Heterogeneous resource mobile sensing: computational offloading, scheduling and algorithm optimisation

Petko Georgiev

Corpus Christi College
University of Cambridge

January 2017

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Heterogeneous resource mobile sensing: computational offloading, scheduling and algorithm optimisation

Petko Georgiev

Summary

Important mobile apps such as digital assistants (Google Now, Apple Siri) rely on sensor data and require low-overhead accurate inferences. Although many sensor types are found in devices, the microphone with its high sampling rates arguably poses one of the greatest sensor-related mobile resource utilisation challenges and has similar generic problems faced by all sensors – e.g., efficient background operation, or obtaining accurate inferences in real time. Resource-heavy inference algorithms often employed by audio sensing apps, and the growing number of simultaneously running on-device audio background services, result in app deployments that routinely struggle to reach a day worth of operation alongside other existing services with a single battery charge. In this dissertation we address the challenges presented by emerging audio workloads on smartphone and wearable devices – we enable accurate and efficient multi-app microphone sensing by carefully managing the full range of heterogeneous processors available to high-end mobile platforms.

We devise three general classes of resource optimisation techniques to empower microphone sensing apps with continuous, accurate, and more energy efficient operation. First, we demonstrate how to best leverage the mobile processor hierarchy (CPU and low-power co-processors) in two multi-tier processor offloading scenarios that run multiple continuous audio sensing apps. We show that by smartly partitioning and filtering the sensor processing we can maximise the exposure of these sensor apps to the more power efficient co-processor units and achieve 24 hours of continuous accurate coverage of encountered sound events with a single battery charge. Second, we revisit the typical structure of microphone apps and re-purpose their algorithm implementation to match the hardware constraints of heterogeneous resources – we adopt data parallel versions for the GPU or small-footprint deep learning algorithms to leverage an ultra low-power Digital Signal Processor (DSP) with limited memory and reduced clock frequency. This results in implementations that are not only an order of magnitude faster or more energy efficient than typical CPU alternatives, but also in the case of deep learning are more accurate. Third, we introduce a novel scheduling framework that maximises energy efficiency by dynamically partitioning and distributing concurrent sensor processing tasks across the full range of heterogeneous resources available to modern System-on-Chips (CPU, low-power DSP, GPU, and cloud). The framework pushes the frontiers of sensor processing offloading – it needs only a fraction of the energy required by general-purpose offloaders to build a schedule, and still the schedules it produces have higher power efficiency, and comply with the application deadlines; all the while still preserving inference accuracy.

The insights drawn and the techniques developed in this dissertation can help towards the growth of next-generation context-aware mobile audio apps, able to operate uninterrupted and concurrently without noticeably affecting the device battery lifetime.

Acknowledgments

First of all I would like to Thank my supervisor, Cecilia Mascolo. She has been mentoring and guiding me throughout my PhD, providing me with invaluable advice not only research-wise. It is thanks to her and the freedom I was given to pursue what I felt passionate about, that I have come to discover unprecedented opportunities for career and personal growth. A heartfelt "Thank you" also goes to Nic Lane, who has become a second adviser to me and whose creativity has been continuously inspiring me throughout my PhD. I can never thank him enough for all the constructive feedback he has been giving me. I know that without both of you, none of my academic and professional achievements would have been possible.

I am grateful to have had plentiful opportunities to collaborate with many talented people during my time at Cambridge. I am eternally indebted to Anastasios Noulas for infecting me with a passion for research during my masters while he was co-supervising my projects. I would also like to thank Kiran Rachuri, David Chu, Sourav Bhattacharya, Fahim Kawsar, Sharad Agarwal, Eduardo Cuervo, Stefan Saroiu, Lenin Ravindranath, Victor Bahl, and Alec Wolman who I was lucky enough to collaborate with during my PhD. I am grateful to Alistair Beresford and Andy Hopper for offering me insightful feedback on my work through the years, and also to Andy Rice for helping me truly appreciate teaching through his courses. My sincere thanks to Puneet Jain for helping me co-organise my first workshop. I thank Lise Gough, Helen Scarborough, and Carol Nightingale for helping me with administrative issues throughout my PhD.

My social life as a PhD student has been largely enabled by the friends I made at the lab: Alessandro, Lorena, David, Sandra, Desi, Sarfraz, Chloe, Xiao, Andrea, Vinicius, Miles, Krittika, Dionysis, Sarah, Neal, Denzil, Ella, Zhao, and others. I am delighted to have had the chance to meet all of them, and I am confident that with many of them I will keep being regularly in touch for years to come. I would also like to thank my closest friends at Corpus: Vreeti, Kerstin, Karl, and Rafe, for their social support has been a refreshment from the Computer Science-heavy environment I have been surrounding myself with. Special thanks go to my friends in Bulgaria: Joro, Mateva, Viki, Deni, Eli, Kosta, Svetla, Ivka. Their remote support and encouragement has been an invaluable constant source of energy. They have always been there for me and I will forever remember the times they believed in me even when I doubted myself. Last but not least, I would like to express my gratitude for getting to know my friends during my internship at Microsoft Research: Shubham, Matt, and Lalith. They have truly made my time in Redmond an experience I would love to repeat.

This dissertation has been devoted to my parents. Their sacrifices, support and unconditional love have fuelled my determination to succeed. Always.

Contents

Chapter 1

Introduction

1.1 The mobile sensing revolution

With the proliferation of high-tech gadgets such as smartphones, smartwatches, and head mounted displays, mobile device adoption rates are soaring. Predictions are already in place that soon for many users, a smartphone might be the only computer they use [56]. Smart Internet-connected devices are revolutionising multiple aspects of our daily living – from the way we search for information via voice commands on the smartphone or smartwatch, to how we track our fitness goals via wristbands, and how we interact with our environment via virtual reality. Services integrated in these mobile devices are evolving from being predominantly simple digital assistants and sources of entertainment to becoming productivity boosters and integral parts of our lives.

A key enabler for the rich array of services provided by mobile apps on these devices is extracting user behaviour and ambient context from sensor data. Smartphone sensors such as accelerometer and microphone are used to track the user's mental and physical well-being [163], whereas using the camera for image sensing in virtual reality headsets is a core building block for tracking the user's surrounding conditions [169]. Further, hands-free control via voice commands is widely adopted by digital assistants found in virtually every type of mobile device (smartwatches, smartphones, head mounted displays): Google Now [22], Microsoft Cortana [33], and Apple's Siri [11] are among the most ubiquitously used audio sensing apps. Allowing for such functionality requires continuously monitoring the ambient environment via the microphone for the detection and recognition of speech. Fitness tracking apps [28, 32, 1], on the other hand, analyse data extracted from motion sensors (accelerometer, gyroscope) to tag activities such as walking, running, cycling, and more. Even though sensor systems and apps are extremely diverse, a unifying element is their need to infer user context and activities from sensors accurately and efficiently.

Mobile Device	Type	Weight	Lifetime	Battery	Inferences Made
Microsoft Band [32]	smartwatch	60g	48 hr	200 mAh	Heart Rate; Sleep; Exercise Routine; Jogging; Hiking; Cycling
Jawbone Up3 [28]	wristband	29g	168 hr	38 mAh	Heart Rate; Sleep; Running; Steps; Calories; Automated Fitness Activity Recognition
LG G Watch R [29]	smartwatch	62g	23 hr [8]	410 mAh	Heart Rate; Sleep, Physical Activity; Running; Steps; Voice Commands; Calories
Google Glass [136]	VR headset	36g	6 hr	570 mAh	Augmented Reality; OCR; Voice Commands

Table 1.1.1: Comparison of recent and/or popular wearables available commercially.

1.2 Energy issues

Despite the proliferation of wearable consumer electronics and ubiquity of smartphones, battery issues are still prominent among mobile users. In Table 1.1.1 we survey the battery form factor and range of inferences for some of the popular wearables shipped in recent years. The LG G Watch, for example, barely manages a day's worth of operation with regular light application usage. The lifetime numbers reported for this and the other wearables here are examples of predominantly light application usage or standby, unfortunately. For power users, or any more complicated scenarios that rely on inferences from higher data rate sensors such as the microphone or camera, the depletion of the battery reserves would be significantly faster. For instance, using the Google Glass for face detection or video capture completely drains the battery in just under 45 minutes [136].

The tracking of one's activities in mobile sensing application scenarios requires a fairly continuous stream of information being recorded. Among mobile devices, most smartphones already boast a quad-core or octa-core processor, therefore, computation is increasingly less of a constraint on these commonly used devices. However, the biggest limiting factor for mobile apps that depend on sensor data is still the inadequate battery capacity of phones and wearables. This problem is exacerbated in the case of sensors such as the microphone or camera, due to their high data rates and complexity of sensor processing algorithms. Further, as more of these sensing apps become deployed on a single device, the high computational requirements of simultaneously running complex sensor processing algorithms add to the energy burden. As a result, duty cycling strategies are commonly adopted, but the side effect is a reduced coverage of captured sensor events, thus often impacting inference accuracy.

1.3 The rise of heterogeneous processors

Recently, a new architecture has been adopted by many mobile SoC manufacturers whereby an additional low-power processor, typically referred to as a co-processor, is added alongside the more powerful primary processor. The aim of this additional low-power processor is to offload computational tasks such as sensor sampling and processing (microphone, camera, accelerometer). For example, the Apple iPhone 7 is equipped with an M10 motion co-

processor [9] that collects and processes data from the accelerometer and gyroscope even when the primary processor is in sleep mode. Similarly, the Motorola Moto X [36] is equipped with a "contextual computing processor" for always-on microphone sensing but is limited to recognising only a narrow set of spoken voice commands.

As the SoCs in mobile devices evolve they are squeezing in an increasingly wide range of different computational units: Graphics Processing Units (GPUs), low-power CPU cores, multi-core CPUs. Even the Android-based LG G Watch R [29] includes a Snapdragon 400 [45] that contains a pairing of a Digital Signal Processor (DSP) and a GPU in addition to the dual-core CPU. Each processor has its own resource profile when performing different types of computation. This creates different trade-offs for them to execute portions of mobile sensing algorithms, depending on their complexity, frequency of execution, exposure to data parallelism or other characteristics. This high processor diversity is relatively recent for mobile devices. *A key challenge in leveraging such a heterogeneity of processor units is effectively managing their computational resources to support the workload of a broad range of sensing apps and inference types.*

Existing use of these alternative processors in mobile apps is mostly exploratory [175, 160] or limited to narrow use cases [140]. Apart from the relative recency of this heterogeneous hardware revolution, another major deterrent towards the more ubiquitous adoption of such units in apps so far is redesigning the software stack of algorithms to match the hardware characteristics of these units. A co-processor such as the Qualcomm Hexagon DSP [43], for instance, delivers ultra low-power operation with an architectural design based on VLIW (very long instruction word) – execution on this highly power efficient unit is partially constrained by a fairly limited amount of runtime memory and reduced clock frequency. To make the most out of the DSP, algorithms need to be either redesigned or carefully partitioned so that lighter always-on stages of the sensor processing remain on the DSP.

Mobile GPUs, on the other hand, have traditionally been used in graphics applications that perform image processing, but their desktop counterparts have proven immensely useful in a big variety of scientific computations including the training stages of many machine learning algorithms. Taking advantage of this resource on modern mobile hardware for sensing scenarios would entail building data-parallel programs which requires a shift from conventional thinking to effectively employ a different programming paradigm. Using heterogeneous hardware often leads to a fragmented algorithm execution, with some parts executed on the GPU, while others run on the CPU or DSP, which brings additional complexity in code maintenance. There is a growing demand for unifying frameworks that 1) allow a single algorithm implementation to be compiled for or deployed across different types of units; 2) understand both sensor algorithm semantics and hardware characteristics to provide specialised heterogeneous implementations that maximise resource efficiency. APIs and frameworks so far remain narrow and inflexible.

To sum up, we are witnessing unprecedented advances in mobile processor technology, and increased quantity and complexity of context-aware apps relying on sensor data, but

battery capacities still remain comparatively limited. To guarantee rewarding mobile user experiences with minimal downtime due to energy drains, there are several core issues that need to be tackled in this mobile context. How do we efficiently support a wide range of concurrent sensing services? Can we guarantee application response time will be sufficiently low without overburdening the battery? How can we use the heterogeneous processors in the quest for lowering the energy barrier of sensing apps? The work presented in this dissertation is a step towards addressing these concerns.

1.4 Mobile audio sensing challenges

The apps enabled by the microphone have attracted increasing attention in the mobile landscape as audio data allows a wide variety of deep inferences regarding user behaviour. Examples are song recognition [13], speaker identification [140], emotion recognition [166, 141], gender estimation [81], speaker counting [189], conversation analysis [130], voice commands [70], ambient sound analysis [142, 143]. Natural language interfaces are possibly the most popular class of microphone-enabled services deployed on mobile devices – they are adopted by digital assistants such as Google Now, Apple Siri, Microsoft Cortana, and Amazon Echo [4]. In this dissertation we primarily focus on optimising resource use for this rich class of sensing apps, but also explore their interaction with other sensing services as a secondary goal in some of our example systems.

1.4.1 Microphone vs. other sensing modalities

There are several reasons why we primarily focus on audio sensing and only occasionally explore other modalities such as the accelerometer and gyroscope in our work. First, we are more mature in our understanding of how to process microphone data compared to other sensors. This is evidenced by the rich variety of consumer-ready microphone sensing apps that we referred to earlier in this section. Second, the microphone already has the diversity of workloads and characteristics that we expect to eventually appear in other modalities. For example, activity [142] and emotion recognition [166] sourced from audio data have long been present as cornerstone user behaviour inferences the accuracy of which has improved over time. Similar types of inferences obtained from alternative sensors such as WiFi [87, 198] are only recently beginning to appear, and with accuracy that may not yet be sufficient to warrant a widespread adoption among mobile users. The microphone thus serves as a good example modality that will allow us to address many generic problems in the mobile sensing domain: e.g., continuous background processing, obtaining inferences in real time, sharing computational resources among multiple sensor apps. Third, the microphone sampling rates are higher than the ones of other sensors such as the accelerometer, gyroscope, GPS and WiFi. This leads to sensing workloads

that require more demanding algorithms to process the data which additionally challenges mobile resource use on battery-powered devices.

1.4.2 List of challenges

Microphone-based apps strain battery reserves in the following ways:

Continuous sensor monitoring. Detecting sounds or speech in the environment often means that the microphone sensor stream needs to be sampled continuously. For example, recognising hotphrases entails the timely detection of certain keywords (e.g., "Ok Google" [41]): continuous monitoring of speech events is necessary for high precision and low response time. Any duty cycling schemes introduced may undermine the user experience by missing relevant sensor events when the microphone is not being sampled. However, continuous processing is energy heavy because it needs to keep a SoC component constantly in an active state. The CPU has often been approached as the primary source of computation but its active state has a high energy cost. As discussed, more recently low-power co-processors have emerged as the ideal sensor hubs capable of low-cost always-on sensor sampling. But their reduced clock frequency and limited memory constrains the complexity of the types of apps that can be deployed for continuous execution there, often restricting the processing to computations as simple as binary filters [160] (e.g., silence vs. noise, speech vs. ambience).

Higher algorithmic complexity. State-of-the-art classification models used in many audio apps (not only mobile) rely on complex algorithms for higher inference accuracy. Hidden Markov Models (HMMs) [164], Gaussian Mixture Models (GMMs) [61], and lately variants on large multilayer Deep/Convolution Neural Networks (DNNs/CNNs) [82] are often adopted in sound and speech recognition use cases, since they provide superior accuracy performance as evidenced by their numerous wins in audio dataset challenges. However, the most accurate models contain thousands or millions of parameters the evaluation of which easily overwhelms mobile device computational resources [166, 129]. While the latest multi-core CPU varieties may be powerful enough to do processing entirely locally, energy-wise they would incur a high overhead, especially if the classification/inference step needs to be performed repeatedly in order to label different parts of the audio stream. As a result, developers are often forced to sacrifice privacy for energy with cloud offloading, or compromise accuracy by downsizing the models. As for the low-power co-processors, the extent to which they can handle more complex algorithms is a promising area of research but there is much work to be done.

Real-time app response. Many of the microphone services mobile users rely on today (e.g., voice commands, song recognition) require real-time sensor processing in order to provide accurate and timely responses. The higher algorithmic complexity of the more accurate audio classification models and low runtime requirements are at odds with the capabilities of local resources that also need to serve the workload of other actively running

Existing Embedded or Mobile System	**Related Audio Analysis Tasks**
Amazon Echo [4]	Sound-type Recognition
Google Home [20]	Keyword Spotting
	Speech Recognition
Auto Shazam [13]	Sound-type Recognition
	Song Recognition
EmotionSense [166]	Emotion Recognition
	Speaker Identification
SocialWeaver [144]	Speaker Identification
	Conversation Analysis
Text-dependent	Speaker Identification
Speaker Verification [182]	Keyword Spotting

Table 1.4.1: Example embedded audio processing systems each requiring multiple related audio analysis tasks.

apps. As a result, higher inference accuracy is often a first property to be traded off; with this realised though a simpler audio model that has fewer parameters and lower runtime.

Multiple related inference tasks. Audio apps supported by modern embedded devices commonly require *multiple types of related perception tasks* to be performed, as shown in Table 1.4.1. For example, the Amazon Echo device responds to simple home user requests (such as, turn on a light [3]) which requires it to perform multiple learning tasks against a continuously processed audio stream, including: (i) recognise if spoken words are present (and not any other type of sound); (ii) perform spoken keyword spotting (as all commands are begun with the same starting word); and, (iii) speech recognition, along with additional dialogue system analysis that allows it to understand and react to the command. If several of those apps are deployed on the mobile device, they will compete for memory and compute resources, leading to a faster battery depletion. Unfortunately, so far the go-to approach has been to apply optimisations for each app separately despite the fact that mobile resources are shared. Apps are predominantly built with the unrealistic assumption that they will be the only one occupying low-power co-processor resources. Interference among these apps when deployed to operate concurrently leads to increased energy consumption at best, and delayed app responsiveness or malfunctioning at worst.

Interference with other non-audio services. Mobile platforms are inherently multi-app, with audio sensing apps not being the sole service running on the device. A smooth mobile user experience is largely perceived through the use of interactive applications, and more computational resources would ideally be allocated to such apps. Sensor apps often provide auxiliary utility services (e.g., voice commands), and they cannot be expected to be given a large fraction of battery life and resources. Hard problems are 1) devising optimisations that allow sensor apps such as background microphone services to operate within limited memory and runtime, and 2) ensuring that other apps remain fully functional

in the presence of these mixed sensor and non-sensor workloads.

To summarise, the workload generated by audio sensing apps challenges mobile resources with frequent processing, high algorithmic complexity, and lack of cooperation when shared resources are tapped into. We need new optimisation mechanisms that promote cooperation in using shared heterogeneous mobile processors, and that can be applied to multiple concurrently running apps to meet the demands of emerging audio sensing services.

1.5 Thesis and its substantiation

In the previous sections we discussed the rising adoption of mobile sensing apps, the energy issues that accompany them, and the recently observed rich heterogeneity in mobile processor technology. We also presented the resource consumption challenges imposed by the concurrent operation of high complexity audio apps, the increasingly popular class of mobile sensing that relies on microphone data to capture user behaviour and context.

Our **thesis** is that *to meet the workload demands of next-generation audio sensing apps and go beyond incremental energy savings we need to consider optimisation techniques that facilitate concurrent operation of audio algorithm chains and that maximise the shared use of the full range of mobile heterogeneous processors.*

We substantiate this statement by building example audio sensing systems with which we demonstrate the benefits of cross-app optimisation[1], careful management of shared processor resources, and redesigning inference and feature algorithm implementations for improved mobile processor utilisation. In particular, we address the following three research challenges in support of our thesis:

- **Research Question 1**. How can we enable the energy efficient concurrent and continuous operation of multiple mobile audio sensing apps with a high algorithm complexity?
- **Research Question 2**. What type of audio sensor inference algorithm specialisation do we need to perform in order to draw benefits from available heterogeneous mobile hardware?
- **Research Question 3**. How can we automate the process of maximising resource utilisation from multiple concurrent audio sensing apps without compromising app accuracy and responsiveness?

To answer these questions we prototype the design and implementation of audio sensing systems that interleave the execution of multiple concurrent deep-inference pipelines. We introduce optimisations to efficiently leverage the shared use of co-processor and GPU

[1]When we refer to sensor app optimisations we typically imply sensor pipeline optimisations, we use the two terms interchangeably.

resources. To validate the generalisability of our designs we deploy and evaluate these systems on hardware platforms found in off-the-shelf smartphones and wearables. At times we also deploy audio sensing apps together with non-audio algorithms. The next section details our contributions and shows how they answer the research questions raised here.

1.6 Contributions and chapter outline

This dissertation explores the use of heterogeneous mobile processors to lower the energy barrier for adopting state-of-the-art audio sensing apps that operate continuously and concurrently on the device. It addresses the three major research problems outlined in the previous section, and offers three main contributions that map to the various chapters as follows:

Contribution 1: Multi-tier processor offloading

In Chapter 3 we study how to leverage low-power co-processor (LPU) resources in a mobile processor hierarchy (CPU, LPU) to reduce the energy consumption of continuous sensing systems with complex concurrent sensor apps capable of obtaining deep inferences about user behaviour. Our major contribution is the design and implementation of a cloudless integrated audio sensing system, DSP.Ear, that extracts emotions from voice, estimates the number of people in a room, identifies the speakers, and detects commonly found ambient sounds, while critically incurring little overhead to the device battery. This is achieved through a series of pipeline optimisations that allow the computation to remain largely on a DSP found in commodity mobile hardware: admission filters, behaviour locality detectors, cross-pipeline optimisations and selective CPU offloading. Through detailed evaluation of our prototype we show that, by exploiting a smartphone's co-processor, DSP.Ear achieves a 3 to 7 times increase in the battery lifetime compared to a solution that uses only the phone's main processor. In addition, DSP.Ear is 2 to 3 times more power efficient than naive DSP solutions without optimisations.

To validate the importance of this class of co-processor oriented optimisations at scale, we prototype a second proof-of-concept rich-inference sensing system, ZOE, and deploy it on a wearable device. It incorporates the latest innovations in SoC technology together with a custom daughter-board to realise a three-tier low-power processor hierarchy. ZOE performs multiple deep sensor inferences about key aspects of everyday life on continuously sensed data and crucially, achieves this without cloud or smartphone support. We analyse our multi-tier processor design to demonstrate that ZOE remains energy efficient (with a typical lifespan of 30 hours), despite its high sensing workload and small form-factor.

Contribution 2: Processor specialised implementations

In Chapters 4 and 5 we study the energy and runtime trade-offs of adopting sensor algorithm implementations that are purpose-built for the alternative mobile processors GPU and DSP, and that target deep audio inference tasks such as spoken keyword spotting and speaker identification.

First, in Chapter 4 we build and evaluate a deep learning inference engine for the DSP and modify the classification stage of the audio sensing pipelines to use lighter-weight deep neural network (DNN) models that comply with DSP memory constraints. We show DNN use is feasible on the DSP and has a low energy and runtime overhead allowing complex tasks such as emotion detection or speaker identification to be performed in real time while preserving or improving the accuracy. In addition, the DNN-based pipelines gracefully scale to larger numbers of inference classes and can be flexibly partitioned across mobile and remote resources. Last, we develop a multi-task learning framework that builds resource-efficient DNN models with shared hidden layers – these models can obtain inferences simultaneously from multiple audio tasks with no accuracy loss. Most importantly, on average, this approach provides almost a 2.1 times reduction in runtime, energy, and memory for four separate tasks assuming a variety of task combinations.

Second, in Chapter 5 we describe the implementation of a novel GPU optimisation engine that leverages a series of structural and memory access optimisation techniques to allow audio algorithm performance to be automatically tuned as a function of GPU device specifications and model semantics. The techniques we develop eliminate memory access bottlenecks by exploiting the faster but smaller in size GPU caches, and also increase the inherent data parallelism of audio processing tasks by decomposing the algorithms into a series of granular computations. We show that parameter optimised audio routines obtain inferences an order of magnitude faster than sequential CPU implementations, and up to 6.5 times faster than cloud offloading with good connectivity, while critically consuming 3-4 times less energy than the CPU. Unless the network has a throughput of at least 20Mbps (RTT is 25 ms or less), with only about 10 to 20 seconds of buffering audio data for batched execution, the optimised GPU audio sensing apps begin to consume less energy than both cloud offloading and low-power DSP implementations; in addition to the GPU always winning on lower latency.

Contribution 3: Scheduling sensor processing

In Chapter 6 we detail the design and evaluation of a novel sensing algorithm scheduler (LEO) that specifically targets the workloads produced by representative sensor apps relying on microphone (and accelerometer) data. The scheduler maximises the performance for both continuous and intermittent mobile sensor apps without changing their inference accuracy by partitioning the sensing algorithm execution across the full range of heterogeneous resources of high-end phones: CPU, DSP, GPU and the cloud. We build a rich

library of sensing algorithms and exploit sensor semantics to pre-define the pipeline partitioning points that LEO uses in the scheduling process to maximise energy gains. LEO runs as a service on the DSP and adopts a fast heuristic scheduling algorithm with a low overhead (<0.5% of the battery daily) that allows sensor processing offloading to be reactive and so frequently performed. Depending on the workload and network conditions, LEO is between 1.6 and 3 times more energy efficient than conventional cloud offloading with CPU-bound sensor sampling. In addition, even if a general-purpose scheduler is optimised directly to leverage a DSP, we find LEO still uses only a fraction ($< 1/7$) of the energy overhead for scheduling and is up to 19% more energy efficient for medium to heavy workloads.

Contribution 1 is intended to answer Research Question 1, Contribution 2 addresses Research Question 2, and Contribution 3 answers Research Question 3. To conclude, we reflect on the insights provided by this dissertation and explore directions for further research in Chapter 7.

1.7 List of publications

During my PhD I have had the opportunity to work on a variety of projects through fruitful collaborations. This led to publications including workshop and conference papers as well as works under submission, not all of which are directly related to the dissertation. In particular, Chapter 3 is based on Georgiev et al. [92] and Lane et al. [126], Chapter 4 draws from Lane et al. [125, 127], and Georgiev et al. [90], Chapter 5 builds on Georgiev et al. [91], whereas Chapter 6 is based on Georgiev et al. [93].

Works related to this dissertation

- **[92]** P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. DSP.Ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 295–309, New York, NY, USA, 2014. ACM
- **[125]** N. D. Lane and P. Georgiev. Can deep learning revolutionize mobile sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 117–122, New York, NY, USA, 2015. ACM
- **[126]** N. D. Lane, P. Georgiev, C. Mascolo, and Y. Gao. ZOE: A cloud-less dialog-enabled continuous sensing wearable exploiting heterogeneous computation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 273–286, New York, NY, USA, 2015. ACM
- **[127]** N. D. Lane, P. Georgiev, and L. Qendro. DeepEar: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings*

of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM [**Best Paper Award**]

- [**93**] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 320–333, New York, NY, USA, 2016. ACM
- [**91**] P. Georgiev, N. D. Lane, C. Mascolo, and D. Chu. Accelerating mobile audio sensing algorithms through on-chip gpu offloading. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 306–318, New York, NY, USA, 2017. ACM
- [**90**] P. Georgiev, S. Bhattacharya, N. D. Lane, and C. Mascolo. Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2017

Other works

- [**94**] P. Georgiev, A. Noulas, and C. Mascolo. The call of the crowd: Event participation in location-based social services. *CoRR*, abs/1403.7657, 2014
- [**95**] P. Georgiev, A. Noulas, and C. Mascolo. Where businesses thrive: Predicting the impact of the Olympic Games on local retailers through location-based services data. *CoRR*, abs/1403.7654, 2014
- [**123**] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 7–12, New York, NY, USA, 2015. ACM
- [**122**] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In *15th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2016, Vienna, Austria, April 11-14, 2016*, pages 1–12, 2016

Chapter 2

Mobile sensing in the heterogeneous resource era

The previous chapter highlighted the importance of mobile sensing apps and argued how they have evolved into sophisticated services. It also hinted at the energy issues accompanying their deployment on smartphones and wearables. In this chapter we delve into the operational semantics of sensor apps, and survey key challenges in the efficient utilisation of heterogeneous processors as a way to reduce the energy footprint of such apps. We argue that we need new mechanisms and algorithms that make a better shared use of the various resources available to mobile platforms (low-power co-processors, CPU cores, GPUs, and wireless connectivity). Here we outline some key limitations in existing OS and API support for sensor apps.

Chapter outline. Section 2.1 introduces the typical operation of sensor apps and classifies the types of workloads generated by them. Section 2.2 focuses on the specifics of audio sensing apps and describes the most popular algorithmic primitives used to build the microphone sensing pipelines. Next it presents concrete examples of audio apps that have been recently put forward by the mobile sensing literature and that we will use in the systems presented in this dissertation. Section 2.3 gives a background on the latest processor technology outlining the limitations and challenges in using these new types of resources. In Section 2.4 we broadly discuss the limits of the current sensor app support, and shed light on the current role of cloud offloading and co-processors as a tool for computationally offloading sensor algorithm execution. Last, we summarise the chapter in Section 2.6.

2.1 Mobile sensor apps: operation and workloads

The systems developed in this dissertation exclusively target sensor apps that are characterised by their need to sample and interpret sensors present in mobile devices. Here, we

Applications	Sensor	Purpose
RunKeeper [48] Accupedo Pedometer [1]	Accel	activity tracking
Shake Gesture Library [50]	Accel	gesture commands
Hot Keyword Spotting [70]	Mic	voice activated services
Shazam [13]	Mic	song recognition
SocioPhone [130] SpeakerSense [140] Crowd++ [94], SocialWeaver [144]	Mic	conversation context
EmotionSense [166] StressSense [141], MoodScope [135]	Mic	emotion recognition
Siri [11], Cortana [33]	Mic	digital assistants
Waze [55], Moovit [35]	GPS	traffic monitoring

Table 2.1.1: Example Sensing Apps.

describe key varieties of sensor apps and overview typical sensing operations. Table 2.1.1 details examples that are either research prototypes or commercially available.

Anatomy of a sensor app. Every sensor app includes specialised code responsible for sensor data sampling and processing, distinctly different from the app specific logic. Irrespective of purpose, the data flow of sensor processing within these apps share many similarities. Processing begins with the sampling of sensors (e.g., microphone, accelerometer). Feature extraction algorithms are then used to summarise collected data (as a vector of values), the aim is for these features to describe the differences between targeted behaviour (e.g., sleep, or running) or context. Identifying which activity or context is present (i.e., to make an inference) in the sensor data requires the use of a classification model. Models are usually built offline prior to writing a sensor app based on examples of different activities. Although inference is the most fundamental sensor operation performed in sensor apps, considerable post-inference analysis is often needed, such as mining sleep or commute patterns.

Sensor app workloads. Just like conventional apps, different combinations of sensor apps are continuously executed. There are two dominant usage models, *continuous sensing* and *triggered sensing*, each with differing user expectations of responsiveness and exerting differing types of energy strain.

Continuous sensing. Most apps of this type are "life-logging" [89] and are commonly used to quantify daily user routines. They aim to capture data from the phone throughout the day: this demands their sensing sampling and processing algorithms to be extremely energy efficient. In contrast, because of the focus on long-run behaviour they can often tolerate large processing delays; for example, users may review data at the end of the day and the analysis is indifferent to events from the last few minutes.

Triggered sensing. This category includes sensor apps that are initiated either explicitly by

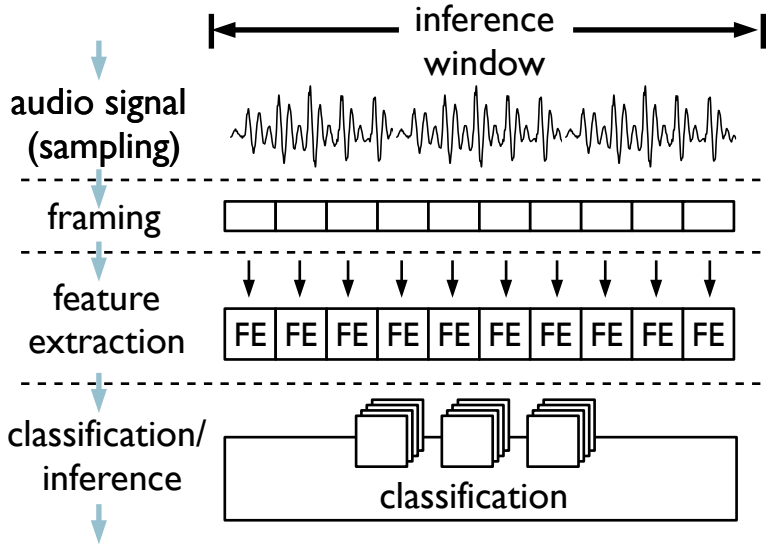


Figure 2.2.1: Audio pipeline structure.

the user or by an event. Examples are apps started by the user to monitor a meeting (Social Weaver [144]) or a workout (Run Keeper [48]). Users often need sensing to be completed during the event in near real-time (e.g., to gauge their effort during an exercise routine to determine if more effort is required). Sensing apps can also be started in response to an event. For instance, a smartphone may need to determine the current ambient context in response to an incoming call to decide if it should be sent straight to voice mail as the user might be driving. These types of sensing apps have a much higher need to be responsive than continuous sensing ones; but as they are often comparatively short lived, energy restrictions may be relaxed to achieve processing deadlines.

A known approach to reduce the high energy coming from the continuous trigger logic evaluation is to instrument apps to use a sensor hub where irrelevant readings are filtered automatically [176]. Companies such as Intel [25], and Apple [27] are embedding a low power micro-controller called a sensor hub in their smartphones. The sensor hub continuously collects sensor data keeping the higher power main processor idle. Subsection 2.3.1 details the operation of this emerging type of low power co-processor and the challenges it presents for use in custom user scenarios.

A key unanswered challenge today is how to maximise the resource efficiency for this diverse and dynamic sensing app workload, while maintaining other mobile device services, e.g., email, music and games, functioning.

2.2 Audio sensing algorithms

Audio sensing primer. Audio sensing apps have the typical sense-transform-classify structure of other sensor apps that categorise human behaviour and activities. We illustrate a common data flow of microphone processing in Figure 2.2.1. The execution begins with the sampling of the microphone where raw data is typically accumulated over a short time window (hundreds of milliseconds up to a few seconds) sufficient to capture distinctive characteristics of sounds and utterances. A typical sampling rate adopted in audio sensing is 8kHz, or the rate of telephone speech, which has proven to be high enough for many app scenarios [143, 142, 140]. At the same time, compared to higher sampling rates, it is less demanding on the device battery.

The next step is subdividing the audio signal into much shorter (e.g., 30ms) segments called *frames* which are subject to preprocessing and feature extraction. Preprocessing is used to accentuate audio signal properties needed by the later stages, and often at this step frame admission control is performed that decides whether further processing is needed. Again, the aim of the features is to summarise the collected data in a way that describes the differences between targeted behaviour or context (e.g., sounds, words, or speaker identity). Identifying the sound class observed in the sensor data in the analysed time window involves the use of classification models. Often the model evaluation applied to the whole window of stacked frame feature data is among the most time consuming stages [143, 142, 140], and as such is a bottleneck of the audio pipeline.

Common features used in audio sensing. In audio processing extracted features are based on either the time or frequency domain. Time-domain features are computed by extracting statistics from the raw audio signal, whereas the frequency alternatives are computed after applying a Fast Fourier Transform (FFT) [64] over the input signal sequence. Features insensitive to volume are preferred in audio processing because they capture acoustic properties that can distinguish between sounds regardless of how loud they are in the environment. Frequency-domain features are typically volume-insensitive which is why they usually dominate the audio algorithm landscape. The two most popular categories of frequency-domain features are Mel Frequency Cepstral Coefficients (MFCC) [88] and Perceptual Linear Predictive (PLP) coefficients [105]. These coefficients are designed to capture properties of human hearing such as vowel perception [105]. MFCCs and PLPs are sufficiently expressive and often are the only type of extracted features in many audio sensing scenarios, most notably speech recognition. Another reason why these coefficients are so ubiquitously used is because they are relatively efficient to compute with an algorithmic complexity of $O(nlogn)$ where n is the number of amplitude values or samples in a frame (e.g., a 30 ms frame with a sampling rate of 8 KHz has 240 samples).

Audio sensing apps characterisation. The apps can be broadly categorised based on the classification model they use for analysis of frame features. By far the two most widely used classification models in the audio sensing domain are Gaussian Mixture Models (GMMs) and Deep Neural Networks (DNNs). Table 2.2.1 gives concrete instances from the

Classifier	Apps
GMM	emotion recognition [166], speaker identification [166, 140], ambient sound classification [143], stress detection [141]
DNN	keyword spotting [70], emotion recognition [100], speech recognition [106], sound event classification [147]

Table 2.2.1: Categorisation of audio sensing apps based on classification model.

audio processing literature (both mobile and general) of their near ubiquitous usage. Most of the extracted features in the pipelines of these apps can be computed relatively quickly, often with latencies that are of the order of tens up to several hundreds of milliseconds, and, as such, are not a serious bottleneck. Undoubtedly optimising all algorithms involved in the pipeline execution is beneficial – yet, for our representative audio sensing app examples larger performance gains can be obtained by targeting the heavier classification stages. Their overhead comes from evaluating the many parameters used in the models for higher accuracy, as well as the poor scaling with the increase in number of classes for the prevailing GMMs. In what follows we detail the operation of these classifiers as some of our key optimisations would be restructuring their execution for higher efficiency.

2.2.1 Gaussian mixture models

The GMM evaluates the probability that certain observations are attributed to the model. In audio sensing the observations are the features extracted from the frames in a window. The GMMs are typically trained offline with one model representing a sound type, an activity class or behavioural category (e.g., a speaker, an emotion, an ambient sound). The stacked frame features are subsequently matched against each model, and the class corresponding to the highest probability is the output of the classification stage. This is a simplified direct application of the Bayes decision rule in the case that the prior probabilities associated with each activity/behaviour class are equal.

More formally, a GMM is a probabilistic model that represents mixtures of probability distributions, it is a weighted sum of M component Gaussian densities. The complete GMM is parametrised by the mean vectors, covariance matrices and mixture weights from all component densities. GMMs are often used in biometric systems, most notably in speaker recognition systems, due to their capability of representing a large class of sample distributions. This use may also be motivated by the intuitive notion that the individual component densities may model some underlying set of hidden classes. For instance, in speaker recognition, these classes can be the speakers broad phonetic events, such as vowels, nasals or fricatives. These acoustic classes reflect some general speaker dependent vocal tract configurations that are useful for characterising the speaker's identity.

It is worth noting that because the component Gaussians are acting together to model the overall feature density, full covariance matrices are not necessary even if the features are

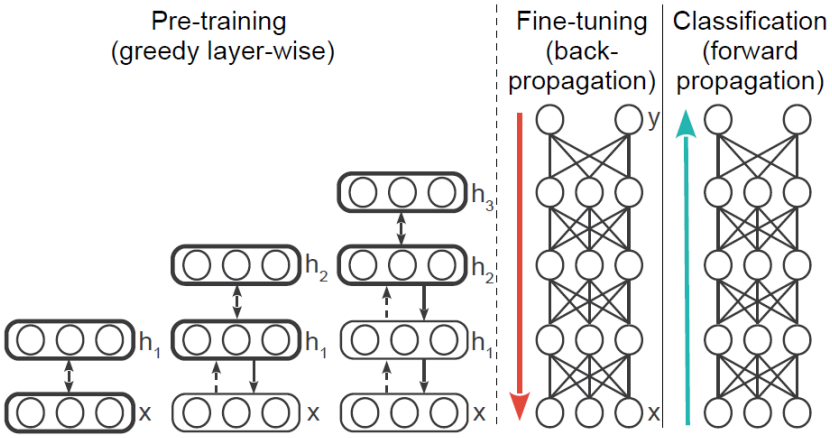


Figure 2.2.2: Example phases of building a Deep Neural Network with 3 hidden layers (h_1, h_2, and h_3), input layer x and output layer y. Shown are the pre-training, fine-tuning and classification phases of a DNN variant called a Deep Belief Network.

not statistically independent [171]. The linear combination of diagonal covariance basis Gaussians is capable of modelling the correlations between feature vector elements. In audio sensing, using diagonal covariance matrices is often preferred in order to reduce the number of parameters [143].

Existing mobile use of GMMs. Until recently GMMs have largely dominated the mobile audio sensing classification landscape [143, 140, 141, 166, 156]. Their effectiveness has been proven in a wide range of sensing scenarios from ambient sound detection [143, 156] through emotion recognition [166], the detection of stressed speech [141] to speaker identification [140, 166]. Often, however, to make local computation on the mobile device feasible and energy friendly, researchers significantly reduce the number of parameters used in the models [140]. Accuracy is usually the first one to be sacrificed through the adoption of smaller models so as to ensure the pipeline execution will not overburden local resources.

2.2.2 Deep learning

Modelling data with neural networks is nothing new, with the underlying technique being in use since the 1940s [146]. Yet this approach, in combination with a series of radical advances (e.g., Hinton et al. [107]) in how such networks can be utilised and trained, forms the foundation of *deep learning* [83]; a new area in machine learning that has recently revolutionised many domains of signal and information processing – not only speech and object recognition but also computer vision, natural language processing, and information retrieval.

Deep Neural Network primer. Many forms of deep learning have been developed with example techniques including Boltzmann Machines, Deep Belief Networks, and Deep

Autoencoders (each detailed in Deng and Yu [83]). Figure 2.2.2 illustrates a common example of deep learning; specifically a Deep Neural Network (or DNN). A DNN is a feed-forward neural network (i.e., the network does not form a cycle) that maps provided inputs (e.g., audio or accelerometer data or features derived from them) to required outputs (e.g., categories of behaviour or context). The network consists of nodes organised into a series of fully connected layers; in-between the input and output layers the DNN contains additional bridging layers (called "hidden" layers). Each node uses an activation function to transform the data/state in the prior layer that in turn is exposed to the next layer. Commonly used node activation functions are drawn from the sigmoid family. A logistic sigmoid $y = \frac{1}{1+e^{-x}}$, for instance, has the property of returning values in the range $(0, 1)$ making it suitable for representing probabilities. Output nodes are an exception, these typically use a softmax function in which the final inference is determined by the node with the largest value (i.e., the conditional probability). See Deng and Yu [83] for more.

A DNN is trained usually in two stages. First, an unsupervised process referred to as "pre-training" is applied to bootstrap hidden node and edge parameters. This stage was a significant breakthrough in deep learning, when it was discovered that this can be effectively done in a greedy layer-wise fashion without labelled data – simplifying the learning when multiple hidden layers are present. This stage, however, may well be skipped when a sufficiently large labelled dataset is available to train the network entirely in a supervised manner. Second, a supervised process occurs, referred to as "fine-tuning", that uses back-propagation algorithms to adjust the parameter values initialised in the previous stage. Parameters are adjusted to minimise a loss function that captures the difference between network inferences and ground-truth labelled data.

Of course, many variations on this training process have been proposed; and similarly DNNs themselves can be utilised in various ways to perform inference. Not only are they used simply as classifiers in isolation (as we do in our study) but they are also chained together to interpret data of differing modalities (e.g., Kahou et al. [117]) , or combined with other types of models (e.g., HMMs, GMMs) to form hybrids (e.g., Han et al. [100]) or act as front-end feature selection phase (e.g., Plötz et al. [159]). Similarly, beyond a basic DNN is a rich family of approaches and machinery such as (the aforementioned) Deep Belief Networks and Boltzmann Machines or others like Convolutional Neural Networks. However, we limit this work to a relatively simple form of deep learning (single DNNs), leaving the exploration of additional techniques for future study.

