

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Leveraging Low-Level Code Optimization on Mobile and Wearable Devices

Alexandre Guimarães Gomes Correia



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

Supervisor: Pedro C. Diniz

Receiving Enterprise: Konk Consulting - Consultoria Informática S.A.

July 15, 2025

Leveraging Low-Level Code Optimization on Mobile and Wearable Devices

Alexandre Guimarães Gomes Correia

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Bruno Lima

Examiner: Prof. Diogo Gomes

Supervisor: Prof. Pedro C. Diniz

July 15, 2025

Resumo

Os dispositivos móveis e vestíveis são capazes de executar tarefas cada vez mais complexas e sensíveis à privacidade, ao mesmo tempo que estão sujeitos a restrições rigorosas de energia e latência. Esta dissertação investiga como a utilização de código de baixo nível pode melhorar o desempenho e a eficiência em dispositivos Android, em contraste com a *standard toolchain* Kotlin/Java. Depois de analisar o estado da arte, foram realizados quatro estudos utilizando Kotlin e C/C++ com o Android Native Development Kit (NDK). Estes estudos centraram-se na geração de números primos, utilizando o Crivo de Eratóstenes, a Fast Fourier Transform (FFT) e a multiplicação de matrizes quadradas. Ainda desenvolvemos uma aplicação para Android que realiza a detecção da *wake word* ou *hotword* utilizando a API em C do TensorFlow Lite, seguido do reconhecimento de comandos de voz para um conjunto limitado de commands, após a *wake word* ser detetada.

A exploração dos algoritmos revelou que o código nativo oferece benefícios limitados para o algoritmo Sieve, alcançando apenas um *speed-up* de 1,4x no melhor cenário, com o desempenho a degradar-se à medida que o tamanho do input aumenta. No entanto, o código nativo pode proporcionar um melhor controlo sobre a utilização da memória, permitindo o processamento de *inputs* de maiores dimensões do que a abordagem em Kotlin. A FFT iterativa implementada em código nativo alcançou um *speed-up* de 5,82x para um tamanho de *input* de 2^{20} , com o desempenho a melhorar à medida que o tamanho do *input* aumenta. Ao passo que o algoritmo de multiplicação de matrizes alcançou um *speed-up* de 6x com um tamanho de matriz de 2048, utilizando a *flag -O3* do compilador que permitiu ativar a vetorização de instruções.

Além disso, desenvolvemos uma aplicação composta por um *wake word detector* e um sistema reconhecimento de comandos de voz utilizando a API em C do TensorFlow Lite, que permitiu a execução em tempo real e *on-device*, ao mesmo tempo que implementámos a filtragem de ruído e voz humana. O nosso objetivo era comparar uma implementação nativa em C com uma em Kotlin para explorar potenciais optimizações. A avaliação empírica confirmou que a *pipeline* de áudio ficou funcional, usando aproximadamente 200 MB de memória e pode durar até 41 horas de funcionamento contínuo num ambiente composto por porções iguais (33,3...% cada) de silêncio, ruído aleatório e voz humana.

Keywords: Android, SDK, NDK, Kotlin, C/C++, native, mobile computing, ODML, wake word, hotword

Abstract

Mobile and wearable devices are capable of executing increasingly complex and privacy-sensitive tasks, all while remaining bound by strict energy and latency constraints. This dissertation investigates how utilizing low-level code can improve performance and efficiency on Android devices in contrast to the standard Kotlin/Java toolchain. After reviewing the state of the art, four studies were conducted using both Kotlin and C/C++ with the Android Native Development Kit (NDK). These studies focused on prime number generation, using the Sieve of Eratosthenes, the Fast Fourier Transform (FFT), and square matrix multiplication. Additionally, we developed an Android application that performs wake word or hotword detection using TensorFlow Lite C API, followed by voice command recognition for a limited set of commands after detecting a wake word.

The exploration of the algorithms revealed that native code offers limited benefits for the Sieve algorithm, achieving only a 1.4x speed-up in the best-case scenario, with performance degrading as the input size increases. However, native code provides better control over memory usage, enabling the processing of larger input sizes than the Kotlin approach. The iterative FFT implemented in native code achieves a speed-up of 5.82x for an input size of 2^{20} , with performance improving as input size increases. While the matrix multiplication algorithm achieved a speed-up of 6x in a matrix size of 2048, utilizing the compiler's -O3 optimization flag, which enabled instruction vectorization.

Moreover, the development of the wake word detector and voice command recognition application using the TensorFlow Lite C API allowed on-device real-time processing while also incorporating noise and voice activity filtering. One of our goals was to compare a native implementation in C with one in Kotlin to explore potential optimizations. Empirical evaluation confirmed a functional audio pipeline that consumed approximately 200MB of memory and could sustain up to 41 hours of continuous listening in an environment composed of equal portions (33,3...% each) of silence, random noise, and human voice.

Keywords: Android, SDK, NDK, Kotlin, C/C++, native, mobile computing, ODML, wake word, hotword

Acknowledgements

I would like to express my sincere gratitude to Pedro Diniz and Daniel Silva. A special thanks to Daniel, whose support and guidance throughout this project were indispensable. His dedication to helping me during my time at KonkConsulting was truly invaluable, and I am grateful for his generosity in sharing his expertise.

I am deeply grateful to my family, especially my mother Manuela and Tiago, for their constant love, encouragement, and understanding throughout my studies. Their support has been a steady source of strength for me.

To my girlfriend Catarina, thank you for always being there to offer comfort and support during moments of frustration. Your unwavering belief in me has meant the world.

Lastly, I want to thank all my friends, particularly those from Messe residence. Over the past few years, their support and friendship have made my academic journey much more enjoyable and rewarding.

Alexandre Correia

*“I have not failed.
I’ve just found 10,000 ways that won’t work.”*

Thomas A. Edison

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	1
1.3	Problem	2
1.4	Research Questions	2
1.5	Contribution	3
1.6	Dissertation Structure	3
2	Literature Review	4
2.1	Introduction	4
2.2	Search Strategy	4
2.3	Inclusion and Exclusion Criteria	5
2.4	Summary	6
3	State of the Art	7
3.1	Introduction	7
3.2	Performance Improvements Through Native Code	7
3.2.1	Speedups Enabled by the NDK	7
3.2.2	Hardware-Accelerated Use Cases	8
3.3	Android NDK and Its Core Components	8
3.3.1	Key Features and Limitations	8
3.3.2	Core Interfaces, APIs, and Low-Level Functionality	8
3.3.3	Libraries and Services	9
3.3.4	Sensors and Hardware Access	9
3.3.5	JNI Integration	9
3.4	Energy Consumption Considerations	9
3.4.1	Cross-Language Comparisons	9
3.4.2	Developer Practices for Energy Optimization	10
3.4.3	Power Consumption Analysis Tools	10
3.5	Security Implications of Native Code	11
3.5.1	Potential Risks in JNI/NDK	11
3.5.2	Trusted Execution Environments and Secure ML	11
3.6	Emerging Use Cases: Machine Learning and IoT	12
3.6.1	Inference Frameworks and NNAPI	12
3.6.2	On-Device Speech Recognition	12
3.6.3	IoT Devices and Hybrid Frameworks	13
3.7	Summary	13

4	Proposed Solution	15
4.1	Introduction and Objectives	15
4.2	Proposed Approach	15
4.2.1	Wake Word Detection	15
4.2.2	Limited Voice Command Recognition	16
4.3	Experimental Process	16
4.4	Expected Results and Relevance	16
5	Proof of Concept: Sieve of Eratosthenes	18
5.1	Introduction	18
5.2	Implementation Details	18
5.2.1	Kotlin Implementation (SDK)	20
5.2.2	C/C++ Implementation (NDK)	22
5.3	Experimental Setup	24
5.3.1	Test Cases and Measurement	25
5.3.2	Device Specifications	25
5.3.3	Clang Compiler	26
5.3.4	Android Flags	26
5.4	Results and Analysis	26
5.5	Conclusions	28
6	Exploring Fast Fourier Transform and Matrix Multiplication	29
6.1	Introduction	29
6.2	FFT Algorithm	29
6.3	Matrix Multiplication	31
6.4	Conclusions	34
7	Wake Word Detection and Voice Command Application	35
7.1	Introduction	35
7.2	Project Setup	36
7.2.1	Development Enviroment	36
7.2.2	Open Wake Word library	36
7.2.3	Android NDK and TensorFlow Lite	38
7.2.4	Java Native Interface functions	38
7.3	Wake Word Detection	39
7.3.1	Audio Capture and Processing	39
7.3.2	Audio Sampling: WAV file vs Mic	41
7.3.3	Audio Recording and Inference Threads	42
7.3.4	Detecting Silence and Human Speech	42
7.4	TensorFlow Inference Runtime: Java API/C API	44
7.5	Attempted Optimizations on Model Inference	44
7.5.1	GPU Inference	44
7.5.2	On-Device Training	45
7.6	Voice Command Recognition	46
7.6.1	Text-to-speech	46
7.6.2	Launch Activities and Background Service	49
7.7	Application Overview	50
7.8	Summary	52

8	Experiments and Results	54
8.1	Introduction	54
8.2	Experimental Setup	54
8.3	Testing Environments	55
8.3.1	Quiet Bedroom Overnight	56
8.3.2	Office Day	57
8.3.3	Car Travel	58
8.4	Memory Usage	59
8.5	Energy Consumption Analysis	60
8.6	Summary	62
9	Conclusions and Future Work	63
9.1	Conclusions	63
9.1.1	Algorithms on Android NDK	63
9.1.2	Android Runtime limitations	64
9.1.3	TensorFlow Lite models	64
9.1.4	Android API limitations	64
9.1.5	Notes on Objectives	65
9.2	Research Questions Revisited	66
9.3	Future Work	67
	References	68

List of Figures

5.1	Performance comparison of Kotlin and C implementations for input size 2^n . . .	27
5.2	Time comparison of <code>evenRemoved</code> implementations (log-log scale)	28
6.1	Execution time of Iterative FFT in Kotlin and C	30
6.2	MM performance comparison on C (-O0 Flag) and Kotlin implementations . . .	32
6.3	Performance comparison: C (-O0 Flag) vs Kotlin for MM	33
6.4	Performance comparison: C (-O3 flag) vs Kotlin for Matrix Multiplication	33
7.1	Normalized audio samples recorded from the microphone while a user is speaking.	43
7.2	Memory consumption on CPU inference (left side) and on GPU inference (right side) extracted from the Android Profiler tools.	46
7.3	Data-flow diagram of the real-time wake-word detection pipeline.	51
7.4	Data-flow diagram of the voice command recognition pipeline.	52
8.1	The heatmap activity of the wake word pipeline was recorded over three hours in a living room where the TV was on. During this time, one person said the wake word twice. The heatmap is divided into segments of 30 seconds each (<code>bin_size=30s</code>), allowing for the calculation of relative activity within those intervals.	55
8.2	Pie chart representing the percentage of each audio event in the previous three-hour recording from the living room.	56
8.3	Heatmap displaying the activity of wake word triggers in a bedroom over nine hours overnight.	56
8.4	Pie chart showing the percentage of different audio events during nine hours in a bedroom overnight.	57
8.5	Heatmap displaying the activity of wake word triggers over almost nine hours at the company's office.	57
8.6	Pie chart showing the percentage of different audio events during almost nine hours at the company's office.	58
8.7	Heatmap activity of wake word pipeline triggers.	58
8.8	Pie chart displaying the percentage of different audio events during a car trip. . .	59
8.9	Memory usage when the application is running as a background service.	59
8.10	Chart illustrating linear regression for predicted battery consumption.	61
9.1	Android stack diagram. [Source: Android Open Source Project, https://source.android.com/docs/core/architecture .]	65

List of Tables

2.1	Results for each search query across the selected databases.	5
2.2	Inclusion and exclusion criteria used to filter results from the search queries. . . .	5
3.1	Application with Intensive Processing [Source: Corbalan et al. [5]]	10
5.1	Speed-up of Kotlin over C for evenRemoved implementation	27
6.1	Speed-up of C over Kotlin in Iterative FFT	30
6.2	Speed-up of C-linear (-O3) over Kotlin-array for Matrix Multiplication	32
7.1	Input and output tensor specifications of the Mel-spectrogram model.	37
7.2	Input and output tensor specifications of the Embeddings model.	37
7.3	Input and output tensor specifications of the Classifier model.	37
8.1	Estimated coefficients for the regression.	61

Abbreviations

API	Application Programming Interface
ART	Android Runtime
DNN	Deep Neural Network
FFT	Fast Fourier Transform
JNI	Java Native Interface
MFCCs	Mel-Frequency Cepstral Coefficients
MM	Matrix Multiplication
NDK	Native Development Kit
ODML	On-Device Machine Learning
ONNX	Open Neural Network Exchange
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
TFLite	TensorFlow Lite
VAD	Voice Activity Detection

Chapter 1

Introduction

1.1 Context

In recent years, mobile and wearable devices have evolved from basic communication systems to large-scale, complex and multifunctional devices that improve our quality of life, both personally and professionally. Applications like fitness trackers and mobile banking are perfect examples that reflect the amount and complexity of the data processed by these systems.

Naturally, this evolution raises concerns related to performance, energy efficiency, and the overall user experience. Most mobile applications are developed using higher-level languages like Java and Kotlin that manage some of these aspects. On the other hand, low-level languages like C and C++ provide developers with greater control over system resources, hardware, and power consumption. This control is essential for creating optimized applications for resource-constrained devices.

This research will be conducted in collaboration with Konkconsulting, a consulting company specializing in HR solutions. However, the company has recently expanded its focus to explore mobile development and the potential advantages of leveraging low-level programming to build and optimize applications for mobile and wearable devices.

1.2 Motivation

The primary motivation behind this research is the increasing demand for applications that need to perform fast and complex real-time operations without compromising energy efficiency. As more people are using this kind of devices, they expect them to be fast and responsive while not draining substantial battery life. To tackle this, low-level programming languages offer the potential to meet these expectations, promising significant advantages, such as higher performance in terms of execution time and energy efficiency [25]. By using this kind of programming languages, the goal is to develop algorithms that are known to be more efficient when implemented in this way, such as numerical computation algorithms, cryptography algorithms, and others.

1.3 Problem

Despite the potential benefits of low-level programming, most mobile and wearable applications are primarily developed using high-level languages, which prioritize developer convenience over hardware optimization while also making implementations more device-agnostic. Consequently, developers cannot control how resources are managed during execution, which can lead to inefficiencies, especially for computationally intensive algorithms. These inefficiencies are more critical when resource constraints such as CPU power, memory, and energy are significant bottlenecks. Heavy computation tasks such as cryptography operations, numerical algorithms, and others are often processed on remote servers due to the high computational cost of these operations. Consequently, this can worsen the user experience and lead to privacy problems when transmitting data [37]. Bringing the execution of these algorithms to the local device could potentially offer significant benefits.

Although low-level languages offer better control over system resources, their steep learning curve and higher susceptibility to errors make them less accessible to many developers, creating a gap between the actual performance of applications and the potential performance that could be achieved using low-level languages.

This research aims to bridge the gap by achieving high performance in the algorithms mentioned earlier. Using both the Android SDK (Software Development Kit) [9] and Android NDK (Native Development Kit) [8], the goal is to implement these privacy-preserving algorithms, compare their performance on execution speed, resource utilization, and energy efficiency across both environments. Additionally, the study will explore the feasibility of developing a library or framework that abstracts the complexity associated with low-level programming and simplifies access to these algorithms.

1.4 Research Questions

1. How can low-level programming like C/C++ be used to optimize performance and energy efficiency in mobile and wearable applications compared to high-level programming approaches?
2. Which types of algorithms or operations are best suited for optimization through low-level programming in mobile and wearable devices?
3. What algorithms have been successfully implemented using the Android NDK (Native Development Kit) so far, and in what scenarios are these implementations most effective?
4. What are good practices and design patterns for using the Android NDK when optimizing applications for mobile and wearable devices?

1.5 Contribution

This dissertation addresses the state of the art regarding the use of low-level programming on Android and its impact on the performance of applications, utilizing the Android NDK. Additionally, it explores and compares several algorithms implemented in both C and Kotlin, leading to valuable insights into Android NDK development. A significant contribution of this dissertation is the development of an Android application capable of wake word detection and voice command recognition using on-device machine learning, designed to perform actions such as launching another application solely through voice commands. The empirical evaluation of the wake word detection also provides insights into how various audio activities of real environments relate to energy consumption and memory usage.

1.6 Dissertation Structure

Chapter 1 presents the dissertation topic by establishing the context, outlining the problem, and explaining the motivation for this investigation.

In Chapter 2, we conducted a systematic literature review on topics relevant to this study, analyzing information gathered from various recognized digital libraries.

In Chapter 3, we present the state of the art concerning the development of low-level software for mobile devices, focusing on its performance impacts such as energy efficiency and security, while also examining the use of on-device machine learning.

In Chapter 4, we explore the problem in greater detail and propose a solution involving an application designed for wake word detection and voice command recognition.

Chapter 5 examines the Sieve of Eratosthenes as a proof of concept proposed by the company, comparing performance on Android NDK with low-level languages against Kotlin on Android SDK.

In Chapter 6, we further explore the comparison between Kotlin and C code on more intensive computing algorithms, including square matrix multiplication and the Fast Fourier Transform (FFT).

In Chapter 7, the proposed solution is developed by creating the wake word detector and the voice command recognition service, utilizing the C API of the TensorFlow Lite library.

In Chapter 8, we present the evaluation of our experimental results regarding the behavior of the audio pipeline for wake word detection, while also measuring energy consumption and memory usage.

Chapter 9 concludes this dissertation with final remarks about the algorithms we explored and the application we developed. We also provide answers to the research questions and present potential improvements for future work.

Chapter 2

Literature Review

2.1 Introduction

The purpose of this literature review is to systematically identify, collect, and analyze existing research on low-level code optimizations for mobile and wearable devices. This focus is essential because these optimizations address the specific limitations of resource-constrained platforms, which can enhance performance, extend battery life, and improve the overall user experience.

2.2 Search Strategy

To ensure a comprehensive review of the literature, systematic search queries were conducted across multiple databases, including IEEE Xplore, ACM Digital Library, and Scopus.

The goal of creating these search queries is to gather accurate information that answers the defined research questions. The queries should be broad enough to cover relevant topics related to mobile and wearable devices, algorithm performance, and low-level programming, particularly utilizing the capabilities of the Android NDK. However, they must also be sufficiently focused to align with the specific scope of the research questions.

After reviewing the scope of the research questions, we combined the first and second questions into a single search query, as well as the third and fourth questions into another search query, due to overlapping themes with slightly different intended outcomes.

Search Query 1:

(mobile OR wearable) AND (device OR comput* OR system*) AND "low-level programming"*

Search Query 2:

android AND (ndk OR "native development kit") AND (optimi OR "design pattern" OR algorithm?)*

The first search query investigates the development of applications for mobile and wearable devices using low-level programming languages, placing a special emphasis on performance and applied optimizations. The second search focuses on the Android NDK, exploring how algorithms

are implemented, how code optimizations utilize its tools, and which design patterns and best practices are applied. The results of our quantitative analysis are summarized in Table 2.1.

Additionally, several searches were conducted on Google Scholar to enhance the information gathered from other platforms. All retrieved articles were exported to the Zotero reference manager for further screening and selection based on the inclusion and exclusion criteria defined in Section 2.3.

Table 2.1: Results for each search query across the selected databases.

Database	SQ1	SQ2	Total
IEEE	20	10	30
ACM	362	239	601
Scopus	184	284	468

2.3 Inclusion and Exclusion Criteria

To ensure the relevance of the search results to the research questions, a set of inclusion and exclusion criteria was established, as shown in Table 2.2. These criteria acted as a filtering mechanism to refine the results, focusing only on studies and findings that aligned with the research objectives.

The inclusion criteria were defined to ensure that only studies directly related to the development of mobile and wearable devices were considered. Specifically, results were included if they were applicable to mobile and/or wearable devices (IC1), utilized Android NDK (IC2), or were related to mobile application development (IC3). The goal of these criteria was to maintain a clear focus on the practical application of low-level programming within the Android ecosystem, particularly through the use of Android NDK. By enforcing these conditions, the search results were limited to sources that provided insights into optimizing performance, implementing algorithms, and utilizing best practices in mobile development.

