UMEÅ UNIVERSITY

Kotlin Multiplatform vs Native Development: A Performance Analysis for Android and iOS

Eric Örnkloo

Eric Örnkloo
Spring 2025
Degree Project in Interaction Technology and Design, 30 credits
Supervisor: Timotheus Kampik
External Supervisor: Abdulsalam Aldahir
Examiner: Ola Ringdahl
Master of Science Programme in Interaction Technology and Design, 300 credits

Abstract

This thesis investigates the performance differences between Kotlin Multiplatform (KMP) and native implementations across the mobile platforms Android and iOS. KMP is a cross-platform framework designed to enable code sharing across different platforms, including mobile, web, and desktop applications. The objective is to identify scenarios where KMP outperforms or underperforms compared to native implementations, analyze factors contributing to performance variations, and determine the best use cases for the adoption of KMP as a framework. The study uses a controlled experimental approach, testing both KMP and native implementations across ten different devices (five Android and five iOS devices) to ensure consistency in the results.

The findings reveal substantial platform-specific variations in performance. KMP displays superior performance in image rendering tests across both platforms (58% to $\sim$83% faster) and shows noteworthy advantages on iOS for startup times ($\sim$57% faster) and CPU (Central Processing Unit) performance ($\sim$81% faster). However, KMP exhibits slower performance on Android for startup times ($\sim$44% slower) and CPU operations ($\sim$26% slower). The study also reveals substantial differences in binary application size. While KMP shows no particular differences on Android, the iOS implementations ended up with $\sim$33 MB and $\sim$1 MB, respectively, for KMP and native implementations.

The results suggest that KMP is particularly well-suited for applications requiring a consistent user interface, those with significant image rendering requirements, and projects prioritizing development efficiency through code sharing. However, the choice to adopt KMP should be made with careful consideration of platform-specific requirements and performance priorities.

Keywords: Kotlin Multiplatform, Cross-Platform Development, Performance Analysis, Mobile Development, Android, iOS

Acknowledgments

I would like to express my appreciation and gratitude to my supervisor at THNX **Abdulsalam Aldahir** for helping me with technical questions and always being ready to support, no matter how many times I interrupted him in his work. A big thanks to **THNX Innovation** for taking me in and letting me do my master's thesis with them. I would also like to extend my thanks to my university supervisor **Timotheus Kampik**, who consistently provided me with excellent advice and tips for writing a good thesis. Lastly, I would like to thank my family for always supporting me and helping me in various ways during my time as a student.

Contents

List of abbreviations

KMP Kotlin Multiplatform

KMM Kotlin Multiplatform Mobile

IDE Integrated Development Environment

OS Operating System

UI User Interface

API Application Program Interface

JSON JavaScript Object Notation

CPU Central Processing Unit

RAM Random Access Memory

1 Introduction

In the rapidly evolving landscape of mobile application development, organizations and developers face increasing pressure to deliver high-performance applications across multiple platforms while maintaining development efficiency. The number of Internet users on mobile devices has seen significant growth in the past decade, with an increase from around 738 million to 4.3 billion users between 2010 and 2020 [17]. This growth has increased the demand for mobile applications in general, and particularly for efficient cross-platform development solutions that can perform at a level with native implementations while reducing the time and costs of development.

At the core of this challenge of efficiently delivering applications lies an important decision: to adopt the cross-platform development approach or to continue using platform-specific frameworks. Native application development, while offering good performance, requires maintaining separate codebases for each platform, leading to increased development time and costs [20]. Cross-platform frameworks like Kotlin Multiplatform (KMP)[1], React Native[2], and Flutter[3] aim to address these challenges, but their performance compared to their native counterparts remains a crucial aspect for organizations and developers to consider.

Performance in mobile applications is influenced by various factors, like CPU utilization, memory allocation, network operations and connectivity, and how efficiently they render the user interfaces (UIs). These factors directly impact user experience, battery life, and overall application responsiveness [11]. While cross-platform frameworks look to provide a more efficient development experience, their actual performance can vary across different use cases and platforms. Understanding how these performance variations take shape is essential for organizations and developers to make informed decisions about which development approach to choose.

This study focuses specifically on KMP, investigating how it performs across various areas compared to its native counterparts. This includes performing CPU-intensive calculations, API (Application Program Interface) requests to test network capabilities, memory allocations, and image component rendering. By conducting these performance comparisons between KMP and native implementations on both Android and iOS platforms, this research looks to identify specific scenarios where KMP outperforms or underperforms, providing insights for organizations and developers deciding which development frameworks to choose.

[1] `https://kotlinlang.org/docs/multiplatform.html`, *accessed at 2025-03-04*.
[2] `https://reactnative.dev/`, *accessed at 2025-03-06*.
[3] `https://flutter.dev/`, *accessed at 2025-03-06*.

1.1 Objectives

The primary objective of this thesis is to investigate the performance differences between KMP and native implementations for Android and iOS mobile applications across various workloads. This investigation will provide insights into the potential trade-offs associated with KMP in mobile application development.

To address this objective, the following research questions have been formulated:

RQ1 — In which scenarios does Kotlin Multiplatform outperform or underperform compared to native implementations?

RQ2 — What factors contribute to variations in performance between Kotlin Multiplatform and native approaches?

RQ3 — In which use cases is Kotlin Multiplatform the better choice, and why?

Ultimately, the study aims to provide a comprehensive understanding of the performance trade-offs between KMP and native development. Furthermore, it will offer recommendations for developers regarding the best approach for different application scenarios.

1.2 External partner

This thesis is conducted in collaboration with THNX Innovation AB, a consultancy firm based in Umeå, Sweden. Established in 2017, THNX Innovation specializes in the development of mobile and web applications, as well as other digital solutions. Their expertise in mobile development will provide both valuable industry insights for this research and access to several physical devices to conduct testing on.

2 Background

This chapter provides a general overview of mobile application development and the most common tools developers use to create applications. Furthermore, it investigates the languages and software used when developing applications for Android and iOS phones. Additionally, the chapter introduces what cross-platform development is, what it does, and how it is used. Finally, the chapter concludes with a look at related work done on the topic of this thesis.

2.1 Mobile Application Development

The term "mobile application development" refers to the process of designing and developing software for digital devices such as smartphones, tablets, and digital assistants [12]. The applications produced through this process have become an essential part of modern computing and communication.

The origins of mobile applications can be traced back to the 1980s with the *Psion Organiser I*, often regarded as the world's first practical pocket computer. This device featured basic productivity tools such as a calculator and a clock. In 1993, IBM introduced the first smartphone, which was a significant advancement in mobile technology. Nearly a decade later, in 2002, BlackBerry introduced new capabilities in mobile phones by enabling users to send and receive emails, laying the foundation for modern mobile communication [31].

In 2024, it was estimated that there were over 6.3 billion smartphone users and 1.4 billion tablet users worldwide. This trend is showing no signs of slowing down in the near future [16]. Approximately 3 billion among these users use Android, while around 1 billion use iPhones. The reflection of these numbers can be seen in the global market share, where, as of November 2024, Android accounted for $\sim$71% of the market. It is important to note that Android is used by multiple smartphone manufacturers, whereas iOS is exclusively available on Apple's iPhone [28].

The continuous growth in the number of mobile users indicates a sustained demand for mobile applications. As of 2024, the *Google Play Store*[1] and the *Apple App Store*[2] hosts approximately 2.87 million and 1.96 million applications, respectively.

[1] `https://play.google.com/store/games?pli=1`, *accessed at 2025-03-10*
[2] `https://www.apple.com/se/app-store/`, *accessed at 2025-03-10.*

2.2 Integrated Development Environments

When creating mobile applications, developers make use of integrated development environments (IDEs). IDEs are software applications that increase productivity by enabling operations including software editing, program building and compiling, and testing. Most IDEs also support syntax highlighting, intelligent code completion, code editing automation, refactoring, and debugging [25].

In today's world of programming, developers often work with multiple IDEs to make use of the strengths of different tools. While some IDEs are specifically designed and optimized for a particular programming language, many developers maintain a toolkit of different IDEs for different purposes. For example, Android Studio is the most popular IDE for Kotlin and Java development, while Xcode is mainly used for Swift and Objective-C. Additionally, there are some multi-language IDEs that support multiple programming languages [25].

The most widely used IDE is Visual Studio[3], which, as of February 2025, is used by approximately 28% of developers. Visual Studio Code[4] comes in second at approximately 15%, and the third place is shared by Android Studio and PyCharm[5] at around 10% [13]. It's worth noting that these statistics point towards primary IDE usage and that many developers often switch between different IDEs depending on what their current project requires.

2.3 Native Android Development

As previously mentioned, Android is the most widely used operating system (OS) for modern smartphones. It was initially developed by the Open Handset Alliance and received financial backing from Google in 2005. Three years later, in September 2008, Google acquired Android and officially launched it as an open-source mobile operating system [7]. Later that same month, the first smartphone running Android OS, the T-Mobile G1 (also known as the HTC Dream), was released to the public [8].

For native Android application development, Android Studio[6] is the official IDE widely used by developers. It was first announced on March 16, 2013, during the Google I/O conference, and the first stable version (1.0) was released in December 2014 [22]. Android Studio is based on IntelliJ IDEA Community Edition, one of the most advanced Java IDEs available at the time [34].

2.3.1 Kotlin and Java

Kotlin[7] and Java[8] are the two most widely used programming languages for Android application development. Java was released in 1995 and is primarily used for mobile applications, game development, and e-business solutions [30]. It is an object-oriented programming language with a syntax derived from C++, making it relatively easy to learn and

[3] `https://visualstudio.microsoft.com/`, *accessed at 2025-03-06.*
[4] `https://code.visualstudio.com/`, *accessed at 2025-03-10.*
[5] `https://www.jetbrains.com/pycharm/`, *accessed at 2025-05-15*
[6] `https://developer.android.com/studio`, *accessed at 2025-02-24.*
[7] `https://kotlinlang.org/`, *accessed at 2025-02-25.*
[8] `https://www.java.com/sv/`, *accessed at 2025-02-25.*

understand. Java features automatic garbage collection, which removes unreferenced objects automatically, and leaves out some of the more complex features like pointers and operator overloading [29].

As the demand for a more developer-friendly language within the Java ecosystem grew, Kotlin was introduced. Developed by JetBrains, Kotlin was officially released on February 15, 2016, and quickly gained popularity. In 2017, Google announced official support for Kotlin, and by May 7, 2019, it had become the preferred language for Android application development [22]. It has since been adopted by over $\sim$60% of professional Android developers due to its ability to enhance productivity, improve developer satisfaction, and increase code safety [4]. One of its key advantages is its syntax, requiring less code compared to Java. This makes Kotlin generally easier to read and maintain, contributing to its growing popularity among Android developers. Additionally, Kotlin is a statically typed language that, unlike Java, fully supports both functional and object-oriented programming paradigms [3].

2.3.2 Strengths and Limitations

Developing applications natively for Android offers several advantages compared to cross-platform approaches. Native applications provide a consistent user interface that aligns with platform-specific design guidelines and offers full access to system resources, including device hardware and platform APIs. Additionally, native development often results in optimized performance, as the code is specifically tailored for the Android environment.

However, native development also comes with certain limitations. One significant drawback is the lack of code re-usability across platforms, requiring separate implementations for iOS and Android. This increases development effort, often requiring two specialized teams with expertise in their respective ecosystems. Furthermore, working in, maintaining, and updating two distinct code bases can lead to higher costs and longer development cycles [32].

2.3.3 Challenges

One of the most significant aspects of native Android development to bring up is the problem of malicious software being deployed onto Android devices. Since Android is the most widely used OS with many different device manufacturers, the risks for large-scale distribution of malware are significant. Developers can hide malicious payloads in applications by abusing the fact that Android supports calling C/C++ library functions via the Java Native Interface [24].

2.4 Native iOS Development

iOS, Apple's mobile operating system, was first introduced in 2007 alongside the launch of the original iPhone. Initially named iPhone OS, it was re-branded to iOS in 2010 and now powers a range of Apple devices, including the iPhone, iPad, and formerly the iPod Touch. As the second most widely used mobile operating system after Android OS, iOS has gone through a significant evolution since its creation [23].

To develop iOS applications, Xcode[9] is required. Released by Apple in 2003, Xcode is an IDE exclusive to Apple hardware. It includes various tools for developing software for iOS, macOS, watchOS, and more. It includes a built-in interface builder for creating graphical user interfaces (GUIs) and an extensive developer documentation. Xcode also supports multiple programming languages, including Swift, Objective-C, C++, and Python [19].

2.4.1 Swift and Objective-C

During the Apple Worldwide Developers Conference (WWDC) in 2014, Apple announced the release of their new programming language called Swift[10]. It is now considered the primary language in the development of applications for iOS, macOS, watchOS, and tvOS. It was created to combat the limitations of its predecessor, Objective-C[11], which is a language derived from C and is considered to have a steep learning curve for developers [15]. Since its creation, Swift has been the main choice for developers. An indicator of its popularity is that as of 2024, $\sim$94% of applications for iOS have adopted Swift [2].

2.4.2 Strengths and Limitations

One of the key strengths of iOS is its stability, making it one of the most reliable mobile operating systems available. Apple's software and hardware integration ensures a smooth experience, minimizing performance issues and system crashes. iOS is also widely regarded as having one of the most polished user interfaces, featuring a visually appealing home screen, well-designed app icons, and a cohesive design that complements Apple's hardware [1].

Furthermore, iOS maintains strong adherence to Web standards and modern development practices, ensuring compatibility with the latest web technologies. Apple also places significant emphasis on cloud storage integration, offering seamless access to iCloud for data backup and file synchronization, enabling information sharing across different Apple devices.

However, the iOS ecosystem also presents several limitations for both its users and developers. As iOS is exclusively designed for and by Apple's hardware, it limits flexibility compared to more open systems like the Android OS. This closed and exclusive nature affects the development process as iOS development is restricted to macOS, and Apple's development tools, including Xcode, are not supported on other operating systems.

Additionally, the mandatory use of Apple hardware creates a barrier of entry for both developers and its users. Apple products are often considered expensive, making them less accessible to a wider audience. This premium pricing also includes accessories and certain services within the Apple ecosystem. For developers and organizations, the cost barrier increases with expenses associated with developing and maintaining iOS applications, like releasing in-app updates and ensuring compatibility across multiple Apple devices [5].

[9] `https://developer.apple.com/xcode/`, *accessed at 2025-02-20.*

[10] `https://developer.apple.com/swift/`, *accessed at 2025-02-26.*

[11] `https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`, *accessed at 2025-02-26.*

2.5 Cross-Platform Development

To address the challenges and limitations associated with native mobile application development, organizations and developers increasingly adopt cross-platform frameworks. These frameworks have gained significant traction in the field of software development, particularly among companies that require applications to function on both Android and iOS platforms [14]. Cross-platform development enables organizations to reduce development time and maintenance efforts, which lowers costs and allows resources to be allocated to other areas.

While cross-platform development offers several advantages, it is a relatively recent approach in mobile application development. Various frameworks, such as Flutter, React Native, and KMP, facilitate this process by enabling code sharing across multiple platforms while still allowing access to native functionalities when necessary.