Existing mobile use of deep learning. As previously described, there are some examples of deep learning being applied in mobile settings. For instance, the speech recognition models used by phones today exploit deep learning techniques (e.g., Deng et al. [57]). Until recently most of these models operated predominantly off-device, in the cloud. Local and hybrid models are now gaining popularity. Some existing application domains of deep learning (such as emotion recognition [100] and others related to audio) are very similar to requirements of mobile sensing and should be able to be adapted for sensor app purposes. Other important sensing tasks like activity recognition are largely unexplored in

Application	Main Features	Inference Model	Frame	Window
Emotion Recognition [166]	PLP [105]	14 GMMs [61]	30ms	5s
Speaker Identification [166]	PLP	22 GMMs	30ms	5s
Stress Detection [141]	MFCC [88], TEO-CB [199]	2 GMMs	32ms	1.28s
Speaker Count [189]	MFCC, pitch [81]	Clustering	32ms	3s
Gender Estimation [189]	pitch	Thresholding	32ms	3s
Ambient Sound [143]	MFCC, Time Domain Features	GMMs	64ms	1.28s
Keyword Spotting [70]	Filterbanks	DNN [106]	25ms	1s

Table 2.2.2: Example audio sensing apps and their properties.

terms of deep learning, with only isolated examples being available (such as for feature selection [159] or non-mobile activity recognition in controlled or instrumented environments [114]). These inference tasks will require more fundamental study as they lack clear analogues in the deep learning literature. Moreover, significant systems research is required to understand how the full range of deep learning techniques can be used locally on mobile devices while respecting energy and latency constraints. For example, mobile OS resource control algorithms aware of how to regulate the execution of one or more instances of deep learning inference are currently missing; as are new deep learning inference algorithms tailored to mobile SoC components like GPUs and DSPs.

2.2.3 Example audio apps

In what follows we detail the operation of representative audio sensing apps outlined in Table 2.2.2 that we implement and use throughout the dissertation as examples. Various combinations of these apps will serve as workload generators for concurrent sensor processing requests. Apart from Speaker Count and Gender Estimation, all other apps use either GMMs or DNNs.

Emotion Recognition. Emotions are an integral part of a user's everyday life. An end-to-end recognition pipeline was proposed by Rachuri et al. [166]. The original version of the app is based on GMMs with diagonal covariance matrices. Each of 14 narrow emotions (Table 2.2.3) is represented by an emotion-specific GMM classifier built by performing Maximum a Posteriori (MAP) adaptation of a 128-component background GMM representative of all emotional speech.

The observations used as input to the GMMs in this app are PLP coefficients [105] extracted from frames over 5 seconds of recorded audio. The audio signal is segmented into 30ms frames with a 20-ms overlap and 32 PLP features are computed from each frame (16 coefficients with their deltas). At runtime, the likelihood of the recorded audio sequence is calculated given each emotion class model and the emotion corresponding to the highest likelihood is assigned. As described by the authors [166], the narrow emotions are grouped together into 5 broad categories that capture sadness, happiness, fear, anger and neutral speech. The final result of the recognition process is thus the broad category to which the classified narrow emotion belongs.

Broad Emotion	Narrow Emotions
Anger	Disgust, Dominant, Hot Anger
Fear	Panic
Happiness	Elation, Interest, Happiness
Neutral	Boredom, Neutral Distant, Neutral Conversation, Neutral Normal, Neutral Tete, Passive
Sadness	Sadness

Table 2.2.3: Emotion categories (adopted from Rachuri et al. [166]).

Speaker Identification. A speaker identification pipeline aims to recognise the current speaker in a room. This app has been adopted by several systems [166, 140] all of which rely on GMMs as the classifier model. The pipeline designed by Rachuri et al. [166] reuses the algorithmic elements introduced in the emotion classification. A 128-component background GMM representative of all available speakers during training is built and MAP adaptation is performed on PLP features from speaker specific utterances to obtain the speaker-dependent models. At runtime, the speaker-specific model that produces the highest likelihood given the audio sequence is identified as the speaker of the audio sample.

Stress Detection. A stress detection app bears resemblance to its emotion recognition cousin with the difference that it is specialised to distinguish between stressed speech and neutral one. A dedicated stress detection algorithm was put forward by Lu et al. in their StressSense app [141]. The input audio signal is divided into non-overlapping 32ms frames and features (Table 2.2.4) are extracted from each frame over a window of 40 frames. Stress inference is thus performed based on acoustic observations from a 1.28-second audio sample. A universal model is built by training two 16-component GMMs representative of stressed and non-stressed speech respectively. At runtime, the probability of the recorded audio sequence given the two models is computed and stress is inferred when the corresponding GMM produces the higher score.

Gender Estimation. Determining the gender of the current speaker is an example audio sensing pipeline that can aid in speaker identification or speaker counting. Previous studies [58] have demonstrated that the most distinctive trait between male and female voices is their fundamental frequency, also known as pitch. Xu et al. [189], for instance, adopt Yin's algorithm [81] to determine the pitch from a 32-ms frame. They compute a series of pitch values from 50% overlapping frames in a 3-second long window and use the mean as a summary statistic over the whole utterance. Gender inferences are made based on the findings of Baken [58] that the average pitch for men typically falls between 100 and 146Hz, whereas for women it is usually between 188 and 221Hz. The algorithm infers a male voice for the whole utterance if the pitch is below 160Hz, reports a female voice if the pitch is above 190Hz and is uncertain in the cases where the value falls between these thresholds.

Speaker Count. An unsupervised counting algorithm to efficiently estimate the occu-

Feature set	Application
pitch [81]	Speaker Count, Gender, Stress
MFCC [88]	Speaker Count, Ambient, Stress
PLP (16 static and 16 delta) [105]	Emotions, Speaker Id
Filterbanks	Keyword
TEO-CB-AutoEnv	Stress
Low Energy Frame Rate [174], Zero Crossing Rate [173], Spectral Flux [174], Spectral Rolloff [131], Spectral Centroid [131], Bandwidth [131], Relative Spectral Entropy [142], Normalised Weighted Phase Deviation [85]	Ambient

Table 2.2.4: Acoustic features.

pancy in a room (i.e., number of speakers) without identifying each speaker separately was proposed by Xu et al. [189] in their Crowd++ app. The algorithm has 2 phases: feature extraction and speaker counting. In the first phase, speech is segmented into 3-second long windows and 20-dimensional MFCC features are extracted from the 32-ms frames with 50% overlap. The speaker counting phase is triggered infrequently and when the conversation is over. A forward pass stage, linear in the number of segments, merges neighbouring segments into clusters represented by the mean values of the MFCC features. Two segments are merged together if the cosine angle between the representative feature vectors of the segments falls below an experimentally identified threshold (e.g., 15°). Thus clusters correspond to audio sequences during which the same speaker is talking. Once this is done, the final stage of the algorithm compares clusters against each other and merges them based on their cosine similarity and the inferred gender of the speaker represented by the cluster. The total number of identified clusters in this final stage is the inferred number of speakers in the conversation.

Ambient Sound Classification. The detection of various ambient sounds in the environment so far has typically been approached with GMM classifiers (e.g., [143, 157]), where each sound category is represented by a separate GMM. The Jigsaw [143] pipeline divides the audio signal into 32-ms non-overlapping frames and extracts features (Table 2.2.4) from each frame in a window of 40 frames. The audio sample length of 1.28 seconds is small enough to account for short-duration sounds, while at the same time being wide enough to capture distinctive acoustic characteristics. The series of frame features in the window constitute the acoustic observations which are then evaluated against the GMM models. The GMMs are usually trained with a standard expectation maximisation (EM) algorithm. This app can be customised to recognise a variety of different sounds depending on the dataset used for analysis.

Keyword Spotting. A small-footprint keyword spotting algorithm was designed by Chen et al. [70], and its aim is to detect a hot phrase spoken by a nearby speaker. The example app is trained to detect an "Ok, Google" command. The audio analysis is performed by segmenting the input signal into frames of length 25ms with an offset of 10ms, i.e.

the frames overlap. Filterbank energies (40 coefficients) are extracted from each frame and accumulated into a group of 40 frames. The features from these frames serve as the input layer to a DNN with as many output layer nodes as there are target keywords (plus an additional sink node to capture other words). The DNN is fully connected and has 3 hidden layers with 128 nodes each. The output of the DNN is raw posterior probabilities of encountering each of the keywords over the last second of data. The DNN feed forwarding is performed in a sliding window with every new frame, resulting in 100 propagations per second (once every 10 ms). After the probabilities are obtained, a post-processing step is applied which smooths the extracted values over a 1-second window. A keyword is detected if the smoothed probability over the audio window exceeds a threshold value.

2.2.4 Datasets

Before moving on we briefly introduce our default datasets used to train the apps above. Throughout the dissertation these will be our primary source of reference, and we will explicitly specify the cases where we change the used dataset.

Emotion Recognition. Similarly to EmotionSense [166] training and testing data can be sourced from the Emotional Prosody Speech and Transcripts library [134]. The dataset consists of voiced recordings from professional actors delivering a set of 14 narrow emotions grouped into 5 broad categories (happiness, sadness, fear, anger and neutral).

Stress Detection. We use a 1-hour dataset of stressed and neutral speech which is a subset of the above mentioned emotions dataset. We regroup the anxiety, panic, fear, cold anger and hot anger narrow emotions into a more general class of stressed speech. The rest of the emotions are flagged as non-stressed speech.

Speaker Identification. 10-minute speech samples are recorded for a total of 22 speakers working in our research group in January 2010.

Gender Estimation and Speaker Count. We extract 24 minutes worth of conversational speech in various contexts from online radio programs. 12 male and 12 female voices are captured in natural turn-taking situations with occasional pauses.

Ambient Sound Classification. The default dataset consists of 40 minutes of various sounds equally split into the 4 categories music, traffic, water and other. We select these sound categories as some examples of commonly encountered types of sounds in everyday settings. The music audio clips are a subset of the GTZAN genre collection [132]; the traffic samples were downloaded from an online provider of free sound effects [16]; the water samples were obtained from the British Library of Sounds [14]; the rest of the sounds were crawled from a subset of the SFX dataset [69].

Speech vs. ambient noise. A common audio sensing use case is having a simple pipeline that makes a binary decision between detected speech or ambient sounds. To train this app, we assemble a dataset that consists of 12 minutes of conversations from online radio

programs and 12 minutes of various ambient sounds including street noise, traffic, water, weather effects, animal pet sounds, machinery, typing, and more. The sources of the ambient noise are as described in the previous paragraph.

Keyword Spotting. The dataset used for this app is the TIMIT continuous speech corpus [129]. It contains recordings of transcribed utterances and sentences from 630 speakers. The samples in this dataset serve as negative examples (non-keyword); we augment the dataset with keyword utterances as positive examples collected from 5 speakers working in our department at the time of collection. The collected keywords vary depending on both where the app is deployed and how it is being used.

Note that for the purposes of our research we are mostly trying to advance mobile resource utilisation and the sample apps we study serve as workload examples. For this reason, we evaluate application accuracy on these datasets for correctness, this gives us a sanity check on our implementation ensuring the app can be used in deployments to fulfil its intended behaviour. Apart from Chapter 4, our concern is being able to replicate the accuracy reported in the mobile sensing literature. Whenever we can we stick to the datasets introduced by the research papers where the original app's design is introduced. In Chapter 4 when we pursue claims on accuracy improvement in addition to runtime and energy performance gains, we employ more comprehensive versions of the datasets with significantly more diverse samples. This allows us to be more confident about how our claims generalise to the studied domains and do not remain within the confines of narrow use cases.

2.3 Mobile heterogeneous processors

As discussed in Chapter 1, mobile hardware has witnessed an unprecedented growth in the variety of heterogeneous processors – many-core CPUs, low-power DSPs, and GPUs. Each of these offers a distinct resource profile that can be taken advantage of by different stages in the execution of an app. Here we highlight the core challenges in leveraging the two most ubiquitously found alternative mobile processors: co-processors and GPUs.

2.3.1 Low-power co-processors

Low-power co-processors for mobile devices are usually narrowly specialised to perform dedicated tasks in an energy friendly manner [140]. Such processing units prove especially handy for continuous sensing where the sensor sampling power consumption costs [160] are several orders of magnitude lower than CPU-driven sensing designs that maintain a wake lock on the main CPU. However, co-processor use often introduces slower computation (due to the units being low-power and generally more limited in capabilities).

Low-power co-processor types

Mobile devices are exposed to a large variety of workloads, from high performance gaming through casual web browsing to background sensing. The realisation that this workload variety is not efficiently serviced by identical general-purpose processor cores resulted in embedded manufacturers introducing specialised low-power cores and co-processors that operate at a reduced clock frequency. Their main purpose is handling lighter computational loads that do not require peak performance such as those found in continuous microphone sensing. An example is the NVidia Tegra 4 SoC family [39] that features a battery saving companion core in addition to the main four CPU cores. This core operates transparently to mobile apps and is used to reduce power consumption when processing load is minimal. ARM's big.LITTLE SoC technology [12], on the other hand, consists of a paring of higher (big) and lower (little) frequency cores. The big.LITTLE software integrated with the mobile OS automatically and seamlessly moves workloads to the appropriate CPU core based on performance needs and the history of a thread's execution. The effect of this load distribution is reportedly 75% reduction in CPU energy for low to moderate performance scenarios [12].

Two very popular co-processor units made available in high-end off-the-shelf smartphone models are iPhone's M10 motion co-processor [9] and Qualcomm's Hexagon QDSP6 [43] of the Snapdragon 800 processor platform. Whereas the former is specialised solely to efficiently offload collection and processing of motion sensing through accelerometer, gyroscope, and compass, the latter excels at low-power multimedia processing. An important distinguishing feature between the two is that, unlike iPhone's M10, the Hexagon QDSP6 supports custom programmability. This is achieved through the publicly released C/assembly-based Hexagon SDK [44] which, to date, has been used for multimedia apps but has yet to be used for sensing. Our main focus of investgation in this dissertation are this type of low-power co-processors (e.g., Qualcomm DSP) that are not transparently managed by the OS software (unlike the Tegra's companion cores and big.LITTLE architecture).

Challenges in using co-processors for sensing

Supporting multiple concurrently running audio inference pipelines on a co-processor such as the Qualcomm DSP is particularly challenging because of the following design space limitations.

Memory Restrictions. The current DSP runtime memory constraints restrict the amount of in-memory data reserved to sensing apps. The Qualcomm Hexagon DSP from the Snapdragon 800 SoC, for instance, has 8MB of runtime memory. Examples of sensing-related data kept in memory are classification model parameters, accumulated inferences or features extracted for further processing. The lack of direct file system support for the DSP imposes the interaction with the CPU which will either save the inferences or perform

additional processing on DSP-computed features. This easily turns into a major design bottleneck if the DSP memory is low, the data generated by multiple sensing apps grows fast, and the power-hungry CPU needs to be woken up from its low-power standby mode specifically to address the memory transfers. Even if memory size increases, limitations will still apply given that multiple apps will most likely need to use it, and given that larger more accurate models are expected to be deployed.

Code Footprint. The DSP restricts the size of the shared object file deployed with app code. The Qualcomm Hexagon DSP, for example, puts a 2MB limit on the compiled binary file size. Deployment issues arise when machine learning models with a large number of parameters need to be initialised but these parameters cannot be read from the file system and instead are provided directly in code which exhausts program space. Sensing inferences performed on the DSP become restricted to the ones the models of which successfully fit under the code size limit.

Performance. The DSP has different performance characteristics. While ultra low-power is a defining feature of the DSP, many legacy algorithms which have not been specifically optimised for the co-processor hardware architecture will run slower. Computations that are performed in real time on the CPU may not be able to preserve this property when deployed on the DSP. The DSP generally supports floating point operations via a 32-bit FP multiply-add (FMA) unit, but some of the basic operations such as division and square root are implemented in software which potentially introduces increased latency in some of the algorithms.

Programmability. The DSP supports only a subset of the symbols commonly found in C programming environments. This limits the range of already developed C/C++ libraries that can be ported to the DSP without modifications such as eliminating file system calls (e.g., printf), replacing memory allocation calls (e.g., new with malloc), or providing explicit software implementations for maths functions (e.g., sine).

2.3.2 GPUs

GPUs are one of the signature heterogeneous processors available in modern SoCs. They are seen not only on the more powerful smartphone platforms, but also on smaller form-factor wearables such as smartwatches based on Snapdragon 400 [45]. Traditionally, mobile GPUs have been used in graphics applications, whereas in server environments GPUs have also been used for scientific computations and machine learning. However, little work has investigated if there will be any energy justified advantages of using them to offload sensor processing on the mobile device.

GPU operation overview

GPUs were originally designed to enable massively parallel computation for image processing and graphics. The core programming paradigm revolves around *data parallelism* where the same computation primitive (e.g., a sequence of operations) is performed simultaneously on different parts of the data. In the graphics domain very often the basic unit that defines the granularity of the parallelism is a pixel [158] – each GPU thread works independently on a separate pixel. It was later discovered that the GPU architecture does not preclude other data parallel computations not necessarily related to graphics from being performed by the GPU. As a result, more generic APIs were designed such as NVidia CUDA [37] and OpenCL [42] that allow a more flexible control over the GPU's resources. A typical GPU program consists of host and device code – the CPU (host) initiates commands submitted to a command queue residing on the GPU (device). Instead the device code contains instructions that will be performed simultaneously by the GPU cores on different parts of some input data. Structurally, a general-purpose GPU program resembles a cascade of commands, some of which prepare data for processing, others submit parallel tasks that will be executed by the GPU, and third read the results from the parallel computation. In the desktop environment, GPU computing has reached a solid level of maturity in terms of accelerating many machine learning algorithms both for training and classification (e.g., Huqqani et al. [112]). Yet, on the mobile side, the GPUs have mostly been used in graphics computations and games [185, 72, 177].

Challenges in using the GPU for sensing

Some of the key challenges in using the GPU for sensing offloading are:

Energy. A limiting factor in frequently approaching the GPU for computation is the energy overhead involved in powering on the unit, ramping up the clock speed, and transferring buffers from the CPU. While mobile hardware manufacturers strive for building more energy-friendly GPUs, the major design highlight of GPUs is their sheer speed and massive data parallelism which requires significantly more power than the low-power coprocessors operating at reduced clock frequency. Continuous sensing is one of the most widespread operational modes found in audio services such as Auto Shazam [13] – GPUs can certainly accelerate the processing but can we afford the energy cost of continuous sensing?

Programmability. Similarly to the other alternative processors like DSPs, programmability issues are prominent with the GPU. Building mobile apps that exploit this resource entails not only redesigning the algorithms to fit the data parallel programming paradigm, but also using frameworks such as OpenCL or CUDA that require skilled programming.

Compute granularity. To make the most of the parallel computational model, sensor processing algorithms need to be decomposed at the right level of granularity. The GPU

algorithm design should allow for a large number of threads (on the order of thousands or more) to be able to work on different pieces of the data stream with little dependencies among them. This typically involves having an in-depth knowledge about the algorithm semantics to facilitate an efficient program reorganisation.

Memory hierarchy. The GPU features several types of memory that introduce various access latency vs. size trade-offs. Larger but slower memories will need to be used less frequently compared to the smaller but faster ones. For maximum benefits, apps that are deployed on the GPU would need to carefully manage these memories in order to avoid access bottlenecks.

2.3.3 Mobile CPU operating states

The challenges of achieving high performance while reducing power consumption in mobile and embedded devices for general computation tasks are typically tackled by allowing the CPU to switch between different states that offer performance-power trade-offs. Processor performance states (also known as P-states) and idle states (C-states) are the capability of a processor to switch between different supported operating frequencies and voltages to modulate power consumption. The number of states is processor specific. If configured properly according to system workload, this feature provides power savings. Higher P-state numbers represent slower processor speeds, and power consumption is usually lower at higher P-states.

At runtime depending on the workload, the mobile OS will switch between these states to maximise performance, maintain a thermal balance, and save power. High performance modes with higher operating frequencies are typically triggered when the display is turned on, whereas lower frequencies are observed when the display is off with some system processes running in the background. The Android OS defines the so called CPU governors which are strategies that define how the CPU switches between its performance and idle states. The supported set of governors and how they map to the processor states are vendor-specific, but a typical mode enabled by default and supported by most systems is OnDemand which offers balance between performance and energy saving. Tasks that cause the CPU load to spike will trigger high-frequency mode operation, and if the CPU load placed by the user abates, the OnDemand governor will step back down through the kernel's frequency levels until it settles at the lowest possible frequency, or the user executes another task to demand a ramp.

The main focus of this dissertation is studying the ability of alternative heterogeneous processors to deliver performance and energy benefits. We do not explore CPU regulation mechanisms as the CPU is better suited for general computation tasks. With recent hardware advances it will rarely if ever be used for continuous sensing as this prevents the CPU to enter its low-power idle states. In our experiments, our main concern is the total sensing system energy consumption for which we measure how the various built sensing app

Application	CPU Latency	DSP Latency	GPU Latency	CPU Energy	DSP Energy	GPU Energy
Emotion Rec. (14 GMMs) [166]	2.410 s	8.941 s	0.294 s	4073 mJ	340 mJ	1020 mJ
Speaker Id. (22 GMMs) [166]	3.533 s	12.837 s	0.448 s	6041 mJ	488 mJ	1570 mJ
Keyword Spotting [70]	0.720 s	2.249 s	0.055 s	1152 mJ	113 mJ	390 mJ

Table 2.4.1: Application latency and energy compared on the CPU, DSP and GPU (default clock frequency). Results are obtained with a Monsoon Power Monitor [34] attached to a Snapdragon 800 Mobile Development Platform for Smartphones [46]. Measurements are performed in background mode where the display is off. The numbers show the energy needed to do the processing on top of the base energy needed to keep the system running. This is measured as the difference between the average total power when performing computation over 10 runs and average power with only default background system processes running.

components interact with the two major CPU operating modes: interactive with display on, and background with display off.

2.4 Limits of sensor app support

Developers of sensor apps today work with black-box APIs that either return raw sensor data or the results of a limited selection of sensing algorithms [7, 31]. The underlying resources (e.g., emerging low power co-processors or system services that regulate sensor sampling rates) that feed these APIs are generally closed and inaccessible to developers. Critically, because the mobile OS lacks the necessary mechanisms to regulate the energy and responsiveness trade-offs of sensing algorithms, how these apps share resources, remains unoptimised. We now describe the limits of co-processors and cloud offloading support of sensor apps.

Low power co-processor – an underutilised resource. Prior to the advent of co-processors, the CPU was used for both sensor sampling and data computation. This resulted in unacceptable energy trade-offs that made many sensing scenarios impractical.

Potential energy savings. To illustrate potential implications for sensing, we perform an experiment with a special development-open version of the Qualcomm Snapdragon 800 SoC [47], shipping in phones (e.g., Nokia Lumia and Samsung Galaxy [51]). Table 2.4.1 compares the energy and latency of a range of microphone processing algorithms on the DSP, CPU and GPU of the Qualcomm SoC. With the DSP we observe an overall reduction of 8 to 15 times in energy consumption to a level at which it becomes feasible for smartphones to perform various sensing tasks continuously.

However today's smartphone co-processors cannot fully address the needs of sensor app workloads because of two critical limitations: (1) APIs remain narrow and inflexible; and (2) the co-processors are closed.

Limited APIs. Sensor engine APIs similar to the ones provided by Apple [10] and Nokia [31] enable a variety of location and physical activity related sensor apps. Yet, the algorithms needed for many other sensor uses such as custom gesture recognition or fall detection are absent. Unless the developer's use case is already supported by the APIs, CPU-based sampling and processing must be used. While APIs supporting more apps are likely to be offered in future, the closed APIs also prevent stages of sensor processing to be divided between the co-processor and other units like the CPU. Without this ability only algorithms simple enough to be run solely on the co-processor can be supported.

Closed to developers. There are two main reasons behind why co-processors are closed. First, embedded components such as co-processors are easily overwhelmed if not carefully utilised. Opening the co-processor requires complex new OS support providing, for example, a multi-programming environment (i.e., concurrent sensor apps) in addition to isolating apps so that excessive resource consumption by one app would not compromise others. Second, an open co-processor requires developers to engage in embedded programming. This significantly increases development complexity, requiring code for each computational unit (DSPs, CPU) and forcing greater hardware awareness.

Cloud offloading alone is not the solution. Although cloud offloading can enable significant reductions in latency [196, 76, 80, 165], just like existing use of co-processors it is unable to fully meet the needs of sensor app workloads, for two primary reasons – sensitivity to network conditions and CPU-bound offloading.

Network conditions. Under good network conditions (e.g., low RTTs, typical 3G/WiFi speeds) offloading sensor processing, like face recognition, can result in energy and latency improvements of 2.5 times [80]. But such conditions are not always present. For example, a survey of more than $12,000$ devices worldwide [184] finds that a sizable 20% of the devices are not exposed to 3G, LTE or WiFi connectivity at least 45% of the time.

CPU-bound offloading. Conventional offloading applied to mobile sensing (e.g., Nirjon et al.[156]) must rely on the CPU for local operations. CPU-based sensing algorithms are highly energy inefficient. As a result, cloud offloading is severely constrained in the variety of sensor apps to which it can be applied. For example, apps that require continuous sensing cannot be supported with offloading alone. Even the emergence of co-processor support in smartphones has not addressed this problem. Because current co-processors only provide the end result of sensor processing (e.g., an inference), they are unable to act as the front-end to a chain of operations that includes stages executed remotely.

As we have seen, neither co-processors nor cloud offloading fully address the needs of a sensor app workload. What is needed are not additional ad-hoc optimisation approaches, but principled techniques and systems services that support concurrent app operation, that have visibility of the sensor algorithms being executed in each app, along with access to the full range of computational units and other resource types available to the device.

2.5 Design principles

Given the highlighted current limitations of sensor app support on mobile devices, and recent trends of increased demand for concurrent audio sensing services, we define 4 primary design principles we believe are key to the success of any integrated audio sensing solution built to last. In particular, throughout this dissertation the following principles lie at the core of our sensing systems:

- **Leverage off-the-shelf heterogeneous processor infrastructure to reduce energy and minimise expensive CPU computation.** We exclusively focus on hardware that is commercially available as opposed to designing customised hardware components. This ensures our solutions can be deployed for a large user base.
- **Jointly optimise resource use for multiple apps simultaneously.** As the number of concurrently running sensor apps grows, it is important we effectively manage the shared mobile resources among them.
- **Exploit domain-specific knowledge and make sensor processing logic visible to the optimisation algorithms.** This allows optimisation to be highly targeted and be able to address specific performance bottleneck operations.
- **Do not sacrifice app accuracy.** Often when runtime and energy performance are critical, app accuracy may be allowed to degrade by replacing inference components with less precise alternatives that are less computationally demanding. Instead, our goal is to preserve accuracy and maintain user experience expectations high by targeting algorithm optimisations that do not decrease accuracy.

2.6 Present dissertation and future outlook

This chapter has reviewed the typical structure of sensor apps as well as the most common algorithmic primitives used in the important subclass of mobile sensing – audio processing apps. We then overviewed the emerging heterogeneous processors on leading edge mobile hardware platforms and identified key challenges accompanying their role in computational offloading scenarios. Last but not least, we discussed the current limitations of sensor app support, arguing that co-processors have yet to be utilised to their full potential, whereas cloud offloading alone is not a solution to the energy and performance issues continuous sensor processing brings up.

The rate at which new mobile user experiences are created based on inferences acquired from sensor data is higher than the speed with which innovations in hardware come forward to aid computational offloading. Natural user interfaces, activity recognition, context-aware notifications are being widely adopted as utility services by mobile apps and purpose-built digital assistants such as Amazon Echo. An ecosystem of sensing modalities, a large

proportion of which target audio data, is emerging straining battery resources in unprecedented ways. Mobile platforms are increasingly becoming equipped with heterogeneous processors but our understanding of how to efficiently utilise them with multiple concurrent apps is still in its infancy.

This dissertation is a step towards gaining a better understanding of the issues raised above. We design three generalisable approaches to computational offloading for sensor apps primarily relying on microphone data: utilising the mobile processor hierarchy efficiently to support the mixed workload of concurrent sensor apps (Chapter 3); delivering processor optimised audio routines for the DSP and GPU (Chapters 4 and 5); and dynamically scheduling sensor tasks in response to fluctuations in resource availability and sensing workload (Chapter 6).

Chapter 3

Multi-tier processor offloading

3.1 Introduction

The previous chapter argued that we require new mechanisms and techniques to better exploit heterogeneous mobile processors for multi-app audio sensing workloads. Here we begin our exploration of such techniques that leverage low-power co-processors on off-the-shelf smartphones and wearables – our aim is to overcome the battery issues that plague mobile devices when performing continuous background sensing. We strive for energy efficient always-on operation with full accurate coverage of sensing events and avoid duty-cycling schemes which are susceptible to inaccuracies. We investigate the following related issues in the context of smartphones and wearables:

1. Continuous sensing of a diverse range of user behaviours and environments. As illustrated in the operation of example audio apps in Chapter 2, increasingly sophisticated sensing algorithms are being developed that can gather deep user activity insights. An understanding needs to be developed about how such sensing techniques can be integrated together into smartphone and wearable-class hardware and run efficiently and simultaneously on continuous sensor streams.

2. Cloud-free operation by exploiting heterogeneous local computation. An important gap in our understanding is precisely how far we can push emerging mobile hardware (e.g., big.LITTLE technology, availability of DSPs) to execute sensing and related algorithms without the assistance of remote computation. Benefits exist for privacy (as data never leaves the device) and energy with purely device-only solutions.

3. User sensor data analysis with near real-time responsiveness. For many sensed activities, near real-time reporting can become pivotal: examples include the calculation of current levels of stress with respect to other days, or simple queries for memory assistance – when did I last visit this place? Supporting such forms of user experience requires investigation of how inference and analysis can be provided on-demand.

To address these issues we study two integrated sensing systems that serve as proof-of-concept examples of what can be achieved with off-the-shelf co-processor components under common mobile form-factors. The first one is a smartphone audio analysis system that operates within the hardware constraints of low-power co-processors (Qualcomm Hexagon DSP [43]) to continuously monitor the phone's microphone and to infer various contextual cues from the user's environment at a low energy cost. Most existing work (e.g., [140, 160]) on using a low-power processor for sensing has employed custom hardware: we build our system using off-the-shelf phones and propose several novel optimisations to substantially extend battery life. Sensing with commodity co-processor support (e.g., [175, 162]) is still exploratory, with offloaded execution of pipelines typically seen in isolation or in pairs. *We take these efforts further by studying how the co-processor in off-the-shelf smartphones can support multiple computationally intensive classification tasks on the captured audio data, suitable for extracting, for instance, human emotional states [166].* In our design we are able to support the interleaved execution of five existing deep-inference audio pipelines, which is done *primarily on the DSP itself* with only limited assistance from the CPU, maximising the potential for energy efficiency.

To further diversify the audio sensing user insights and explore the capabilities of wearable-scale commodity components, we analyse the sensing workload exerted by a novel wearable sensor, ZOE. It continuously senses a more comprehensive set of user behaviours and contexts that span personal, social and place-oriented sensing. Towards this goal, ZOE incorporates a range of state-of-the-art sensing algorithms designed for devices more powerful than typical wearable hardware. We demonstrate this workload can be serviced with a wearable device – without cloud assistance – and still provide acceptable energy and responsiveness. This is possible through a hardware prototype and a combination of workload optimisations and resource management algorithms. The prototype for ZOE is designed around the Intel Edison SoC [24] that includes a dual-core 500 MHz CPU and 100 MHz Micro Controller Unit (MCU). In our design, the Edison is coupled with another MCU present on a custom designed daughter-board to form a 3-tier computational hierarchy. Each tier in the hierarchy offers a different computation and energy trade-off that, when managed efficiently, enables ZOE to meet user experience needs.

Chapter outline. In Section 3.2 we give an overview of our first sensing system, DSP.Ear, deployed on a smartphone development board. Section 3.3 elaborates on implementation details including a series of generalisable techniques for optimising interleaved sensing pipelines. Section 3.4 provides an extensive evaluation of the system battery lifetime under two modes of operation: standalone and mixed with other smartphone workloads. Section 3.5 moves on to outline the targeted operation of our proof-of-concept wearable, and evaluate its sensing workload given a set of optimisation techniques aimed at extending the battery life. Section 3.6 discusses some of the issues accompanying the two systems. Section 3.7 compares our systems with existing work, and Section 3.8 concludes the chapter with a summary of our contributions.

3.2 Smartphone system overview

In this section, we introduce the main architectural components of our smartphone audio sensing system, DSP.Ear, as well as a high-level workflow of its operation. The system has been designed to perform the continuous sensing of a range of user behaviours and contexts, in near real-time, by leveraging the energy-efficient computation afforded by commodity DSPs found in a number of recent smartphones. To achieve this design goal we have addressed two fundamental challenges:

Supporting multiple complex audio-based inference pipelines. To recognise a broad set of behaviours and contexts we must support a variety of complex inference pipelines. We have implemented a series of concurrently operating representative audio pipelines based on published research: ambient noise classification [142], gender recognition [81], speaker counting [189], speaker identification [140], and emotion recognition [166]. Their detailed operation is presented in Chapter 2.

Operating within commodity co-processor mobile architectural limits. Leveraging low-power co-processor capabilities requires our design to cope with a number of architectural bottlenecks. As discussed in Section 2.3.1 of Chapter 2, the DSP is constrained by memory, programmability and processor speed. It can easily be overwhelmed by the high sampling rates of the microphone and the bursts of computation needed to process audio data deep into pipelines, depending on context – for example, when the user is in conversation requiring a number of inferences (such as emotion, speaker identification).

We overcome these challenges through our system architecture and implementation that interleaves the execution of five inference pipelines principally across a single standard DSP – critically *our design enables each pipeline to be largely executed directly on the DSP and minimises the frequency to offload computation to the primary CPU*. A key design feature is a series of pipeline execution optimisations that reduce computation through cross-pipeline connections in addition to leveraging common behavioural patterns.

Figure 3.2.1 shows the overall system architecture. In DSP.Ear, the phone's microphone is continuously sampled (i.e., without any interruption) on the DSP, which applies a series of admission filters to the sampled audio. Light-weight features are then extracted to determine the presence of acoustic events. If the sampled audio passes an initial admission filter that filters out non-sound samples, then volume-insensitive features are extracted. Further, an additional filter splits the execution into two processing branches – one for speech, and the other for ambient sounds – depending on the output of a decision tree classifier. We now briefly describe each of these execution branches.

Speech processing branch. Under this pipeline branch we perform the following human voice-related inferences.

Gender Estimation. A binary classification is performed to identify the gender of the speaker. The output of this classifier also assists with subsequent pipeline stages that

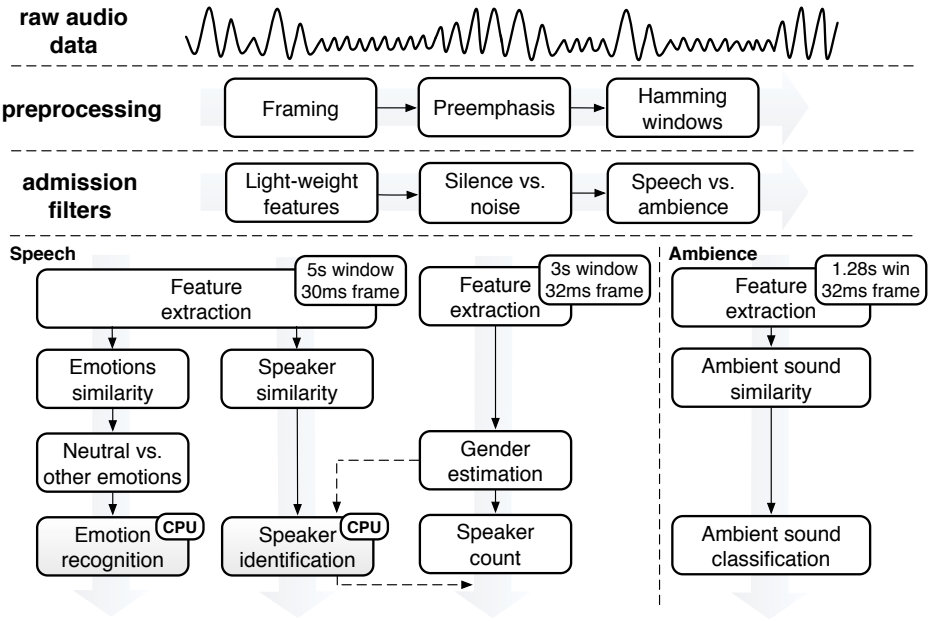


Figure 3.2.1: DSP.Ear Architecture.

estimate the number of nearby people (Speaker Count) and recognising which person is speaking (Speaker Identification).

Speaker Count. To find the number of speakers in a conversation, we implement an adapted version of the Crowd++ unsupervised algorithm [189]. We continuously extract features from 3-second long utterances and perform on-stream merging of subsequent segments into the same cluster depending on whether the feature vectors are close enough. Once we detect that the conversation has stopped after a minute of no talking, we schedule the final stage of the speaker counting algorithm.

Emotion Recognition. We divide the audio stream into 5-second long samples and on the DSP extract a different feature set required by the emotion and speaker identification stages. A light-weight similarity detector computes summary statistics (mean and variance) over the extracted acoustic features and compares the summary vectors of the previous and current audio segment. If the features are similar enough, the emotion labels are propagated from the previous inference.

The emotion classification stage consists of several steps and is mainly performed on the CPU. The first step is applying an admission filter that accounts for whether the emotion is neutral or not. It operates faster than running the full pipeline of all emotions and often saves computation given that neutral emotions dominate in everyday settings [166]. If the emotion is not neutral, the finer category classification continues with determining the narrow non-neutral emotion such as happiness, sadness or fear.

Speaker Identification. The speaker identification algorithm uses the same set of features required for the emotion recognition task and again makes inferences based on audio recordings of the same length (5 seconds). An identical similarity detector calibrated with a different similarity threshold eliminates redundant classifications when the cosine angle between summary feature vectors of subsequent audio recordings is sufficiently low. In the classification stage when the likelihood of offline trained speaker models is derived for the audio recordings, the already estimated genders are cross-correlated with the models to reduce the number of computed likelihoods to the set of gender-speaker matches.

Ambient sound processing branch. This pipeline branch deals with the detection of everyday sounds. We train in an offline manner several classifier models that represent examples of commonly encountered sounds in natural environments (music, water, traffic, other). The classifiers require the computation of an additional set of summary features over a commonly adopted 1.28-second window [141, 189]. At runtime, an ambient sound similarity detector intercepts the classification process and firstly compares the feature vectors of subsequent windows for similarity. If the acoustic fingerprints are sufficiently close to each other, the same type of sound is inferred which bypasses the expensive classification process. In the alternative case, the final stage of the ambient processing pipeline consists of finding the sound model that with the highest probability matches the input sequence of acoustic observations. This classification step becomes more expensive with the increase in the number of sounds against which the audio input is being matched.

3.3 Smartphone system implementation

3.3.1 Concurrent inference pipeline support

In what follows, we describe four general categories of audio inference pipeline optimisations designed to enable each pipeline to operate concurrently within the (largely) hardware limitations of our target prototype platform. These optimisation categories include: *admission filters* enabling the early elimination of unnecessary pipeline stages; *behavioural locality detection* for reducing the frequency of full inference computation; *selective CPU offloading* for delegating tasks the DSP is unable to process; and *cross-pipeline optimisations*.

Admission Filters

We adopt three distinct admission filters based on the combination of implemented audio inference pipelines.

Silence filtering. A large proportion of the time users are situated in silent environments where the ambient noise is low. In such cases performing audio processing is a waste of phone resources. Similarly to Lu et al. [142] we divide the audio stream into a series of

frames and compute the Root Mean Square (RMS) and spectral entropy which we test against experimentally determined thresholds to decide whether the frame is silent or not. We use a shorter frame of 32ms which allows us to increase the precision with which we detect the onset of sounds. In a manner similar to Lu et al. [140] this stage of the processing can be delegated to the low-power co-processor and thus eliminate the need for adopting duty cycling schemes which may miss sound events. Once the frame is determined to contain some acoustic event, all subsequent frames in a time window are admitted for further processing. If a time window occurs such that all frames inside are flagged as silent, frame admission ceases.

Neutral emotion biasing. As reported by Rachuri et al. [166] between 60% and 90% of the time the emotions encountered in human speech are neutral. This finding implies that being able to quickly flag an utterance as being neutral or not could provide significant savings in processing power. In most cases the binary decision would result in the emotion being classified as neutral which can bypass the time-consuming comparisons against all emotion class models. We therefore build 2 GMMs, one representative of neutral emotions and a filler one capturing the rest. When the emotion recognition part ensues we first perform a classification against these two models and finer emotion recognition proceeds only if the detected valence is not neutral.

Speech detection. The majority of the application scenarios we consider require the analysis of human voice which is why ensuring that such processing occurs only when speech is encountered is mandatory. The coarse category classification of the audio samples into speech and ambient noise is a frequently performed step that occurs whenever the environment is not silent. It needs to be fast and efficient while at the same time retaining high accuracy to reduce the false positive rate and avoid unnecessarily triggering the expensive speech processing pipelines. We adopt a strategy that has been established in previous works [142, 140] where we use non-overlapping frames in a window from which we extract features and perform binary classification via a J48 decision tree on the whole audio window. We use 32ms as the frame length and a window of 40 frames which amounts to 1.28 seconds of audio sampling before deciding whether the sound sample contains speech or ambient noise. The low energy frame rate [174] as well as the mean and variance over a window of the features shown in Table 2.2.4 are used.

Leveraging behavioural locality in audio

Many human activities and behaviours, such as taking a shower or being in a specific emotional state, last much longer than a few seconds. However, conventionally our inference pipelines perform the entire inference process each time a new audio data segment arrives from the microphone – even though often the prior user activity or context will be still on-going. The largest computational waste is the incremental Bayesian process of testing each GMM model used to represent possible behaviour or sound categories to find the most likely model given the data segment. Instead, similarly to Lu et al. [143], we adopt a

scheme that computes the similarity between the current and previous feature vectors and triggers the full classification step only if the new acoustic signature differs significantly from the previous one. For Ambient Sound Classification we use summary values (mean and variance) of the features to build vectors that will be compared from subsequent windows. We compute the cosine angle between these vectors and if it remains within a threshold δ we do not run the full classification pipeline and instead propagate the prior category. For Emotion and Speaker Recognition we compute the mean and variance of the PLP coefficients in an audio sample and compare subsequent segments for similarity. If these derived vectors are similar enough we reuse the label of the previous recognised emotion or speaker. Note that this strategy is reasonable since the emotional state is unlikely to oscillate violently between subsequent speech utterances.

Selective CPU offloading

With the deep inference multi-app workload of the audio pipelines, the DSP processing capabilities are challenged in several ways. The limited runtime and shared object file memory, as discussed in Section 2.3 of Chapter 2, mean that the DSP may not be able to load the parameters for all classification models. Therefore, even in the cases when we could afford to opportunistically perform heavy and time-consuming classifications on the DSP, there will be a limit on the type of inferences we could make. The DSP is also unable to indefinitely accumulate useful application data such as features and inferences without approaching the CPU to transfer memory contents. These constraints drive the need for selectively offloading computation on the main CPU where it could perform the tasks the DSP is unable to handle.

To accomplish this offloading the DSP needs to interact with the CPU to transfer feature buffers for further processing. Often the CPU will be in sleep mode at the point of DSP interaction initiation which means that the DSP needs to wake up the CPU. This is currently supported on the Snapdragon 800 platform through a mechanism known as FastRPC that is based on remote procedure calls. The rate at which the CPU is woken up is critical to the energy profile of the full system as waking the CPU is accompanied with a power consumption overhead originating from two sources. First, the wake-up itself consumes power that is an order of magnitude higher than the standby power. Second, once the buffer transferring is over, the CPU remains idle for some time before going back to sleep/standby mode. In our experiments with commercial smartphones such as Samsung Galaxy S2 this time can be up to 15 seconds. Therefore, reducing the wake-up rate proves to be of utmost importance for maintaining a low-energy profile. Note that waking up the CPU happens within a second, so we do not need to wake it up in advance. Power overhead remains despite this.

