962–3977.

[76] Chen, Y., Lu, S., Fu, C., Blaauw, D., Dreslinski, R. G., Mudge, T., and Kim, H. S., "A Programmable Galois Field Processor for the Internet of Things," *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[77] Li, Z., Dong, Q., Saligane, M., Kempke, B., Yang, S., Zhang, Z., Dreslinski, R., Sylvester, D., Blaauw, D., and Kim, H. S., "3.7 A 1920x1080 30fps 2.3TOPS/W stereo-depth processor for robust autonomous navigation," *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 62–63.

[78] Bang, S., Wang, J., Li, Z., Gao, C., Kim, Y., Dong, Q., Chen, Y. P., Fick, L., Sun, X., Dreslinski, R., Mudge, T., Kim, H. S., Blaauw, D., and Sylvester, D., "14.7 A 288 uW programmable deep-learning processor with 270KB on-chip weight storage using non-uniform memory hierarchy for mobile intelligence," *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 250–251.

[79] Chen, Q., Yang, H., Mars, J., and Tang, L., "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, ACM, New York, NY, USA, 2016, pp. 681–696.

[80] Hsu, C.-H., Zhang, Y., Laurenzano, M. A., Meisner, D., Wenisch, T., Dreslinski, R. G., Mars, J., and Tang, L., "Reining in Long Tails in Warehouse-Scale Computers with Quick Voltage Boosting Using Adrenaline," *ACM Trans. Comput. Syst.*, Vol. 35, No. 1, March 2017, pp. 2:1–2:33.

[81] Lin, Y., Lee, H., Woh, M., Harel, Y., Mahlke, S., Mudge, T., Chakrabarti, C., and Flautner, K., "SODA: A Low-power Architecture For Software Radio," *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 89–101.

[82] Barry, B., Brick, C., Connor, F., Donohoe, D., Moloney, D., Richmond, R., O'Riordan, M., and Toma, V., "Always-on Vision Processing Unit for Mobile Applications," *IEEE Micro*, 2015, pp. 56–66.

[83] Chen, Y.-H., Emer, J., and Sze, V., "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[84] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O., "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[85] Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O., "DaDianNao: A Machine-Learning Supercomputer," *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[86] Wu, M., Iyer, R., Hoskote, Y., Zhang, S., Zamora, J., Fabila, G., Klotchkov, I., and Bhartiya, M., "Design of a low power SoC testchip for wearables and IoTs," *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015.

[87] Tan, C., Kulkarni, A., Venkataramani, V., Karunaratne, M., Mitra, T., and Peh, L.-S., "LOCUS: Low-power Customizable Many-core Architecture for Wearables," *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, ACM, New York, NY, USA, 2016, pp. 11:1–11:10.

[88] Liu, X. and Qian, F., "Measuring and Optimizing Android Smartwatch Energy Consumption: Poster," *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, 2016.

[89] LiKamWa, R., Wang, Z., Carroll, A., Lin, F. X., and Zhong, L., "Draining Our Glass: An Energy and Heat Characterization of Google Glass," *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014.

[90] Carter, N. P., Agrawal, A., Borkar, S., Cledat, R., David, H., Dunning, D., Fryman, J., Ganev, I., Golliver, R. A., Knauerhase, R., Lethin, R., Meister, B., Mishra, A. K., Pinfold, W. R., Teller, J., Torrellas, J., Vasilache, N., Venkatesh, G., and Xu, J., "Runnemede: An Architecture for Ubiquitous High-Performance Computing," *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[91] Guo, Y., Zhuge, Q., Hu, J., Qiu, M., and Sha, E. H. M., "Optimal Data Allocation for Scratch-Pad Memory on Embedded Multi-core Systems," *2011 International Conference on Parallel Processing*, 2011.

[92] Majo, Z. and Gross, T. R., "Matching Memory Access Patterns and Data Placement for NUMA Systems," *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, ACM, New York, NY, USA, 2012, pp. 230–241.

[93] Muddukrishna, A., Jonsson, P. A., and Brorsson, M., “Locality-aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors,” *Sci. Program.*, Vol. 2015, Jan. 2016, pp. 5:5–5:5.

[94] Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A., “A Case for NUMA-aware Contention Management on Multicore Systems,” *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’11, USENIX Association, Berkeley, CA, USA, 2011, pp. 1–1.

[95] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y., “TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments,” *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’11, USENIX Association, Berkeley, CA, USA, 2011, pp. 2–2.

[96] Park, J. J. K., Park, Y., and Mahlke, S., “Chimera: Collaborative Preemption for Multitasking on a Shared GPU,” *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’15, ACM, New York, NY, USA, 2015, pp. 593–606.

[97] Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., and Guo, M., “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing,” *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 358–369.

[98] Yang, H., Breslow, A., Mars, J., and Tang, L., “Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers,” *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, ACM, New York, NY, USA, 2013, pp. 607–618.

[99] Du Bois, K., Eyerman, S., Sartor, J. B., and Eeckhout, L., “Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior,” *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, ACM, New York, NY, USA, 2013, pp. 511–522.

[100] Saez, J. C., Prieto, M., Fedorova, A., and Blagodurov, S., “A Comprehensive Scheduler for Asymmetric Multicore Systems,” *Proceedings of the 5th European Conference on Computer Systems*, EuroSys ’10, ACM, New York, NY, USA, 2010, pp. 139–152.

[101] Suleman, M. A., Mutlu, O., Qureshi, M. K., and Patt, Y. N., “Accelerating Critical Section Execution with Asymmetric Multi-core Architectures,” *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, ACM, New York, NY, USA, 2009, pp. 253–264.

[102] Koufaty, D., Reddy, D., and Hahn, S., “Bias Scheduling in Heterogeneous Multicore Architectures,” *Proceedings of the 5th European Conference on Computer Systems*, EuroSys ’10, ACM, New York, NY, USA, 2010, pp. 125–138.