Although cross-platform development reduces development time and costs, it is not without its disadvantages. One of the main challenges today is the limited number of libraries that are compatible with both platforms. Without compatibility of these libraries, developers have to find and use platform-specific libraries instead, which slows down the development process. Since cross-platform development is becoming more popular and an increasing number of libraries that support cross-platform development are being released into beta versions, issues like this will have a much lesser impact [27].

2.5.1 Kotlin Multiplatform

KMP, previously known as Kotlin Multiplatform Mobile (KMM), is a cross-platform framework designed to enable code sharing across different platforms, including mobile, web, and desktop applications. Unlike other cross-platform solutions that rely on a single runtime environment, KMP allows developers to share business logic while maintaining the ability to write platform-specific code where required [9].

KMP was first introduced as “Kotlin Multiplatform Projects” during KotlinConf 2017 by JetBrains. In 2020, its focus shifted towards code sharing between iOS and Android, leading to its official release as Kotlin Multiplatform [18]. Since then, KMP has continued to evolve, offering improved tooling and compatibility with native APIs, and seamless integration with existing Kotlin-based projects.

2.5.2 Strengths and Limitations

Cross-platform development offers several advantages that make it an attractive approach for mobile application development. The primary strength lies in its cost-effective nature. Cross-platform development significantly reduces long-term maintenance costs by enabling developers to maintain a single codebase rather than separate implementations for each target platform. This approach eliminates the need to implement features or fix bugs multiple times across different platform-specific codebases, making the maintenance process a lot more manageable [10].

Furthermore, KMP reduces the required technical expertise and skillsets. Cross-platform frameworks allow developers to leverage a single programming language and development environment regardless of the target platform. This decreases the learning curve and in-

creases the number of developers who can contribute to a project, as they no longer need specialized knowledge of multiple platform-specific APIs. The abstraction layer provided by KMP presents a unified API that helps developers avoid the complexities of platform-specific implementations [10].

Additionally, the abstraction of platform-specific details allows developers to focus on application logic and user experience rather than platform-specific features. This development process is particularly valuable in environments with fewer resources, where specialized native developers for each platform might be unavailable or too expensive. The economic considerations become especially significant over time, as supporting and maintaining multiple codebases becomes increasingly expensive, making cross-platform development an attractive option for organizations and developers [10].

Although the cross-platform approach can provide potential benefits, several significant limitations must be considered before adopting it in mobile application projects. Performance limitations are one of the concerns, as cross-platform technologies generally demonstrate reduced efficiency compared to native implementations. While native applications directly utilize a platform's core programming language and APIs, cross-platform frameworks must transform code into platform-specific languages, creating a processing and compiling overhead that can become a problem in computationally demanding applications [33].

In addition, cross-platform frameworks often experience delays in supporting new OS features, as framework developers must implement abstractions for each platform's unique APIs following OS updates. This creates a temporary gap period between the release of new platform features and their availability to cross-platform developers. This means that applications requiring the most recent platform features may face implementation delays or require platform-specific implementations to access these capabilities [33].

2.6 Related Work

There have been several works that have been conducted in this area. One report discusses performance evaluations of KMP, but only focusing on comparing KMP and iOS native in Swift. They compiled a benchmark suite and measured the performance metrics execution time, memory consumption, and CPU usage. For future work, the report suggests performing further experiments on a larger scale with varying devices of different quality and hardware. It also brings up the idea of researching differences in potential energy consumption and security concerns between the implementations [26].

Another report investigates the comparison between KMP and their respective native implementations for Android and iOS. The report looks into both the performance metrics of the applications and the experience of developing in KMP. The results suggest KMP iOS implementation performs worse and KMP Android implementation performs better than their native counterparts [6].

A separate study tested performance and performed a quantitative analysis of the codebase to assess KMP's usability. The results gathered suggest that no apparent performance or quality differences were found when comparing KMP and Android native applications. The KMP iOS implementation, however, faced difficulties in application size, slow launch times, and performance problems [9].

A GitHub repository was identified that explores a performance comparison between Flutter, KMP, and native implementation for Android and iOS. The repository consists of a sample app with a single screen that allows the user to load cat pictures from a public API and show them in a horizontal list. The results listed in the repository indicate that the native implementations outperformed both Flutter and KMP when evaluating startup speeds [21].

Despite these previous studies, questions are still open regarding the practical implications of adopting KMP in real-world applications. Existing research primarily focuses on performance metrics (execution time, memory consumption, CPU usage) and developer experience, but gaps remain in evaluating energy efficiency, security concerns, and long-term maintainability across platforms. Additionally, prior studies have often been limited in scope, using specific benchmark suites or controlled experiments rather than larger-scale, real-world applications.

This project aims to expand on the state of the art by providing a broader comparison that incorporates a more diverse set of devices to conduct testing on. By addressing these aspects, this work will contribute to a more comprehensive understanding of KMP's viability for cross-platform development.

3 Methodology

This section presents the systematic approach used to evaluate performance differences between KMP and native mobile development frameworks. It highlights benchmarking protocols, measurement tools, test device specifications, and standardized implementation strategies to ensure objective comparisons. Through performance testing across identical applications, this research quantifies differences in key performance metrics using platform-specific profiling tools. The evaluation framework brought forward focuses on six performance aspects:

- **CPU Performance**: Measuring execution time and CPU utilization when sorting one million random integers.
- **Memory Management and Allocation**: Comparing memory usage and allocation speed when creating small and large objects.
- **API Calls and Network Performance**: Evaluating HTTP request completion times and response sizes for small and large payloads.
- **Image Rendering**: Measuring the time required to render grids of 10 and 50 images.
- **Application Startup Time**: Recording the duration from launch to interactive state.
- **Binary Size**: Comparing the installed application package size between implementations.

3.1 Implementation and Source Code

The complete source code for this study, including both KMP and native implementations, is available in a public repository[1]. The repository contains the full implementation of all test cases and is organized into three main projects:

- `MasterThesisKMP/` - The KMP implementation
- `MasterThesisIOSNative/` - The iOS native implementation
- `MasterThesisAndroidNative/` - The Android native implementation

[1] `https://github.com/ericornkloo/kmp-native-comparison`

3.2 Performance Measurement and Evaluation Methodology

This section details the methodological approach used to evaluate and compare the performance characteristics of KMP and native implementations. The evaluation framework consists of platform-specific profiling tools and a standardized testing protocol designed to ensure objective results.

3.2.1 Performance Measurement Tools

To measure performance differences between implementation approaches, this study uses a combination of manual instrumentation and platform-specific profiling tools to gather precise metrics. For gathering the metrics, *Android Profiler*[2] and *Xcode Instruments*[3] are used to gather information about network communication, CPU utilization percentage, and execution times for specific methods.

Manual instrumentation with console logging is primarily used for memory management, image rendering, and app startup monitoring. This involved programmatically recording timestamps and resource utilization metrics at key execution points. By placing measurements and print statements at critical parts in the code, precise data is collected for areas where the profiling tools are unavailable.

Android Profiler

Android Profiler, integrated within Android Studio, is used to analyze performance metrics on Android devices. This suite provides real-time profiling of multiple system aspects. The following profiling components are used in this study:

- **System Trace**: Measures CPU utilization percentages during performance-intensive operations.
- **CPU Profiling**: Captures execution times and CPU utilization for specific methods.
- **Live Telemetry**: Monitors real-time performance metrics to observe CPU load during test execution.
- **Network Analysis**: Gathers information about network communication, including request durations and payload sizes.
- **Memory Allocation Tracking**: Tracks memory consumption and allocation data.

Xcode Instruments

Xcode Instruments, Apple's tool for analyzing performance, is used for profiling the iOS implementations. This tool provides similar functionality to Android Profiler but with iOS-specific insights. The following instruments are utilized:

- **Time Profiler**: Measures execution speed of specific operations during testing.

[2] `https://developer.android.com/studio/profile`, *accessed at 2025-03-08.*
[3] `https://developer.apple.com/tutorials/instruments?changes=2`, *accessed at 2025-03-08.*

- **CPU Profiler**: Captures CPU utilization during testing.
- **Network Profiler**: Monitors API requests and measures response times during network performance testing.
- **Allocations Profiler**: Tracks memory allocation patterns and usage to evaluate differences in memory efficiency.

3.2.2 Testing Protocol

To ensure reliable and reproducible results, a standardized testing protocol is established and applied consistently across all implementation approaches and test devices. Each benchmark is conducted according to the following protocol:

1. Devices are restored to a baseline state before testing (cleared cache, closed background apps).
2. Device battery levels are maintained above 90%.
3. Each test is repeated five times per device per implementation to ensure reliability in the measurements.
4. Results are recorded as mean values of the five test repetitions.
5. Close-to-identical app builds are used across all devices of the same platform.
6. Testing sessions are limited to 30 minutes per device to prevent heating of devices.

3.3 Test Equipment

The tables below detail the technical specifications of Android and iOS devices used in the benchmark tests, spanning multiple device generations from 2016 to 2023. The selection includes varying processor architectures, Random Access Memory (RAM) capacities (2-12 GB), and storage configurations to provide comprehensive performance insights across different hardware capabilities. These specifications provide essential context for interpreting the benchmark results, as performance differences often directly reflect hardware capabilities.

Android Devices

Table 1 Android Test Devices Specifications

Device	Year	OS	RAM	Processor	Storage
Google Pixel 7 Pro	2022	14	12 GB	Google Tensor G2 (5nm), Octa-core (2×2.85 GHz, 2×2.35 GHz, 4×1.80 GHz)	256 GB
Lenovo Tab M10 HD	2020	10	2 GB	MediaTek Helio P22T (12nm), Octa-core (4×2.3 GHz Cortex-A53, 4×1.8 GHz Cortex-A53)	32 GB
OnePlus 7T Pro	2019	12	8 GB	Snapdragon 855+ (7nm), Octa-core (1×2.96 GHz, 3×2.42 GHz, 4×1.78 GHz)	256 GB
Google Pixel 4 XL	2019	13	6 GB	Snapdragon 855 (7nm), Octa-core (1×2.84 GHz, 3×2.42 GHz, 4×1.78 GHz)	64 GB
Google Pixel	2016	13	4 GB	Snapdragon 821 (14nm), Quad-core (2×2.15 GHz, 2×1.6 GHz)	32 GB

iOS Devices

Table 2 iOS Test Devices Specifications

Device	Year	iOS	RAM	Processor	Storage
iPhone 15 Plus	2023	18.4.1	8 GB	Apple A16 Bionic (4nm), Hexa-core (2×3.46 GHz Everest, 4×2.02 GHz Sawtooth)	128 GB
iPhone 14 Pro Max	2022	18.4.1	6 GB	Apple A16 Bionic (4nm), Hexa-core (2×3.46 GHz Everest, 4×2.02 GHz Sawtooth)	256 GB
iPhone 12	2020	17.5.1	4 GB	Apple A14 Bionic (5nm), Hexa-core (2×3.1 GHz Firestorm, 4×1.8 GHz Icestorm)	128 GB
iPad Pro 10.5”	2017	17.7.1	4 GB	Apple A10X Fusion (10nm), Hexa-core (3×2.38 GHz Hurricane, 3×1.3 GHz Zephyr)	64 GB
iPhone 8	2017	15.5	2 GB	Apple A11 Bionic (10nm), Hexa-core (2×2.39 GHz Monsoon, 4×1.42 GHz Mistral)	64 GB

3.4 Test Implementations

The test applications were deliberately designed to be simple, avoiding platform-specific optimizations or workarounds whenever possible. This helps direct performance comparison between KMP and native approaches without implementation-specific additions that further complicate comparisons. Each test focuses on isolated performance aspects with near-identical implementations on the different platforms, ensuring that potential differences when compared are not affected as much by variations in how the applications were developed.

3.4.1 Execution Speed (CPU Performance)

This benchmark test sorts one million random integers using the platform-specific, default sorting algorithm using “sort()”. Important to note is that the sorting algorithms differ between Swift and Kotlin. Swift uses a hybrid algorithm combining Insertion Sort and Quicksort called Timsort[4] and Kotlin uses a modified Mergesort implementation[5]. While this

[4] `https://github.com/swiftlang/swift/blob/main/stdlib/public/core/Sort.swift`, *accessed at 2025-05-14*.

[5] `https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.collections/sort.html`, *accessed at 2025-05-14*.

difference in sorting method affects direct cross-platform comparisons, it accurately represents real-world development scenarios where platform-specific sorting implementations would be used.

Test Design

This test is selected to provide a consistent, processor-intensive task that resembles real-world computational needs in mobile applications. Sorting large datasets is a common requirement in many applications, from data visualization to search functionality. A test that sorts a million random integers creates a reasonable computational load to stress the CPU while, at the same time, being practical in the sense that repeated testing across multiple devices is conducted.

Implementation Methodology

Each platform implementation follows identical steps:

1. Initialize an array of one million elements.
2. Populate with random integers (0 to 999,999).
3. Execute the standard, language-specific sorting algorithm.

Measurement Approach

Using the platform-specific profiling tools Xcode Instruments and Android Profiler, the test measures the total execution time required for each implementation to complete the sorting operation. Additionally, the percentage of CPU usage during the test will be recorded.

Test Protocol

To ensure consistent and reliable results, this test follows these steps:

1. Create an array with randomly generated integers.
2. Execute the sorting operation and record performance metrics.
3. Repeat the test five times.
4. Calculate average performance across the test runs.

The results from these measurements provide data on how efficiently each platform handles processor-intensive operations and thereby offer insights into the potential performance differences between the implementation approaches.

3.4.2 Memory Management and Allocation

This test evaluates how efficiently each platform handles memory allocation, specifically focusing on the performance of these operations.

Test Design

The benchmark tests how well each platform handles memory by performing two types of allocations that developers commonly encounter in real-world apps. These tests were created to test the memory management systems of each platform and compare how they might handle different scenarios.

Implementation Methodology

The implementation of these tests is conducted equally across all platforms using two different allocation patterns to evaluate different aspects of memory management. The first test focuses on small object allocation, where each implementation creates 100,000 small objects of 1 KB each, totaling approximately 100 MB of memory. This approach emulates common scenarios where applications handle numerous small data structures.

The second test examines large object allocation behavior by creating 100 large objects of 1 MB each, also totaling approximately 100 MB. This approach allows for direct comparison between how the platforms handle the same memory allocation using differently sized allocations.

Measurement Approach

Using platform-specific profiling tools and manual instrumentation, the test examines two different memory allocation measurements. One being the memory delta (MB) (i.e., the actual change in memory usage after allocation), and the other being allocation time (ms) (i.e., the time to complete the allocation operation). Using these values, the memory increase rate (MB/s) is then calculated.

Test Protocol

To ensure consistent results, the testing procedure follows these steps:

1. Begin with a baseline memory measurement.
2. Perform small object allocations and record metrics.
3. Perform large object allocations and record metrics.
4. Record and log all measurements during the process.

The results from these measurements provide insights into how each platform's memory management system handles allocations of different-sized objects and the efficiency of the implementation approaches on various test devices.