The time between two subsequent buffer transfers from the DSP to the CPU defines the CPU wake-up rate. This time is dependent on several factors, the most pronounced of which are the runtime memory limitations of the DSP as well as the frequency of encoun-

tered sound events. As the DSP can only accumulate a certain amount of data before waking up the CPU to offload computation, the offloading must work in synergy with the similarity detectors to reduce the number of transferred acoustic features and increase the number of inferences made by the DSP. The speech features computed over 5 seconds of audio recording consume 2 orders of magnitude more memory than the ambient features which means that the rate at which the CPU is woken up is largely determined by the proportion of the time speech is detected. We can formalise the time Δt in seconds between two DSP-CPU interactions by the following equation:

$$\Delta t = \gamma + \frac{(M_L - M_M)}{M_P} \times (1 + \min(S_E, S_S)) \times \tau$$

where,

- γ is the total time in seconds during which silence and ambient sounds interleave the speech;
- M_L is the DSP memory limit;
- M_M is the memory consumed by the audio module parameter initialisation, the pipeline inferences plus the accumulated ambient features;
- M_P is the size of the bottleneck features (speech PLP in our case);
- S_E and S_S are the fractions of similar emotions and speakers identified by the similarity detectors;
- τ is the sampling window in seconds over which speech features are extracted (currently equal to 5 seconds).

In essence, the formula says that the time during which features are accumulated by the space-demanding apps ($\frac{(M_L - M_M)}{M_P} \times \tau$) can be increased thanks to the similarity detectors ($\times(1 + \min(S_E, S_S))$), and that the total time before memory is exhausted depends on it. Note that we need to take the minimum of S_E and S_S as the speech features are shared by the two classification algorithms. To give a perspective on this time Δt, if we assume that γ is 0 (no silence/ambience in between the speech), no classifications are saved and the memory limit is 8MB we easily run out of co-processor space in around 9 minutes. On the other hand, if we encounter only ambience and no speech we can accumulate ambient features for well over 9 hours in the absence of silence. To summarise, the frequency with which a user is involved in conversations is a prime determiner of how often we resort to selective CPU offloading. Maximising the time between subsequent DSP-CPU interactions is crucial for maintaining a low-power system profile.

Cross-pipeline optimisations

Our design leverages the following inter-pipeline links to allow contextual hints extracted by early simpler pipeline components to benefit later more complex inference phases.

Gender filtering for speaker identification. We make use of the gender estimation pipeline to reduce the number of GMMs against which we perform speaker identification. If the gender of the speaker whose GMM is being evaluated does not match the inferred gender from the audio sequence we do not compute the probability for that model.

Speaker count prior from speaker identification. We boost the accuracy of estimating the number of nearby people with a prior based on the number of unique speakers found by Speaker Identification. From Speaker Identification the number of speakers is estimated for those individuals our system possesses a speaker model. This is used to set a prior on the likelihood of each Speaker Count category of nearby crowd size. If Speaker Identification recognises an unknown voice (i.e., a voice too dissimilar from all available speaker models) then category priors are adjusted to reflect the potential for additional nearby people.

Speech detection activated speaker identification and speaker count. Only if speech is detected by an admission filter are later complex inferences of Speaker Count and Speaker Identification made. Otherwise these pipelines of audio analysis are short circuited and never performed.

3.3.2 Prototype hardware and implementation

In order to show how these ideas can improve performance we implemented the framework on a Snapdragon 800 Mobile Development Platform for Smartphones (MDP/S) with an Android Jelly Bean OS [46] (Figure 3.3.1). Versions of this processor platform are present in a large number of commercial devices (including Google Nexus, Samsung Galaxy, and Nokia Lumia series) which is critical for validating the feasibility of DSP.Ear at scale. Access to the low-level co-processor APIs is granted through the C-based Hexagon SDK of which we use version 1.0.0. The audio processing algorithms are implemented in C through the Elite firmware framework which is part of the SDK and is designed for the development of audio modules. We duplicate the C functionality of the audio processing for the Android OS where we utilise the Native Development Kit (NDK) to interface with the Java code. This is needed so that we can compare the system performance and efficiency against a CPU-only implementation. The microphone sampling on the CPU is performed in Java.

The DSP programmability is open to selected development devices such as the MDP/S but not to commodity smartphones featuring the same Snapdragon 800 processor. Currently the version of the C programming language supported on the DSP includes only a subset of the standard libraries commonly found in recent C compiler implementations. This drives the need for porting and modifying audio processing code from other libraries specifically

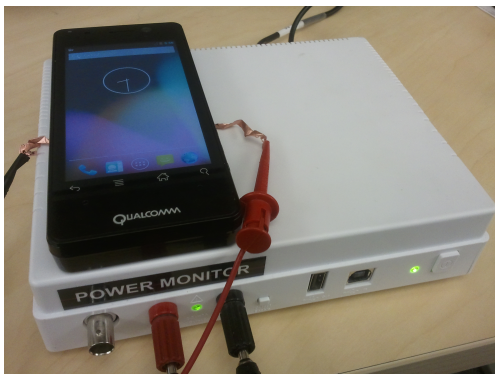


Figure 3.3.1: Snapdragon 800 Mobile Development Platform (MDP) [46] used for the system development.

for the DSP. We adapt common algorithms such as Fast Fourier Transforms, feature implementations and GMM classification from the HTK Speech Recognition Toolkit (HTK) [23]. The training of the GMMs for the emotion and speaker recognition models is performed in an offline manner through the HTK toolkit, while the ambient mixture models are trained through the scikit-learn Python library [49]. The microphone sampling rate used for all applications is 8kHz.

The audio module is deployed via a compiled shared object file with the system code. As already mentioned in Chapter 2 the limit on the maximum allowed size of this file introduces constraints to the number of classification models that could be kept on the DSP at runtime. Since the DSP does not have a file system of its own, the model parameters cannot be loaded from a file but need to be initialised directly through code. The compiled code for one emotion or ambient model occupies approximately 260KB or 87KB of the shared object file respectively. This leads to a maximum of 5 emotion or 16 ambient models that could be used at runtime by the DSP given that the code for the functioning of the integrated system also requires space. Our prototype keeps loaded on the DSP 2 GMMs for the "neutral vs. all" emotion admission filter and 4 ambient mixture models as some examples of commonly found sounds in everyday life (music, traffic, water and a sink model capturing other sounds).

There are three hardware threads on the DSP of which we effectively use two. The hardware threads are architected to look like a multi-core with communication through shared memory. Threading is orchestrated via POSIX-style APIs provided by the DSP RTOS (real-time OS) knows as QuRT [43]. Data is transferred via contiguous ION/RPCMem memory buffers carved out into Android OS memory and mapped to DSP memory. Currently, the Speaker Count algorithm is executed on-the-fly in the main processing thread where the audio buffers become available. The PLP feature extraction required for Emotion and Speaker Identification, as well as the neutral emotions admission filter, are performed together in a separate thread as soon as 5 seconds of audio recording are accumulated. Whereas enabling multi-threading naturally consumes more power in mW than running

each pipeline individually, the latency is reduced so that the overall energy consumption in mJ remains roughly equal to the case of running the pipelines sequentially. This observation confirms a near perfect power scaling for the multi-threaded support of the DSP and it is a crucial feature for energy efficiency advertised by Qualcomm [43].

3.4 Smartphone system evaluation

In this section we provide an extensive evaluation of the proposed system and its various components. The main findings can be summarised to the following:

- The only runtime bottlenecks on the DSP are the classification stages of the emotion and speaker recognition.
- Our design is between 3 and 7 times more power efficient than CPU-only baselines.
- The optimisations are critical for the extended battery lifetime of the system as they allow the DSP+CPU solution to operate 2 to 3 times longer than otherwise possible.
- Under common smartphone workloads the system is able to run together with other apps for a full day without recharging the battery in about 80-90% of the daily usage instances.

In Subsection 3.4.1 the evaluation highlights the accuracy, runtime and power profiles of the pipelines in isolation. Subsection 3.4.2 details a study on the parameters and importance of the introduced optimisations. Last, Subsection 3.4.3 gives an evaluation of the full system energy consumption compared against three baseline models.

3.4.1 Inference pipeline micro-benchmarks

Here we focus on testing the individual audio pipelines with regard to the accuracy, runtime and power consumption characteristics. All measurements performed on the DSP are reported for a default clock frequency and a Release version of the deployed code. We show that among the pipelines the emotion and speaker recognition are a processing bottleneck for the DSP, whereas the feature extraction stages, ambient classification and speaker counting can be run efficiently and in near real time on the DSP. Finally, most app algorithms are an order of magnitude more power efficient when executed on the DSP.

Accuracy. In Table 3.4.1 we show the accuracy of each implemented pipeline is in line with already published results. We report the performance of the algorithms for correctness, with the datasets described in Chapter 2 and recorded in relatively clean environments. Nevertheless, detailed analysis of the algorithmic accuracy under more challenging conditions can be found in the original papers [166, 189, 143].

Application	Dataset	Accuracy
Speech Detection	24 mins of speech and sounds	91.3%
Ambient Sound Classification	40 mins of ambient sounds	92.9%
Emotion Recognition	Emotional Prosody [134]	70.9%
Speaker Identification	22 speakers, 220 mins of speech	95.0%
Gender Estimation	24 mins, 12 male, 12 female speakers	92.8%
Speaker Count	24 mins, 12 male, 12 female speakers	*1.1**

Table 3.4.1: Average accuracy of the pipelines. Results are obtained with 5-fold cross validation. *Average Error Count Distance (AECD) measures the absolute difference between the actual and reported number of speakers. Results are consistent with the Crowd++ [189] reported results for private indoor environments.

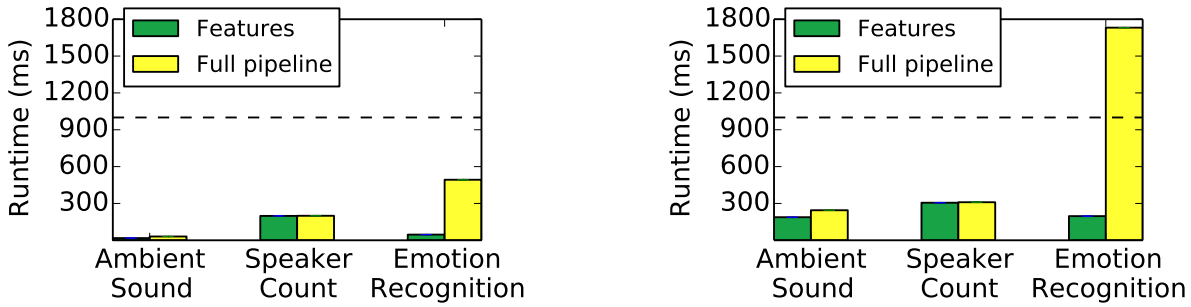
	CPU	DSP
Silence	7.82 ms	45.80 ms
Speech	11.20 ms	66.42 ms

Table 3.4.2: Normalised runtime for processing 1 second of audio data for the silence and speech admission filters.

Latency. The execution of any classification pipeline of the system is preceded by the admission filters ensuring that no unnecessary additional processing is incurred. Since they are always applied to the audio input, being capable of running fast and efficiently is of prime importance. In Table 3.4.2 we demonstrate that although the DSP is between 5 and 7 times slower than the CPU, the admission filters occupy a small 0.045-0.066 fraction of the audio processing per second.

It is worth pointing out that since Figure 3.4.1 portrays normalised runtimes per one second of audio sampling, we can easily distinguish between the application scenarios that can be run in real time on either of the two processing units (CPU and DSP). The quad-core Krait CPU performs all tasks in real time, while for the DSP the emotion recognition (14 GMMs) and speaker identification (22 GMMs) use cases are bottlenecks. The PLP feature extraction stage shared by the two application scenarios, however, consumes less than 12% of the total emotion recognition execution making these features computation real-time on the DSP. The rest of the apps can easily run all of their stages on both the CPU and the DSP without introducing delays.

A noticeable difference between the two processing units is that the DSP operates between 1.5 and 12 times slower than the CPU on the various classification scenarios. The variability in the slow-down factor hints that the DSP treats the computation of the acoustic features differently from the CPU. This is because although the DSP supports floating point operations, some of them such as division and square root are implemented in software. In contrast, modern processors have dedicated hardware instructions for the same set of operations. The pitch computation algorithm, for instance, which is quadratic in the



(a) CPU

(b) DSP

Figure 3.4.1: Normalised runtimes per one second of audio sensing for the various apps when execution happens on (a) the CPU and (b) the co-processor. Note that the Speaker Identification pipeline (not shown) shares the same algorithmic components as Emotion Recognition, the results are qualitatively the same modulo the total number of GMMs used.

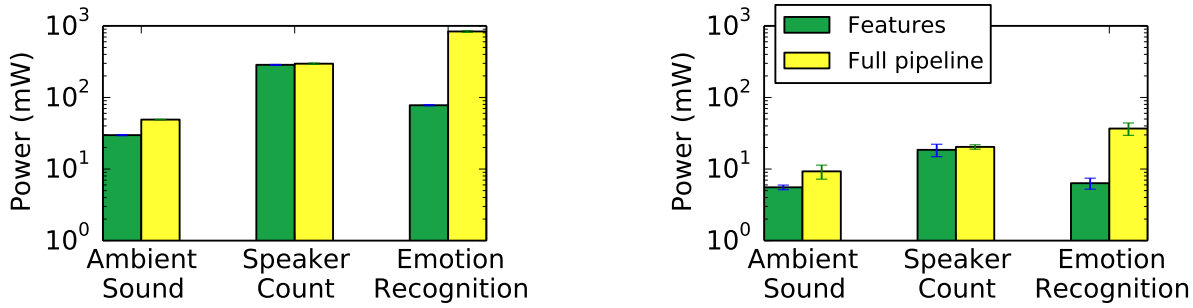
number of the samples in a frame, is composed predominantly of additions and multiplications which favor the co-processor hardware. As we can see from the figure, the DSP is only 1.5 times slower than the CPU on the speaker count feature set where the pitch estimation dominates the MFCC computation. On the other hand, the ambient features computation on the DSP takes slightly over 11 times longer than the CPU. The basic operations for these features include not only floating point division, but also taking the square root and performing discrete cosine transforms, all of which incur a processing overhead.

Power consumption. In this part, we provide critical insights on the relative efficiency of the DSP while performing the various algorithmic tasks compared to the CPU. The measurements have been obtained through a Monsoon Power Monitor [34]. The reported values account only for the processing required by the algorithms without including the cost of maintaining the CPU awake and pulling audio data from the microphone sensor. This is done for this subsection only and so that we can better compare and contrast the energy overhead incurred by the pipeline stages themselves. The average power consumed by maintaining a wake lock on the CPU with a screen off on the MDP device is 295mW, while keeping the microphone on adds around 47mW for a total of 342mW which is consistent with the reported values by Lu et al. [140]. The sampling of one microphone on the DSP with 8kHz maintains a current of $0.6 \sim 0.9$mA ($2 \sim 4$mW) which is comparable to other sensors on low-power chips [160]. Since continuously sampling the microphone on the DSP is not a publicly released functionality yet, we have obtained the exact values for the MDP board through the Qualcomm support team.

As already mentioned in the previous section, the admission filters are performed always as a first step in the processing which is why it is also important for them to be energy efficient. Table 3.4.3 shows this is indeed the case. The DSP is around 7 times more energy

	CPU	DSP
Silence	12.23 mW	1.84 mW
Speech	17.61 mW	2.54 mW

Table 3.4.3: Normalised average power consumption in mW for the silence and speech admission filters.



(a) CPU

(b) DSP

Figure 3.4.2: Average power consumed by the various apps when running on (a) the CPU and (b) the co-processor. Values on the Y axis are in a logarithmic scale.

efficient for both tasks than the CPU. The overhead of continuously checking whether the environment is silent is negligible on the DSP as the power does not exceed 2mW. In the non-silent case, the additional energy cost of performing decision tree classification on the type of sound is very low on the DSP, being merely 0.7mW on top of the first admission filter.

The stages of the app algorithms, on the other hand, are more compute-heavy and consume much more power (Figure 3.4.2). Despite this, *the DSP is an order of magnitude more energy efficient than the CPU.* The emotion recognition for instance consumes 836mW on average on the CPU, while this number drops to merely 37mW for the full pipeline on the co-processor. For the emotion/speaker detection task the DSP is thus more than 22 times more power efficient than the main processor. Similarly, the full speaker count algorithm requires an average of 296mW of power on the CPU, whereas on the co-processor the same execution consumes barely 21mW. To put these numbers into perspective, if we add the required power for maintaining the audio sensing on the CPU, a standard 2300mAh battery with 3.8 V would last less than 14 hours if it performs only speaker counting on the mobile phone and nothing else. If we assume that the CPU drains the battery with 30mW of power in standby state [21], and the DSP microphone sampling consumes 4mW on average, then adding the 21mW of the DSP for the same speaker counting task means that the battery would last for more than 154 hours if it only counts speakers on the co-processor.

	Neutral	Non-Neutral
Neutral	77.26%	22.74%
Non-Neutral	24.20%	75.80%

Table 3.4.4: Confusion matrix for neutral vs. emotional speech. Results are obtained via 10-fold cross validation.

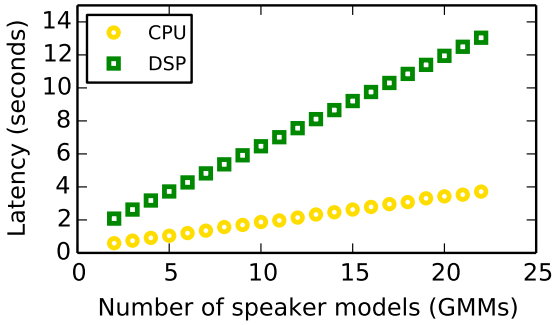


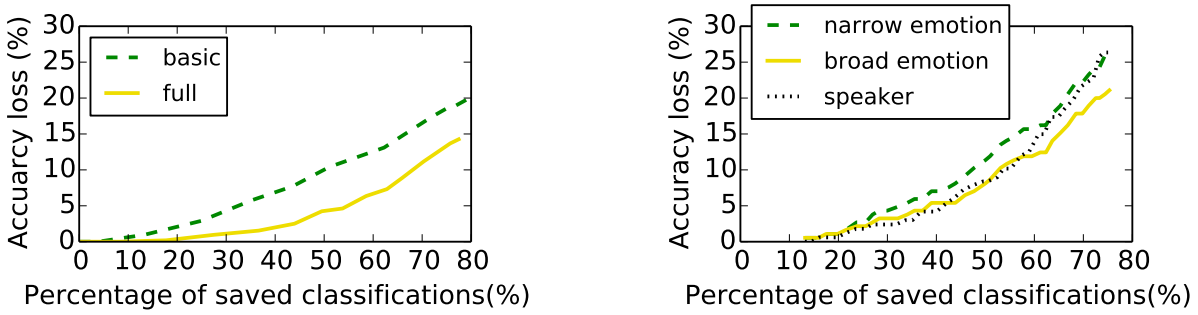
Figure 3.4.3: Runtime of the emotion and speaker recognition use cases as a function of the number of GMMs.

3.4.2 Optimisation benchmarks

In this section we elaborate on several important aspects of the system design related to the introduced optimisation techniques. We comment on the expected trade-offs between the accuracy and the savings in the processing.

Admission filters: neutral emotions. Here we discuss the implications of adding an admission filter that attempts to disambiguate between neutral and other emotions as a step that intercepts the full emotion recognition pipeline. We recall that the filter uses 2 GMMs, one representative of neutral speech, and another one absorbing the rest of the emotions. Table 3.4.4 demonstrates that such a model achieves an overall accuracy of 76.62% and a false negative rate, i.e. neutral emotions predicted as non-neutral, of around 23%. While false negatives unnecessarily trigger the full pipeline of evaluating non-neutral narrow emotion models, in practice, the introduction of such an admission filter early into the pipeline is worthwhile even with this level of inaccuracy because of the following reasons. First, the figures outperform the overall accuracy of 71% demonstrated by EmotionSense [166] on the full set of emotions. Second, as discussed by Rachuri et al. neutral speech occurs between 60% and 90% of the time making the short-circuiting possible for the majority of use cases even when false negative errors occur.

The importance of the neutral emotion biasing becomes more pronounced when we take into account the following fact. Due to the DSP memory constraints which prohibit the



(a) Ambient sounds

(b) Emotions/Speakers

Figure 3.4.4: The percentage of misclassified sounds/ emotions/ speakers as a function of the proportion of saved GMM classifications due to engaging the similarity detectors. The similarity between subsequent sounds is computed based on two cases for the ambient sounds: the basic (without MFCC) and full set of features (with MFCC).

deployment of more than 5 emotion models on the DSP, the full pipeline needs to be executed on the power-hungry CPU. However, if the admission filter, which has 2 GMMs, is deployed and executed fast on the DSP, for more than 60% of the time the emotion will be flagged as neutral and the processing will remain entirely on the low-power unit. In Figure 3.4.3 we demonstrate that this scenario can be achieved. We plot the runtime of the speaker/emotion recognition pipeline as a function of the number of speaker/emotion models (GMMs) involved in the classification step. As can be seen from the figure, a pipeline with 2 GMMs, which corresponds to the neutral emotion biasing, can be executed in less than 5 seconds which is the EmotionSense base audio sampling period. In other words, this step can be executed in real time on the DSP and the emotion processing can indeed remain there for the majority of use cases given the dominance of neutral speech in everyday conversation settings.

Furthermore, when the emotion is flagged as non-neutral the further processing needs to occur only on the set of narrow models comprising the other broad emotions (happy, sad, afraid, angry). Thus, the revised full pipeline leads to the evaluation of the likelihood for a total of 10 GMMs (2 for the admission filter plus 8 for the narrow non-neutral emotions) as opposed to 14 in the original version of EmotionSense.

Locality of sound classification. In this part of the analysis we shed light on the consequences of adding similarity detectors to the processing pipelines. We recall that we exploit behavioural locality so that when acoustic features from neighbouring audio windows are sufficiently similar, classifications are bypassed and the sound category label from the previous inference is propagated to the next window. This optimisation introduces false positive errors when features from subsequent windows are similar but the sound categories are not the same.

In Figure 3.4.4(a) we plot the proportion of similarity false positives as a function of

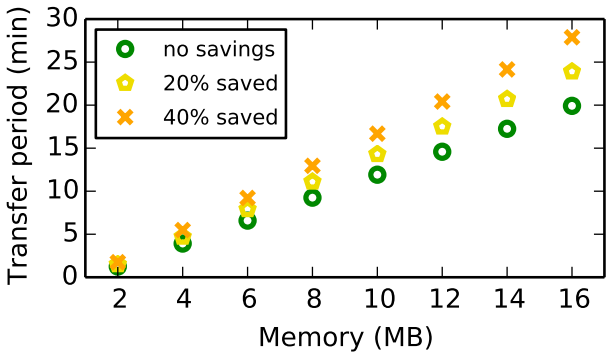


Figure 3.4.5: Time in minutes before the CPU is woken up by the DSP as a function of the runtime memory constraint on the co-processor. The additional gain in time is also given when the similarity detectors eliminate a percentage of the classifications by propagating class labels, discarding features and thus freeing space.

the saved GMM classifications when the similarity detector component is introduced into the ambient processing pipeline. To obtain the figure we vary the similarity distance threshold between the vectors representing subsequent acoustic fingerprints. The threshold in the figure is implicitly represented by the number of saved computations where higher thresholds result in more saved classifications and larger errors. We notice that when we exploit the full feature set with MFCCs (Table 2.2.4 from Chapter 2) to compute the similarity between subsequent audio windows we can save around 50% of the classifications at the expense of a 4.3% penalty in the classification accuracy. Likewise, on the broad emotion recognition task we are able to save 20% of the classifications at the expense of an accuracy loss of only 1%.

Selective CPU offloading. As discussed in Section 3.3.1 the DSP is subject to runtime memory constraints which affects the rate at which the CPU is approached to perform further processing on accumulated acoustic features. The main sources of space concerns are the PLP features occupying nearly 64KB when extracted once every 5 seconds and any ambient features remaining unlabelled because of the unavailability of the full set of sound models on the DSP. As already discussed the intensity of detected speech is the dominating factor in how often the CPU is woken up by the DSP. To provide a lower bound on the amount of time spent before the CPU needs to be interrupted from its sleep mode, we consider the case when human voice is continuously detected.

In Figure 3.4.5 we plot the time the DSP can be actively processing speech audio data before running out of space and waking up the CPU to transfer acoustic feature buffers. In the current mobile development platform the memory limit is 8MB which results in the CPU being approached for processing by the DSP once every 9 minutes assuming no emotion/speaker classification savings are allowed and no accuracy losses are incurred.

Involving the similarity detectors into the pipeline leads to propagating a proportion of the class labels and discarding the corresponding features which frees up space. We can thus extend the wake-up time to 13 minutes at the expense of a modest 5% loss in accuracy when 40% of the features are discarded because of class label propagation. Note that delaying the transfers of the feature buffers to the CPU is desirable energy-wise since the average power consumed to wake it up is generally high. We measure 383mW on average on the MDP during the wake-up process which may last several seconds. In addition, the CPU does not go immediately to low-power standby mode and typically on production phones the CPU may remain idle for 10-20 seconds (such as Samsung Galaxy S, S2, S4) after the processing is over which incurs a power consumption overhead.

3.4.3 Full system evaluation

In this subsection we provide an exhaustive evaluation of the full system given various workload settings and latency requirements.

Assumptions. Based on the analysis performed in the previous sections, we assume default parameters for several of the system components. We activate the similarity detectors so that the percentages of saved classifications are 50%, 40% and 20% for ambient sounds, speakers and emotions respectively. This is done so that we maintain a reasonable accuracy loss of $4\% - 5\%$ for the ambient sounds and speakers and 1% for the emotions. Given the detailed experiments on the distribution of emotions performed by Rachuri et al. [166], we expect neutral emotions to be encountered around 60% of the time which is when the DSP is capable of performing the entire emotion processing. We use the mobile system parameters of a popular smartphone, Google Nexus 5, featuring the Qualcomm Snapdragon 800 platform and having a battery of capacity 2300mAh. The estimated standby time is officially reported to be up to 300 hours which translates to an average standby power of 30mW [21]. Based on measurements we performed on the MDP and other popular smartphones such as Samsung Galaxy S, S2 and S4 we assume the CPU remains idle for 15 seconds after all processing is over and before going to deep sleep (standby) mode. By default, we wake up the CPU once every 11.1 minutes when speech is detected as we run out of space given a runtime memory constraint for the DSP of 8MB. Last but not least, the cross-pipeline optimisation of tagging the speech features with the detected gender allows us to reduce the number of speaker models against which we evaluate the likelihood of the speech features. Our assumption is that the gender detection is able to eliminate roughly half of the speaker models which leaves us with 11 GMMs given our test dataset of 22 speakers.

Unconstrained battery usage. Here we give an overview of how the system drains power given that the full battery capacity is available for use solely by the system. The goal is to contrast how the system fares against baselines, whereas estimates for realistic battery drains under common workloads are given in the next section. We compare the model against three baselines including a CPU-only solution, and two solutions for the CPU and

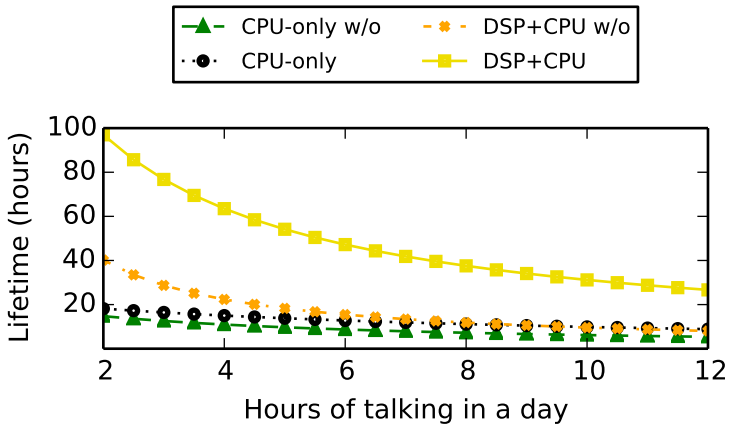


Figure 3.4.6: Lifetime in hours of the system running on the CPU only or the DSP+CPU as a function of the proportion of detected speech during a 24-hour day. Two versions of the system are provided: without (w/o) and with optimisations.

DSP respectively without the introduced improvements of similarity detectors, neutral emotions admission filter and cross-pipeline optimisations. We vary the distribution of sound types a user encounters in everyday settings to demonstrate the level of dependence of the system on the type of processing performed (voice/ambient cases). We fix the proportion of silence in a day to 1/3 of all types of sounds which corresponds to roughly 8 hours of a night's sleep.

In Figure 3.4.6 we vary the amount of detected speech during the day as the voice processing has the most pronounced effect on the battery given that the execution there relies on the heavier pipelines of the system. A first observation is that the DSP solution with optimisations is between 3 and 7 times more power-efficient than the CPU-only solutions. With unconstrained battery usage and 4.5 hours of talking per day the integrated system with optimisations is able to last almost 60 hours exceeding considerably the 14.4 hours reached by a CPU-only solution with the same optimisations. Note that the targeted 4.5 value is an average number of hours spent in conversations in a day as found by Lee et al. [130]. As we increase the amount of detected speech from 4.5 to 12 hours per day, the system longevity significantly decreases where we witness a 58% drop in the total hours of runtime for the DSP case and 41% drop for the CPU-only case.

Another insight is that the optimisations that have been introduced provide a noticeable improvement in the battery lifetime, especially for the DSP case where for 4.5 hours of talking the total hours jump from 20 to 60. The optimisations are so crucial that we observe the following phenomenon: using the DSP without them reaches a point at around 8 hours of speech where the CPU + co-processor design is less efficient than simply having the optimisations on the CPU. This is expected since the major energy burden are the emotion

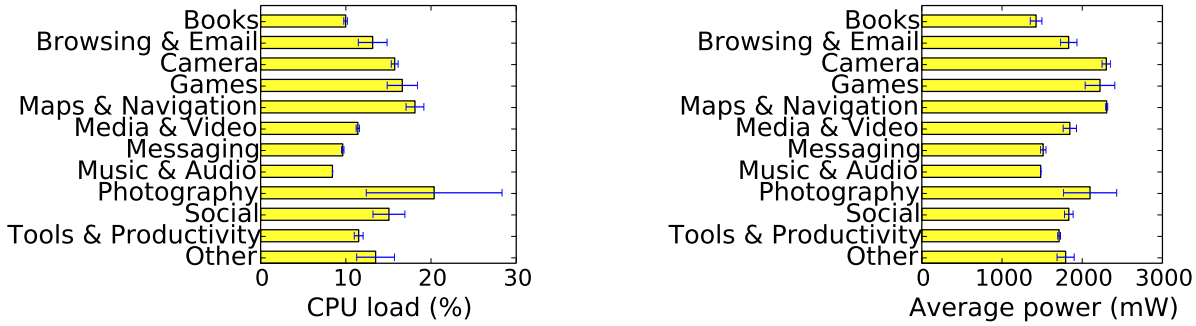
Category	Examples of Profiled Applications
Books	Amazon Kindle, Bible
Browsing & Email	Firefox, Yahoo! Mail
Camera	Camera, Panorama 360 Camera
Games	Angry Birds, Fruit Ninja
Maps & Navigation	Google Maps, Street View
Media & Video	Android Video Player, VPlayer
Messaging	GO SMS Pro, KakaoTalk
Music & Audio	n7player Music Player, Winamp
Photography	Adobe Photoshop Express, Photo Editor
Social	Facebook, Twitter
Tools & Productivity	Advanced Task Killer, Easy Battery Saver
Other	Skype, Super Ruler Free

Table 3.4.5: Categories of smartphone apps used in the CPU workload evaluation.

recognition and speaker identification classifications which always run on the CPU. In this case, running the optimisation procedures on the DSP is critical for enabling the truly continuous sensing of the microphone. The battery is able to last 2 to 3 times more if the mentioned optimisations are added to the DSP+CPU solution.

CPU workload analysis. In this final subsection we study the implications of running DSP.Ear together with other common workloads generated by smartphone users. For this purpose, we use a dataset provided by the authors of AppJoy [192]. It consists of traces of hourly application usage from 1320 Android users, and is collected between February—September 2011 as part of a public release on the Android marketplace. The number of apps found in the user traces exceeds 11K which renders the energy and CPU workload profiling of all apps impractical. Therefore, we group the apps with similar workload characteristics into categories, as shown in Table 3.4.5, and profile several typical examples from each category. To obtain measurements we run 1-2 minute interactive sessions (open-run-close) with each app using the Trepn Profiler [54] for the CPU load and the Power Monitor [34] for the power consumption.

A primary consideration when analysing how the continuous audio sensing workloads incurred by our system interact with the smartphone usage patterns is the energy and CPU overhead of running the CPU speech processing together with other active apps on the phone. A major burden is the interactive usage when the screen is on [67], where the display energy overhead is attributed to the LCD panel, touchscreen, graphics accelerator, and backlight. The Android OS maintains one foreground app activity at a time [192], making the app currently facing the user and the screen the main sources of energy consumption. When running the audio pipelines on the Snapdragon MDP device, we observe that the power consumption is additive as long as the normalised CPU load (across cores) remains below 80%. The apps from the various categories as shown in Figure 3.4.7(a) rarely push the CPU beyond 25% in interactive mode. The additional CPU load of performing



(a) CPU load (%)

(b) Power (mW)

Figure 3.4.7: The average (a) CPU load and (b) power of interactively (screen on) running apps from the given categories.

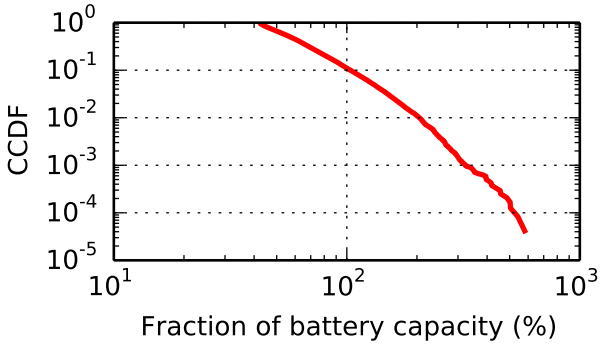
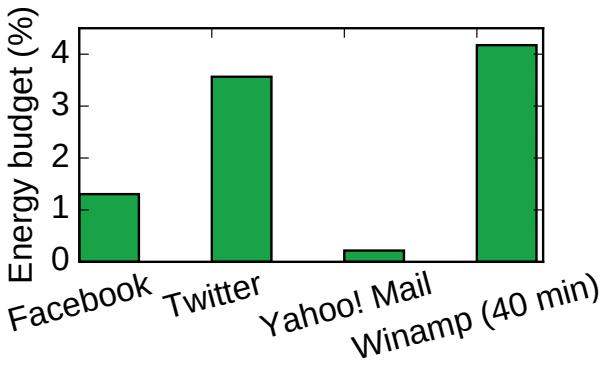


Figure 3.4.8: Complementary cumulative distribution function (CCDF) of the percentage of battery capacity drained in 24 hours of smartphone usage. It shows the proportion of time users will exhaust battery at least the percentage shown on the x axis during the day. Percentages greater than 100% mean that the mobile user needs to recharge the battery at least once during the day.

the speaker/emotion identification is 15% for a total of 40% cumulative utilisation. This remains below the threshold beyond which the energy consumption stops being additive and outgrows the value under normal workloads.

To evaluate the overhead of running our system together with other smartphone usage workloads, we replay the app traces from the AppJoy dataset with measurements performed on the Snapdragon MDP device. In Figure 3.4.8 we plot the percentage of the battery capacity (2300mAh) expended in 24 hours of operation. Since the dataset does not provide explicit information for background app usage when the screen is off, the figure accounts for the interactive usage of the phone, our system running in the background and the CPU standby power draw. This is a best case scenario for our system where third-



Apps	Background Load
Facebook	232KB per 1 hour
Twitter	54.5KB per 15 min
Yahoo! Mail	513 KB, 7 e-mails
Winamp	40 min of playback

Figure 3.4.9: Energy budget as percentage of the battery capacity expended in a day by the background services of 4 popular mobile apps. Measurements are obtained with the Snapdragon MDP where server data from Facebook, Twitter and Yahoo! Mail is downloaded via WiFi.

party background services are disabled, and periodic server data synchronisation and push notifications are off. *In this scenario, in* 90*% of the days the battery is able to sustain at least a full day of smartphone usage without charging.* This is encouraging news, since for the significant proportion of daily usage instances, users need not worry about their everyday charging habits.

The next scenario focuses on adding the overhead of running typical background services such as the ones started from social networking apps, mail clients, news feeds, and music playback. In the AppJoy dataset popular services of this category are Facebook (50% of users), Gmail or Yahoo! Mail (69% of users), Twitter (10% of users), Pandora radio or Winamp (15% of users); most of them sync user timelines periodically in the background, or adopt push notifications. In Figure 3.4.9 we present the energy budget required by 4 representative background services to run over a day with default sync settings. The measurements account for the worst case when the CPU needs to be woken up on every occasion any of the services needs an update. Given this and an extrapolated 40 minutes of background music playback per day, the total energy budget remains below 10%. *If we aggressively add this workload to all daily traces, we find that in more than* 84*% of the instances, the users will not need to charge their phone before the full day expires.* This is again encouraging, since even when users have several popular apps running in the background, they can, in a considerable proportion of the cases, engage in uninterrupted audio life logging.

3.5 Wearable system workload analysis

In the sections so far we have presented the design and comprehensive evaluation of an audio sensing system that exerts a heavy cloud-free sensor processing workload. We have introduced a series of generalisable pipeline optimisation techniques and have demonstrated their importance in enabling the continuous energy efficient operation of multiple apps. In this section we continue our investigation on the benefits of multi-tier processor offloading by addressing the challenges posed by the workload of another proof-of-concept rich-inference sensing system, ZOE. Its key design goals are:

- Unlike DSP.Ear, ZOE is intended to be deployed on a dedicated wearable, but similarly to DSP.Ear, it should primarily exploit off-the-shelf capabilities (such as the resources of an Intel Edison board).
- ZOE supports a richer set of behavioural inferences adding additional types of recognised activities (accelerometer, WiFi) to a set of heavier-to-process audio inferences.

Our most important finding from the analysis reinstates DSP.Ear's insights, namely that *the careful distribution of sensing algorithm stages across heterogeneous processors can drastically reduce the system energy profile compared to naive deployments.* We will now briefly describe the targeted usage scenarios of ZOE, present the intended sensing workload and evaluate the energy efficiency of the system in light of a series of performance control design choices. We rely on variants of the tuning techniques for concurrent pipeline support presented earlier in this chapter: specifically we adopt *admission filters*, *cross-pipeline optimisations*, and *triggered sensing* (an analogue of selective CPU offloading). Thus, ZOE serves as another example of the techniques and design choices that allow us to push the boundaries of modern mobile hardware platforms to provide rich user experiences via sensing.

3.5.1 Targeted user experiences and sensing workload

Targeted operation. With the dimensions defined by an Intel Edison board, ZOE is designed to be worn constantly during everyday activities. ZOE aims to capture a broad set of user actions and ambient conditions that capture a diverse set of inferences from three main areas of life (viz. personal, social, place sensing). This data can be accessed through a dialogue subsystem with supporting functionality built for convenience to facilitate the interaction with ZOE. Users are able to spontaneously ask spoken questions regarding their data (see Table 3.5.1). Significantly, ZOE can provide all of its functionality without cloud support or offloading computation to the user's smartphone. This addresses some aspects of user privacy concerns by allowing all data to remain on the device rather than data leaking to the cloud for analysis. We envision many scenarios of potential use for ZOE including mHealth, life-logging, cognitive assistance, persuasive systems, productivity, and

How much time did I spend driving over the last week?
How many different places did I visit in the last four days?
Compared to the previous time I was here am I more stressed?
What fraction of my time do I spend listening to music?

Table 3.5.1: Representative sentences supported by the dialogue subsystem of ZOE

Category	Subtype	Inferences	Algorithms
Personal	Transportation	{*motorised*, *non-motorised*}	Transportation Mode [104]
	Phy. Activities	{*walk*, *stationary*, *run*, *other*}	Activity Recognition [143]
	Stress Detection	{*stressed*, *neutral*}	StressSense [141]
Social	Social Interaction	{*conversation*, *other*}	Speech Detection [142]
	Conversation Analysis	estimators: *dominance*, *turns*	SpeakerID [166]
Place	Place Recognition	{*home*, *work*, *other*}	WiFi Fingerprinting [120]
	Place Context	occupancy estimation	Speaker Count [189]
		estimators: *music*, *noise*, *chatter*	Ambient Sound [143]

Table 3.5.2: Sensor inference capabilities of ZOE

time accounting in which users want to optimise their activities by maximising what they can accomplish each day.

Sensing algorithms workload. We integrate into ZOE a broad set of microphone-based algorithms together with accelerometer and WiFi example apps that infer user behaviours and ambient contexts. Table 3.5.2 summarises the specific inferences made by ZOE continuously throughout the day. Inferences are selected to provide well-rounded sensing coverage across key areas of the user's life (personal, social and place). The stream of inferences generated by this component are organised into a local User Knowledge Store designed to support dialogue queries. Using a few relatively simple inference processing mechanisms even single inferences generate multiple knowledge base entries. For example, as the user is detected to arrive home by ZOE, many behavioural data points are created, including: the duration of the commute, the distance, and a simple carbon footprint estimate. By storing this information in structured storage, supporting a diverse set of user inquiries is simplified.

Overall, the data-flow of each individual algorithm within our sensing module is, at a high-level, very similar. *All sensors are continuously sampled* with this data feeding into pre-processing stages that perform operations like applying a low-pass filter or estimating the direction of gravity. Domain specific features are then extracted and classification models applied to obtain inferences (e.g., does the collected data correspond to the user driving or not driving).

Accelerometer and gyroscope processing algorithms (transportation mode and activity inferences) share pre-processing techniques and have a number of overlapping common features. Furthermore, both utilise the same classifier design and combine a static classifier with simple temporal smoothing. WiFi-radio processing is very simple in comparison and is not shared with any other algorithm. By far the most complex is audio processing. The

Processor	Operation
Quark MCU	Silence Filtering
Daughter-board MCU	Stationary Filtering, Mizell Calibration Physical Activities Classifier Transportation Classifier Shared Inertial Features Transportation Features
Edison CPU	Shared Audio Features, Stress Detector User Noise Detector, WiFi Similarity Conversation Analysis, SpeakerID Stress Features

Table 3.5.3: Allocation of ZOE workload to hardware components

initial stages of microphone data processing share many similarities with DSP.Ear. Again we encode two pipeline branches, one for human speech and one for background ambient sound. Shared pre-processing includes silence detection and coarse category classification {music, voicing, other}. In contrast, DSP.Ear employed a simpler 2-class speech detection classifier that separates voicing from other ambient sounds. Deep processing occurs along the voicing branch with stages including binary user-oriented Speaker Identification and Stress Detection, while ambient audio is mined for various characteristics of the place (e.g., number of nearby people, noise levels). Sharing features is common in the voicing branch as are models in some cases (such as GMMs). Furthermore, social interaction metrics such as turns and dominance (that are somewhat distinct from the other classifiers included in the module) reuse whole components together – for example, conversation is detected using coarse classification, the presence of silence periods along with Speaker Identification to recognise that a user is actually involved in the conversation and simply not just nearby.

3.5.2 Performance control

In this subsection we detail our main architectural and performance optimisation considerations that aim to enable the continuous and efficient logging of sensed user activities. The heart of our prototype is the Intel Edison [24] SoC that is paired with a custom daughter-board containing a programmable micro-controller, wireless charger, speaker and sensors (viz. gyroscope, accelerometer and microphone). The Edison itself has a further two processing units, the primary one being a dual-core 500 MHz Atom processor which is supported by a 100 MHz Quark processor. WiFi is also built-in directly to the Edison SoC which is required for both sensing and user interaction tasks. Essentially they form a three-tier processing hierarchy. Collectively, different configurations of these tiers offer a variety of energy usage and computational trade-offs that we leverage towards high levels of responsiveness and energy efficiency in ZOE.

MCU utilisation and sensor preprocessing with admission filters. Table 3.5.3

summarises the placement of primary ZOE components between computational tiers. The MCUs excel at continuous sensor sampling at a low power cost and we take advantage of this by allocating to these units key filter-type computations conforming with the memory and runtime constraints. The daughter-board MCU samples the accelerometer without interruption at 50 Hz, whereas the Quark monitors the microphone continuously at 8KHz. The processing units liberate the Atom CPU from performing sensor sampling allowing it to go into a low-power standby state when the MCUs can handle the sensor events from the stream of incoming data.

