Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Performance Evaluation of Kotlin Multiplatform on iOS

Quantitative comparison between Kotlin and Compose Multiplatform against Native Swift for UI Rendering Hardware Interactions

VANJA VIDMARK

Performance Evaluation of Kotlin Multiplatform on iOS

Quantitative comparison between Kotlin and Compose Multiplatform against Native Swift for UI Rendering Hardware Interactions

VANJA VIDMARK

Master's Programme, Computer Science, 120 credits
Date: August 12, 2025

Supervisor: Elena Troubitsyna
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Framna AB
Swedish title: Prestandautvärdering av Kotlin Multiplatform på iOS
Swedish subtitle: Jämförelse av Kotlin och Compose Multiplatform med native Swift för UI-rendering och hårdvaruinteraktioner

Abstract

Cross-platform development is becoming increasingly popular because it allows code reuse between Android and iOS. In contrast, native development typically requires maintaining separate codebases for each platform. Kotlin Multiplatform enables shared business logic, while Compose Multiplatform extends this to the user interface. However, cross-platform solutions are often associated with lower performance, and the performance of Kotlin and Compose Multiplatform on iOS remains relatively unexplored. Therefore, this thesis evaluates the performance of Kotlin and Compose Multiplatform on iOS in comparison to native Swift. The study focuses on hardware-related operations and UI rendering. To evaluate this, two functionally equivalent applications are implemented, each including six benchmarks: three hardware focused and three UI focused. The performance metrics are execution time, CPU usage, memory consumption, frames per second, and delayed frames. The results show that Kotlin Multiplatform has comparable CPU usage to native Swift in the hardware benchmarks but consumes significantly more memory. Execution time results are mixed, varying by task. In the animation benchmarks, Compose Multiplatform consistently uses more CPU and exhibits frame delays during more complex animations, whereas the native implementation maintains perfect performance with no delayed frames. These findings suggest that Kotlin and Compose Multiplatform are suitable for applications with simple animations and moderate performance requirements, but cannot compare to native Swift in more complex scenarios. The main contributions of this thesis are a reproducible benchmarking methodology and empirical data that can assist developers and organizations in choosing an appropriate development strategy for cross-platform applications.

Keywords

Kotlin Multiplatform, Compose Multiplatform, Cross-platform, iOS, Kotlin, Swift, Native, Benchmarking, Performance evaluation, Mobile

Sammanfattning

Plattformsoberoende utveckling blir allt mer populärt eftersom det möjliggör delning av kod mellan Android och iOS. Detta till skillnad från native-utveckling, som vanligtvis kräver separata kodbaser för varje plattform. Kotlin Multiplatform möjliggör delning av kod, medan Compose Multiplatform utökar detta till att även omfatta ett delat användargränssnitt. Plattformsoberoende lösningar förknippas dock ofta med sämre prestanda än native utveckling, och hur väl Kotlin och Compose Multiplatform presterar på iOS är fortfarande relativt outforskat. Därför utvärderar detta examensarbete prestandan för Kotlin och Compose Multiplatform på iOS i jämförelse med native Swift. Studien fokuserar på hårdvarunära operationer och rendering av användargränssnitt. För att utvärdera detta implementerades två applikationer, var och en med sex tester: tre fokuserade på hårdvara och tre på användargränssnitt. De mått som utvärderas är exekveringstid, CPU-användning, minnesförbrukning och bildfrekvens (frames per second). Resultaten visar att Kotlin Multiplatform har jämförbar CPU-användning med native Swift i hårdvarutesterna, men använder avsevärt mer minne. Resultaten för exekveringstid varierar beroende på uppgift. I användargränssnitt-testerna använder Compose Multiplatform konsekvent mer CPU och uppvisar nedsatt bildfrekvens vid mer komplexa animationer, medan den native-implementerade versionen uppnår perfekt prestanda och bildfrekvens. Dessa resultat tyder på att Kotlin och Compose Multiplatform är lämpliga för applikationer med enklare animationer och måttliga prestandakrav. De huvudsakliga bidragen från detta arbete är en strukturerad och reproducerbar benchmark-metodik samt empirisk data som kan hjälpa utvecklare och organisationer att välja en lämplig utvecklingsstrategi för sina applikationer.

Nyckelord

Kotlin Multiplatform, Compose Multiplatform, Plattformsoberoende, iOS, Kotlin, Swift, Native, Benchmarking, Prestandautvärdering, Mobil

Acknowledgments

I would like to thank my supervisor, Elena Troubitsyna, for her support throughout the process and for providing valuable guidance and feedback. I would also like to thank my examiner, Mads Dam, for his constructive comments and insightful suggestions for improvement.

A special thank you to my supervisor at Framna AB, Jinyan Liu, whose genuine interest in the topic and constant encouragement were truly appreciated. Her technical knowledge was both helpful and inspiring. I would also like to thank Elsa Rimhagen for stepping in as a supervisor and offering great ideas and support.

Stockholm, August 2025
Vanja Vidmark

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

API	Application Programming Interface
ARC	Automatic Reference Counting
CMP	Compose Multiplatform
CPU	Central Processing Unit
FPS	Frames Per Second
GC	Garbage Collection
GCD	Grand Central Dispatch
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
JVM	Java Virtual Machine
KMP	Kotlin Multiplatform
RAM	Random Access Memory
SDK	Software Development Kit
UI	User Interface
VM	Virtual Machine

Chapter 1

Introduction

In modern mobile application development, supporting both Android and iOS is essential, as together they account for over 99% of all smartphone users [1]. Traditionally, this has required maintaining separate codebases for each platform, leading to increased development costs, longer time-to-market, and potential inconsistencies between implementations. To mitigate these challenges, cross-platform frameworks allows developers to share code across multiple platforms. Among the popular cross-platform solutions are React Native, Flutter, and Kotlin Multiplatform (KMP). While React Native and Flutter have been extensively researched and adopted, KMP remains a relatively unexplored alternative, particularly in the context of iOS development.

This thesis explores the performance trade-offs of KMP compared to native Swift for iOS applications, with a specific focus on User Interface (UI) rendering and hardware interactions. The study aims to bridge an existing research gap by evaluating key performance metrics, including execution time, Central Processing Unit (CPU) usage, memory usage, Frames Per Second (FPS) and delayed frames. These insights will be valuable for developers and organizations considering KMP as an alternative to native development for iOS applications.

1.1 Background

Cross-platform development frameworks have gained significant popularity due to their potential to reduce development effort while maintaining high performance. KMP enables developers to write shared business logic in Kotlin while keeping the UI platform-specific [2]. Compose Multiplatform (CMP)

further extends KMP by providing a declarative UI framework similar to Jetpack Compose, allowing also to share the UI across Android, iOS, and other platforms [3]. Together, KMP and CMP offer a complete solution for cross-platform development.

KMP integrates well with Android, as it runs on the Java Virtual Machine (JVM), similar to native Kotlin apps. However its performance on iOS remains uncertain due to differences in compilation, execution, memory management and graphics engine.

Existing studies on KMP for iOS indicate promising but mixed results. One study suggest that KMP's startup time is comparable to Swift but results in a larger app size [4]. Another show that KMP is faster than Swift in certain computational tasks while consuming more CPU and memory [5]. Studies on other cross-platform frameworks such as Flutter and React Native have evaluated UI performance, and hardware access, but similar studies on KMP are lacking. This thesis addresses this research gap by providing a structured benchmark evaluation of KMP against native Swift.

1.2 Problem

In mobile application development, cross-platform frameworks offer the advantage of code reuse across multiple operating systems. However, these solutions are often perceived as having lower performance, difficulties in accessing device hardware and worse user experience, compared to fully native applications [6].

The main problem addressed in this thesis is whether KMP and CMP can provide comparable performance to native Swift for iOS applications. Specifically, the study investigates the impact of KMP on UI performance and hardware interactions, evaluating whether it introduces significant performance overhead.

The reason this study focuses on hardware interactions and UI performance is that both are known to cause issues in cross-platform development. Accessing hardware requires native Application Programming Interfaces (APIs), which can be problematic or even infeasible in some frameworks [6]. While this has been thoroughly studied in frameworks such as Flutter and React Native, it remains less explored in KMP. Similarly, UI performance, particularly animations, is suspected to be weaker in cross-platform solutions. Since animations are a key part of the user experience and have been widely studied in other frameworks, their performance in KMP needs further investigation.

1.2.1 Research Questions

- **RQ1:** Does Kotlin Multiplatform perform comparably to native Swift on iOS when performing hardware-related operations?
- **RQ2:** Does Kotlin Multiplatform and Compose Multiplatform perform comparably to native Swift on iOS in terms of UI-rendering?

1.3 Purpose

The purpose of this thesis is to evaluate KMP and CMP as a viable alternative to native Swift for iOS applications. This thesis is relevant for software engineers, mobile developers, and stakeholders involved in evaluating cross-platform development strategies. It provides empirical data to help them make informed decisions about adopting KMP and CMP. If KMP and CMP can offer comparable performance to native Swift, it could present an attractive alternative that balances development efficiency with high performance.

Beyond its practical implications, this study contributes with empirical data that can support ongoing academic and industry evaluations of cross-platform mobile performance. Additionally, it proposes a structured approach to benchmarking critical hardware interactions, such as camera access, and local file operations, as well as UI performance in animations.

1.4 Goals

The goal of this project is to benchmark KMP and CMP against native Swift for iOS applications. This has been divided into the following sub-goals:

1. **Industry Contribution**: Evaluate KMP's and CMP's suitability for iOS development, providing insights for companies that are considering cross-platform solutions.
2. **Academic Contribution**: Contribute to research on cross-platform frameworks by introducing structured benchmarks for KMP's/CMP's UI and hardware performance.

1.5 Research Methodology

This study employs a quantitative research methodology focusing on empirical performance measurements to evaluate KMP against native Swift

development for iOS. The research combines a literature review, experimental benchmarking, and statistical analysis.

The first phase consists of a literature review to establish the theoretical foundation of cross-platform performance evaluation, identify relevant prior research, and define the research questions.

The second phase involves the development of two functionally equivalent mobile applications: one using KMP and CMP and the other using native Swift. These apps will be deployed on a controlled test environment and performance metrics, including execution time, CPU usage, memory consumption, FPS and delayed frames, will be collected.

In the third phase, the collected data will be analyzed using t-tests to determine whether KMP and CMP introduces significant performance overhead compared to native Swift.

Alternative research methods were considered during the planning of this study. One option was to conduct a qualitative user study where participants would evaluate and compare cross-platform solutions based on usability or perceived performance. However, this approach was discarded in favor of a more objective, benchmark-driven methodology. This decision was made to maintain a cleaner and more focused scope, prioritizing measurable performance characteristics over subjective impressions.

It was also considered to include additional cross-platform frameworks, specifically Flutter, as a third point of comparison. However, this was ultimately excluded due to time constraints.

Finally, testing on both iOS and Android was considered. However, the host company Framna expressed a particular interest in iOS performance. Given that KMP is closely aligned with native Android in terms of runtime and development stack, any performance discrepancies on Android were expected to be minimal. Although introducing a native Android baseline was considered to validate this assumption, it was ultimately excluded due to time constraints.

1.6 Delimitations

This study focuses exclusively on KMP and CMP and its performance in comparison to native development, without considering other cross-platform frameworks. Additionally, the scope is limited to iOS, meaning no comparisons will be made to the performance on Android.

The performance tests will be conducted on a single iPhone to ensure that any observed differences result solely from the comparison between Swift and

KMP/CMP, rather than variations in hardware or iOS versions.

The evaluation will be purely quantitative and limited to execution time, CPU usage, memory usage, FPS, and delayed frames per second.

1.7 Structure of the Thesis

Chapter 2 provides relevant background on mobile development, with a particular focus on cross-platform development, KMP and CMP. Chapter 3 outlines the research methodology and describes the design and execution of the benchmarks. Chapter 4 presents the technical details of the benchmark implementations for both platforms. Chapter 5 presents the measured performance data and analyzes the cause of the performance differences, and lastly, Chapter 6 concludes the findings of the thesis, reflects on the limitations and gives suggestions for future work.

Chapter 2

Background

This chapter provides background on mobile app development, with a focus on cross-platform approaches. It introduces KMP and CMP, describing their architecture and behavior on iOS. Finally it summarizes related work and highlights the research gap addressed in this thesis.

2.1 Mobile App Development

Mobile applications are software designed to run on mobile devices, such as smartphones and tablets. Mobile app development has evolved significantly over the past few decades. The first smartphone, IBM's Simon, introduced built-in applications as early as 1994, but it was not until the launch of the iPhone in 2007 and the subsequent introduction of app stores in 2008 that mobile apps became a major industry [7].