3.4.3 API Calls and Network Performance

This benchmark evaluates network performance across platforms, measuring HTTP request handling, response processing, and data serialization efficiency.

Test Design

The benchmark performs API calls to retrieve user data with two distinct payload sizes (10 and 250 records of user data), selected to provide a realistic comparison of how each platform handles different API load scenarios.

Implementation Methodology

Each platform implementation uses its native networking library to conduct the tests. The three implementations KMP, Android native, and iOS native, used Ktor[6], OkHttp[7], and URLSession[8], respectively.

The test procedure follows these steps across all implementations:

1. Initiate GET request to the RandomUser API with specified result count (10 or 250).
2. Process JSON (JavaScript Object Notation) response and deserialize into typed objects.
3. Display results to the UI.
4. Measure total request time (ms) and response size (KB) using profiling tools.

Measurement Approach

Using platform-specific profiling tools and manual instrumentation, the test captures two key metrics: response time (ms) to measure overall network performance and response size (KB) to ensure payloads stay consistent across implementations. This enables direct comparison of networking efficiency between native and cross-platform approaches.

Test Protocol

The benchmark executes API calls with the payload sizes of 10 and 250 records. Each test is performed multiple times across various devices to account for differences in network connections and hardware. The measurements reveal differences in how each platform's network stack handles HTTP requests, processes JSON data, and manages network operations in general. These factors are important for application responsiveness and efficiency in real-world scenarios.

3.4.4 Image Rendering

This benchmark evaluates image rendering performance across platforms, measuring the time it takes for images to fully render on screen. This metric is important for understanding application responsiveness and overall user experience when displaying visual content.

[6] `https://ktor.io/`, *accessed at 2025-03-13.*
[7] `https://square.github.io/okhttp/`, *accessed at 2025-03-13.*
[8] `https://developer.apple.com/documentation/foundation/urlsession`, *accessed at 2025-03-13.*

Test Design

The benchmark dynamically loads and renders two specific quantities of images, 10 and 50, chosen to represent both lightweight and more intensive image rendering scenarios. This approach enables direct comparison of rendering performance, typical in mobile applications.

Implementation Methodology

Each platform implementation uses its native UI framework. For KMP, Android native, and iOS native, the frameworks used are Compose Multiplatform[9], Jetpack Compose[10], and SwiftUI[11], respectively.

The test procedure follows these steps across all implementations:

1. Fetch random images from a public API endpoint.
2. Load images into memory and prepare for rendering.
3. Display images in a scrollable list component.
4. Measure total rendering time of all images in the list.

Measurement Approach

The test programmatically measures image rendering time built directly into the applications. When the rendering test is initiated, each implementation starts a precise timer at the beginning of the operation. This timer stops when all images are successfully rendered in the application. Lastly, the total rendering time is recorded and displayed in the UI.

This approach provides a measurement of the actual time required for all images to be fully rendered on screen, which relates to the user experience of waiting for visual content to appear in real-world applications.

Test Protocol

The benchmark executes image loading and rendering with two specific batch sizes. The first test renders a small batch of 10 images, while the second test renders a larger batch of 50 images. Each test is performed five times across various devices to ensure reliable results. During testing, a single key metric is collected: the total rendering time in milliseconds required for all images to fully appear on screen. This approach provides a direct measurement of the performance differences between implementations.

By comparing rendering times between the native and KMP implementations for both smaller and larger batch sizes, the test provides clear insights into the relative image rendering performance of each approach. This specific focus on rendering time correlates with performance differences users would experience when interacting with image-heavy UIs.

[9] `https://www.jetbrains.com/lp/compose-multiplatform/`, *accessed at 2025-03-13*.
[10] `https://developer.android.com/jetpack/compose`, *accessed at 2025-03-13*.
[11] `https://developer.apple.com/documentation/swiftui`, *accessed at 2025-03-13*.

3.4.5 App Startup Time

This benchmark evaluates application launch performance, an important factor for user experience and perceived responsiveness of the application.

Test Design

The test measures the time required for each application to initialize, load essential components, and become fully responsive. App startup performance directly impacts user experience, with faster launches leading to higher user satisfaction. The benchmark captures three distinct phases of the startup process:

1. Time to first frame: The duration until the first UI elements appear on screen.
2. Time to interactive: The point at which the application becomes fully responsive to user input.
3. Total startup time: The period from app launch to UI rendering completion.

Implementation Methodology

Each platform implementation uses manual instrumentation to measure the application startup process. The iOS implementations utilize the Swift Date() API[12], which provides timestamp functionality with millisecond precision to track each phase of the launch sequence. Similarly, the Android implementations leverage System.currentTimeMillis()[13] to get comparable timing accuracy across the different devices and hardware configurations. Both approaches make use of timing markers at specific points in the application life cycle, enabling precise calculation of the intervals between initialization, first frame rendering, and interactive state readiness.

Measurement Approach

To get comparable results for each of the test benchmarks, print statements for the timestamps in milliseconds, using with the previously mentioned date/time libraries, is used. This approach ensures the different phases of application startups are measured accurately and, thereby, enables a fair comparison between the implementation approaches.

Test Protocol

To ensure consistent measurements, the test protocol used in this these tests includes clearing application cache before each test run, terminating all background processes, performing each startup test five times, and finally, computing the average values across the test runs for each of the devices.

[12] `https://developer.apple.com/documentation/foundation/date`, *accessed at 2025-05-14*.
[13] `https://developer.android.com/reference/android/os/SystemClock`, *accessed at 2025-05-14*.

3.4.6 Binary Size

This benchmark test analyzes application package size, a factor that affects device storage requirements and download times.

Test Design

The test measures the final application package size across implementations. Binary size is important for mobile applications, where download size can significantly impact installation rates. Smaller application packages generally lead to higher rates of completed installations and reduce the impact on the storage of the user's device.

Implementation Methodology

For binary size measurements, both Android and iOS applications were built using their respective release configurations. The Android application was built using Gradle's release build type, while the iOS application was built using Xcode's release configuration.

Measurement Approach

Application binary size is measured by examining the installed application's storage directly on each test device. After installation, the device's built-in storage management interface provides accurate information about the application's total installed size. This approach captures all components of the application, including executable code, frameworks, and resources.

Test Protocol

To ensure an accurate binary size comparison between the approaches, a consistent protocol is followed across all test applications. Each application is built using production-ready release configurations. The applications were then installed on their respective test devices in a clean state, without prior versions or cached data. After installation, the device's built-in storage management interface is used to record the exact storage in megabytes. This approach provides a fair basis for comparing the relative size impact between native and KMP implementations by capturing the complete on-device storage requirements under identical conditions.

3.5 Analysis of Results

The analysis of the results follows a structured approach to investigate and discuss the potential performance differences between the implementations.

3.5.1 Comparative Performance Analysis

Each benchmark category is analyzed independently, comparing KMP implementations against their native counterparts. The measurements, both mean values and percentage differences, highlight potential performance differences between the implementations.

3.5.2 Cross-Platform Consistency Analysis

The analysis also looks into how KMP performs across the two platforms compared to their native counterpart. This will provide insights into whether potential performance differences follow similar patterns no matter the operating systems or if certain platform-specific variations exist.

3.5.3 Threshold-Based Categorization

To more easily determine the significance of the test results, a threshold-based categorization is created:

- Negligible difference: $<5\%$ variation.
- Minor difference: 5-15% variation.
- Significant difference: 15-30% variation.
- Critical difference: $>30\%$ variation.

These thresholds are selected to provide a structured way of categorizing performance differences. While these values are somewhat random, they serve as a categorization template to help draw conclusions regarding the results. For example, negligible differences are reserved for performance impacts that are not likely to be noticed, and critical differences are meant for results that noticeably impact user experience. The limitations of this categorization approach is discussed in the “Limitations of the Study” section.

3.6 Evaluation Methods

A selection of evaluation methods and criteria has been brought forward to conduct analysis and draw conclusions from the test results.

3.6.1 Quantitative Statistical Analysis

The evaluation calculates mean values from five test runs for each performance metric across the implementations. This approach ensures reliable data for each test category.

Furthermore, when evaluating differences between the implementation approaches, relative performance ratios are calculated to showcase performance differences as percentages (e.g., KMP implementation is X% slower than native). These ratios provide an easy and effective way of measuring the performance differences.

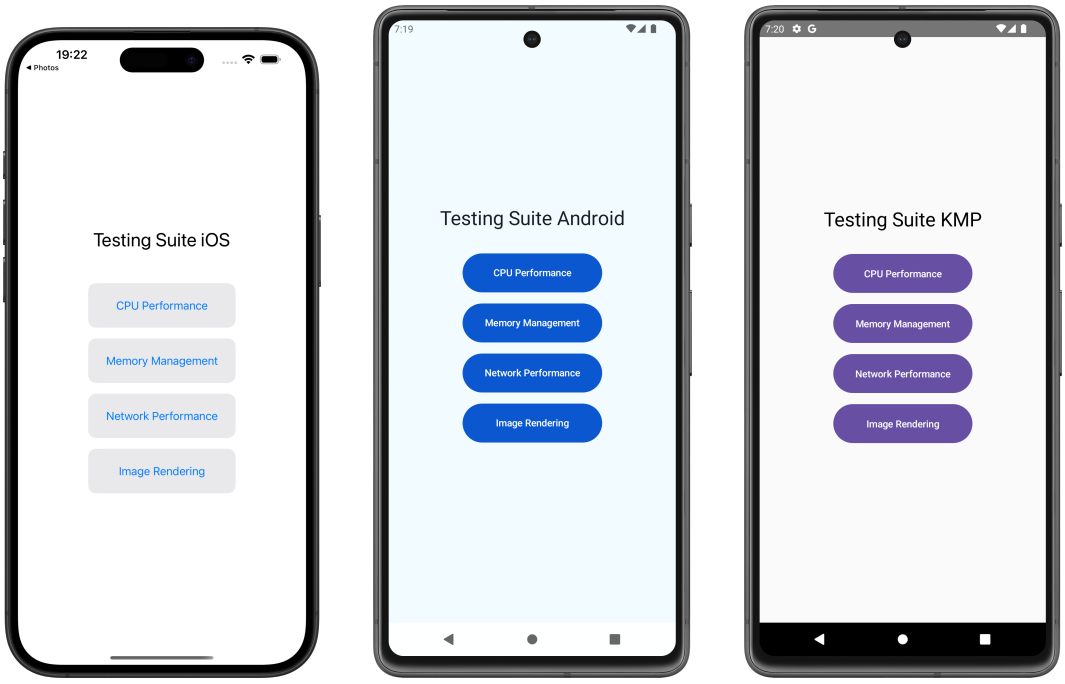
3.6.2 Metric-Specific Evaluation Criteria

Each performance test result is evaluated using criteria specific to its nature:

- CPU Performance: Sorting time and CPU utilization.
- Memory Management: Allocation time, memory allocated, and allocation rate.
- Network Performance: Request completion time and payload size.
- Image Rendering: Time to render images of varying amounts.
- App Startup: Timestamps for first frame, interactivity, and total startup time.
- Binary Size: Installed application size differences.

3.7 Test Application Overview

To conduct the performance comparisons between KMP and native implementations, three test applications were developed with identical functionality. These applications can be seen as controlled environments for conducting performance benchmarks across all previously mentioned test categories.



(a) iOS Native Implementation

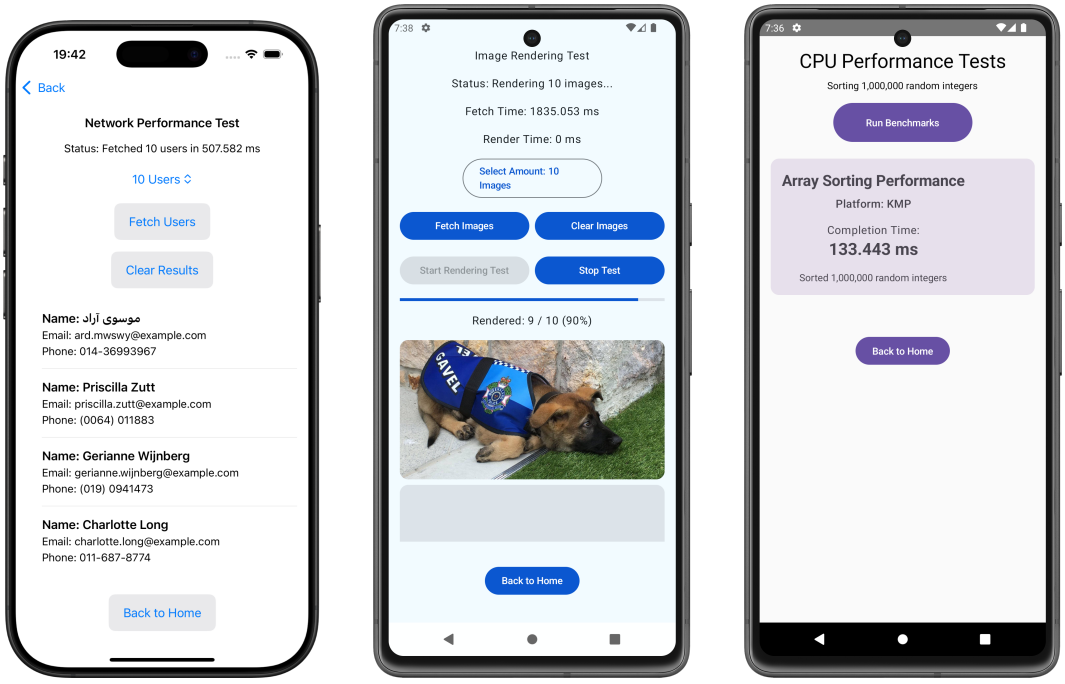
(b) Android Native Implementation

(c) Android KMP Implementation

Figure 1: Home screens of test applications showing identical functionality and navigation options across implementations.

Each application implements five performance tests, accessible through a similar navigation interface (see Figure 1). The tests are designed to investigate specific performance aspects of

these mobile applications while keeping the implementation consistent across all platforms to ensure a fair comparison.



(a) iOS Native Implementation

(b) Android Native Implementation

(c) Android KMP Implementation

Figure 2: Screens of test applications during or after execution of test cases.

Figure 2 show examples of the actual test screens from the applications, displaying how performance tests are presented to users and how results are presented during and after execution.

4 Results

4.1 Performance Results by Category

The following sections present the performance testing results across the six benchmark categories: CPU performance, memory management, network performance, image rendering, application startup time, and application binary size.

All results presented are the mean values calculated from five test runs per device per implementation. A mean value for all three implementations as a whole is also included. Performance differences are presented as percentages, where native implementations serve as a baseline for comparison (e.g., a +20% value means the native implementation outperformed KMP by 20%. A negative percentage value indicates KMP performed better).

Additionally, included in every section are tables and figures with summarized results and implementation-specific comparisons. Finally, a brief identification of key findings will be presented for each benchmark. The measurement data is also available in Appendix B for reference.

4.1.1 CPU Performance

This section presents the test results of the sorting of one million integers using platform-specific, default sorting algorithms. The benchmark measured execution time (ms) and CPU utilization (%) during the sorting operation across native and KMP implementations.