All microphone and accelerometer algorithms rely on a front-end of shared pre-processing routines residing partly in the MCUs or the Atom CPU. In the case of accelerometer-based ones (activities and transportation mode) these routines are placed on the daughter-board MCU and concern admission control and calibration. Compared to the Qualcomm Hexagon DSP which we exploit in DSP.Ear, the daughter-board MCU is considerably less powerful computationally with an IT 8350 processor (from ITE) running at 48 MHz and only 64KB of RAM. That is why we are able to execute here only accelerometer-based algorithms that process data produced at a much lower frequency compared to the microphone sensor. The admission control pre-processing on this MCU is simple: by applying a low-pass filter along with a threshold it seeks to recognise when the device is completely stationary; perhaps when a user leaves the device on a table. By filtering out these events the remainder of the accelerometer stages are only exercised when there is actually data to be processed. Calibration orients accelerometer values into gravitational units using parameters learned by applying the Mizell technique [149].

The MCU needs to periodically communicate its accumulated inferences with the Atom CPU which is responsible for transferring the labelled data to the knowledge base. There is a communication overhead in the transition of the CPU from sleep to high-power active state. To minimise the interactions between the two processing units we set the synchronisation period to 20 minutes which is when the main CPU needs to be woken up to merge the inferences in the database. This triggered computation is an analogue of selective CPU offloading defined in DSP.Ear, the difference being that here the MCU executes full pipelines and needs to transfer only inference labels. These occupy much less space than features, which is why accumulating labels can be sustained for long periods without exhausting MCU memory.

For the microphone, the front-end of shared stages are only comprised of two processes. The design for these two processes are adopted from Lu et al. [142]. First, a *Silence Filter* deployed on the Quark MCU separates silence from actual sound – a threshold-based decision is made based on the Root Mean Square (RMS) [174]. Second, for those microphone samples that contain non-silence, a coarse-grain category classifier deployed on the Atom CPU is used to recognise frames containing voicing, music and other. We note that the Quark co-processor running at 100MHz is much more limited in its computational capabilities compared to the Qualcomm Hexagon DSP, which is why we were able to deploy only a single filter that is less computationally demanding than the original Silence

Sensing Pipeline	Sensor	Dataset	Accuracy
Physical Activity	Accel	10 mins of activities	82%
Transportation Mode	Accel	150h transportation data [104]	85%
Stress Periods	Mic	Emotional Prosody [134]	71%
Place Recognition	WiFi	LifeMap [73]	90%
Place Context	Mic	local business ambience [186]	74%
Speaker Id (5s)	Mic	22 people, 220 mins of speech	95%
Speaker Id (3s)	Mic	22 people, 220 mins of speech	94%

Table 3.5.4: Accuracy of the pipelines.

Filter used in DSP.Ear. Nevertheless, this co-processor assisted filtering is a powerful tool in reducing system energy consumption when the wearable is in quiet environments – the power-hungry Atom CPU can be put to sleep during the microphone sampling and filtering.

Cross-pipeline optimisations. Similarly to DSP.Ear, we resort to leveraging cross-pipeline connections to further optimise the sensor processing. First, we perform *speech activated* Stress Detection, Speaker Identification and Keyword Spotting to ensure that these apps are rightly triggered only when the acoustic context is human voice. Second, we execute the heavy speech recognition that interprets user commands only after a successful ZOE keyword activation by the owner – Keyword Spotting and Speaker Identification act as triggers for Speech Recognition. This ensures that the system responds to commands only when the user actually intended to issue a query.

3.5.3 System lifetime evaluation

In this part of the analysis we show the total system energy consumption and the overall benefits of the performance control optimisations introduced in the previous subsection. The most notable result is:

- *Our design is practical* as it allows the user to wear the device for well over a day ($\approx 30h$) with a single battery charge. A key enabler for this is the 3-tier design which offers a more than 30% relative improvement in the battery lifetime to alternatives.

We don't elaborate on the sensing algorithm accuracy which has been extensively studied in existing literature. Instead, we find our implementations achieve performance that is in line with already published research (Table 3.5.4). We also skip a detailed analysis on the performance characteristics of the Dialogue Subsystem which has been implemented to offer access to the collected mobile user behavioural data.

System lifetime analysis. We compare our heterogeneous design (Atom CPU + Quark co-processor + MCU) against three baseline models that exclude either the Quark, the MCU or both of the additional processing chips. In this experiment we generate synthetic sensing workloads by varying the amount of encountered speech. This is a prime determiner

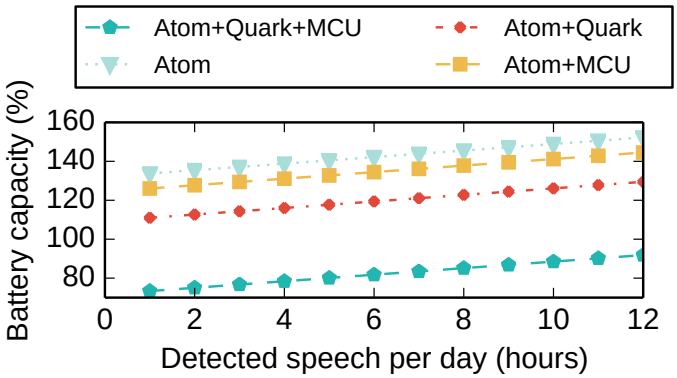


Figure 3.5.1: Percentage of the battery needed to handle the sensor processing of a daily workload (8h of silence, 1h to 12h of speech, 20 questions per day). Values higher than 100% indicate the battery needs to be recharged at least one more time during the day to sustain the workload.

of the sensing workload because the heavier sensor pipelines are triggered by the presence of human voice: dialogue subsystem interactions, stress detection, as well as device owner recognition and conversation analysis. We fix the time users are in a silent environment to 8 hours which roughly amounts to a night's sleep. We also assume 20 questions per day on average submitted to the dialogue system as life logging inquiries. The battery used for the reported measurements is a typical 1500mAh Lithium-ion model with 3.7 volts.

In Figure 3.5.1 we plot the percentage of the battery capacity required by ZOE to process a day's worth of sensing as a function of the number of hours users spend in conversations. The first critical observation is that our system design is the only among the alternatives that is able to make the battery last at least one day with a single charge. With 4.5 hours of speech, which is found to be an average amount of time users spend in conversations throughout the day [130], the system needs 78% of the battery capacity to handle a 24-hour processing load. This leaves us with *a system lifetime of* 30 *hours*. In contrast, the 2-tier solutions as well as the bare bones Atom-only deployment require 109% or more of the battery to handle the same workload which means that the Edison wearable needs to be recharged before the day expires. Overall, *the* 3*-tier design offers more than* 30% *improvement in the system longevity* and this is largely attributed to the fact that part of the sensor processing is offloaded to the dedicated low-power co-processors.

When either of the two additional chips (Quark or MCU) is left out, the sensor sampling that would otherwise be performed there will need to be delegated to the general-purpose Atom CPU. This prevents the CPU from going to low-power standby mode and instead keeps the processor active with an average current draw of 63mA. When the MCU is removed, the accelerometer sampling keeps the CPU busy, whereas when the Quark is

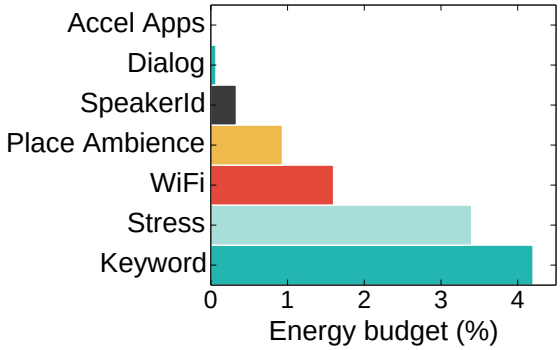


Figure 3.5.2: Breakdown of the energy consumed by the various system component computations for the workload scenario of 4.5h of speech, 11.5h of ambient context, 15min WiFi duty cycle and 20 dialogue commands per day. The reported energy budget reflects the additional energy of the raw processing on top of what the Atom CPU consumes in an active state.

not used, the microphone sampling maintains the CPU awake. Both of the 2-tier designs do not offer significant energy improvement over the Atom-only deployment because 1) their limited memory and performance capabilities allow them to run only simple pipelines such as silence filtering that on the Atom run in less than 2ms, and 2) in both cases the Atom CPU is burdened with sampling one or both of the sensors where the energy hog is maintaining a high-power active state. As shown in Figure 3.5.2, the energy budget for the additional processing of the system components remains cumulatively below 12%, meaning that the majority of the system energy is spent in keeping the CPU active to do the processing in the non-silent periods.

3.6 Discussion and limitations

In what follows, we outline key issues with our implementations.

Programmability. Due to imposed limitations on the supported devices by the publicly released APIs, the development of DSP.Ear cannot be performed directly on a standard commodity phone with the same Snapdragon 800 processor. While this work is a proof-of-concept implementation through which we provide generalisable techniques, questions of programmability are still prominent. Further, due to limitations in early Edison compiler support, we are only able to perform only a few basic ZOE inference stages on the Intel Quark co-processor, a situation that is being currently improved.

Co-processor efficiency. Our current implementations do not fully take advantage of the more advanced co-processor capabilities such as DSP optimised assembly instructions or fixed point arithmetic. The primary advantage of using the C programming language with

floating point operations for the system implementations is that it allows for the prototyping of more advanced algorithms and importing legacy code. However, the challenges in supporting advanced algorithms are not solely based on the need to perform floating point operations or to reduce runtime through assembly optimisations. Limitations still remain — for example, encoding the range of necessary models for each type of supported classification places significant strain on memory reserved for program space on the Qualcomm DSP (e.g., speaker identification requires one separate model for each speaker recognised).

Absence of end-to-end user studies. The focus and contribution of the two works is the systematic study of the hardware capabilities under the workloads DSP.Ear and ZOE present, along with a series of proposed techniques that allow these workloads to achieve acceptable levels of energy efficiency and responsiveness. Of particular interest in future user studies will be user reaction to dialogue as a means of interaction as well as understanding what type of inferences users are most keen on accessing.

Is custom hardware warranted? It is likely that a ZOE-like software system could have been designed for a number of other mobile platforms. However, overall it would have failed to meet the battery lifetime requirements needed to support the heavy sensing workload for a full day with a single battery charge. For example, the energy efficiency on alternative platforms like a smartwatch is expected to be significantly worse than ZOE's due to lack of additional low-power MCUs.

3.7 Related work

DSP.Ear and ZOE touch upon a wide range of topics, here we discuss the most salient prior work in each area.

Smartphone sensing and activity recognition. The design of DSP.Ear and ZOE contribute to the area of human activity inference by investigating how a diverse set of existing techniques can be combined into a single integrated sensing module; in turn this module can run on platforms or form-factors where few have been previously seen individually (and certainly not being all supported simultaneously). Along with the recent explosion of techniques targeting smartphones (e.g., Nirjon et al. [156], Lee at al. [130], and Tan et al. [180]), there is a long history of platform agnostic algorithms from activity recognition (e.g., Bao and Intille [60]). However, even though many share building block components (features, related classes) few studies exist that investigate their usage together (e.g., Ju et al. [116]). Towards mobile devices with a more complete understanding of the life of users, DSP.Ear incorporates 5 and ZOE more than 10 distinct sensor inferences that have previously been often studied in isolation. The few mobile systems that seek a more complete monitoring of user behaviour and environment often focus on health (e.g., Denning et al. [84], and Lane et al. [128]).

Heterogeneity and mobile resource bottlenecks. Energy has been a considerable

focus of a variety of mobile sensing systems [153, 138, 124]. Towards such challenges, low-power processors and multiple tiers of computational units are commonly used approaches [59, 179, 183]. ZOE and DSP.Ear add to this knowledge by exploring the use of co-processors in addition to the CPU to allow a comparatively very heavy sensing and interaction workload to remain feasible, especially in terms of energy. Similar tiered processor architectures include Nirjon et al. [155], Priyantha et al. [160], and Lu et al. [140] that combine high- and low-power processors for energy efficient sensing. Even commercially, many smartphones today use DSPs and other processors to continuously sense and wait for one specific keyword to be spoken before initiating a dialogue. Although the core of the techniques we use have appeared in such systems, it is the first time they have been applied to many of the more complex sensing algorithms (e.g., seen in DSP.Ear or ZOE's social sensing monitoring) and especially when they are integrated into a single system.

Software-based techniques for continuous sensing and cloud offloading. As energy is a major bottleneck for a variety of sensing systems, prominent software-based approaches for managing energy consumption have been devised such as adaptive duty cycling [78, 143, 187], triggered sensing [194, 124, 151], and exploiting relations between various contexts [153]. Adaptive duty cycling techniques enhance the static duty cycling by adjusting the interval according to various metrics such as user's context, energy budget, or mobility. While these techniques enhance duty cycling efficiency, they do not completely fill the gap to perform always-on sensing. In order for these schemes to support continuous sensing, especially when a stream of on-going events need to be captured, they would need to keep the main CPU awake, which would reduce the battery life considerably. Using a low-power sensor to trigger a high-power one [151] and techniques that exploit the relation between context items [153] are applicable to specific scenarios. An example is accelerometer-based triggering of GPS, but it might be difficult to generalise these schemes across different sensors.

CloneCloud [76] achieves energy efficiency and reduces latency by offloading a part of the execution from a local virtual machine on the phone to device clones running in the cloud. Rachuri et al. [165] build a system that distributes computation between the phone and the cloud to balance energy-latency trade-offs. Although there is a similarity between these works and balancing computation between the processors in a hierarchy (CPU and co-processor), the challenges faced by these schemes are very different. While network related issues are an important consideration in the former, memory, computation, and energy limitations are the main focus in the latter.

Wearables and personal sensing hardware. DSP.Ear and ZOE distinguish themselves, particularly from commercial systems, largely because of the depth and range of its user inferences and exploration of features like dialogue systems, heterogeneous computation and constraints like operating without a cloud. Wearable systems that seek the type of breadth as ZOE are almost always based on vision or audio data because of the richness of the two modalities. ZOE relies on other sensors as well but makes strong use of the microphone which sets itself apart in terms of sensing algorithms with wearables like Ha

et al. [98], Mayberry et al. [145], and Hodges et al. [108]. Examples of microphone-based wearables include Rahman et al. [167] but this system has a specific narrow focus on internal body sounds captured with a specially designed microphone. Instead, ZOE has a number of commonalities with the MSP [74] that was also a heavy user of the microphone. However, while ZOE monitors general user activities, as the MSP does, it also provides detailed measurements of user state and social interactions; in addition to providing a dialogue system through which to offer the user this information.

3.8 Conclusions

In this chapter we have studied the trade-offs of using multi-tier processor hierarchies found in state-of-the-art mobile hardware to perform continuous sensing and logging of a variety of complex sound-related behaviours and contexts. First, we have developed DSP.Ear, an integrated audio sensing system with multiple interleaved inference pipelines that are able to run continuously together 3 to 7 times longer than when deployed entirely on the CPU. DSP.Ear is also 2 to 3 times more power efficient than a naive DSP-based design. Second, we have presented a comprehensive workload analysis for a prototype wearable, ZOE, that leverages the type of heterogeneous high-performance computation increasingly available to this class of mobile device. Through a mixture of hardware- and software-based approaches, ZOE offers a first-of-its-kind user experience by enabling the continuous sensing of a range of deep inferences typically seen in isolation or in pairs in more powerful hardware. All of this is provided without cloud assistance, which provides significant privacy assurances to the user.

These substantially extended system lifetimes were largely enabled by the optimisations devised in this chapter. They facilitate the decomposition and distribution of the pipeline stages of multiple concurrent apps across processors. However, we have predominantly relied on example applications the implementations of which were originally designed for CPUs. In the next chapter we demonstrate that higher performance and accuracy gains can be achieved if we also substitute algorithmic elements from the original pipelines with co-processor targeted implementations.

Chapter 4

DSP-optimised deep audio sensing

4.1 Introduction

In the previous chapter we showed how to statically partition the mobile sensing workload from multiple concurrent applications across heterogeneous processors in an attempt to lower the energy barrier for the adoption of richer multi-app sensing services. We have demonstrated the importance of maximising low-power DSP resource utilisation but we have mainly applied pipeline filtering and splitting techniques without modifying the underlying state-of-the-art algorithmic primitives. In this chapter, we investigate whether we can further specialise the algorithm implementations for a low-power co-processor in a way that achieves both improved accuracy and performance. Critically, we seek versatile machine learning models that are robust to a diverse set of inferences and at the same time can be easily deployed on the DSP, i.e. they feature significantly reduced runtime and memory requirements.

A strong candidate for fundamental advances in how mobile sensor data is processed is *deep learning*; an emerging area of machine learning that has recently generated significant attention—enabling, for example, large leaps in the accuracy of mature domains like speech recognition, where previously only incremental improvements had been seen for many years [57]. Promisingly, achieving such levels of robust inference (as seen in speech) often requires overcoming similar data modelling challenges (e.g., noisy data, intra-class diversity) to those found in mobile sensing.

It is somewhat surprising that deep learning techniques are mostly absent from the vast majority of mobile sensing prototypes that are deployed and evaluated. Limited usage exists coming in the form of largely cloud-based models that provide, for example, speech and object recognition within mobile commercial services [57]. Perhaps this is partially due to the computational overhead associated with deep learning: usage of deep neural networks, *even just for inference*, can require amounts of memory, computation and energy

that overwhelm the resources available to this class of hardware. For this reason, more recently, methods are beginning to be explored [102, 71, 70] how the inference-time usage of deep learning models can be optimised to fit within embedded device limits.

What is missing today are systematic studies to understand how advances in deep learning can be applied to inference tasks relevant to mobile sensing. Here, we begin to examine this timely issue with an exploratory study into the potential for deep learning to address a range of core challenges to robust and resource efficient audio sensing on the low-power DSP of embedded devices.

Chapter outline. In the next Section 4.2 we sketch the research directions of our study. In Section 4.3 we describe the prototype of a mobile DNN classification engine capable of a variety of sensor inference tasks. The role of the engine is to classify sensor data on the mobile device, assuming deep model training is performed in an offline manner with conventional tools. The design of the engine is tailored towards the runtime and memory constraints of the low-power DSPs present in many already available smartphones (e.g., Samsung Galaxy S5, Nexus 6). As a result, this engine achieves resource efficiencies not possible if only using a CPU.

Section 4.4 details benefits to inference accuracy and resource efficiency by adopting deep learning techniques. For example, we show our DNN engine can achieve higher accuracy levels for audio sensing using significantly simpler features (a 71 times reduction in features), relative to modelling techniques more typically used. We also discover that we can build DNNs with a resource overhead close to the most simple comparison models, yet simultaneously have accuracy levels equal to or better than any tested alternative. Moreover, our DNN implementation gracefully scales to large numbers of inference categories unlike other models used today.

Section 4.5 focuses on optimising DSP resource utilisation when multiple related audio inference tasks are deployed together on the mobile device. As discussed in Section 1.4 of Chapter 1 this use case is gaining popularity with multi-functional devices such as Amazon Echo. Closely related learning tasks are known to be often suitable for, and benefit from, being modelled under a multi-task learning approach [68, 139, 77, 139]. However, typically these approaches are used towards improving model robustness and accuracy. In this section, we ask if they can also play an important role in reducing the computational resources necessary for deep neural networks.

We extend our inference engine with a deep multi-task framework in which a single deep neural network is used for multiple perception tasks that share hidden layers, and tasks are simultaneously trained. Through comprehensive evaluation we find that for several related audio analysis tasks, commonly used in combination for embedded applications (speaker identification, emotion recognition, stress detection and ambient scene analysis), the reductions in runtime, memory and energy consumption are on average 2.1 times across various multi-task combinations. In addition, we demonstrate that for the related audio analysis tasks, a multi-task approach can replace individual task-specific models with

integrated network(s) of shared hidden layers *with little loss in accuracy.*

4.2 Study design

We now detail our study into the suitability and benefits of deep learning when applied to mobile audio sensing.

Study aims. Four key issues are investigated:

- *Accuracy:* Are there indications that deep learning can improve inference accuracy and robustness in noisy complex environments? Especially when sensor data is limited, either by features or sampling rates. (See Section 4.4.1.)
- *Feasibility:* How practical is it to use deep learning for commonly required sensing tasks on today's mobile devices? Can we push today's hardware to provide acceptable levels of energy efficiency and latency when compared with conventional modelling approaches? (See Section 4.4.3.)
- *Scalability:* What are the implications for common scalability challenges to mobile sensing if deep learning is adopted? For example, how well does it perform as the number of monitored categories of activities expands? (A common bottleneck in forms of mobile sensing such as audio [140]). Moreover, how easily can deep learning inference algorithms be partitioned across computational units (i.e., cloud offloading), a frequently needed technique to manage mobile resources [80]. (See Section 4.4.4 and Section 4.4.5.)
- *Multi-task performance:* What type of deep learning algorithm specialisation can we adopt to optimise resource use when multiple related audio tasks are deployed together? Can we employ multi-task learning techniques such as replacing individual networks with a single shared model to gain memory and computational performance? Can we maintain high both accuracy and performance with this modelling approach? (See Section 4.5.)

By examining these important first-order questions regarding deep learning in the context of mobile sensing our study highlights new directions for the community, as well as providing the foundation for follow-up investigations.

4.2.1 Audio analysis tasks

We focus on a subset of the commonly used audio-related learning tasks presented in Section 2.2 of Chapter 2 and listed in Table 4.2.1: Speaker Identification, Emotion Recognition, Stress Detection and Ambient Scene Analysis (an analogue of Ambient Sound Classification). These tasks are selected because they have a common overall pipeline structure,

Audio Task	Inferences Made
Speaker Identification	106 speakers
Emotion Recognition	afraid, angry, happy, neutral, sad
Stress Detection	stressed, neutral
Ambient Scene Analysis	19 sound categories (street, cafe, etc.)

Table 4.2.1: Audio analysis tasks investigated.

are designed to recognise sound classes, and may be found together in multi-app audio deployments (such as Speaker Identification and Ambient Scene Analysis in embedded devices like Amazon Echo). In this section we introduce the datasets used for analysis, a detailed description of the application semantics can be found in Chapter 2. Specifically, we reuse the default datasets for emotions and stress first outlined in Chapter 2, and we substitute the speaker and ambient scene datasets with much larger ones in order to test the robustness of the deep learning classifiers against a volatile large number of inferred classes.

Speaker Identification. The dataset we use here is utterances from 106 speakers from the Automatic Speaker Verification Spoofing and Countermeasures Challenge [188] with a total of ≈ 61 hours of speech. The dataset bears similarity to the well-known TIMIT one used for Speaker Identification [129]. An audio sample in our case consists of 5 seconds of speech from a speaker, as this duration has been used for mobile sensing tasks in social psychology experiments [166].

Emotion Recognition. We use the Emotional Prosody Speech and Transcripts library [134] where 2.5 hours of emotional speech is delivered by professional actors. For this task, a sample consists of 5 seconds of emotional speech.

Stress Detection. We use a 1-hour dataset of stressed and neutral speech which is a subset of the above mentioned emotions dataset. The length of the inference window for the Stress Detection is 1.28 seconds.

Ambient Scene Analysis. We use the LITIS Rouen Audio Scene dataset [168] with ≈ 24 hours of ambient sounds grouped into 19 sound categories. Each sound sample is 1.28 seconds long, a commonly adopted window in mobile audio sensing [143] when the goal is to capture a variety of sounds that may come and go in the environment.

4.3 Deep engine prototype implementation

4.3.1 Hardware configuration

We implement a DNN feed-forward propagation classifier engine in C for the Hexagon DSP of a Qualcomm Snapdragon 800 MDP [46], which is also used for the development

of DSP.Ear presented in Chapter 3. As already discussed in Section 3.3, this development board allows us to precisely measure DSP performance, and allow measurements to generalise to situations where the DSP is installed on different platforms (e.g., smartwatches, home appliances and phones). We use the same version of the Hexagon SDK to build our DNN engine. Our DSP implementation allows several key parameters to be changed, namely the number of hidden layers and their size, the number of features in the input layer, the number of classes in the output layer, as well as the node activation function.

4.3.2 Model architecture

We explore sizes of the models for audio sensing that are fairly constrained compared to the larger networks used in computer vision and image recognition. Reduced network sizes are preferred to comply with runtime and memory constraints where processing should typically happen in real time on the mobile device to be useful (e.g., speech recognition). We examine model sizes that are comparable to other models applied in embedded settings: 3 hidden layers with 128 nodes each for keyword spotting [70] and 4 hidden layers with 256 nodes each for speaker verification [182]. Our default models have 3 hidden layers with either 128, 256, or 512 nodes each, which are similarly architected to Chen et al. [70], but might have more nodes per layer. With 900 nodes per layer, a model already completely exhausts the runtime memory limit of the DSP. Having the deep network parameters preloaded in DSP memory is essential because the alternative of using the CPU to continuously transfer network parameters on demand is too costly energy-wise. We stress that we aim to provide best results possible within the constraints of embedded hardware, where particularly the memory footprint can severely restrict the overall size and architecture of the models used.

4.3.3 Reducing input feature complexity

The size of the input layer of a DNN equals the number of extracted features in an audio inference window. For typical audio sensing tasks such as the ones outlined in Chapter 2, the features are often accumulated over multiple frames in a window, resulting in hundreds or thousands of input layer nodes. Here, we examine the opportunity to significantly reduce the input layer evaluation overhead by unifying and simplifying both the type and number of features used in audio processing pipelines. We adopt *statistical summaries* of conventional audio filter banks as the input layer to our deep architecture. This is an unusually simple representation, but because audio analysis tasks are by their nature low-complexity (relative to other audio tasks like speech recognition), we observe empirically that it is – *counter to existing practice* – sufficient (reported later).

Until now usual practice has been to define model specific features or use variants of PLP coefficients [105] or MFCC [88]. A good candidate for common features across the various

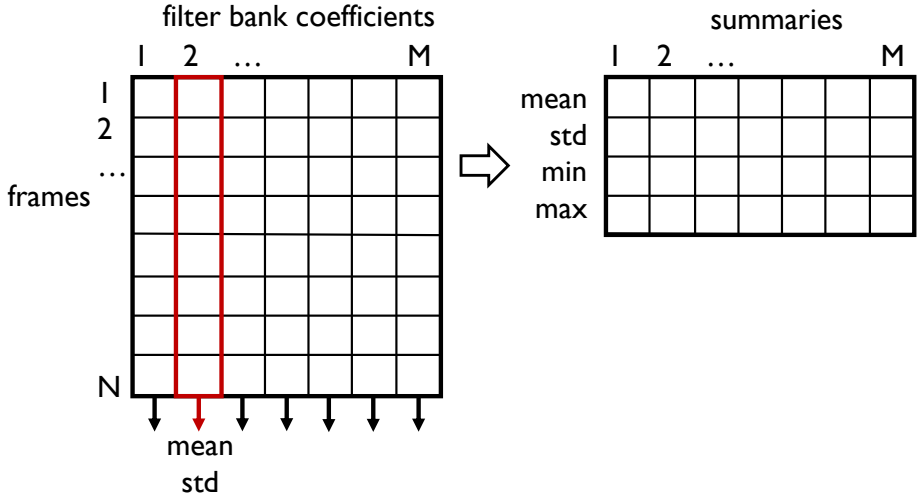


Figure 4.3.1: Extracting summary features from the original set.

audio processing tasks are the log filter banks [178] which are an early step in the PLPs and MFCCs computational pipelines. On speech recognition tasks [82] it has been shown that using filter banks does not compromise the accuracy of deep network models, and can even boost it.

In our design, we take the use of filter banks one step further by using summaries of filter bank coefficients in the following manner, that have the benefit of requiring significantly fewer processor resources. Figure 4.3.1 illustrates the summarisation process. We extract filter banks from each frame over a time window, and summarise the distribution of the values for each coefficient across the frames in a window with statistical transformations (min, max, std, mean, median, 25-percentile and 75-percentile). This allows us to significantly reduce the number of features used in the classification process. For instance, a Speaker Identification pipeline [166] that extracts features every 10ms over a 5-second window would result in $500 \times M$ filter bank coefficients, whereas resorting to summary ones we would need only a fraction ($7 \times M$, i.e. ≈ 71 times fewer). Further, using summary features allows us to have an input layer with the same size across audio tasks regardless of the length of the inference window. A 5-second audio window used in Speaker Identification would have the same input layer as a 1.28-second window used in Ambient Scene Analysis. This is useful when building a shared DNN model across multiple related audio analysis tasks.

4.4 Inference accuracy and resource efficiency

In our next set of results we examine accuracy, energy and latency properties of DNNs applied to common behavioural inference tasks. The key results from our experiments are:

- a simple DNN model with a 71 times reduction in the number of input features provides comparable or superior accuracy against learning techniques in common usage;
- DNN use is feasible on the DSP and has a low energy and runtime overhead allowing complex tasks such as emotion detection or speaker identification to be performed in real time while preserving or improving the accuracy;
- DNN solutions are significantly more scalable as the number of recognised classes increase;
- Splitting models between computational units (e.g., a local device and cloud) is more flexible with a DNN that offers energy/latency trade-offs at a fine granularity.

Our early results point to the ability of DNNs to provide energy and latency trade-offs that will be suitable for a wide range of mobile sensing scenarios; while also having beneficial resource characteristics not found in any other commonly used model.

Experiment setup. We use the model architecture detailed in Section 4.3 to evaluate the energy and latency characteristics of the four inference domains detailed there. We train Deep Belief Networks with Gaussian RBMs in Python via Theano [181]. We fix the learning rate to 0.05 and use a rectified linear unit (ReLU) [193] activation function. As is common for multi-class audio inference tasks [129], we report classification accuracy averaged over 10 trials as a model performance metric. The datasets are divided into training, development and test sets with an 80%-10%-10% split. We limit the total training time to 200 epochs across experiments, but generally observe that for the larger datasets the accuracy improves further if we allow a slower training time with an increased number of epochs. The input layers of the DNNs use the 7 summaries of the 24 filter bank coefficients over the frames. The DNN model is further used to implement a keyword spotting example [70] which brings to light cloud offloading benefits studied in our cloud partitioning experiment. The GMMs are set up with 128 mixture components [166].

4.4.1 Inference accuracy

Baselines. As discussed in Chapter 2, by far the most commonly used classification model in audio sensing are GMMs [143, 166]. Variants of these models, before the advent of deep learning, provided state-of-the-art performance in a wide range of audio processing scenarios including speech recognition and ambient sound classification. That is why we adopt the GMM-based apps built for our integrated sensing systems DSP.Ear and ZOE from Chapter 3 as baselines in our analysis here.

Results. In Table 4.4.1 we compare the accuracy of the memory-compliant deep learning alternatives against the GMM versions. A first observation is that the deep networks outperform the original pipeline setup across most evaluated tasks (except Ambient Scene

	MFCC/PLP GMM	Filter banks GMM	MFCC/PLP DNN	Filter banks DNN
Speaker Identification	82.5%	80.1%	84.7%	85.8%
Emotion Recognition	70.9%	66.5%	81.0%	81.5%
Stress Detection	71.0%	67.2%	80.1%	80.7%
Ambient Scene Analysis	74.1%	63.3%	66.4%	85.2%

Table 4.4.1: Accuracy of the DNN models compared against GMMs as a function of the features used in the input layer – handcrafted MFCC/PLP or summary filter banks. The DNN structure is 3 hidden layers with 512 nodes each. The GMMs have 128 mixture components.

Analysis) even when using the handcrafted MFCC/PLP features as the source for the input layer. The Emotion Recognition and Stress Detection apps, for instance, enjoy a large 8-10% absolute boost. Further, even with the significant loss of feature complexity, *the filter banks DNN provides superior accuracy results for* ***all*** *audio sensing tasks*. This alternative outperforms all others, suggesting that we can safely resort to the computationally much simpler features with no accuracy loss. The results also imply that the accuracy improvement is not solely attributed to the use of a different set of features: the GMMs with filter banks fail to reach the performance levels of the deep networks.

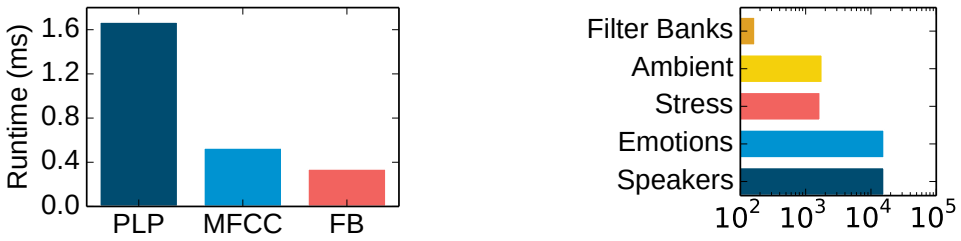
Note that for Speaker Identification and Ambient Scene Analysis we use significantly larger datasets compared to the ones used in Chapter 3 with many more inference classes per task, which is the reason for the different baseline accuracy achieved here. By resorting to these larger datasets we are able to assess modelling techniques with respect to their level of robustness to the complexity of the audio analysis tasks.

To sum up, the incorporation of deep learning for the range of audio inference tasks presented here results in higher accuracy levels despite the memory-limited smaller-sized DNNs, significantly reduced complexity of the input features, and the volatile large number of classes for some of the inference tasks.

4.4.2 Filter bank summaries efficiency

We now investigate the runtime trade-offs of adopting the unusually low-complexity filter bank summary features instead of the handcrafted MFCC and PLP coefficients that dominate the audio processing landscape for shallow models.

Improved runtime. In Figure 4.4.1a we plot the time needed to extract features from a single 30ms frame as a function of the feature type. The filter banks computation occupies only a fraction of the total time needed to extract the PLPs used in Emotion Recognition and Speaker Identification or MFCCs used in Ambient Scene Analysis and Stress Detection. The runtime reduction is 4.9 times for the former and 1.6 times for the latter. Further, as already shown in Table 4.4.1 and the previous section replacing handcrafted features with filter bank summaries does not compromise the accuracy of the DNN models. Adopting the



(a) Feature extraction runtime (b) DNN input layer size (feature count)

Figure 4.4.1: (a) Runtime needed to compute features from one 30ms frame. (b) Size of the DNN input layer (number of features per audio processing window) across the various audio sensing tasks compared against the use of window-agnostic summary filter banks (FB).

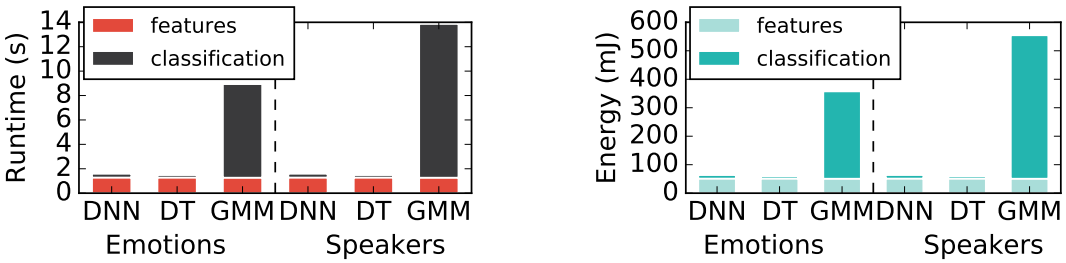
summaries results in accuracy levels that match or surpass the originally provided features across all audio sensing tasks.

These reductions in the compute time of the features are critical for low-power processors that operate with a reduced clock frequency. As discussed in Chapter 2 and confirmed with evaluation in Chapter 3, the DSP achieves its ultra low power profile at a price. Some of the floating point operations such as square root are implemented in software, and the handcrafted PLP and MFCC features make use of such more advanced operations, which taxes runtime performance. Instead, reducing feature complexity by eliminating such operations as in the case of the simpler filter banks enables computation to be much faster – with benefits to reduced energy consumption and arriving faster at an inference.

Input layer efficiency. The extracted filter bank summaries serve as the input layer to the deep networks, and as discussed in Section 4.3 they succinctly describe the distribution of the filter bank values across the frames in a window. The advantages of this representation are that i) the size of the DNN input layer is significantly reduced as shown in Figure 4.4.1b; and ii) we can have a shared input layer for all audio sensing tasks regardless of the size of the inference window and number of frames inside it. Compared to an input layer that contains all PLP coefficients for Emotion Recognition or Speaker Identification, the size of the filter banks variant is 2 orders of magnitude less, resulting in an input layer propagation that is about 95 times faster.

4.4.3 Feasibility results

In this experiment we provide insights with respect to the DSP runtime and energy footprint of DNNs compared against other techniques (DT, GMM) widely used in the mobile sensing literature. In Figure 4.4.2 we plot the latency and energy profiles of two of the sound-related apps detailed in Section 4.2: all apps described in that section use an iden-



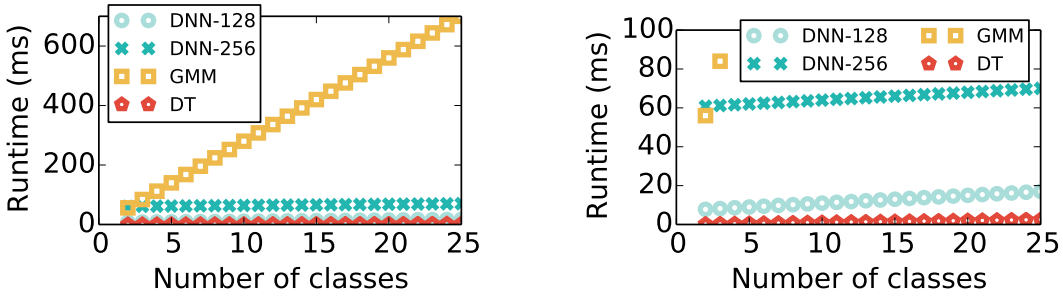
(a) Latency (b) Energy

Figure 4.4.2: Latency and energy of the emotion recognition and speaker identification when deployed on the DSP. The example DNNs have 3 hidden layers with 512 nodes each. The emotions app uses 14 GMMs, the speaker identification uses only a subset (22) of the models but still incurs a prohibitively high runtime and energy overhead. In contrast, the DNNs feature a low latency/energy overhead similar to a Decision Tree (DT). Note that because of the GMM high runtime, the DNN and DT appear to have the same runtime, but in fact the DT is several times cheaper than the DNN, the similarity is relative to the expensive GMM computation.

tically structured GMM-driven pipeline, with the difference being the number of class models in the classification stage. As a demonstration, we pick two of these pipelines with a different number of inference classes. The GMM version of the Emotion Recognition task, for example, runs for approximately 9 seconds and requires 350mJ on the DSP to process 5 seconds of audio data. A most notable observation is that *the DNN classification overhead of the relatively small deep network is extremely low compared to a GMM-based inference and is closer to the overhead of a simple Decision Tree*. We recall that both the Emotion Recognition and Speaker Identification operate on acoustic features extracted from 5 seconds of audio samples which means that *the DNN versions of the apps, unlike the GMM-based implementations, can perform complex sound-related inferences in real time with comparable or superior accuracy*. The prohibitively high GMM overhead stems from both the large amounts of features (500×32) serving as acoustic observations and the additive nature of the classification where one full GMM is required per class.

4.4.4 Scalability results

In this part of the analysis we shed light on how the DNN scales with the increase in the number of inferred classes. Mobile context inference tasks often require a larger number of behaviours or activities being recognised such as multiple activity categories [143] (e.g. still, running, walking with phone in pocket, backpack, or belt etc.), multiple words, emotional states or speakers [166]. In Figure 4.4.3 we plot the runtime of the classification stage of the three models (DT, GMM, DNN) as a function of the number of recognised contextual categories. Again, the DNN behaves in manner similar to a simple Decision Tree where



(a) Full scale (0 to 600 ms)

(b) Zoom in (0 to 100 ms)

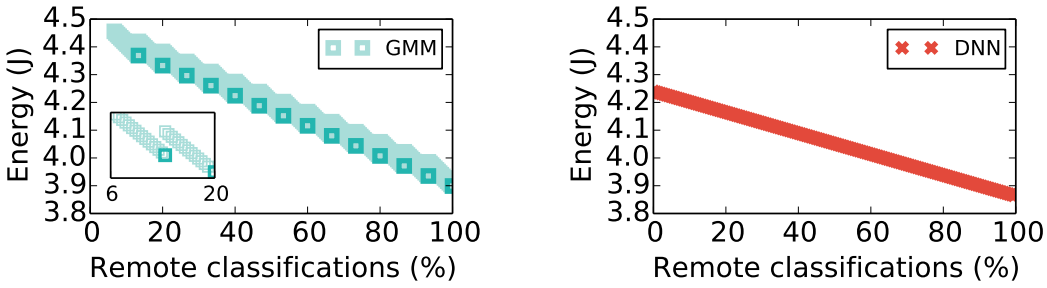
Figure 4.4.3: DSP runtime of the inference stage of the various classifiers as a function of the number of classes. The results suggest that DNNs scale extremely well with the increase in the number of classes, in a manner similar to a DT, while often providing superior accuracy.

the larger number of supported classes does not significantly affect the overall inference performance. The runtime of the feed-forward stage of a deep neural network is dominated by the propagation from the input and multiple hidden layers which are roughly invariant to the number of classes in the output layer. The GMM-based classification computes probability scores for each class represented by an entire GMM so that an inference with 25 added categories/classes is 25 times more expensive than one with a single class. This justifies the more than 11 times slower inference compared to a 256-node DNN [100] for 25 recognised categories and an identical number ($750 = 25 \times 30$) of input features for all models.

We note that using accurate GMMs with many parameters for classification is taxing not only on the runtime but also the memory footprint of the DSP. As exemplified in the DSP.Ear system from Chapter 3, we were forced to reduce the number of models deployed there to comply with the memory constraints. Using a scalable classifier such as the DNN eliminates the need for selective CPU offloading, it enables the DSP to operate independently from the CPU and execute a full pipeline of high complexity in real time.

4.4.5 Cloud partitioning results

In this experiment we investigate the benefits of DNN-based inference usage with respect to cloud offloading. To set up the experiment we consider a speech recognition scenario where a set of keywords need to be detected from voice on the mobile device. A common DNN approach adopted in speech processing [70, 100] is repeatedly invoking the DNN feed-forward stage on short segments, such as once every 10ms in a keyword spotting app [70], and then performing post-processing on the sequence of extracted DNN scores for obtaining the final inference, such as the probability of encountering a keyword. In Figure 4.4.4b



(a) GMM cloud offloading

(b) DNN cloud offloading

Figure 4.4.4: Energy footprint of a speech recognition inference model based on GMMs or DNNs when a proportion of the classifications are performed in the cloud. For the GMM case a zoom-in for the 6% to 20% partition range is also provided. A DNN with 3 hidden layers and 128 nodes per layer is invoked every 10ms similarly to Chen et al. [70], whereas 15 GMMs with 128 components are used once every second. Experiment duration is 15 seconds with a WiFi connection assumed (5Mbps uplink). DNN usage allows for a graceful reduction in the energy consumption unlike the choppy GMM offloading.

we demonstrate that the high frequency of DNN propagations facilitates *cloud offloading decisions to be performed at a fine level of granularity with a graceful reduction in the total energy consumption when a larger proportion of the DNN inferences are performed in the cloud.*

In contrast, a GMM-based approach would usually increase the total amount of time acoustic observations (features) are accumulated before resorting to an inference. This together with the overhead of evaluating the probability of multiple GMMs (e.g. one per keyword) for a single inference, lead to the much choppier falls in the energy consumption for this model when a percentage of the GMM computations are offloaded to the cloud, as illustrated in Figure 4.4.4a. This phenomenon is portrayed in Figure 4.4.4a with the saw-like shape of the energy curve. We highlight that *such a curve is harder to control to a specific energy budget.* Situations where a certain number of the per-class GMM inferences need to be performed remotely may often be encountered because of latency/resource constraints, for instance, which introduces the above mentioned local-remote split inefficiencies. The DNN energy curve with a smoother gradient is therefore largely preferable.

4.5 Multi-task audio DNNs with shared hidden layers

The previous sections demonstrated the accuracy and resource efficiency benefits of applying small-footprint deep learning models to audio sensing tasks in common usage. We have studied techniques that enable each individual task to be optimised separately, but multiple continuous audio applications are routinely required to be deployed together in

embedded systems. Examples of such related tasks were given in Table 1.4.1 of Chapter 1. In this section we investigate ways in which deep learning can be exploited for the benefit of multiple concurrently running audio analysis tasks.