The mobile app market is primarily dominated by two platforms: Android, developed by Google, and iOS, developed by Apple. In February 2025 71.7% of smartphone users uses Android, and 27.8% uses iPhone, leaving less than one percent to other operating systems such as KaiOS and Windows [1]. Each platform provide different Software Development Kits (SDKs) and programming languages to support app creation.

2.1.1 Native Development

Native mobile development refers to building applications specifically for one platform, most commonly Android or iOS. Developing natively enables full use of the device's hardware capabilities, and is known for its high performance and a smooth user experience. However, writing several versions

of the same code increases development time, cost, and team size. It can also lead to inconsistencies, as business logic must be implemented separately for each platform [6].

Swift

Swift is Apple's modern programming language for developing applications across iOS, macOS, watchOS, and tvOS. It is open source and was released in 2014. Before Swift, the standard language for Apple platforms was Objective-C, which had been in use since the 1980s. Swift was introduced as a safer and faster alternative, but is fully interoperable with Objective-C, so it can be intergrated in existing projects. Developers primarily use Xcode, Apple's official Integrated Development Environment (IDE) [8].

Kotlin

Kotlin is a modern open source programming language developed by JetBrains and officially released in 2011. It runs on the JVM and is interoperable with Java. Kotlin was designed to be more concise, expressive, and safer than Java. Most Android development is done using Android Studio, the official IDE, which has full support for Kotlin [9].

2.1.2 Cross-platform Development

Cross-platform development refers to creating apps that run on multiple operating systems. Popular cross-platform frameworks include Flutter, React Native, KMP and Ionic. Cross-platform development can reduce development time and costs, making it an appealing option for startups or teams with limited resources. However, performance and user experience is generally considered to be worse compared to fully native apps, and accessing hardware through platform-specific APIs is often more complicated. Cross-platform development can be categorized into four main approaches: web, hybrid, interpreted, and generated applications [6].

Web Applications

Web applications are browser-based applications developed with common web technologies like HTML, CSS and JavaScript. They are accessed through a browser, and require no installation. However, they suffer from limited hardware access, rely on a constant internet connection, and typically offer lower performance compared to native apps [6].

Hybrid Applications

Hybrid applications combine elements of web and native applications. They wrap a web application inside a native container, allowing hardware access through APIs. Hybrid apps are built using web technologies like HTML and JavaScript but unlike web apps, they must be downloaded to the device [6]. Examples of frameworks using the hybrid approach is Ionic [10] and Apache Cordova [11].

Interpreted Applications

Interpreted applications execute code through an interpreter at runtime, rather than being fully compiled into native machine code beforehand. It often allows for fast development. However, because the logic is interpreted rather than compiled, there is a runtime performance overhead [6]. React Native [12] is a famous example of an interpreted cross-platform framework.

Generated Applications

Generated applications, commonly referred to as cross-compiled apps, take shared code and compile it into fully native binaries for each target platform. Tools using this approach translate a single codebase into platform-specific executables, which gives the possibility of near-native performance and access to device hardware through native APIs [6]. Kotlin Multiplatform [2] and Flutter [13] are examples of generated frameworks.

2.2 Kotlin Multiplatform

KMP is a cross-platform framework developed by JetBrains that uses the generated approach [2].

2.2.1 KMP Project Structure

In KMP each project supports a number of targets which specify the platforms that the code can be compiled for. Some examples are JVM, JavaScript, Android, iOS, or Linux [14].

KMP organizes code into source sets, which determine how code is shared across different targets [14]. There are three types:

- **Common:** Contains shared code that runs on all targets. As much code as possible is placed here.

- **Platform-specific:** Contains code specific to a single target. Native UI components or calls to native APIs are typically placed here.
- **Intermediate source sets:** Contains code shared between a subset of targets but not all. For example, an intermediate iOS source set can contain code used by all iOS targets, including `iosX64`, `iosArm64`, and `iosSimulatorArm64`. Which means that, for example, a native API call does not have to be written separately for each type of iOS target [14].

Figure 2.1 illustrates the division into source sets. In this example, the common source set (purple) is shared across Android and iOS, while the intermediate iOS source set (blue) includes all iOS targets. The platform-specific source sets (orange) contain code exclusive to individual platforms. The division into source sets is defined by the developer in the `buid.gradle.kts` file.

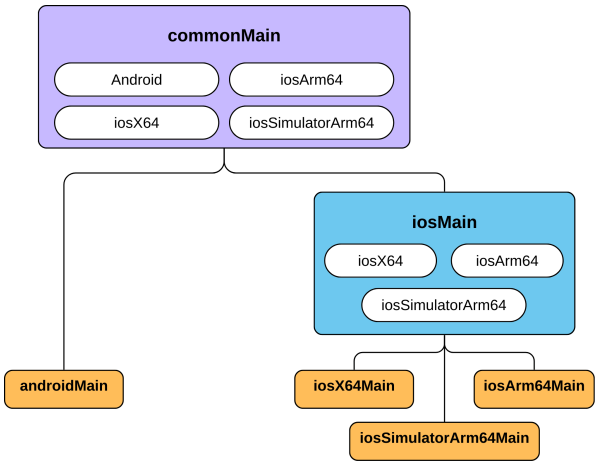

Figure 2.1: Example of division into source sets. Figure inspired by [14].

When compiling code for a specific target, the common source set, along with any matching platform-specific or intermediate source set, is used as input. The result is platform-specific binaries generated for the target platform [14].

2.2.2 Expect/Actual Mechanism

KMP uses an expect/actual mechanism to enable platform-specific implementations while maintaining shared code. In the common source set,

developers define an expect function, class, or interface without providing an implementation. The corresponding actual implementation is then defined in platform-specific or intermediate source sets, tailored to each target platform. During compilation, Kotlin automatically pairs each expect declaration with its matching actual implementation, ensuring that platform-specific functionality integrates correctly with shared code [15].

2.2.3 Kotlin/Native

Kotlin/Native is the compiler that KMP uses to generate native binaries from the Kotlin code, enabling execution on platforms that do not support a Virtual Machine (VM), such as iOS [16].

When a KMP application runs on iOS, its shared Kotlin code is compiled with Kotlin/Native, producing a precompiled binary that can interoperate with Swift and Objective-C. This allows Kotlin code to be integrated into existing iOS projects, giving developers the flexibility to decide which parts should be shared and which should be implemented natively [16].

Kotlin/Native also provides direct access to many Swift and Objective-C frameworks within the Kotlin code, without extra setup [16]. For instance, in the iOS source set of the KMP app, the AVFoundation framework [17], which is used to capture and process audiovisual media on iOS, can be imported through the line `import platform.AVFoundation.*`. This enables access to the device's Camera using the same API as a native Swift app.

2.2.4 Memory Management

Kotlin/Native and Swift uses different memory management systems. When creating a KMP app for an iOS target, these two memory management systems cooperate [18].

Kotlin/Native employs a tracing mark-and-sweep Garbage Collection (GC), similar to the JVM. It manages memory by deallocating objects that are no longer accessible from the program's roots. The GC starts when memory usage is high or after a certain time interval. It begins with a *mark* phase that is stop-the-world, which pauses all threads while marking all reachable objects. Then the *sweep* phase runs concurrently to remove unreachable objects [19].

On the other hand, Swift and Objective-C, manage memory using Automatic Reference Counting (ARC). ARC automatically tracks the number of active references to each object and frees the memory when no references remain. Every time a variable holds a reference to an object, ARC increases its

reference count. When all references are removed, the object is deallocated.

A disadvantage of ARC is its inability to handle cyclic references, where two or more objects hold strong references to each other. These cycles prevent proper deallocation and can lead to memory leaks [20]. Kotlin/Native's GC, on the other hand, can handle such cycles. A disadvantage of Kotlin/Native's GC is that its mark phase temporarily pauses application threads, which may introduce performance overhead or a visible lag [19].

When running a KMP app on iOS, both Kotlin/Natives GC and Swift's ARC system operate side by side. Each system manages its own objects, but when a Swift object is referenced in Kotlin, ARC tracks it on the Swift side while its lifetime is ultimately controlled by Kotlin's GC. This can delay deallocation until the next GC cycle, leading to temporary increases in memory usage compared to a native Swift application [18].

2.2.5 Concurrency and Scheduling

Modern mobile devices have multiple cores, allowing several threads to execute code in parallel. Mobile applications always run a main UI thread responsible for rendering the UI and handling interactions. In simple apps, all logic may run on the main thread, but when asynchronous operations are introduced, such as fetching data from an API or writing to local storage, these are typically handled on background threads to keep the UI responsive. Applications that perform heavier computations should also distribute work across multiple threads to avoid blocking the main thread.

Kotlin and Swift have some differences in how concurrency and scheduling work. In Kotlin, `coroutines` serve as the main unit of concurrent execution. Coroutines in Kotlin are launched within a scope and scheduled by a dispatcher, which determines the thread used for execution [21]. Dispatchers manage their own task queues and thread pools, though some may share threads. When launched, a coroutine is added to the dispatcher's queue and executed by an available thread [22]. The implementation details of coroutine scheduling has not been found in official documentation, but according to community discussions Kotlin's default dispatcher previously used Java's `ForkJoinPool` for coroutine scheduling on JVM targets, but now uses custom dispatchers on both JVM and Native targets. This scheduler contains key concepts from `ForkJoinPool`, like thread-local queues and work-stealing [23].

In Swift, the most common unit of concurrent work is the `Task`. Swift organizes tasks using dispatch queues, part of the Grand Central Dispatch

(GCD) framework. A dispatch queue is a First in, First Out queue to which tasks can be submitted for execution [24]. This is conceptually similar to Kotlin's dispatchers, however a key difference is how threads are managed. In Kotlin, each dispatcher has its own task queue and thread pool (though some may share underlying threads), while Swift's GCD uses a centralized, system-managed thread pool to execute tasks from all dispatch queues [25]. This means that threads in Swift are less tightly coupled to individual queues than in Kotlin. This reduces the developers control, but could have a positive impact on performance. Since thread management in Swift happens at the system level rather than the application level, the app may use less memory. It could also lead to lower CPU usage, as the system scheduler can balance the needs of all running applications more effectively [25]. No documentation describing in detail how scheduling is done within GCD has been found.

2.2.6 Runtime Integration

KMP and Swift differ in how they interact with the iOS runtime, particularly in terms of system integration and required infrastructure. Swift is native to iOS and tightly integrated with system-level frameworks, allowing it to rely on built-in functionality for concurrency and memory management. KMP, by contrast, targets iOS through Kotlin/Native and must include additional runtime components within the app binary. For instance, Kotlin's coroutine framework is implemented as part of the Kotlin standard library which must be bundled within the app binary [21], unlike the Swift app where GCD operates at the system level [25]. Similarly, Kotlin/Native includes its own GC for memory management, whereas Swift relies on ARC, which is provided by the iOS runtime and does not require bundling with the app [20]. These differences can increase binary size and memory usage in KMP apps.

In addition, bridging between Kotlin and Swift may require certain data types to be wrapped or converted to ensure compatibility across language boundaries, potentially introducing further overhead in cross-platform scenarios.

2.3 Compose Multiplatform

Compose Multiplatform (CMP) is a UI-framework that allows sharing user interface code across multiple platforms using Kotlin. By combining CMP with KMP, both business logic and UI can be shared across targets, creating a complete cross-platform solution.

CMP extends Jetpack Compose, originally developed for Android, to also support desktop, iOS, and web platforms. Currently, iOS support is in Beta, while web support remains experimental [3].

When rendering a Compose UI within a SwiftUI application, a bridge allows Compose-based views to be displayed alongside native SwiftUI components. The SwiftUI lifecycle is preserved, and both Compose and SwiftUI elements can be rendered simultaneously within the same interface [26].

2.3.1 Graphics and Animations

A Compose UI is built using composable functions, which are declarative building blocks that describe the UI structure and behavior. Jetpack Compose updates the screen in three main phases. The *composition phase*, which runs the composable functions and builds a composition tree. A composition tree is a data structure that represents the current UI state. The second phase is the *layout phase* which uses the composition tree to measure each element and determine its size and position in the 2D space. Finally, it reaches the *drawing phase* in which the pixels are drawn on screen [27].

Rather than rendering pixels directly, Compose generates drawing commands that are passed to the graphics engine Skia. Skia is a 2D graphics library written in C++, developed by Google, and used in, for example, Google Chrome, Flutter, and Jetpack Compose. It provides low-level APIs for drawing shapes, text, images, and paths, enabling rendering and visual consistency across multiple platforms [28].