Summary of Results

Figure 3 and Table 3 below show the percentage differences in execution times between KMP and native implementations.

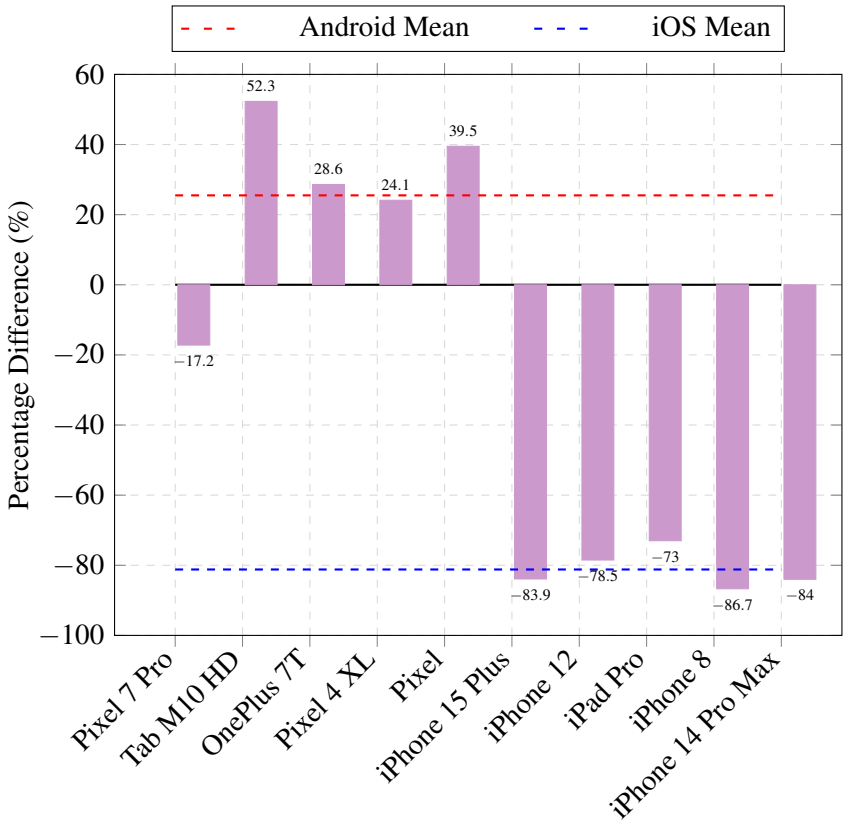


Figure 3: CPU Performance: Execution Time Percentage Difference Between KMP and Native Implementations

Table 3 CPU Performance: Execution Time Percentage Difference (KMP vs. Native)

Device	**Execution Time (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	-17.2%	Minor Better
Lenovo Tab M10 HD	+52.3%	Critical Worse
OnePlus 7T Pro	+28.6%	Significant Worse
Google Pixel 4 XL	+24.1%	Significant Worse
Google Pixel	+39.5%	Critical Worse
Android Mean	*+25.5%*	*Significant Worse*
	iOS Devices	
iPhone 15 Plus	-83.9%	Critical Better
iPhone 12	-78.5%	Critical Better
iPad Pro	-73.0%	Critical Better
iPhone 8	-86.7%	Critical Better
iPhone 14 Pro Max	-84.0%	Critical Better
iOS Mean	*-81.2%*	*Critical Better*

Figure 4 and Table 4 below show the percentage differences in CPU utilization between KMP and native implementations.

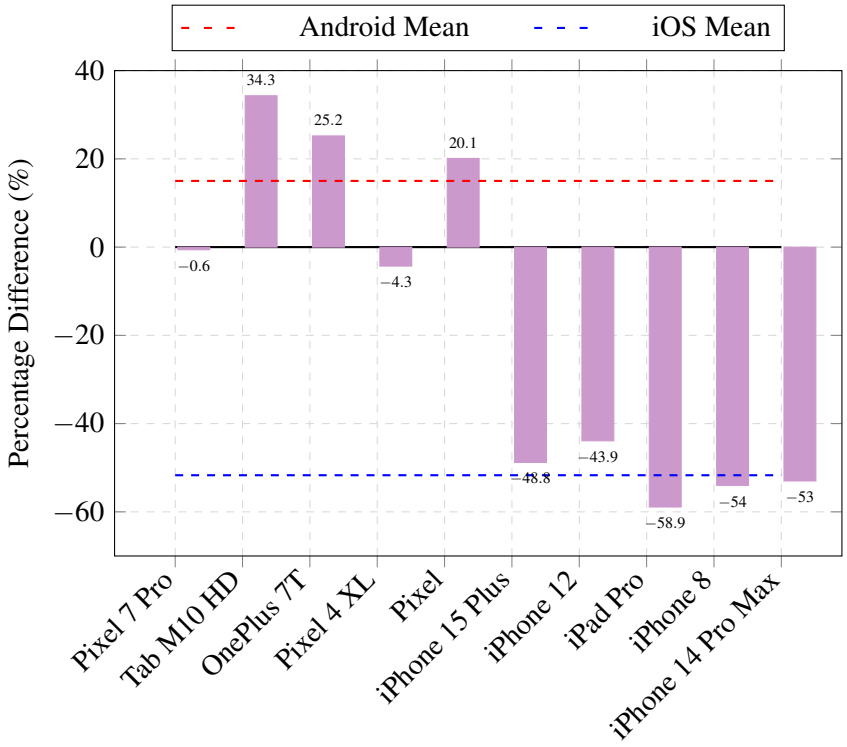


Figure 4: CPU Performance: CPU Utilization Percentage Difference Between KMP and Native Implementations

Table 4 CPU Performance: CPU Utilization Percentage Difference (KMP vs. Native)

Device	**CPU Utilization (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	-0.6%	Negligible Better
Lenovo Tab M10 HD	+34.3%	Critical Worse
OnePlus 7T Pro	+25.2%	Significant Worse
Google Pixel 4 XL	-4.3%	Negligible Better
Google Pixel	+20.1%	Significant Worse
Android Mean	*+15.0%*	*Significant Worse*
	iOS Devices	
iPhone 15 Plus	-48.8%	Critical Better
iPhone 12	-43.9%	Critical Better
iPad Pro	-58.9%	Critical Better
iPhone 8	-54.0%	Critical Better
iPhone 14 Pro Max	-53.0%	Critical Better
iOS Mean	*-51.7%*	*Critical Better*

Key Findings

The CPU performance tests reveal significant platform-specific differences between KMP and native implementations.

Regarding execution times, KMP show varying performance across the different platforms. On Android, it is $\sim$26% slower on average, while on iOS, it performs $\sim$81% faster on average, with all devices showing great improvements.

Furthermore, in terms of CPU utilization, KMP display varying performances again. On Android, it utilized 15% more of the CPU on average, and on iOS, it used $\sim$52% less.

These findings may be impacted by the differences in sorting algorithms since the results show that KMP is critically better in both measured aspects compared to the native iOS implementation.

4.1.2 Memory Management

In this section, the results of the memory management performance tests will be presented, focusing on two metrics: small and large object allocation (10,000 and 100 objects, respectively). The tests measured memory deltas (the actual allocated memory in MB calculated using the difference in memory before and after the allocations) and allocation times (ms) across the two implementations.

Summary of Results

Figure 5 and Table 5 below show the percentage differences in memory delta for small objects between KMP and native implementations.

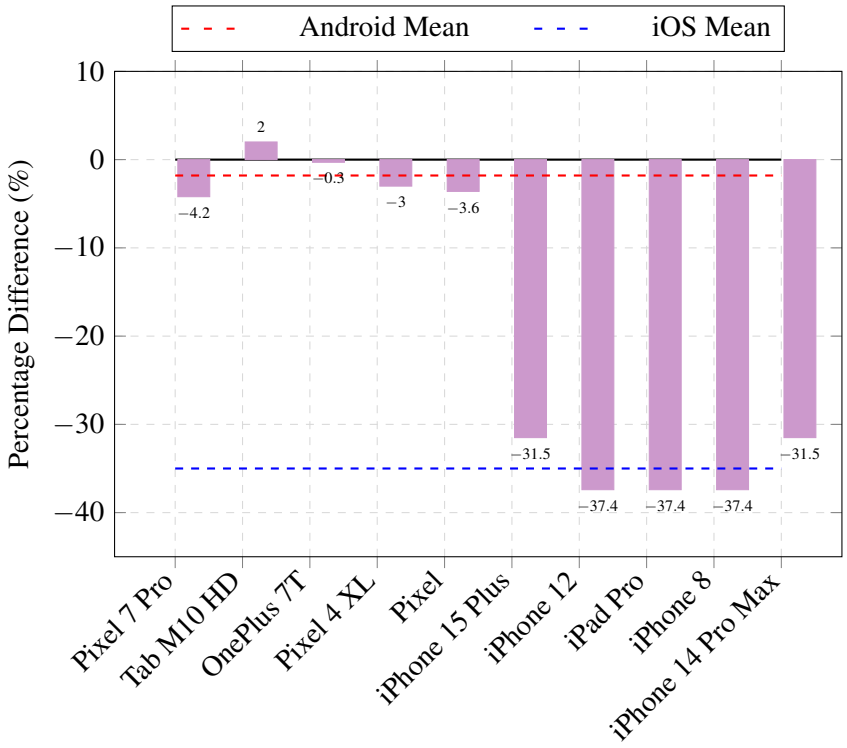


Figure 5: Memory Management: Percentage Difference for Small Objects Memory Delta (KMP vs. Native)

Table 5 Memory Management: Percentage Difference for Small Objects Memory Delta (KMP vs. Native)

Device	**Memory Delta (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	-4.2%	Minor Better
Lenovo Tab M10 HD	+2.0%	Minor Worse
OnePlus 7T Pro	-0.3%	Negligible
Google Pixel 4 XL	-3.0%	Minor Better
Google Pixel	-3.6%	Minor Better
Android Mean	*-1.8%*	*Minor Better*
	iOS Devices	
iPhone 15 Plus	-31.5%	Significant Better
iPhone 12	-37.4%	Significant Better
iPad Pro	-37.4%	Significant Better
iPhone 8	-37.4%	Significant Better
iPhone 14 Pro Max	-31.5%	Significant Better
iOS Mean	*-35.0%*	*Significant Better*

Figure 6 and Table 6 below show the percentage differences in allocation time for small objects between KMP and native implementations.

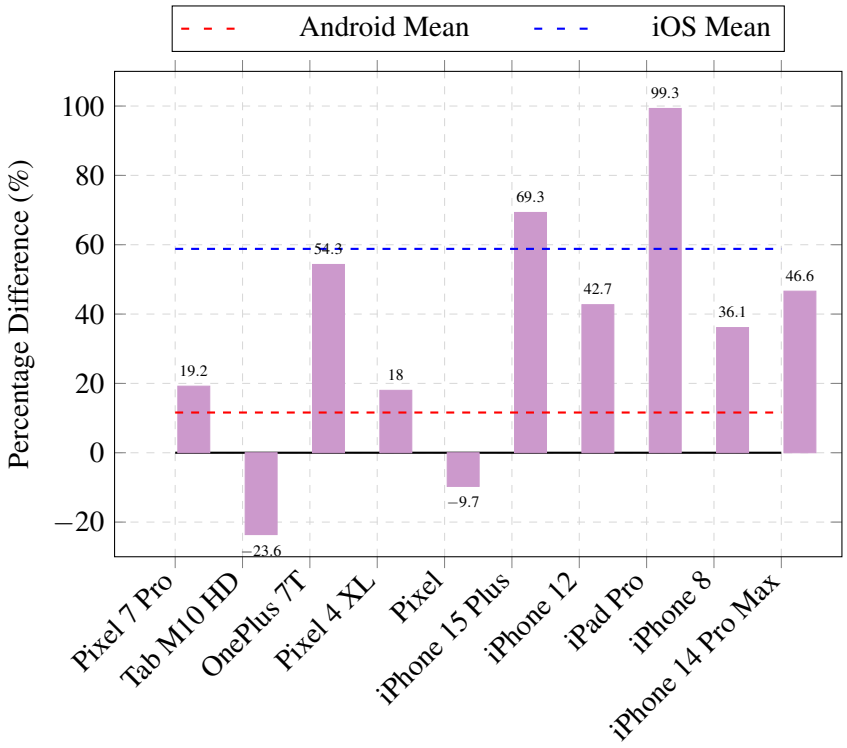


Figure 6: Memory Management: Percentage Difference for Small Objects Allocation Time (KMP vs. Native)

Table 6 Memory Management: Percentage Difference for Small Objects Allocation Time (KMP vs. Native)

Device	**Allocation Time (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	+19.2%	Minor Worse
Lenovo Tab M10 HD	-23.6%	Minor Better
OnePlus 7T Pro	+54.3%	Significant Worse
Google Pixel 4 XL	+18.0%	Minor Worse
Google Pixel	-9.7%	Minor Better
Android Mean	*+11.6%*	*Minor Worse*
	iOS Devices	
iPhone 15 Plus	+69.3%	Significant Worse
iPhone 12	+42.7%	Significant Worse
iPad Pro	+99.3%	Critical Worse
iPhone 8	+36.1%	Significant Worse
iPhone 14 Pro Max	+46.6%	Significant Worse
iOS Mean	*+58.8%*	*Significant Worse*

Figure 7 and Table 7 below show the percentage differences in memory delta for large objects between KMP and native implementations.

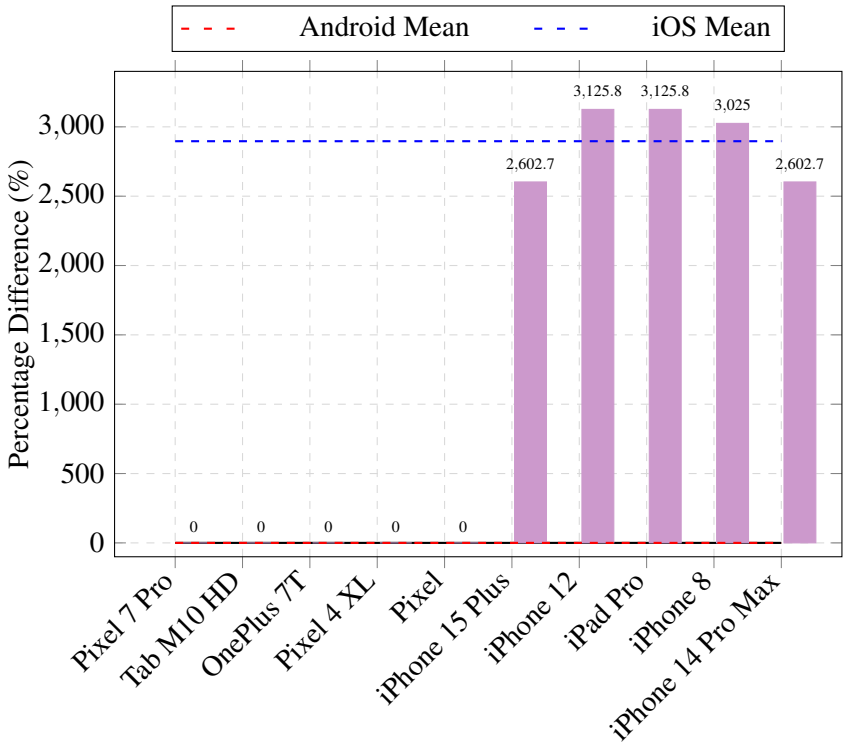


Figure 7: Memory Management: Percentage Difference for Large Objects Memory Delta (KMP vs. Native)

Table 7 Memory Management: Percentage Difference for Large Objects Memory Delta (KMP vs. Native)

Device	**Memory Delta (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	0.0%	Negligible
Lenovo Tab M10 HD	0.0%	Negligible
OnePlus 7T Pro	0.0%	Negligible
Google Pixel 4 XL	0.0%	Negligible
Google Pixel	0.0%	Negligible
Android Mean	*0.0%*	*Negligible*
	iOS Devices	
iPhone 15 Plus	+2602.7%	Critical Worse
iPhone 12	+3125.8%	Critical Worse
iPad Pro	+3125.8%	Critical Worse
iPhone 8	+3025.0%	Critical Worse
iPhone 14 Pro Max	+2602.7%	Critical Worse
iOS Mean	*+2896.4%*	*Critical Worse*

Figure 8 and Table 8 below show the percentage differences in allocation time for large

objects between KMP and native implementations.