In contrast, the exclusion criteria were designed to eliminate results that were redundant, irrelevant, or outside the research scope. Duplicated results were removed to prevent redundancy and ensure a more diverse set of findings (EC1). Additionally, studies that focused solely on theoretical concepts without direct relevance to implementation were excluded (EC2). Finally, results

Table 2.2: Inclusion and exclusion criteria used to filter results from the search queries.

I/E	ID	Description
Inclusion	IC1	Result is applied to mobile and/or wearable devices
	IC2	Result uses Android NDK
	IC3	Result is related to mobile application development
Exclusion	EC1	Result is duplicated
	EC2	Result is out of the application developer scope
	EC3	Result was published before 2010

published before 2010 were discarded to ensure that only recent advancements and up-to-date methodologies were analyzed (EC3).

By applying these inclusion and exclusion criteria, the filtering process ensured that only the most relevant and high-quality results were considered for further analysis. This approach helped narrow the research focus and provided a clear methodology for selecting studies that offered valuable insights into mobile development, algorithm performance, and low-level programming using the Android NDK.

2.4 Summary

In summary, the literature review process consisted of creating targeted search queries and applying them across various databases. The results were systematically screened using clear inclusion and exclusion criteria. This approach ensured that only studies directly relevant to the dissertation's objectives were included, specifically emphasizing low-level programming for mobile and wearable devices and focusing on the use of Android NDK.

Chapter 3

State of the Art

3.1 Introduction

As mobile and wearable devices (including Wear OS devices) continue to handle increasingly demanding tasks, ranging from real-time multimedia processing to continuous sensor data analysis, developers seek ways to improve application performance, energy efficiency, and security. While some optimizations require kernel or hardware-level access, many studies focus on user-space techniques that remain feasible to standard app developers. Among these, the Android Native Development Kit (NDK) is often cited for its potential to reduce overhead by leveraging C/C++ libraries and direct system calls, bypassing portions of the Dalvik/ART virtual machine.

However, new challenges emerge, such as how to measure energy consumption meaningfully [23, 5], how to ensure security in native libraries [30, 38, 24], and how best to deploy advanced machine learning workloads [12, 19, 32].

3.2 Performance Improvements Through Native Code

3.2.1 Speedups Enabled by the NDK

One of the earliest studies on NDK performance is by Ki-Cheol Son and Lee [17], who demonstrated that an augmented reality toolkit (NyARToolKit) could be sped up by nearly 1.87x after porting certain functionalities from Java to C via the NDK. Their results underline a premise of native development: compute-intensive operations, such as image processing or object tracking, often see measurable gains in C/C++ compared to Java.

More recent work by Ponomarev [26] reinforces that bypassing layers of the Android runtime can improve execution times, if developers also profile their code and use optimizations like SIMD instructions. However, these gains are not strictly related to the application workload. Therefore, native C/C++ may not always outperform well-tuned Java/Kotlin.

3.2.2 Hardware-Accelerated Use Cases

In addition to generic speedups, hardware acceleration is a major incentive for low-level approaches. Bloessl et al. [3] showed how real-time stream data processing with GNU Radio can leverage Single Instruction Multiple Data (SIMD) instructions and GPU-accelerated libraries on Android. Although their solution requires deeper low-level skills, they achieved performance unattainable with Java-based APIs alone.

3.3 Android NDK and Its Core Components

Although most Android apps are developed in Java or Kotlin, the NDK offers the option to write performance-critical sections in C/C++. This duality provides flexibility but adds complexity. Beyond general performance gains, the Android NDK provides specific tools, APIs, and libraries that allow developers to work closer to the hardware. Some research suggests the NDK is most effective for tasks like physics simulations or image processing, such as NyARToolKit [17].

3.3.1 Key Features and Limitations

The NDK grants access to lower-level features like sensors or advanced graphics libraries, and supports high-performance tasks such as augmented reality. However, there are trade-offs:

- *Overhead vs. Performance:* Although native code can avoid some Java Virtual Machine overhead, not all workloads see uniform benefits.
- *Complexity:* Managing cross-language calls (JNI) and debugging can increase the development workload.

3.3.2 Core Interfaces, APIs, and Low-Level Functionality

The NDK exposes a stable subset of POSIX functions (file I/O, sockets, pthreads) plus Android-specific extensions:

- *libc* (bionic), *libm*, *libdl*, *liblog*, *libandroid*
- OpenGL ES and Vulkan (headers and libraries for graphics and compute)

Developers can implement a full Activity in C/C++ via NativeActivity and NativeAppGlue, managing app lifecycle and input without Java code.

The NDK's *libc* (bionic) supports a range of POSIX calls (open, read, write, sockets). Certain non-public or unsupported calls are excluded. The NDK also offers pthreads, mutexes, and condition variables, plus memory allocation (malloc, calloc, free) and mapping (mmap). Developers should profile carefully to ensure they actually benefit from these low-level options. Additionally, BSD sockets are available for TCP/UDP, along with other POSIX-level functions (DNS resolution, HTTP, TLS) or third-party libraries like cURL and OpenSSL.

3.3.3 Libraries and Services

The NDK includes a subset of OpenSL ES for low-latency audio, as well as APIs such as AMediaCodec, AMediaExtractor, and AMediaFormat for hardware-accelerated media handling. Additionally, NDK headers provide native access to NNAPI, allowing neural network acceleration from C/C++ code without the need to use Java. However, it is important to note that NNAPI was deprecated¹ in Android 15. While NNAPI can still be used, it is not recommended for performance-critical applications. Instead, Google has developed TensorFlow Lite, which is available in Play Services [11], to better support demanding tasks.

3.3.4 Sensors and Hardware Access

While many hardware features are wrapped in Java APIs, the NDK offers ASensorManager and ASensorEventQueue for reading sensor data natively. Some capabilities (e.g., camera or fingerprint) still require Java, but can be bridged via JNI.

3.3.5 JNI Integration

Java Native Interface (JNI) is the bridge between native C/C++ code and the Android framework. It can call Java methods (e.g., SensorManager, CameraManager) that are not directly exposed through the NDK. Though powerful, JNI requires careful consideration to minimize overhead and avoid memory leaks.

3.4 Energy Consumption Considerations

3.4.1 Cross-Language Comparisons

Several comparative studies explore energy efficiency in different Android development approaches (Java vs. JavaScript vs. C/C++). Oliveira et al. [23] noted that JavaScript was more energy efficient in some cases, whereas Java and C++ excelled in raw speed. Significantly, they found that hybrid solutions—mixing Java with either JavaScript or C++—could reduce energy consumption by as much as 100x compared to a pure Java solution.

Corbalan et al. [5] also compared energy usage across multiple development frameworks, including the Android SDK and NDK. The study examines energy consumption, CPU performance, and execution time. A significant difference between these two development frameworks is evident, especially in applications requiring intensive processing, where Corbalan et al. implemented an algorithm to calculate the following series:

$$series = \sum_{j=1}^5 \sum_{k=1}^{100000} \left(\log_2(k) + \frac{3k}{2^j} + \sqrt{k} + k^{j-1} \right) \quad (1)$$

¹<https://developer.android.com/ndk/guides/neuralnetworks/migration-guide>

The series in Equation 1 is part of their methodology for quantifying the impact of CPU load on energy consumption in mobile applications. Additionally, the data presented in Table 3.1 was extracted from the study by Corbalan et al. [5], which compares energy consumption, CPU load, and execution duration between the Android SDK and NDK for the application that involves intensive processing.

Table 3.1: Application with Intensive Processing [Source: Corbalan et al. [5]]

Framework	Energy (mWh.) $\bar{E}$	CPU load (%) $\bar{C}$	Duration (s.) $\bar{T}$
Android NDK	1.789	32.434	9.745
Android SDK	3.463	32.468	18.568

The Android SDK, with Java/kotlin, demonstrates a higher energy consumption than the Android NDK, which utilizes C/C++ for processing. This is due to the efficient way NDK handles computational tasks, leveraging lower-level, platform-specific optimizations that reduce execution time and CPU load. In contrast, while applications developed using Java/kotlin are more straightforward, they are limited by the constraints of the Android Runtime virtual machine, leading to higher energy consumption. In this context, the NDK proves far more suitable for CPU-intensive tasks such as those requiring complex mathematical computations or real-time data processing.

3.4.2 Developer Practices for Energy Optimization

Li and Halfond [20] investigated various energy-saving programming practices for Android apps, concentrating on optimizing network requests, memory usage, and computational processes. They discovered that bundling smaller network packets into larger ones significantly reduced energy consumption. Other practices, such as optimizing memory usage and minimizing unnecessary method invocations, also resulted in measurable power savings. However, some methods, like reducing memory usage, had minimal effects on overall energy consumption. These findings emphasize the importance of targeting specific areas, such as network communication and computational efficiency, to achieve substantial energy savings.

Although their research focuses mainly on Java-based Android development, many of their principles also apply to C/C++ development using the Android Native Development Kit (NDK). For instance, avoiding repeated function calls within tight loops and efficiently managing data structures can significantly improve energy efficiency in NDK applications.

3.4.3 Power Consumption Analysis Tools

Ponomarev [26] emphasizes the importance of profiling tools like Android Profiler and Systrace in confirming whether perceived performance improvements result in actual reductions in power consumption. These tools enable developers to monitor key performance indicators, including CPU load, memory usage, and network performance. This monitoring is crucial for ensuring that optimizations not only enhance speed but also reduce energy usage. While the Power Profiler

tool in Android Studio Profiler displays the device's power consumption, not all power data is accessible on every device. Power measurements are obtained through the On Device Power Rails Monitor (ODPM)². However, this tool is only available on Google devices, specifically the Pixel 6 and later models running on API level 29 or higher. Consequently, on devices that do not support ODPM, only battery power and current measurements are accessible.

GreenDroid, introduced by Couto et al. [6], is a tool designed to monitor and analyze power consumption in Android applications. By instrumenting the source code and utilizing a power consumption model, GreenDroid estimates the energy consumed by various application components during execution. Its unique feature is the integration of a power consumption model, which allows it to link energy usage to specific parts of the application's source code. Through this model, developers can identify and address inefficiencies at the code level. The primary limitation of GreenDroid is its requirement for source code instrumentation, which may not always be practical or desirable for every application.

In contrast, PowDroid, presented by Bouaffar et al. [4], focuses on profiling the energy consumption of Android applications without requiring access to their source code. Unlike GreenDroid, PowDroid functions as a black-box tool, making it suitable for applications where source code instrumentation is not feasible. By utilizing system-wide metrics collected through tools like Batterystats and Battery Historian, PowDroid calculates energy consumption in joules by analyzing battery drain and hardware component usage during app execution. This approach provides a practical and scalable solution for developers seeking to benchmark and profile the energy efficiency of applications based on their actual runtime behavior.

3.5 Security Implications of Native Code

3.5.1 Potential Risks in JNI/NDK

Although the NDK can enhance performance, Ruggia et al. [30] warn of the risks posed by malicious apps embedding harmful logic in C/C++ libraries. Such tactics are not always detected by static analyzers that mostly target Dalvik/ART bytecode. Similarly, Wei et al. [38] propose a JNI-aware static analysis framework (JN-SAF), showing that generic app scanners may overlook or oversimplify native paths, resulting in missed vulnerabilities. Later work by Park et al. [24] refines these analyses using binary decompilation, reinforcing that security in an NDK context requires specialized methods.

3.5.2 Trusted Execution Environments and Secure ML

As machine learning gains momentum on mobile devices, Hu et al. [12] propose using Trusted Execution Environments (TEEs) to secure entire DNN models. Though not strictly about the

²<https://developer.android.com/studio/profile/power-profiler>

NDK, TEEs often use low-level code to deliver real-time performance and ensure data confidentiality. This approach follows the principle that mixing trusted and untrusted code can introduce new attack surfaces unless carefully designed [38].

3.6 Emerging Use Cases: Machine Learning and IoT

3.6.1 Inference Frameworks and NNAPI

Febvay [7] presented a comprehensive performance comparison of several open-source deep learning inference frameworks tailored for mobile and embedded devices. The findings highlight that low-level software optimizations, such as the use of specialized kernels, memory access strategies, and hardware-specific acceleration, are crucial for determining overall inference performance. The study shows that even when using the same model architecture, differences in how frameworks manage low-level operations can lead to significant variations in both speed and predictive accuracy.

Lai et al. [19] demonstrated the integration of the Android NNAPI with the TVM runtime. Although they successfully performed this integration, the resulting performance metrics were somewhat disappointing, likely due to limitations within NNAPI. They had anticipated that these issues would be resolved with the ongoing development of NNAPI. However, it is now known that NNAPI has been deprecated in Android 15, and there are more effective alternative solutions available, such as TensorFlow Lite, as mentioned by Febvay [7].

3.6.2 On-Device Speech Recognition

Automatic speech recognition (ASR) on edge devices or locally is appealing for tasks that require low latency, offline accessibility, and robust privacy protections. Instead of sending raw audio data to the cloud, a complete wake-word detection and speech-to-text process is performed directly on the device itself.

In this context, Sharma and Bhalley [33] demonstrated that a complete speech-to-text pipeline can operate locally on standard mobile hardware using PocketSphinx [29]. By employing a lightweight acoustic model, a compact language model, and an automatically generated pronunciation lexicon, their system successfully recognized continuous speech in real time without needing a network connection. This proves that on-device ASR is not only feasible but also privacy-preserving. Subsequent studies have applied this same approach to various tasks, including smart-home voice control and offline dictation, particularly in situations with limited connectivity.

3.6.2.1 Wake Word Detection

Recent advancements have led to the development of several compact engines that can perform the wake-word or keyword-spotting stage entirely on the edge. One such engine is Picovoice Porcupine [15], which operates with a small memory footprint and supports both standard and custom triggers. This makes it a popular choice for resource-constrained smart speakers and wearable

devices. However, it is essential to note that Picovoice Porcupine requires an API access key to execute its machine learning models, which Picovoice servers must verify. This limitation means that users need an internet connection, even though the model runs locally.

OpenWakeWord (OWW) [1] is a fully open alternative that features an Apache-2.0 implementation, which includes pre-trained ONNX and TensorFlow Lite models, as well as an easy-to-use command-line interface (CLI). This allows developers to quickly create new wake phrases without relying on cloud resources. However, a substantial amount of training data is necessary to achieve performance and accuracy comparable to that of the pre-trained models, as OWW models are trained on thousands of hours of audio. Furthermore, a significant limitation is that the library is exclusively written in Python. As a result, integrating it into native Android applications requires workarounds and is not supported out of the box.

3.6.2.2 Speech-to-Text Tools

Once activated using wake-word models, the device must convert speech to text locally. Several modern engines can achieve this quality that is comparable to cloud APIs. Picovoice Leopard [14] performs complete automatic speech recognition (ASR) on CPUs as small as a Raspberry Pi while achieving word error rates around 9.7% and offering features such as diarization and timestamps. On the other hand, Whisper [27] from OpenAI offers high multilingual accuracy using large Transformer models that can be optimized for real-time GPU inference, along with lightweight versions for low-powered devices. For mobile-class CPUs, the Kaldi-derived Vosk [13] decoder provides over 20 offline language packs, each around 50 MB in size. Meanwhile, PocketSphinx continues to focus on the smallest memory footprints, requiring about 10 MB, specifically for command-and-control grammars.

These toolkits make end-to-end speech recognition practical on-device while respecting user privacy and functioning in bandwidth-limited environments.

3.6.3 IoT Devices and Hybrid Frameworks

Most references focus on smartphones, but Sayeedi et al. [32] consider ML deployment on IoT devices, comparing high-level and low-level libraries. Their study indicates that selecting lower-level libraries, such as libtorch, can lead to significant speed improvements, though it might also complicate code management. This aligns with Oliveira et al. [23], who concluded that hybrid solutions could be a good compromise between development efficiency and performance.

3.7 Summary

In general, these studies point to three main conclusions: NDK-based optimizations can deliver speedups for compute-heavy tasks [17, 26, 3], although energy savings are not guaranteed, with several authors [23, 5, 20] urging holistic profiling and caution against assuming lower-level code is inherently more efficient. At the same time, native code introduces security trade-offs [30, 24],

and advanced ML or IoT use cases [12, 19, 32] highlight the need for specialized integration and optimization.

Chapter 4

Proposed Solution

4.1 Introduction and Objectives

The goal of this chapter is to present the approach for the development of a local voice recognition system capable of detecting a specific wake word and handling a small set of voice commands. This approach aims to reduce latency, safeguard user privacy, and mitigate dependence on cloud-based services. By implementing voice recognition on-device, it becomes possible to offer a responsive and secure user experience, particularly in scenarios where internet connectivity is unreliable or user data sensitivity is a concern.

There are two main objectives in this approach. First, it seeks to identify an optimal strategy to recognize a wake word with high accuracy and minimal false positives under diverse conditions. Second, it aims to enable users to issue brief voice commands, such as speed dialing contacts or opening specific applications, without requiring continuous server interaction. Because smart-watches and other mobile devices often have constrained resources, low-level techniques will be applied to maintain a low computational footprint while still delivering real-time performance.

4.2 Proposed Approach

The proposed approach for local voice recognition is centered on the wake word detection and the recognition of a limited set of commands. Both functionalities rely on lightweight machine learning models and native optimizations using Android NDK to ensure low power consumption and near-instantaneous responses.

4.2.1 Wake Word Detection

Wake word detection constitutes the first layer of the voice recognition pipeline. In this layer, the system is continuously monitoring incoming audio but operates in a low-power mode to preserve battery life. Whenever a user emits a sound that is the designated wake word, the system transitions into a higher-power state to capture and process the speech that follows.

The implementation of this detection typically involves segmenting the audio into short frames and converting them into feature representations such as Mel-Spectrograms or Mel-Frequency Cepstral Coefficients (MFCC), which are widely used for modeling speech signals thanks to their ability to approximate the human auditory system's perception of sound [2]. These MFCC features are subsequently input into a neural network model, typically a compact convolutional or recurrent architecture, which classifies each audio frame as either containing the wake word or not. By leveraging the Android NDK, the system can offload computationally heavy tasks to efficient C/C++ libraries, thereby reducing inference time and ensuring a smooth user experience.

4.2.2 Limited Voice Command Recognition

Once the system recognizes the wake word, it enters a more focused recognition mode, expecting the user to issue one of several predefined commands. These commands may include functions like speed dialing a contact, starting music playback, or opening a specific application. The system can utilize a constrained and lightweight neural network model for classification because the vocabulary is limited to a short set of command phrases. To accommodate user variations in accent, pitch, and common background noises, an optional personalization feature can be integrated. In this feature, the user can record several samples for each command, allowing the model to adapt to individual speech patterns. This adaptation can be performed directly on the device, ensuring user privacy by avoiding the need to send recordings to external servers.

4.3 Experimental Process

To validate the overall effectiveness of the proposed system, it is crucial to carefully plan the experimental process and evaluation protocol. This process begins by defining various testing environments that incorporate a diverse mix of audio types, such as voice, music, and random ambient noises, while also varying the audio volume and duration. The audio will be recorded on a testing device, and we will collect data to characterize the audio mix and measure energy consumption during the experiment.

In terms of the models used for wake word and speech recognition, we will evaluate them based on metrics such as accuracy, recall, and precision. A key metric in this phase is the false-positive rate, especially for wake word detection. A high false-positive rate could lead to unnecessary power consumption and user frustration by repeatedly activating the system when the wake word was not actually spoken.

4.4 Expected Results and Relevance

It is expected that a properly tuned wake word detection model will offer high accuracy, activating the system only when triggered by the correct sounds and minimizing unintended activations. For

the limited set of voice commands, a high accuracy performance is expected, especially in favorable acoustic environments. The personalization feature, if implemented, should boost accuracy by enabling the model to learn the subtle characteristics of each user's speech patterns.

In practical terms, this approach can significantly improve the user experience by removing dependence on remote servers. By implementing local inference, latency is minimized, security is enhanced since audio data can remain entirely on the user's device, and battery usage is kept within acceptable limits for the device. Ultimately, providing voice recognition with minimal resource requirements can promise more natural, hands-free interactions, especially for users who regularly wear smartwatches or rely on smartphones in situations where screen interaction is inconvenient or unsafe, such as while driving.

Chapter 5

Proof of Concept: Sieve of Eratosthenes

5.1 Introduction

In this chapter, we aim to compare the performance between low-level and high-level implementations of the Sieve of Eratosthenes running on mobile devices. This classical algorithm is well known for finding prime numbers up to a given limit n . The comparison serves as a foundation for evaluating the performance benefits of low-level development compared to practical, modern, high-level programming language approaches.