We propose a novel approach to modelling the multiple audio analysis tasks that is based on multi-task deep learning. It results in a degree of shared hidden layers between each task, depending on the cross-task impact on accuracy, and memory/computational constraints of the embedded target platform. This has multiple benefits:

- the potential for supporting a larger number of inference tasks through a shared representation that complies with embedded memory constraints;
- the advantage of fitting bigger and potentially more accurate networks with more parameters instead of compromising model size to make room for multiple models;
- a substantially reduced runtime and energy facilitated by the evaluation of a single shared model as opposed to performing multiple classifications separately.

We find that multi-task learning successfully integrates tasks while preserving accuracy due to their similar structure and lower complexity (relative to other audio tasks like speech recognition).

Building shared representations and deploying them on the embedded device is preceded by an offline training optimisation step the purpose of which is to determine a *deployment configuration*, or what combinations of tasks should be integrated to have a shared representation and what would the DNN model sizes be given the memory constraints. A configuration results in some subsets of tasks having a shared representation (possibly all), while others might end up with individual models. The final chosen configuration is subject to change depending on the goal of the deployment; typically we strive for the most accurate, energy efficient and fastest configuration.

The rest of this section describes the most salient points of the shared-layer training approach, and then outlines the offline optimisation process that prepares a multi-task configuration for constrained use on embedded hardware. The section concludes with a comprehensive evaluation of the resource efficiency advantages of the multi-task framework.

4.5.1 Simplified architecture by sharing layers

Figure 4.5.1 portrays an example architecture of the proposed multi-task audio Deep Neural Network (DNN). In this figure, all tasks are shared; although the decision to integrate all tasks into a single network is left as a hyper-parameter decision based on how accuracy of each tasks varies when tasks are combined.

In our architecture, the input and hidden layers are shared across potential combinations of audio analysis tasks. These layers can be considered as a universal feature transformation front-end that captures acoustic observations. The softmax output layers of tasks that are

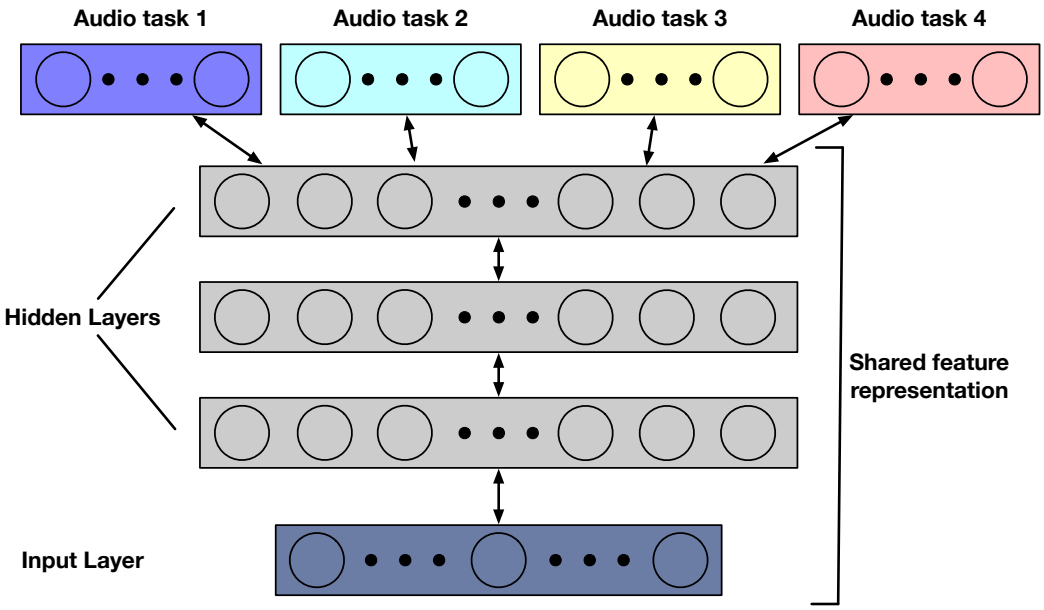


Figure 4.5.1: Architecture of the multi-task DNN.

combined are not shared, but each audio sensing task has its own softmax layer that acts as a classifier that estimates the posterior probabilities of the sound categories specific to each audio analysis task. Any task that is not determined to be joined within shared layers remains as a separate network within the collection.

The key to the successful learning of the multi-task DNN is to train the model for all the audio tasks simultaneously. We adopt minibatch Stochastic Gradient Descent (SGD) where each batch contains samples from all training data available. To accomplish this we randomise the samples across audio tasks before feeding them into the DNN training procedure. Each minibatch contains stratified samples, i.e. the larger datasets cast proportionally more samples.

The fine-tuning of the multi-task DNN can be carried out using a conventional backpropagation algorithm. However, since a different softmax layer is used for each separate audio sensing task, the algorithm is adjusted slightly. When a training sample is presented to the multi-task DNN trainer, only the shared hidden layers and the task-specific softmax layer are updated. Other softmax layers are kept intact. The entire multi-task DNN and its training procedure can be considered as an example of multi-task learning. After being trained, the multi-task DNN can be used to recognise sounds from any task used in the training process.

Since the shared layers represent a form of feature transformation that is trained end to end with multiple datasets, eventually all the network parameters contribute to the learnt hidden structure. As such, all nodes and connections are required in the classification process.

4.5.2 Optimising the multi-task configuration

Prior to placing a multi-task configuration on the embedded platform, we perform an offline optimisation step that aims to decide the level of integration of the various tasks – a shared representation may be built for only a subset of the tasks if the accuracy is critically impacted otherwise. A deployment configuration consists of the set of DNN models (shared and/or individual) that cover the range of inferences supported by the audio sensing tasks. The optimisation process is guided by a set of hyper-parameters that control the search space. Depending on the end goal or required level of accuracy for each task, one deployment configuration may be preferred over another at different times. The hyper-parameters are:

- *accuracy criterion* – it compares a list of accuracies against another one. Each candidate deployment configuration has an associated list of accuracies as observed on the validation sets for each audio task. At the end of the training optimisation process, the configuration that is best according to the criterion is chosen. Example criteria are picking the candidate configuration that gives the highest average accuracy, or selecting the configuration that minimises the mean accuracy loss across tasks.

- *network topology* – it specifies the overall deep neural network layout but not the exact size of the model. By default, we explore DNNs with hidden layers each of which has an identical number of nodes. This topology has proven effective in a variety of audio analysis tasks such as keyword spotting [70], and text-dependent speaker verification [182]. We vary the number of nodes per layer, but to limit the search space we constrain the numbers to being powers of two. Typical small-footprint networks trained for embedded devices feature 128 [70], or 256 [182] nodes in one hidden layer. Models in a single configuration generally can have different network sizes unless they share the layers. In our experiments to reduce the search space we exploit same-size topologies for both the shared and individual models.

- *embedded memory limit* – the total size of the DNNs in a configuration is largely constrained by the maximum amount of memory that can be used at runtime. Audio tasks that perform continuous sensing are typically deployed on a low-power co-processor (e.g., the Qualcomm Hexagon DSP) that runs its own real-time OS and operates independently from the power-hungry general-purpose CPU. A prominent limitation is the amount of runtime memory available to such low-power units. To maximise energy efficiency we would want to maintain the co-processor operation largely independent of the CPU, and use its own memory without resorting to interactions with the CPU for processing or parameter transfers. We use this hyper-parameter as a leading constraint, any configuration for which the DNN models exceed the memory limit is discarded by the multi-task optimisation process.

- *combination search policy* – this parameter restricts the search space to the exploration of certain types of configurations in an attempt to speed the optimisation

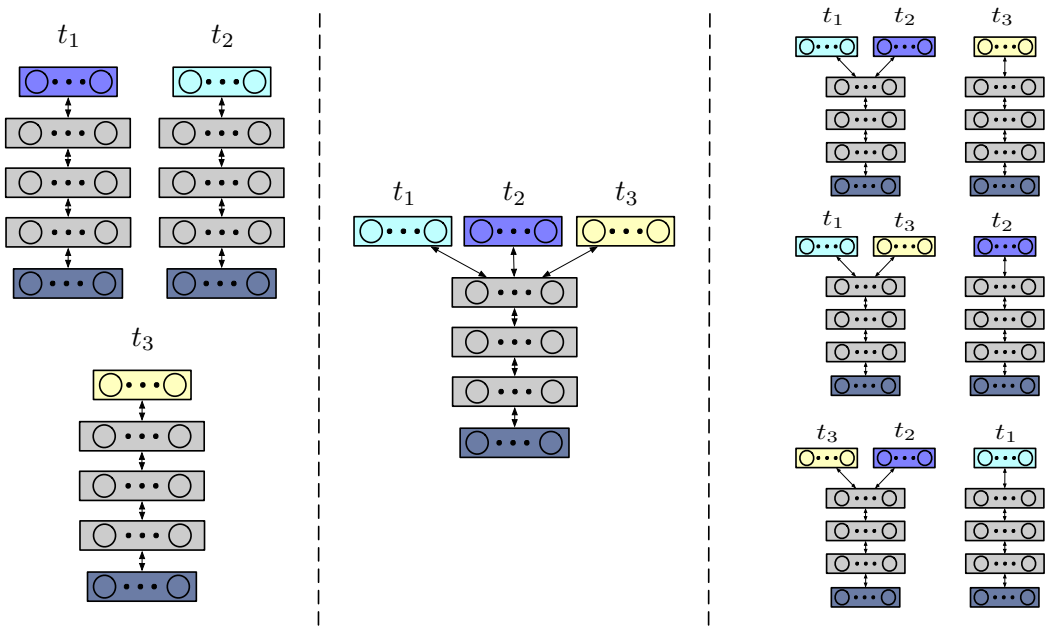


Figure 4.5.2: Different deployment configurations for 3 audio analysis tasks (t1, t2, and t3). We can have independent models for each task (left), all tasks can share the layers of the DNN (middle), or we can deploy a pair of tasks with a shared representation and a task with an independent model (right).

process. When the number of audio analysis tasks is small, the search policy could simply be a brute-force exhaustive search, which is what we adopt by default. With the increase in number of tasks, the search space grows exponentially. With a total of three related tasks the number of different combinations is five as shown in Figure 4.5.2 – one possibility is having a shared representation across all tasks; another one is having individual models for each task as is common practice; there are also 3 configurations where two of the tasks share the hidden layers of a DNN, and one is separate. An example policy to restrict the search space is to explore only configurations that have no more than 2 DNNs in total across tasks.

The optimisation process works as follows. For each combination of tasks in our search policy we build various sized networks of the topology supplied as a hyper-parameter. We train the DNNs and obtain for each audio analysis task the accuracy observed on its validation set. Given that the configuration complies with the memory limit, we use the accuracy criterion to compare the current configuration with the best one found so far. At the end, we have one configuration that defines how audio analysis tasks are combined with the sizes and parameters of the DNN(s).

4.5.3 Multi-task classification efficiency

We now study the performance benefits of using DNN architectures with shared hidden layers for sound classification as opposed to models built separately for each audio sensing task. The most prominent results from our experiments are:

- *Accuracy robustness to multi-task mixing.* Multi-task audio inference performs surprisingly well when various combinations of related but distinctly different tasks are mixed together to have a shared representation. On average across multi-task configurations we never observe accuracy drops larger than 1.5% compared to the best performing similarly sized individual models.

- *Improvements to runtime, energy and memory.* When sharing the models across different combinations of audio tasks, the performance gains in terms of reduced runtime and memory footprint are on average 2.1 times.

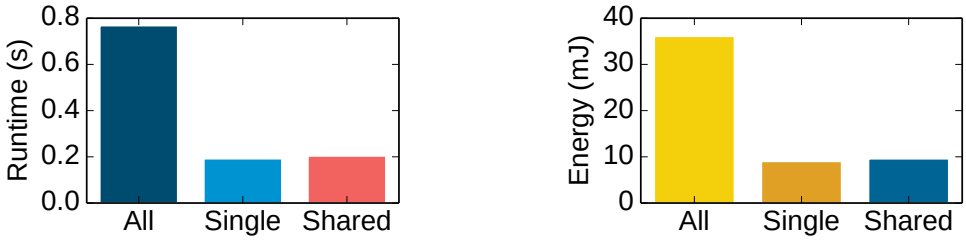
- *Scalability to number of integrated tasks.* With our multi-task optimisation framework, classification efficiency scales extremely well with the number of related tasks that are mixed together. Due to the high degree of sharing with n audio tasks the gains in runtime, energy and memory are almost n-fold in the best case when all tasks are combined. Critically, this often comes with little to no loss in accuracy.

Accuracy. A key issue accompanying the many techniques that typically compress DNN model sizes is the extent to which the original model accuracy is changed. Often, performance benefits in the runtime dimension go hand in hand with accuracy drops that may or may not be acceptable. In this case, however, we observe that training shared-hidden-layer DNNs does not compromise accuracy when pairing different combinations of audio analysis tasks, i.e. the accuracy remains comparable with no significant reductions. In fact, as demonstrated in Table 4.5.1, for some of the audio sensing tasks there are even tangible accuracy boosts when using all tasks to train a shared DNN. For instance, the Stress Detection score raises from 80.7% to 85.4%, and the Emotion Recognition from 81.5% to 83.4%. This phenomenon can be explained by the fact that the DNN shared hidden layers are non-trivial feature transformations that summarise the audio input: using multiple datasets simultaneously results in overall more data being used to learn discriminative feature transformations. The multi-task transfer learning is especially beneficial for the smaller datasets such as the Stress Detection and Emotion Recognition ones, where the learning process is augmented with audio knowledge (training samples) from other datasets.

Note that if we were to reduce the network size to fit all models in scarce DSP memory, we would typically pay the price of lower accuracy. Instead, having a shared network model allows us to use a much larger model within the same amount of space and with the same number of concurrent sensing tasks, without the need to compromise accuracy because of hardware constraints.

	Single DNN	Max-task DNN	Average Mixing Accuracy
Speaker Identification	85.8%	85.1%	84.7% (±1.2%)
Emotion Recognition	81.5%	83.4%	85.8% (±1.6%)
Stress Detection	80.7%	85.4%	83.3% (±2.0%)
Ambient Scene Analysis	85.2%	84.8%	83.7% (±1.0%)

Table 4.5.1: Accuracy of the multi-task DNN models with shared hidden layers compared against individually built models. In all cases the DNN structure has 3 hidden layers with 512 nodes per layer. All DNNs are trained with the same hyper-parameters and the same number of epochs. The Max-task DNN combines all audio analysis tasks. The average mixing accuracy shows the mean performance when the corresponding audio task is mixed with the other tasks in pairs or triples. Numbers in brackets show the standard deviation.



(a) Runtime

(b) Energy

Figure 4.5.3: Runtime and energy for performing DNN classification on the low-power Hexagon DSP when all single models (All), a single model (Single), or the DNN with shared hidden layers (Shared) are run. Single models are run sequentially where each model does each task separately, and the shared one leverages the multi-task design to output all inferences.

Runtime and energy reductions. The total runtime and energy for performing audio classification for several tasks with a shared DNN is comparable to that of evaluating a single DNN model for one task. For example, the runtime and energy spent cumulatively for all tasks can be up to 4 times less when using a 4-task DNN with shared hidden layers as can be seen from Figures 4.5.3a and 4.5.3b. On average the performance improvements when pairing different subsets of the audio analysis tasks are 2.1 times as illustrated in Table 4.5.2. When combining pairs of tasks, for instance, there are 12 different configurations that can be explored – 6 of these configurations feature a paired DNN plus 2 individual networks, and the other 6 consist of 2 coupled DNNs both with shared hidden layers. On average for these configurations inferences are obtained approximately 1.7 times faster while also critically consuming that much less energy for the computation. These massive reductions in the runtime and energy footprint of the audio tasks are possible because the majority of the compute time for the DNN forward propagation is spent in the hidden layers. The softmax output layer typically has much fewer nodes which match the number

	Single	**Pairs**	**Triples**	**Quadruple**	**Global**
Reduction factor	1.0x	1.7x	2.0x	4.0x	2.1x
# of configurations	1	12	4	1	18

Table 4.5.2: Total average reductions in the runtime and energy when the audio tasks are combined in pairs, triples, or quadruples. Total number of different configurations is 18 with an average reduction of 2.1 times across all configurations.

	3 hidden layers 256 nodes each	**3 hidden layers 512 nodes each**	**3 hidden layers 1024 nodes each**
Single	0.73MB	2.6MB	9.2MB
Shared	0.80MB	2.7MB	9.4MB
All	2.92MB	10.4MB	36.8MB

Table 4.5.3: Memory required by the various DNN models as a function of the size of the hidden layers.

of inferred sound categories, and is just one out of the multiple layers of the deep network architecture.

A prominent result that can be extracted from these observations is that *using joint representations across audio tasks scales extremely well with the increase in number of tasks.* As demonstrated, for n tasks the expected performance gains are n-fold in the best case when our optimisation process decides to integrate all tasks. More importantly, we have also shown this coupling of models can be achieved with very little to no loss in accuracy across the audio analysis scenarios despite their largely disparate types of inferences made.

Memory savings. In terms of memory, a 4-task DNN with hidden layers can occupy up to 4 times less space compared to laying out the parameters for multiple individually built networks. The total amount of memory occupied by this space-optimised DNN is 2.7MB when the network has 3 hidden layers with 512 nodes each, as shown in Table 4.5.3. If we were to use single networks that are almost similarly sized for each of 4 audio sensing tasks, we would need more than 10MB of memory which exceeds the runtime memory limit of 8MB for the Qualcomm Hexagon DSP. Instead, the shared representation allows us to save scarce DSP memory resources.

Shared representation vs alternatives. The types of audio analysis tasks closely match the ones studied by the DeepEar audio sensing system [127] that is deployed on the same series of Snapdragon embedded platforms. Whereas DeepEar also resorts to a DSP implementation to enable efficient continuous background sensing, its architecture is limited in several ways. First, the range of the supported audio inference tasks at runtime is lower which is a natural consequence of modelling each audio analysis task with a dedicated deep network model. In its deployment, DeepEar is forced to leave out one of the investigated four tasks and downsize another just to meet the embedded memory

requirements. In contrast, a shared representation is extremely scalable since it allows an arbitrary number of simple audio sensing tasks to be integrated together, as long as accuracy is preserved to be sufficiently high. In fact, as we have demonstrated, accuracy not only remains comparatively the same, for some tasks it can even be superior due to the advantages of adopting extra training data from related tasks. Further, downsizing the models as is done by DeepEar may result in unwanted accuracy drops depending on the complexity of the inferences or the dataset. Our ambient scene analysis task features a large-scale dataset and supports a much wider range of 19 inference classes compared to the corresponding 4-class ambient sound classification used by DeepEar. We find that a similarly sized DNN (3 layers with 256 nodes each) trained identically, and adopting the best performing filter bank summaries as input, on this more comprehensive dataset would yield an accuracy of 82% which is lower compared to the shared-layer alternative we propose here (84.8%). Finally, the shared model representation enables the total inference time and energy consumption to be about 2 to 3 times lower compared to DeepEar. Our findings suggest that new cross-task modelling approaches such as the one introduced here are necessary to overcome limits of mobile hardware.

4.6 Related work

The DNN classification engine we built for the DSP of mobile SoCs is most closely related to two active areas of deep learning research: optimising deep models for embedded platform constraints and forms of multi-task learning for deep neural networks.

In particular, due to memory being a key bottleneck for embedded systems and deep models considerable efforts are under way to investigate different forms of model compression. A common approach is to use SVD to approximate matrices such as those that describe the weights between layers, usage of these methods for audio modelling have been common for quite some time [191, 103]. Related approaches use hash functions to also aggregate parameters [71] and as well as use forms of quantisation towards similar ends [96]. Instead of approximating the network architecture, alternative methods seek to prune the connections of the network determined to be unimportant [102]. However, collectively these methods sit as complementary to the ones we explore here. We study a framework that 1) replaces traditional models used in mobile sensing (i.e. GMM, DT) with DNNs, and 2) replace multiple networks with fewer networks that perform more than one task through multi-task learning as a means to reduce resource consumption at inference time. Therefore, any of these approaches for optimising a single network also remain applicable to our framework.

We are not the first to attempt to train a deep network with a shared set of hidden layers that is able to support multiple learning tasks. However, we were unable to find prior examples of this multi-task learning being used to overcome the limits of embedded hardware. Motivations related to model robustness have been known for quite some time;

empirical findings demonstrate for instance exposure to related – yet still distinct – inference tasks can lead to the learning of robust features and assisting in the generalisability of the whole model. Domains where such results are seen include: multi-language audio discrimination [111] and information retrieval tasks [139]. In Collobert and Weston [77], a deep multi-task learner is shown that provides inferences including: parts-of-speech tags, named entity tags, semantic role assignment. This approach has also proven to be effective in blending complementary objectives towards final discrimination; one example of this being Li et al. [133] where image segmentation and saliency detection are treated as two tasks within a multi-task deep framework for salient object detection. Although adopting such techniques does not always lead to a reduction in network footprint, that is necessary to reduce resource consumption – for example, when non-shared task-specific components of the architecture are larger than those that are shared. For this reason, we examine an approach where shared layers dominate, which maximises the reduction in resource usage (like memory), and in terms of accuracy simply attempt to *match* the levels seen in models with an identical architecture but focused on a single audio analysis task.

4.7 Conclusions

In this chapter, we have investigated the potential for techniques from deep learning to address a number of critical barriers to mobile sensing surrounding inference accuracy, robustness and resource efficiency. Significantly, we performed this study by implementing a DNN inference engine by broadly using the capabilities of modern mobile SoCs, and heavily use the DSP. Our findings showed accuracy boosts, likely increases to inference robustness, and acceptable levels of resource usage, when DNNs are applied to a variety of mobile audio sensing tasks such as emotion and speaker recognition. Furthermore, we highlighted beneficial resource characteristics (e.g., class scaling, cloud offloading) missing from models in common use today (e.g., GMMs). In addition to all these base model benefits, we discovered that extensions to deep learning techniques can be applied for even larger performance gains when multiple related audio analysis tasks are deployed together on the DSP. In particular, a multi-task learning approach where DNN representations are shared among tasks affords significant opportunities to increase inference-time resource efficiency. Our experiments showed, on average, a 2.1 times reduction in runtime, energy, and memory is possible under our multi-task framework when assuming common combinations of four typical audio analysis inferences. Critically, we demonstrated that for the modest-scale DNNs able to be supported in representative low-power DSP hardware this approach does not reduce accuracy for any tested audio analysis task, despite such resource gains.

This chapter is a step towards understanding how deep learning can be used in mobile contexts and provides a foundation for more complete studies. It explored deep-learning-based algorithm specialisations targeted at a low-power co-processor, but more research

is needed to understand how other processors can be optimally exploited to benefit from deep learning and other techniques in common mobile sensing usage. This is the focus of Chapter 5 that turns to the GPU for audio sensing offloading.

Chapter 5

GPU sensing offloading

5.1 Introduction

The latest in mobile hardware boasts a range of heterogeneous processors, from CPUs to DSPs and GPUs. The relative recency of such prominent mobile resource diversity coupled with programmability challenges, however, has precluded comprehensive exploration of how different types of computation on these processors would benefit mobile apps. In the previous chapter we made an attempt to lower the barrier for the adoption of low-power DSPs for complex deep-inference audio sensing tasks. Specifically, we implemented a processor-specialised deep learning engine that can execute algorithms constrained in time and space to match the DSP hardware. In this chapter we continue our investigation into building mobile sensing algorithms targeted at a concrete heterogeneous resource: we turn to the exploitation of the mobile GPU.

GPUs are the method of choice for executing high computational loads and accelerating compute-intensive applications in domains such as computer vision [185, 72, 112] and deep learning [38, 52, 181]. But GPUs like any complex processor architecture need to be used smartly to maximise their throughput and efficiency. There have been extensive studies for graphics and games [185, 112, 185, 158] including mobile [72], but analysis has largely ignored other general-purpose GPU computations on a mobile device such as audio apps that rely on the power-hungry microphone sensor.

Computational offloading to cloud [75] or low-power co-processors, as we have done in the previous two chapters with DSP.Ear and our deep audio inference engine, are obvious solutions to try and keep audio apps functional on the mobile device. However, it is not immediately obvious whether accelerating these sensing apps via GPU offloading will result in energy-justified performance boosts compared to the alternatives. Questions that we investigate in this chapter are: *What trade-offs do we get in terms of speed and energy if we express audio sensing algorithms in a GPU-compliant manner? How can we best take*

advantage of the general-purpose computing capabilities of mobile GPUs to offload audio processing? When should we prefer GPU computation to a resource such as a low-power DSP or cloud?

In this chapter we present a GPU offloading engine that leverages parallel optimisation techniques that allow us to auto-tune the performance of audio routines. Without such optimisation, naively parametrised GPU implementations may be up to 1.5 times slower than multi-threaded CPU alternatives, and consume more than 2 times the energy of cloud offloading. To the best of our knowledge, we are the first to identify generalisable GPU parallel optimisations that are *applicable across multiple algorithms* used in audio sensing. Previous efforts [197, 97] have focused on isolated use cases with techniques heavily relying on the specifics of the concrete application scenario (e.g., automated speech recognition).

Chapter outline. We begin the chapter by discussing the general GPU execution model and highlighting the challenges in deploying audio sensing apps in Section 5.2. Section 5.3 briefly describes our GPU audio optimisation engine, and Section 5.4 details our performance tuning techniques. Key audio-specific structural and memory access parallel patterns are introduced here to allow us to automatically tune the GPU performance boost of audio pipelines. These patterns 1) increase the data parallelism by allowing a larger number of threads to work independently on smaller portions of the audio input stream; and 2) strategically place data needed by the threads into GPU memory caches where access latency is lower and the data can be reused.

Section 5.5 elaborates on some of the implementation details. In Section 5.6 we extensively evaluate our framework to find that *for time-sensitive audio apps, and when energy is less of a concern, there is no better option than using GPU optimised audio routines.* Algorithms tuned for the GPU can deliver inferences an order of magnitude faster than a sequential CPU deployment, and 3.1 or 6.5 times faster than cloud offloading with good connectivity for deep audio inferences such as Speaker Identification and Keyword Spotting, respectively. Perhaps more surprising, *for tasks that are more continuous but tolerate short delays (of* 10-20 *seconds) GPU is also the best choice.* When raw data is accumulated for batched processing, algorithms optimised for the GPU begin to consume less energy than both cloud offloading with fast connections and a low-power DSP. The batching delays are sufficiently short to support the operation of not only life-logging style behaviour monitoring apps that tolerate large delays, but also apps that deliver context aware services and notifications such as conversation analysis [94, 130].

Section 5.7 discusses major issues concerning our design, whereas Section 5.8 outlines related work. We conclude the chapter with a short summary of contributions in Section 5.9.

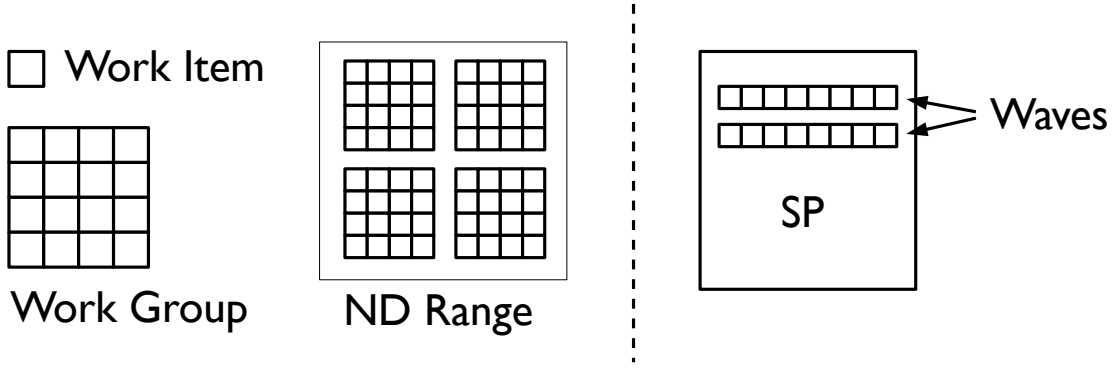


Figure 5.2.1: OpenCL thread model and a GPU Shader Processor. The SP features 2 waves of 8 work items each, it can run 16 threads in total simultaneously. The work items of one work group are executed on a single SP.

5.2 GPU execution model and challenges

In this section we will elaborate on the GPU execution model and highlight some of the challenges it presents for audio sensing. We use OpenCL's terminology [42] and Qualcomm Adreno GPU [2] as an example for GPU architecture and programming model, but our discussion and conclusions apply equally to other GPU platforms, such as NVIDIA with CUDA [37].

To an OpenCL programmer, a computing system consists of a *host* that is traditionally a CPU, such as the Snapdragon 800 Krait CPU, and one or more *devices* (GPU) that communicate with the host to perform parallel computation. Programs written in OpenCL consist of host code (C API extensions) and device code (OpenCL C kernel language) – communication between the two is performed by issuing commands to a command queue through the host program space. Example commands are copying data from host to device memory, or launching a *kernel* for execution on the device. Kernels specify the data-parallel part of the program that will be executed by the GPU threads. When a kernel is launched, all the threads execute the same code but on different parts of the data.

Thread model and compute granularity. All the threads generated when the kernel function is called are collectively known as a grid (or ND Range) and are organised in a two-level hierarchy independently from the underlying device architecture. Figure 5.2.1 illustrates this organisation. The grid consists of *work groups* each containing a set of threads known as *work items*. The exact thread scheduling on the GPU is decoupled from the work groups and is vendor specific although it shares a lot of similarities among GPU varieties. Switching from a group of work items to another occurs when there is a data dependency (read/write) that must be completed before proceeding and is done to mask these IO latencies.

One of the challenges of implementing GPU-friendly algorithms is providing the right level

of work item granularity. If the GPU threads are too few, the GPU will struggle with hiding memory access latency due to not being able to switch between compute-ready threads while others are stalled on a memory transaction. Audio sensing algorithm execution revolves around the analysis of frames, and a natural candidate for data parallelisation is to let each work item/thread analyse a frame. However, the number of frames in an inference window is on the order of tens to hundreds, whereas the GPU typically requires thousands of threads for any meaningful speedups to begin to appear. A challenge is organising the audio algorithm execution in a way that allows more work items to perform computation.

Managing memory-bound audio kernels. Work items have access to different memory types (global, constant, local/shared, or private) each of which provides various size vs. access latency trade-offs. Global memory is the largest but also the slowest among the memories. Private memory is exclusive to each work item and is very limited in size, whereas the shared memory is larger and accessible by all work items in a group. Often, a *compute to global memory access (CGMA) ratio* is used as an indicator of the kernel efficiency – the higher the ratio is, the more work the kernel can perform per global memory access, the higher the performance.

Typical algorithms used in audio sensing need to read a large number of model parameters which they apply to the frame data, but the number of floating point operations per read is relatively low making audio kernels *memory-bound*. In order to squeeze maximum performance out of the mobile GPU (highest speed and thus lowest energy consumed), algorithms will need to reduce the global memory traffic by intelligently leveraging the smaller but lower access latency memories (shared and private). The challenge is enabling appropriate memory optimisation strategies that keep the CGMA ratio high while maintaining a suitable level of granularity for the work items.

Summary. GPUs are a powerful platform for general-purpose computing programmed by language abstractions such as OpenCL and CUDA. An unanswered challenge is how and what performance control techniques we can leverage that depend on the algorithm semantics rather than a concrete hardware configuration.

5.3 Optimisation engine overview

To address the GPU deployment challenges presented in the previous section, we build a library of OpenCL auto-tunable audio routines that form the narrow waist of audio processing pipelines found in the mobile sensing literature (e.g., filter bank feature extraction, GMM and DNN inference). This library builds upon a set of structural and memory access techniques that expose a set of tunable audio model-dependent *control-flow parameters* which we can control in a pre-deployment step with an optimisation engine. The goal of this engine is to provide the best match between the domain-specific library implementation

Pattern	Type	Applicability
fan-out	structural	GMM, DNN, feature extraction sub-phases
vectorisation	memory access	ubiquitous
sliding window	memory access	DNN, pre-emphasis
tiling	memory access	GMM, filter banks

Table 5.3.1: Parallel optimisation patterns taxonomy.

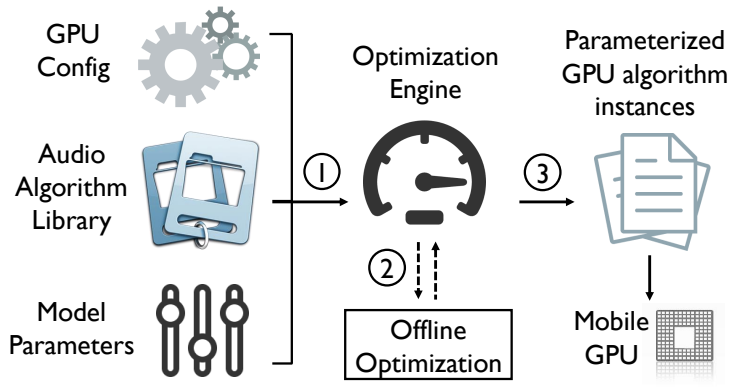


Figure 5.3.1: High-level optimisation engine workflow.

and mobile GPU hardware constraints. The engine helps to avoid cumbersome hand tuning, instead automatically parametrises the audio kernel routines with large performance boosts for some of the algorithms. This requires *zero* change in the kernel code itself, the parameters are passed through OpenCL commands as kernel arguments at runtime. A high-level workflow is illustrated in Figure 5.3.1.

The pre-deployment step is a three-staged process, where the engine first loads as input audio model parameters such as the DNN layout and queries the GPU device specification (e.g., GPU shared cache size) in order to be able to estimate optimum values for the GPU algorithm control-flow parameters. In the second stage, the engine performs the optimisation step by solving a series of linear and quadratic equations and outputs a configuration file with GPU-kernel parameter values required by our audio library. The third stage is loading the values from the locally persisted config file to parametrise the audio algorithms upon initialisation of concrete sensor apps.

To provide high-performance parallel implementations, we build the techniques listed in Table 5.3.1 that enable control over the following parameters. Empirically we found that for memory-bound audio kernels, these provide a sweet spot of tunable but not too complex parameters with a big impact on mobile GPU performance:

- **frame fan-out factor (ϕ)** – defined as the total number of audio frame processing GPU threads. A higher value results in an increase in the number of concurrent

threads that can work independently.

- **per-thread compute factor (κ)** – defined as number of computed output values per GPU thread. By optimising this the engine attempts to maximise the number of computations each thread can perform relative to its memory reads and writes (favouring compute-bound operation instead of memory-bound).

Manipulating the first parameter is achieved in our library through *the frame fan-out* structural optimisation pattern. The core idea behind it is to split the audio analysis so that each GPU thread can work on a subset of the output values extracted from an audio frame. The second parameter is tightly related to a set of memory access patterns that reduce expensive global memory traffic and increase the per-thread compute factor. These techniques are: 1) *Vectorisation* that consolidates slow global memory reads into a single load operation which is possible thanks to the sequential nature of accessing values from the audio stream. In our examples, the engine selects larger read batches and can fetch into the thread registers up to x values from memory, where x is vendor specific (for Qualcomm Adreno $x = 16$, for NVIDIA Tegra X1 $x = 4$ [40]). 2) *Memory Sliding Window* and *Memory Tiling*: the techniques allow threads to collaboratively load data into shared memory where this data can be subsequently reused with lower latency to produce multiple output values. These are critical optimisations since global memory access is arguably the most prominent bottleneck we observe in the widely used audio classification and feature extraction algorithms.

5.4 Parallel control-flow optimisation

In this section we detail the structural and memory access parallel patterns that enable the optimisation engine to parametrise the audio routines in our library. Throughout the exposition we show how the techniques are applied to the two most popular types of audio pipelines presented in Chapter 2: GMM-based and DNN-based. As concrete examples we focus on the computationally more demanding applications the classification models of which have a large number of parameters and are better equipped to reap benefits from parallel execution. Namely, these apps are Speaker/Emotion Recognition (GMM-based) and Keyword Spotting (DNN-based).

5.4.1 Inter- and intra-frame fan-out

This pattern controls the level of data parallelism by allowing a larger number of concurrent threads to perform independent computations on the input data. We can support such a mode of operation thanks to the way audio pipelines process frames – repeatedly mixing the frame samples/coefficient with multiple parameters (GMM mixtures, DNN network weights). Independent computations are performed not only among different frames but

also within a single frame, a phenomenon which we call the *frame fan-out*. This allows the total amount of threads, or *fan-out factor* (ϕ), to be relatively high. It can be computed as follows: $\phi = \frac{n*\nu}{\kappa}$ where n is the number of frames, ν is the total number of output values per frame, and κ is the number of computed values per GPU thread (*per-thread compute factor*). This structural optimisation is applicable across both feature extraction (filter bank computation) and classification phases. The next two examples illustrate how this pattern can be applied:

GMM fan-out. The input for this classification phase is the extracted feature coefficients from all frames. In the Speaker Identification pipeline there are 32 PLP frame features and a total of $n = 500$ frames per inference window (one frame every 10ms for 5s). Each GMM has $\nu = 128$ mixtures each of which computes a probability score by mixing the 32 PLP coefficients from a frame with the parameters (mean and variance) of 32 Gaussian distributions. With a per-thread compute factor of $\kappa = 1$, we could let each OpenCL work item estimate the probability score for one mixture per frame resulting in a fan-out factor of $\phi = 500 \times 128$. We enable the kernel to generate this massive number of work items by letting them write intermediate scores to global memory and a separate kernel is launched to sum the scores.

DNN fan-out. Similarly, the input data for the Keyword Spotting DNN classification is the extracted filter bank energies from the frames. In a 1-second inference window there are a total of $n = 100$ network propagations (one per new frame every 10ms). A DNN kernel computes partial results across all input frames by performing the feed forward propagation for one layer across the frames simultaneously. Multiple kernels are launched each of which computes the node activation values for the next layer. With $\nu = 128$ nodes in the hidden layers we could let each OpenCL work item compute the activation for one node per frame offset ($\kappa = 1$) resulting in a fan-out factor of $\phi = 100 \times 128$.

The role of the optimisation engine is to provide optimum values for the control-flow parameters ϕ and κ. n and ν are determined directly by the audio model specification, whereas the final value of ϕ depends on κ, or the amount of work each GPU thread is assigned. The engine tunes the per-thread compute factor κ since we observe that *maximum GPU performance may not necessarily be reached when parallelism is highest* (the fan-out factor ϕ reaches its maximum when $\kappa = 1$). The memory access patterns in the following section enable each GPU thread to perform more computation ($\kappa > 1$) relative to the number of its memory reads and writes.

5.4.2 Memory access control

Tuning audio kernel performance with the per-thread compute factor κ is closely related to how memory access is managed by the threads in a work group. Increasing the number of computations per thread per global memory access and thus finding optimum values for κ depends on maximum exploitation of the faster but limited in size GPU memories. We

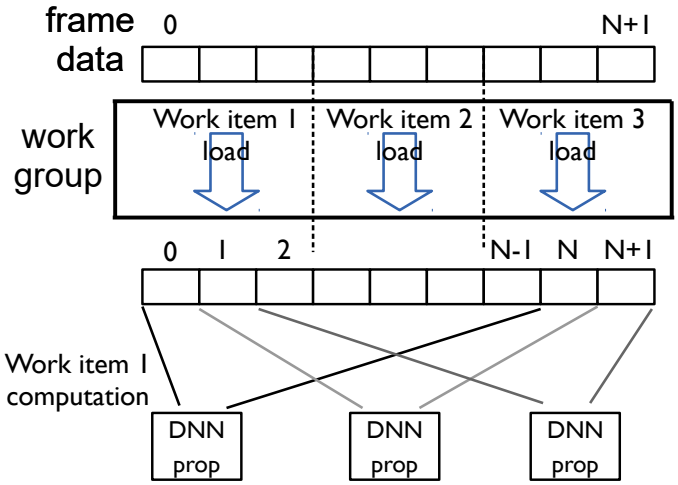


Figure 5.4.1: Sliding window example. The DNN activation of one node in the first layer (denoted by DNN prop) is performed by each work item in a work group. The required data for one such computation is N frames. The figure explicitly shows the computation for work item 1 which is performed for 3 frame offsets from the accumulated frame input data (0 to $N-1$, 1 to N, and 2 to $N+1$). Each work item in a group can load a part of the data needed for the DNN layer activation, but will use all data loaded by the peers in the group, as work item 1 does. A work item computes the activation for one node in a layer (DNN prop), but since more data is accessible from the collaborative loading, the item can reuse its parameters to compute the activation for the same node for 3 different frame offsets.

discuss several key strategies, enabled by the specifics of digital signal processing, to lower the number of global memory operations. These strategies either 1) batch global memory transactions into fewer operations, or 2) let the *threads in a work group collaboratively load data needed by all of them into shared memory where access latency is lower.*

Vectorisation. When kernels read the input data features or parameters, for instance, they access all the adjacent values in a frame. As a result, the memory access can be vectorised and consolidated with vector load operations that fetch multiple neighbouring values at once from global to private memory. For example, when a thread requires the 32 PLP coefficients from a frame, it can use the OpenCL *vloadx* operation to issue 2 reads with 16 values (*vload16*) fetched simultaneously instead of performing 32 reads for each coefficient separately.

Shared memory sliding window. Often the input raw audio or feature stream is processed in sliding window steps, i.e. the input is divided into overlapping frames over which identical computations are performed. Example scenarios where this type of processing is commonly applied are the feature extraction (PLP, MFCC [88]), or the classification of the feature stream into observed phenomena (as is the case for the DNN Keyword Spotting,

Algorithm 1 Shared Memory Use Kernel Template

```
1: Input: (i) Pointer to input buffer (in), (ii) Pointer to output buffer (out), (iii) Thread local id (lid),
   (iv) Thread group id (gid), (v) Shared memory maximum size (max_s).
                                                                          ▷ Collaborative data load:
2: loadOffset ← compute_load_offset(lid)
3: inputOffset ← compute_input_offset(gid, lid)
4: __shared floatN data[max_s]                                  ▷ shared memory declaration
5: if loadOffset < max_s then
6:     data[loadOffset] ← vloadN(0, &in[inputOffset])                          ▷ vectorised
7: barrier(CLK_LOCAL_MEM_FENCE)          ▷ wait for all threads to finish loading data
                                                                  ▷ Shared data processing:
8: for (i = 0; i < x; ++i) do
9:     localDataOffset ← compute_local_offset(lid, i)
10:    result ← process(&data[localDataOffset])
11:    outputOffset ← compute_output_offset(gid, lid, i)
12:    out[outputOffset] ← result
```

see Figure 5.4.1). The data overlap is usually quite substantial – the feature extraction phase for the Speaker Identification pipeline, for instance, uses 30ms frames (240 samples at a sampling rate of 8kHz) with a 10ms frame rate (an offset of 80 samples) resulting in a 66% data overlap between subsequent frames. We can exploit this property of the audio stream processing to let the threads in a work group collaboratively load a larger chunk from the input spanning samples from multiple frames into shared memory (where access latency is lower), and let each thread reuse its loaded parameters by applying them against several offsets from the input. The higher the data overlap for adjacent frames, the larger the opportunity the threads have to load more adjacent regions with fewer read operations, and the more computation they can perform per global memory read. The pattern enables the control of the per-thread compute factor κ by increasing the CGMA ratio of the kernel operation.

Algorithm 1 shows example kernel pseudo code where the work items cooperate to load data into shared memory. The key advantage is that each thread can use a single vectorised fetch which is only a small proportion of the actual data needed from global memory. Collectively, however, all threads are able to load the data needed by their peers in the work group. The number of adjacent input regions x over which the threads in a work group perform computations are limited by the maximum size of the shared memory reserved to a work group. For Adreno 330 that size is 8KB. The optimisation engine estimates the maximum x as a function of the model size and shared memory constraints, the pattern exposes x as the per-thread compute factor ($\kappa = x$).