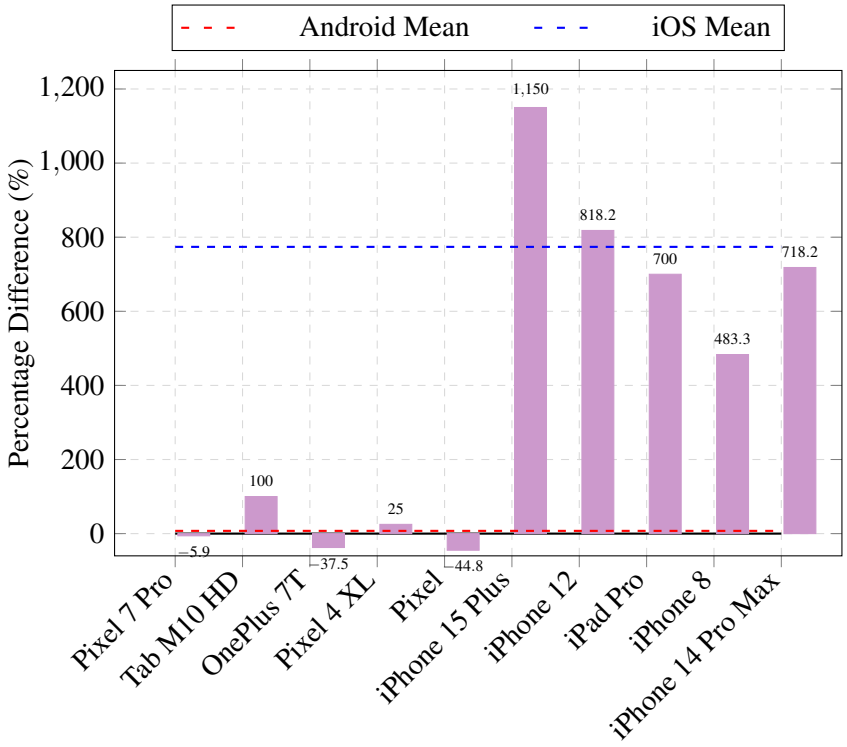


Figure 8: Memory Management: Percentage Difference for Large Objects Allocation Time (KMP vs. Native)

Table 8 Memory Management: Percentage Difference for Large Objects Allocation Time (KMP vs. Native)

Device	**Allocation Time (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	-5.9%	Minor Better
Lenovo Tab M10 HD	+100.0%	Critical Worse
OnePlus 7T Pro	-37.5%	Significant Better
Google Pixel 4 XL	+25.0%	Significant Worse
Google Pixel	-44.8%	Critical Better
Android Mean	*+7.4%*	*Minor Worse*
	iOS Devices	
iPhone 15 Plus	+1150.0%	Critical Worse
iPhone 12	+818.2%	Critical Worse
iPad Pro	+700.0%	Critical Worse
iPhone 8	+483.3%	Critical Worse
iPhone 14 Pro Max	+718.2%	Critical Worse
iOS Mean	*+773.9%*	*Critical Worse*

Key Findings

The memory management tests brings forward some interesting insights about the memory allocation differences between KMP and native implementations.

The most significant finding in this test is the extremely low memory delta of ∼3 MB in iOS native implementations for large objects, compared to the expected ∼100 MB (see Table 19 in Appendix B). This suggests a difference in how Swift handles memory allocations behind the scenes for large objects.

Additionally, both Android native and KMP display a major difference in allocation times between small and large objects. For small objects, allocation times look to be reasonable (+100 ms), while allocations of large objects only take ∼1ms (see Table 18 and Table 19 in Appendix B). This suggests that KMP's method of managing its memory could be dependent on the size and number of objects.

Finally, the results reveal a clear trade-off between memory efficiency and allocation speed. iOS native implementations showcase a high memory efficiency for large objects but at the cost of longer allocation times, while KMP implementations maintain consistent memory usage patterns but with varying allocation time performance.

4.1.3 Network Performance

In the following section, the results of the network performance tests will be presented. For both implementations, the focus will be on the request times (ms) for two different payload sizes (10 and 250 records).

Summary of Results

Figure 9 and Table 9 below show the percentage differences in request time for small payloads between KMP and native implementations.

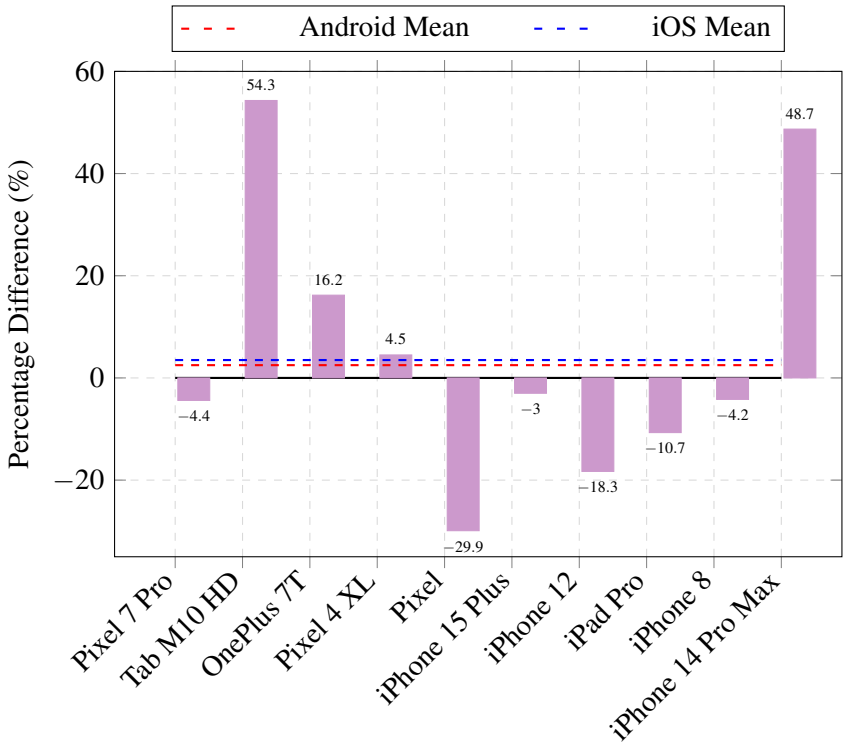


Figure 9: Network Performance: Percentage Difference for Small Payload Request Time (KMP vs. Native)

Table 9 Network Performance: Percentage Difference for Small Payload Request Time (KMP vs. Native)

Device	**Request Time (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	-4.4%	Minor Better
Lenovo Tab M10 HD	+54.3%	Significant Worse
OnePlus 7T Pro	+16.2%	Minor Worse
Google Pixel 4 XL	+4.5%	Minor Worse
Google Pixel	-29.9%	Significant Better
Android Mean	*+2.5%*	*Minor Worse*
	iOS Devices	
iPhone 15 Plus	-3.0%	Minor Better
iPhone 12	-18.3%	Minor Better
iPad Pro	-10.7%	Minor Better
iPhone 8	-4.2%	Minor Better
iPhone 14 Pro Max	+48.7%	Significant Worse
iOS Mean	*+3.5%*	*Minor Worse*

Figure 10 and Table 10 below show the percentage differences in request time for large payloads between KMP and native implementations.

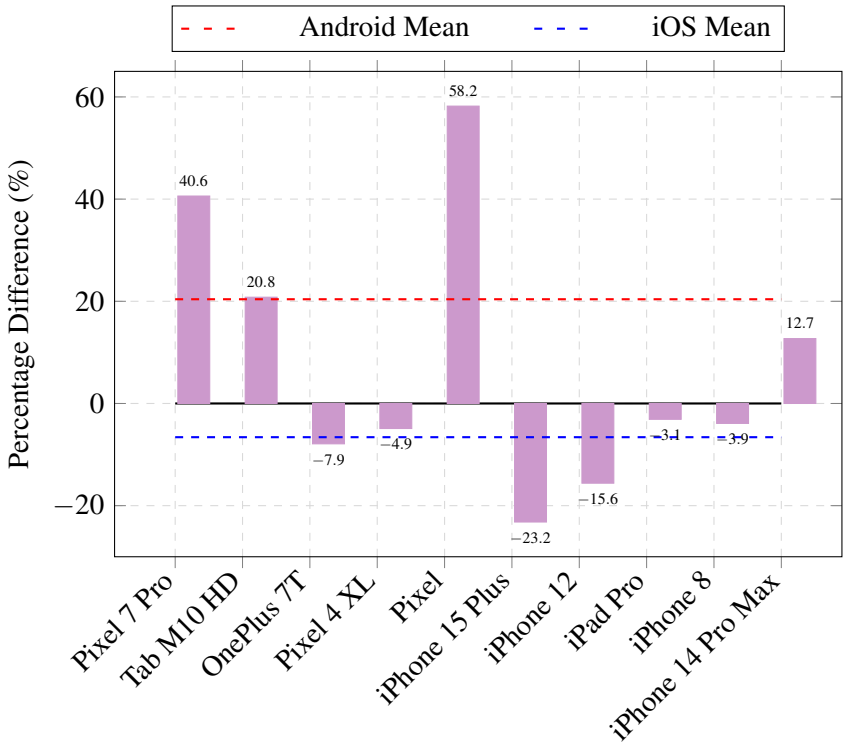


Figure 10: Network Performance: Percentage Difference for Large Payload Request Time (KMP vs. Native)

Table 10 Network Performance: Percentage Difference for Large Payload Request Time (KMP vs. Native)

Device	**Request Time (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	+40.6%	Significant Worse
Lenovo Tab M10 HD	+20.8%	Minor Worse
OnePlus 7T Pro	-7.9%	Minor Better
Google Pixel 4 XL	-4.9%	Minor Better
Google Pixel	+58.2%	Significant Worse
Android Mean	*+20.4%*	*Minor Worse*
	iOS Devices	
iPhone 15 Plus	-23.2%	Minor Better
iPhone 12	-15.6%	Minor Better
iPad Pro	-3.1%	Minor Better
iPhone 8	-3.9%	Minor Better
iPhone 14 Pro Max	+12.7%	Minor Worse
iOS Mean	*-6.6%*	*Minor Better*

Key Findings

The network performance tests reveal some insights about how the performance of KMP and native implementations differ.

The difference in performance on average between the two implementations when comparing the small payload requests is almost negligible. However, when executing the same comparison but with a large payload request, the differences are more pronounced (around +20% for Android vs. approximately -7% for iOS). This might imply that native Android implementations handle large payload requests better than their native counterpart in iOS.

4.1.4 Image Rendering

In this section, the results of the image rendering performance tests will be presented, focusing on rendering times (ms) across different image counts (10 and 50 images) for all implementations.

Summary of Results

Figure 11 and Table 11 below show the percentage differences in rendering time for a small image count between KMP and native implementations.

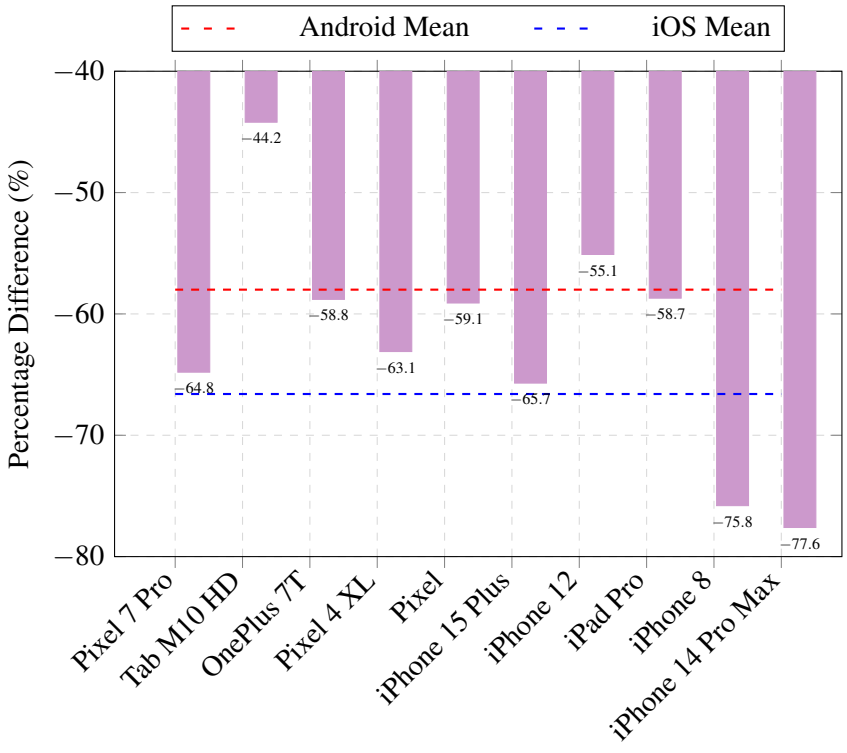


Figure 11: Image Rendering: Percentage Difference for Small Image Count (KMP vs. Native)

Table 11 Image Rendering: Percentage Difference for Small Image Count (KMP vs. Native)

Device	Rendering Time (%)	Category
	Android Devices	
Google Pixel 7 Pro	-64.8%	Critical Better
Lenovo Tab M10 HD	-44.2%	Significant Better
OnePlus 7T Pro	-58.8%	Critical Better
Google Pixel 4 XL	-63.1%	Critical Better
Google Pixel	-59.1%	Critical Better
Android Mean	*-58.0%*	*Critical Better*
	iOS Devices	
iPhone 15 Plus	-65.7%	Critical Better
iPhone 12	-55.1%	Critical Better
iPad Pro	-58.7%	Critical Better
iPhone 8	-75.8%	Critical Better
iPhone 14 Pro Max	-77.6%	Critical Better
iOS Mean	*-66.6%*	*Critical Better*

Figure 12 and Table 12 below show the percentage differences in rendering time for a large image count between KMP and native implementations.