Our analysis focuses on achieving a good implementation, not only to reduce the execution time of the algorithm for a given n but also to manage the impact on memory consumption. To address these topics, we will explore feasible parallelization techniques in both high-level and low-level approaches. Additionally, we will investigate memory-efficient data structures for storing the array of prime numbers. Furthermore, to maximize performance in the low-level implementation, we will conduct experiments with various Clang compiler flags.

5.2 Implementation Details

The Sieve of Eratosthenes algorithm will be implemented on mobile devices. For this purpose, Kotlin was the programming language chosen for the high-level approach, while C/C++ will be used for the low-level approach, alongside the JNI (Java Native Interface) tools, utilizing the Android NDK (Native Development Kit). Kotlin is a modern programming language that is widely used for Android applications; C and C++ are old but very popular programming languages, faster than most high-level languages, such as Java, on which the Kotlin language is based, as Pereira et al. demonstrated in [25]. In the low-level implementation, we consider testing both C and C++ languages to determine if using the C++ Standard Library has any impact on the execution time and memory consumption of this algorithm when compared to pure C code.

Algorithm 1: Sieve of Eratosthenes Pseudo-code

Input: An integer $n > 1$
 Create array $isPrime[2 \dots n]$, initialized to true;
 Set $isPrime[0] \leftarrow false, isPrime[1] \leftarrow false$;
for $i \leftarrow 2$ **to** $\sqrt{n}$ **do**
 if $isPrime[i]$ **then**
 for $j \leftarrow i^2, i^2 + i, i^2 + 2i, \dots, n$ **do**
 $isPrime[j] \leftarrow false$;
 end
end
end
Output: All indexes i such that $isPrime[i]$ is true

Low- and high-level approaches were implemented as suggested in the pseudo Algorithm 1; this was considered the `default` code version. To enhance the overall performance of the algorithm, we first modified the `default` version by applying parallelization on the outer loop of the algorithm, resulting in the `parallel` version. Next, we eliminated all even numbers from the initial array, `isPrime`, since none, except for the number two, are prime numbers. This modification led to the `evenRemoved` version. Finally, we developed the `bitArray` version, which introduces a data structure that is essentially an array of bits, reducing memory consumption. All changes were cumulative, meaning that each later version included all the previous modifications.

Regarding the input parameters of the algorithm, we had to consider that in Kotlin, the maximum number of elements an array can take is `INT_MAX-1`. As this limit depends on the JVM implementation, to reproduce the algorithm, the maximum value we can use for n is $2^{31} - 1$, since the `isPrime` array has n elements. However, in C/C++, we can use a larger number of elements, as the first limitation is the available memory, and the second one would be for $n \geq 2^{64}$ if we had enough memory.

The output of this algorithm provides two main insights: we can determine how many prime numbers exist in the interval $[1, n]$, and we can identify the prime numbers based on the indices of the `isPrime` array. We opted against the first approach due to performance limitations when implementing it on the Android NDK side. In the C/C++ algorithm, we use data structures that are incomprehensible to the JVM, so to return the whole array of prime numbers, we would need to make multiple JNI calls to create a valid object for Kotlin. However, these calls are very time-consuming and do not reflect the complexity of the algorithm for finding prime numbers. For those reasons, we adopted the first approach, which only requires returning the size of that array without any additional JNI calls.

5.2.1 Kotlin Implementation (SDK)

The Kotlin implementation of the Sieve of Eratosthenes followed the pseudo-code presented in Algorithm 1, changing only the returned value as explained at the end of Section 5.2. The Listing 5.1 presents the default implementation in Kotlin, where we used the `BooleanArray` class to create the `isPrime` data structure with the boolean flags. The elements of this array are of Boolean type, which uses one byte of memory. Therefore, executing the function with a parameter n will allocate n bytes for the array elements, in addition to the memory required for the array structure itself, such as the header and length information.

```
1  suspend fun sieveKotlin(n: Int): Int = withContext(Dispatchers.Default) {
2      val isPrime = BooleanArray(n + 1) { true }
3
4      if (n >= 0) isPrime[0] = false
5      if (n >= 1) isPrime[1] = false
6
7      val limit = sqrt(n.toDouble()).toInt()
8
9      for (i in 2..limit) {
10         if (isPrime[i]) {
11             for (j in i * i..n step i) {
12                 isPrime[j] = false
13             }
14         }
15     }
16
17     return@withContext isPrime.count { it }
18 }
```

Listing 5.1: Kotlin default implementation

We must also be aware of concerns related to Android threads and the execution of programs within them. A Main thread, also known as the UI Thread, is responsible for managing everything on the user interface. As it manages the user interface, it cannot be blocked by long-running tasks. If this occurs, the system will terminate the process to prevent the application from blocking the device and present an "application not responding" (ANR) dialog. Therefore, to execute an algorithm like the Sieve of Eratosthenes, we need to use Kotlin Coroutines as suggested in [10], which allows us to run asynchronous code on threads other than the Main thread. When executing a coroutine, we can define a context that determines the thread or threads used for its execution.

In this case, we have chosen the `Dispatchers.Default` context, as indicated in line 1 of Listing 5.1. This context is recommended for performing long-running tasks in the background of an application. It creates a maximum number of threads equal to the number of CPU cores, which is particularly useful for the parallel version in Kotlin.

5.2.1.1 Parallelization using Kotlin Coroutines

In the `parallel` version of the Kotlin implementation, we continue to use the Kotlin coroutines. However, this time, we need to launch multiple coroutines to parallelize the iterations of the Sieve algorithm. This is achieved by creating a `launch` code block within a context of the type `Dispatchers.Default`, as presented in the Listing 5.2. This way, the coroutines are launched on the default threads, allowing the workload to be distributed across the available CPU cores.

```
1  suspend fun sieveParallelKotlin(n: Int): Int = withContext(Dispatchers.  
    Default) {  
2      val isPrime = BooleanArray(n + 1) { true }  
3  
4      if (n >= 0) isPrime[0] = false  
5      if (n >= 1) isPrime[1] = false  
6  
7      val limit = sqrt(n.toDouble()).toInt()  
8  
9      val jobs = (2..limit).map { i ->  
10         launch(Dispatchers.Default) {  
11             if (isPrime[i]) {  
12                 for (j in i * i..n step i) {  
13                     isPrime[j] = false  
14                 }  
15             }  
16         }  
17     }  
18  
19     jobs.forEach { it.join() }  
20  
21     val primesCount = isPrime.count { it }  
22  
23     return@withContext primesCount  
24 }
```

Listing 5.2: Kotlin parallel implementation

5.2.1.2 Removal of even numbers

To reduce memory consumption, we modified the previous algorithm by eliminating even numbers from the Boolean array since none of these can be prime numbers. This change also required us to adjust how we access and assign values in the array.

5.2.1.3 Introducing a Bit Array

Initially, the objective was to implement this approach using the `BitSet` structure built in Kotlin. However, we rapidly discovered that this data structure was not thread-safe, so we could not par-

allelize it without further changes. However, after we implemented manual synchronization with a simple lock mechanism, a thread would block the entire array, creating a lot of overhead, which would not make this a viable solution. Unfortunately, Kotlin does not provide thread-safe data structures for storing an array of bits. This limitation led to an investigation to find an open-source library that implements such a thread-safe data structure with a fine-grained locking mechanism to test the viability of this approach.

We discovered two different implementations of a thread-safe `BitSet`. However, we encountered several issues during integration into our algorithm. In the first version [16], we faced problems when executing the algorithm with different values for n . Regardless of their size, certain values caused memory allocations to exceed the limit, while others functioned as expected. This inconsistent behavior led us to conclude that the implementation of `BitSet` was flawed, so we moved on to the second implementation.

In the second version [18], we encountered a compatibility issue related to the supported Android API level. The implementation of `ConcurrentBitSet` uses the `VarHandle` Java class with a byte array, which generates bytecode that is considered unsafe on ART versions with more restrictive bytecode verification. This verification is enforced by the ART, and on our testing device, the bytecode verifier does not allow the use of the `VarHandle` class with a byte array. Android devices or emulators that operate on newer API levels with a more tolerant bytecode verifier should be compatible with this implementation. Consequently, we could only test it on compatible emulators, as this ART feature cannot be changed on production devices. Fortunately, we gained access to a borrowed device running API level 34, where the bytecode verifier did not flag the previous code as unsafe, thus allowing us to test the `bitArray` algorithm. However, the execution time results were much worse than those of the previous `evenRemoved` version in both Kotlin and C implementations. Therefore, we concluded that it was not worth attempting to work around this compatibility issue, and we did not pursue further evaluation of this approach.

5.2.2 C/C++ Implementation (NDK)

The low-level implementation of the Sieve of Eratosthenes follows the same principles as the Kotlin version, but this time, we use long variable sizes instead of integers. This implementation allows us to test values for n that exceed the `MAX_INT` limit. As mentioned previously, we wanted to verify whether there were any performance improvements when using the STL library over the pure C code, and we implemented the next changes in both C and C++, taking advantage of structures from the `vector` and `algorithm` libraries. However, the execution times were only lower in the `default` version when using vectors instead of dynamic allocations in C; other cases showed worse performance results. Therefore, we decided to focus more on the C implementations.

Regarding memory consumption, in this implementation, each element of the array is represented by a `char`, which occupies one byte, the same amount of memory as a `Boolean` in Kotlin.

```

1  extern "C"
2  JNIEXPORT jlong JNICALL
3  Java_com_example_mytestapp_SieveCKt_sieveC (
4      JNIEnv *env,
5      jclass clazz,
6      jlong n ){
7
8      char* isPrime = (char*)malloc((n + 1) * sizeof(char));
9      if (isPrime == NULL) return NULL;
10
11     for (long long i = 0; i <= n; i++) {
12         isPrime[i] = true;
13     }
14
15     if (n >= 0) isPrime[0] = false;
16     if (n >= 1) isPrime[1] = false;
17
18     unsigned long long limit = sqrt((double)n);
19
20     for (unsigned long long i = 2; i <= limit; i++) {
21         if (isPrime[i]) {
22             for (unsigned long long j = i * i; j <= n; j += i) {
23                 isPrime[j] = false;
24             }
25         }
26     }
27
28     long long primesCount = 0;
29     for (unsigned long long i = 2; i <= n; i++) {
30         if (isPrime[i]) {
31             primesCount++;
32         }
33     }
34
35     free(isPrime);
36
37     return primesCount;
38 }

```

Listing 5.3: C default implementation

5.2.2.1 Parallelization with OpenMP

To perform this low-level parallelization, we used the OpenMP library. Firstly, we added the directive `#pragma omp parallel for` in the outer loop in line 11 of the Listing 5.3 to parallelize each of the iterations through n . The compiler divided the interval of n numbers among the available threads, which, in this case, matched the number of cores. However, this approach statically

assigned chunks of n to the threads, resulting in an imbalanced workload. Consequently, some threads finished their processing earlier than others, while the remaining work could have been more evenly distributed among all available threads. Therefore, to correct this behavior, we updated the directive to `#pragma omp parallel for schedule(dynamic)`. This approach enables chunks to be assigned to active threads, allowing them to receive new chunks of work as they complete their current tasks until all work is finished.

In line 29 of Listing 5.3, we introduced the OMP directive `#pragma omp parallel for reduction(+:primeCount)` to parallelize the counting of prime numbers in the array. However, this last directive did not significantly impact the execution time.

5.2.2.2 Removal of even numbers

Similar to the Kotlin version, we removed all even numbers from the initial array to reduce the memory consumed in this implementation.

5.2.2.3 Bit Array Approach

In the C programming language, bit sets or arrays are not also available in built-in libraries, so we used an implementation proposed by Silva et al. here [34]. This bit array structure was introduced in C, and the execution times were measured and verified that it was around 2x slower than the `evenRemoved` version in C because of the assumed overhead of bit manipulation.

5.3 Experimental Setup

In this section, we will describe the experimental setup and testing environment used to evaluate the different versions of the algorithms. We will also provide details about the Clang compiler flags used for compiling C code and the JVM arguments utilized when building the Android application that executes the algorithms. Additionally, we will present other Android flags included in the `AndroidManifest.xml` file, along with several configurations that were tested to assess their effect on performance metrics.

To test the algorithms described in Section 5.2, we developed an application using Android Studio IDE. We started by creating a Native C++ project that used Kotlin as the primary programming language, targeting a minimum SDK level of API 30 ("R", Android 11.0). We retained the default build configuration language, which is Kotlin DSL, and in the Native C++ section, we left the C++ Standard as the Toolchain Default. To ensure the accuracy of the measured times, we used the Active Build Variant in `release` mode, and for the Active ABI (Application Binary Interface), we selected `arm64-v8a`, as this is the recommended ABI for 64-bit ARM CPUs.

In our tests, we executed each algorithm five times and measured the average execution time, following three warm-up iterations before the actual runs. These warm-up iterations are recommended when evaluating performance metrics, as they help eliminate the first-run overhead, populate memory and CPU caches, and allow the Android Runtime (ART) to run the JIT compiler and optimize the code for subsequent runs.

In order to validate the accuracy of the prime number count, we conducted a cross-verification of the results returned by each algorithm against a reference value corresponding to the specified parameter n .

5.3.1 Test Cases and Measurement

In the test environment, we created multiple test cases using powers of two as the n parameter, ranging from 2^{24} to 2^{32} . However, we could not test every value across all algorithm variations. For instance, in Kotlin, the maximum array size is limited by `INT_MAX-1` value since arrays are indexed by integers. In practice, however, we were unable to reach this limit because, for n values higher than 2^{28} , Kotlin algorithms triggered an `OutOfMemory` exception. This occurs because the ART (Android Runtime) memory allocation is near its limit, and the Android garbage collector does not free enough memory for subsequent test case executions. We explore this issue in more detail in Section 5.4.

In regard to measurement methodology, the execution time for each algorithm was assessed using the `measureTime` function in Kotlin. This function returns a `Duration` object that can be interpreted in various time units. For our analysis, we opted for milliseconds. However, alternative units such as seconds and nanoseconds are also available.

5.3.2 Device Specifications

All tests that this study is based on were conducted using the same physical device. The specifications of that device are described below.

- **Device Name:** Realme X2 Pro
- **OS:** Android 11, API 30
- **Chipset:** Qualcomm SM8150 Snapdragon 855+ (7 nm)
- **CPU:** Octa-core (1x2.96 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.8 GHz Kryo 485) (4 cores based on Cortex-A76, 4 cores based on Cortex-A55 ¹)
- **GPU:** Adreno 640 (700 MHz)
- **RAM:** 8GB
- **Internal Memory:** 128GB

¹https://en.wikichip.org/wiki/qualcomm/snapdragon_800/855_plus

5.3.3 Clang Compiler

For the configuration of the Clang compiler, we need to include two commands in the default `CMakeLists.txt` file. These commands are essential for enabling OpenMP parallelization and targeting a specific ARM architecture.

```
1 find_package(OpenMP REQUIRED)
2 target_compile_options(${CMAKE_PROJECT_NAME} PRIVATE -fopenmp -march=armv8-a)
```

Listing 5.4: Clang compiler flags added to `CMakeLists.txt` file.

These commands add the necessary compiler flags for OpenMP and specify the target ARM architecture as `armv8-a`.

5.3.4 Android Flags

In the project file `AndroidManifest.xml`, we defined two Android flags as follows:

```
1 android:hardwareAccelerated="true"
2 android:largeHeap="true"
```

Listing 5.5: Android flags added to `AndroidManifest.xml` file.

The first flag enables hardware acceleration for graphics and drawing operations. This feature allows the application to leverage the device's GPU capabilities, reducing the CPU workload for these tasks.

Setting the `android:largeHeap="true"` flag signals to the Android system that the application requires a larger heap size. This can mitigate the risk of `OutOfMemoryError` exceptions, particularly during algorithm executions with large input sizes n . It is important to note that the heap size granted is device-dependent. On our specific device, we observed the default heap growth limit to be approximately 192MiB, which increased to around 576MiB when this flag was enabled. However, these values may vary across different devices.

5.4 Results and Analysis

The cumulative modifications to both versions of the Sieve resulted in a gradual performance improvement for each programming language. However, there was an exception when the concept of bit array was introduced in both low- and high-level approaches. After testing algorithms that utilized the bit array, we observed that the execution time increased massively (around 2x slower) even though memory consumption decreased. Even though resource management is a significant concern on mobile devices, we prioritize the response time and latency of algorithms in this context. Therefore, we do not consider this approach to be viable.

By examining the various C algorithm versions on the left side of Figure 5.1, it is evident that the implementation yielding the shortest execution time was the `evenRemoved` version. This implementation is also parallelized and uses only half the memory required by both the `default` and `parallel` versions. In Kotlin, the best-performing algorithm in terms of execution time was also the `evenRemoved` version.

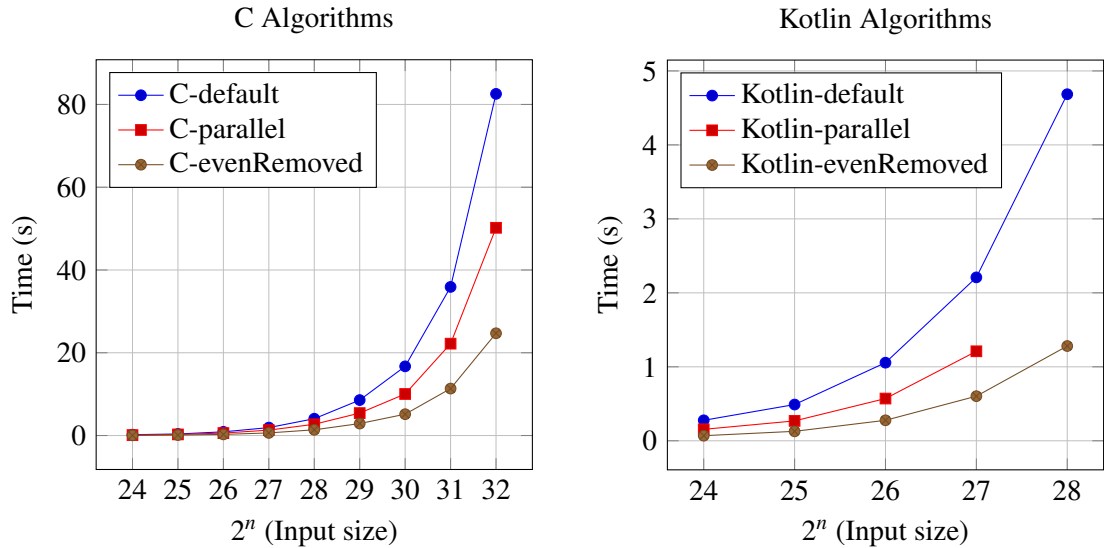


Figure 5.1: Performance comparison of Kotlin and C implementations for input size 2^n

It's important to note that for the Kotlin algorithms in Figure 5.1, we only achieved results up to an input size of 2^{28} . Due to memory consumption limitations in the Android Runtime, this was the largest input size we could handle. Beyond this threshold, we encountered `OutOfMemory` exceptions as the Android virtual machine was unable to allocate enough memory to accommodate the array of prime numbers.

Table 5.1: Speed-up of Kotlin over C for `evenRemoved` implementation

2^n (Input size)	Kotlin (s)	C (s)	Speed-up (Kotlin / C)
24	0.069	0.048	1.44×
25	0.128	0.120	1.07×
26	0.277	0.276	1.00×
27	0.603	0.634	0.95×
28	1.281	1.387	0.92×

In Table 5.1, the speedup of execution time between Kotlin and C implementations of the `evenRemoved` version is approximately 1 for every input value tested in both programming languages.