Shared memory tiling. As discussed in the description of the frame fan-out in Section 5.4.1, when audio processing pipelines work on a frame they usually produce multiple output values (e.g., feature coefficients or probability scores) by combining the frame data with multiple parameters. The procedure can be treated as *generalised dense matrix-matrix multiplication* with the two matrices being: 1) an input matrix $I_{(n,d)}$ with n input frames

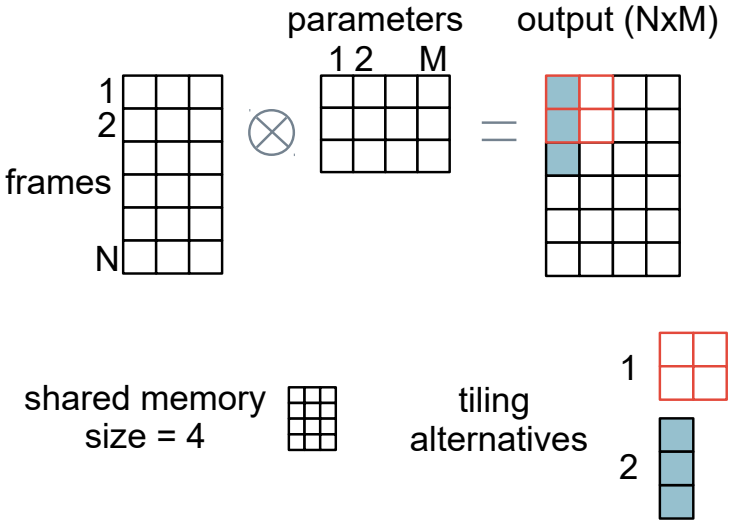


Figure 5.4.2: Tiling example. There are N frame data rows and M parameter columns that can be combined to yield N×M output values. The shared memory size of a work group is limited to 4 frame rows or parameter columns in total. There are 2 tiling alternatives. Alternative 1 fills shared memory by loading data for 2 frames and 2 parameters leading to 4 computed values in total from the work group. Alternative 2 loads data for 3 frames and 1 parameter leading to 3 computed values in total.

each of which has d elements (samples/coefficients); and 2) a parameter matrix $P_{(k,d)}$ with k parameters each of which has dimensionality d. The result of the combination of the two matrices is a matrix $O_{(n,k)} = I_{(n,d)} \bigotimes P^T_{(k,d)}$ where $\bigotimes$ can be a generalised operation that performs a reduction over d elements from the two matrices (a row from O and a column from P^T). Prominent applications of this operation can be found in the computation of the filter bank coefficients, the GMM mixture probability estimation, and the DNN network propagation. An example reduction $\bigotimes$ used in the GMM classification stage is shown in the following equation:

$$o_{ij} = -\frac{1}{2}(g_j + \sum_{0 \leq s < d} (x_{is} - m_{js})^2 v_{js}) \tag{5.1}$$

where o_{ij} is one element from the output matrix; m_j, v_j, and g_j are the Gaussian parameters of a mixture component; and x_i are feature values from frame i.

The straightforward implementation of a GPU kernel to compute matrix O would be to let each thread compute one output value o_{ij} and load independently an input row i and a parameter column j. However, a strategy for reducing global memory traffic is to introduce a model-specific version of *tiled algorithms* used in matrix-matrix multiplication [121]. The core idea is to let the work groups of a kernel partition the output matrix into *tiles* so that the total data for each tile fits into shared memory. Our goal is to have the threads in a work

group collaboratively load both input data and parameters into shared memory in a way that maximises the number of computations per global memory read. This can be achieved if the number of computed values ({number of frames} N_f times {number of parameters} N_p) per loaded data is as large as possible for the entire work group. Figure 5.4.2 illustrates the pattern in operation.

5.4.3 Parameter estimation

Vector size x. The engine selects the vector size for batched memory reads by querying with OpenCL commands whether the audio kernel can successfully be compiled with the given value for x. The engine enumerates the possible values for x in descending order and picks the highest value under which the kernel successfully compiles. Compilation may fail when the size of the vectorised loads exhausts the thread register space.

Memory sliding κ. Optimising this parameter involves solving a linear equation with respect to the GPU shared memory size, and input model parameters. If S_M is the total number of values the shared memory can accommodate, S_F is the input region size, and r is the frame offset in number of input values, then the maximum κ is computed in the following manner: $\kappa = \left\lfloor \frac{S_M - S_F}{r} \right\rfloor + 1$.

Memory tiling κ. The optimisation engine makes a two-staged decision: whether to activate this pattern and if activated how to best parametrise it. The decision is based on the type of the model used for audio analysis (filter banks, GMM, or DNN), input model dimensions (size of the model parameters), maximum work group size, and shared memory size. The optimisation engine estimates the work group size and an optimum number of output matrix values each thread in the work group should compute so that global memory accesses are minimised. This is implemented by solving a quadratic equation with respect to N_f and N_p under the shared memory constraints. In our examples, the pattern would be activated for the filter banks and GMM computation, but not for the DNN where the size of the network is prohibitively large for any meaningful subset to be effectively exploited from shared memory.

Frame fan-out ϕ **and work group size**. The engine estimates the fan-out factor in a final step by reading the specified audio model parameters (e.g., number of frames) and having κ determined in a prior step. We strive to select the largest work group size possible. Similarly to vectorisation, this can be done by exhaustively trying to compile the kernels with values up to the maximum size allowed. Sizes are enumerated in multiples of the *preferred group multiple parameter* which can be queried for a GPU with OpenCL commands.

5.5 Implementation

Hardware and APIs. In a manner similar to what we have done in Chapters 3 and 4, we prototype the audio sensing algorithms on a Snapdragon 800 Mobile Development Platform for Smartphones (MDP) [46] with an Android 4.3 Jelly Bean OS featuring Adreno 330 GPU present in mobile phones such Samsung Galaxy S5 and LG G Pro2. All GPGPU development is done with OpenCL 1.1 Embedded Profile and we reuse utilities for initialising and querying the GPU from the Adreno SDK that is openly available [2]. In addition, we reuse baseline versions of the audio sensing pipelines for the Qualcomm Hexagon DSP [43] built for DSP.Ear and our deep audio engine. The DSP has 3 hardware threads and we use the *dspCV* thread pool library shipped with the Hexgagon SDK to also build multithreaded versions of these algorithms for the DSP. Interfacing between the CPU and DSP is done as described in Chapter 3 with the Android Native Development Kit (NDK). Multithreaded CPU versions are built by using C++11 threads – we leverage the 4 cores available to the Snapdragon Krait CPU. Again, power measurements are performed with a Monsoon Power Monitor [34] attached to the MDP. The classifier models are trained with the HTK toolkit [23] for the Speaker/Emotion Recognition and the Theano python library [181] for the Keyword Spotting.

Kernel compilation. Building the OpenCL kernels can be done either via reading the sources from a string and compiling them at runtime or by pre-compiling the source into a binary. We use pre-compiled binaries that drastically reduce the kernel load time and that need to be produced once per deployment.

5.6 Evaluation

In this section we provide an extensive evaluation of the parallel optimisations and performance of representative audio sensing algorithms when deployed on the mobile GPU. The main findings are:

- Optimising the control-flow parameters is critical – naively parametrised GPU implementations may be up to 1.5 times slower than multithreaded CPU baselines and consume more than 2 times the energy of offloading batched computation to the cloud.
- The total speedup against a sequential CPU implementation for an optimised GMM-based and a DNN-based pipeline running entirely on the GPU is 8.2 and 13.5 times respectively, making the GPU the processing unit of choice for fast real-time feedback. The optimised GPU is also 3 to 4 times more energy efficient than sequential CPU implementations, but can consume up to 3.2 times more energy than a low-power DSP.

- After a certain batching threshold (10 to 20 seconds) of computing multiple inferences in one go, optimised GPU algorithms begin to consume less energy than cloud offloading with good throughput (5Mbps and 10Mbps), in addition to obtaining inferences 3.1 and 6.5 times faster for the GMM and DNN-based pipelines respectively.

5.6.1 Experimental setup

Experiments are performed with the display off and the algorithms are executing in the background, as is typical for sensing apps to operate in such mode. We denote with *Audio Optimised GPU (a-GPU)* the output from the optimisation engine as described in Section 5.3. Throughout the section we use the following baselines.

- *CPU sequential (CPU).* Our primary baseline implementations have a sequential workflow implemented in DSP-compatible C.
- *DSP sequential (DSP).* Our DSP baselines reuse the C compatible sequential CPU implementations but are run on the DSP via the Qualcomm Hexagon SDK.
- *CPU multi-threaded (CPU-m).* We implement variants of the pipeline algorithms where the bottleneck classification stage of the audio pipeline (occupying $> 90\%$ of the runtime in our examples) is parallelised across multiple threads. The GMM classification is restructured so that each GMM model probability is the unit of work for a separate thread. We maintain a pool of 4 threads which equals the number of physical cores of the Snapdragon Krait CPU – adding more threads did not lead to any performance benefits. The DNN classification is restructured so that the network propagation step is performed in parallel: each thread processes the activation of one node in a layer.
- *DSP multi-threaded (DSP-m).* Similarly to the multi-threaded CPU versions of the algorithms, we adopt DSP alternatives that parallelise the pipeline classification. The parallel DSP variants have logically the same threading model as the CPU one, with the difference that the DSP thread pool size is 3, i.e. it equals the number of hardware threads. We use the worker pool utility library *dspCV* which speeds up the execution for Computer Vision algorithms on the DSP.
- *Naive GPU (n-GPU).* We add the most naively parametrised GPU implementations as explicit baselines in addition to showing different parameter configurations. For all algorithmic building blocks (GMM, DNN, and features) in this naive baseline category we set the vectorised load factor $x = 1$, and the per-frame compute factor to be $\kappa = \nu$ which results in a frame fan-out factor of $\phi = n$.

Again, we build our analysis around the GMM-based and DNN-based computationally demanding audio pipelines as discussed in Section 5.4.

Energy Measurements. All energy measurements are taken with a Monsoon Power

Computation	Parameters	Speed-up Gain
GMM	$x = 1$, $\kappa = \nu$	3.6x
	$x = 16$, $\kappa = \nu$	12.8x
	$x = 16$, $\kappa = 1$	15.6x
	$\mathbf{x = 16}$, $\mathbf{\kappa = 4}$ **(tiling)**	**16.2x**
DNN	$x = 1$, $\kappa = \nu$	1.8x
	$x = 16$, $\kappa = \nu$	4.8x
	$x = 16$, $\kappa = 1$	13x
	$\mathbf{x = 16}$, $\mathbf{\kappa = 5}$ **(sliding window)**	**21.3x**
Filter banks	$x = 1$, $\kappa = \nu$	1.7x
	$x = 16$, $\kappa = \nu$	4x
	$x = 16$, $\kappa = 1$	6x
	$\mathbf{x = 16}$, $\mathbf{\kappa = 2}$ **(tiling)**	**6.6x**

Table 5.6.1: Parameter configuration speed-ups vs. CPU sequential implementation. Lines in bold show the results for the parameter values found by the optimisation engine.

Monitor attached to our MDP device. By default, the experiments were performed with a display off, with no services running in the background except system processes. This reflects the case when GPU offloading is done in the background, as in a continuous sensing scenario. The power evaluation setup closely matches the one reported by DSP.Ear in Chapter 3. Each application is profiled separately for energy consumption by averaging power over 10 runs on the CPU, DSP and GPU.

5.6.2 Pattern optimisation benchmarks

In this subsection we study how the application and parametrisation of the various optimisation techniques presented in Section 5.4 affect the GPU kernel runtime performance. Table 5.6.1 shows different points in the parameter space compared to the engine optimised configuration.

Pattern speedups. A first observation is that using vectorisation with larger batches ($x = 16$) provides a significant boost across all algorithms. The speedup of the most naive GMM kernels jumps from 3.6 to 12.8 times, the DNN ones – from 1.8 to 4.8 times, and the filter banks – from 1.7 to 4 times. The success of this simple technique can be attributed to the fact that the mobile GPU is optimised to efficiently access multiple items with a single instruction.

Another observation is that increasing the fan-out factor ϕ by setting $\kappa = 1$ provides tangible runtime boosts. As illustrated, speedups increase from 12.8 to 15.6 times for the GMM, and from 4.8 to 13 times for the DNN. Interestingly, the higher fan-out provides benefits even though the number of global memory accesses is raised by issuing writes of intermediate values to a scratch memory. The reason for this is that the fan-out pattern prominently increases the total number of work items and as a consequence there are more opportunities for the GPU to hide memory access latencies – whenever a group of threads is stalled on a memory read or write operation, with a higher probability the GPU can find another group that can perform computation while the former waits.

	GMM full pipeline	GMM classification	DNN full pipeline	DNN classification
DSP	-8.8x	-8.6x	-4.5x	-4.0x
DSP-m	-3.2x	2.5x	-2.1x	-1.5x
CPU (runtime)	1.0x (1573 ms)	1.0x (1472 ms)	1.0x (501 ms)	1.0x (490 ms)
CPU-m	3.0x	3.4x	2.8x	2.9x
n-GPU	3.1x	3.6x	1.8x	1.8x
a-GPU	**8.2x**	**16.2x**	**13.5x**	**21.3x**

Table 5.6.2: Speedup factors for one run of the various pipeline implementations compared against the sequential CPU baseline. Negative numbers for the DSP variants show that the runtime is that amount of time slower than the CPU baseline. CPU average runtime in ms is given for reference in brackets.

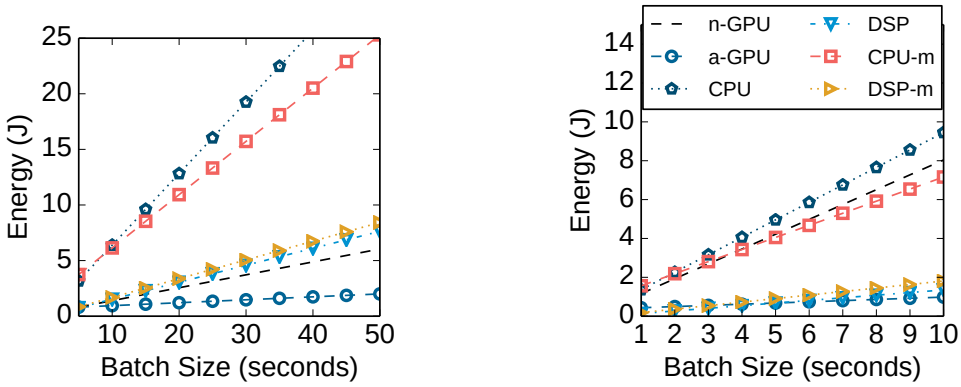
Last but not least, the more advanced tiling and sliding window techniques tuned by the optimisation engine provide noticeable speedup improvements over the straightforward use of shared memory. The sliding window optimisation boosts the second best DNN kernel speedup from 13 to 21.3 times, which is also the highest cumulative gain observed across all algorithms. Optimally parametrised tiling, on the other hand, brings the overall GMM speedup to 16.2 times and filter banks to 6.6 times. In these cases, increasing κ further results in suboptimal use of shared memory – since its size is limited, the work items can fetch only a proportion of the total data they need, the rest needs to be loaded from the slower global memory into thread registers.

Summary. The engine optimised kernels allow GPU computation to exhibit much higher performance than naively parametrised baselines. The optimisation techniques can be ubiquitously applied across multiple stages of the pipelines.

5.6.3 GPU pipeline runtime and energy

We compare our engine optimised GPU pipelines against the baselines listed in Section 5.6.1. Table 5.6.2 shows the runtime for running the pipelines on the various processing units. The most prominent observation is that the optimised GPU implementation is the fastest one. For instance, the full GMM and DNN pipelines are 8.2 and 13.5 times faster than a sequential CPU implementation respectively, an order of magnitude faster. In comparison, the CPU multi-threaded alternatives are around 3 times faster only. If the GPU is not carefully utilised, the naively parametrised GPU implementations may be up to 1.5 times slower than the multicore variants (e.g., DNN pipeline). The reason why the audio-optimised GPU fares so much better than both multicore CPU and naive GPU alternatives is because massive data parallelism is enabled by the parallel techniques – the hundreds of cores on the GPU can work on multiple small independent tasks simultaneously and hide memory access latency. This is especially true for the classification tasks that are 16.2 (GMM) and 21.3 times (DNN) faster than their sequential CPU counterparts.

In Figure 5.6.1 we plot the energy needed by the various units to execute the pipeline logic repeatedly on batched audio data. For one-off computations the cheapest alternative



(a) GMM pipeline

(b) DNN pipeline

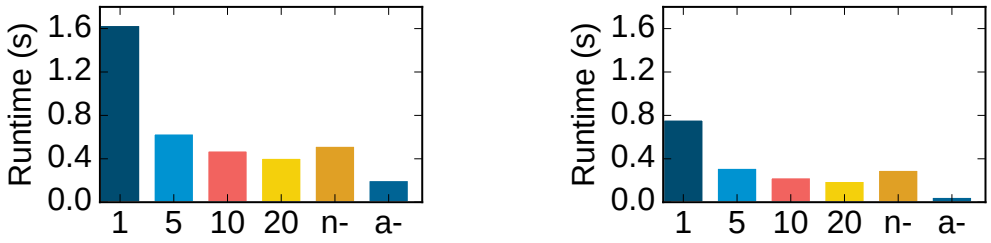
Figure 5.6.1: Energy (J) as a function of the audio processing batch size in seconds. Legend is shared, axis scales are different.

energy-wise is the DSP which can be up to 3.2 times more energy efficient than the optimised *a-GPU* for the DNN pipeline. Yet, the a-GPU is between 3 and 4 times more energy efficient than the sequential CPU for both applications. *If high performance is of utmost priority for an application, the GPU is the method of choice for on-device real-time feedback* – when optimised, GPU offloading will obtain inferences much faster and significantly reduce energy compared to the CPU.

A notable observation is that as the size of the buffered audio data increases, the a-GPU begins to outperform energy-wise the low-power DSP. For instance, with batch sizes of 10 and 6 seconds for the GMM and DNN pipeline respectively, the a-GPU provides 1.6 and 1.1 times lower energy and is more than 50 times faster. If applications can tolerate small delays in obtaining inferences, batched GPU computation will both deliver results faster than the DSP and save energy.

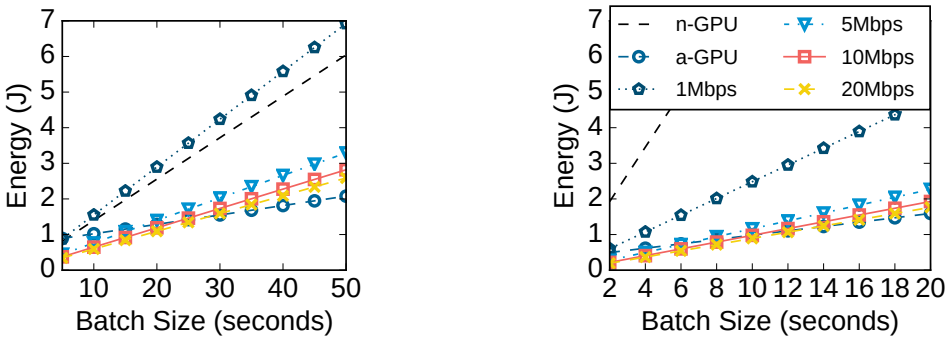
5.6.4 GPU sensing vs. cloud offloading

Cloud baselines. We compare the performance of the sensing algorithms on the GPU against the best performing cloud alternatives. For the DNN-style Keyword Spotting application, the cheapest alternative is to send the raw data directly for processing to the cloud because the application generates as many features as the size of the raw input data. For the GMM-style Speaker Identification pipeline, the cheapest alternative is to compute the features on the DSP and send them for classification to the cloud. However, this variant is > 10 times slower than computing the features on the CPU. Further, we find that when sending the features to the cloud, establishing the connection and transferring the data dominate the energy needed to compute features on the CPU. For this reason, the GMM cloud baseline in our experiments computes features on the CPU and sends them



(a) GMM (b) DNN

Figure 5.6.2: End-to-end latency for computing the audio pipelines on the GPU vs cloud. Numbers on the x axis show throughput in Mbps, and n- and a- refer to n-GPU and a-GPU.



(a) GMM (b) DNN

Figure 5.6.3: End-to-end energy as a function of the batch computation size for running the audio pipelines on the GPU vs cloud offloading. Legend is shared.

for classification to a remote server.

Latency results. In Figure 5.6.2 we plot the end-to-end time needed to offload one pipeline computation to the cloud and compare it against the total time required by the GPU to do the processing (including the GPU kernel set-up and memory transfers). We assume a window size of 64KB and vary the network RTT so that the throughput ranges from 1 to 20Mbps. Given this, the a-GPU implementation is 3.1 and 6.5 times faster than the good 5Mbps cloud alternative for the GMM and DNN pipelines respectively. This comes at the expense of a 1.6 and 1.4 times increase in the energy for a one-off computation for the two pipeline types respectively. The takeaway is that *if speed is favoured over energy, the GPU should be used for local processing because it will deliver inference results several times faster than cloud offloading even with good connectivity.*

Energy results. In Figure 5.6.3 we plot the total energy required to offload batched

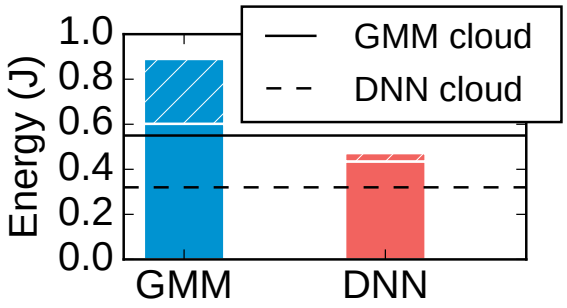


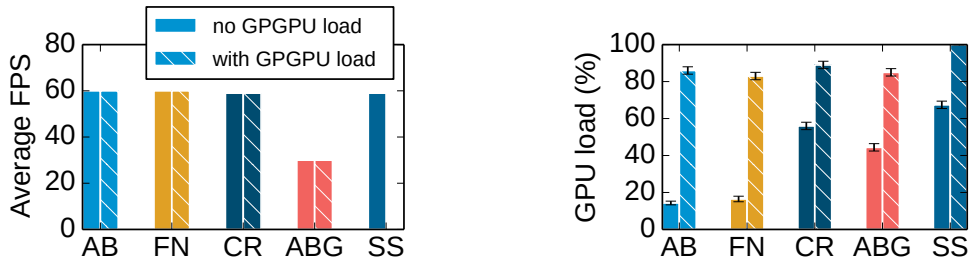
Figure 5.6.4: Energy consumed for one-off computation for the GMM- and DNN-based audio apps. Checkered bars show energy spent for processing, the rest is GPU tail energy. Lines show the total energy consumed by cloud offloading for the two apps for a network connection with an RTT of 104 ms.

pipeline computations to the cloud as a function of the batch size in seconds for which raw audio data is queued for processing. The most notable outcome is that *the a-GPU competes energy-wise with good connectivity cloud offloading in addition to being multiple times faster*. After a certain batching threshold, the total processing with the optimised a-GPU consumes even less energy. For instance, unless the network has a throughput of 20Mbps (and an RTT of 25 ms) the GMM-style pipeline starts getting cheaper than the faster connections after 20 seconds of buffered audio, and the DNN Keyword Spotting pipeline – after only 10 seconds of audio data. The reason for this phenomenon (batched processing is less expensive than cloud and one-off computation is not) is that the initial kernel set-up and memory transfer costs are high, but once paid, the a-GPU offers better energy per second for audio sensing.

In Figure 5.6.4 we plot a breakdown of the energy for a one-time run of the audio apps on the GPU to quantify the GPU setup overhead. Overall, the amount of energy spent in the GPU tail states is over 65% of the total consumption for both applications which confirms the prohibitively high setup/tear-down GPU costs. With a network that has an average RTT of 104ms (translating to $\approx$ 5Mbps throughput), the energy spent by cloud offloading is less than the GPU setup cost. Unless the audio app is highly sensitive to the runtime, cloud offloading may provide a desirable trade-off between energy and latency.

Another critical result is that optimising GPU execution is crucial – the naive n-GPU is more expensive ($>$ 2 times) energy-wise than almost all types of cloud offloading when batching. With the better but still unoptimised baselines with highest fan-out ($\kappa = 1$), the batching threshold for preferring GPU execution over cloud is higher (e.g., $\approx$ 14 seconds for the DNN algorithm), delaying the application response time further than what could be achieved with the engine optimised version.

Summary. Although cloud offloading has a significant computational lead over mobile,



(a) Frame rate (FPS)

(b) total GPU load

Figure 5.6.5: Average frame rate and GPU load when the games run without and with additional GPGPU load. Games in the experiment are Angry Birds (AB), Fruit Ninja (FN), Crossy Road (CR), Angry Birds GO (ABG), and Subway Surfers (SS). Legend is shared.

the GPU now provides advantages that makes local processing highly desirable – it is faster, less susceptible to privacy leaks as execution is entirely local, works regardless of connection speed, and even competes with cloud in terms of energy.

5.6.5 GPGPU and graphics workloads

Here we investigate how GPGPU computations interfere with other GPU workloads such as those used for graphics processing – *will the background GPU computation affect negatively the user experience?* We schedule the execution of the audio sensing pipelines (either Speaker Identification or Keyword Spotting) to run continuously for a minute on the GPU while the mobile user is interacting with other applications that are known to strain the GPU resources, i.e. games. We pick 5 hugely popular Android games with multi-million downloads and different play styles (Angry Birds, Fruit Ninja, Crossy Road, Angry Birds GO, and Subway Surfers), and observe the effect GPGPU computation has on the perceived gameplay quality. To quantify this we measure the average frame rate (frames per second (FPS)) with GameBench [17] during gameplay over 5 1-minute long runs.

In Figure 5.6.5 we show the aggregate results from the experiments. *For all games except Subway Surfers the GPGPU computation does not change the original frame rates of the games, although the total GPU load substantially increases.* For instance, the render-heavy racing Angry Birds GO maintains an average frame rate of 30 FPS both with and without the added audio sensing workload, even though the total GPU load jumps from 44% to 85%. For this and the other three games with similar behaviour (Angry Birds, Fruit Ninja, Crossy Road), the effect can be explained by the facts that 1) the original load the games place on the GPU is not too high, 2) GPU rendering is time-shared with GPGPU computation, and 3) the audio sensing kernels are short-duration (a single kernel execution never exceeds tens of milliseconds). With the endless runner Subway Surfers,

however, the original average GPU load is already very high ($\approx$ 70%), and adding the GPGPU computation results in a screen freeze so that the game becomes unresponsive. This can be attributed to the fact that the OS does not treat the GPU as a shared resource and there is a lack of isolation of the various GPU workloads. One way to approach this is introduce OS-level abstractions that provide performance guarantees [172].

5.7 Discussion and limitations

Here we survey key issues related to the applicability of the GPU parallel optimisations for audio sensing.

5.7.1 Implications

Privacy. Our findings suggest that algorithms optimised for embedded-class GPUs can bring the much coveted privacy guarantees to devices such as Amazon Echo [4] and Google Home [20], if the operation remains entirely on the device itself. These assistants respond to simple home user requests (such as, "turn on the light"), but are known to heavily exploit cloud offloading. With the help of our techniques doing the processing locally on the GPU can be done faster than cloud offloading, and without exposing sensitive information to untrusted third parties.

Servicing multi-app workloads. GPUs will play a crucial role in offloading the sensing workloads of digital assistants as they cannot be serviced by the DSP capabilities alone. Amazon Echo, for instance, performs multiple audio sensing tasks on a continuously processed audio stream, including: 1) detect the presence of speech vs other sounds; 2) perform spoken keyword spotting (as all commands are triggered by the same starting word); and, 3) speech recognition, along with additional dialogue system analysis that allows it to understand and react to the command. These tasks collectively are well beyond the DSP processing and memory capabilities as demonstrated in Chapter 3 with the design of DSP.Ear; in such multi-app audio sensing scenarios approaching the mobile GPU with routines that maximise runtime performance and minimise energy consumption is critical.

Energy reductions. Audio sensing algorithms are notorious for their continuous monitoring of the sensor stream. Whereas DSP offloading is massively adopted as the go-to power reduction approach for apps such as hot keyword spotting, with the increase in number of concurrent audio sensing services mobile users adopt (e.g., Google Now, Auto Shazam), the DSP will have to selectively process a subset of the algorithm stages. In multi-app scenarios, optimally using the GPU as we have done in this work will be instrumental in keeping the power-hungry CPU or privacy-invading cloud offloading at bay.

5.7.2 Discussion

Performance on other mobile GPU varieties. Although it is highly likely there will be a difference in the exact values for the performance boosts on other GPU models (such as NVIDIA's Tegra), we expect qualitatively similar results when deploying the pipelines there. For example, speedups from the GPU data parallelism will be sufficiently high to deliver real-time performance for applications that can afford the energy costs. This is because the optimisations we have performed can be generalised to any OpenCL-compliant GPU architecture, and do not rely on vendor-specific features.

Parallelising other audio sensing algorithms. The core mechanics behind the optimisation patterns can be applied to other classifiers such as Support Vector Machines (SVM), and deep learning network topologies such as Convolutional Neural Networks (CNN). This is because the patterns depend on how the classification is applied to the audio data stream (in sliding windows, combining model parameters with frame data independently to different frame offsets), rather than fully depend on the concrete algorithm implementation.

GPU vs. multicore CPUs. As single-thread performance for microprocessor technology is levelling off, multiple cores will become major drivers for increased performance [62] (e.g., up to 61 for Intel Xeon Phi [26]). Developers will likely be faced with similar data parallel challenges – increasing the total number of concurrent tasks for better utilisation, and efficiently leveraging memory caches to mask access latency. As OpenCL manages heterogeneous parallel algorithms transparently from the underlying multicore architecture, the developed OpenCL-compliant optimisation techniques will prove valuable to multicore CPUs as well.

GPU programmability. GPUs are notoriously difficult to program – even if the algorithm exhibits data parallelism, restructuring it to benefit from GPU computation often requires in-depth knowledge about the algorithm mechanics. In fact, automated conversion of sequential to parallel code has been an active area of research [66, 65], but fully automating the parallelisation process still remains a big challenge. We provide a portable OpenCL library of parallel implementations of popular algorithms used in audio sensing (e.g., GMMs, DNNs) and expect developers will either compose their own pipelines by reusing the OpenCL host and kernel code, or by applying the insights from the parallel optimisation patterns to their own implementations.

5.8 Related work

Sensor processing acceleration and efficiency. A large body of research has been devoted to the use of heterogeneous computation via low-power co-processors [160, 140, 175] and custom-built peripheral devices [183] to accelerate or sustain power efficient processing for extended periods of time. Little Rock [160] and SpeakerSense [140] are among the first

to propose the offloading of sensor sampling and early stages of audio sensing pipelines to low-power co-processors – the processing enabled by such early units is extremely energy efficient but limited by their compute capabilities to relatively simple tasks such as feature extraction. Shen et al. [175] study more complex inference algorithms for continuous operation on DSPs but with DSP.Ear from Chapter 3 we demonstrate such units can be easily overwhelmed and often the energy efficiency comes at the price of increased inference latency.

General-purpose GPU computing. GPUs have been used as general-purpose accelerators for a range of tasks, the most popular applications being computer vision [185, 72, 112] and image processing [177, 158]. Object removal [185] and face recognition [72] on mobile GPUs have been showcased to offer massive speedups via a set of carefully selected optimisation techniques. Although the techniques found in the graphics community as well as in the field of speech processing (fast spoken query detection [197]) address similar data parallel challenges to what we identify (increasing thread throughput, careful memory management), these techniques remain specific to the presented use cases.

The GPU implementation of automatic speech recognition based on GMMs [97], for example, proposes optimisations that are tightly related to the specific speech recognition pipeline organisation. Instead, we target general techniques that are applied at the level of the machine learning model, or the level of organisation related to multiple algorithms for processing audio data (e.g., computation with overlapping frames). As such, the insights drawn from our work are directly applicable to a full class of algorithms that build upon commonly adopted machine learning models in audio sensing.

Packet routing [101] and SSL encryption [113] leverage GPUs to increase processing throughput via batching of computations, but none of these works is focused on studying energy efficiency aspects which are critical for battery-powered devices.

GPU resource management. PTask [172] is an OS-level abstraction that attempts to introduce system-level guarantees such as fairness and performance isolation, since GPUs are not treated as a shared system resource and concurrent workloads interfere with each other. The relative difficulty in manually expressing algorithms in a data parallel manner may lead to missed optimisation opportunities – works such as those of Zhang et al. [195] and Jog et al. [115] attempt to streamline the optimisation process. The former improves GPU memory utilisation and control flow by automatically removing data access irregularities, whereas the latter addresses problems with memory access latency at the thread scheduling level. Both types of optimisations are complementary to our work – we optimise the general structure of the parallel audio processing algorithms, while the mentioned frameworks tune parallel behavior of already built implementations. Last but not least, Sponge [109] provides a compiler framework that builds portable CUDA programs from a high-level streaming language. Instead, we study the trade-offs mobile GPUs provide for sensing, and build on top of OpenCL which together with the Qualcomm Adreno GPU dominate the mobile market.

5.9 Conclusions

In this chapter we studied the trade-offs of adopting audio sensing routines specialised to run on a mobile GPU. We devised an optimisation engine that leverages a set of structural and memory access parallel patterns to auto-tune GPU audio pipelines. As a result, our optimised GPU routines are an order of magnitude faster than sequential CPU implementations, and up to 6.5 times faster than cloud offloading with a 5Mbps throughput. We discovered that with only 10-20 seconds of batched audio data, the optimised GPU begins to consume less energy than both cloud offloading with good throughput and a low-power DSP.

The insights drawn in this and the previous chapter can help towards the growth of the next-generation mobile audio apps that adopt advanced DSP and GPU capabilities for extreme runtime and energy performance. Despite these considerable gains in accuracy, runtime and energy, the optimisations were primarily applied at the algorithm level for each app individually. What is missing from our studies so far is the exploration of scheduling approaches that automate the process of resolving resource contention among multiple apps when they share heterogeneous processor time and memory. This is the concern of the next chapter.

Chapter 6

Scheduling sensor processing

6.1 Introduction

The previous two chapters explored the potential of customising mobile sensing algorithms for two heterogeneous processor types, a low-power co-processor and a GPU. Chapter 3 evaluated how sensing systems could efficiently exploit the processor hierarchy to statically partition sensor execution across resources and reduce their energy consumption to acceptable operational levels. These optimisations, however, are inherently *static* in nature: they target either the implementation of specific sensing algorithms, or a pre-defined distribution of sensor processing stages across resources. In this chapter we present a scheduling framework that *dynamically* places sensor tasks for execution on various processors or in cloud in response to fluctuations in resource availability and sensor processing load.

We re-examine some key ideas scattered in existing computational offloading approaches to answer the question: *can we maximise resource utilisation from multiple concurrent sensor apps by better placement of the underlying algorithms across resources and without compromising app accuracy and responsiveness*? A variety of capable scheduling/offloading approaches have been proposed [154, 80, 150, 116, 176] but they either have different optimisation goals or have not fully addressed the above question in the emerging context of concurrent sensor apps with diverse deadlines running on recent off-the-shelf heterogeneous mobile SoCs.

We introduce LEO – a purpose-built sensing algorithm scheduler that targets specifically the workloads produced by sensor apps. LEO builds upon the solid body of related work to demonstrate that further increases in the efficiency of sensor processing *without reducing the app accuracy* can be achieved by bringing together four key ideas scattered across existing systems: 1) full usage of heterogeneous hardware, 2) joint sensor app optimisation, 3) frequent schedule re-evaluation, and 4) exposing algorithm semantics to the optimisation model. As a result, LEO is able to optimally offload individual stages of sensor processing

across the complete range of heterogeneous computational units (viz. the co-processor, CPU, cloud, and provisionally a GPU). A key innovation in LEO is that all offloading decisions can be performed on the smartphone co-processor, this enables scheduling to be low energy and frequently performed. Consequently, LEO adjusts how sensor algorithms are partitioned across each computational unit depending on fluctuating resources (e.g., network conditions, CPU load) in addition to which sensor apps are executing concurrently. With LEO, the energy and responsiveness trade-offs of sensing algorithms of all apps are *jointly* optimised, leading to more efficient use of scarce mobile resources.

Chapter outline. In Section 6.2 we give a bird's eye view of the scheduling engine execution, whereas Section 6.3 details the major design components that collectively define the system. Section 6.4 describes the implementation of the scheduler as well as the interfaces necessary to access our extensive library of algorithms for feature extraction and classification needed for common forms of context inference. We provide a systematic evaluation of LEO's overhead and schedule optimality, and a thorough analysis of the system energy savings under a variety of network conditions and workloads in Section 6.5. We find that compared to a principled general-purpose offloader that leverages the DSP in addition to cloud, LEO requires about 7 to 10 times less energy to build a schedule, and still reduces the energy consumption by up to 19% for medium to heavy sensor workloads. Section 6.6 discusses key issues related to the scheduler design, and Section 6.7 compares our system with existing sensor optimisation techniques and orchestrators. Finally, Section 6.8 concludes the chapter with reflections on our contributions.

6.2 Scheduling framework overview

Towards addressing the shortcomings of general computational offloading for sensor app workloads, LEO is a sensor algorithm scheduling engine that maximises the energy efficient execution of a variety of sensor apps. LEO is used by developers via a set of Java/C APIs with a unified interface (see Section 6.4 for details). Through the Java API developers can specify the sequence of sensing algorithms (e.g., feature extraction, classifier model) required by their app to collect and process data. In turn, LEO leverages the internal structure of the sensing algorithms predefined in our library of algorithmic components to partition the execution of each algorithm across all available computational units (even conventionally closed DSPs found in smartphone SOCs, along with the CPU, cloud, and a GPU). Because of the rich portable library of sensing algorithms (ranging from DNNs to domain-specific features for emotion recognition) LEO is able to support a wide range of sensor processing (and thus apps).

LEO re-examines several concepts related to scheduling/offloading from previous systems and re-evaluates them in the context of concurrent sensor apps running on off-the-shelf heterogeneous mobile SoCs. LEO shows that we do not need to compromise the utility/accuracy of apps or sacrifice their responsiveness in order to gain substantial energy savings.

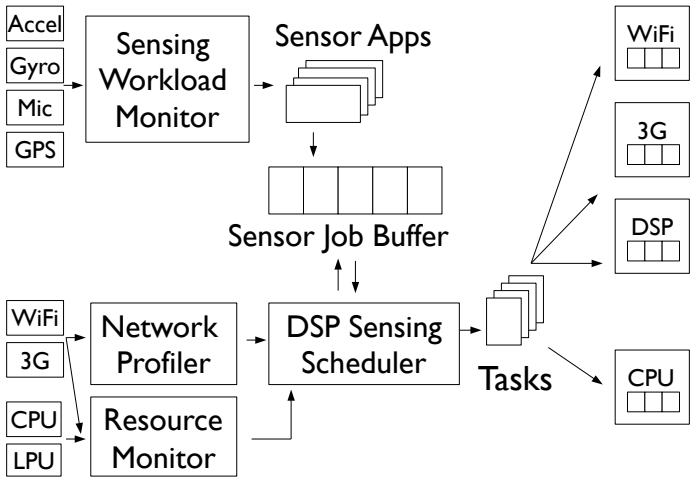


Figure 6.2.1: LEO architectural components.

Instead, *maximising resource utilisation can be performed by a smarter distribution of concurrent sensor algorithms with well known semantics across multiple resources.* To achieve this, LEO:

1) considers offloading decisions collectively for heterogeneous mobile processors and cloud. LEO solves a Mixed Integer Linear Programming (MILP) global resource optimisation problem that directly targets energy efficiency as its objective.

2) jointly optimises resource use for multiple apps simultaneously. This promotes cooperation in using network bandwidth or scarce co-processor memory/computation. As a result, maximising resource use is done across the full sensing ecosystem rather than leaving individual apps to do guesswork on when resources are busy.

3) exposes the internal structure of the pipeline stages to the offloading engine for fine-grained energy control. LEO provides a rich set of reusable algorithmic components (feature extraction, classification) which are the building blocks of common sensor pipelines. By leveraging sensor processing specific knowledge LEO decomposes pipelines into more granular units, orchestrates and interleaves their execution even under tight latency constraints by working on multiple smaller tasks in parallel.

4) frequently re-evaluates the schedule to remain responsive to sensor processing requests. Sensor apps generate mixed workloads with near real-time and delayed execution requirements. To provide timeliness guarantees while coping with changes in network conditions and bursts of generated sensor events such as detected speech, noise, or motion, LEO computes fast, reactive schedules that are frequently revised. A key enabler for this is the ability of LEO to run as a service on one of the hardware threads of the low power DSP where the scheduler solves heuristically the global optimisation problem mentioned above.

6.2.1 Architectural overview

In Figure 6.2.1 we show a birds-eye view of the system architecture and its operational workflow. The system supports a mixture of near real-time and delay-tolerant sensor processing apps. The pipeline stages of these apps are typically triggered by sensing events such as the detected presence of speech from the sensor data streams, or by logic embedded in the sensor app. Example apps with their trigger contexts are recognising emotions from voice, or counting the number of steps when the user is walking. Over time and as sensor events are encountered, the apps generate *job* definitions which are buffered requests for obtaining an inference (e.g., detected emotion) from the sensor data. Periodically, the sensor offloading scheduler, known as *DSP Sensing Scheduler*, inspects the sensor job buffer for the generated workload of sensor processing requests, and makes scheduling decisions that answer the questions: 1) How should the pipelines be partitioned into sensor tasks? and 2) How should these tasks be offloaded if needed?

Sensing Workload Monitor. A set of binary filters (e.g., "silence vs. noise", "speech vs. ambient sounds", "stationary vs. moving") comprise the *Sensing Workload Monitor* which continuously inspects the sampled sensor data on the Low Power Unit (DSP in our case) for the presence of relevant events (for triggered sensing apps). Once such events are detected or in response to the sensor app, job requests are placed in a global queue that buffers them together with the raw sensor data.

DSP Sensing Scheduler. This component represents the core scheduling algorithm that decides how the pipeline execution should be partitioned into sensor tasks and where these tasks should be executed (DSP, CPU, cloud or potentially GPU). The scheduler inspects the sensor job buffer once every t seconds for processing requests where t is a configurable system parameter currently set to a short window of 1 second. Queued tasks are periodically scheduled and executed before the next period expires. Critically, LEO defines a mathematical optimisation problem that can be solved frequently and maintains a short rescheduling interval in order to systematically re-evaluate fleeting optimisation opportunities and remain responsive to apps such as voice activation services with near real-time requirements. The high levels of responsiveness and frequent on-demand optimisations are largely enabled by two key design choices. First, the scheduler reorganises the structure of sensor pipelines to create more modular processing tasks via three key techniques detailed in Section 6.3.1: *Pipeline Partitioning*, *Pipeline Modularisation*, and *Feature Sharing*. Second, the scheduler employs a fast heuristic optimisation solver (based on metaheuristics) that is executed with an ultra low overhead on the DSP to find a near optimal assignment of tasks to resources.

Resource Monitor. A *Resource Monitor* provides feedback to the DSP Sensing Scheduler with regard to changing CPU load or network conditions such as connecting to or disconnecting from a WiFi network.

Network Profiler. Similarly to MAUI [80], a *Network Profiler* sends a fixed 10KB of data

and measures the end-to-end time needed for the upload. Fresh estimates are obtained every time the scheduling engine ships data for processing to a remote server. To keep measurements fresh, profiler data is sent every 15 mins in case no offloading has been done.

Offline Profiling. Last, an offline app profiler obtains estimates of the energy consumption and latency for each of the app pipeline stages (feature extraction and classification) measured on the CPU, DSP, and for some algorithms on the GPU. The measurements serve as an input to the DSP Sensing Scheduler that distributes sensor pipeline tasks across offloading resources. The profiling session is a one-time operation performed offline since the mobile OSs have limited APIs for performing fine grained energy measurements [80] and only report the percentage of the battery left, largely insufficient to cover the profiling needs.