When running a CMP application on iOS, the rendering relies on Skiko (Skia for Kotlin), which acts as a bridge between Kotlin code and the native Skia library [29]. The drawing commands are sent from Compose, via Skiko, to Skia as operations on a `SkCanvas`. Skia then rasterizes these commands into pixels using its Graphics Processing Unit (GPU) backend [28]. Finally, the rendered output is embedded and displayed inside a native `UIViewController` on iOS.

Native SwiftUI applications use Apple's Core Animation framework to handle animations. Core Animation is a framework that enables smooth transitions by animating changes to visual properties such as position, size, and opacity. Rather than redrawing views for each frame, Core Animation caches the view's content into bitmaps. When a property changes, it updates the layer's state information and hands both the cached bitmap and the new state to the GPU, which renders the new frame. This means that animations

are created by manipulating already rendered content in hardware, rather than having the CPU re-render each frame manually [30].

This drawing model contrasts with the view rendering of Jetpack Compose, where changes typically require redraws on the main thread, which is CPU-intensive. By shifting this work to the GPU, Core Animation achieves high performance for simple animations like fades, moves, and resizes [30].

A central challenge with animations on mobile devices is that mobile devices are battery-powered and have limited CPU and GPU resources, and rendering graphics is among the most power-intensive operations. Therefore, it is important that animations are rendered efficiently [31].

2.4 Related Work

The related work is organized into three sections. The first reviews general approaches to evaluating and comparing the performance of mobile development frameworks. The second summarizes empirical studies on cross-platform frameworks other than KMP. The final section focuses specifically on studies related to KMP.

2.4.1 Performance Evaluation of Mobile Frameworks

Ahti et al. [32] attempt to find an evaluation framework for assessing cross-platform mobile development tools. The study emphasizes that user experience should be in focus rather than developer-centric metrics. They propose a framework that combines quantitative measures such as app start time, memory usage, and CPU usage, with qualitative aspects and perceived user experience.

Karami et al. [33] conducted a systematic review of 75 studies on cross-platform mobile development. The findings show that most studies evaluate simple prototype apps rather than real-world applications, and focus on the frameworks Cordova, Titanium, React Native, Xamarin, Ionic, or Flutter. Common evaluation criteria include CPU usage, memory consumption, and UI quality, assessed either through metrics or user studies.

Biørn-Hansen et al. [34] present a survey of cross-platform development research, similarly concluding that the most commonly used performance metrics are CPU usage, memory consumption, and energy usage. The study also highlights a lack of empirical evaluation of newer frameworks and emphasizes the importance of including a native implementation as a baseline for comparison.

These three studies helped shape the scope of this thesis. A newer and less studied framework, KMP, is selected as the target for evaluation. The study applies commonly used quantitative performance metrics while maintaining a focus on user experience. In line with prior recommendations, a native app is implemented to serve as a baseline.

2.4.2 Cross-platform Research

This section contains a summary of the previous research on cross-platform mobile development, with other frameworks than KMP. In Table 2.1 previous research targeting camera access, local file operations or UI operations, such as animations or list scrolling is listed.

Table 2.1: Overview of relevant research on cross-platform development

Ref	Year	Platform	Framework	Metrics
[35]	2017	Android, iOS	Xamarin	Execution time of reading and writing files to local storage.
[36]	2018	Android, iOS	Apache Cordova, Xamarin, Appcelerator Titanium	UI rendering time and UI response time.
[37]	2019	Android, iOS	React Native, Ionic, Xamarin	FPS, CPU and memory consumption of several animations including moving images and navigation.
[38]	2019	Android	React Native, Ionic/Cordova	CPU and memory consumption when scrolling through a list.
[39]	2020	Android	React Native, Ionic, Flutter, MD2, NativeScript	Execution time, CPU, and Random Access Memory (RAM) when writing to local file system.

Ref	Year	Platform	Framework	Metrics
[40]	2020	Android, iOS	Flutter	CPU during list scrolling and navigation transitions. Conducted a user study evaluating the look and feel.
[41]	2021	Android, iOS	Flutter, React Native	Execution time, CPU, RAM, and FPS during list scrolling, capturing photo and video, and displaying animations.
[42]	2021	Android	Java, Flutter, Kotlin/Native	CPU, RAM, and execution time for reading, writing, and deleting files in local file system.
[43]	2022	Android	Flutter	CPU and memory usage of an image animation and an infinite scroll list.
[44]	2023	Android	React Native, Flutter	CPU, memory usage, and number of janky frames when scrolling through infinite list and capturing a photo and saving it to device gallery.
[45]	2023	Android, iOS	Flutter	CPU and RAM of two types of animations.
[46]	2023	Android	Flutter, React Native, Ionic	CPU, memory usage and energy consumption of a rotating images animation.

Camera

Camera functionality is widely used in modern mobile apps, such as Instagram and Snapchat for capturing media, or banking apps for scanning documents. Previous studies have targeted camera performance on React Native and Flutter. One study compared React Native and Flutter on Android by measuring camera performance over a fixed time period, finding that Flutter captured more photos but also used more CPU [44]. Another study compared

React Native and Flutter for image capture and found that Flutter outperformed React Native on both iOS and Android. However, no native baseline was included for comparison [41]. No existing study has evaluated camera performance in KMP, highlighting the need for further investigation.

File operations

Local file operations, such as saving and accessing media or configuration files, are commonly used in mobile apps, for example Spotify for storing offline content. Several studies have evaluated file performance in cross-platform frameworks, focusing on execution time, CPU, and memory usage. One study compared Xamarin with native iOS and Android, finding that file read and write operations were slower on Xamarin, particularly on iOS. These files contained 10 000 decimals of pi [35]. Another study compared React Native, Ionic, Flutter, MD2 and NativeScript by measuring the time and resource usage required to load an image from internal storage. Flutter was slowest but used least RAM, and React Native and Ionic used most CPU [39]. A third study evaluated read, write, and delete performance using 100 MB files in Java, Flutter, and Kotlin/Native. Flutter was fastest, but used the most memory [42].

List scroll

List scrolling is a common interaction in mobile apps, used in social media feeds, messaging apps, and shopping platforms. Previous studies have examined scrolling performance in cross-platform frameworks with different approaches. Some studies scrolled through a list containing only text data [38, 43, 40], while another included images in the list [44]. Most studies automated the scrolling in code, but others study scrolled manually up and down, which may introduce variability between measurements [40, 47].

A key finding throughout these studies is that Flutter often used the same amount or less CPU than the native implementations [40, 44]. However one study counters this with the opposite result that Flutter used more CPU and less memory than native Android [43]. For React Native on the other hand, the CPU usage was high [38], while memory usage was comparable to the native counterparts on both Android and iOS [38, 44].

Other animations

Evaluating animation performance has been approached with different methods in previous research. One study evaluated a re-scaling animation

where a star icon continuously changed size. It compared native Android with Flutter and found that Flutter consumed significantly more CPU and RAM [43].

Another study examined a LottieFiles animation that included movements of multiple elements and changes in their visibility by fading elements in and out. It found that all cross-platform frameworks on Android introduced higher memory usage and a greater number of janky frames compared to native implementations. On iOS, Xamarin and React Native rendered the most frames, but React Native consumed more CPU, while Xamarin used more memory [37].

A third study evaluated both a LottieFiles animation and a custom animation featuring scaling, movement, fading, and color changes. It found that Flutter had higher CPU and memory usage than both native Android and native iOS implementations, regardless of the animation type [45].

A fourth study focused on a rotating image animation on Android, where 28, 112, 252, or 448 images were displayed and rotated nine times over 18 seconds. It found that React Native consumed the least energy, using an option that offloads animations to native code. However, it also had the highest memory usage. Flutter used less CPU and memory than React Native with fewer images but performed worse as the number of images increased. Ionic showed the worst performance overall [46].

No research was found evaluating animations in KMP or CMP, highlighting a gap in current research.

2.4.3 Kotlin Multiplatform Research

In Table 2.2 there is an overview of previous research on KMP, CMP, or Kotlin/Native, which is the compiler used in KMP.

One study from 2019 developed a KMP app and measured app startup time, app size, and lines of code. It found that the KMP app was 18% larger than the native iOS version, while startup time was not significantly different. The KMP implementation also had significantly fewer lines of code, indicating a high degree of code reuse [4].

Another study also evaluated startup time and app size, but used CMP for the user interface. It confirmed that the KMP app size was significantly larger. However, in contrast to the previous findings, it reported that the KMP/CMP app had a slower startup time than native iOS [47]. This study also measured scroll performance but relied on manual interactions instead of automated scrolling. Moreover, it faced challenges in interpreting results due

Table 2.2: Overview of research on Kotlin Multiplatform

Ref	Year	Platform	Framework	Metrics
[4]	2019	Android, iOS	KMP	App startup time, app size, and lines of code.
[42]	2021	Android	Java, Flutter, Kotlin/Native	CPU, memory, and execution time for reading, writing, and deleting files in local file system.
[5]	2023	iOS	KMP	Networking, serialization, local database operations, and four Computer Language Benchmarks Game (CLBG) benchmarks.
[47]	2024	Android, iOS	KMP/CMP	Startup latency, app size, and scroll performance.
[48]	2024	Android, iOS	KMP, Ionic/-Capacitor	Rating based on long-term feasibility, development environment, maintainability, native code integration, access to device hardware, platform-specific functionality, and security.
This study	2025	iOS	KMP	Camera access, local file operations, scroll performance, and animations.

to limited documentation [47]. Because of these limitations, the results may be unreliable, and this study will repeat the scroll performance benchmark in my study using an automated and repeatable setup.

A more comprehensive study compared KMP and native Swift across several operations, including networking, serialization, local database access, and four Computer Language Benchmark Game (CLBG) tasks. The results portrayed KMP positively, showing that it generally achieved faster execution times, though at the cost of increased CPU and memory usage.

Similar to my study, one earlier paper benchmarked Kotlin/Native for local

file read/write performance [42]. However, that study used Kotlin/Native as a standalone compiler, not as part of KMP, and it only evaluated Android, leaving iOS unstudied.

2.5 Summary

Previous studies on KMP for iOS have primarily focused on high-level metrics such as startup time, app size, and code reuse. These studies have shown that KMP applications are generally larger than their native counterparts, while startup performance has varied across results [4, 47]. Another evaluated networking and database access, showing that KMP can outperform native Swift in speed, but with increased CPU and memory usage [5].

While hardware features such as camera access and file system usage have been evaluated in other cross-platform frameworks like Flutter and React Native, no prior study has systematically benchmarked these operations on iOS using KMP. Similarly, although UI performance, especially scrolling and animations, has been studied in other frameworks, there is a lack of reliable data on how CMP performs in these areas, particularly when it comes to automated, repeatable testing methods.

This thesis aims to address these gaps by evaluating both hardware and UI related performance in KMP and CMP compared to native Swift on iOS. Specifically, the benchmarks target camera usage, local file operations, scroll performance, and animations.

Chapter 3

Method

This chapter describes the research methodology used in this thesis, including the benchmark design, benchmarking process, data collection strategy, analysis techniques, and the test environment.

3.1 Research Process

This thesis investigates the performance of KMP and CMP compared to native Swift on iOS. The research process began with a literature review, focusing on KMP as well as prior research on other cross-platform frameworks. Based on this review, a research gap was identified, leading to the formulation of the following two research questions.

- RQ1: Does Kotlin Multiplatform perform comparably to native Swift on iOS when performing hardware-related operations?
- RQ2: Does Kotlin Multiplatform and Compose Multiplatform perform comparably to native Swift on iOS in terms of UI-rendering?

To answer the research questions, six benchmarks were designed. Three targeting hardware-related operations and three focusing on UI performance. Descriptions of these benchmarks are provided in Sections 3.3.1 and 3.3.2.

Before implementation, an in-depth review of the Kotlin and Swift programming languages, including their memory management models, was conducted to ensure that best practices were followed during development in KMP, CMP, and native Swift. Two applications were developed, each containing implementations of the six benchmarks: one using KMP/CMP and

the other using Swift. A detailed description of the benchmarking applications is provided in Chapter 4.

Both apps were deployed and executed in a controlled environment on the same iPhone device. For the hardware-related benchmarks, CPU usage, memory usage, and execution time were measured. For the UI-related benchmarks, CPU usage, FPS, and delayed frames was measured. Once the data was collected, it was analyzed as described in Section 3.4.

3.2 Research Paradigm

This thesis follows a positivist research paradigm, which assumes that objective knowledge can be gained through observation, measurement, and empirical analysis. The study adopts a quantitative and experimental approach, focusing on measurable performance metrics to evaluate and compare two software implementations: KMP/CMP and native Swift.