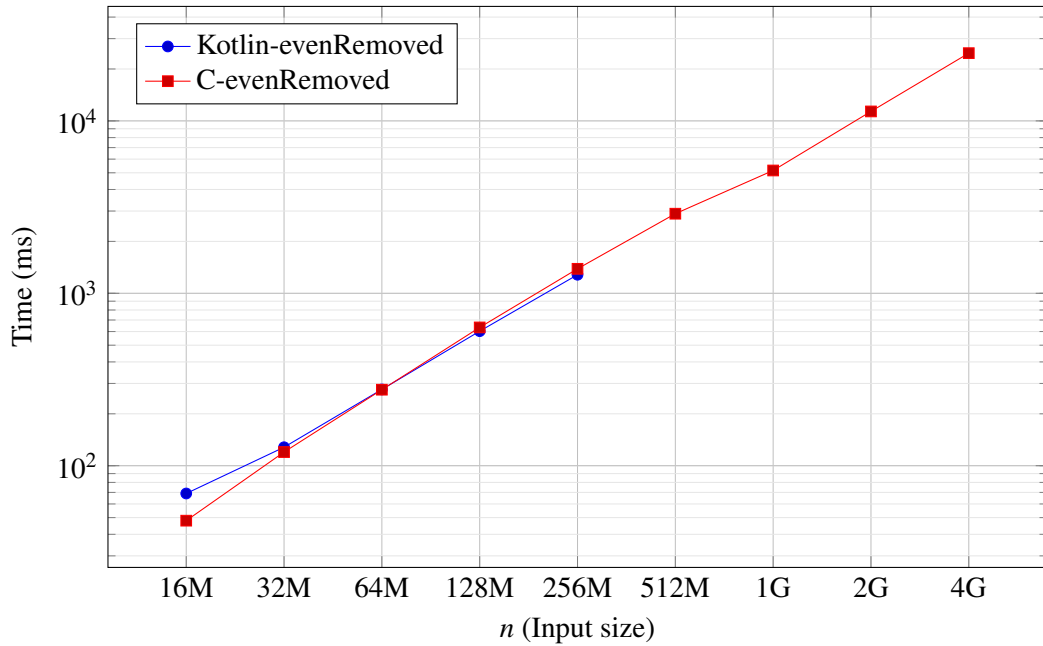


Figure 5.2: Time comparison of `evenRemoved` implementations (log-log scale)

5.5 Conclusions

In conclusion, this study indicates that the Sieve of Eratosthenes algorithm demonstrates similar execution times in both C and Kotlin. However, the tests conducted were limited to lower values of n due to memory consumption issues associated with the Android Runtime (ART) when using Kotlin. Consequently, we could not determine whether this performance would remain consistent with higher values of n .

The primary takeaway from this investigation is that to optimize memory usage on a device, it is preferable to use a low-level programming language like C rather than the more modern Kotlin. However, since our tests were limited to lower values of n and the performance was similar, we cannot conclude that low-level languages provide better performance regarding execution time on Android. Nonetheless, we can confidently affirm that if we need to allocate a significant amount of memory, we must choose Android NDK development.

One reason we could not achieve significantly better results using a low-level approach on Android might be related to the choice of algorithm. We believe that the algorithm employed did not require enough intensive computation to reveal a notable difference in execution times. Therefore, we decided to explore other algorithms, such as the Fast Fourier Transform and Matrix Multiplication, in Chapter 6.

Chapter 6

Exploring Fast Fourier Transform and Matrix Multiplication

6.1 Introduction

In this chapter, we explore the Fast Fourier Transform (FFT) and Matrix Multiplication (MM) algorithms. These algorithms require more intensive computations, particularly floating-point operations, compared to the Sieve of Eratosthenes discussed earlier. This makes them suitable candidates for evaluating the potential performance benefits of low-level implementations on heavy computational tasks. For the FFT, we utilized an open-source implementation developed by Nalon [22], while the MM algorithm was implemented by ourselves following the mathematical calculation and making several optimizations along the way. The methodology used for the testing environment and analysis of the algorithms remained consistent with the approach described for the Sieve of Eratosthenes in Chapter 5.

6.2 FFT Algorithm

The Fast Fourier Transform (FFT) is a fundamental algorithm for efficiently computing the Discrete Fourier Transform (DFT). Its applications span across fields such as signal processing, audio analysis, and image compression. In our work, we used the repository by Nalon [22] containing FFT implementations in both Kotlin and C. This choice ensured consistency in the algorithmic structure across the two programming languages.

The repository offered three variations of the FFT:

- **Direct Fourier Transform (Direct FT):** the most straightforward, but also the slowest implementation.
- **Recursive FFT:** a divide-and-conquer approach leveraging the Cooley-Tukey algorithm.
- **Iterative FFT:** an optimized version of the Cooley-Tukey algorithm using in-place computation and bit-reversal indexing to achieve better cache performance.

After comparing all three methods, the iterative FFT was chosen for both Kotlin and C implementations because it demonstrated the best execution time in both environments. All measurements and comparisons presented in this section are based on this variant.

Figure 6.1 shows the execution times for iterative FFT on input sizes of 2^{20} , 2^{21} , and 2^{22} , illustrating a clear performance advantage for the C implementation.

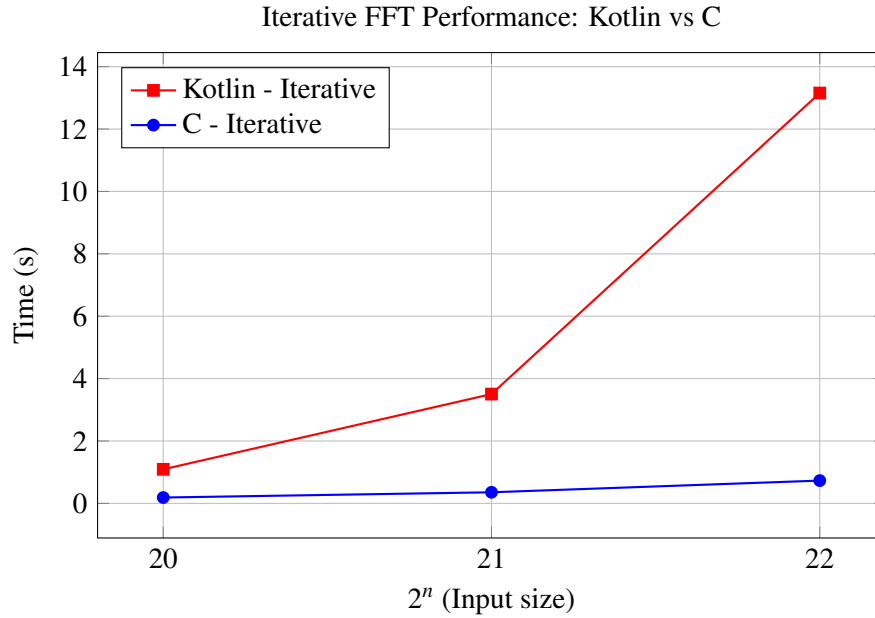


Figure 6.1: Execution time of Iterative FFT in Kotlin and C

Table 6.1: Speed-up of C over Kotlin in Iterative FFT

2^n (Input size)	Kotlin (s)	C (s)	Speed-up
20	1.0918	0.1875	$5.82\times$
21	3.5039	0.3556	$9.85\times$
22	13.1541	0.7304	$18.01\times$

Initially, when we observed a significant gap in execution times, we suspected it might be related to the compiler's optimizations for vectorization. As a result, we decided to investigate this aspect in both programming languages.

Despite these significant performance gains, the analysis of the compiled machine code revealed that the C implementation was not being vectorized. In other words, no SIMD (Single Instruction, Multiple Data) instructions were generated by the compiler, even in release build settings. By analyzing these results, we believe that the observed performance improvements are likely due to better memory access patterns, increased control over computations, and reduced overhead from the runtime environment rather than the explicit use of hardware parallelism. However, this remains an assumption that we could not confirm.

In contrast, Kotlin, which runs on Android Runtime (ART), faces significant performance issues, that we suspect are related to dynamic memory management and limited low-level control. These factors worsen as input size increases, which is evident in the growing performance gap. In conclusion, the iterative FFT emphasizes the significant difference in computational efficiency between native C code and Kotlin on Android. Even without compiler vectorization, C demonstrates a significant speed-up, further supporting its suitability for performance-critical numerical applications.

6.3 Matrix Multiplication

Initial implementations of MM were written in Kotlin, followed by a corresponding C version compiled using the Android NDK. We began with a naive three-loop implementation, allowing us to observe each programming language's baseline behavior. In Kotlin, the matrix is an array of arrays of integers, while in C, it is represented as a linear array of integers.

Following this, we explored loop reordering as a key optimization, observed in Listing 6.1. By rearranging the order of the loops in the standard triple-loop matrix multiplication algorithm, we aimed to take better advantage of the cache locality. Specifically, placing the innermost loop on the index that accesses elements of the same row or column sequentially allows memory accesses to be more predictable and cache-friendly. This reordering had a clear performance impact on both C and Kotlin implementations, and this was the approach we used to compare both programming languages.

```
1   for (int i = 0; i < n; i++) {  
2       for (int k = 0; k < n; k++) {  
3           for (int j = 0; j < n; j++) {  
4               C[i * n + j] += A[i * n + k] * B[k * n + j];  
5           }  
6       }  
7   }
```

Listing 6.1: Loop reordered matrix multiplication (C implementation)

We chose matrix multiplication to evaluate the potential of compiler optimizations, particularly vectorization, due to its regular access patterns and relative algorithmic simplicity. The C implementation demonstrated significant improvements in execution time across all matrix sizes. However, in the Kotlin implementation, we could not verify whether the Android Runtime applied any vectorization. Our inspection of the generated bytecode in the `.dex` file showed no indications of vectorization, but this does not conclusively mean that vectorization is absent in the machine code produced by the Android Runtime (ART) because the bytecode instruction set does not contain any vector instructions.

The performance results illustrated in Figures 6.2 through 6.4, along with the speed-up values reported in Table 6.2, reveal critical insights into how different language runtimes and optimization levels affect matrix multiplication.

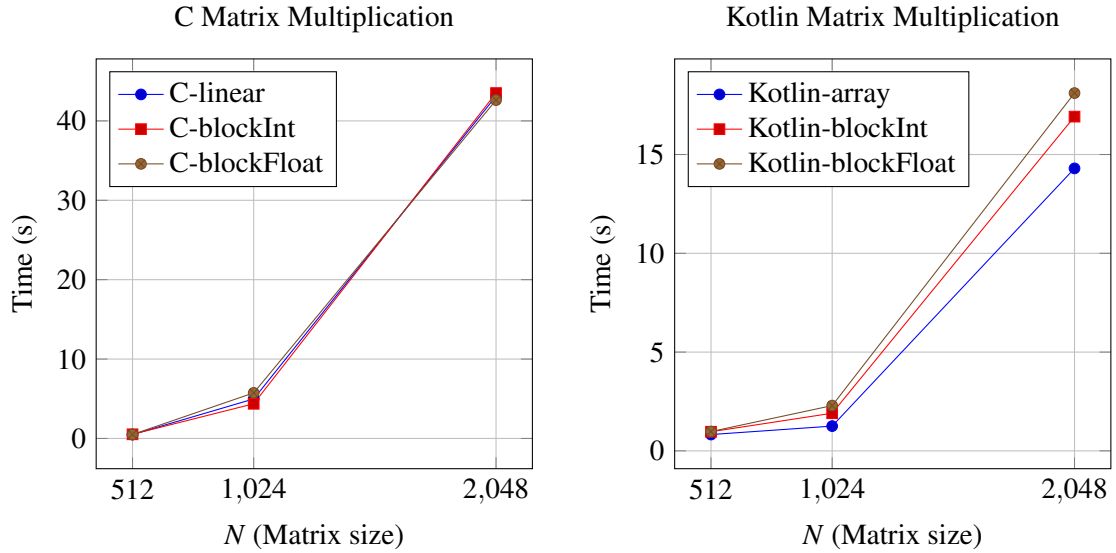


Figure 6.2: MM performance comparison on C (-O0 Flag) and Kotlin implementations

Figure 6.2 compares the baseline performance of various implementations in both C and Kotlin. All C results were generated using the -O0 compiler flag (no optimization), and they consistently show higher execution times than Kotlin for larger matrices. In contrast, Kotlin performs more steadily across increasing matrix sizes, likely benefiting from internal ART optimizations, garbage collection, and efficient memory allocation.

Figure 6.3 isolates the linear implementation in C and the array implementation in Kotlin, emphasizing that, without optimization, C is notably slower, particularly at $n = 2048$, which takes over 43 seconds to complete versus 14.3 seconds in Kotlin. This illustrates that unoptimized native code can easily underperform compared to managed environments, especially when relying on default compiler behavior.

Table 6.2: Speed-up of C-linear (-O3) over Kotlin-array for Matrix Multiplication

N	Kotlin (s)	C (-O3) (s)	Speed-up
512	0.826333	0.036145	22.87×
1024	1.253333	0.269224	4.66×
2048	14.297667	2.186385	6.54×

Once the -O3 optimization flag is enabled (as shown in Figure 6.4 and Table 6.2), the performance profile of C changes significantly. By using this optimization flag, we enable vectorization, which utilizes the CPU's SIMD vector, therefore justifying the considerable speedup compared to Kotlin's execution time.

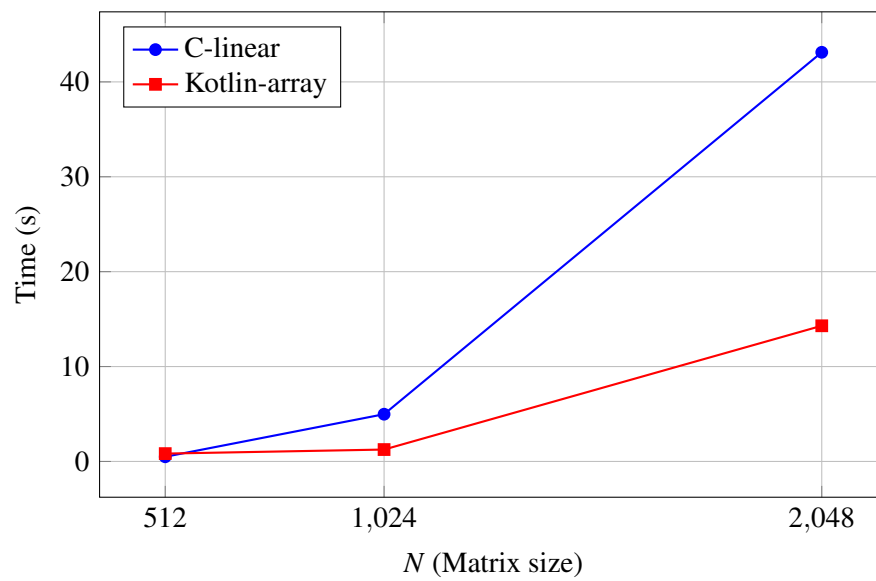


Figure 6.3: Performance comparison: C (-O0 Flag) vs Kotlin for MM

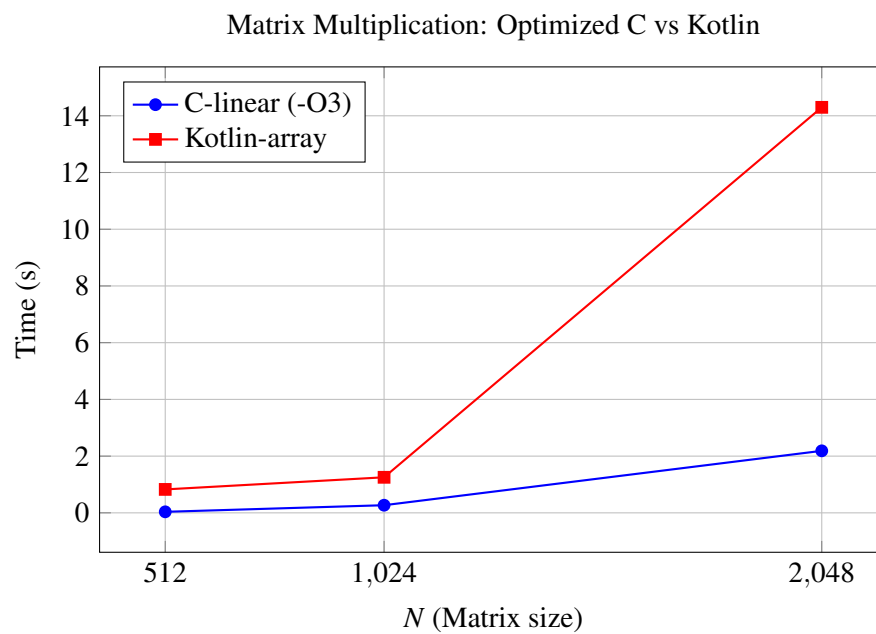


Figure 6.4: Performance comparison: C (-O3 flag) vs Kotlin for Matrix Multiplication

6.4 Conclusions

The performance evaluations of both FFT (Fast Fourier Transform) and Matrix Multiplication reveal consistent patterns. The JVM-based Kotlin implementation benefits from automatic memory management and exhibits stable scaling behavior. However, it faces performance bottlenecks due to a lack of vectorization and runtime overhead. In contrast, C implementations gain significant advantages from manual loop restructuring compared to Kotlin. Additionally, the use of compiler optimization flags improved the performance of the C implementation, achieving a speedup of 6.54, with a matrix size of 2048. Also, caution is required with release build configurations because mistakes, such as accidentally compiling in debug mode, can diminish these performance gains. Additionally, neither C (with -O0) nor Kotlin showed signs of auto-vectorization for these algorithms, even when using vector-friendly code structures. Manual SIMD optimization may be necessary to achieve further improvements.

In summary, our findings emphasize the critical importance of low-level control in performance-sensitive applications on constrained devices. While Kotlin offers convenience and safety, native code delivers the raw speed necessary for demanding workloads, such as FFTs and large-scale matrix operations.

Chapter 7

Wake Word Detection and Voice Command Application

7.1 Introduction

This chapter introduces an Android mobile application designed to detect a specific hotword or wake word in the user's speech. Once the hotword is detected, the app listens for a voice command in real-time. If a command is recognized, the application performs the corresponding action. Essentially, it functions as a voice assistant tool, similar to other well-known options like Alexa, Siri, and Google Assistant.

One main difference between these tools and our implementation remains in the fact that while the Original Equipment Manufacturers (OEMs) voice assistants take advantage of the unlimited access to the Android Framework and the entire Android API, including the System API, which is only available to partners and OEMs ¹. This level of access enables the OEMs to utilize system services such as the `VoiceInteractionService` ² and create always-on hotword detectors that are executed closer to the hardware responsible for signal processing in real-time, commonly known as the Digital Signal Processor (DSP).

We utilize pre-trained machine learning models that operate with a C-based inference engine using the Android NDK. These models are responsible for detecting the wake word within a given audio excerpt captured from the microphone. When the hotword is triggered, the application starts analyzing the audio and transcribing the spoken words into text. This text is processed by another machine learning model, which interprets the input and identifies the user's intention behind the voice command. After validating this intention, the application translates it into a specific action that is automatically executed.

This system is designed for on-device use and operates completely offline, with no need for communication with external servers. All models are executed locally on the device, ensuring

¹<https://source.android.com/docs/core/architecture>

²<https://developer.android.com/reference/android/service/voice/VoiceInteractionService>

they are lightweight enough to meet the resource constraints of mobile devices, including memory, energy, and computational power.

7.2 Project Setup

7.2.1 Development Environment

This application, along with the algorithms tested in Chapters 5 and 6, was developed using the Android Studio IDE. Android Studio is one of the most widely used integrated development environments (IDEs) for mobile development. It also offers tools for creating applications for various environments and operating systems, including Android TV for televisions, Android Auto for automotive systems, and Wear OS for wearable devices. To automate and manage the building process of our application, we use Gradle, specifically Gradle Kotlin DSL. This tool is the default and fully supported option for project configuration in the Android Studio IDE, including dependency management, code packaging, and compilation.

In terms of building configuration, we set the minimum target SDK to 30, as our testing device ran Android 11, which corresponds to API level 30. For native development, we used the Android NDK r27, as it was the latest long-term support (LTS) version available at the time of our research.

7.2.2 Open Wake Word library

Our previous research identified two tools that provide ready-to-use wake-word detection: Picovoice Porcupine and the OpenWakeWord (OWW) library. Further investigation revealed that Picovoice's software is not entirely free. While it does offer a free plan, users must obtain an `AccessKey` to utilize the Picovoice SDK and run its models. This key must be provided to Picovoice servers to monitor whether the developer has reached usage limits, which also requires an internet connection. In contrast, the OpenWakeWord library is completely free under the Apache License 2.0. It provides an easy-to-use command-line interface for model inference and includes several pre-trained models for common wake words. Despite these advantages, OWW is implemented only in Python by default, which limits its straightforward use in Android applications. Therefore, the proposed solution was to extract the pre-trained models from the library and run them on another inference framework compatible with Android, preferably with a C/C++ and Java/Kotlin API implementation.