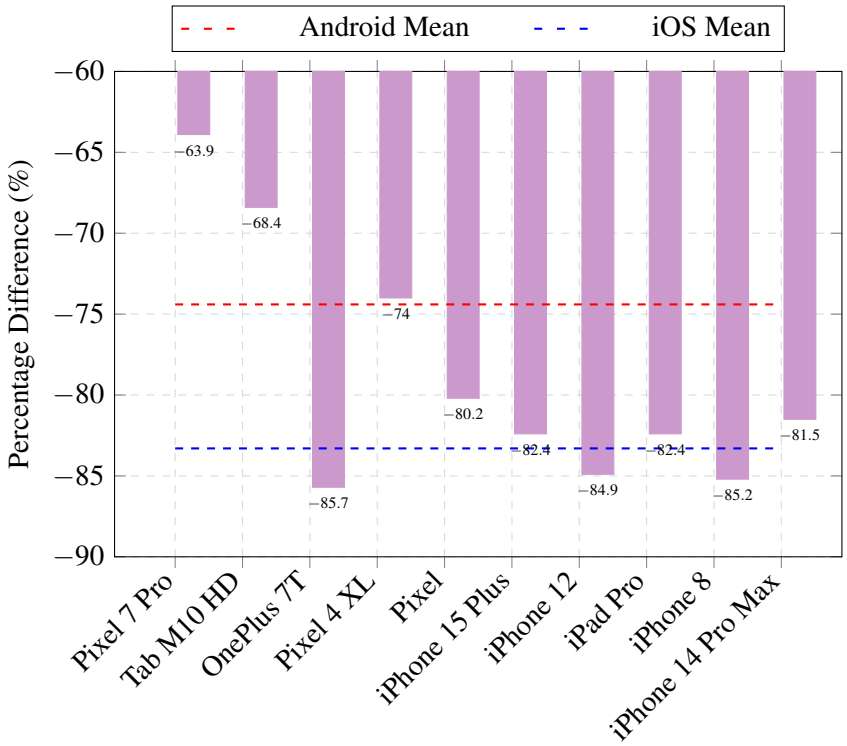


Figure 12: Image Rendering: Percentage Difference for Large Image Count (KMP vs. Native)

Table 12 Image Rendering: Percentage Difference for Large Image Count (KMP vs. Native)

Device	Rendering Time (%)	Category
Android Devices		
Google Pixel 7 Pro	-63.9%	Critical Better
Lenovo Tab M10 HD	-68.4%	Critical Better
OnePlus 7T Pro	-85.7%	Critical Better
Google Pixel 4 XL	-74.0%	Critical Better
Google Pixel	-80.2%	Critical Better
Android Mean	*-74.4%*	*Critical Better*
iOS Devices		
iPhone 15 Plus	-82.4%	Critical Better
iPhone 12	-84.9%	Critical Better
iPad Pro	-82.4%	Critical Better
iPhone 8	-85.2%	Critical Better
iPhone 14 Pro Max	-81.5%	Critical Better
iOS Mean	*-83.3%*	*Critical Better*

Key Findings

The most significant finding to highlight for these comparisons is that KMP consistently outperforms its native counterpart for all devices and image counts. KMP is 58.0% faster on Android and 66.6% faster on iOS when rendering 10 images. For the rendering of 50 images, the performance difference rises even more to 74.4% faster on Android and 83.3% faster on iOS.

In addition to this, another notable finding is that the results are consistently weighed in favor of one specific platform and does not depend, as much as the other tests, on factors like hardware differences between different devices.

These findings highlight KMP's superior performance in image rendering tasks.

4.1.5 Application Startup Time

In this section, the results of the app startup tests will be presented, with a focus on the three metrics: time to first frame, time to interactive, and total application startup time.

Summary of Results

Figure 13 and Table 13 below show the percentage differences in time to first frame during app startup between KMP and native implementations.

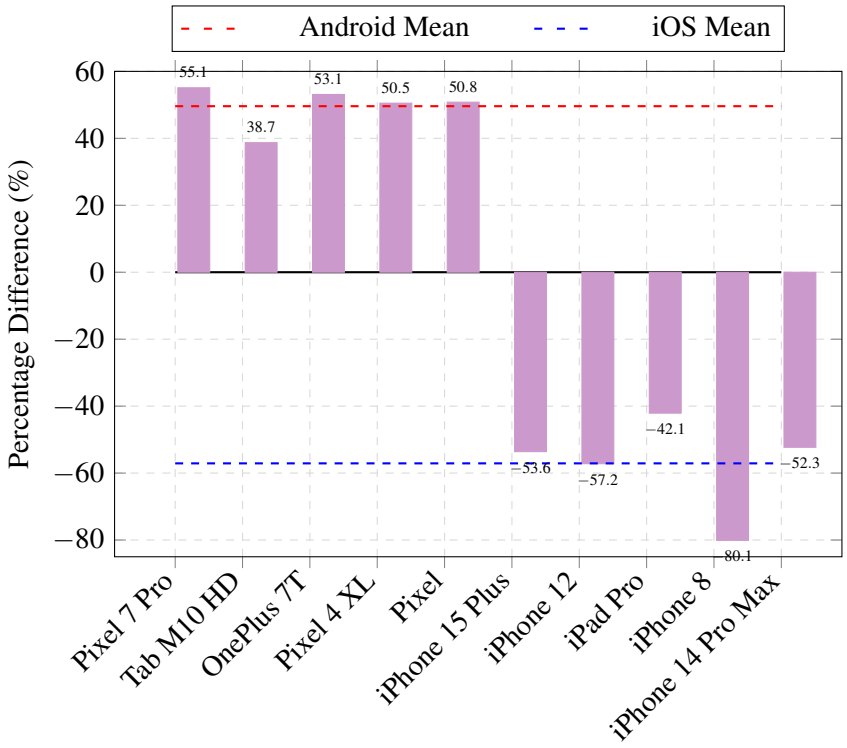


Figure 13: App Startup: Percentage Difference for Time to First Frame (KMP vs. Native)

Table 13 App Startup: Percentage Difference for Time to First Frame (KMP vs. Native)

Device	**Time to First Frame (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	+55.1%	Significant Worse
Lenovo Tab M10 HD	+38.7%	Significant Worse
OnePlus 7T Pro	+53.1%	Significant Worse
Google Pixel 4 XL	+50.5%	Significant Worse
Google Pixel	+50.8%	Significant Worse
Android Mean	*+49.6%*	*Significant Worse*
	iOS Devices	
iPhone 15 Plus	-53.6%	Critical Better
iPhone 12	-57.2%	Critical Better
iPad Pro	-42.1%	Significant Better
iPhone 8	-80.1%	Critical Better
iPhone 14 Pro Max	-52.3%	Critical Better
iOS Mean	*-57.1%*	*Critical Better*

Figure 14 and Table 14 below show the percentage differences in time to interactive during app startup between KMP and native implementations.

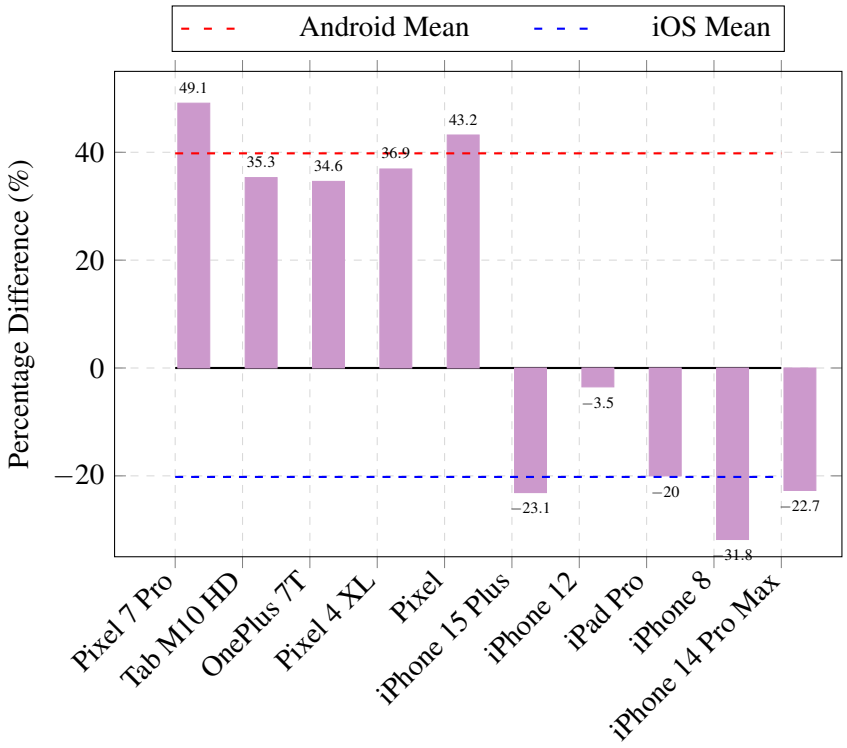


Figure 14: App Startup: Percentage Difference for Time to Interactive (KMP vs. Native)

Table 14 App Startup: Percentage Difference for Time to Interactive (KMP vs. Native)

Device	**Time to Interactive (%)**	**Category**
	Android Devices	
Google Pixel 7 Pro	+49.1%	Significant Worse
Lenovo Tab M10 HD	+35.3%	Significant Worse
OnePlus 7T Pro	+34.6%	Significant Worse
Google Pixel 4 XL	+36.9%	Significant Worse
Google Pixel	+43.2%	Significant Worse
Android Mean	*+39.8%*	*Significant Worse*
	iOS Devices	
iPhone 15 Plus	-23.1%	Minor Better
iPhone 12	-3.5%	Negligible
iPad Pro	-20.0%	Minor Better
iPhone 8	-31.8%	Significant Better
iPhone 14 Pro Max	-22.7%	Minor Better
iOS Mean	*-20.2%*	*Minor Better*

Figure 15 and Table 15 below show the percentage differences in total startup time between KMP and native implementations.

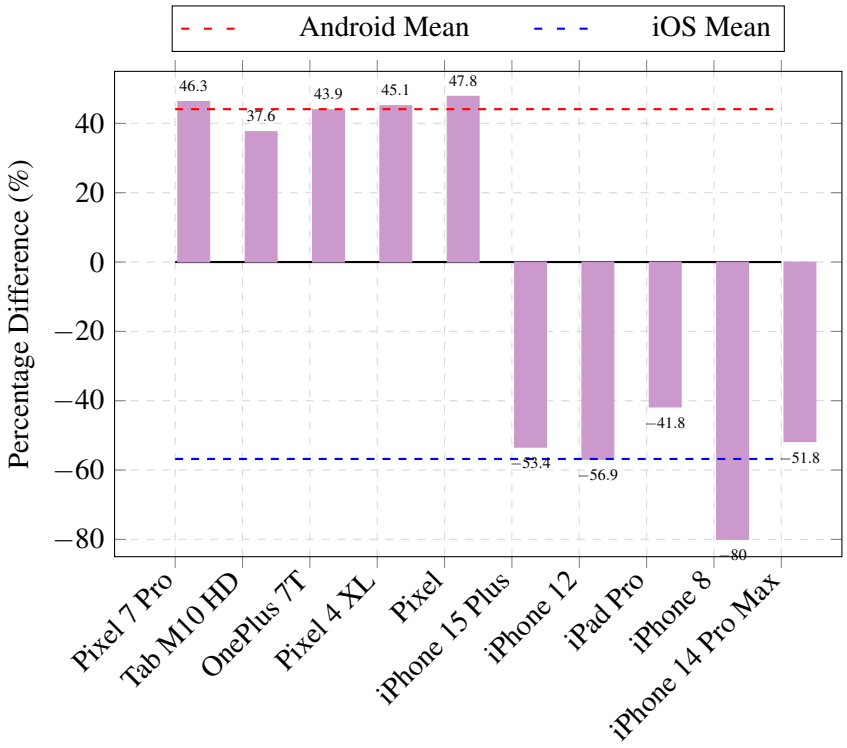


Figure 15: App Startup: Percentage Difference for Total Startup Time (KMP vs. Native)

Table 15 App Startup: Percentage Difference for Total Startup Time (KMP vs. Native)

Device	Total Startup Time (%)	Category
	Android Devices	
Google Pixel 7 Pro	+46.3%	Significant Worse
Lenovo Tab M10 HD	+37.6%	Significant Worse
OnePlus 7T Pro	+43.9%	Significant Worse
Google Pixel 4 XL	+45.1%	Significant Worse
Google Pixel	+47.8%	Significant Worse
Android Mean	*+44.1%*	*Significant Worse*
	iOS Devices	
iPhone 15 Plus	-53.4%	Critical Better
iPhone 12	-56.9%	Critical Better
iPad Pro	-41.8%	Significant Better
iPhone 8	-80.0%	Critical Better
iPhone 14 Pro Max	-51.8%	Critical Better
iOS Mean	*-56.8%*	*Critical Better*

Key Findings

The app startup performance tests unveil several insights about the differences in application startup scenarios between KMP and native implementations.

The most significant finding is the distinct difference in performance between Android and iOS. Across all three of the metrics measured, KMP struggles to keep up with native Android, with KMP performing significantly worse. The KMP iOS application, on the other hand, outperforms its native counterpart in all three test categories, showing that there are some clear platform-specific differences here.

Additionally, the performance differences in these tests seem to be more pronounced on older devices. The two tablets from each of the OS categories (iPad and Lenovo Tab) also seems to struggle compared to their mobile device competitors.

These findings highlight the complexities between platform-specific startup processes, hardware capabilities, and the different phases of app startup in determining the performance characteristics of KMP versus native implementations.

4.1.6 Binary Application Size

For this section, the binary application sizes of all four applications will be presented. The application sizes were measured after performing release builds and installing the applications on the devices. The measurements were taken directly from the installed applications on the devices to ensure accurate representation of the final binary sizes that end users would experience.

Summary of Results

Figure 16 and Table 16 below show the binary application sizes for both KMP and native implementations across platforms.