In line with this paradigm, benchmarking is used as the primary method. Benchmarking allows system performance to be evaluated by executing representative tasks in a controlled and repeatable environment. By isolating specific operations and measuring how each framework performs under identical conditions, the study aims to draw objective conclusions based on the collected data.

3.3 Benchmarking

A benchmark is a program or task designed to simulate real application behavior under controlled conditions. It is used to evaluate and compare system performance. Since testing real applications can be impractical or costly, benchmarking can simplify the process by serving as approximations of real applications [49].

There are several types of benchmark programs, each with different purposes. *Microbenchmarks* focus on a small and specific part of a system, and is useful for understanding the performance of specific operations such as API-calls. *Synthetic benchmarks* aim to emulate the instructions and resource usage patterns of real applications, without performing any meaningful computations. This is useful for measuring overall system performance or stress-testing, but not for benchmarking a specific operation. *Application-based benchmarks*, on the other hand, evaluate performance using complete, real-world apps and provide insights into how a system handles practical use

cases [49]. In this study, a hybrid approach is adopted. Each benchmark is modular and focused on a specific operation, similar to a microbenchmark, but also designed to represent realistic mobile applications, which allows for both fine-grained control and results that are meaningful in real-world contexts.

3.3.1 Hardware Benchmarks

To answer RQ1, three hardware related benchmarks are designed, targeting the camera and local file system of the device. These benchmarks were chosen because they are commonly used in modern mobile applications, making it important to understand their impact on app performance. Additionally, the performance of these features has been studied using other cross-platform frameworks such as Xamarin, React Native, Ionic, and Flutter, but has not yet been thoroughly examined using KMP. The details about the benchmarks can be seen in Table 3.1.

Table 3.1: Hardware-related benchmarks

Benchmark	**Description**	**Metrics**
Camera	The device camera is used to capture an image and save it to device gallery.	Execution time, CPU, memory usage
File Write	Writes a 100 MB binary file to the application's internal storage.	Execution time, CPU, memory usage
File Read	Reads a 100 MB binary file from the application's internal storage.	Execution time, CPU, memory usage

All hardware benchmarks measure execution time, CPU, and memory usage. These metrics were selected because they are commonly used in similar research, making the results comparable to previous studies. Moreover, these metrics have an impact on user experience and are closely tied to energy consumption, which is particularly important in mobile applications where resource efficiency is critical.

3.3.2 UI Benchmarks

RQ2 is investigated through three UI-related benchmarks focusing on different types of animations. Scrolling through a list, animating an image by fading it in and out of visibility, and performing a heavier animation involving multiple images. The scroll benchmark was chosen because it is commonly used in related research on cross-platform frameworks. The visibility animation was proposed in discussion with the supervisor at Framna, as visibility changes are among the most common types of animations in mobile applications. The multiple image animation was designed to stress-test the framework's handling of more complex and demanding animations. The benchmarks are summarized in Table 3.2.

Table 3.2: UI-related benchmarks

Benchmark	**Description**	**Metrics**
Visibility animation	A static 1 MB image is animated by fading in and out of visibility for 30 seconds.	CPU, FPS, Delayed frames/s
List scroll	A list is automatically scrolled for 30 seconds. Each list item consists of a 1 MB image and a text displaying an increasing number. The image is randomly selected from 10 available image assets.	CPU, FPS, Delayed frames/s
Multiple image animation	200 small 2 KB images are continuously re-scaled and faded in and out of visibility over 30 seconds.	CPU, FPS, Delayed frames/s

These benchmarks measure CPU usage, FPS, and delayed frames. These metrics were chosen because they are widely used in existing UI performance research and provide insight into both rendering smoothness and system resource usage. FPS and delayed directly impact perceived responsiveness and user experience, while CPU usage can signal inefficient rendering or animation logic.

3.3.3 Data Collection

The data collection process is illustrated in Figure 3.1. It shows three distinct branches, depending on whether the benchmark targets hardware or UI performance.

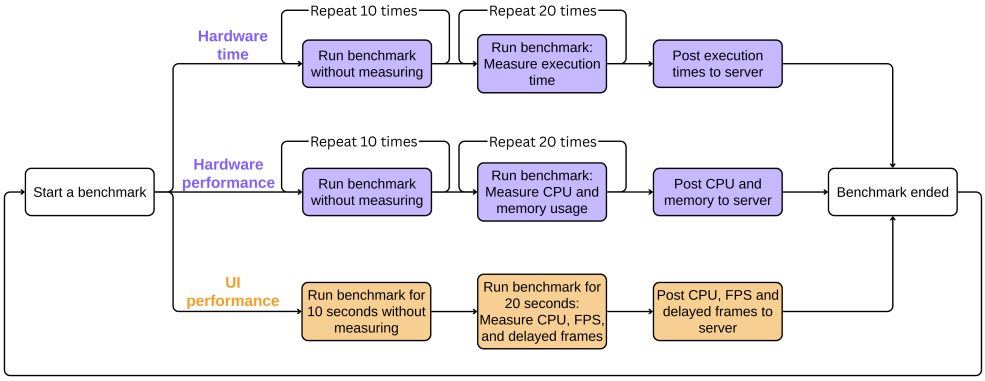

Figure 3.1: Data collection flow of benchmark apps

For hardware benchmarks, execution time and resource usage are measured in separate runs to avoid interference in the execution time measurements from sampling overhead. Each hardware benchmark begins with a warm-up phase, where the task is executed 10 times without measurement. To measure execution time, the task runs once per iteration for 20 iterations, and the duration of each run is recorded and sent to a local Python server. For CPU and memory usage, metrics are sampled continuously at each new frame, around 60 times per second, and the collected data is also posted to the server.

UI benchmarks has a 10 seconds warmup phase followed by 20 seconds of measurements. CPU usage is sampled at each rendered frame and using the timestamps of each frame, the number of frames rendered within the previous second (FPS) is calculated, along with an indication of whether the frame was delayed. The results are then posted to the server.

Each benchmark was run in 5 separate runs. Hardware benchmarks were measured a total of 100 times, and UI benchmarks were measured over a total of 100 seconds, both excluding the warm-up phase.

The measurements were divided into five separate runs, rather than conducting all 100 iterations consecutively, to reduce the risk of anomalies caused by factors such as device overheating or temporary memory buildup. Separating the runs allowed the device time to cool down and reset system resources between tests, contributing to more stable and reliable performance

data.

No personal or sensitive data was collected. All measurements involved system-level metrics on a test device. Therefore, the data collection poses no ethical concerns.

3.3.4 Benchmark Validity

Each benchmark is designed to simulate a real-world task. The code has been reviewed and tested to ensure that it performs the intended operation. For the UI benchmarks, correctness is easily verified through on-screen visual feedback. For hardware-related benchmarks, manual verification is required.

In the camera benchmark, the correct number of photos is verified by checking that they have been added to the device's photo gallery. In the file write benchmark, after pausing performance measurements, the existence of the written file is programmatically confirmed before it is deleted. The file read benchmark automatically prints an error if the file to be read does not exist. If any anomalies are detected during execution, the entire benchmark run is discarded. Furthermore, hardware operations are isolated from UI rendering to ensure that only the intended operation is being measured.

3.3.5 Benchmark Reliability

Performance metrics are collected in a controlled environment using a single physical device. The device is fully charged and remains connected to power during benchmarking. To minimize external interference, flight mode is enabled, screen brightness is set to maximum, and all other applications are closed to reduce background activity. The benchmark app is restarted between each benchmark run, and a few seconds are waited after launching the app before starting a benchmark to allow the system to stabilize. All benchmarks include a number of warm-up iterations before real measurements begin. This approach allows the system to reach a stable state and helps exclude potential outliers caused by initialization overhead.

All user interactions, such as scrolling and taking photos, are fully automated. This eliminates the risk of human error affecting timing, accuracy, or consistency between runs.

3.3.6 Test Environment

The software versions used for both implementations are listed in Table 3.3 and Table 3.4 lists the specifications of the test phone used to run both apps. The

Table 3.3: Software specifications

Software	**Version**
Andriod Studio	2024.2.2 (Ladybug)
XCode	15.4 (15F31d)
Kotlin	2.1.0
Swift	5.10
Kotlin Multiplatform plugin	2.1.0
Compose Multiplatform plugin	1.8.0

Table 3.4: Test Device specifications

Model	iPhone 13 mini
Model number	MNFF3QN/A
iOS-version	18.3.2
Processor	A15 Bionic
GPU	4-core Apple GPU
RAM	4 GB
Storage	128 GB

apps were run as *release builds* immediately from XCode's *Run* operation. The Release build configuration ensures that the code is executed under the same optimization and runtime conditions as it would be in production, eliminating debug-only behaviors and makes sure the benchmarking reflect realistic application behavior [50].

3.4 Data Analysis

The method used for analyzing performance differences is the *Welch's two-sample t-test*. It is used to compare the means of two independent groups when their sample sizes or variances may differ. It is appropriate in this context, as the two implementations are tested independently, and the sampling distribution of the means can be assumed to be normal due to the Central Limit Theorem. The null hypothesis H_0 states that there is no significant difference

in performance between the two implementations:

$$H_0 : \mu_{\text{KMP}} = \mu_{\text{Swift}}$$

The t-test compares the sample means using the following formula:

$$t = \frac{\bar{x}_{\text{KMP}} - \bar{x}_{\text{Swift}}}{\sqrt{\frac{s^2_{\text{KMP}}}{n_{\text{KMP}}} + \frac{s^2_{\text{Swift}}}{n_{\text{Swift}}}}}$$

where $\bar{x}$ represents the mean, s^2 the variance, and n the number of observations for each implementation. The result is a t-statistic, which is used to compute a p-value [51]. If the p-value is below the significance threshold $\alpha = 0.05$, the null hypothesis is rejected and the difference in means is statistically significant.

The software that will be used for data analysis is Python. The `numpy` library is used for processing and organizing the collected benchmark data, `scipy` is used to perform t-tests, and `matplotlib` is used to visualize the results.

Chapter 4

Implementation

This chapter provides details about the two benchmarking applications developed. Section 4.1 presents an overview of both applications, while Section 4.2 details the measurement sampling process. Finally, Section 4.3 offers an in-depth explanation of each benchmark. All source code for both applications and measurement logic can be found in Appendix A.

4.1 Benchmarking Apps

4.1.1 Kotlin Multiplatform App

The app's main screen, displays buttons to launch each benchmark (Figure 4.1). A bottom navigation bar allows switching between hardware benchmarks, UI benchmarks, and a view with additional actions. These *other* actions are:

- *Request all necessary permissions:* Requests camera, photo library, and local network access. It is separated to prevent permission prompts from interfering with benchmark measurements.
- *Write files for Read benchmark:* Writes the 30 files used in the read benchmark. This is done in advance to prevent the files from being cached by a prior write operation.
- *Sample idle state memory:* Measures memory usage over 30 seconds while the app is idle. Results are posted to the server.

Pressing a benchmark button invokes the `BenchmarkRunner`, which handles the setup for performance measurements. In KMP, this runner

Figure 4.1: Screenshots from KMP/CMP app

is defined as a Kotlin interface with its implementation written in Swift. The runner logic is implemented in Swift to share the same measurement infrastructure, across both implementations. This ensures consistent and reliable results. For details on the measurement setup, see Section 4.2.

Each benchmark is implemented as a class in its own package within the shared source set (`commonMain`), exposing a `runBenchmark` method. Listing 4.1 shows an example of how the scroll benchmark is invoked.

Listing 4.1: Example invocation of a benchmark, in Swift

```
let scrollBenchmark = ScrollBenchmark()
try await scrollBenchmark.runBenchmark(n: duration)
```

For UI benchmarks, all logic resides in shared code. For hardware benchmarks, KMP's `expect/actual` mechanism is used to handle platform-specific implementations, such as calls to native APIs.

4.1.2 Native Swift App

The native app follows the same structure as the KMP application. The home screen contains buttons to launch each benchmark (Figure 4.2), as well as the *other* actions. When a button is pressed, the `BenchmarkRunner` is invoked which in turn invokes the respective benchmarks. As in the KMP application, the `BenchmarkRunner` handles measurement setup, and each benchmark is implemented as a class with an exposed `runBenchmark` method.