In this library, wake word detection is achieved by sequentially running three different models: the mel-spectrogram model, the embeddings model, and the classifier model.

The mel-spectrogram model is based on the mel-spectrogram function from the PyTorch framework. It has a size of 1.04 MiB and operates using 32-bit floating-point values. The model accepts a dynamically sized input tensor, allowing us to provide audio frames of various lengths. It computes the corresponding mel-spectrogram with 32 mel bands, which determines the frequency resolution of the mel-spectrogram. In Table 7.1, we can observe the tensor shape dimensions. The

dimensions with a negative value are dynamic, meaning their size is unknown until runtime, while the dimensions with positive values are fixed in size.

Table 7.1: Input and output tensor specifications of the Mel-spectrogram model.

Model	Tensor	Description
Mel-Spectrogram	Input	Shape Signature: $[-1, -1]$ Dtype: <code>float32</code>
	Output	Shape Signature: $[-1, 1, -1, 32]$ Dtype: <code>float32</code>

The embeddings model is the core component that extracts audio features by converting mel-spectrogram inputs into general-purpose speech audio embeddings. As stated in the documentation of the OWW library, this model is based on Google’s `speech_embeddings` model. It operates on 32-bit floating-point values and has a file size of 1.27 MiB.

As shown in Table 7.2, the model accepts an input tensor comprising a mel-spectrogram with 76 time frames and 32 mel bands. The output tensor produces a single 96-dimensional embedding vector, which represents relevant audio features extracted from the mel-spectrogram.

Table 7.2: Input and output tensor specifications of the Embeddings model.

Model	Tensor	Description
Embeddings	Input	Shape Signature: $[-1, 76, 32, 1]$ Dtype: <code>float32</code>
	Output	Shape Signature: $[-1, 1, 1, 96]$ Dtype: <code>float32</code>

The final model in this pipeline is the classifier. It operates on 32-bit floating-point values and has a size of 1.22 MiB. As presented in Table 7.3, the input tensor for this model receives 16 embedding vectors, creating a 16 x 96 matrix of embeddings. The model processes this information and compresses it down to a single floating-point value, representing the confidence score that indicates the probability of detecting the wake word.

Table 7.3: Input and output tensor specifications of the Classifier model.

Model	Tensor	Description
Classifier	Input	Shape Signature: $[1, 16, 96]$ Dtype: <code>float32</code>
	Output	Shape Signature: $[1, 1]$ Dtype: <code>float32</code>

To clarify how the models work together, the process begins with audio data, which can be captured through a microphone or extracted from an audio file. This audio data is input into

the first model, which calculates the approximate mel-spectrogram values for each audio sample. These values are then passed on to the second model, responsible for extracting audio embeddings. Finally, the extracted embeddings are fed into a classifier model, which outputs a float value representing the probability of detecting the targeted wake word for which this model was trained. This library includes different variants of classifier models, each trained to detect specific wake words such as "Alexa", "Hey Jarvis", and others. Our application depends on these key models to process audio data and accurately detect the wake word.

However, the data flow between these models is not direct. This means that the output tensor data must be manipulated so the next model can receive it in the appropriate input tensor format. Since the tensors do not match exactly between models, we must utilize buffers to save data from previous iterations. This approach ensures that the data structure has a matching size for the input tensors. A more detailed explanation of this approach is provided in Section 7.7.

7.2.3 Android NDK and TensorFlow Lite

From the study presented by Febvay [7], we discovered several machine learning frameworks supported by mobile devices, most of which are available for Android. Among all the frameworks, two particularly caught our attention. The first, TensorFlow Lite, is noted for its practicality and overall average performance in inference time. The second framework, Paddle Lite, stands out due to its extensive optimizations for ARM-based CPUs, achieving the best performance on CPU inference time among all the frameworks presented in the study.

Ultimately, we decided to use the TensorFlow Lite library because of its user-friendly setup, as noted by Febvay. Additionally, due to the project's time constraints, we opted for an approach that was achievable within the allotted timeframe. Given that the performance is acceptable, it will not be a bottleneck in our application. The name of this library was changed to LiteRT (Lite Runtime) in 2024. However, since TFLite (TensorFlow Lite) is the term most commonly used in the documentation and the actual library code, this dissertation will consistently refer to it as TFLite, even though the official name is LiteRT.

7.2.4 Java Native Interface functions

The application is mainly developed using the Android SDK, meaning that the code is mainly written in Kotlin. However, to execute the models in the TensorFlow library using C, we need a bridge to connect the Kotlin side of the application with the C side, and that is where the Java Native Interface comes in. The JNI³ is an interface that allows the bytecode generated from Java or Kotlin to interact with the compiled native code in C.

Essentially, we declare in Kotlin the entry point function as `external` and it is linked to a C function generated using the JNI naming convention. Within this C function, the JNI environment allows us to obtain a pointer to the data passed through the parameters or, in some cases, a copy of that data.

³<https://docs.oracle.com/en/java/javase/24/docs/specs/jni/index.html>

For example, when using the `Get<PrimitiveType>ArrayElements` functions with array parameters, these functions may either create a copy of the data or offer a direct pointer to its address. Although the developer cannot control which option is chosen, they can identify what happened by checking the `isCopy` argument. If `isCopy` is `JNI_FALSE`, the returned address points directly to the original Java/Kotlin array, which the ART has pinned to prevent the garbage collector from moving it during the critical section. Otherwise, if `isCopy` is `JNI_TRUE`, the ART has allocated native memory, copied the elements, and returned a pointer to that buffer.

Additionally, the corresponding calls to `Release<PrimitiveType>ArrayElements` allow us to choose whether to commit any changes back to the managed array or simply release the native buffer. A practical alternative that prevents an explicit release step is to use region calls, such as `Get<PrimitiveType>ArrayRegion` and `Set<PrimitiveType>ArrayRegion`. These functions copy data to or from the managed array in a single call and manage all memory operations internally.

7.3 Wake Word Detection

7.3.1 Audio Capture and Processing

The Android operating system supports audio encoding in various formats, including AAC LC, FLAC, and PCM. Our application captures audio from the microphone using the PCM format, which is sampled at a rate of 16 kHz (16000 samples per second) in mono mode (one channel only). Each sample is represented by a 16-bit floating-point number in half-precision format. We also take advantage of a dedicated audio source mode provided by the Android Media library, which is specifically designed for voice recognition. The minimum number of audio samples that can be obtained from the microphone varies based on the device hardware and its Android version. In our case, the minimum audio buffer size is 1280 samples, corresponding to a duration of 80 milliseconds at a sampling frequency of 16 kHz. It is important to note that the audio capture and processing described in this subsection are implemented on the Java/Kotlin side of the application.

We utilized the three pretrained models from the OWW library to create a pipeline that processes audio data from the microphone and outputs the probability of detecting a specific wake word. The process begins with the mel-spectrogram model, which computes mel-spectrogram values for each audio sample. These values are then passed to the embeddings model, which extracts relevant audio features. Finally, the extracted features are fed into the classifier model, which calculates the probability of detecting a specific wake word in the provided audio chunk. The classifier requires a minimum of 2 seconds of processed data to operate.

It seems logical to update our audio buffer every 2 seconds, as the wake word is short enough to fit within this time frame. However, this approach is not optimal because there are instances when the wake word could be detected if it falls precisely within the 2-second window extracted from the microphone. On the other hand, if audio samples containing the wake word are split between two different time frames, the model may fail to detect the word correctly.

To address this problem, we increased the frequency of audio extraction. As mentioned earlier, the minimum amount of audio we could sample from the microphone is 80 ms (or 1280 samples). However, extracting audio data and processing the models every 80 ms can be costly and significantly increase the device's energy consumption. Therefore, we maintain an internal audio buffer of 2 seconds but read new audio samples from the system audio buffer every 250 ms. This interval is inspired by the frequency at which the acoustic model is updated in the Siri system, which is set at 200 ms [28].

In this strategy, new audio samples are appended to the end of the internal buffer, while the oldest 250 milliseconds of samples are removed, preserving a constant 2-second buffer length. As a result, the models can process the newly updated buffer and decrease the likelihood of splitting the wake word across different frames, since the frames are now much smaller than the length of the wake word.

However, this approach introduces another challenge regarding system efficiency. By feeding the models a 2-second buffer every 250 milliseconds, there is 87.5% of data that remains unchanged, resulting in unnecessary recomputation. To address this issue, we implemented a 2-second mel spectrogram buffer that updates only with new 250 ms of data, allowing the embeddings model to process fresh audio embeddings each update step. This strategy mirrors the real-time wake word detection functionality implemented in the OpenWakeWord library.

Nonetheless, this method still seems somewhat redundant because we continually update the embeddings model with overlapping data segments. Ideally, new audio data should be inserted directly into the embeddings model without reprocessing previous segments. To achieve this, the model would need to maintain and manage historical data, introducing a notion of state. This idea was explored by Rybakov et al. [31], who proposed architectures that incorporate internal or external states to handle streaming inference efficiently.

7.3.1.1 Streaming and Non-streaming Models

Rybakov et al. [31] introduced the concept of adapting models for streaming inference, where data is processed incrementally rather than as complete sequences. A streaming model can continuously handle small portions of incoming data, providing real-time predictions and significantly reducing detection latency compared to non-streaming models, which wait for the entire input before generating results.

Additionally, they differentiate between models with internal state and external state. Models with internal state manage their own history, maintaining buffers of previous frames that are updated automatically during inference. In contrast, models with external state treat these buffers as part of their input and output, requiring developers to manually manage the states between successive inference calls. This distinction is crucial for efficient wake word detection in streaming applications.

To adapt models for streaming inference, we needed to create our own models with a different internal structure. This will involve specifying certain layers in the neural network that are designed for processing streaming audio data. Additionally, we would require more time for training

and fine-tuning these new models, even though we already had the models pre-trained. As a result, we set this solution aside and proceeded with our previous implementation.

7.3.2 Audio Sampling: WAV file vs Mic

In the initial phase, we tested the model using audio extracted from the microphone. However, we needed to start processing audio files for better development productivity. This audio processing on Android presented some challenges because most of the libraries we found did not work immediately or as expected. The Android library has no available functionality for resampling audio from a file (e.g., WAV format), it is only possible through microphone audio. Therefore, we needed an external library to perform this audio resampling.

We explored two Java libraries that allow for effective loading and management of audio files: jLibrosa⁴ and TarsosDSP [35]. The jLibrosa is a Java library that offers functionalities similar to the Python library librosa. It is specifically designed for audio signal processing and feature extraction, particularly in the fields of speech and music analysis. With jLibrosa, users can load audio files, compute spectrograms, and extract Mel-frequency cepstral coefficients (MFCCs). On the other hand, the TarsosDSP is an open-source Java audio processing library that supports both real-time and offline audio analysis, synthesis, and transformation. It includes a wide range of features such as pitch detection, filtering, effects, audio input/output support, and tools for audio feature extraction, also including MFCCs and spectrograms.

Following that, we began by testing audio resampling using the jLibrosa library. While it functioned well with shorter audio files of only a few seconds, it struggled with longer files exceeding 40 minutes. This issue arises from the library's implementation, which requires loading and processing the entire audio file into memory. This can lead to Out of Memory (OOM) exceptions on Android systems. Consequently, we could not apply the chunked strategy in audio processing using this tool.

We encountered several issues with TarsosDSP that we initially attributed to the fact that this library had not received any core updates in nine years. However, we later realized that the actual problem was linked to the different implementations for JVM and Android. The JVM version utilized FFmpeg to resample and convert the audio into a 16 kHz mono channel with 16-bit floating-point samples. This program can be easily installed on Windows or Linux-based systems, but the same does not apply to Android systems, where user privileges are more restricted. One example⁵ provided by the library creators indicated that developers must manually install an executable binary of FFmpeg in the Android file system for the library to function properly. While this may have been feasible with older versions of Android, in our specific case with Android 10, we faced additional challenges. Despite it not being a recent Android version, after copying the

⁴<https://github.com/Subtitle-Synchronizer/jlibrosa>

⁵https://0110.be/posts/Decode_MP3s_and_other_Audio_formats_the_easy_way_on_Android

ARM-compiled binary to the expected folder, the audio conversion tool returned an error indicating the permission to execute the binary file was denied, even though the file had the necessary permission flags activated.

The solution we found involved using a specific library called FFmpegKit, which implements the same program mentioned earlier, specifically for Android. This library allows us to convert audio files effectively. We converted the WAV audio files to a format with a sample rate of 16 kHz, mono channel, and 16-bit sample values. After that conversion, we processed the file as an input stream of bytes, skipping the WAV file header to access the sample values directly. This process ensures that the sample values are available in 16-bit floating-point format, making them ready for insertion into the mel spectrogram model.

7.3.3 Audio Recording and Inference Threads

After developing both implementations for audio file inputs and microphone inputs, we noticed a significant difference in inference times between the two. For the audio file source, the inference time was approximately 2 ms for the mel-spectrogram model. In contrast, the microphone source showed considerable variability, with inference times fluctuating between 10 ms and 15 ms, or even higher at times.

We began by analyzing the logs for each inference step of the models that were running. Later, we observed that if we stopped the audio recording before the last inference step was completed, specifically when the stop button was clicked, it resulted in faster inference times. This finding suggested that the issue was related to the audio recording process, rather than being due to any errors in our implementation.

After conducting a thorough investigation, we discovered that the model's inference was running on the same thread as the continuous audio recording. This conflict was causing interference and slowing down the model's execution. To resolve this issue, we created a dedicated thread for each audio recording and model inference. This adjustment ensured that inference times are approximately equal in both approaches, as expected.

7.3.4 Detecting Silence and Human Speech

The initial energy consumption of the system was higher than we had targeted. It consumed approximately 0.051% of battery per minute, allowing it to last only one full day on a single battery charge, while the device was only running the wake word detection application. This led us to explore strategies for reducing energy consumption and enhancing the overall performance of the system.

When the wake word detection system is activated using audio collected from the microphone, we encounter various types of audio. The audio may consist of the user speaking, background environmental noise, silence, or a combination of these elements. This differentiation allows us to apply preprocessing to the audio data before using it to detect the wake word.

The first improvement we implemented involved detecting silence in the audio. We established a threshold for the amplitude of the audio waveform after normalizing it. In Figure 7.1, we can see a user speaking in a quiet environment, which highlights the silent segments where computational resources are being wasted. After analyzing the audio samples from those segments, we observed that the absolute value of the normalized samples did not exceed 0.02. As a result, we established the audio threshold at 0.02. If the maximum absolute value of the audio samples in a specific window falls below this threshold, that array of samples will be disregarded and excluded from the wake word detection process. Since we derived this value from analyzing the audio processing of this specific device, it is possible that this threshold could differ on other devices.

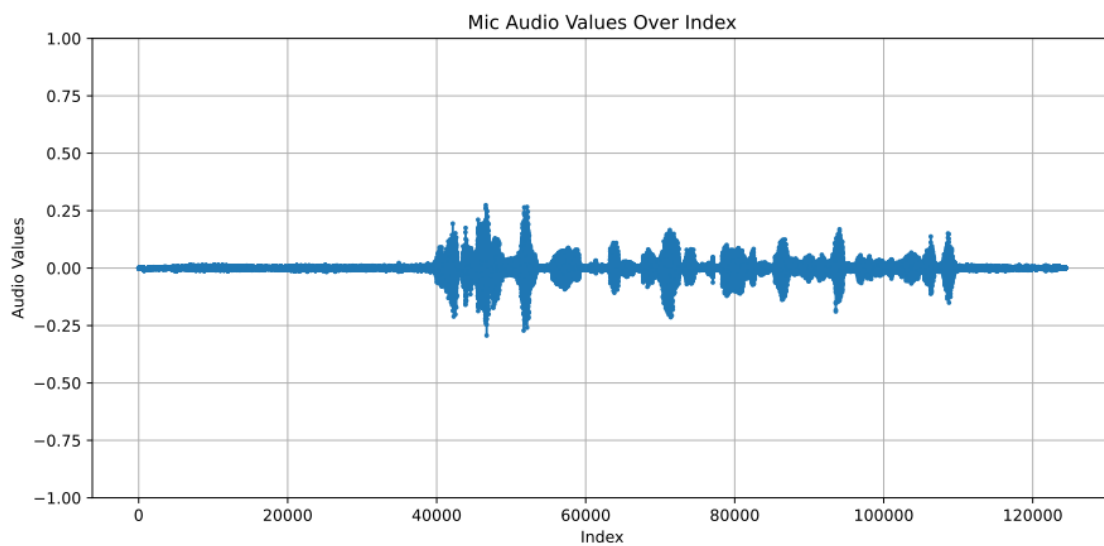


Figure 7.1: Normalized audio samples recorded from the microphone while a user is speaking.

The second enhancement focuses on environmental noise. Even when audio samples are recorded, they may not necessarily contain the user’s voice. To determine whether the captured sound is human speech, we employed a library called Silero [36], which is a “pre-trained enterprise-grade Voice Activity Detector (VAD)”. Silero VAD uses Deep Neural Networks (DNN) and runs on the ONNX Runtime Mobile for efficient execution. To utilize this, we integrated an implementation of the library designed for Android ⁶, and since the maximum available audio frame size at a rate of 16kHz was 1536, we reduced the sliding window of audio to 96ms. With this improvement, we have successfully reduced energy consumption to approximately 0.022% per minute during a test run conducted on a quiet night (see Figure 8.4). Additionally, during a test performed while on a car trip, the consumption was on average 0.026% of battery per minute (see Figure 8.8). This reduction allows the background service to operate on the device for significantly longer periods.

⁶<https://github.com/gkonovalov/android-vad>

7.4 TensorFlow Inference Runtime: Java API/C API

After implementing wake word detection using the Java API for TensorFlow Lite, we began analyzing the inference times of the models. However, we were unable to test all the models because the mel spectrogram model could not be loaded with the Java API. This issue arose from the Java API's lack of support for models with dynamically sized tensors, which specifically affected this model since it accepts a variable number of audio samples as input. Consequently, we continued our analysis using only the other two models.

Since we did not extract the mel spectrogram values from the audio, the input for both models consisted of buffers filled with random data. After measuring the inference times for these models, we found that the classifier model had an average inference time of 2 milliseconds, while the embeddings model averaged 27 milliseconds. These results were similar to those obtained using the C API, leading us to conclude that the inference runtime is consistent across both APIs.

A closer examination of the TensorFlow Lite source code on Google Git ⁷ revealed that the Java API employs JNI wrapper functions to access the native functionality for executing model inference. Furthermore, a comment from a member of the TensorFlow organization on GitHub ⁸ suggested that performance would be comparable between the Java API and the native API, given that ByteBuffer or FloatBuffer is used for input and output tensors.

In contrast, the embeddings model and the classifier model both utilize fixed-size input tensors. As a result, we did not experience any issues when loading and executing these models using the Java API. As we expected, the inference time remains roughly the same when utilizing the C API.

7.5 Attempted Optimizations on Model Inference

This section outlines the two optimization strategies we explored and the practical considerations that influenced them. First, we examined the use of GPU acceleration with TensorFlow Lite delegates, aiming to fully or partially offload the inference models to mobile GPUs in order to reduce latency and energy consumption. Second, we assessed the feasibility of on-device continual training, which personalizes the detector for new speakers but requires models that are specifically structured for incremental updates.

7.5.1 GPU Inference

TensorFlow Lite uses the CPU as the default hardware for running model inference. To use different types of hardware, such as GPUs, TPUs, or DSPs, we need to employ specific hardware drivers known as delegates. Upon checking the availability of the GPU delegate on our testing device, we discovered that it was not available. Although GPU inference is reported to be supported on GPUs

⁷https://android.googlesource.com/platform/external/tensorflow/%2B/6b511124eb0/tensorflow/lite/java/src/main/native/nativeinterpreterwrapper_jni.cc

⁸<https://github.com/tensorflow/tensorflow/issues/41825#issuecomment-671429103>

that feature OpenGL ES 3.1⁹, we were unable to utilize this feature on our device. When we called the `isGpuDelegateAvailable` method at runtime, it always returned `false`.

We then tested the GPU delegate on another device running Android 12, equipped, which was equipped with the Exynos 9611 chipset and Mali-G72 MP3 GPU, and we were able to successfully create model interpreters.