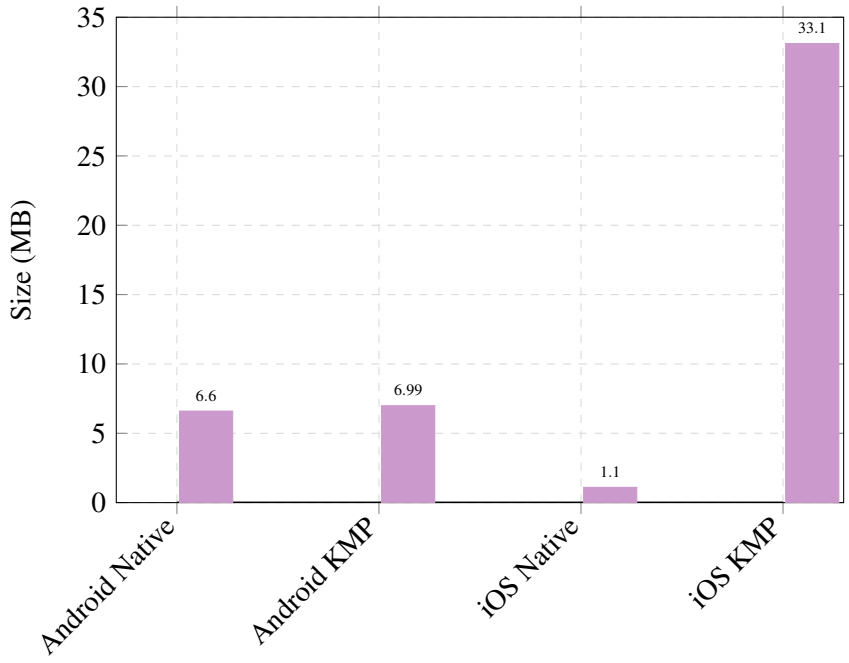


Figure 16: Binary Application Size Comparison (KMP vs. Native)

Table 16 Binary Application Size Comparison (KMP vs. Native)

Platform	Size (MB)	Category
Android Native	6.6	Baseline
Android KMP	6.99	Minor Increase
Android Difference	*+5.9%*	*Minor Increase*
iOS Native	1.1	Baseline
iOS KMP	33.1	Critical Increase
iOS Difference	*+2909.1%*	*Critical Increase*

Key Findings

The measurement of the binary application sizes of the different implementations provide some insights into the trade-offs between cross-platform development and platform-specific optimizations.

As we can see, the native applications closely follow each other in terms of binary application size. The two KMP implementations, on the other hand, showcase a critical difference in sizes, where the Android KMP and iOS KMP implementations have binary app sizes of $\sim$7 MB and $\sim$33 MB, respectively.

Furthermore, when we look at the iOS implementations, we can identify a clear and distinct difference with KMP showing a 33 MB increase in binary app size compared to its native counterpart.

These findings shine a light on how much binary application sizes can vary depending on the adopted implementation approach.

4.1.7 Summary of Results

The comprehensive performance tests and analysis across the different devices revealed some platform-specific differences between KMP and native implementations.

For CPU performance, KMP showed varying results across platforms, performing $\sim$26% slower on Android but $\sim$81% faster on iOS for execution time, with similar patterns in CPU utilization.

When measuring the memory management capabilities of the implementations, iOS display extreme variations, particularly in large object allocation, where native only allocated around $\sim$3 MB of memory and KMP allocated the expected 100 MB. Android, on the other hand, exhibited minimal differences in memory usage.

Additionally, both platforms show a relatively minor difference in network performance, with Android KMP being slightly slower ($\sim$3%) and iOS KMP slightly faster ($\sim$7%) for large payloads.

Also, KMP consistently outperform native implementations in rendering images across both platforms, showing 58% faster rendering on Android and $\sim$67% faster on iOS for small

image counts. For large image counts, the difference in rendering time is even greater, with the Android and iOS KMP implementations rendering images ∼74% and ∼83% faster, respectively, than their native counterparts.

In the application startup tests, KMP exhibit significant platform-specific differences performing ∼44% slower on Android but ∼57% faster on iOS.

Lastly, one of the most significant differences between the implementations is in binary size, where KMP shows a minor 5.9% increase on Android but a critical almost 3000% increase on iOS compared to native implementations (∼33 MB vs. ∼1 MB).

These results highlight the complex trade-offs between cross-platform and platform-specific development, with KMP showing particular strengths in image rendering and iOS performance, but some downsides in binary size and Android startup times.

5 Discussion

In this section, the findings and their implications are discussed, and how they could be interpreted. The analysis looks into potential reasons behind the performance differences observed, both the technical and the architectural factors that may have affected the end results. This includes examining the underlying system architectures of both KMP and native implementations to try and explain the varying performance measurements across the different tests.

Additionally, this section will discuss how the findings in this thesis might influence development decisions and what they might suggest about the current state of cross-platform development using KMP.

5.1 Addressing the Research Questions

Based on the testing and analysis performed in this study, we can now address the research questions that started this investigation:

RQ1 — In which scenarios does Kotlin Multiplatform outperform or underperform compared to native implementations?

RQ2 — What factors contribute to variations in performance between Kotlin Multiplatform and native approaches?

RQ3 — In which use cases is Kotlin Multiplatform the better choice, and why?

5.1.1 RQ1: Performance Scenarios

The study reveals that KMP outperforms native implementations in some scenarios—one being image rendering performance across both platforms—where it is 58% to around ∼83% faster. For application startup times on iOS, KMP is ∼57% faster and ∼7% faster in network performance on iOS for large payloads.

On the other hand, KMP underperforms in startup times on Android where it is ∼44% slower. It also lacks in executing networking and CPU-intensive operations on Android where it is ∼3% and ∼26% slower, respectively.

5.1.2 RQ2: Contributing Factors

The performance differences between KMP and native implementations can be traced to several important factors that perhaps, combined, can explain some of the variations. First, platform-specific memory management approaches play a major role. Each platform use its

own optimized methods for handling memory allocation and recycling which impacts the performance when handling large amounts of data.

Additionally, the platforms use different sorting algorithms, which could lead to significant differences in performance when handling data or executing CPU-intensive operations. The integration with platform-specific lifecycle and initialization processes could also introduce additional overhead in certain situations.

Also, the efficiency of Compose Multiplatform's rendering pipeline, which seems to be particularly effective in rendering images, could display some variations in performance depending on the architecture of the underlying platform—whether it is iOS or Android.

Lastly, certain platform-specific optimization and architectural differences affect the overall performance, where each platform has its own advantages and disadvantages.

5.1.3 RQ3: Optimal Use Cases

Based on our results in this study, KMP looks to be a suitable choice in several scenarios. First, it is a great approach for applications that require identical appearance and behavior across different platforms, where KMP's UI framework provide significant advantages. Especially noteworthy is its superior performance in image rendering, which makes it an ideal choice for applications with lots of visual elements.

Additionally, KMP proves to be effective for applications with moderate complexity and those who do not require a lot of platform-specific implementations. In those cases, the efficient development process and code sharing properties of KMP could provide great advantages without sacrificing performance.

Despite KMP's strengths in certain aspects, there are situations where it may not be the best choice. On Android, providing a quick startup process in an application could be crucial for user experience which points to the native implementation being the better approach to take.

When expecting CPU-intensive operations on Android, KMP's native counterpart also seem to perform better. Lastly, applications that require significant amounts of platform-specific functionality, may encounter difficulties with KMP's abstraction layer, which implies that the more suitable approach would be the native implementation.

5.2 Analysis of Test Results

The performance testing benchmarks brought forward in this study reveal several interesting findings and platform-specific anomalies that will be highlighted in this section.

5.2.1 Platform-Specific Memory Management Differences

One of the most interesting aspects of the findings is the huge difference in how the iOS and Android platforms handle memory allocations, particularly for large objects. As previously noted, the iOS native implementation shows an unusually low memory delta of $\sim$3.4 MB for large object allocations, while both Android native and KMP implementations display the expected $\sim$100 MB allocation. This difference in memory allocation shines a light on

fundamental differences in how the different platforms handle memory allocations.

The iOS devices display a very different memory usage pattern compared to the Android Native and KMP implementations. Despite our test implementation attempting to force physical memory allocation by writing to the allocated memory blocks, iOS still exhibits a much lower memory delta of $\sim$3.4 MB compared to the expected $\sim$100 MB. This indicates that iOS may use a combination of virtual memory management and deferred physical memory allocation, even when applications attempt to force immediate allocation programmatically.

The Android Native and KMP implementations, on the other hand, showcase a more direct approach to allocating memory. The requested memory is immediately allocated and reflected in the system, making the memory allocation operations easier to predict and measure. The reason for this might be because they are implemented using Kotlin, which most likely handles memory allocation in a different manner than iOS.

These differences in memory management approaches make direct comparisons between platforms challenging and suggest that developers should consider platform-specific memory optimization strategies when implementing cross-platform solutions. The iOS behavior, while potentially more memory-efficient, can make memory usage metrics less predictable and harder to measure accurately.

5.2.2 Sorting Algorithm Impact on CPU Performance

The CPU performance tests revealed a significant performance advantage for KMP on iOS (81.2% faster) while showing a disadvantage on Android (25.5% slower). This performance difference could partially be attributed to the different sorting algorithms used by each platform. Swift's default sorting method uses Timsort, which is a hybrid sorting algorithm that combines insertion sort and merge sort. Kotlin's implementation uses a modified merge sort, which has different performance characteristics, particularly with different data distributions.

Although the choice of sorting algorithm may seem like a minor implementation detail, it might have had a bigger impact than expected on the performance of the test benchmark. This finding emphasizes the importance of understanding underlying platform-specific characteristics when evaluating cross-platform development.

5.2.3 Image Rendering Performance Anomaly

Perhaps the most surprising finding is KMP's superior performance in image rendering across both platforms, with improvements in rendering speed ranging from 58.0% to 83.3% faster than native implementations. These results suggest that KMP's Compose Multiplatform framework might utilize more efficient rendering pipelines than the native UI frameworks.

Additionally, since KMP performed substantially better than its native counterparts, it could remove the need for creating platform-specific solutions and still achieve superior performance.

Lastly, this performance advantage KMP holds seems to be more pronounced when rendering a larger quantity of images, which may suggest that KMP's Compose Multiplatform

framework is more efficient at handling multiple images simultaneously or that its image rendering system becomes more efficient as the number of images increases.

The superior performance of KMP in the image rendering tasks for both platforms challenges common assumptions about native implementations always providing better performance. This finding suggests that cross-platform solutions can not only match but, in some scenarios, even exceed native performance in specific scenarios, particularly in graphics-intensive applications.

5.2.4 Network Performance Consistency

When it comes to the network performance benchmarks, which proved to be the most even of comparisons, there is only a small difference between KMP and native implementations. The KMP implementation on Android is just slightly slower at $\sim$3% and on iOS slightly faster at $\sim$7% for large payloads. The consistency of these results may suggest that network operations on the different implementations are much less affected by the implementation approach itself, and are more dependent on the OS's built-in network components.

Furthermore, the values we gather (less than 7% faster/slower) show that KMP's networking implementation using Ktor delivers a performance that is comparable to the platform-specific implementations of network libraries.

The consistent network performance across the different implementations highlights that certain aspects of application development may be less affected by the choice of development approach. This finding is particularly relevant for applications where network operations are important, as it suggests that cross-platform solutions can deliver an on-par performance to native implementations.

5.2.5 Platform Dependency in Startup Performance

The measured startup times of the implementations highlight a clear difference in how KMP performs depending on the platform. On Android, KMP displays a significant dip in performance with being $\sim$44% slower than its native counterpart. The iOS version of KMP shows, on the other hand, an even greater increase in performance by being $\sim$57% faster than the native iOS application.

These differences may suggest that KMP's startup performance is significantly influenced by how well it integrates with each platform's native lifecycle and initialization processes. The significant performance advantage on iOS could be a sign that KMP's underlying architecture aligns more naturally with iOS's application lifecycle. Additionally, the slower startup performance on Android might be an indicator that KMP requires more time to adapt to Android's different startup requirements.

The significant variations in startup performances between platforms highlight how platform-specific features may, in some cases, influence how effective the cross-platform implementations are. This finding adds another thing to the list of things to consider when evaluating cross-platform development approaches.

5.2.6 Binary Size Implications

The binary size measurements, while seemingly arbitrary, reveal a significant difference in how KMP affects application size across the different platforms. While KMP on Android only displays a minor application size increase of $\sim$6%, the iOS counterpart of KMP shows a staggering $\sim$3000% increase.

These observations suggest that on Android, KMP can use the platform's native Kotlin support and shared libraries more efficiently, resulting in minimal overhead. The iOS implementation, however, requires additional components to make Kotlin work with iOS, including a bundled framework and special code to connect with iOS's native features. These differences in architecture can explain the lack in performance—the iOS version needs to include the KMP runtime and additional tools to make Kotlin work on its OS—which may explain why it is so much larger in size, while the Android version can use the platform's native Kotlin support.

Looking at all of these findings, we can suggest that the choice between KMP and native development should be made with careful consideration of the specific platform requirements and what the performance priorities of the application are. While KMP showcases promising results in certain areas, like image rendering and general performance on the iOS version, it also introduces significant challenges in other areas, like the size of the app on iOS and the startup times on Android.

5.3 Development Experience Analysis

Based on the experience of developing the benchmark applications presented in this study, KMP proved to be a smooth and efficient way of developing applications, especially for developers already familiar with Kotlin and its UI toolkit Jetpack Compose. Being able to write code that can be shared across the different platforms while maintaining native performance made the development workflow relatively straightforward.

The development process is surprisingly seamless for several reasons. Firstly, the approach of utilizing a shared codebase removes the need for writing the same code twice for the two platforms, significantly reducing the development time and the risk of potential inconsistencies in the implementation. Secondly, the environment in which the apps are developed, Android Studio, offered great support for KMP development, which makes the whole development process easier.

Having said that, I believe it is important to note that this positive experience is probably, in part, due to the relatively simple nature of the test application. It does not require the implementation of many platform-specific components or complex UI elements, which are common in more complex applications. What this may suggest is that KMP might be particularly well-suited for applications that:

1. Require identical appearance and behaviour across different platforms
2. Is not too complex and do not require a lot of platform-specific features
3. Can benefit from shared code, while still maintaining native-like performance

These insights, along with the test result analysis performed, provide a more complete picture of when and how KMP might be the right choice for mobile application development.

5.4 Limitations of the Study

Although this study provide insights into the performance of KMP compared to native implementations in Android and iOS, some clear limitations should be acknowledged.

The first, and one of the more significant limitations that directly affected this study, is the inability to conduct a meaningful and proper energy consumption testing. While it is quite an important aspect of modern mobile applications, measuring it proves challenging, especially on the iOS side. While gathering detailed data on energy consumption and battery usage on Android is very straightforward, accessing the same metrics using Swift and Xcode is a surprisingly difficult task. Xcode Instruments provides limited access to its battery usage information, which makes a fair comparison between the implementations not feasible. These difficulties are the reason a comparison of energy consumption and battery usage between the implementations was left out.

Furthermore, the test applications in this study are intentionally developed with simplicity in mind and with a focus on specific raw performance aspects. No particular exploration into platform-specific optimizations, which could improve the performance, is made. Although this type of approach may allow for more controlled and comparable environments, it may not fairly represent the performance characteristics and the complexity of real-world applications.

Additionally, the performance measurements in this study are based on the specific implementations of the benchmark applications. Developers trying to recreate these results might use a different approach, which may, in turn, produce different performance results. The findings of this study should be viewed in the context of the specific implementation choices made.

Similarly, the study's focus is on immediate performance metrics and not long-term performance. Aspects like memory leaks or performance under extended use, aspects that are important for realistic, real-world applications, are not investigated.

Moreover, the study does not put any particular focus on the user experience end of KMP development, making it difficult to provide any recommendations for the UI design aspects. For example, just because an app can quickly render a large quantity of images or because it is able to function well without allocating a lot of memory, does not mean that it automatically leads to a better user interaction.