Figure 4.2: Screenshots from native Swift app

4.2 Measurements

Performance data is collected in Swift and is based on the open-source library *GDPerformanceView-Swift* [52], a library used for measurements in other similar work [5]. Originally, *GDPerformanceView-Swift* is designed to display performance metrics (CPU, memory, and FPS) as an overlay on top of the iOS status bar during app runtime. In this project, the code have been modified to log all measurements to a remote server instead. Additionally, a function for detecting delayed frames was added. The core mechanisms for sampling CPU usage, memory usage, and FPS remain unchanged from the original library. The upcoming sections contains more detailed explanations of each metric.

4.2.1 Execution Time

Execution time refers to the duration required to complete a single iteration of a benchmark. All measurements are taken using the `systemUptime` property of the `ProcessInfo` class, which returns the time the system has been awake since the last restart. To measure execution time, `systemUptime` is sampled immediately before and after a benchmark iteration. The difference between these timestamps represents the duration of the iteration [53]. Listing 4.2 shows an example of this measurement in Swift.

Listing 4.2: Example of execution time measurement in Swift

```
for i in 0..<n {
    let start = ProcessInfo.processInfo.systemUptime
    // execute benchmark iteration i
    let duration = ProcessInfo.processInfo.systemUptime
       - start
    performanceCalculator.sampleTime(duration: duration)
}
```

The `sampleTime` method appends the duration to a shared list, `timeDurations`, using a barrier write on a concurrent queue, `timeDurationsQueue`. The barrier ensures only one write occurs at a time, preserving the order of samples and avoiding race conditions.

Once all iterations are complete, a synchronous read is used to access the list. This blocks until all writes are finished, ensuring consistency. The durations are then formatted and sent to the remote server, and the list is cleared for the next run.

In the KMP implementation, the `systemUptime` property cannot be accessed from shared code, so it is accessed via an `expect/actual` function, as shown in Listing 4.3.

Listing 4.3: Execution time measurement in Kotlin

```
import platform.Foundation.NSProcessInfo
actual fun getTime(): Double {
    return NSProcessInfo.processInfo.systemUptime
}
```

4.2.2 CPU Usage

CPU usage refers to the percentage of the device's total processing capacity consumed by the application during execution. It provides insight into how computationally intensive a benchmark is. To measure CPU usage, all active threads in the current process are iterated, and their usage is summed. Idle threads are excluded.

CPU usage is sampled continuously during the benchmarks using the `GDPerformanceView-Swift` library [52]. It utilizes `CADisplayLink` to trigger a callback on every screen refresh, ideally 60 times per second, see Listing 4.4.

Listing 4.4: Callback triggered at every screen refresh using CADisplayLink

```
@objc func displayLinkAction(displayLink: CADisplayLink)
    {
```

```
        self.linkedFramesList.append(frameWithTimestamp:
            displayLink.timestamp)
        self.takePerformanceEvidence(timestamp:
           displayLink.timestamp)
        previousFrameTimestamp = displayLink.timestamp
    }
```

At each refresh, a function called *takePerformanceEvidence* is called, which samples the metrics and adds them to a shared list (see Listing 4.5).

Listing 4.5: Sampling metrics during each display refresh

```
func takePerformanceEvidence(timestamp: TimeInterval) {
        let cpuUsage = self.cpuUsage()
        let memoryUsage = self.memoryUsage()
        let fps = self.linkedFramesList.count
        let delayed = self.delayedFrames(
           currentTimestamp: timestamp)
        let measurement = "\(cpuUsage)|\(fps)|\(delayed)
           |\(memoryUsage)|\(timestamp)"
        self.addMeasurement(measurement)
    }
```

The measurements are appended to the list via a barrier-protected concurrent dispatch queue. This approach ensures thread safety by allowing multiple concurrent reads but enforcing exclusive access for writes, preserving the order of measurements and preventing race conditions. This pattern is the same as in the execution time tracking logic. Once the benchmark completes, all measurements in the buffer are posted to a remote server, and the list is cleared for the next run.

4.2.3 Memory Usage

Memory usage is tracked via the `phys_footprint` field obtained from the process's virtual memory information using the `task_info` system call. This value reflects the actual physical memory footprint of the application at a given moment and is converted to megabytes before being recorded.

Memory is sampled continuously during the benchmark using the same mechanism as CPU usage. As shown in Listings 4.4 and 4.5, measurements are triggered at every screen refresh via `CADisplayLink`, approximately 60 times per second and the memory value is extracted inside the `takePerformanceEvidence` method. The memory usage is appended to the same barrier-protected list as the CPU measurements. After

the benchmark completes, the measurements are posted to a remote server and the buffer is cleared for future runs.

4.2.4 FPS

The FPS metric is used to estimate how smoothly the application renders content. It reflects how many frames were rendered during the last second.

To compute FPS, each frame's timestamp is appended to a linked list, which operates as a time-based sliding window. Every time a new frame is rendered, that is when `CADisplayLink` triggers `displayLinkAction` (see Listing 4.4), a new node with the current timestamp is added to the list. Immediately after insertion, any nodes older than one second compared to the latest timestamp are pruned by traversing from the head of the list and removing outdated entries.

At any point, the number of nodes remaining in the list corresponds to the number of frames rendered in the past second, that is the current FPS. As seen in Listing 4.5, the length of the list is recorded at each screen refresh.

4.2.5 Delayed Frames

Delayed frames represent instances where the application failed to render the next frame within the expected time budget. On a 60 Hz display, this budget is approximately 16.67 ms per frame. If rendering takes longer than this interval, one or more frames may be dropped or appear delayed to the user, leading to visible stutter or lag.

To estimate the number of delayed frames, the benchmark measures the time elapsed between two consecutive callbacks from `CADisplayLink`, which is designed to fire once per screen refresh. As seen in Listing 4.4, the timestamp of each callback is recorded. In `takePerformanceEvidence` (Listing 4.5), the time since the previous frame is compared to the ideal frame interval. For every 16.67 ms that the callback was delayed beyond the expected interval, one delayed frame is counted.

This method provides a conservative estimate of rendering hiccups, capturing both dropped and late frames. It complements the FPS measurement by revealing when frames are rendered inconsistently, even if the average frame rate remains high.

4.3 Benchmark Implementations

This section contains a more detailed and technical description of the implementation of each benchmark.

4.3.1 Camera

The camera benchmark measures the performance of capturing and saving photos using the device's camera. It runs entirely in the background with no UI interaction, to avoid introducing noise into the results.

Each benchmark run captures 30 photos in total, the first 10 photos serve as warmup and are excluded from the results, while the remaining 20 are used for measurement. All photos are captured using Apple's `AVFoundation` framework [17] via `AVCaptureSession` and `AVCapturePhotoOutput`. The captured images are saved directly to the photo library using the `PHPhotoLibrary` class [54].

To ensure consistent conditions, all photos were taken indoors under stable lighting. The default `.photo` session preset was used, and no changes were made to resolution or image quality settings.

Both the native Swift and the KMP implementations use the same camera setup and underlying API. The key difference is that the Swift version calls `AVFoundation` API directly, while the KMP version interacts with them via Kotlin/Native interoperability using `expect/actual` declarations.

4.3.2 File Write

The file write benchmark measures the performance of writing binary files to disk. Each run writes 30 files, each 100 MB in size. The first 10 files serve as a warmup phase and are excluded from the results. Like the camera benchmark, it runs entirely in the background without any UI interaction, minimizing measurement noise. Once the benchmark completes, all files are deleted to avoid storage buildup between runs.

The file content is generated once during the initialization of the benchmark and reused across all write operations. It consists of a `Data` object in Swift, or a `ByteArray` object in Kotlin, filled with zeroed bytes. This ensures consistent file size and content across iterations, while excluding data generation from the measured workload.

All file operations are handled using Apple's `Foundation` framework. Files are created using the `FileManager` class and written using

`FileHandle` [55]. The `write(contentsOf:)` method is used to synchronously write data to the file, followed by `synchronize()` to flush in-memory buffers and ensure the data is fully written to disk. This guarantees that the benchmark measures actual disk I/O rather than buffered writes.

In the KMP implementation, the same underlying Swift methods are used, but they are accessed through Kotlin/Native interop using `expect/actual` declarations. The Swift `write` method expects data in the form of an `NSData` object, which is a native Objective-C class used for handling binary data. Since `NSData` is not accessible from shared Kotlin code, the binary data is initially represented as a `ByteArray` in Kotlin and then converted to `NSData` before writing.

4.3.3 File Read

The file read benchmark measures the performance of reading previously written binary files from disk. Each run reads 30 files in random order, each 100 MB in size. Like the other hardware benchmarks, the first 10 files serve as a warmup, and the benchmark runs entirely in the background without any UI interaction. The files contain binary data with only zeroes. To prevent the operating system from serving cached data from memory, the files are written ahead of time by a separate method, and the app is restarted between each run of the read-benchmark.

Each read operation uses Apple's `Foundation` framework. Files are opened with `FileHandle` and read to the end using `readToEnd()`, which synchronously reads all data until the end of the file [55]. In the native Swift implementation, the data is returned as an`Data` object.

The KMP implementation calls the same underlying Swift APIs, using Kotlin/Native interop with `expect/actual` declarations. However, unlike Swift where the `NSData` result is dropped, the KMP version converts `NSData` into a Kotlin `ByteArray`.

4.3.4 Visibility Animation

The visibility animation displays a single 10MB full-screen image on a white background and continuously fades it in and out of visibility. In Figure 4.3 screenshots from the KMP/CMP implementation is showed. The Swift implementation looks visually identical.

The Swift visibility benchmark displays the image inside a full screen `ZStack`, with its opacity alternating between 0 and 1 using a smooth linear

Figure 4.3: Screenshot from visibility animation in KMP/CMP app

animation. The opacity change is driven by toggling a local state variable inside a repeating animation block. This animation is triggered externally on a fixed interval, and it continues as long as `VisibilityController` indicates the benchmark is active.

The Kotlin visibility benchmark diplays the image in a full-screen `Box` and loops its opacity linearly between 0 and 1. It uses Jetpack Compose's `rememberInfiniteTransition` to drive the animation declaratively. Like the Swift version, the animation runs continuously while `VisibilityController.isRunning` is true, and completion is detected via `LaunchedEffect`.

4.3.5 List Scrolling

The list scrolling benchmark measures the performance of automatic vertical scrolling through a list containing images and a text label with an increasing number. The images are randomly selected from a pool of 10 preloaded assets, each 1MB in size, and 600×400 pixels. Both versions use lazy rendering to optimize memory usage, loading only visible items and nearby content. In Figure 4.4, screenshots from the KMP/CMP implementation are shown. The Swift implementation is visually identical.

In the Swift implementation, the list is displayed using a `LazyVStack` inside a `ScrollView`, with 100 items defined statically. Image selection is

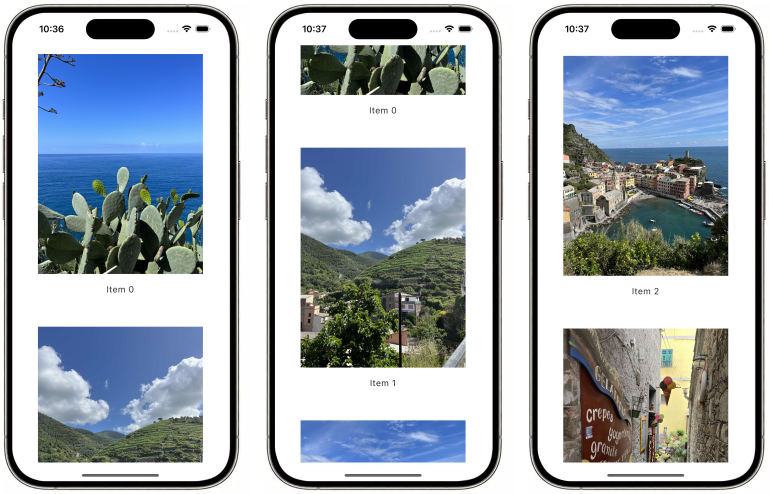

Figure 4.4: Screenshot from scroll animation in KMP/CMP app

randomized once per item on first appearance using SwiftUI state. Automatic scrolling is achieved using a repeating timer that calls `scrollTo()` on a `ScrollViewProxy` every 400 milliseconds with a linear animation. The animation cycles continuously while the benchmark is active.

The KMP implementation mirrors this logic using Jetpack Compose's `LazyColumn`, with 100 statically defined items. Each item randomly selects one of the 10 image resources using a Compose `remember` block. Scrolling is driven by a coroutine that continuously calls `animateScrollToItem()` on the list state.

4.3.6 Multiple Images Animation

The multiple images animation benchmark measures the system's ability to animate a large number of visual elements in parallel. The screen is filled with 200 small 2 KB star images (png), each independently animated with a pulsing scale and opacity effect. Figure 4.5 shows a screenshot from both the KMP and native Swift implementation, there the left is KMP and the right is Swift.