However, we encountered issues related to model compatibility with the GPU. There is a specific set of supported ML operations for the GPU¹⁰, and two of the models required for wake word detection—the mel-spectrogram model and the classifier model—contained nodes in their graphs that performed operations not included in this set. In such cases, the framework’s default behavior is to run the unsupported nodes using the CPU delegate while executing the remaining parts of the model graph on the GPU. Unfortunately, this split execution often results in slower performance compared to executing the model solely on the CPU, leading the application to trigger warning messages regarding the unsupported operations on the GPU delegate, such as:

```
1 WARNING: op code #42 cannot be handled by this delegate.
```

If we encounter these warnings, it indicates that the model is not suitable for GPU acceleration. Addressing this problem would require restructuring the internal structure of the models, which is not feasible in our case since we are using pre-trained models in their exported versions. Therefore, we cannot modify their layers or node operations without rebuilding and retraining them completely.

Despite this limitation, we adapted the application to execute unsupported models on the CPU while running the embeddings model on the GPU. We measured both the model execution time and memory consumption. Regarding execution time, we observed a speedup of 2.3 for the embeddings model, reducing the average execution time from 26.6 ms to 11.6 ms. However, the overall memory consumption of the application increased by 22 MB compared to the full execution of the model on the CPU. As shown in Figure 7.2, the increase in memory consumption was primarily due to the Code category. This suggests that the rise in memory usage may be connected to the framework requiring the loading of additional code to support GPU inference, such as the OpenGL library.

7.5.2 On-Device Training

On-device model training presents certain challenges related to the model’s structure. The model needs to be specifically designed for this type of training. According to the documentation, a TensorFlow model must include the following function signatures: `train`, `infer`, `save` and `restore`¹¹. However, our current models do not meet this structure, and adapting them would require us to rebuild and retrain them, which is not the focus of our study.

⁹<https://blog.tensorflow.org/2019/01/tensorflow-lite-now-faster-with-mobile.html>

¹⁰https://ai.google.dev/edge/litert/performance/gpu#supported_ops

¹¹https://ai.google.dev/edge/litert/models/ondevice_training

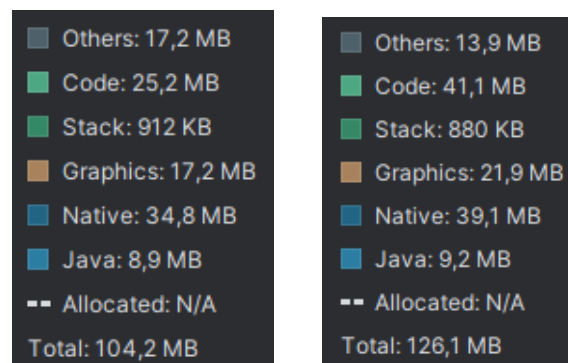


Figure 7.2: Memory consumption on CPU inference (left side) and on GPU inference (right side) extracted from the Android Profiler tools.

7.6 Voice Command Recognition

In this section, we describe the post-process of audio after the wake word has been detected by the classifier model. This system is divided into two different phases: the first is where the human speech is transcribed to text, and the second is where we extract the voice command present in the processed text.

7.6.1 Text-to-speech

As previously described in Section 3.6.2.2, there are several speech-to-text tools capable of performing on-device processing varying in terms of their size and resource utilization, but also regarding their performance in the accuracy and precision. However, not all of these platforms provide this functionality without an internet connection. For example, the Picovoice Leopard has a similar limitation as the Picovoice Porcupine, requiring API calls to verify that a user has not exceeded the free plan limits, despite being capable of on-device processing.

Among the available tools, PocketSphinx and Vosk are better suited for smaller devices with limited resources. They also offer extensive documentation and example applications for mobile development, which facilitates their integration into our application. In a subsequent study, Martins et al. [21] conducted a comparison of the PocketSphinx and Vosk platforms across various tests involving different phrase complexities and audio environments. The findings indicated that Vosk achieved a 4% reduction in error rate compared to PocketSphinx.

Considering these factors, we chose to integrate the Vosk speech recognition toolkit for transcribing user speech. Using Vosk, we successfully transcribed human speech captured by the device. While this tool was relatively simple to implement and integrate into our application, it does come with some trade-offs regarding precision and accuracy in noisier environments.

After we obtained the transcribed text from the user's speech, we faced another challenge: matching the input text with the actual voice command that the user intended to trigger. This could not be solved simply by comparing the input text with the predefined voice command. Users

should have the ability to activate the same voice command using different combinations of words that may not match the exact command but still convey the same intention.

To address this issue, we propose using an additional machine learning model for intent classification. This model analyzes the transcribed speech and assigns probability values to each specific intent within a defined set of user intentions, such as playing music or requesting weather information.

We searched for a pre-trained model that could achieve our goals using only device resources and without an internet connection. Our primary source for results was the Hugging Face model and dataset library. Although we found several models capable of performing this functionality, none of them were exported in TFLite format, which was the type we had been using previously. We aimed to utilize the same model type to reuse our existing system and ensure consistency across the entire framework without having to run multiple machine learning engines for different models.

We ultimately selected a classification model named `BERT-tiny-Massive-intent`¹². This model was trained on the MASSIVE dataset¹³ and is based on the BERT implementation for natural language processing. Although this model was not available in TFLite format, pre-trained versions were accessible using the PyTorch and ONNX frameworks. Both frameworks offer tools specifically designed for converting models to TFLite format.

The tools we found for this conversion are `onnx2tf` and `optimum`. The `onnx2tf` tool¹⁴ is specifically meant to convert ONNX models to TFLite format, making it easier to deploy on mobile or edge devices. On the other hand, `optimum`, developed by Hugging Face¹⁵, streamlines the export and optimization process for Transformer models and includes support for conversion to TFLite using the TensorFlow backend.

Initially, we attempted to use the ONNX tool to convert a model from the ONNX format to the TFLite format. However, we encountered several issues due to version compatibility with the TensorFlow library. Specifically, some dependencies for these tools, such as TensorFlow Addons, were incompatible with newer versions of TensorFlow.

To address this, we created a virtual environment with TensorFlow (TF) version 2.13 and TensorFlow Addons (TFA) version 0.23. Once we managed to execute the tool, we faced a new obstacle: it could not convert the ONNX model because a dependency called `onnx-sim`, which simplifies the model graph and adjusts tensor shape mismatches, was missing. Unfortunately, it was not possible to install `onnx-sim` in the Python 3.12 environment we were using.

As a result, we had to create a new Python environment with version 3.10 and reinstall all the necessary TensorFlow packages along with the ONNX conversion tools and their dependencies. Despite our efforts, we still could not successfully convert the model. This time, the issue was related to tensor transposition, which failed because it was expected to return a tensor with the shape `[1, 512, 1]`, but the output was always `[512]`. We decided not to explore further the

¹²<https://huggingface.co/gokuls/BERT-tiny-Massive-intent>

¹³<https://huggingface.co/datasets/AmazonScience/massive>

¹⁴<https://github.com/PINTO0309/onnx2tf>

¹⁵<https://huggingface.co/docs/optimum/main/en/exporters/overview>

issues regarding Python environment versions and tensor model differentiations, as this was not our main focus, and we had already spent more time on this matter than anticipated.

After facing challenges with the ONNX tools, we decided to take a different approach by using the model file extracted with the PyTorch framework to convert it to TFLite format. To use the `optimum-cli` tool we mentioned earlier, we first needed to export the model to a `.safetensors` file format, which is required for running the `optimum-cli` tool. The `safetensors` file format is commonly used for storing and sharing model weights and is widely utilized within the Hugging Face ecosystem. Once we had the `safetensors` file and the `config.json` file downloaded from the model's repository, we successfully converted the model into TFLite format using the following command:

```
1 optimum-cli export tflite \  
2     --model          ./model_directory \  
3     --task           text-classification \  
4     --batch_size     1 \  
5     --sequence_length 128 \  
6     --quantize int8-dynamic \  
7     model_directory_tflite/
```

Listing 7.1: Export command for TFLite format

We obtained the TFLite model file, which we tested using a Python script in a PC environment, and it functioned perfectly. However, there was still one issue regarding the input data required for the model to operate effectively on our mobile device.

The model has three different input tensors: `model_input_id`, `model_attention_mask`, and `model_token_type_ids`. Each tensor receives information processed by an external tokenizer, which converts the user's voice command into text. In our Python script, we used a tokenizer from the Transformers library, which utilized a `tokenizer.json` file from the model's repository to configure the tokenizer. However, the Python Transformers library is not supported on Android. Therefore, we need to find a workaround to tokenize the text in order to execute the intent classification model. After conducting some research, we discovered a library called TensorFlow Lite Task Library¹⁶ that provides various tools for supporting and executing key machine learning models, including image and text classification.

This library features an API called `BertNLClassifier`, which enables the classification of input text into various categories using BERT natural language classifier models. The library takes a TFLite model as input and is responsible for tokenizing the input text and retrieving the model's classification output, effectively creating a complete pipeline for our model. However, after conducting investigations and tests with this tool, we found that our model required specific metadata related to the tokenizer configuration in order for the API to function correctly.

¹⁶https://ai.google.dev/edge/litert/libraries/task_library/overview

As we explored different methods to insert metadata into the model, we utilized the Metadata Writer API ¹⁷, but none of our attempts were successful. Either we could not add the metadata to the model, or if it seemed to be added, the model failed to execute in the Android application due to issues with the metadata. We did not invest much time in understanding why this approach was failing and decided to shift our focus to externally tokenizing the text before providing it to the model. This alternative would give us more control over the data and improve our ability to debug the issues.

To accomplish this task, we needed to implement the tokenizer ourselves. Fortunately, after some research, we discovered a Java library called the Deep Java Library. This is an engine-agnostic framework for deep learning in Java, and it includes an implementation of the Hugging Face tokenizer we had previously used to test the model in Python.

After integrating this tokenizer into our application, we were able to retrieve the necessary data for the three input tensors required by the model. The first tensor, `model_input_id`, is the main input token sequence, which contains the integer IDs of the tokens representing the input text after tokenization.

The second tensor, `model_attention_mask`, serves as a mask, indicating which tokens are actual data (represented by 1) and which are padding (represented by 0). This mask helps the model ignore the padded tokens during attention computations.

The third tensor, `model_token_type_ids`, is primarily used for tasks involving sentence pairs, such as question answering. In cases of single-sentence classification, like intent detection, this tensor typically consists of all zeros. It helps distinguish tokens between two different sentences.

Finally, we successfully executed the model using the TFLite inference engine with the tensor data, allowing us to retrieve the probability for each output intent. This probability is represented as a floating-point number ranging from 0 to 1. We only consider the model's output valid if the intent with the highest probability exceeds 0.5. This threshold was empirically verified to ensure that the result is relevant. Therefore, if the probability is lower than 0.5, the application will not trigger any action. The model was trained with a set of 60 intentions, some of which are `play_music`, `weather_query`, and `audio_volume_mute`. These were the specific intentions we focused on for testing and demonstration purposes, as they are associated with actionable responses.

7.6.2 Launch Activities and Background Service

Next, we needed to associate the selected intents with the related activity we wish to launch on Android to trigger the action that reflects the user's intention.

In the case of the `play_music` intent, our goal was to launch the default music player and start playing any music using the intent `Intent(Intent.CATEGORY_APP_MUSIC)`. For the `weather_query` intent, we aimed to launch the default application associated with the `Intent.CATEGORY_APP_WEATHER`. However, this intent category was introduced in API 33,

¹⁷https://ai.google.dev/edge/litert/models/metadata_writer_tutorial

whereas our testing device operates on API 30. To resolve this issue, we created a settings page within the application that allows the users to select their preferred application for both intents from all the applications available in `Intent.CATEGORY_LAUNCHER`.

To implement this approach, we needed to add a new permission in the `AndroidManifest.xml` file, which is the `android.permission.QUERY_ALL_PACKAGES`. This permission allows our application to access a list of all installed packages on a device. However, it is generally recommended to avoid using this permission, as any application published on Google Play must obtain approval for its use and provide a clear justification for needing it ¹⁸.

Until this point, our application has been completely visible on the screen. However, we aimed to allow users to interact with their phones while the application is closed. To accomplish this, after users configure the application for each intent, the main activity can launch a background service that runs the wake word detection system without displaying any graphical components. This service continues to function even when the device is locked.

However, we encountered a challenge when attempting to launch another application from this background service in response to the user's intent trigger. No action or error log was generated, and it seemed that the Android system simply ignored the request while continuing to execute the service. After researching the Android documentation ¹⁹, we found that starting from Android 10 (API level 29) and higher, there are restrictions on launching activities from apps running in the background. These restrictions mainly arise from the lack of a visible activity, or application in the back stack, that could initiate another application.

One potential solution ²⁰ was to place the application in a privileged partition of the Android system. However, this would require root access to the device. Another option was the signing of the application with a platform key, which would grant the application OEM permissions. However, this approach is not easily feasible for a typical mobile developer without privileged access to the Android system. Additionally, the source we found that proposed these solutions was not very detailed or straightforward. As a result, we decided not to pursue these options further due to time constraints. Consequently, our application only launches new activities when it is visible on screen.

7.7 Application Overview

The application we developed offers real-time wake word detection along with basic voice command recognition. The real-time wake word detection system operates according to the strictly feed-forward path illustrated in Figure 7.3.

The audio PCM samples are captured from the device microphone at a frequency of 16 kHz in 96 ms frames. The Figure 7.3 shows how audio is processed every 96 ms of new audio data. Initially, a lightweight noise detection mechanism filters out segments that primarily consist of a

¹⁸<https://developer.android.com/training/package-visibility>

¹⁹<https://developer.android.com/guide/components/activities/background-starts>

²⁰<https://medium.com/make-android/start-android-activity-or-service-from-the-background-689f49167d80>

constant, silent background, reducing the workload for subsequent neural models. Only frames that exhibit some form of noise activity are analyzed by the Silero voice activity detector (VAD). The VAD acts as a gate, ensuring that only frames containing human speech move into the green area of the wake word detection pipeline, where the actual detection of the wake word occurs.

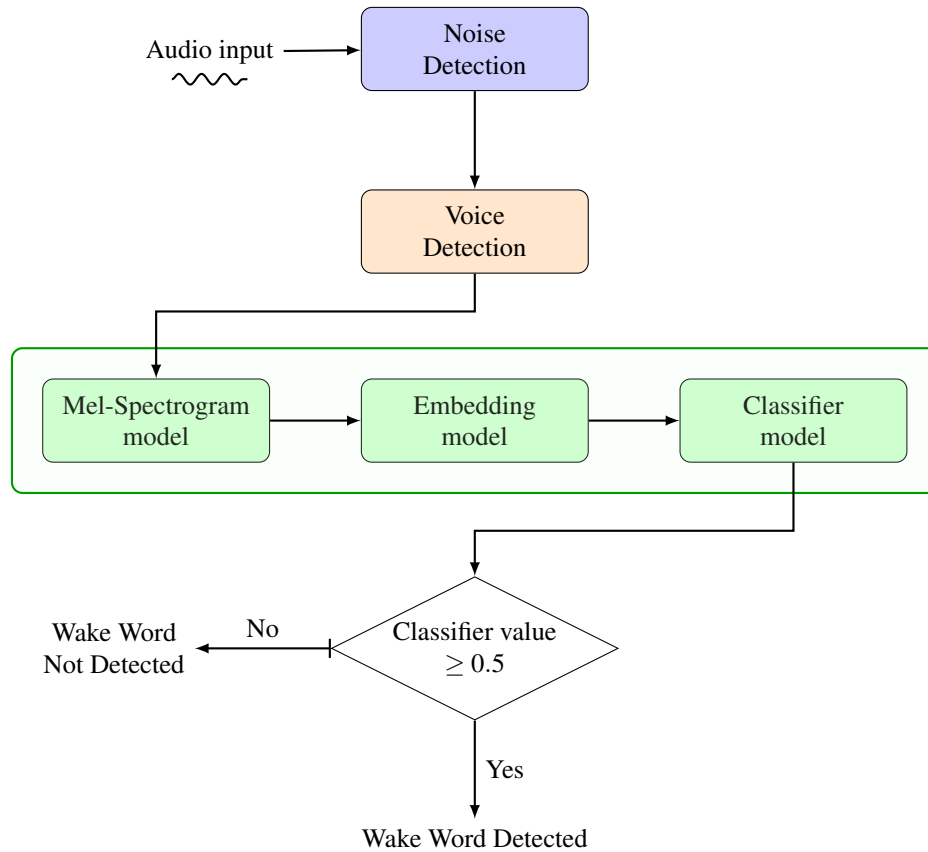


Figure 7.3: Data-flow diagram of the real-time wake-word detection pipeline.

The green area represents the three-stage pipeline for wake word detection. It begins with the computation of the mel-spectrogram from the input audio frame using a mel-spectrogram neural network model. Once the output tensor is produced by this first model, it adds new data to a ring buffer. The size of this ring buffer is designed to match the input tensor of the next model in the pipeline, which is the embeddings model.

The input tensor shape for the embeddings model is $[-1, 76, 32, 1]$, so we create a ring buffer with a size of 2432. In each iteration, the ring buffer receives a list of new mel-spectrograms. This new data is appended to the end of the ring buffer while the oldest mel-spectrograms are removed, ensuring that the buffer maintains the required size for the input tensor of the embeddings model.

The same method is employed to calculate the vector of embeddings from the previous ring buffer. Since the input tensor for the classifier model requires a matrix of 16 embedding vectors, it is necessary to create an additional ring buffer. This new buffer will store consecutive embedding vectors, ensuring we have the appropriate data ready for the classifier model's input tensor. Finally, the classifier model will compute the probability of detecting the wake word during this iteration.

based on the ring buffer of audio embeddings. If the resulting floating-point value exceeds 0.5, the model determines that the wake word has been detected. This threshold of 0.5 is the one recommended by the OWW library.

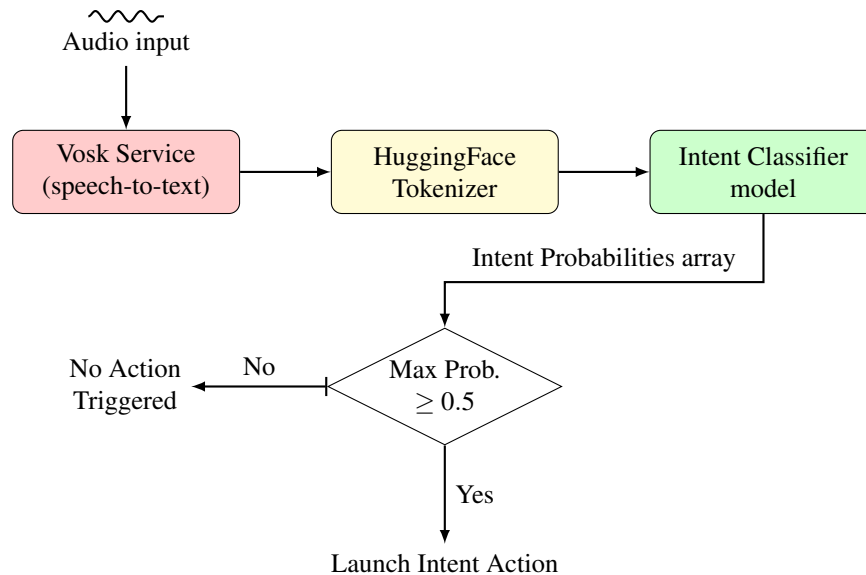


Figure 7.4: Data-flow diagram of the voice command recognition pipeline.

Once the wake word is detected, the application enters a new phase where it utilizes a background service using the Vosk library for speech-to-text recognition. As presented in Figure 7.4, it records audio until it detects human speech, which is transcribed in real-time. When a pause after the speech is detected, the service retrieves the transcribed text. This text is then sent to the tokenizer, which extracts the relevant data for the intent classification model. The model evaluates this data and returns a vector of floating-point values corresponding to each intent it has been trained on. The application selects the intent with the highest probability, and if that probability exceeds 0.5, it validates the voice command.

Once a voice command is validated, it triggers a specific action or intent. For example, users can issue voice commands to play music, which will open the music player and start playback, or to launch the weather application.