Finally, the threshold-based categorization used to classify performance differences (negligible, minor, significant, and critical) is another limitation of this study. While these thresholds are useful to provide a structured way to interpret the results, they are quite random and may not accurately reflect the real-world impact of performance differences. For example, a 15% difference in one type of test benchmark might have a more significant impact on user experience than a 30% difference in another. This categorization approach should be viewed as a simplified framework used for analysis rather than proof of performance measurement significance. These limitations should be considered when interpreting the study's findings and making decisions about the adoption of KMP for mobile application development.

6 Conclusion

The objective of this study is to investigate the potential performance differences of Kotlin Multiplatform when compared to its native counterparts in Android and iOS in various testing benchmarks. Through systematic testing and analysis, we have gained valuable insight into the strengths and limitations of KMP as a cross-platform development framework.

The study reveals several significant platform-specific performance differences. KMP displays a greater performance when rendering images across both platforms. On iOS, KMP highlights some advantages in startup time where it is $\sim$57% faster, and for CPU performance, being $\sim$81% faster. However, on Android, the results points more towards KMP being slower in terms of startup times where it takes $\sim$44% longer and a $\sim$26% slower CPU performance. Some of the results also highlight fundamental underlying differences in the different implementations. For example, the almost 3000% larger application size on the iOS version of KMP compared to its native counterpart, which, as discussed before, may be due to additional components to make Kotlin work with iOS. Important to note is that this study does not investigate if and how this difference in binary size changes if the application is scaled up/down.

The findings this study brings forward suggest that KMP is a good fit for applications requiring a consistent appearance across platforms, applications with a focus on image rendering, and projects prioritizing development efficiency and the sharing of code. However, the choice to adopt KMP as a development approach should be made with careful consideration of the performance priorities of the project and potential platform-specific requirements.

In conclusion, although KMP demonstrates significant advantages in certain areas, it also presents challenges in others. The results of this study show that KMP is a viable cross-platform development approach, but developers must carefully consider the benefits of the code sharing aspects to the potential performance impacts in various areas when considering KMP for future projects.

6.1 Future Work

Future research could improve upon this research by expanding the scope of testing and include a more diverse set of devices and platforms, and incorporating additional performance metrics and user experience considerations. An investigation into energy consumption and battery usage data across the different implementation approaches could also be useful.

Additionally, an interesting area of exploration is the evaluation of the user experience aspects of applications developed using KMP. Another aspect worth looking into is how the performance differences vary when utilizing different platform-specific optimizations and libraries.

Lastly, research into how KMP performs when applied to bigger and more complex implementations that more closely resembles real-world applications would be beneficial for developers and organizations considering KMP as a development approach.

References

[1] Akinlolu Adekotujo, Adedoyin Odumabo, Ademola Adedokun, and Olukayode Aiyeniko. A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios. *International Journal of Computer Applications*, 176(39):16–23, 2020.

[2] Luciano Baresi, Massimiliano Di Penta, Giovanni Quattrocchi, and Damian Andrew Tamburri. How have ios development technologies changed over time? a study in open-source. In *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '24, page 33–42, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3647632.3647988.

[3] Rohini Deshmane. Introduction to kotlin. Accessed: 2025-02-11. URL: https://medium.com/@rohinideshmane.21/introduction-to-kotlin-5f39b31610e0.

[4] Android Developers. Develop android apps with kotlin. Accessed: 2025-03-12. URL: https://developer.android.com/kotlin.

[5] DDI Development. Pros and cons of ios app development. Accessed: 2025-02-28. URL: https://ddi-dev.com/blog/programming/pros-and-cons-of-ios-app-development/.

[6] Eric Engström and Ludvig Näslund. Navigating the mobile landscape : A comparative study of native and cross-platform development approaches in swift, kotlin, and kotlin multiplatform, 2024.

[7] GeeksforGeeks. Introduction to android development. Accessed: 2025-02-10. URL: https://www.geeksforgeeks.org/introduction-to-android-development/.

[8] Kent German. A brief history of android phones. Accessed: 2025-02-10. URL: https://www.cnet.com/tech/mobile/a-brief-history-of-android-phones/.

[9] Michał Guśpiel. Mobile app development using kotlin multiplatform, 2024. URL: https://www.theseus.fi/handle/10024/852201.

[10] DI FH Norbert Haberl. Cross platform development possibilities and drawbacks of the xamarin platform. *FH Joanneum, University of Applied Sciences*, 2015.

[11] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance optimization for mobile applications. *IEEE Transactions on Software Engineering*, 48(8):2879–2904, 2022. URL: https://ieeexplore.ieee.org/abstract/document/9397392, doi:10.1109/TSE.2021.3071193.

[12] IBM. What is mobile application development? Accessed: 2025-02-10. URL: https://www.ibm.com/think/topics/mobile-application-development.

[13] The PYPL Index. Top ide index. Accessed: 2025-02-11. URL: https://pypl.github.io/IDE.html.

[14] JetBrains. What is cross-platform mobile development?, 2025. Accessed: 2025-02-10. URL: https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-mobile-development.html.

[15] Nikhil Kodali. The introduction of swift in ios development: Revolutionizing apple's programming landscape. *NeuroQuantology*, 12(4):471–477, 2014.

[16] Lauren. Mobile app download statistics & usage statistics (2025). Accessed: 2025-02-10. URL: https://buildfire.com/app-statistics/.

[17] Walter T. Nakamura, Edson Cesar de Oliveira, Elaine H.T. de Oliveira, David Redmiles, and Tayana Conte. What factors affect the ux in mobile apps? a systematic mapping study on the analysis of app store reviews. *Journal of Systems and Software*, 193:111462, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0164121222001509, doi:10.1016/j.jss.2022.111462.

[18] Ekaterina Petrova. Update on the name of kotlin multiplatform. Accessed: 2025-02-28. URL: https://blog.jetbrains.com/kotlin/2023/07/update-on-the-name-of-kotlin-multiplatform/.

[19] Payal N. Ingole Priyanka V. Kanoi. History and features of a developer tool in ios: Xcode. Accessed: 2025-02-25. URL: https://ijarcs.info/index.php/Ijarcs/article/download/1712/1700/3399.

[20] C.P Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, 2012. doi:10.1109/INDCON.2012.6420693.

[21] Jacob Ras. Flutter versus native versus kmp: a performance comparison, 2023. Accessed: 2025-02-04. URL: https://github.com/jacobras/flutter-vs-native-vs-kmp.

[22] Teresa Reidt. A brief history of android studio. Accessed: 2025-02-11. URL: https://emteria.com/learn/history-android-studio.

[23] Kunal Relan. *Introduction to iOS*, pages 1–12. Apress, Berkeley, CA, 2016. doi:10.1007/978-1-4842-2355-0_1.

[24] 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.*, 28(2), February 2025. doi:10.1145/3712308.

[25] Amazon Web Services. What is an ide (integrated development environment)? Accessed: 2025-02-11. URL: https://aws.amazon.com/what-is/ide/.

[26] Anna Skantz. Performance evaluation of kotlin multiplatform mobile and native ios development in swift. Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2023.

[27] Nikola Stanic and Stefan Ćirković. Analysis of approaches to developing kotlin multiplatform applications and their impact on software engineering. In *10th International Scientific Conference Technics, Informatics and Education-TIE 2024*. Faculty of Technical Sciences Čačak, University of Kragujevac, 2024.

[28] Backlinko Team. iphone vs android statistics. Accessed: 2025-02-10. URL: https://backlinko.com/iphone-vs-android-statistics.

[29] Tpoint Tech. Features of java. Accessed: 2025-02-11. URL: https://www.tpointtech.com/features-of-java.

[30] Tpoint Tech. History of java. Accessed: 2025-02-11. URL: https://www.tpointtech.com/history-of-java.

[31] Capitol Technology University. A brief history of mobile apps. Accessed: 2025-02-10. URL: https://www.captechu.edu/blog/brief-history-of-mobile-apps.

[32] Spyridon Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. 09 2013. doi:10.1145/2490257.2490292.

[33] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, page 213–220, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2490257.2490292.

[34] Katherine Chou Xavier Ducrohet, Tor Norbye. Android studio: An ide build for android. Accessed: 2025-02-11. URL: https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html.

A Appendix: Additional Resources

A.1 Source Code Repository

The complete source code for this study is available in a public repository[1]. The repository contains the full implementation of all test cases and is organized into three main projects:

- `MasterThesisKMP/` - The Kotlin Multiplatform implementation
- `MasterThesisIOSNative/` - The iOS native implementation
- `MasterThesisAndroidNative/` - The Android native implementation

The project includes detailed documentation in its README file, explaining the setup process.

[1] `https://github.com/ericornkloo/kmp-native-comparison`

B Appendix: Test Results

B.1 CPU Performance Test Results

Table 17 CPU Performance Test Results: Mean Values by Device

Device	Execution Time (ms)		CPU Utilization (%)	
	Native	**KMP**	**Native**	**KMP**
Android Devices				
Google Pixel 7 Pro	255.5	211.6	17.6	17.5
Lenovo Tab M10 HD	643.6	980.1	20.1	27.0
OnePlus 7T Pro	220.8	284.0	15.1	18.9
Google Pixel 4 XL	145.6	180.7	11.6	11.1
Google Pixel	611.7	853.4	34.3	41.2
Android Mean	*375.4*	*502.0*	*19.7*	*23.1*
iOS Devices				
iPhone 15 Plus	689.0	111.0	59.0	30.2
iPhone 12	573.4	123.4	53.1	29.8
iPad Pro	346.4	93.6	34.1	14.0
iPhone 8	1492.0	198.2	70.6	32.5
iPhone 14 Pro Max	663.6	106.4	57.5	27.0
iOS Mean	*752.9*	*126.5*	*54.9*	*26.7*

B.2 Memory Management Test Results

Table 18 Memory Management Test Results: Small Objects (10000)

Device	Memory Delta (MB)		Allocation Time (ms)	
	Native	**KMP**	**Native**	**KMP**
Android Devices				
Google Pixel 7 Pro	102.1	97.8	106.4	126.8
Lenovo Tab M10 HD	100.8	102.8	171.1	130.8
OnePlus 7T Pro	100.8	100.5	74.4	114.8
Google Pixel 4 XL	120.5	116.9	127.3	150.2
Google Pixel	103.2	99.5	146.9	132.6
Android Mean	*105.5*	*103.5*	*125.2*	*131.0*
iOS Devices				
iPhone 15 Plus	142.6	97.7	47.6	80.6
iPhone 12	156.1	97.7	62.5	89.2
iPad Pro	156.2	97.7	56.8	113.2
iPhone 8	156.2	97.7	89.5	121.8
iPhone 14 Pro Max	142.6	97.7	49.4	72.4
iOS Mean	*150.7*	*97.7*	*61.2*	*95.4*

Table 19 Memory Management Test Results: Large Objects (100)

Device	Memory Delta (MB)		Allocation Time (ms)	
	Native	**KMP**	**Native**	**KMP**
Android Devices				
Google Pixel 7 Pro	100.4	100.4	1.7	1.6
Lenovo Tab M10 HD	100.4	100.4	0.7	1.4
OnePlus 7T Pro	100.4	100.4	1.6	1.0
Google Pixel 4 XL	100.4	100.4	0.8	1.0
Google Pixel	100.4	100.4	2.9	1.6
Android Mean	*100.4*	*100.4*	*1.5*	*1.3*
iOS Devices				
iPhone 15 Plus	3.7	100.0	1.6	20.0
iPhone 12	3.1	100.0	2.2	20.2
iPad Pro	3.1	100.0	2.7	21.6
iPhone 8	3.2	100.0	3.6	21.0
iPhone 14 Pro Max	3.7	100.0	2.2	18.0
iOS Mean	*3.4*	*100.0*	*2.5*	*20.2*

B.3 Network Performance Test Results

Table 20 Network Performance Test Results

Device	Small Payload (ms)		Large Payload (ms)	
	Native	**KMP**	**Native**	**KMP**
	Android Devices			
Google Pixel 7 Pro	271	259	347	488
Lenovo Tab M10 HD	219	338	384	464
OnePlus 7T Pro	271	315	469	432
Google Pixel 4 XL	222	232	406	386
Google Pixel	421	295	431	682
Android Mean	*281*	*288*	*407*	*490*
	iOS Devices			
iPhone 15 Plus	234	227	538	413
iPhone 12	279	228	404	341
iPad Pro	214	191	392	380
iPhone 8	265	254	410	394
iPhone 14 Pro Max	279	415	409	461
iOS Mean	*254*	*263*	*431*	*398*

B.4 Image Rendering Test Results

Table 21 Image Rendering Test Results

Device	Small Count (ms)		Large Count (ms)	
	Native	**KMP**	**Native**	**KMP**
	Android Devices			
Google Pixel 7 Pro	4144	1457	5332	1923
Lenovo Tab M10 HD	3330	1858	8812	2788
OnePlus 7T Pro	3101	1278	10560	1511
Google Pixel 4 XL	5218	1925	11201	2913
Google Pixel	4246	1739	12604	2494
Android Mean	*4008*	*1651*	*9702*	*2326*
	iOS Devices			
iPhone 15 Plus	3452	1185	6698	1181
iPhone 12	2538	1139	8041	1213
iPad Pro	2765	1142	7987	1406
iPhone 8	4769	1155	9908	1470
iPhone 14 Pro Max	4970	1111	6461	1196
iOS Mean	*3699*	*1146*	*7819*	*1293*

B.5 Application Startup Time Test Results

Table 22 App Startup Test Results: Time to First Frame

Device	Time to First Frame (ms)	
	Native	**KMP**
Android Devices		
Google Pixel 7 Pro	292	453
Lenovo Tab M10 HD	1360	1886
OnePlus 7T Pro	358	548
Google Pixel 4 XL	396	596
Google Pixel	942	1421
Android Mean	*670*	*981*
iOS Devices		
iPhone 15 Plus	248	115
iPhone 12	892	382
iPad Pro	1407	815
iPhone 8	473	94
iPhone 14 Pro Max	222	106
iOS Mean	*648*	*302*

Table 23 App Startup Test Results: Time to Interactive

Device	Time to Interactive (ms)	
	Native	**KMP**
Android Devices		
Google Pixel 7 Pro	383	571
Lenovo Tab M10 HD	1517	2053
OnePlus 7T Pro	489	658
Google Pixel 4 XL	517	708
Google Pixel	1089	1559
Android Mean	*799*	*1110*
iOS Devices		
iPhone 15 Plus	562	432
iPhone 12	1206	1164
iPad Pro	1789	1431
iPhone 8	604	412
iPhone 14 Pro Max	537	415
iOS Mean	*940*	*771*

Table 24 App Startup Test Results: Total Startup Time

Device	**Total Startup Time (ms)**	
	Native	**KMP**
Android Devices		
Google Pixel 7 Pro	322	471
Lenovo Tab M10 HD	1382	1902
OnePlus 7T Pro	387	557
Google Pixel 4 XL	417	605
Google Pixel	970	1434
Android Mean	*694*	*994*
iOS Devices		
iPhone 15 Plus	249	116
iPhone 12	895	386
iPad Pro	1410	820
iPhone 8	485	97
iPhone 14 Pro Max	224	108
iOS Mean	*653*	*305*