Each star is randomly positioned on the screen and chosen from a set of 10 different star images. To avoid all animations running perfectly in sync, a random start delay between 0 and 1 second is applied individually to each star's animation cycle. Once started, each star continuously fades in and

Figure 4.5: The multiple image animation benchmark in KMP (left) and Swift (right)

out with opacity oscillating between 0.0 and 1.0 using a linear transition, while simultaneously growing and shrinking in size between 0.7× and 1.3×. Both transitions have a 0.5 second duration per direction, forming a 1-second animation loop.

The behavior is implemented differently across platforms but achieves the same visual result. In the Swift version, two separate timers control opacity and scale per star, each launched after its own random delay and animated using `withAnimation`, similar to the Visibility benchmark. In the KMP version, Jetpack Compose's `rememberInfiniteTransition` handles both animations, using `StartOffset` to apply randomized delays for each star's animation.

Chapter 5

Results and Discussion

In this chapter, the results of the benchmarking study are presented and analyzed, with the goal of answering the research questions. Section 5.1 addresses RQ1, which concerns the performance of the hardware benchmarks, and Section 5.2 focuses on RQ2, which evaluates the UI rendering performance. The chapter concludes with Section 5.3, which provides a method discussion that reflects on potential improvements, and threats to validity.

5.1 RQ1: Hardware Benchmarks

This section presents and analyzes the benchmark results used to evaluate RQ1, which investigates whether KMP performs comparably to native Swift on iOS for hardware-related operations. Excluding warmup runs, each benchmark was executed 100 times across 5 separate runs. Execution time measurements are summarized in Table 5.1, CPU usage in Table 5.2 and memory usage in Table 5.3. All of these tables contain three rows, representing each of the hardware benchmarks, and six columns. The first two columns show the mean and standard deviation of the metric in the KMP implementation, while the next two correspond to the Swift implementation. The final two columns report the t-statistic and p-value from the t-tests comparing the two implementations. P-values below 0.05 are considered statistically significant, and are marked with an asterisk (*) in the tables. The raw performance data is found in Appendix A.

Table 5.1: t-test results of execution time (ms) across hardware benchmarks

	KMP (ms)		Swift (ms)			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Camera	641	43.6	632	60.1	1.21	0.227
File Write	274	218	177	203	3.24	0.0014*
File Read	135	24.6	94.8	3.66	16.1	< 0.001*

Execution time analysis

Table 5.1 presents the execution time results for all hardware benchmarks. Firstly, we observe no significant difference in execution times for the camera benchmark. The mean execution time for the KMP implementation is 641 ms, while the native Swift implementation has a mean of 632 ms. The p-value from the t-test is 0.227, which exceeds the 0.05 significance threshold, indicating that the difference is not statistically significant. This outcome is expected, as both implementations ultimately rely on the same underlying native framework, `AVFoundation`, for handling camera operations. In the Swift implementation, `AVFoundation` can be accessed easily in the code. However, the KMP implementation must invoke the same functionality through platform interoperability, using `expect/actual` declarations and calling native platform APIs via Kotlin's Objective-C/Swift interop layer. These results suggest that the KMP bridge introduces no significant execution time overhead.

In the file writing and file reading benchmarks, execution time is noticeably affected. For file writing, the KMP implementation has a mean execution time of 274 ms, compared to 177 ms for the Swift implementation. This means file writing is approximately 1.55 times slower on KMP. Similarly, for file reading, KMP averages 135 ms while Swift averages 94.8 ms, making KMP approximately 1.42 times slower in this benchmark. Both of these p-values are less than 0.05, indicating a significant difference.

Both the KMP and Swift implementation rely on the same underlying framework, `FileHandle`, but despite this the KMP version has a clear slowdown. Bridging via Objective-C/Swift interop likely introduces some overhead, but it is unlikely to be the main cause of the performance gap, especially given the minimal impact observed in the camera benchmark. A more probable explanation is the conversion cost between Kotlin's

`ByteArray` and Objective-C's `NSData`. When writing, the data must be converted from `ByteArray` to `NSData`, and when reading, the returned `NSData` must be translated back to a `ByteArray` before it can be used in shared Kotlin code. These conversions are expensive and likely a key contributor to the increased execution times observed in both read and write operations.

Similar execution time overhead has been observed in a previous study on Xamarin, which found that Xamarin was slower than native Swift at both reading and writing operations [35]. This suggests that such slowdowns are not unique to KMP, but are also present in other cross-platform frameworks.

CPU analysis

Table 5.2: t-test results of CPU usage (%) across hardware benchmarks

	KMP (%)		Swift (%)			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Camera	23.5	11.0	23.7	11.2	-0.482	0.630
File Write	17.0	9.57	24.5	10.7	-18.8	< 0.001*
File Read	36.4	4.24	43.3	1.88	-40.1	< 0.001*

Table 5.2 summarizes CPU usage across the hardware benchmarks. There is no significant difference in CPU usage between the native and KMP camera implementations. The KMP implementation averages 23.5% CPU usage, while the native implementation averages 23.7%, with a p-value of 0.630. This result is consistent with the execution time analysis. As previously discussed, both implementations rely on the same underlying `AVFoundation` framework, and the KMP interop layer does not introduce any noticeable CPU overhead in this case.

Interestingly, both the file writing and file reading benchmarks show lower CPU usage in the KMP implementation compared to Swift. File writing uses a mean of 17% CPU in KMP versus 26.8% in Swift, and file reading shows 36.4% in KMP versus 43.3% in Swift. Both differences are statistically significant. A possible explanation stems from the fact that the KMP benchmarks also uses longer execution times, and the additional time spent in KMP may not be due to CPU-intensive work but rather due to memory-related operations, such as the conversion between `ByteArray` and `NSData`. These

conversions could increase execution time without proportionally increasing CPU load, whereas the Swift implementation performs more of the file operations directly and more efficiently, leading to higher but shorter CPU utilization.

Memory usage analysis

Table 5.3: t-test results of memory usage (MB) across hardware benchmarks

	KMP (MB)		Swift (MB)			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Camera	117	37.1	14.0	1.51	167	< 0.001*
File Write	1076	462	111	0.0645	93.3	< 0.001*
File Read	889	219	58.7	29.6	105	< 0.001*

Table 5.3 summarizes the memory usage of the hardware benchmarks, and the results of the t-tests. The KMP implementation consistently uses significantly more memory than the native Swift implementation across all hardware benchmarks. In the camera benchmark, KMP consumes on average 117 MB compared to Swift's 14 MB, which is more than 8 times higher. For file writing, the difference is even more pronounced, with KMP using 1076 MB on average versus just 111 MB for Swift, a nearly 10 times increase. Similarly, in the file reading benchmark, KMP reaches 889 MB on average, while Swift remains at 58.7 MB, over 15 times lower. All of these differences are statistically significant with p-values well below 0.001, indicating that the elevated memory usage in KMP is a consistent effect.

One cause of the difference in memory usage is likely the fact that the KMP application uses a mark-and-sweep GC, while the native implementation relies on Swift's ARC, as described in Section 2.2.4. In the native Swift app, memory is released immediately once the last reference to an object is removed. In contrast, the KMP app retains allocated memory until a GC cycle is triggered, either after a certain time interval or when memory usage surpasses a certain threshold [19].

Figure 5.1 illustrates the memory usage during a single run of the file reading benchmark, where 20 files of 100 MB each, previously written to disk, are read sequentially. In the native Swift plot, 20 distinct peaks can be observed, each corresponding to one file read operation. After

each peak, memory usage quickly drops, indicating that the file content is promptly released. This behavior clearly reflects how ARC deallocates memory immediately once all references are removed. In contrast, the KMP plot shows that memory is not released after each read. Instead, it accumulates and is freed in larger chunks during GC cycles, likely triggered by the high memory pressure.

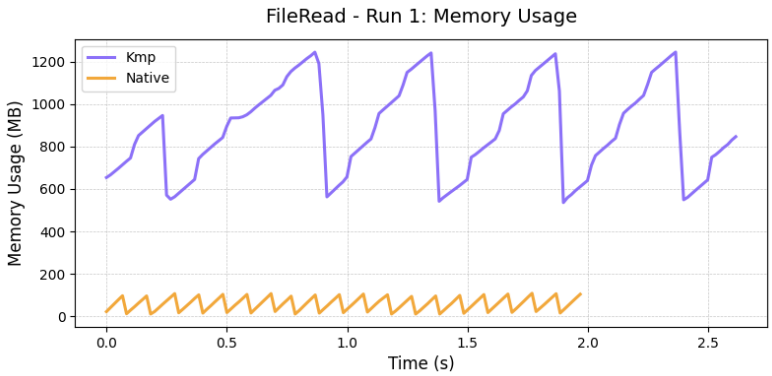

Figure 5.1: Memory plot of one execution of the file read benchmark

All five runs of the file reading benchmark follow a similar memory usage pattern to the one shown in Figure 5.1. The same pattern of memory accumulation and sudden release is also observed in the KMP implementations of the camera and file writing benchmarks. This behavior helps explain the consistently higher memory usage in KMP, as well as its substantially higher standard deviation compared to the native Swift implementations. The variability is a direct result of GC cycles, which introduce uncertainty to the time of memory release.

In the file reading benchmark, a higher standard deviation is also observed in both CPU usage and execution time for KMP. This, too, can be attributed to the differences between Kotlin/Native's GC and Swift's ARC. During the benchmark runs, occasional spikes in CPU usage and slower-than-average read operations were observed in the KMP implementation. These are likely caused by the GC mark phase interrupting normal execution, in contrast to ARC, which releases memory continuously and with less disruption.

The high memory usage of KMP cannot be explained solely by the differences between Kotlin/Native's GC and ARC; another contributing factor may be how background tasks are scheduled. Many of the operations in the camera and file benchmarks are asynchronous by nature. Capturing an image, saving it to the photo library, or reading and writing to local storage are

all tasks offloaded to background threads to keep the UI responsive. While the benchmarks themselves do not run tasks concurrently, the offloading still requires scheduling. As explained in Section 2.2.5, Kotlin relies on multiple coroutine dispatchers that manage thread creation and task scheduling at the application level. This requires the KMP app to maintain its own thread pools and task queues, increasing memory overhead. In contrast, Swift uses dispatch queues handled by the system-level GCD framework, which schedules tasks across all apps using a shared thread pool. This integration reduces the need for per-app scheduling infrastructure and likely leads to more efficient memory usage.

Table 5.4: Idle state memory usage

	Mean (MB)	**Std (MB)**
KMP	49.4	0.600
Native Swift	11.4	0.488

In addition to memory management and scheduling factors, the KMP implementation exhibits elevated memory usage even when the application is idle (see Table 5.4). This difference likely stems from the runtime overhead introduced by Kotlin/Native, as mentioned in Section 2.2.6. The KMP application includes memory for the Kotlin runtime itself, which provides core features such as memory management, coroutine scheduling, and type metadata. Unlike Swift, where ARC and GCD are implemented at the system level and do not require inclusion in the application binary, Kotlin must bundle these capabilities directly with the app. This contributes to a larger memory footprint and may also impact binary size and startup performance. While these factors were not measured in this study, prior work has shown that KMP applications tend to have significantly larger binaries than their Swift counterparts [4, 47], with one study also reporting longer startup times [47], further supporting the presence of notable runtime overhead.

Another reason for the high memory usage observed specifically in the file operations benchmarks is the overhead introduced by data conversions required for successful interoperability between Kotlin and the native iOS APIs. These conversions increase memory usage by requiring duplicate in-memory representations, both the original `ByteArray` and the resulting `NSData` (or vice versa) will coexist temporarily during file reads and writes. These duplicated allocations inflate memory usage, especially when working with large files, and contribute to the significant difference in memory

consumption between the KMP and native implementations.

A previous study found similar results that KMP applications consumed more memory than their native Swift counterparts across tasks such as networking, serialization, and local database access. A key explanation was the difference in memory management. KMP's GC led to delayed deallocation and higher memory peaks, while Swift's ARC showed more stable behavior [5]. These findings support the results in this thesis, suggesting that the memory overhead observed in hardware-related operations is part of a broader trend in KMP across multiple domains.

Hardware benchmarks summary

The results of these benchmarks show that KMP can achieve performance comparable to native Swift for certain hardware-related operations, particularly when data exchange between Kotlin and native code is minimal. For example, the camera benchmark exhibited no significant difference in execution time or CPU usage, suggesting that KMP interoperability is efficient when working with native APIs like `AVFoundation` that encapsulate most of the work within the platform layer.