7.8 Summary

This is a fully offline Android application designed to execute on-device machine learning models for wake word detection and voice command recognition. Audio is captured at a frequency of 16 kHz and is pre-filtered to eliminate silence and environmental noise that does not qualify as human speech. For wake word detection, we utilize three pre-trained models (mel-spectrogram, embeddings, and classifier) extracted from the OpenWakeWord library, connecting them using ring buffers. The audio stream is processed with a sliding window of 96 ms to ensure that the

wake word is not missed. We use JNI to bridge Kotlin with a C-based TensorFlow Lite runtime, allowing all models to share the same lightweight engine.

Once the confidence value for the wake word exceeds 0.5, we employ the Vosk library to transcribe the spoken voice command in real-time. We also utilize a small intent classifier model, which has been converted to TFLite format, with the transcribe text tokenized using the Deep Java Library. This intent model calculates a confidence level across a set of 60 intents. However, for demonstration and testing purposes, we currently only support two actions: playing music and opening the weather app. When the maximum confidence level exceeds 0.5, the action corresponding to the voice command is triggered by launching the relevant application.

While we considered GPU offloading and on-device training, we ultimately did not use these options due to the pre-trained models containing unsupported operations and the absence of input tensors for on-device training. In our comparison of model execution using the Java and C APIs from TensorFlow Lite, we found that performance was very similar. Further investigation revealed that both APIs use the same native libraries, which explains the comparable performance.

Chapter 8

Experiments and Results

8.1 Introduction

This chapter presents the empirical evaluation of the wake-word detection pipeline developed for our proposed application. We start by describing the experimental setup, justifying our choice of a physical test device, and outlining the steps taken to maintain consistent operating conditions. Next, we outline three representative acoustic scenarios: an overnight bedroom, a typical office day, and an in-car travel environment. We explain how the pipeline classifies each 96 ms audio frame as either `silence`, `noise`, `voice`, or `wake_word`. For each scenario, we visualize the activity using stacked heatmaps and summarize the proportions of audio classes with pie charts, providing a clear view of how the classifier performs in realistic environments.

8.2 Experimental Setup

We had two options for selecting a testing device for our application: an emulated device on the host PC or a physical device. We chose to use a physical device for testing and evaluating our application. Since the application interacts with the external environment through the microphone sensor, we believe it is more practical to test on a real device. Emulated devices can present issues with audio input, as they typically gather data from the host device or an external microphone. This could negatively impact the application’s performance and introduce unnecessary complications.

The Android device used is the same one where we conducted tests on the Sieve of Eratosthenes, Matrix Multiplication, and Fourier Transform algorithms, as described in Section 5.3.2. In the testing environments, we maintained consistent internal conditions for the device. This included being connected to a Wi-Fi network, setting the screen brightness to 50%, disabling Bluetooth, and having no SIM card on the device.

8.3 Testing Environments

In this section, we outline the tests conducted on our application and the environments in which they were performed. For each test, we captured the ambient sound using the device and recorded the audio frames or buffers processed. Each frame was classified as containing noise, voice, wake word, or silence. It is important to note that if an audio frame contains voice, it is also classified as noise. Similarly, a wake word can only be detected in audio classified as containing voice and noise beforehand.

This concept is illustrated in Figure 8.1, which represents an audio recording taken over three hours in a living room with the TV turned on. During this time, one person triggered the wake word twice. The chart displays a stacked heatmap representing noise activity at the bottom, voice activity above it, and wake word activity at the top.

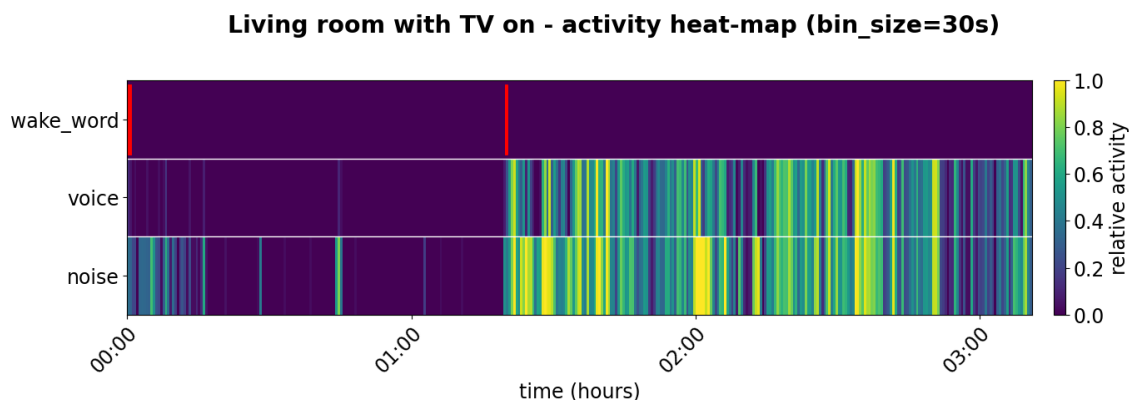


Figure 8.1: The heatmap activity of the wake word pipeline was recorded over three hours in a living room where the TV was on. During this time, one person said the wake word twice. The heatmap is divided into segments of 30 seconds each (`bin_size=30s`), allowing for the calculation of relative activity within those intervals.

The wake word layer differs from the other two as it is represented solely in red and is not a heatmap like the layers below. It shows a red strip indicating where the wake word was detected on the timeline. We needed to approach this wake word layer differently because its activity is minimal compared to the other layers. Consequently, it is difficult to detect in the heatmap, as wake word activity occurs in only 0.02% of the audio frames, as presented in Figure 8.2.

An important note regarding the interpretation of these graphics relates to the pie chart in Figure 8.2. As mentioned earlier, voice is also considered a type of noise. However, when representing the percentages of silence, noise, voice, and wake word detections, we define noise strictly as audio events that are classified as noise but not as voice. This approach ensures that the percentages accurately reflect the audio activity without overlapping information.

In the following subsections, we will present additional testing environments, along with heatmaps that illustrate various types of audio, and pie charts for the same period. We will also measure the device's energy consumption throughout the entire testing process in the upcoming tests. Energy consumption will be measured using the device's battery percentage capacity.

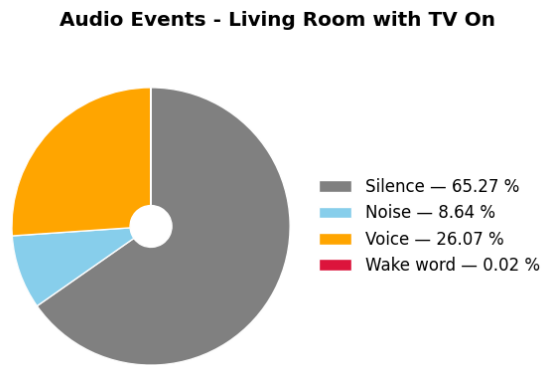


Figure 8.2: Pie chart representing the percentage of each audio event in the previous three-hour recording from the living room.

Although Android Studio has a tool called Power Profiler, it is only available for measuring energy consumption on devices that support On Device Power Rails Monitor (ODPM). This feature is limited to Google devices, specifically the Pixel 6 and later Pixel models running Android 10 and higher. Since we do not have access to these devices, we compared battery capacity values using the device settings instead.

8.3.1 Quiet Bedroom Overnight

This next test was conducted during nine hours in a quiet bedroom on one night. The test began with the device at 85% battery, and when we finished using the application, the battery level was still at 73%. This resulted in a total battery usage of 12%.

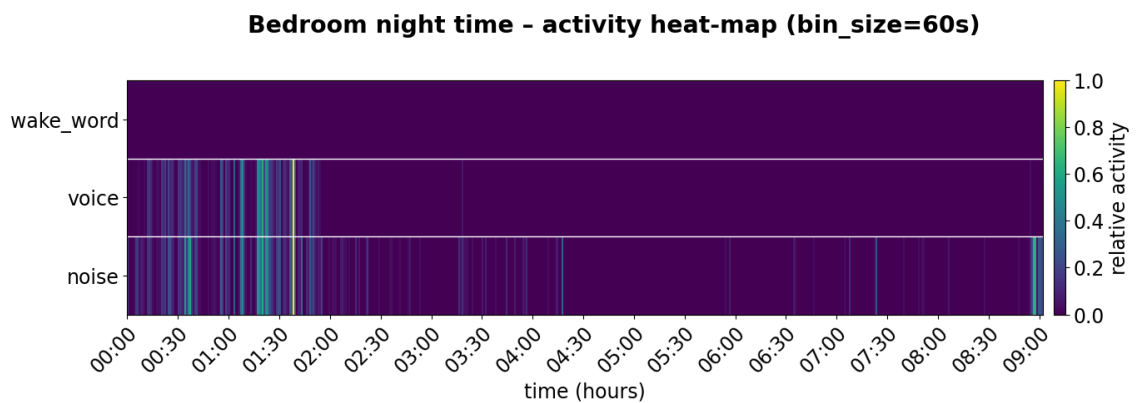


Figure 8.3: Heatmap displaying the activity of wake word triggers in a bedroom over nine hours overnight.

In Figure 8.3, the audio activity heatmap indicates that the recorded audio predominantly consisted of silence. The only exception was during the first two hours, when some noise was detected and classified as human speech. Following that period, the audio remained largely silent, with minimal noise activity, which is typical during the night.

The pie chart in Figure 8.4 also reflects that the audio was mostly silent during the night and shows that the wake word was not detected at any time.



Figure 8.4: Pie chart showing the percentage of different audio events during nine hours in a bedroom overnight.

8.3.2 Office Day

The test was conducted over the course of one day at the company's office. We started with the application initialized at 83% battery on the device, and it concluded at 68%. The total battery usage during the nearly nine hours spent in the office was 15%.

In Figure 8.5, we observe that there was a higher level of audio activity classified as voice at the beginning and the end of the day. Additionally, upon examining the noise layer in the heatmap more closely, we can see that around the three-hour mark, there was very little activity recorded. This coincides with lunchtime, a period when most people are out of the office.

As anticipated, no activity was detected on the wake word layer of the heatmap during the day, since no one said the wake word.

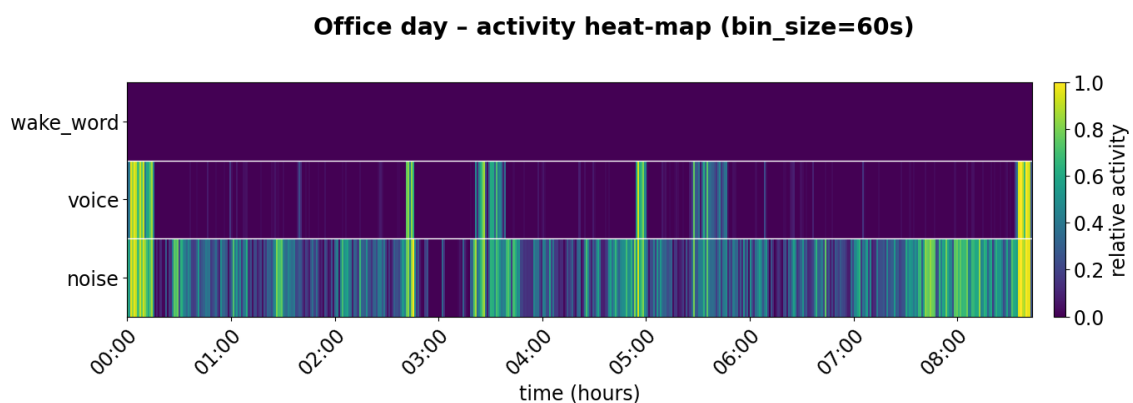


Figure 8.5: Heatmap displaying the activity of wake word triggers over almost nine hours at the company's office.

In Figure 8.6, we can also observe that there is 0% of audio events detected as a wake word. Most of the recorded audio was silence, with only 8.48% consisting of voice.



Figure 8.6: Pie chart showing the percentage of different audio events during almost nine hours at the company's office.

8.3.3 Car Travel

The third test we conducted took place during a car trip lasting nearly an hour and a half, consuming 2% of the battery. The environment was quite noisy throughout the journey, primarily due to the sounds emitted by the car and the music playing inside. As illustrated in Figure 8.7, the noise layer consistently displays a bright yellow color, indicating maximum activity at all times. This means that within all audio bins of 15 seconds, each audio frame of 96ms processed during this 15-second period was identified as noise. The continuous noise pattern ends just before the audio recording stops, which coincides with the moment the car came to a halt. The voice activity detected was mainly a result of the music being played in the car. No activity was detected in the wake word layer during this brief car trip, as expected.

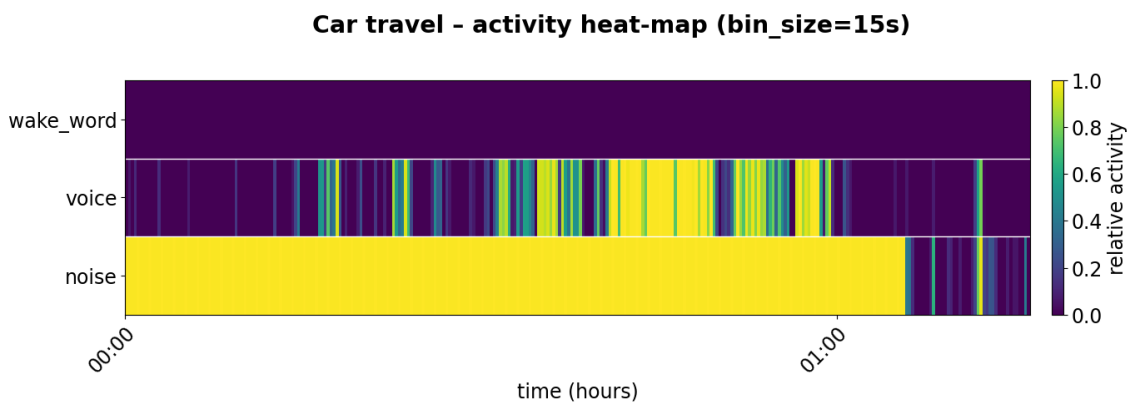


Figure 8.7: Heatmap activity of wake word pipeline triggers.

To validate the findings presented in the heatmap, we can refer to the pie chart in Figure 8.8, which indicates that the audio primarily consisted of noise, accounting for 57.63% of the total. As explained earlier in Section 8.3, this percentage solely pertains to the audio classified as noise, excluding any voice samples. When we combine the percentages of both noise and voice, the total reaches 88.25%. This result aligns well with our observations of the noise layer in the heatmap

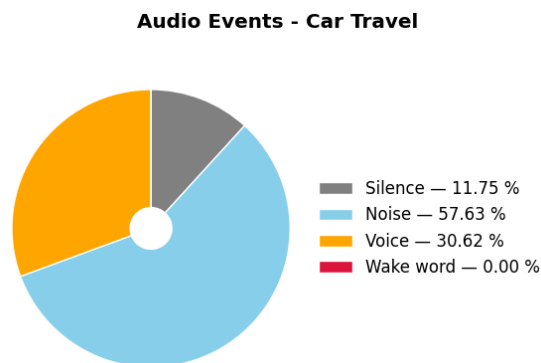


Figure 8.8: Pie chart displaying the percentage of different audio events during a car trip.

shown in Figure 8.7.

8.4 Memory Usage

The Figure 8.9 shows how the application manages memory usage. We can observe a significant increase in memory consumption compared to when the application consisted solely of the wake word detection system, presented in Figure 7.2. After implementing voice command recognition using Vosk for speech-to-text, along with a new model for intent classification and a tokenizer for its input, the memory usage rose considerably. This increase is primarily due to the speech-to-text model, which has a size of 67.6 MB.

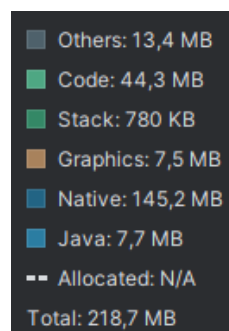


Figure 8.9: Memory usage when the application is running as a background service.

The memory consumption of 218.7MB refers to when the application is running in a background service, which means that activities associated with voice commands cannot be launched. However, the `play_music` voice command can still activate music playback even if the music application is not visible on the screen, as it utilizes a Media action to play the music. It is important to note that memory consumption increases in the graphics category when the application is visible.

8.5 Energy Consumption Analysis

To evaluate the energy consumption of the wake word detection pipeline, we conducted a comprehensive set of tests in various environments. This included the previously mentioned environments from Section 8.3, as well as new settings such as recording a TV while playing movies, capturing audio in a living room with different levels of background noise and voice activity, recording YouTube playback featuring continuous human speech, and capturing the playback of purely instrumental music.

For each test run, we logged the following variables:

1. the test duration d (s),
2. the battery drop ΔBatt (%-points),
3. the share of time labelled as *silence* (S), *noise* (N) and *voice* (V).

Since we collect data using different scales, ranging from battery usage to audio activity values, it is necessary to standardize this information across various test runs. This is important because the same percentage of audio activity in one test run may represent different information compared to another run if their durations differ. By standardizing these variables, we ensure that our test data is correctly formatted, allowing the OLS model to accurately predict battery usage of each case. Therefore, we multiplied each audio activity percentage by the test run duration, resulting in the following variables:

$$\text{dur_x_noise} = d_{\text{hours}} \times N, \quad \text{dur_x_voice} = d_{\text{hours}} \times V, \quad \text{dur_x_silence} = d_{\text{hours}} \times S.$$

These terms have the unit “% hours”, which means 30% noise for 2h is represented as 0.6 %h. Therefore, using these three composite predictors, while omitting the raw duration and the raw percentages, we can eliminate perfect collinearity. The original variables satisfy the equation $S + N + V = 100\%$. Including these variables alongside the duration would result in one column being an exact linear combination of the others, leading to a singular regression matrix. To address this, we condense the test duration by multiplying it by the audio activity percentage. This approach allows us to represent each class with a single measure while maintaining all the information about both the test length and the audio activity mix. Additionally, it ensures that the predictors remain sufficiently independent, enabling stable estimates of the coefficients.

After conducting eleven different tests in various environments, we decided to create a statistical model using the statsmodels Python library. We employed the Ordinary Least Squares (OLS) model to predict battery consumption for each test run, based on the variables mentioned earlier.

After comparing the actual battery usage to the predicted values, we conducted a linear regression analysis (Figure 8.10) and calculated the R-squared value. This assessment was aimed at

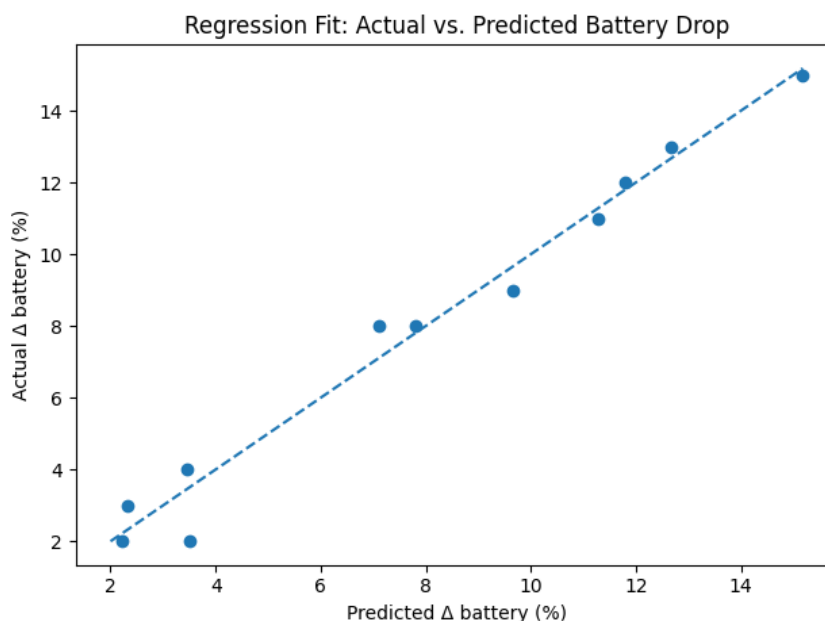


Figure 8.10: Chart illustrating linear regression for predicted battery consumption.

determining whether battery consumption correlates with variations in audio activity and its duration. The regression achieved an $R^2 = 0.9691$, indicating that the three predictors together explain approximately 96% of the variation in battery loss across the eleven tests.

Table 8.1: Estimated coefficients for the regression.

Predictor	Coefficient β
<code>dur_x_silence</code>	0.0124
<code>dur_x_noise</code>	0.0220
<code>dur_x_voice</code>	0.0381

We also calculated the coefficients for each predictor, as shown in Table 8.1. These coefficients indicate the additional battery consumption expected for every extra unit of "% hour" that the pipeline spends in a specific audio class, while keeping all other classes constant. For example, a coefficient of 0.0381 pp/%h for `dur_x_voice` means that one hour of pure voice is predicted to drain an additional 0.0381 percentage points of battery.