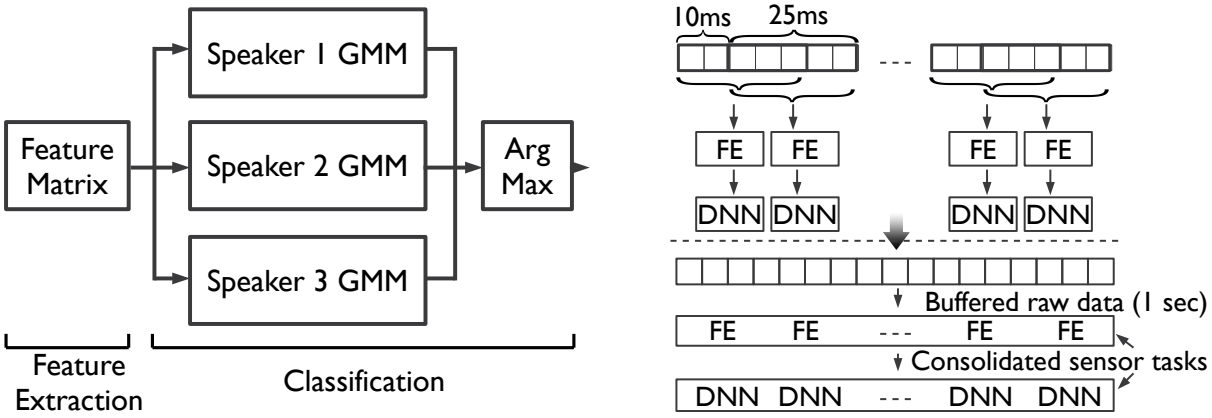
6.3 Framework design components

LEO is designed to manage the offloading needs of sensor apps with both near real-time and delayed deadline requirements. In this section we detail how LEO leverages algorithm semantics to optimise resource use, and also formally define the optimisation problem that jointly decides for concurrent apps how to execute their algorithms across resources.

6.3.1 Restructuring sensor app execution

Pipeline Partitioning. Sensing pipelines are decomposed into logical chunks of computations to increase the granularity of the sensor tasks and enable their more comprehensive exposure to the offloading components (cloud, DSP, and GPU). This can potentially lead to more efficient local-remote splits and parallelise execution across multiple resources to meet the tighter deadlines of near real-time apps.

Our framework exploits the typical structure of audio sensing apps presented in Section 2.2 of Chapter 2 to define the types of sensor tasks distributed across resources. Pipeline tasks are divided into two primary types: 1) *feature extraction*, typically represented by a single task per app; and 2) *classification*, which may be further decomposed into multiple identical tasks the output of which is combined by a simple aggregation operation (e.g., *max*, *sum*). For instance, the inference stage of many of the apps detailed in Chapter 2 is usually organised around multiple class models (GMMs). The Speaker Identification example is revisited here in Figure 6.3.1a – we have one model for each speaker. During the classification process, the extracted audio features that summarise the acoustic observations are matched against each model in turn. The speaker, the model of which with highest probability corresponds to the features, is the end output of the pipeline. The aggregation operation for this and other GMM-based pipelines is thus “max”.



(a) Pipeline Partitioning (b) Pipeline Modularisation

Figure 6.3.1: Sensor pipeline restructuring techniques.

The ways in which signal processing algorithms can be partitioned is predefined in our library. At runtime LEO decides whether and how to leverage the partition points by solving a global resource allocation problem and managing cross-resource communication with custom implemented message passing. As a result, apps that use the algorithms we offer automatically benefit from efficient task distribution without them knowing the details of the pipeline execution plan. Further, the decomposition into shorter duration tasks enables the pipeline stages of concurrent apps to be interleaved – the result being higher utilisation of the energy efficient but memory-constrained DSP.

Pipeline Modularisation. Partitioning the pipelines into their natural processing stages increases the granularity at which sensor apps operate. However, depending on the app subject to partitioning, the technique may sometimes produce a large number of sensor tasks that unnecessarily pollute the resource optimisation space with a forbiddingly high number of offloading configurations. *The goal of the Pipeline Modularisation is to consolidate multiple sensor tasks generated at a high frequency into a single modular computational unit processed at a reduced frequency.*

The DNN-based Keyword Spotting app introduced in Chapter 2 and revisited in Figure 6.3.1b, for example, generates 100 classification and feature extraction tasks per second. This is because it extracts features from 25ms frames, and performs classification (neural network propagation and smoothing) on a sliding window of 40 frames every 10ms. This high frequency of computations enables the app to maintain almost instantaneous responsiveness to detected hot phrases. However, at a small latency price we can reduce the amount of tasks a hundredfold if we group all feature extractions or classifications in a second into a modular unit by buffering the sensor data and performing them together on the accumulated speech samples. Thus, processing is performed at a reduced frequency once every second, greatly simplifying the search for the optimal task allocation, while at the same time still maintaining near real-time responsiveness of the app to up to 2 seconds

after the detection of a hot phrase.

Feature Sharing. The pipeline decomposition allows us to register modular identifiable tasks into the queue of sensor processing requests. Each feature extraction task is identified by a reference to a position in the sensor stream and the type of features to be extracted from the raw data. This allows LEO to detect overlapping computations and eliminate redundancies when multiple apps require the same set of features for their stages deeper into the pipeline. One example of shared features are the PLP coefficients needed by the Speaker Identification and Emotion Recognition apps.

6.3.2 LEO optimisation solver

The LEO solver uses the restructured pipeline components as well as data collected by the Sensor Workload Monitor and Network Profiler as an input to a global joint optimisation problem that determines which and where sensor app tasks should be offloaded. LEO solver's goal is to find a multi-app partitioning strategy that minimises the mobile device energy consumption subject to latency constraints.

The solver takes advantage of the pipeline reorganisation techniques introduced in the previous subsection to generate modular sensor task definitions with *loose data dependencies*: feature extraction output serves as input to classification and tasks from the same pipeline stage can be computed in parallel across units. Formally, the DSP Sensing Scheduler solves a mixed integer linear programming problem (MILP) with relaxed data dependencies constraints. The objective is to minimise the total energy consumption for processing all sensor tasks generated in a time window τ by all actively running sensor apps:

$$\text{Min} \quad \sum_{i,q,u} x_{iqu} e_{iqu} + \sum_{i} F_{mem}(i, x_{iq\xi}) w_{uplink} p_{\xi} \tag{6.1}$$

where

- x_{iqu} denotes the total number of computations from app i's pipeline stage q that will be processed on computational unit $u \in \{\text{CPU, DSP, GPU }\}$ (or networked resource ξ when $u = \xi \in \{\text{3G, WiFi, Bluetooth}\}$).
- e_{iqu} indicates the energy consumed by performing these computations on the resource u ($e_{iq\xi} = 0$).
- $F_{mem}()$ is a linear function mapping the number of remotely executed sensor tasks to the size of the application data needed for network transfer.
- w_{uplink} – most recently estimated uplink speed (Kbps).
- p_{ξ} is the estimated average power in mW for performing the network transfer.

The objective expresses the estimated total energy expenditure from performing computations on the assigned offloading resources plus the energy needed to transfer data for remote execution. The offloading schedule is subject to the following constraints:

- The total execution time for processing tasks locally (Equation 6.2) and remotely (Equation 6.3) must not exceed the time window τ:

$$\text{s.t.} \quad \forall u \sum_{i,q} x_{iqu} t_{iqu} \leq \tau k_u \tag{6.2}$$

$$\text{s.t.} \sum_{i} F_{mem}(i, x_{iq\xi}) w_{uplink} + \sum_{i,q} x_{iq\xi} t_{iq\xi} \leq \tau \tag{6.3}$$

  Here t_{iqu} is the time in seconds required by computation of type q to be performed on computational unit u, and k_u is the number of concurrent threads on resource u.

- The total number of sensor tasks offloaded across resources must be equal to the number of tasks $n_{iq}(\tau)$ generated by the buffered processing requests in time window τ.

$$\text{s.t.} \quad \sum_{u} x_{iqu} = n_{iq}(\tau) \tag{6.4}$$

We note that the typical restructured pipeline computations from our representative examples can be easily executed with subsecond latencies. As demonstrated in the evaluation of DSP.Ear in Chapter 3, the feature extraction stages for many apps are already executed within a second for both the CPU and DSP. The low latencies enable us to shrink the offloading window τ to 1 second. This also helps with fast reactive dispatching of computations that require tight timeliness guarantees (e.g., Keyword Spotting). In our implementation, although the tasks from apps that do not need near real-time requirements are scheduled in a batch with other tasks under tight 1-second constraints, their actual execution is postponed and rescheduled at a later stage if executing them before the next rescheduling interval expires means that the power-hungry CPU will be used.

6.3.3 Running the solver on the DSP

The optimisation problem defined in the previous section would typically be solved with standard optimisation tools such as GLPK [18] or *lp_solve* [30]. However, we observe that the underlying algorithm that systematically explores the scheduling configurations to find the optimal solution is too heavy to be performed frequently. In fact, when we set the time window for buffering processing requests to 30 seconds, and increase the number of scheduled apps to 9, the algorithm takes seconds to minutes to complete even on the quad-core Snapdragon CPU. For our aims the general-purpose solver scales poorly with the increase in number of apps and processing requests. Instead, we adopt a heuristic algorithm that can be run efficiently on the DSP to constantly re-evaluate a near optimal

Algorithm 2 DSP Sensing Scheduler Heuristic Algorithm

Require: Number of generations n, mutation probability α

```
1:  function HeuristicSearch(n, α)
2:      P ← InitialSchedulePopulation()
3:      x ← SelectBestSchedule(P)
4:      for i ← 1 to n do
5:          O ← GenerateOffspringSchedules(P)
6:          for c ∈ O do
7:              c ← Mutate(c, α)
8:              if i%2 = 0 then
9:                  c ← LocalImprovement(c)
10:         P ← SelectNextGenerationSchedules(P ∪ O)
11:         x ← SelectBestSchedule({x} ∪ P)
12:     return x
```

offloading schedule. We sacrifice the absolute optimality for substantial reductions in the scheduling overhead both in terms of energy and latency.

Heuristic Algorithm. The concrete framework we use is based on *memetic algorithms* (MAs) [152, 79] which are population-based metaheuristics that are highly scalable, are able to incorporate domain-specific knowledge, and are less prone to local optima. However, the scheduling algorithm is not restricted to a concrete choice as long as it conforms to several criteria: it is fast (preferably with polynomial complexity to the number of sensor apps and resources), it finds solutions that are close to optimal, and it is deployable on the DSP. We experimentally find that the memetic algorithm is one that satisfies all these requirements.

The algorithm takes as input sensor pipeline tasks, available offloading resources, and the constraints listed in Section 6.3.2 that define the feasible solutions. Algorithm 2 outlines the pseudo code of our heuristic. The basic structure of the algorithm is an iterative procedure that maintains a *population* of candidate schedules the quality of which improves over time and is measured through a *utility* function (our objective defined in Equation 6.1). Each offloading configuration is represented as shown in Figure 6.3.2 and each cell in the table corresponds to the value of the decision variable x_{iqu} defined in our problem statement 6.3.2.

The memetic algorithm defines a series of schedule transformation (mutation and local search) and recombination (crossover) operations that are applied systematically to the population of schedules updated each iteration. The iteration consists of creating candidate offspring schedules (line 5) from selected parents and subsequently choosing the surviving set of schedules (line 10) that will constitute the next generation. Parent schedules are selected for reproduction with a probability proportional to their utility. Reproduction is performed through a standard two-point crossover operation illustrated in Figure 6.3.3. Once offspring schedules are generated, two key transformations are applied to each child

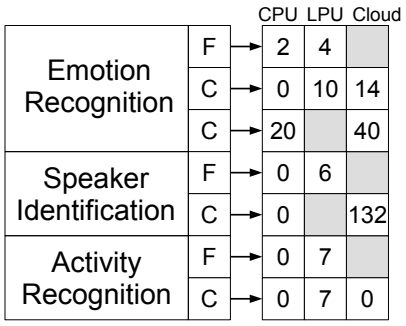


Figure 6.3.2: Schedule representation.

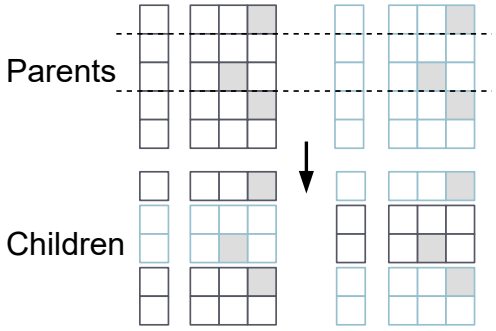


Figure 6.3.3: Two-point crossover.

Application	Sensor	Main Features	Inference Model	Frame	Window
Activity Rec. [143]	Accel	Freq. and Time Domain	J48 DT	4s	4s
Step Counting [63]	Accel	Time Domain	WPT	4s	4s
Speaker Count [94]	Mic	MFCC [88], pitch [81]	Clustering	32ms	3s
Emotion Rec. [166]	Mic	PLP [105]	14 GMMs [61]	30ms	5s
Speaker Id. [166]	Mic	PLP	22 GMMs	30ms	5s
Stress Detection [141]	Mic	MFCC, TEO-CB [199]	2 GMMs	32ms	1.28s
Keyword Spotting [70]	Mic	Filterbank Energies	DNN [106]	25ms	1s

Table 6.3.1: Implementation overview of the sensing apps used in our scheduling workloads. The window shows the amount of time features are accumulated from frames before an the classification/inference stage is triggered. Frame lengths shown are the default used in the original works. The used sensor sampling rates are 50Hz for the accelerometer and 8kHz for the microphone. WPT (Windowed Peak Thresholding), DT (Decision Tree), GMMs (Gaussian Mixture Models), DNN (Deep Neural Network).

(lines $6-9$): mutation to promote diversity, and local search to increase utility. The local search step improves the utility of a newly produced offspring schedule by searching for better neighbour solutions and replacing the original schedule with the neighbour the fitness of which is highest. Finally, the best s schedules survive through the next iteration, where s equals the population size. To reduce the runtime of the algorithm we limit the number of generations n to 100 and perform the local improvement step every other generation. We resort to standard textbook parameters of the algorithm [79]: population size of 10, 5 parents, 20 child schedules, and a mutation probability set to 0.1.

6.4 Framework implementation

Here we discuss the implementation details of the system prototype and example sensing algorithms used in the evaluation. As in Chapters 3, 4, and 5, the development is performed on a Snapdragon 800 Mobile Development Platform for Smartphones (MDP/S) [46]. We exploit the following resources on this platform: Hexagon DSP, Adreno GPU, Krait CPU and wireless connectivity.

Application model. Similarly to ORBIT [150] we adopt a component-based programming model where we provide an extensive library of reusable signal processing algorithms which developers can use to build and integrate their sensing pipelines into application code. We refactor our sensing algorithm implementations from the systems presented in Chapters 3, 4, and 5 to organise them into a library of routines. After the reimplementation we have 7 domain-specific categories of feature extraction algorithms and 5 popular machine learning models covering the narrow waist of computations for multiple apps from the mobile sensing literature. With this library we were able to prototype and incorporate 2 accelerometer and 5 microphone sensing apps listed in Table 6.3.1.

The audio pipelines used in our workload analysis cover the range of examples given in Chapter 2: various sound recognition apps, together with keyword spotting and unsupervised speaker counting. Although for many of the apps we were able to design much more accurate and efficient classifiers based on DNNs as demonstrated in Chapter 4, we stick to the original pipeline design. This promotes diversity in the types of sensing algorithms used and enables us to evaluate how the scheduling framework can cope with heavy workloads that feature both compute-intensive GMMs and DNNs.

One of the major advantages of resorting to a library-based approach is that we have full control over how the various signal processing tasks can be decomposed, i.e. we can expose the sensing algorithms to our pipeline restructuring techniques (Section 6.3.1) without involving the developer in the process. Instead, LEO fully automates the partitioning decisions at runtime.

The algorithms are subject to our pipeline reorganisation techniques so that all app pipelines are partitioned as discussed in Section 6.3.1, and the Keyword Spotting app is restructured by the Pipeline Modularisation technique. Feature Sharing is enabled for the Emotion Recognition and Speaker Identification apps. We give a recap of the most prominent elements from the algorithm implementations in Table 6.3.1.

APIs and accessibility. To enable app components to run on heterogeneous computational processors and cloud, all accelerometer and audio processing algorithms are implemented in C with a unified interface following the guidelines of the Hexagon SDK. In Figure 6.4.1 we provide a subset of the C signature conventions the various audio processing methods must comply to in order to take advantage of LEO's partitioning capabilities. We maintain the same copies of the C code on the DSP, CPU and on the server. To facilitate the integration of the signal processing components with Java application code we have built a Java Native Interface (JNI) bridge library that interfaces between the CPU and DSP. Further, we have defined a high-level Java API with some notions borrowed from Google Cloud Dataflow programming model [19] (applying a sequence of transforms to the input stream) that can help developers specify custom pipelines built from the reusable components we provide. Sample code defining a speaker identification pipeline with 2 speaker models and a voice-activation trigger is given in Figure 6.4.1. The high level components ultimately map to the C functions with conventionalised signatures – e.g., the WindowTransform class accepts as an argument to its constructor the name of the C

```
/* Java pipeline specification:
sequence of transforms
applied to the sensor stream */
Pipeline p = new AudioPipeline();
p.trigger(new FrameTrigger("Speech_trigger"));
p.apply(new WindowTransform("PLP_windowFeatures")
    .frameSize(240).frameRate(80).window(500))
 .apply(new ParallelTransform(new Transform[] {
        new GMMTransform("speaker1.gmm"),
        new GMMTransform("speaker2.gmm")
        }, AggregatorType.Argmax));

/* DSP compatible C signature conventions */
// initializes a struct from the memory
// referenced by ptr
void* X_init(int8_t* ptr, void* params);
// amount of memory in bytes (size of state struct)
uint32_t X_getSize(void* params);
// trigger function (e.g., speech detection)
int X_trigger(int16_t* buffer, int size);
// feature extraction
void X_windowFeatures(void* me, int16_t* buffer,
        int size, float** outData,
        int nRows, int nCols);
// inference/classification
float X_windowInference(void* me, float** inData,
    int nRows, int nCols);
```

Figure 6.4.1: Example code snippets showcasing the APIs supported by LEO. The Java API can be used to define the structure of a pipeline. The C API conforms to the set of conventions given by the Qualcomm Elite SDK for audio processing on the DSP. The Java pipeline definition is mapped to a set of DSP compatible C routines.

method (*PLP_windowFeatures*) that will be invoked to process accumulated raw audio data. Developers can additionally define their custom algorithm components by providing a C implementation that conforms to the conventions we follow.

The wrapper functionality defines the structure of a sensor pipeline which consists of two primary computation types: 1) context triggers, added by the *trigger* method and executed as an optional first processing step; and 2) a series of transforms (feature extraction, classification) chained through the *apply* method so that the output from one transform serves as the input to another. Transforms are of two primary subtypes: primitive and composite. Primitive transforms are executed on a single computational unit as an atomic non-preempted operation. Composite transforms consist of multiple primitive transforms: currently we support a parallel transform operation that allows multiple transforms to

work concurrently with the same input on different computational units. The pipeline specification is a one-time operation needed to register the type and sequence of C methods that will be scheduled for execution on the various computational units.

Offloading service. As discussed in Chapter 3, the DSP has three hardware threads architected to look like a multi-core system. LEO runs continuously on the DSP as a service where we reserve one thread for the offloading decision logic. The pipeline stages of the sensing algorithms (additionally extracted features and classification) are executed in the other two threads. All CPU-DSP interactions (loading model parameters, sharing data) in our framework are facilitated through the FastRPC mechanism supported by the Hexagon SDK, in the same way as it was done for DSP.Ear and our deep audio inference engine. An Android background service task that is running on the CPU waits for the DSP to return a list of sensor tasks with assigned resources. If the CPU is in sleep mode, it is woken up by the DSP through a return from a FastRPC call to manage the assignment of tasks to other resources (cloud or GPU).

The schedule evaluation is timed periodically every t seconds (currently $t = 1$), with the primary unit of time being the length of audio frames. LEO accumulates raw sensor data in circular buffers, filters the type of data based on the registered triggers (motion, speech) and runs the scheduling algorithm every n-th frame (e.g., every 34th frame when its length is 30ms). The raw data buffers accumulate sensor samples over the largest registered window of processing for the corresponding sensor (e.g, 5 seconds for the microphone). The registered context triggers are typically short-duration routines (less than 5ms per 30ms audio frame, and less than 1ms per 4s accelerometer frame) and we interleave their execution with the schedule evaluation in the same thread.

Offline training. We expect developers to train their machine learning models offline and provide the model parameters required by our library of classification algorithms in the form of text files which we parse when an app is started/accessed for the first time. For instance, the Gaussian Mixture Models are encoded in the format defined by the HTK toolkit [23] which we used for the training of the DSP.Ear algorithms from Chapter 3. Our parsing utility is written in C++ so that it can supply the model parameters directly in the JNI bridge.

GPU support. We add GPU support for the most computationally demanding algorithms by reusing the implementations built in Chapter 5: Speaker Identification, Emotion Recognition and Keyword Spotting. Enabling GPU parallel execution with our framework requires incorporating additional energy into the scheduling objective function that captures the overhead of approaching the GPU (setting up computation and transferring buffers from the CPU host to GPU device memory). Further, custom algorithms not included in LEO's library would require additional programmer effort to provide an OpenCL implementation in addition to the DSP-compliant C version used by default. Interfacing with the GPU is mediated through the background CPU service which flattens the matrix of features required by the heavy classification stages into contiguous blocks of memory that can be copied with OpenCL commands to GPU device memory.

6.5 Prototype evaluation

In this section we evaluate LEO's overhead, the energy improvement over baseline offloading and the energy consumption when running example sensing apps under common smartphone usage and varying network conditions. By default, we use the base version of LEO that handles the heterogeneous processing units with full algorithm support and unified C programming model (CPU, DSP, and cloud). We discuss the implications of incorporating the GPU as an extra resource in a separate subsection. The main findings are:

- The cross-app scheduling decisions made by LEO are on average 5% worse than those made by an "oracle" knowing the optimal offloading of tasks to resources.
- LEO is between 1.6 and 3 times more energy efficient than off-the-shelf CPU-based APIs enhanced with conventional cloud offloading of classification tasks. Compared to a general-purpose offloader enhanced to use the DSP, LEO requires only a fraction of the energy ($< \frac{1}{7}$) to build a schedule and is still up to 19% more energy efficient for medium and heavy workloads.
- The overhead of the DSP Sensing Scheduler is low ($< 0.5\%$ of the battery daily), allowing frequent rescheduling.
- Considering the smartphone daily usage patterns of 1320 Android users, in more than 90% of the cases LEO is able to operate for a full day with a single battery charge with other apps running on the phone, while providing advanced services such as keyword spotting, activity and emotion recognition.

6.5.1 Baselines definition

Here we introduce commonly found scheduling/offloading alternatives that we are used to compare the performance of LEO.

- *DSP+Cloud.* This strategy uses the DSP for feature extraction and then ships the features to a remote server for processing.
- *CPU+Cloud.* This is an alternative that follows a conventional cloud offloading strategy where features are extracted locally on the CPU and then sent to the cloud for classification.
- *Greedy Local Heuristic.* A greedy strategy offloads sensing tasks locally on the mobile device by first pushing to the DSP the CPU-expensive computations. The tasks are sorted in descending order of their CPU-to-DSP energy ratio so that those with the largest *energy gain* factors are prioritised for execution on the DSP.

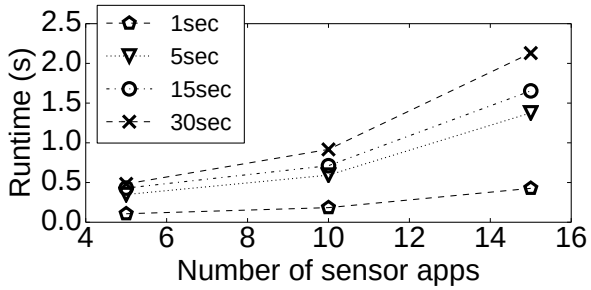


Figure 6.5.1: Runtime of the scheduling algorithm as a function of the number of apps and the length of the rescheduling interval.

- *Delay-Tolerant.* To demonstrate the huge performance boosts from delayed sensor inference execution, we provide a delay-tolerant version of LEO that runs the optimisation solver with relaxed deadline constraints once every minute ($\tau = 60$ seconds).
- *MAUI-DSP.* MAUI [80] is a general-purpose offloader that targets all categories of mobile apps and relies on code annotations to tag methods that can be executed remotely. It orchestrates cloud offloading for managed code, whereas sensor processing specialised for heterogeneous processors runs in an unmanaged environment. The core program partitioning algorithm is similarly based on mathematical optimisation but performs only binary split decisions (execute locally on the CPU or remotely on a server). We implement the MAUI optimisation model with *lp_solve* [30] as an enhanced baseline capable of leveraging the DSP locally in the following manner. All pipeline methods that can be executed remotely are flagged as remote-able, and the solver runs its local-remote binary partitioning logic with respect to the DSP, i.e. it will always prefer the low-power DSP over the CPU to compute sensor tasks locally. The solver logic is written in the AMPL modelling language [6] and runs as a service in cloud, while the mobile device is intended to communicate schedules by sending input parameters (sensing tasks identifiers, resource availability) as a JSON string. MAUI annotations are not explicitly implemented, and neither is runtime profiling of the methods. Instead, the solver leverages domain-specific information such as the type of pipeline methods, their expected runtime and energy characteristics from the offline profiling.

6.5.2 LEO's overhead

Runtime. The runtime of the scheduling algorithm solving the optimisation problem on the DSP is largely determined by the number of sensing apps to be managed. In Figure 6.5.1 we plot the runtime of the scheduler as a function of two parameters: the number of managed sensor apps and the rescheduling interval. The runtime for scheduling the execution of 5 apps every second is 107 milliseconds which allows frequently revising

the offloading decisions at the expense of a relatively small added latency. In addition, if the algorithm runs once every second when there are generated sensor tasks (triggered by the presence of relevant events such as speech or motion), the daily energy budget for the scheduling on a 2300mAh battery would amount to $< 0.5\%$. We can attribute the success of this relatively low overhead to two factors. First, although the optimisation search space grows exponentially in the number of apps, the low latency is enabled by the heuristic algorithm scaling polynomially instead of exponentially with the number of apps. Second, the low energy is maintained by running the scheduler entirely on the DSP and independently from the energy-hungry CPU.

Scheduling capacity. The total response time of LEO is 2 times the rescheduling interval plus the time needed to produce a schedule (which amounts to ≈ 2.1 seconds). In other words, the scheduler provides soft real-time guarantees which are sufficient for notification-style sensing apps (e.g., mute the device when entering a conversation, or trigger services based on voice commands). Whereas the scheduling algorithm can solve optimisation problems within 500ms for 15 apps on the DSP, typically only a small proportion of the apps will be executed on the DSP as it becomes easily overwhelmed. With 2 hardware threads reserved for app-specific logic, the DSP is able to process with real-time guarantees the feature extraction stages of several apps of the complexity of our examples. Longer-running classification stages need to be broken down into subcomputations, in which case the DSP could typically process a subset of these additional tasks from this stage for one to two more apps. This break-down is achieved through the Pipeline Partitioning discussed in Section 6.3.1.

DSP memory. With the runtime memory limit of 8MB on the DSP (see Section 2.3 of Chapter 2), we use 2MB for system parameters and application models (including 1 DNN and 5 emotion or speaker GMMs). If the DSP Sensing Scheduler revises the joint app schedule every 30 seconds, we would also need approximately 480KB of memory to buffer raw microphone data sampled at a rate of 8KHz. We recall that we use the buffering to monitor the exact workload generated by the currently activated sensor apps. The rest of the memory is reserved to useful application data such as accumulated inferences.

6.5.3 Optimality of offloading schedules

Here we investigate how close our scheduling heuristics as well as straw-man offloading variants are to an optimal schedule. We generate example sensing tasks to be offloaded and compare the generated solutions to the optimal ones produced by an optimisation solver. We create sensing traces with a workload that matches 30 seconds of relevant sensing context (detected motion and speech) and vary the number of apps to be scheduled in each trace. For each number of apps we create 10 different traces (where applicable) by changing the underlying set of sensor apps. Application sets are sampled with repeats from the family of implemented example apps shown in Table 6.3.1. The generated example configurations are expressed as mixed integer linear programming (MILP) problems via the

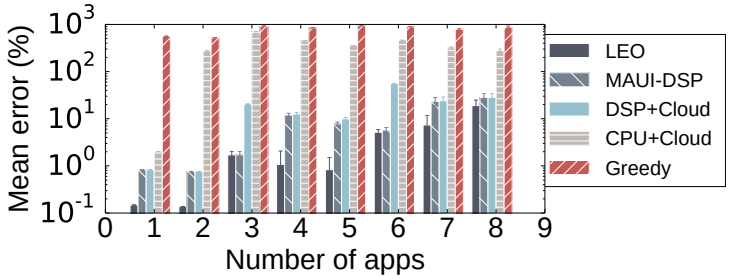


Figure 6.5.2: Deviation from the energy of the optimal solution when the offloading variants generate a schedule.

AMPL modelling language [6] and fed to the optimisation solver GLPK [18]. We observe significant, on the order of minutes or more, delays in the termination of the solver when the number of scheduled apps exceeds 8, which is why we limit this number when presenting the results.

In Figure 6.5.2 we plot how far off percentage-wise the offloading solutions are from the global optimum found by the GLPK solver. The results show that *LEO produces generally good solutions that are on average within* 5% *away from the optimal ones*. In contrast, the closest among the alternatives, DSP+Cloud and MAUI-DSP, are on average 19% and 10% away from the optimum respectively. As expected, LEO's (and the alternatives') error increases with the rise in number of scheduled apps to reach 19% when 8 apps are scheduled. Nevertheless, we believe that the DSP Sensing Scheduler provides a much needed optimality trade-off to make the offloading decisions practical and executable on the DSP.

6.5.4 LEO vs alternatives

In this subsection we compare the performance of LEO in terms of energy consumption against the commonly found offloading baselines defined in Section 6.5.1.

Experimental setup. We introduce three example scenarios covering the spectrum from light to heavy sensing app usage. Depending on the scenario, the user has subscribed for the services of a different subset of the apps shown in Table 6.5.1. To maintain a mixture of apps with a variety of deadlines, we coarsely split the apps into two groups in the following way. We impose near real-time requirements for the accelerometer-based inference algorithms as well as the Keyword Spotting, Speaker Counting and Stress Detection apps by setting their inference deadlines to be equal to their processing period, and set the heavier Emotion Recognition and Speaker Identification pipelines to be tolerant to larger 10-second delays in obtaining an inference (double their period). The delay tolerance for these sensor apps is set as an example to diversify the timeliness requirements.

	Heavy (H)	Medium (M)	Light (L)
Activity Recognition	✓	✓	✓
Step Counting	✓	✓	
Speaker Counting	✓		✓
Emotion Recognition	✓		✓
Speaker Identification	✓	✓	
Stress Detection	✓	✓	
Keyword Spotting	✓	✓	

Table 6.5.1: Applications used in the workload scenarios.

We generate 100 1-minute long sensor traces per scenario with relevant events sampled from a uniform random distribution. Such events are detected speech and motion that trigger the generation of sensor jobs. We note that even though the length of the sensing trace appears relatively short, it is sufficiently long to enable the sensor apps to generate a large number of jobs. An Emotion Recognition app, for instance, will create 12 jobs per minute given continuous speech and features are extracted for classification every 5 seconds (Table 6.3.1), whereas the Keyword Spotting app would produce 60 jobs per minute. The saturation of sensing context (speech, motion) that generates pipeline tasks varies from 5% to 100% in the artificial traces. For instance, a trace that is 1 minute long might contain 20 seconds of speech (33%) spread throughout the whole interval and grouped into several patches of continuous speech. We replay the traces for each offloading strategy and evaluate the energy consumption depending on the produced distribution of computations among offloading resources and CPU.

System load and energy profiling. Power measurements are obtained with a Monsoon Power Monitor [34] attached to the MDP. The average base power of maintaining a wake lock on the CPU with a screen and WiFi off is 295mW for the MDP. Each app is profiled separately for energy consumption by averaging power over 10 runs on the CPU, DSP and GPU where applicable. To obtain the power contributed by the sensor processing algorithms only, we subtract the base power from the total power of running the apps in background mode with a screen off. No other background services are running during the profiling apart from the system processes. We confirm that the total sensing system energy consumption is additive as long as the normalised CPU load on the MDP remains below $\approx$ 80%. Thus, the total energy for a sensing trace is an additive function of the energy expenditure of individual tasks under moderate CPU utilisation. As reported in the evaluation of DSP.Ear from Chapter 3, popular mobile apps from various categories that follow a processing pattern different from the sense-transform-classify one rarely push CPU utilisation beyond 25%. To mitigate any potential interference, we limit the amount of concurrent CPU threads working on sensing tasks to two which keeps the extra CPU load incurred by sensor processing below 40%. If the scheduler decides to use the CPU for computation under high system loads, interference with other services is inevitable unless sensor processing is cancelled. In such extreme conditions, delays in the response time of

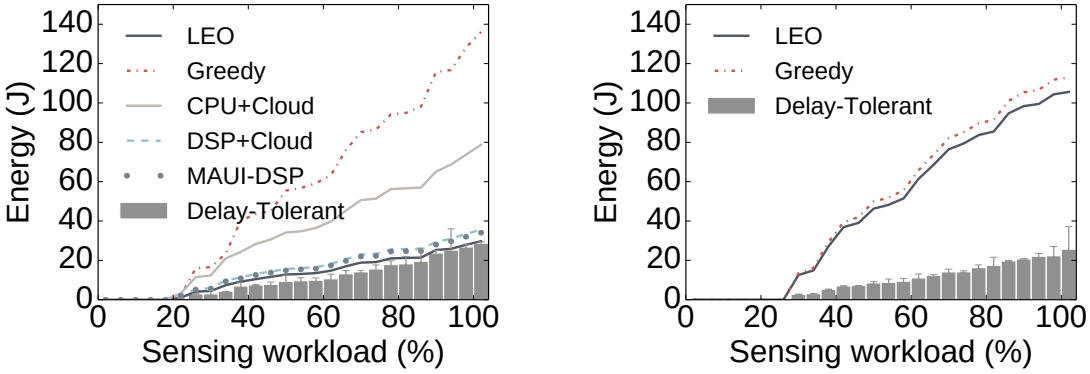
	WiFi 5Mbps			WiFi 1Mbps			3G 0.8Mbps			3G 0.4Mbps			No connectivity		
	H	M	L	H	M	L	H	M	L	H	M	L	H	M	L
LEO Delay	0.87	0.86	1.00	0.89	0.70	1.00	0.57	0.50	0.58	0.37	0.33	0.32	0.23	0.21	0.30
Greedy	5.30	4.36	4.74	3.97	3.60	3.19	2.64	2.54	1.90	1.64	1.69	1.02	1.04	1.08	1.00
CPU+Cloud	2.86	2.67	2.91	2.49	2.40	3.02	2.07	2.13	1.77	1.75	1.90	1.67	n/a	n/a	n/a
DSP+Cloud	1.17	1.24	1.00	1.20	1.24	1.00	1.23	1.30	1.00	1.21	1.34	1.00	n/a	n/a	n/a
MAUI-DSP	1.13	1.19	1.00	1.11	1.15	1.00	1.08	1.16	1.00	1.04	1.17	1.00	n/a	n/a	n/a

Table 6.5.2: Mean factors showing the amount of energy expended by the baselines relative to LEO. A factor of x means that the offloading alternative expends x times the amount of energy for the same workload-connectivity scenario. See next figure for example baseline energy absolute values in Joules.

services is expected (e.g., video playback, game play) as well as an increase in the total energy consumption.

Baseline comparison results. In Table 6.5.2 we display the relative amount of energy incurred by the offloading strategies when compared to LEO. The numbers show how many times the total energy of LEO an offloading alternative consumes. *In all of the resource availability and workload scenarios LEO succeeds in delivering better energy profiles than the alternatives.* The cloud-based baselines, for example, that always perform the classifications remotely and *do not perform cross-app resource optimisation*, fail to spot optimisation opportunities where processing the classification stages (deeper into the pipeline) on the DSP may be cheaper than remote computations. As a result, the CPU+Cloud strategy consistently consumes 1.6 to 3 times more energy than LEO, whereas the DSP+Cloud alternative introduces significant 17% to 34% overheads under heavy and medium workloads. Compared to CPU+Cloud, the DSP+Cloud baseline reduces energy consumption by ≈ 2 times on average across the workloads, which is a significant improvement. This energy reduction can be thought of as the gains of simply leveraging a DSP without benefiting from any cross-resource optimisation. With principled scheduling of sensor tasks, as we have already shown, energy gains can be up to 34% higher with LEO than with DSP+Cloud for medium to heavy workloads.

In Figure 6.5.3 we plot the mean energy expended by the sensing pipelines when following the various offloading schedules as a function of the sensing workload (e.g. proportion of speech captured by the microphone) for the sensing traces. Under relatively good WiFi throughput (5Mbps) LEO consumes 30J for performing all sensor tasks when there is 100% saturation of the sensing context (continuous speech/walking) in the trace. To put this number into perspective and assuming 4.5 hours of talking on average during the day [130], LEO would drain 26% of the capacity of a standard 2300mAh battery (such as the one of a Nexus 5 [21] based on the Snapdragon 800 SoC) to fully process the sensor data for the above mentioned set of accelerometer and microphone apps while importantly maintaining timeliness guarantees. The best among alternatives MAUI-DSP scheduler with its $\approx 20\%$ energy overhead would drain the notably higher 31% of the battery to process the same amount of sensor data.



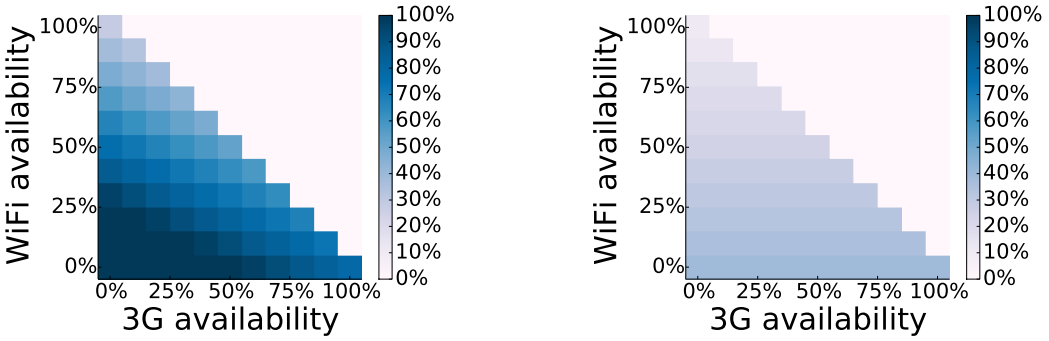
(a) WiFi 5Mbps

(b) No connectivity

Figure 6.5.3: Energy consumption of the offloading strategies compared against the delay-tolerant LEO as a function of the sensing workload saturation for the medium load scenario (M).

Why not MAUI? A general-purpose offloader such as MAUI-DSP significantly outperforms naive cloud offloading alternatives, yet there are multiple workload scenarios where LEO can maximise the energy gains even further. For example, under medium and heavy loads LEO can be up to 19% more efficient compared to the version of MAUI enhanced with DSP processing capabilities. This improvement can be attributed to two factors: 1) MAUI's local-remote binary partitioning model does not explicitly model the heterogeneity of local resources and may miss more optimal scheduling configurations that involve multiple local processing units; 2) MAUI applies its offloading decisions structurally for the program at the method level, whereas LEO exploits algorithm semantics to allow copies of the same method with different data (e.g., speaker GMM probability estimation) to be run in parallel on different processing units and cloud. In addition, the original MAUI uses the network to communicate schedules with a remote server to save energy, but still network transfers are mediated through the CPU in a high-power active state (hundreds of mW). Compared to our DSP Sensing Scheduler, MAUI would require about 7 to 10 times more energy for the schedule computation given a rescheduling frequency of 1 second.

Why not Wishbone? Wishbone's [154] original problem formulation targets a slightly different goal: increase data rate and reduce network bandwidth. Minimising network bandwidth is not a good proxy for energy because recent mobile devices boast powerful multi-core CPUs that can easily cope with complex processing locally – this will incur a high energy cost but will minimise the amount of transferred data. To target on-device energy consumption, the objective needs to be redefined as we have done and latency constraints need to be added to ensure apps remain responsive. Further, frequently using an optimisation solver such as GLPK [18] incurs a large energy and computation overhead. We find that scheduling the tasks of no more than 7 or 8 sensor apps such as the examples we have implemented requires on average 100ms on the Snapdragon CPU which is fast enough



(a) Medium (b) Light

Figure 6.5.4: Percentage of the battery capacity needed for processing 4.5 hours of sensing context under varying network availability.

but the energy cost of running the scheduler there is high – more than 30 times higher than what LEO requires. Given this overhead the only alternative is to run Wishbone less frequently: the solver would need to schedule defensively the execution across resources for multiple apps. In our experiments this leads to missed opportunities of using resources and energy consumption that is more than 3 times higher than what LEO can deliver.

6.5.5 Practicality considerations

Varying network availability. In Figure 6.5.4 we plot the percentage of the battery needed by the system to fully process the sensor data for a sensing workload of 4.5 hours spent in conversations [130] under the medium and light application scenarios (Table 6.5.1) and as a function of the network availability. The consumed energy is mindful of the sampling of the sensors and the overhead of waking up the CPU. We vary the amount of time when the system would be able to offload part of the classifications via WiFi or 3G. The network throughput is set to 5Mbps for WiFi and 0.4Mbps for 3G (median uplink throughput for 3G is dependent on carrier but remains around 0.3-0.4Mbps [110]). According to a recent study of smartphone usage patterns of more than 15K mobile users [184], 50% of them are connected to WiFi, LTE or 3G for at least 80% of the time. Being able to offload processing assuming such cumulative wireless coverage in Figure 6.5.4 corresponds to draining around 67% of a 2300mAh battery for medium workloads and barely 27% for light scenarios. The figures for the medium workload are high but we stress we maintain near real-time responsiveness for most of the apps. *Should we relax the deadline constraints to use Delay-Tolerant LEO, we can drop these numbers to merely* 25% *and* 12% *for the medium and light scenarios respectively.*

Smartphone usage. To understand how the workloads in Table 6.5.1 affect the user experience we analyse a dataset of 1320 smartphone users, provided to us by the authors of AppJoy [192], where interactive mobile app usage is logged on an hourly basis. We

	WiFi		3G		
	5Mbps	1Mbps	0.8Mbps	0.4Mbps	local
LEO-GPU	1.00	1.00	0.74	0.53	0.38
DSP+GPU	2.08	1.55	1.03	0.64	0.46
LEO-GPU (5s)	1.00	0.76	0.49	0.29	0.22
DSP+GPU (5s)	1.13	0.80	0.51	0.31	0.25

Table 6.5.3: Mean factors showing the amount of energy expended by the alternatives relative to LEO. The bracketed names refer to the same scheduling strategies when the rescheduling interval is set to 5 seconds which relaxes the deadline constraints for real-time apps and promotes batched GPU execution.

replay the user daily traces with LEO running in the background and add to all traces the workload of typical background services (a mail client, Facebook, Twitter, music playback) in a manner similar to DSP.Ear from Chapter 3. Assuming the previously mentioned 80% wireless network coverage and 4.5 hours of speech on average in a day (as found by SocioPhone [130]), we find that with the Delay-Tolerant version of LEO *for more than* 80% *or* 93% *of the daily usage instances the users would be able to sustain a full* 24*-hour day of operation without recharging a* 2300*mAh battery when the sensing apps from the medium and light scenarios are active respectively.*

6.5.6 GPU acceleration

In this subsection we investigate the implications of scheduling computations when an additional massively parallel heterogeneous processor such as the Qualcomm Adreno 330 GPU [2] is added to the pool of resources available to LEO. We build two scheduling alternatives to streamline our analysis: 1) *LEO-GPU* which follows LEO's scheduling logic and brings the GPU as an optional resource for the heavy classification algorithms (as discussed in Section 6.4); 2) *DSP+GPU* which always uses the GPU for the algorithms that can be executed there, extracts features on the DSP and any routines that cannot meet the deadlines on either of these processors are run on the CPU.