In contrast, file operations highlight KMP's limitations, where execution time is slower and memory usage is significantly higher. This is likely due to data conversions between `ByteArray` and `NSData`, and the overhead of the Kotlin/Native runtime and GC.

In conclusion, the answer to RQ1 is nuanced, KMP can perform comparably to native Swift on iOS in some hardware scenarios, but significant overhead emerges in use cases with large data flows between the native APIs and shared code.

5.2 RQ2: UI Benchmarks

This chapter presents and analyzes the benchmark results used to evaluate RQ2, which examines whether KMP and CMP perform comparably to native Swift on iOS in terms of UI rendering. Each UI benchmark was executed for 100 seconds, excluding warmup time, and split across 5 separate 20 second runs. CPU usage measurements are summarized in Table 5.5, FPS in Table 5.6 and delayed frames per second in Table 5.7. Similarly to the hardware benchmark tables, all of these tables contain three rows, representing each of the UI benchmarks, and six columns. The first two columns show the mean and standard deviation of the metric in the KMP/CMP implementation,

while the next two correspond to the Swift implementation. The final two columns report the t-statistic and p-value from the t-tests comparing the two implementations. P-values below 0.05 are considered statistically significant, and are marked with an asterisk (*) in the tables. The raw performance data is found in Appendix A.

CPU usage analysis

Table 5.5: t-test results of CPU usage (%) across UI benchmarks

	KMP/CMP (%)		Swift (%)			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Visibility	22.8	1.47	3.75	0.284	982	< 0.001*
List scroll	23.7	3.54	9.79	1.21	285	< 0.001*
Multiple animations	96.9	2.05	39.6	1.94	1470	< 0.001*

Table 5.5 presents the CPU measurements for all UI benchmarks. Across all three UI benchmarks, CMP has significantly higher CPU usage than the native Swift implementation.

The most pronounced gap appears in the visibility benchmark. CMP averages 22.8% CPU usage, and Swift has a mean of 3.75% for Swift. The list scroll benchmark shows a similar pattern, though less extreme, with CMP averaging 23.7% and Swift 9.79%. Finally, the multiple animations benchmark imposes the highest load, with CMP reaching 96.9% average CPU usage, compared to 39.6% for Swift.

These results can largely be explained by how the two frameworks implement animations. In SwiftUI, animations such as opacity or scale changes are triggered using `withAnimation`, which delegates the animation to Core Animation. Core Animation handles interpolation and rendering on the GPU, using cached layer bitmaps, and avoids recomputing or redrawing on the CPU [30].

In contrast, CMP animations are driven by `rememberInfiniteTransition` and `animateFloat`. These APIs compute animation values on the main thread and trigger redraws at each frame. While they do not trigger a complete recomposition, they do produce updated draw commands for Skia on every frame. In the multiple animations benchmark, 200 small star composables are each animated independently for both alpha and scale. This leads to 200 sets

of animation evaluations and draw command updates per frame, resulting in a sustained high CPU load.

The list scroll benchmark does not involve explicit animations for alpha or scale, but still relies on continuous `animateScrollToItem()` calls driven by a coroutine loop in CMP. This causes frequent layout passes and draw updates, which again cannot match the CPU efficiency of Swift's Core Animation based scrolling, where the GPU handles much of the motion through bitmap translation.

In summary, Swift's use of Core Animation for GPU-accelerated rendering offloads most of the animation workload from the CPU, resulting in consistently low CPU usage, while CMP performs more of the animation and rendering logic on the main thread, which leads to significantly higher CPU usage across all benchmarks.

FPS and delayed frames analysis

Table 5.6: t-test results of FPS across UI benchmarks

	KMP/CMP		Swift			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Visibility	60.0	0	60.0	0	nan	nan
List scroll	58.0	0.885	60.0	0	-169	< 0.001*
Multiple animations	47.1	1.35	60.0	0	-651	< 0.001*

Table 5.7: t-test results of delayed frames/s across UI benchmarks

	KMP/CMP		Swift			
Benchmark	**Mean**	**Std**	**Mean**	**Std**	**t-stat**	**p-value**
Visibility	0	0	0	0	nan	nan
List scroll	0.93	0.778	0	0	12.0	< 0.001*
Multiple animations	7.16	2.23	0	0	-651	< 0.001*

Table 5.6 presents the FPS measurements and Table 5.7 the delayed frames per second for all UI benchmarks. In the visibility animation benchmark, both

the CMP and the native Swift implementation achieve a consistent 60.0 FPS, and 0 delayed frames. Since the two samples are identical, no p-value can be computed, but it is evident that there is no performance difference between the two. This suggests that, despite differences in graphics engines, the CMP implementation is capable of matching native performance for a simple image animation.

In the list scroll benchmark, differences between the CMP and native Swift implementations emerge. The CMP version achieves a mean frame rate of 58 FPS (0.93 delayed frames/s), while the Swift implementation maintains a steady 60 FPS (0 delayed frames/s). Both of these differences are statistically significant. This result is somewhat surprising, as CPU usage in the scroll benchmark is similar to the visibility benchmark, where the FPS and delayed frames were not negatively affected.

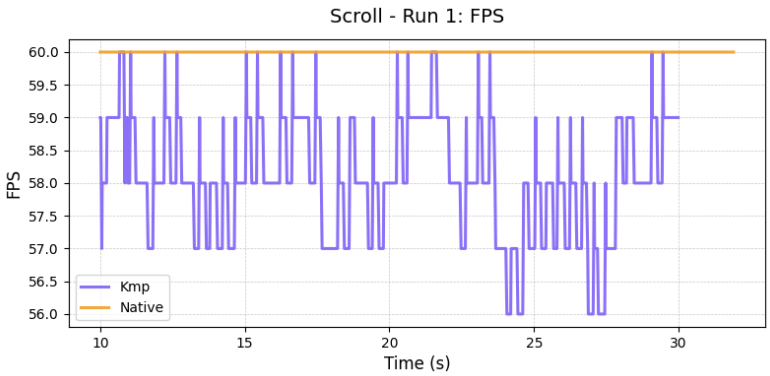

Figure 5.2: FPS plot of one execution of the scroll benchmark

To understand the cause of the frame drops, we examine a time series plot of the FPS during one execution of the scroll benchmark (Figure 5.2). The frame drops occur consistently and are evenly distributed throughout the run, as also reflected in the low standard deviation. There is no apparent correlation between the delayed frames and spikes in CPU usage, memory fluctuations, or mark-and-sweep GC events. This suggests that the frame drops are unlikely to be caused by CPU overload or main thread blocking. Instead, the bottleneck may lie in the rendering pipeline, either in the GPU itself or in the communication between Skia and the GPU. If too many or too complex draw commands are issued, the system may fail to complete rendering before the frame deadline, resulting in delayed frames even when CPU usage is low.

The reason the scroll benchmark appears to place a heavier load on the GPU than the visibility benchmark, is likely because the visibility animations

contains a single image with only changing opacity, which is lightweight and easily cached by the GPU. In contrast, the scroll benchmark renders multiple images per frame, increasing the number of draw operations. Had the scroll benchmark used a single repeated image, it is likely that no delayed frames would have occurred.

In the multiple animations benchmark, the CMP implementation exhibits significantly worse performance, with an average frame rate of 47.1 FPS, and 7.16 delayed frames/s. In contrast, the native Swift implementation sustains a constant 60 FPS with zero frame delays. Similar to the list scroll benchmark, the frame delays are evenly distributed over time, indicating that they are not caused by transient events such as Kotlin's GC or localized spikes in CPU usage.

In this case, however, the low and stable frame rate can be explained by the consistently high CPU load, which averages 96.9% (see Table 5.5). A strained CPU increases the risk of missing frame deadlines. When animation updates and draw preparation take too long, the system cannot prepare and deliver draw commands in time for the GPU, resulting in lower FPS and frames missing their deadline.

UI Benchmarks Summary

The results show that KMP/CMP does not achieve CPU performance comparable to native Swift. Across all three benchmarks, CMP consistently exhibits higher CPU usage, with particularly large differences in the visibility and multiple animations tests.

However, in terms of FPS and frame delay, CMP performs well for simple, single-element animations, as demonstrated in the visibility benchmark, where both platforms maintain a perfect 60 FPS. When multiple images are involved, as in the scroll and multiple animations benchmarks, a performance tradeoff becomes apparent, with CMP showing lower FPS and more delayed frames.

Previous studies comparing animation performance in cross-platform frameworks to native Swift have reported similar findings. One study found that both React Native and Xamarin exhibited higher CPU usage and rendered fewer frames compared to native implementations [37]. Another study observed that Flutter consumed more CPU than native Swift when rendering two types of animations [45], and a separate study found that Flutter also used more CPU during list scrolling tasks [40]. These results indicate that not only CMP, but also other cross-platform frameworks, struggle to match the animation performance of native Swift solutions, highlighting the challenge

of competing with highly optimized systems like Core Animation.

In conclusion, the answer to RQ2 is that KMP/CMP does not match native Swift in terms of performance. However, for applications with lower performance requirements and simple UI animations, CMP remains a viable and acceptable alternative.

5.3 Threats to Validity

In this section, methodological considerations are discussed, primarily focusing on implementation details that may have unintentionally influenced the results.

First, the act of measuring performance may introduce overhead. The performance calculator samples measurements at every frame and consumes system resources, adding a small but consistent layer of noise. This likely affects both implementations similarly, but still impacts the values.

Second, the measurement logic was implemented in Swift to ensure consistent data collection across both platforms. Ideally, a language-agnostic tool like Instruments would have been used, but no practical method for exporting the data was found. As a result, functions like `sampleTime()` in KMP call Swift's `systemUptime` via the interop layer. This may introduce som overhead but was considered worth it to be able to use the same measurement tools on both implementations.

A third thing to consider is the effect of the UI on the hardware benchmarks. Even though nothing except the home screen is rendered, simply rendering the screen and including the Compose runtime could introduce background overhead. Running each benchmark in an isolated application, instead of a combined app for all benchmarks, could reduce this effect and may be explored in future work.

Finally, design choices regarding which APIs to benchmark also matter. For example, Apple's `FileHandle` was selected for file I/O due to its support for flushing buffers, but other native alternatives like `NSData.write` or `NSOutputStream` may yield different performance results. Similarly, different animation libraries could change the observed behavior. Broader testing is needed to generalize findings across a wider range of use cases and APIs.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The purpose of this thesis was to evaluate whether KMP, and CMP can offer comparable performance to native Swift on iOS. The focus was on two key areas, hardware-related operations and UI rendering performance. A series of benchmarks were implemented in two versions of a test app, one using KMP/CMP and the other using native Swift. These benchmarks measured execution time, CPU usage, memory consumption, FPS, and delayed frames. The study aimed at answering the following research questions:

- **RQ1:** Does Kotlin Multiplatform perform comparably to native Swift on iOS when performing hardware-related operations?
- **RQ2:** Does Kotlin Multiplatform and Compose Multiplatform perform comparably to native Swift on iOS in terms of UI rendering?

The findings indicate that the answer to RQ1 is mixed. KMP can perform comparably to native Swift in certain hardware scenarios. In the benchmark evaluating camera performance, no significant overhead was introduced in terms of execution time or CPU, indicating efficient interoperability with native APIs. However, in the file operations benchmarks, the KMP implementation exhibited longer execution times but, on average, lower CPU usage. All three KMP implementations consumed more memory than their native Swift counterparts. These differences are largely attributable to the contrasting memory management strategies (Kotlin/Native's GC vs. Swift's ARC), the overhead of converting between `ByteArray` and `NSData`, and potentially also to differences in task scheduling and the inclusion of additional runtime components in the KMP implementation.

For RQ2, the results are more clear. CMP does not match the CPU efficiency of SwiftUI, with consistently higher CPU usage across all UI benchmarks, particularly in scenarios involving complex animations. The native implementations maintained a perfect frame rate with no delays across all benchmarks, while the CMP version could only keep up during the visibility animation, which involved just a single image. In the scroll and multiple animations benchmarks, where multiple images were rendered simultaneously, CMP began to show delayed frames. Overall, CMP can be considered a viable option for applications with simple UI requirements and moderate performance needs, but it is less suitable for graphically intensive or animation-heavy applications.

Both research questions were successfully addressed, and the goals of the thesis were met. The benchmarks provide empirical data that clarify the performance trade-offs involved in adopting KMP/CMP for iOS development. The main contribution of this thesis is a structured, reproducible methodology for evaluating performance trade-offs between KMP/CMP and native Swift, along with concrete results that can help developers make informed architectural decisions.