Speech is the most power-intensive class, followed by environmental noise at 0.0220 pp/%h. Silence incurs a measurable overhead of 0.0124 pp/%h, reflecting the baseline cost of maintaining the wake-word pipeline, even during inactivity. All three effects are positive, supporting the idea that longer exposure to any category increases energy use. However, the intensity of the content varies, suggesting that the type of content is nearly as important as its duration. As anticipated based on the application implementation and the complexity of the different pipeline stages, the differences between audio types align with our expectations.

8.6 Summary

The wake word detection system demonstrated reliable performance across three different scenarios. In the quiet-bedroom test, nearly every frame was classified as silence, with no false triggers detected. Similarly, the office-day and car-travel recordings showed that the model accurately distinguished between voices and background noise, avoiding any incorrect wake-word activations. In the living room experiment, intentional activations were detected precisely at the expected timestamps, confirming the system’s end-to-end accuracy for genuine wake-word events.

The energy study demonstrates that always-on inference is feasible on standard hardware. A linear regression analysis of eleven diverse test runs resulted in an $R^2 = 0.9691$, indicating that a simple combination of the duration-weighted shares of silence, noise, and voice accounts for nearly all observed variations in battery drain. The coefficients of 0.0124 percentage points per hour for silence, 0.0220 percentage points per hour for environmental noise, and 0.0381 percentage points per hour for voice quantify the incremental battery costs associated with each acoustic category. This reveals that human speech consumes roughly three times more battery than baseline silence.

It is important to note that we did not test or evaluate the voice command recognition process that occurs after the wake word is detected. This part of the application relies entirely on the performance of the Vosk library’s text-to-speech model, which can vary depending on different environments and the presence of background noise. Since this tool was not part of our work, we chose not to analyze it further. However, it is worth mentioning that the entire pipeline functions as expected from the moment the wake word is detected until the specific application, associated with the triggered voice command, is launched.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This section outlines the main conclusions drawn during the development of this dissertation. We specifically examined the use of the Android NDK with C/C++ in comparison to Java/Kotlin. Our analysis revealed that using the Android NDK does not always guarantee improved overall performance. Any potential performance gains are highly dependent on the specific problem being addressed and cannot be assumed simply by opting for the Android NDK.

Additionally, we discussed several limitations and challenges we faced as non-privileged Android developers. We encountered various compatibility issues related to the TensorFlow Lite models we chose, as well as the model conversion tools we utilized.

9.1.1 Algorithms on Android NDK

This dissertation began with an exploration of the Sieve of Eratosthenes algorithm, followed by studies on Matrix Multiplication and the Fast Fourier Transform (FFT). One key takeaway regarding the Sieve algorithm was the impact of memory usage and allocation limits when using the Android Runtime (ART) with Java/Kotlin. In contrast, native development with the Android NDK allowed for significantly higher memory limits, approaching the hardware's maximum capacity. Notably, the execution times did not differ significantly between the Kotlin and C implementations.

Further analysis of more computationally intensive algorithms, such as Matrix Multiplication and FFT, revealed more substantial performance improvements in execution times when the clang compiler was configured to generate optimized machine code using optimization flags, such as `-O3`. This configuration enabled enhanced performance by leveraging vector instructions for floating-point operations.

9.1.2 Android Runtime limitations

During the development of our proof of concept and the proposed solution for wake word detection and voice recognition, we encountered several limitations stemming from the lack of control over the internal behavior of the Android Runtime (ART). Although there are some flags we can supposedly set, like increasing the memory allocation limit, we were never able to utilize them successfully. This may be due to the fact that these options are not available on production devices. Regardless, this situation reduces the control that mobile developers have over managing the device's resources for the sake of system security and integrity.

This lack of control over memory usage when developing in Java/Kotlin within ART can be mitigated by shifting our focus to native development using the Android NDK. With native development, we can compile our C/C++ code with the desired configuration, allowing it to execute without excessive oversight from the operating system. This flexibility enables applications to utilize the maximum available free memory.

9.1.3 TensorFlow Lite models

While experimenting with various models in TFLite format, we encountered several compatibility issues, particularly when attempting to execute them on Android devices by offloading tasks to the GPU. These complications arose because the internal configurations of the models we used were not suitable for the GPU delegate. Upon reflection, it may be more beneficial to develop our own model with an internal configuration specifically designed for GPU offloading, as well as customized layers for on-device training and audio streaming. However, since this was not the primary focus of the dissertation and the effort required would have been substantial, we chose not to pursue this approach of creating and training new models. Instead, our attention centered on pre-trained models that we found, which had been fine-tuned using large and diverse datasets and showed promising evaluation results from both the creators and our empirical tests.

The models we discovered that were not in TFLite format presented additional complications and challenges during conversion, even when using appropriate tools. This experience led us to conclude that there is a certain exclusivity regarding the compatibility of models within the machine learning framework in which they were created and trained.

9.1.4 Android API limitations

The Android operating system offers the Android API as the primary means of communication with the Android Framework and, by extension, the operating system itself. However, a portion of this API, known as the System API, is not fully accessible to regular Android developers. As illustrated in Figure 9.1, Access to the System API is restricted to Original Equipment Manufacturers (OEMs) and their partners.

While certain features, such as access to the Digital Signal Processor (DSP), could enhance our implementation of a wake word detection system, we cannot utilize them because we do not have the capability to sign our applications as a manufacturer or elevate their privileges using

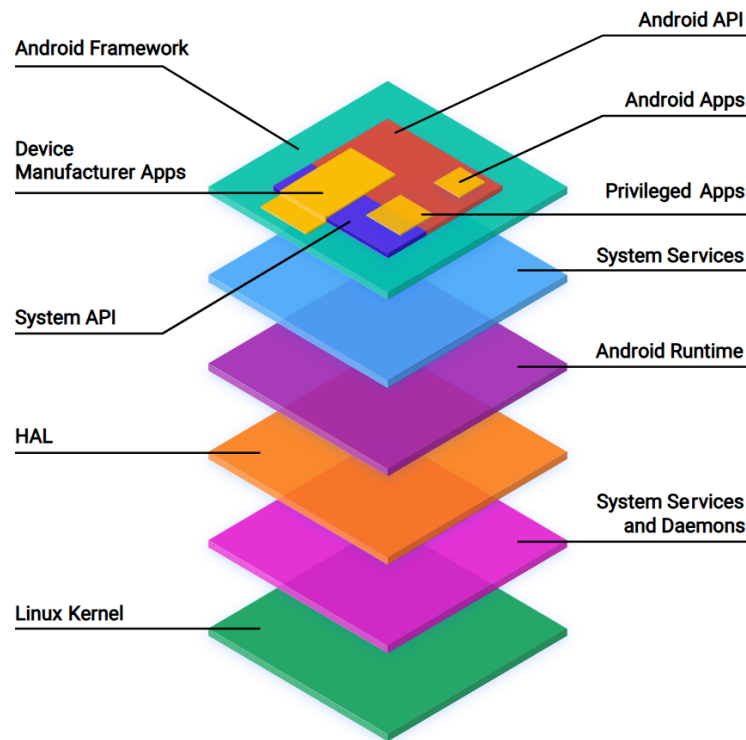


Figure 9.1: Android stack diagram. [Source: Android Open Source Project, <https://source.android.com/docs/core/architecture>.]

any official manner. Although it might be technically feasible to implement this on a rooted device, where we could potentially run the application with root privileges, this approach is still not practical for a widespread deployment.

9.1.5 Notes on Objectives

It is important to note that the objective of this dissertation was to focus not only on low-level development for mobile devices but also for wearable devices. Initially, we concentrated on mobile devices due to their greater accessibility and practicality for testing. However, as the development process progressed, our attention gradually shifted away from wearable devices. Although, we believe that our performance optimization findings for Android may also be applicable to Wear OS devices, considering their scale and resource constraints.

Additionally, we aimed to develop a library that would abstract the complexity of the solution we proposed, as mentioned in Section 1.3. However, this objective began to diverge from the overall direction of the dissertation. This shift occurred because the TensorFlow Lite inference engine utilized in our solution, both with the Java API and the C API, employed native code implemented using the Android NDK. Consequently, the performance of the inference engine was quite similar in both approaches. Nevertheless, this allowed us to demonstrate the relevance of our objective, as major development tools like TensorFlow also utilize the Android NDK to provide

a wrapper for loading tensor data and execute machine learning models on the native inference engine.

Finally, although our goal was to develop our own models, as stated in Section 4.2, we ultimately did not pursue this path. We discovered pre-trained models that performed well and fit our needs, and since machine learning was not the main focus of our study, we decided to proceed with these pre-trained models. However, as previously mentioned in Section 9.1.3, this decision led to certain limitations.

9.2 Research Questions Revisited

In this section, we revisit the previously outlined research questions and provide insights based on findings from our experiments and implementations.

1. **RQ: How can low-level programming like C/C++ be used to optimize performance and energy efficiency in mobile and wearable applications compared to high-level programming approaches?** A: One way that low-level programming with C/C++ can enhance execution performance is through the use of floating-point operations. By utilizing the Android NDK, we can enable compiler optimizations that generate machine code using vector instructions. This benefit is not assured when developing in Java/Kotlin. While the Android Runtime (ART) does support auto-vectorization, it cannot ensure that vectorization will take place every time it is feasible. Due to limitations in our testing equipment, we were unable to draw valuable conclusions regarding energy efficiency on these algorithms.
2. **RQ: Which types of algorithms or operations are best suited for optimization through low-level programming in mobile and wearable devices?** A: Algorithms that require floating-point operations, such as matrix multiplication and digital signal processing algorithms like the FFT, are ideal candidates for development in low-level languages using the Android NDK. This approach can improve performance compared to Java/Kotlin implementations running on ART.
3. **RQ: What algorithms have been successfully implemented using the Android NDK so far, and in what scenarios are these implementations most effective?** A: Our investigation revealed that many libraries, including TensorFlow Lite, which we utilized as the inference engine for running our machine learning models, have low-level code implementations deep within their source code. These native implementations are accessed through the API of the library using the JNI, which serves as a bridge between the native library and a higher-level Java implementation.
4. **RQ: What are good practices and design patterns for using the Android NDK when optimizing applications for mobile and wearable devices?** A: When discussing best practices and design patterns, we did not delve deeply into specific details. However, we noted that a common practice is to abstract the use of native libraries by implementing wrappers,

which make low-level implementations easily accessible. The JNI is useful for developing these wrappers, as it enables the communication between the native code and the application written in Java/Kotlin. Additionally, TensorFlow offers a concise set of helper functions that utilize JNI to facilitate this interoperability.

9.3 Future Work

Regarding future work on this project, there are several potential improvements to explore. One key area is implementing GPU inference on-device, which would significantly enhance model inference time and reduce energy consumption. Additionally, incorporating on-device training could personalize voice models for individual users, potentially enhancing the user experience and increasing model accuracy. However, as mentioned in Section 9.1.3, these solutions require extra efforts for model adaptation and re-training.

Another promising avenue is the concept of a "model-less model", introduced by Zhou et al. [39]. This approach aims to transform a machine learning model into a customized executable program, similar to compiling a C program into an executable file. Beyond reducing model size, the authors suggest that this method can accelerate model inference by 24.3% and decrease memory consumption by 36.0% on ARM64 platforms.

To distribute the developed solution to other devices, an important adjustment would be to implement an adaptive silence threshold, as the threshold we defined is specific to our device's microphone. Calculating a threshold based on a microphone test requested from the user could help address this issue. Alternatively, we could explore other methods to determine what constitutes silence based on decibel values.

References

- [1] openwakeword. Available at <https://github.com/dscripka/openWakeWord> (20/04/2025). Cited on page 13.
- [2] Zrar Kh. Abdul and Abdulbasit K. Al-Talabani. Mel Frequency Cepstral Coefficient and its Applications: A Review. *IEEE Access*, 10:122136–122158, 2022. Conference Name: IEEE Access. Cited on page 16.
- [3] Bastian Bloessl, Lars Baumgärtner, and Matthias Hollick. Hardware-Accelerated Real-Time Stream Data Processing on Android with GNU Radio. In *Proceedings of the 14th International Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization*, WiNTECH '20, pages 103–109, New York, NY, USA, 2020. Association for Computing Machinery. Cited on pages 8 and 13.
- [4] Fares Bouaffar, Olivier Le Goaer, and Adel Nouredine. PowDroid: Energy Profiling of Android Applications. In *2nd International Workshop on Sustainable Software Engineering (SUSTAINSE)*, Melbourne, Australia, November 2021. Cited on page 11.
- [5] Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. Development frameworks for mobile devices: a comparative study about energy consumption. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 191–201, Gothenburg Sweden, May 2018. ACM. Cited on pages ix, 7, 9, 10, and 13.
- [6] Marco Couto, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. GreenDroid: A tool for analysing power consumption in the android ecosystem. In *2015 IEEE 13th International Scientific Conference on Informatics*, pages 73–78, Poprad, Slovakia, November 2015. IEEE. Cited on page 11.
- [7] Mathieu Febvay. Low-level Optimizations for Faster Mobile Deep Learning Inference Frameworks. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 4738–4742, Seattle WA USA, October 2020. ACM. Cited on pages 12 and 38.
- [8] Google for Developers. Android ndk. Available at <https://developer.android.com/ndk/guides> (25/02/2025). Cited on page 2.
- [9] Google for Developers. Android sdk tools. Available at <https://developer.android.com/tools> (25/02/2025). Cited on page 2.
- [10] Google for Developers. Multi-threading & callbacks primer. Available at <https://developer.android.com/courses/extras/multithreading> (03/03/2025). Cited on page 20.

- [11] Google AI for Developers. Litert in google play services. Available at https://ai.google.dev/edge/litert/android/play_services (16/04/2025). Cited on page 9.
- [12] Bin Hu, Yan Wang, Jerry Cheng, Tianming Zhao, Yucheng Xie, Xiaonan Guo, and Yingying Chen. Secure and Efficient Mobile DNN Using Trusted Execution Environments. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS '23*, pages 274–285, New York, NY, USA, July 2023. Association for Computing Machinery. Cited on pages 7, 11, and 14.
- [13] Alpha Cephei Inc. Vosk is a speech recognition toolkit. Available at <https://alphacephei.com/vosk/> (20/04/2025). Cited on page 13.
- [14] Picovoice Inc. Picovoice leopard. Available at <https://picovoice.ai/platform/leopard/> (20/04/2025). Cited on page 13.
- [15] Picovoice Inc. Picovoice porcupine. Available at <https://github.com/Picovoice/porcupine> (20/04/2025). Cited on page 12.
- [16] Netflix Inc. Netflix Open Source Platform. Threadsafebitset. Available at <https://github.com/Netflix/hollow/blob/master/hollow/src/main/java/com/netflix/hollow/core/memory/ThreadSafeBitSet.java> (11/03/2025). Cited on page 22.
- [17] Ki-Cheol Son and Jong-Yeol Lee. The method of android application speed up by using NDK. In *2011 3rd International Conference on Awareness Science and Technology (iCAST)*, pages 382–385, Dalian, China, September 2011. IEEE. Cited on pages 7, 8, and 13.
- [18] Alexander A. Kropotin. Concurrentbitset. Available at <https://github.com/ololx/concurrent-bitset> (11/03/2025)). Cited on page 22.
- [19] Ming-Yi Lai, Chia-Yu Sung, Jenq-Kuen Lee, and Ming-Yu Hung. Enabling Android NNAPI Flow for TVM Runtime. In *Workshop Proceedings of the 49th International Conference on Parallel Processing, ICPP Workshops '20*, pages 1–8, New York, NY, USA, 2020. Association for Computing Machinery. Cited on pages 7, 12, and 14.
- [20] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for Android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014*, pages 46–53, New York, NY, USA, June 2014. Association for Computing Machinery. Cited on pages 10 and 13.
- [21] Vanessa Martins, Lazaro Queiroz, Aron Brito, Davi Barbosa, Tiago Silva, Ana Patrícia F. Magalhaes, Josemar Souza, and Jorge Campos. COMPARING POCKETSPHINX AND VOSK RECOGNITION IN HUMAN SPEECH DECODING. In *Proceedings of IV Brazilian Humanoid Robot Workshop (BRAHUR) and V Brazilian Workshop on Service Robotics (BRASERO)*, volume 1, page 1, July 2021. Cited on page 46.
- [22] José Alexandre Nalon. Fast fourier transform. Available at <https://github.com/jnalon/fast-fourier-transform> (20/03/2025). Cited on page 29.
- [23] Wellington Oliveira, Renato Oliveira, and Fernando Castor. A Study on the Energy Consumption of Android App Development Approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 42–52, Buenos Aires, Argentina, May 2017. IEEE. Cited on pages 7, 9, and 13.

- [24] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. Static Analysis of JNI Programs via Binary Decompilation. *IEEE Transactions on Software Engineering*, 49(5):3089–3105, May 2023. Conference Name: IEEE Transactions on Software Engineering. Cited on pages 7, 11, and 13.
- [25] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017*, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery. Cited on pages 1 and 18.
- [26] Ponomarev E. OPTIMIZING ANDROID APPLICATION PERFORMANCE: MODERN METHODS AND PRACTICES. *Sciences of Europe*, (149):61–63, September 2024. Cited on pages 7, 10, and 13.
- [27] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. Cited on page 13.
- [28] Apple ML Research. Hey siri: An on-device dnn-powered voice trigger for apple’s personal assistant. Available at <https://machinelearning.apple.com/research/hey-siri> (25/04/2025). Cited on page 40.
- [29] CMU research. Cmusphinx - pocketsphinx. Available at <https://cmusphinx.github.io/> (20/04/2025). Cited on page 12.
- [30] Antonio Ruggia, Andrea Possemato, Savino Dambra, Alessio Merlo, Simone Aonzo, and Davide Balzarotti. The Dark Side of Native Code on Android. *ACM Trans. Priv. Secur.*, January 2025. Just Accepted. Cited on pages 7, 11, and 13.
- [31] Oleg Rybakov, Natasha Kononenko, Niranjana Subrahmanya, Mirko Visontai, and Stella Laurenzo. Streaming keyword spotting on mobile devices. In *Interspeech 2020*, pages 2277–2281, October 2020. arXiv:2005.06720 [eess]. Cited on page 40.
- [32] M.F.A. Sayeedi, J.F. Deepti, A.M.I.M. Osmani, T. Rahman, S.S. Islam, and M.M. Islam. A Comparative Analysis for Optimizing Machine Learning Model Deployment in IoT Devices. *Applied Sciences (Switzerland)*, 14(13), 2024. Cited on pages 7, 13, and 14.
- [33] Ajay Shiv Sharma and Rahul Bhalley. ASR — A real-time speech recognition on portable devices. In *2016 2nd International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Fall)*, pages 1–4, September 2016. Cited on page 12.
- [34] Daniel Silva and Fábio Gaspar. The sieve of erastosthenes. Available at <https://github.com/Dannyps/feup-cpar2> (07/03/2025). Cited on page 24.
- [35] Joren Six, Olmo Cornelis, and Marc Leman. TarsosDSP, a Real-Time Audio Processing Framework in Java. In *Proceedings of the 53rd AES Conference (AES 53rd)*, 2014. Cited on page 41.
- [36] Silero Team. Silero vad: pre-trained enterprise-grade voice activity detector (vad), number detector and language classifier. <https://github.com/snakers4/silero-vad>, 2024. Cited on page 43.

- [37] Ji Wang, Jianguo Zhang, Weidong Bao, Xiaomin Zhu, Bokai Cao, and Philip S. Yu. Not just privacy: Improving performance of private deep learning in mobile cloud. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 2407–2416, New York, NY, USA, 2018. Association for Computing Machinery. Cited on page 2.
- [38] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1137–1150, New York, NY, USA, 2018. Association for Computing Machinery. Cited on pages 7, 11, and 12.
- [39] Mingyi Zhou, Xiang Gao, Pei Liu, John Grundy, Chunyang Chen, Xiao Chen, and Li Li. Model-less Is the Best Model: Generating Pure Code Implementations to Replace On-Device DL Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 174–185, New York, NY, USA, 2024. Association for Computing Machinery. Cited on page 67.