In Table 6.5.3 we show the proportion of LEO's energy the GPU-enhanced alternatives would spend in order to process the workloads from Table 6.5.1 under varying network connectivity. For each connectivity use case, the numbers are averaged across the heavy, medium, and light scenarios. With slower connections, the GPU-enhanced strategies spend a fraction (< 0.75 times) of vanilla LEO's energy to process the same workloads which suggests that the GPU is a viable offloading option cheaper than CPU and cloud offloading. With faster connections under tight deadline constraints (rescheduling every second by default), LEO-GPU spends the same amount of energy as LEO which means that the GPU is not used in these faster connectivity scenarios. In our experiments from Chapter 5 we find the GPU can deliver results faster than 5Mbps cloud under tight deadlines (≈ 6.5

times for the Keyword Spotting and ≈ 3 times for the Speaker/Emotion Recognition) but consumes more power which is dominated by the GPU initialisation stage repeated every second. As already shown in Chapter 5, however, batching has a desirable effect on energy consumption. If we pay the setup costs once and batch multiple computations for GPU execution, LEO-GPU (5s) begins to find opportunities where the total GPU energy is lower than 1Mbps cloud offloading. In other words, LEO-GPU automatically discovers we can compensate for the initially consumed high power with sheer GPU speed.

6.6 Discussion and limitations

We now examine key issues related to LEO's design.

Beyond the DSP. LEO is extensible beyond the three computation classes in our prototype to n-units by profiling each supported sensor algorithm under each new processor. As we have shown in Section 6.5.6, support for a GPU processor can be easily incorporated into LEO's resource pool modelling but may require extra programmer effort. We anticipate future LEO versions will provide a more comprehensive GPU support and fan-out feature extraction to multiple DSP varieties.

Extending sensor support. We largely focus on the microphone and accelerometer sensors as examples of a high and a low-data rate sensors, respectively. As these sensors provide a large variability in their requirements, they are an effective combination to understand the performance of LEO. However, our overall scheduling design is generic enough to support other phone sensors as long as the apps using them have a sense-transform-classify execution flow similar to the one we find among popular audio sensing apps.

Device fragmentation. Despite interacting with a co-processor, LEO remains portable to various phone models: LEO's components are OS-specific rather than device-specific, with two exceptions. First, each DSP variety needs a runtime and sensor algorithm library. Scaling LEO to use multiple DSPs would require adding support in the scheduling service for communication across different computational units and providing compatible implementations for the sensor algorithms. However, units such as the DSP in the Qualcomm 800 SoC is in dozens of smartphones, and recent DSP programmability trends revolve around adopting standard embedded programming tools such as C. Second, kernel drivers are needed to interface non-CPU components to the OS. But drivers are required only for each {OS, component} combination.

Programmability. We have provided Java wrapper functionality which allows developers to specify custom chains of sensor processing with a few lines of code when the library of pre-built algorithmic components are used. We acknowledge this may not always be possible, in which case the developers can integrate custom algorithms by providing DSP compatible C routines that conform to a set of conventions (briefly outlined in Section 6.4) most of which are set by the Qualcomm Hexagon SDK we used in our prototype.

Custom algorithms that do not conform with LEO's partitioning conventions will not benefit as much from the scheduler as structured algorithms. As long as the runtime of these custom algorithms is within the rescheduling interval, LEO will be able to find energy reduction opportunities for concurrent sensing apps without compromising the performance of the introduced new algorithms. This is because the algorithm execution can be treated as a single computational unit that involves the full pipeline (without exposing finer implementation details). When the custom algorithms are long running (severely exceeding the rescheduling interval), and given that the scheduler is not pre-emptive, there might be suboptimal resource utilisation choices in light of unforeseen future resource availability. However, such cases are expected to be rare since mobile sensor processing [141, 166, 140, 144, 94, 143] is typically periodic over the sensor stream with short repeated tasks to maintain tight mobile resource consumption and timeliness guarantees.

Proprietary sensor processing. Exposing an app's sequence of sensor processing steps to LEO entails intellectual property risks, but this is a problem relevant to a class of emerging orchestrators that operate with domain-specific signal processing knowledge [116, 119, 150]. As these solutions mature, new approaches will be developed to handle security risks. If developers trust the OS, sandboxing techniques [190] can be applied to prevent LEO from leaking sensitive information such as parameters for the classification models. If customised sensor processing C or OpenCL routines need to be added, code obfuscation techniques can be taken advantage of.

6.7 Related work

SpeakerSense [140], Little Rock [160], AudioDAQ [183], DSP.Ear and ZOE utilise low-power co-processors or purpose-built peripheral devices to achieve energy savings while supporting a fixed set of constantly running sensing applications. However, none of the above mentioned are designed to dynamically balance the workload when the set of actively running sensor apps changes.

Why not general-purpose offloaders? General-purpose offloaders [80, 161, 170, 196, 76, 165, 99], do not target the diverse sensor processing workloads explicitly and, as a result, may miss substantial optimisation opportunities. For example, MAUI [80] defines an offloading optimisation problem that performs binary local-remote split decisions (CPU vs. cloud) and inspects the general program structure but does not take advantage of domain-specific signal processing knowledge. As we have shown in Section 6.5, such knowledge can be leveraged for even greater energy benefits. Odessa [161] instruments individual apps to make offloading decisions for improving makespan and throughput but performs only per-app performance tuning instead of cross-app optimisations when apps share scarce mobile computing resources. Code in the Air [170] assumes that wireless connectivity is plentiful and that cloud offloading is the ultimate solution to reduce energy and improve throughput. With the advent of low-power co-processors these assumptions are seriously challenged: we

have demonstrated that optimal offloading configurations for sensor processing workloads are the ones that utilise a combination of all available computational resources. Last, VM migration mechanisms [76, 99] offer performance benefits but are difficult to deploy across architecturally different platforms.

Why not other sensor orchestrators? Existing sensor orchestrator frameworks [119, 116, 118, 137, 143, 122] approach the optimisation space from different angles in order to improve sensor processing on resource-constrained mobile devices, and often provide complementary functionality that can be executed side by side with LEO. Reflex [137], for example, eases programmability for co-processors [137] but does not explicitly optimise for energy efficiency. MobileHub [176] automatically rewrites app binaries to provide a sensor data notification filter that buffers data on the co-processor unlikely to result in an application notification and thus trigger pipeline processing. However, when the later stages of sensor processing pipelines are triggered and execution cannot be bypassed, applications will benefit from LEO automatically distributing chunks of these later-phase computations across resources. CAreDroid [86], on the other hand, presents a framework for automatically choosing among different implementations of the same sensor processing logic that leads to highest performance gains given the current device and user context. Again, we argue that once the relevant processing for an application is determined, sensor computations can be further optimised by *jointly* deciding for the currently active sensor apps on which resource their execution logic should be run. ORBIT [150] similarly to LEO uses profile-based partitioning of application logic to determine the most appropriate resource to use for a processing task issued by a data-intensive embedded app, but does not focus its optimisation on multiple simultaneously running apps.

Orchestrator [119] does not scale well with the increase in number of offloading configurations as it systematically explores subsets of offloading plans the number of which grows exponentially with offloading components and sensor apps. Wishbone [154] is very closely related to our work and we build upon some of its fundamentals (linear programming formal model, exploiting data flow semantics for the partitioning). As we have demonstrated in Section 6.5, it was originally designed to maximise a different optimisation objective and in the case of frequent rescheduling incurs a high energy overhead. SymPhoney [116] introduces a powerful utility-based model to deal with resource contention of sensor apps locally, whereas we attempt to maximise the efficiency of multiple apps with their original maximum utility across the various resources (DSP, CPU, GPU and cloud).

6.8 Conclusions

We have presented LEO, a mobile sensor inference algorithm scheduler enabling concurrent execution of complex sensor apps while maintaining near real-time responsiveness and maximising energy efficiency. LEO makes this possible by restructuring and optimally partitioning sensor algorithms (from simultaneously running apps) across heterogeneous

computational units, and revising this allocation dynamically at runtime based on fluctuations in device and network resources.

Chapter 7

Reflections and future work

The chapters presented a variety of optimisation approaches that empower mobile audio sensing apps with accurate and energy efficient continuous operation on smartphones and wearables. In this chapter we reflect on our contributions and how they support our thesis by answering the three major research questions outlined in Chapter 1. We also provide directions for future work.

There are two recent trends in the mobile app landscape that have fuelled the research in this dissertation. First, there is the growing adoption of mobile apps that rely on sensor data to track user behaviour and context. It is not unusual for a mobile user to install multiple of these apps on their device, and many of them are hungry for reliable and efficient audio sensing frameworks. Microphone-enabled apps such as Shazam (song recognition) and voice commands supported by digital assistants have a central role – they are capable of delivering rich inferences about user behaviour, but have high computational demands. Second, the range of heterogeneous processors found in modern mobile SoCs has significantly diversified to include not only CPUs and GPUs but also low-power cores and co-processors. Each of these processors has its own resource profile offering different trade-offs in the execution of sensing algorithms. Consequently, our thesis was that *to meet the workload demands of next-generation audio sensing apps and go beyond incremental energy savings we need to consider optimisation techniques that facilitate concurrent operation of audio algorithm chains and that maximise the shared use of the full range of mobile heterogeneous processors.*

To support our thesis we investigated several related threads of research. We studied how to make a better use of the processor hierarchy (low-power co-processors and CPU) by splitting the sensing algorithm execution in more effective ways. We validated our multi-tiered offloading approach by building two sensing systems for a smartphone and a wearable. We then specialised some of the core audio sensing algorithms for the DSP and GPU in an attempt to deliver highly efficient implementations that understand the hardware. Finally, we introduced dynamic scheduling that distributes sensor processing on

demand across the full range of heterogeneous processors (CPU, DSP, GPU) and cloud.

7.1 Summary of contributions

Here we revisit the research questions driving the work presented in the dissertation and provide a summary of our major contributions.

[**Research Question 1**] How can we enable the energy efficient concurrent and continuous operation of multiple mobile audio sensing apps with a high algorithm complexity?

[**Contribution 1**] In Chapter 3 we investigated how to leverage the hierarchy of low-power co-processors and CPU to support the energy efficient operation of two continuous sensing systems that infer multiple complex user behaviour cues such as expressed emotions or the number of speakers in a conversation. We presented a series of optimisation techniques that maximise the exposure of pipeline stages to co-processors or allow sensor processing to remain largely on the DSP with minimal assistance from the power-hungry CPU: admission filters, behaviour locality detectors, selective CPU offloading and cross-pipeline optimisations. We validated the importance of these techniques by comprehensively evaluating the energy consumption of the two proof-of-concept systems that relied on them. Consequently, we showed continuous sensing of deep audio inferences can be sustained with a single battery charge for at least 24 hours even in the presence of other apps running on the studied platforms.

[**Research Question 2**] What type of audio sensor inference algorithm specialisation do we need to perform in order to draw benefits from available heterogeneous mobile hardware?

[**Contribution 2**] In Chapters 4 and 5 we studied the advantages of designing audio algorithm implementations that are purpose-built for a DSP and a GPU. Chapter 4 devised memory-constrained deep learning models for fast, robust and accurate inference on low-power co-processors. We evaluated our inference engine based on Deep Belief Networks across a variety of audio sensing tasks to show that significant, an order of magnitude, reductions in runtime and energy can be achieved. We further demonstrated in Chapter 5 that large performance gains can also be attained with the design of a novel GPU optimisation engine. It employed key structural and memory access data parallel patterns to allow audio algorithm performance to be dynamically tuned as a function of GPU device specifications and model semantics. We showed that parameter optimised audio routines outperform all alternatives runtime-wise, and in batching scenarios also consume less energy than both a low-power DSP and cloud offloading.

[**Research Question 3**] How can we automate the process of maximising resource utilisation from multiple concurrent audio sensing apps without compromising app accuracy and responsiveness?

[**Contribution 3**] In Chapter 6 we implemented and evaluated the prototype of a novel

sensor algorithm scheduling framework, LEO, that targets the mixed workload of a variety of sensor apps with different timeliness requirements. The scheduler relies on a comprehensive library of sensing algorithms that serves as the building block for popular services found in the mobile sensing literature. LEO exploits model semantics to pre-define pipeline partitioning points so as to effectively distribute the computational load from multiple concurrent apps across heterogeneous processors and cloud. LEO runs as a service on the low-power DSP unit with a low overhead (<0.5% of the battery daily) thanks to a fast heuristic scheduling algorithm that solves a global resource optimisation problem. We found that LEO is not only between 1.6 and 3 times more energy efficient than conventional cloud offloading with CPU-bound sensor sampling, but also generates lower energy schedules than a general-purpose mobile app offloader with only a fraction of the power budget required to make the scheduling decisions.

To summarise, we believe that optimising shared resource use for concurrent audio sensing apps as well as specialising algorithm implementations for heterogeneous processors are the most critical types of performance tuning techniques to be considered in order to deliver accurate and low-energy system profiles. We expect our contributions will encourage developers to rethink their audio sensing designs and abandon the conventional approaches (e.g., exclusive cloud offloading) that we have proved obsolete in the heterogeneous resource era. Ultimately, this dissertation has laid the foundation for new research directions that enable audio sensing algorithms and system-level components to make better use of mobile hardware.

7.2 Future directions

The work we have done so far touches on some of the most immediate questions that the proliferation of sensor apps together with advances in processor technology raise. Opportunities for expanding on the current body of research are numerous, and here we summarise some of the more salient points that might need to be addressed.

Automated integration of existing sensor apps. In order to take advantage of the advanced DSP and GPU computation capabilities, existing apps with custom sensor processing routines have several options, none of which fully frees the developer from the burden of reimplementing some of their custom logic. We aid developers in building their DSP and GPU sensing apps by enabling them to reuse our library of algorithmic components. We also allow developers to integrate with our scheduling framework customised algorithm versions that comply with the set of conventions offered by our library and by the officially released SDKs for each processor. However, more research is needed in order to better automate the process of re-targeting CPU implementations to other units. The ideal case would be to design an all powerful compiler framework that takes existing C or Java code and produces highly optimised parallel versions for the DSP and GPU. However, fully automating the conversion process in such a way is a very hard problem [66, 65].

Static analysis techniques accompanied with code annotations, or custom domain specific languages, are more realistic and could help towards minimising the developer effort of migrating an existing code base to the DSP or GPU. New programmability mechanisms need to be investigated that provide a reasonable trade-off between the amount of effort exerted by the developer and the level of performance optimisation achieved in an automated manner.

TensorFlow [53] is a machine learning framework developed by Google that allows algorithms to be expressed as a symbolic graph of operations which are automatically compiled and executed on underlying hardware. So far the framework has largely been used for large-scale machine learning in the data centre, but its use is being extended to mobile devices where a single TensorFlow model can be mapped to DSP or GPU resources. This is a significant step towards bridging the gap between ever-increasing machine learning model requirements and local mobile computation. Yet the framework is quite general and research is needed into how it can be efficiently exploited for continuous sensing and signal processing tasks where co-processors need to be utilised without interruption, and where sensing workloads change dynamically based on user context.

Advanced hardware optimisations. A promising area of research is exploring how using more advanced hardware capabilities of state-of-the-art processors can improve sensor processing performance. Optimised assembly instructions, fixed point arithmetic, approximate computing [148], or dynamic voltage and frequency scaling are some examples of what can be further used to maximise sensor algorithm efficiency. For dynamic power scaling, for instance, the different operational clock frequencies of DSPs and GPUs provide specific power profiles which can be scaled automatically based on the current sensor workload. Further, deep learning accelerator chips [5, 15] are gaining a lot of attention recently and their role in the mobile sensor app landscape has yet to be determined. Given how typical sensor processing algorithms rely on machine learning primitives targeted by such accelerators, it is important to assess how such specialised chips can effectively service sensor workloads or interact with other components such as DSPs that excel at low-power sensor sampling and filtering.

Cross-device offloading. In the era of wearable electronics mobile users are becoming increasingly equipped with multiple devices: a smartwatch paired with a smartphone is a popular combination of mobile devices seen together. In this dissertation we have investigated computational offloading schemes that either operate entirely locally or use cloud services via a wireless link. However, a smartwatch, for example, might be able to offload via Bluetooth algorithm execution to a smartphone, thus taking advantage of nearby device computational resources. Cross-device offloading will prove a viable option when unused low-power units on the more powerful mobile platform are approached or when the smartphone is being charged – with no energy loss for the smartphone during charging, the less powerful smartwatch platform would harness execution beyond its capabilities or energy budget. A direction for future research is understanding what scheduling schemes are suitable to efficiently leverage this ecosystem of interconnected devices.

⋆ ⋆ ⋆

This dissertation has demonstrated that it is possible to run multiple audio sensing apps capable of deep inferences about user behaviour on smartphones and wearables without adversely affecting battery life. This has largely been enabled by a series of optimisation techniques that partition and distribute sensor processing across heterogeneous processors available in leading edge mobile SoCs. The insights drawn from building our integrated sensing systems are key enablers for the energy efficient and accurate continuous operation that audio sensing apps desperately need in order to deliver new rewarding user experiences. We hope this work lays the foundation for further examination of mobile sensing optimisation that focuses on internal inward looking optimisations of key machine learning and signal processing algorithms (rather than treating these as black boxes) not only for audio but other modalities as well, that collectively seek to increase the utilisation of heterogeneous mobile processors on embedded SoCs.

Bibliography

[1] Accupedo Pedometer. http://www.accupedo.com/. Accessed: 2017-06-25.

[2] Adreno GPU SDK. https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu. Accessed: 2017-06-25.

[3] Alexa Commands. https://www.cnet.com/how-to/the-complete-list-of-alexa-commands/. Accessed: 2017-06-25.

[4] Amazon Echo. http://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E. Accessed: 2017-06-25.

[5] AMD Radeon Instinct GPU Accelerator for Deep Learning. http://www.anandtech.com/show/10905/amd-announces-radeon-instinct-deep-learning-2017. Accessed: 2017-06-25.

[6] AMPL modeling language. http://ampl.com/. Accessed: 2017-06-25.

[7] Android Sensor APIs. http://developer.android.com/guide/topics/sensors/index.html. Accessed: 2017-06-25.

[8] Android Wear hardware review: Sometimes promising, often frustrating. http://tinyurl.com/m9zb3yz. Accessed: 2017-06-25.

[9] Apple iPhone 7 Specification. http://www.apple.com/uk/iphone-7/specs/. Accessed: 2017-06-25.

[10] Apple Motion Core API. https://developer.apple.com/library/iOs/documentation/CoreMotion/Reference/CoreMotion_Reference/index.html. Accessed: 2017-06-25.

[11] Apple Siri. https://www.apple.com/uk/ios/siri/. Accessed: 2017-06-25.

[12] ARM big.LITTLE technology. https://www.arm.com/products/processors/technologies/biglittleprocessing.php. Accessed: 2017-06-25.

[13] Auto Shazam. https://support.shazam.com/hc/en-us/articles/204457738-Auto-Shazam-iPhone-. Accessed: 2017-06-25.

[14] British Library of Sounds. http://sounds.bl.uk/. Accessed: 2017-06-25.

[15] Deep Learning Accelerator and Fathom Software Framework. http://www.movidius.com/news/movidius-announces-deep-learning-accelerator-and-fathom-software-framework. Accessed: 2017-06-25.

[16] Free Sound Effects. http://www.freesfx.co.uk/. Accessed: 2017-06-25.

[17] GameBench. https://www.gamebench.net/. Accessed: 2017-06-25.

[18] (GLPK) GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. Accessed: 2017-06-25.

[19] Google Cloud Dataflow. https://cloud.google.com/dataflow/model/programming-model. Accessed: 2017-06-25.

[20] Google Home. https://home.google.com/. Accessed: 2017-06-25.

[21] Google Nexus 5. https://www.qualcomm.com/products/snapdragon/smartphones/nexus-5-google. Accessed: 2017-06-25.

[22] Google Now. http://www.google.co.uk/landing/now/. Accessed: 2017-06-25.

[23] HTK Speech Recognition Toolkit. http://htk.eng.cam.ac.uk/. Accessed: 2017-06-25.

[24] Intel Edison. http://www.intel.com/content/www/us/en/do-it-yourself/edison.html. Accessed: 2017-06-25.

[25] Intel Unveils New Merrifield Smartphone Chip With Integrated Sensor Hub. https://www.laptopmag.com/articles/intel-merrifield-smartphone-chip. Accessed: 2017-06-25.

[26] Intel Xeon Phi. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. Accessed: 2017-06-25.

[27] iPhone 5s M7 Motion Coprocessor. https://www.apple.com/iphone-5s/specs/. Accessed: 2017-06-25.

[28] Jawbone Up 3. http://jawbone.com/store/buy/up3. Accessed: 2017-06-25.

[29] LG G Watch R. https://www.qualcomm.com/products/snapdragon/wearables/lg-g-watch-r. Accessed: 2017-06-25.

[30] lpsolve MILP Solver. http://lpsolve.sourceforge.net/5.5/. Accessed: 2017-06-25.

[31] Lumia SensorCore SDK. https://www.nuget.org/packages/LumiaSensorCoreSDK/. Accessed: 2017-06-25.

[32] Microsoft Band. http://www.microsoft.com/Microsoft-Band/. Accessed: 2017-06-25.

[33] Microsoft Cortana. http://www.windowsphone.com/en-gb/how-to/wp8/cortana/meet-cortana. Accessed: 2017-06-25.

[34] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/. Accessed: 2017-06-25.

[35] Moovit. http://www.moovitapp.com/. Accessed: 2017-06-25.

[36] Motorola Moto X. http://www.motorola.com/us/FLEXR1-1/moto-x-specifications.html. Accessed: 2017-06-25.

[37] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2017-06-25.

[38] NVIDIA Deep Learning SDK. https://developer.nvidia.com/deep-learning. Accessed: 2017-06-25.

[39] NVIDIA Tegra 4 family. http://www.nvidia.com/object/tegra-4-processor.html. Accessed: 2017-06-25.

[40] NVIDIA Tegra X1. http://www.nvidia.com/object/tegra-x1-processor.html. Accessed: 2017-06-25.

[41] Ok Google. http://www.makeuseof.com/tag/ok-google-20-useful-things-you-can-say-to-your-android-phone/. Accessed: 2017-06-25.

[42] OpenCL. https://www.khronos.org/opencl/. Accessed: 2017-06-25.

[43] Qualcomm Hexagon DSP. https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor. Accessed: 2017-06-25.

[44] Qualcomm Hexagon SDK. https://developer.qualcomm.com/mobile-development/maximize-hardware/multimedia-optimization-hexagon-sdk. Accessed: 2017-06-25.

[45] Qualcomm Snapdragon 400. https://www.qualcomm.com/products/snapdragon/processors/400. Accessed: 2017-06-25.

[46] Qualcomm Snapdragon 800 MDP. https://developer.qualcomm.com/hardware/mdp-820. Accessed: 2017-06-25.

[47] Qualcomm Snapdragon 800 Processors. http://www.qualcomm.com/snapdragon/processors/800. Accessed: 2017-06-25.

[48] RunKeeper. http://runkeeper.com/. Accessed: 2017-06-25.

[49] Scikit-Learn Python Library. http://scikit-learn.org/stable/. Accessed: 2017-06-25.

[50] Shake Gesture Library Windows Phone 8. http://code.msdn.microsoft.com/windowsapps/Shake-Gesture-Library-04c82d5f. Accessed: 2017-06-25.

[51] Snapdragon 800 Smartphones. http://www.qualcomm.com/snapdragon/smartphones/finder. Accessed: 2017-06-25.

[52] TensorFlow. https://www.tensorflow.org/. Accessed: 2017-06-25.

[53] TensorFlow Mobile. https://www.tensorflow.org/mobile/. Accessed: 2017-06-25.

[54] Trepn Profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler. Accessed: 2017-06-25.

[55] Waze Social GPS Maps and Traffic. https://www.waze.com/. Accessed: 2017-06-25.

[56] Will a smartphone replace your PC? http://www.govtech.com/blogs/lohrmann-on-cybersecurity/will-a-smartphone-replace-your-pc.html. Accessed: 2017-06-25.

[57] *Recent Advances in Deep Learning for Speech Research at Microsoft*. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), May 2013.

[58] R. J. Baken. *Clinical Measurement of Speech and Voice*. Taylor & Francis Ltd, London, 1987.

[59] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 272–285, New York, NY, USA, 2007. ACM.

[60] L. Bao and S. S. Intille. Activity recognition from user-annotated acceleration data. In *Pervasive computing*, pages 1–17. Springer, 2004.

[61] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[62] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[63] A. Brajdic and R. Harle. Walk detection and step counting on unconstrained smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 225–234, New York, NY, USA, 2013. ACM.

[64] E. O. Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[65] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceeding of the 41st Annual International Symposium on*

Computer Architecuture, ISCA '14, pages 217–228, Piscataway, NJ, USA, 2014. IEEE Press.

[66] S. Campanoni, T. M. Jones, G. H. Holloway, G.-Y. Wei, and D. M. Brooks. Helix: Making the extraction of thread-level parallelism mainstream. *IEEE Micro*, 32(4):8–18, 2012.

[67] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[68] R. Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, July 1997.

[69] G. Chechik, E. Ie, M. Rehn, S. Bengio, and D. Lyon. Large-scale content-based audio retrieval from text queries. In *Proceedings of the 1st ACM International Conference on Multimedia Information Retrieval*, MIR '08, pages 105–112, New York, NY, USA, 2008. ACM.

[70] G. Chen, C. Parada, and G. Heigold. Small-footprint keyword spotting using deep neural networks. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'14, 2014.

[71] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. *ICML-15*, 2015.

[72] K. T. Cheng and Y. C. Wang. Using mobile GPU for general-purpose computing – a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–4, April 2011.

[73] J. Chon and H. Cha. LifeMap: A smartphone-based context provider for location-based services. *IEEE Pervasive Computing*, 10(2):58–67, Apr. 2011.

[74] T. Choudhury, G. Borriello, S. Consolvo, D. Haehnel, B. Harrison, B. Hemingway, J. Hightower, P. P. Klasnja, K. Koscher, A. LaMarca, J. A. Landay, L. LeGrand, J. Lester, A. Rahimi, A. Rea, and D. Wyatt. The mobile sensing platform: An embedded activity recognition system. *IEEE Pervasive Computing*, 7(2):32–41, Apr. 2008.

[75] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 54–67, New York, NY, USA, 2011. ACM.

[76] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[77] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[78] I. Constandache, S. Gaonkar, M. Sayler, R. R. Choudhury, and L. P. Cox. EnLoc: Energy-efficient localization for mobile phones. In *INFOCOM*, pages 2716–2720. IEEE, 2009.

[79] C. Cotta and A. J. Fernndez. Memetic algorithms in planning, scheduling, and timetabling. In K. P. Dahal, K. C. Tan, and P. I. Cowling, editors, *Evolutionary Scheduling*, volume 49 of *Studies in Computational Intelligence*, pages 1–30. Springer, 2007.

[80] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62. ACM, 2010.

[81] A. de Cheveigné and H. Kawahara. YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[82] L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2013.

[83] L. Deng and D. Yu. Deep learning: Methods and applications. Technical Report MSR-TR-2014-21, January 2014.

[84] T. Denning, A. Andrew, R. Chaudhri, C. Hartung, J. Lester, G. Borriello, and G. Duncan. Balance: towards a usable pervasive wellness application with accurate activity inference. In *Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, page 5. ACM, 2009.

[85] S. Dixon. Onset Detection Revisited. In *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-06)*, pages 133–137, Montreal, Quebec, Canada, Sept. 2006.

[86] S. Elmalaki, L. Wanner, and M. Srivastava. CAreDroid: Adaptation framework for android context-aware applications. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 386–399, New York, NY, USA, 2015. ACM.

[87] B. Fang, N. D. Lane, M. Zhang, A. Boran, and F. Kawsar. BodyScan: Enabling radio-based sensing on wearable devices for contactless activity and vital sign monitoring. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 97–110, New York, NY, USA, 2016. ACM.

[88] Z. Fang, Z. Guoliang, and S. Zhanjiang. Comparison of different implementations of MFCC. *J. Comput. Sci. Technol.*, 16(6):582–589, Nov. 2001.

[89] J. Gemmell, G. Bell, and R. Lueder. MyLifeBits: a personal database for everything. *Communications of the ACM (CACM)*, 49(1):88–95, January 2006. also as MSR-TR-2006-23.

[90] P. Georgiev, S. Bhattacharya, N. D. Lane, and C. Mascolo. Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2017.

[91] P. Georgiev, N. D. Lane, C. Mascolo, and D. Chu. Accelerating mobile audio sensing algorithms through on-chip gpu offloading. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 306–318, New York, NY, USA, 2017. ACM.

[92] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. DSP.Ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 295–309, New York, NY, USA, 2014. ACM.

[93] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 320–333, New York, NY, USA, 2016. ACM.

[94] P. Georgiev, A. Noulas, and C. Mascolo. The call of the crowd: Event participation in location-based social services. *CoRR*, abs/1403.7657, 2014.

[95] P. Georgiev, A. Noulas, and C. Mascolo. Where businesses thrive: Predicting the impact of the Olympic Games on local retailers through location-based services data. *CoRR*, abs/1403.7654, 2014.

[96] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev. Compressing deep convolutional networks using vector quantization. *ICLR-15*, 2015.

[97] K. Gupta and J. D. Owens. Compute & memory optimizations for high-quality speech recognition on low-end GPU processors. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[98] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81, New York, NY, USA, 2014. ACM.

[99] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153–166, New York, NY, USA, 2013. ACM.

[100] K. Han, D. Yu, and I. Tashev. Speech emotion recognition using deep neural network and extreme learning machine. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[101] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[102] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *NIPS-15*, 2015.

[103] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu. Reshaping deep neural network for fast decoding by node-pruning. In *ICASSP-14, May 4-9, 2014*, pages 245–249, 2014.

[104] S. Hemminki, P. Nurmi, and S. Tarkoma. Accelerometer-based transportation mode detection on smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 13:1–13:14, New York, NY, USA, 2013. ACM.

[105] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *J. Acoust. Soc. Am.*, 57(4):1738–52, Apr. 1990.

[106] G. Hinton, L. Deng, D. Yu, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. S. G. Dahl, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012.

[107] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[108] S. Hodges, L. Williams, E. Berry, S. Izadi, J. Srinivasan, A. Butler, G. Smyth, N. Kapur, and K. Wood. SenseCam: A retrospective memory aid. In *UbiComp 2006: Ubiquitous Computing*, pages 177–193. Springer, 2006.

[109] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 381–392, New York, NY, USA, 2011. ACM.

[110] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 165–178, New York, NY, USA, 2010. ACM.

[111] J.-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In *ICASSP-13*, May 2013.

[112] A. Huqqani, E. Schikuta, S. Yea, and P. Chena. Multicore and GPU parallelization of neural networks for face recognition. In *International Conference on Computational Science, ICCS*, Procedia Computer Science, pages 349–358, London, UK, June 2013. Elsevier.

[113] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 1–14, Berkeley, CA, USA, 2011. USENIX Association.

[114] S. Ji, W. Xu, M. Yang, and K. Yu. 3D convolutional neural networks for human action recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(1):221–231, Jan. 2013.

[115] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM.

[116] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. SymPhoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[117] S. E. Kahou, C. Pal, X. Bouthillier, P. Froumenty, c. Gülçehre, R. Memisevic, P. Vincent, A. Courville, Y. Bengio, R. C. Ferrari, M. Mirza, S. Jean, P.-L. Carrier, Y. Dauphin, N. Boulanger-Lewandowski, A. Aggarwal, J. Zumer, P. Lamblin, J.-P. Raymond, G. Desjardins, R. Pascanu, D. Warde-Farley, A. Torabi, A. Sharma, E. Bengio, M. Côté, K. R. Konda, and Z. Wu. Combining modality specific deep neural networks for emotion recognition in video. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*, ICMI '13, pages 543–550, New York, NY, USA, 2013. ACM.

[118] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. SeeMon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[119] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Eigth Annual IEEE International Conference*

on Pervasive Computing and Communications, PerCom 2010, March 29 - April 2, 2010, Mannheim, Germany, pages 135–144, 2010.

[120] D. H. Kim, Y. Kim, D. Estrin, and M. B. Srivastava. SensLoc: Sensing everyday places and paths using less energy. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 43–56, New York, NY, USA, 2010. ACM.

[121] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[122] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In *15th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2016, Vienna, Austria, April 11-14, 2016*, pages 1–12, 2016.

[123] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 7–12, New York, NY, USA, 2015. ACM.

[124] N. D. Lane, Y. Chon, L. Zhou, Y. Zhang, F. Li, D. Kim, G. Ding, F. Zhao, and H. Cha. Piggyback crowdsensing (PCS): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 7:1–7:14, New York, NY, USA, 2013. ACM.

[125] N. D. Lane and P. Georgiev. Can deep learning revolutionize mobile sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 117–122, New York, NY, USA, 2015. ACM.

[126] N. D. Lane, P. Georgiev, C. Mascolo, and Y. Gao. ZOE: A cloud-less dialog-enabled continuous sensing wearable exploiting heterogeneous computation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 273–286, New York, NY, USA, 2015. ACM.

[127] N. D. Lane, P. Georgiev, and L. Qendro. DeepEar: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[128] N. D. Lane, M. Lin, M. Rabi, X. Yang, A. Doryab, H. Lu, S. Ali, T. Choudhury, A. Campbell, and E. Berke. *Bewell: A smartphone application to monitor, model and promote wellbeing*. IEEE Press, 2011.

[129] H. Lee, P. Pham, Y. Largman, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *NIPS-09*, pages 1096–1104. Curran Associates, Inc., 2009.

[130] Y. Lee, C. Min, C. Hwang, J. Lee, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song. SocioPhone: Everyday face-to-face interaction monitoring platform using multi-phone sensor fusion. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 375–388, New York, NY, USA, 2013. ACM.

[131] D. Li, I. K. Sethi, N. Dimitrova, and T. McGee. Classification of general audio data for content-based retrieval. *Pattern Recognition Letters*, 22(5):533–544, 2001.

[132] T. Li. Musical genre classification of audio signals. In *IEEE Transactions on Speech and Audio Processing*, pages 293–302, 2002.

[133] X. Li, L. Zhao, L. Wei, M. Yang, F. Wu, Y. Zhuang, H. Ling, and J. Wang. DeepSaliency: Multi-task deep neural network model for salient object detection. *CoRR*, abs/1510.05484, 2015.

[134] M. Liberman, K. Davis, M. Grossman, N. Martey, and J. Bell. Emotional prosody speech and transcripts. 2002.

[135] R. LiKamWa, Y. Liu, N. D. Lane, and L. Zhong. MoodScope: Building a mood sensor from smartphone usage patterns. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 389–402, New York, NY, USA, 2013. ACM.

[136] R. LiKamWa, Z. Wang, A. Carroll, F. X. Lin, and L. Zhong. Draining our glass: An energy and heat characterization of google glass. *CoRR*, abs/1404.1320, 2014.

[137] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 13–24, New York, NY, USA, 2012. ACM.

[138] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy trade-off for continuous mobile device location. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 285–298, New York, NY, USA, 2010. ACM.

[139] X. Liu, J. Gao, X. He, L. Deng, K. Duh, and Y. Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 912–921, 2015.

[140] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu. SpeakerSense: Energy

efficient unobtrusive speaker identification on mobile phones. In *Proceedings of the 9th International Conference on Pervasive Computing*, Pervasive'11, pages 188–205, Berlin, Heidelberg, 2011. Springer-Verlag.

[141] H. Lu, D. Frauendorfer, M. Rabbi, M. S. Mast, G. T. Chittaranjan, A. T. Campbell, D. Gatica-Perez, and T. Choudhury. StressSense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 351–360, New York, NY, USA, 2012. ACM.

[142] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. SoundSense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 165–178, New York, NY, USA, 2009. ACM.

[143] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 71–84, New York, NY, USA, 2010. ACM.

[144] C. Luo and M. C. Chan. SocialWeaver: Collaborative inference of human conversation networks using smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 20:1–20:14, New York, NY, USA, 2013. ACM.

[145] A. Mayberry, P. Hu, B. Marlin, C. Salthouse, and D. Ganesan. iShadow: Design of a wearable, real-time mobile gaze tracker. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 82–94, New York, NY, USA, 2014. ACM.

[146] W. S. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biology*, 5(4):115–133, Dec 1943.

[147] I. McLoughlin, H. Zhang, Z. Xie, Y. Song, and W. Xiao. Robust sound event classification using deep neural networks. *Trans. Audio, Speech and Lang. Proc.*, 23(3):540–552, Mar. 2015.

[148] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, Mar. 2016.

[149] D. Mizell. Using gravity to estimate accelerometer orientation. In *Proceedings of the 7th IEEE International Symposium on Wearable Computers*, ISWC '03, pages 252–, Washington, DC, USA, 2003. IEEE Computer Society.

[150] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing. ORBIT: A smartphone-based platform for data-intensive embedded sensing applications. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 83–94, New York, NY, USA, 2015. ACM.

[151] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: Rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 323–336, New York, NY, USA, 2008. ACM.

[152] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P Report 826, California Institute of Technology, 1989.

[153] S. Nath. ACE: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[154] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Prole-based Partitioning for Sensornet Applications. In *NSDI 2009*, Boston, MA, April 2009.

[155] S. Nirjon, R. Dickerson, J. Stankovic, G. Shen, and X. Jiang. sMFCC: Exploiting sparseness in speech for fast acoustic feature extraction on mobile devices – a feasibility study. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 8:1–8:6, New York, NY, USA, 2013. ACM.

[156] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 403–416, New York, NY, USA, 2013. ACM.

[157] S. Ntalampiras, I. Potamitis, and N. Fakotakis. Acoustic detection of human activities in natural environments. *Journal of Audio Engineering Society*, 2012 2012.

[158] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim. Design and performance evaluation of image processing algorithms on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):91–104, Jan. 2011.

[159] T. Plötz, N. Y. Hammerla, and P. Olivier. Feature learning for activity recognition in ubiquitous computing. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1729–1734. AAAI Press, 2011.

[160] B. Priyantha, D. Lymberopoulos, and J. Liu. LittleRock: Enabling energy-efficient continuous sensing on mobile phones. *IEEE Pervasive Computing*, 10(2):12–15, 2011.

[161] M. Ra, A. Sheth, L. B. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*, pages 43–56, 2011.

[162] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 1–10, New York, NY, USA, 2012. ACM.

[163] M. Rabbi, S. Ali, T. Choudhury, and E. Berke. Passive and in-situ assessment of mental and physical well-being using mobile sensors. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 385–394, New York, NY, USA, 2011. ACM.

[164] L. R. Rabiner. Readings in speech recognition. chapter A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[165] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, pages 73–84, New York, NY, USA, 2011. ACM.

[166] K. K. Rachuri, M. Musolesi, C. Mascolo, P. J. Rentfrow, C. Longworth, and A. Aucinas. EmotionSense: A mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, Ubicomp '10, pages 281–290, New York, NY, USA, 2010. ACM.

[167] T. Rahman, A. T. Adams, M. Zhang, E. Cherry, B. Zhou, H. Peng, and T. Choudhury. BodyBeat: A mobile system for sensing non-speech body sounds. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 2–13, New York, NY, USA, 2014. ACM.

[168] A. Rakotomamonjy and G. Gasso. Histogram of gradients of time-frequency representations for audio scene detection. *CoRR*, abs/1508.04909, 2015.

[169] S. Rallapalli, A. Ganesan, K. Chintalapudi, V. N. Padmanabhan, and L. Qiu. Enabling physical analytics in retail stores using smart glasses. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 115–126, New York, NY, USA, 2014. ACM.

[170] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM.

[171] D. A. Reynolds. Gaussian mixture models. In *Encyclopedia of Biometrics, Second Edition*, pages 827–832. 2015.

[172] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the*

Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[173] J. Saunders. Real-time discrimination of broadcast speech/music. In *Proceedings of the Acoustics, Speech, and Signal Processing, 1996. On Conference Proceedings., 1996 IEEE International Conference - Volume 02*, ICASSP '96, pages 993–996, Washington, DC, USA, 1996. IEEE Computer Society.

[174] E. Scheirer and M. Slaney. Construction and evaluation of a robust multifeature speech/music discriminator. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97)-Volume 2 - Volume 2*, ICASSP '97, pages 1331–, Washington, DC, USA, 1997. IEEE Computer Society.

[175] C. Shen, S. Chakraborty, K. R. Raghavan, H. Choi, and M. B. Srivastava. Exploiting processor heterogeneity for energy efficient context inference on mobile phones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 9:1–9:5, New York, NY, USA, 2013. ACM.

[176] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 227–238, New York, NY, USA, 2015. ACM.

[177] N. Singhal, I. K. Park, and S. Cho. Implementation and optimization of image processing algorithms on handheld GPU. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 4481–4484, Sept 2010.

[178] M. Smith and T. Barnwell. A new filter bank theory for time-frequency representation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):314–327, Mar 1987.

[179] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 261–274, New York, NY, USA, 2005. ACM.

[180] W.-T. Tan, M. Baker, B. Lee, and R. Samadani. The sound of silence. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 19:1–19:14, New York, NY, USA, 2013. ACM.

[181] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[182] E. Variani, X. Lei, E. McDermott, I. L. Moreno, and J. Gonzalez-Dominguez. Deep neural networks for small footprint text-dependent speaker verification. In *ICASSP-14*, pages 4052–4056. IEEE, 2014.

[183] S. Verma, A. Robinson, and P. Dutta. AudioDAQ: Turning the mobile phone's

ubiquitous headset port into a universal data acquisition interface. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 197–210, New York, NY, USA, 2012. ACM.

[184] D. Wagner, A. Rice, and A. Beresford. Device analyzer: Understanding smartphone usage. In *10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2013.

[185] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro. Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study. In *ICASSP*, pages 2629–2633. IEEE, 2013.

[186] H. Wang, D. Lymberopoulos, and J. Liu. Local business ambience characterization through mobile audio sensing. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 293–304, New York, NY, USA, 2014. ACM.

[187] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 179–192, New York, NY, USA, 2009. ACM.

[188] Z. Wu, T. Kinnunen, N. Evans, J. Yamagishi, C. Hanilci, M. Sahidullah, and A. Sizov. ASVspoof 2015: the first automatic speaker verification spoofing and countermeasures challenge. In *INTERSPEECH 2015, Automatic Speaker Verification Spoofing and Countermeasures Challenge, colocated with INTERSPEECH 2015, September 6-10, 2015, Dresden, Germany*, Dresden, ALLEMAGNE, 09 2015.

[189] C. Xu, S. Li, G. Liu, Y. Zhang, E. Miluzzo, Y.-F. Chen, J. Li, and B. Firner. Crowd++: Unsupervised speaker count with smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 43–52, New York, NY, USA, 2013. ACM.

[190] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, Bellevue, WA, 2012. USENIX.

[191] J. Xue, J. Li, and Y. Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In F. Bimbot, C. Cerisara, C. Fougeron, G. Gravier, L. Lamel, F. Pellegrino, and P. Perrier, editors, *INTERSPEECH*, pages 2365–2369. ISCA, 2013.

[192] B. Yan and G. Chen. AppJoy: Personalized mobile application discovery. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*.

[193] M. D. Zeiler, M. Ranzato, R. Monga, M. Z. Mao, K. Yang, Q. V. Le, P. Nguyen, A. W. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units

for speech processing. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 3517–3521, 2013.

[194] D. Zhang, T. He, Y. Liu, Y. Gu, F. Ye, R. K. Ganti, and H. Lei. Acc: Generic on-demand accelerations for neighbor discovery in mobile applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 169–182, New York, NY, USA, 2012. ACM.

[195] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.

[196] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, Oct. 2014. USENIX Association.

[197] Y. Zhang, K. Adl, and J. Glass. Fast spoken query detection using lower-bound dynamic time warping on graphical processing units. In *In Proc. ICASSP*, pages 5173–5176, 2012.

[198] M. Zhao, F. Adib, and D. Katabi. Emotion recognition using wireless signals. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 95–108, New York, NY, USA, 2016. ACM.

[199] G. Zhou, J. H. L. Hansen, and J. F. Kaiser. Nonlinear feature based classification of speech under stress. *IEEE Transactions on Speech and Audio Processing*, 9(3):201–216, 2001.