In summary, KMP and CMP should be adopted with caution when performance is critical. For applications where UI responsiveness and resource efficiency are top priorities, native Swift remains the stronger choice. However, for less performance-sensitive apps, KMP/CMP can be a good option, offering the benefit of shared code and fast cross-platform development.

6.2 Limitations

This section outlines key limitations and challenges that may have influenced the results and generalizability of the study.

One of the primary limitations is that all benchmarks were executed on a single iOS device. As such, the findings may not generalize to other devices with different hardware capabilities, such as older models or newer phones with more advanced GPUs. A broader device set would be needed to assess performance consistency across hardware generations.

Another limitation is that the study focused on CPU, memory, execution time, and frame rate metrics but did not measure GPU usage. In the UI benchmarks, Swift's use of Core Animation offloads rendering to the GPU, which helps explain its low CPU usage and consistent frame rates. However, the actual GPU load remains unknown. This may present Swift in a more

favorable light and limits the completeness of the UI performance analysis. Future work could incorporate GPU monitoring tools to provide a more comprehensive view.

In addition, CMP is still a relatively young and evolving framework. Performance issues observed today may improve over time as the framework matures. Meanwhile, SwiftUI and Apple's rendering stack have benefited from years of optimization. This means the results presented here are time-sensitive and may not reflect future performance of these tools.

On the Kotlin/Native side, one technical limitation was the inability to force GC between operations. In a similar study on Android, the authors were able to trigger a GC cycle manually between each file write, resulting in more stable memory usage readings [42]. This was not possible on iOS, adding some variability to the results.

6.3 Future Work

Several directions remain open for future research to build on the findings of this study.

First, future benchmarking efforts should aim to test across a wider range of devices. This would help determine how performance characteristics vary with different hardware capabilities, such as older iPhones or newer models with more advanced GPUs. Similarly, GPU metrics should be included to gain a more complete understanding of rendering performance. Measuring GPU load would be especially valuable for interpreting the efficiency of the SwiftUI framework, which offload significant work to the GPU via Core Animation.

As CMP continues to evolve, it would also be valuable to revisit these benchmarks over time. Since it is a relatively young framework, performance improvements may occur rapidly, and repeating the study in future versions could reveal meaningful progress.

Another interesting direction would be to benchmark more realistic application scenarios where hardware-intensive operations and UI rendering are performed together. This could help reveal how Kotlin/Natives mark-and-sweep GC and interop-related memory operations affect UI responsiveness. In this study, no noticeable frame delays due to GC were observed, likely because the UI benchmarks did not perform sufficiently memory-heavy operations to trigger GC. A combined test involving simultaneous UI rendering and large memory allocations could provide more insight into the interplay between Kotlin Native's GC and frame timing.

In addition to performance metrics, user studies could complement technical benchmarks by evaluating perceived experience. For example, while objective metrics like frame rate and delayed frames provide a clear performance picture, they do not fully capture how users perceive responsiveness. Therefore, future research could investigate whether slightly reduced performance in CMP actually leads to a worse user experience, or whether it remains acceptable for most use cases.

6.4 Reflections on Sustainability Aspects

This thesis contributes not only to technical knowledge, but also touches on broader environmental and economic aspects.

From an environmental perspective, performance optimization and the choice of development tools can have a direct impact on energy consumption. Efficient use of system resources, particularly the CPU, can lead to reduced battery consumption in mobile applications. This does not only consume less energy, but also contributes to longer battery lifespans, as the battery needs less frequent charging.

From an economic standpoint, the evaluation provided in this thesis can help companies make more informed decisions when choosing between native and cross-platform development strategies. KMP and CMP offer the potential to reduce development and maintenance costs by enabling shared codebases across iOS and Android. By identifying where these tools perform well and where they fall short, this study supports organizations in selecting the most cost-effective approach for their specific use case.

In summary, the evaluation of cross-platform tools like KMP and CMP is not only a technical matter but also one with implications for sustainability and cost-efficiency in modern mobile development.

References

[1] StatCounter, "Mobile operating system market share worldwide - february 2025," https://gs.statcounter.com/os-market-share/mobile/worldwide/#yearly-2009-2025, 2025, accessed: 2025-03-04. [Pages 1 and 7.]

[2] JetBrains, "Kotlin multiplatform," https://kotlinlang.org/docs/multiplatform.html, n.d., accessed: 2025-03-04. [Pages 1 and 9.]

[3] ——, "Compose multiplatform," https://github.com/JetBrains/compose-multiplatform/blob/master/README.md, 2025, accessed: 2025-03-03. [Pages 2 and 14.]

[4] A.-K. Evert, "Cross-platform smartphone application development with kotlin multiplatform: Possible impacts on development productivity, application size and startup time," Master's Thesis, KTH Royal Institute of Technology, 2019. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1368323&dswid=8094 [Pages 2, 19, 20, 21, and 48.]

[5] A. Skantz, "Performance evaluation of kotlin multiplatform mobile and native ios development in swift," Master's Thesis, KTH Royal Institute of Technology, 2023. [Online]. Available: https://kth.diva-portal.org/smash/record.jsf?pid=diva2:1793389&dswid=5765 [Pages 2, 20, 21, 33, and 49.]

[6] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2490257.2490292 p. 213–220. [Pages 2, 8, and 9.]

[7] BoardActive Software, “A brief history of mobile apps,” https://www.boardactive.com/post/a-brief-history-of-mobile-apps, 2020, accessed: 2025-03-11. [Page 7.]

[8] Apple, “Swift,” https://www.apple.com/ca/swift/, n.d., accessed: 2025-04-09. [Page 8.]

[9] JetBrains, “Kotlin on android. now official,” https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/, 2017, accessed: 2025-04-09. [Page 8.]

[10] Apache, “Ionic framework - the cross-platform app development leader,” https://ionicframework.com/, n.d., accessed: 2025-03-11. [Page 9.]

[11] ——, “Apache cordova,” https://cordova.apache.org/, n.d., accessed: 2025-03-11. [Page 9.]

[12] Meta, “React native - learn once, write anywhere,” https://reactnative.dev/, n.d., accessed: 2025-03-11. [Page 9.]

[13] Google, “Flutter,” https://flutter.dev/, n.d., accessed: 2025-03-04. [Page 9.]

[14] JetBrains, “The basics of kotlin multiplatform project structure,” https://kotlinlang.org/docs/multiplatform-discover-project.html, 2025, accessed: 2025-03-02. [Pages xi, 9, and 10.]

[15] ——, “Expected and actual declarations,” https://kotlinlang.org/docs/multiplatform-expect-actual.html, 2024, accessed: 2025-03-12. [Page 11.]

[16] ——, “Kotlin native,” https://kotlinlang.org/docs/native-overview.html, 2025, accessed: 2025-03-12. [Page 11.]

[17] Apple, “Avfoundation,” https://developer.apple.com/av-foundation/, n.d., accessed: 2025-04-09. [Pages 11 and 37.]

[18] JetBrains, “Integration with swift/objective-c arc,” https://kotlinlang.org/docs/native-arc-integration.html, 2025, accessed: 2025-03-03. [Pages 11 and 12.]

[19] ——, “Kotlin/native memory management,” https://kotlinlang.org/docs/native-memory-manager.html, 2024, accessed: 2025-03-03. [Pages 11, 12, and 46.]

[20] Apple, “Automatic reference counting,” https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/, 2014, accessed: 2025-03-03. [Pages 12 and 13.]

[21] JetBrains, “Coroutines,” https://kotlinlang.org/docs/coroutines-overview.html, 2025, accessed: 2025-07-12. [Pages 12 and 13.]

[22] ——, “Coroutinedispatcher,” https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/, n.d., accessed: 2025-07-12. [Page 12.]

[23] ——, “Commonpool default for coroutines,” https://discuss.kotlinlang.org/t/commonpool-default-for-coroutines/11965, 2019, accessed: 2025-07-12. [Page 12.]

[24] Apple, “Dispatchqueue,” https://developer.apple.com/documentation/dispatch/dispatchqueue, n.d., accessed: 2025-07-12. [Page 13.]

[25] ——, “Dispatch,” https://developer.apple.com/documentation/dispatch, n.d, accessed: 2025-07-30. [Page 13.]

[26] JetBrains, “Integration with the swiftui framework,” https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-swiftui-integration.html, 2024, accessed: 2025-05-05. [Page 14.]

[27] Android Developers, “Jetpack compose phases,” https://developer.android.com/develop/ui/compose/phases, 2025, accessed: 2025-05-27. [Page 14.]

[28] Skia, “Documentation,” https://skia.org/docs/, 2025, accessed: 2025-05-05. [Page 14.]

[29] JetBrains, “Skiko - skia for kotlin,” https://github.com/JetBrains/skiko, 2025, accessed: 2025-05-05. [Page 14.]

[30] Apple, “Core animation programming guide, core animation basics,” https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreAnimation_guide/CoreAnimationBasics/CoreAnimationBasics.html#//apple_ref/doc/uid/TP40004514-CH2-SW3, 2015, accessed: 2025-05-01. [Pages 15 and 50.]

[31] T. Capin, K. Pulli, and T. Akenine-Moller, “The state of the art in mobile graphics research,” *Computer Graphics and Applications, IEEE*, vol. 28, pp. 74–84, 2008. doi: 10.1109/MCG.2008.83 [Page 15.]

[32] V. Ahti, S. Hyrynsalmi, and O. Nevalainen, "An evaluation framework for cross-platform mobile app development tools: A case analysis of adobe phonegap framework," in *Proceedings of the 17th International Conference on Computer Systems and Technologies*. Association for Computing Machinery, 2016. doi: 10.1145/2983468.2983484 p. 41–48. [Page 15.]

[33] P. Karami, I. Darif, C. Politowski, G. El Boussaidi, S. Kpodjedo, and I. Benzarti, "On the impact of development frameworks on mobile apps," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023. doi: 10.1109/APSEC60848.2023.00023 pp. 131–140. [Page 15.]

[34] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, "A survey and taxonomy of core concepts and research challenges in cross-platform mobile development," *ACM Computing Surveys*, vol. 51, pp. 1–34, 11 2018. doi: 10.1145/3241739 [Page 15.]

[35] P. Grzmil, M. Skublewska-Paszkowska, E. Łukasik, and J. Smołka, "Performance analysis of native and cross-platform mobile applications," *Informatics, Control, Measurement in Economy and Environmental Protection (IAPGOS)*, vol. 7, p. 50–53, 2017. doi: 10.5604/01.3001.0010.4838 [Pages 16, 18, and 45.]

[36] X. Jia, A. Ebone, and Y. Tan, "A performance evaluation of cross-platform mobile application development approaches," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. Association for Computing Machinery, 2018. doi: 10.1145/3197231.3197252 p. 92–93. [Page 16.]

[37] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, "Animations in cross-platform mobile applications: An evaluation of tools, metrics and performancek," *Sensors*, vol. 19, no. 9, p. 2081, 2019. doi: 10.3390/s19092081 [Pages 16, 19, and 53.]

[38] S. Huber and L. Demetz, "Performance analysis of mobile cross-platform development approaches based on typical ui interactions," in *Proceedings of the 14th International Conference on Software Technologies*. SCITEPRESS – Science and Technology Publications, 2019. doi: 10.5220/0007838000400048 pp. 40–48. [Pages 16 and 18.]

[39] A. Biørn-Hansen and Others, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," *Empirical Software Engineering*, vol. 25, pp. 2997–3040, 2020. doi: 10.1007/s10664-020-09827-6 [Pages 16 and 18.]

[40] M. Olsson, "A comparison of performance and looks between flutter and native applications," Master's Thesis, Blekinge Institute of Technology, 2020. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1442804/FULLTEXT01.pdf [Pages 17, 18, and 53.]

[41] D. Mota and R. Martinho, "An approach to assess the performance of mobile applications: A case study of multiplatform development frameworks," in *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering*. SCITEPRESS - Science and Technology Publications, 2021. doi: 10.5220/0010497401500157 pp. 150–157. [Pages 17 and 18.]

[42] K. Wasilewski and W. Zabierowski, "A comparison of java, flutter and kotlin/native technologies for sensor data-driven applications," *Sensors*, vol. 21, no. 10, p. 3324, 2021. doi: 10.3390/s21103324 [Pages 17, 18, 20, 21, and 57.]

[43] H. Andersson, "A comparison of the performance of an android application developed in native and cross-platform," Bachelor's Thesis, Blekinge Institure of Technology, 2022. [Online]. Available: https://bth.diva-portal.org/smash/get/diva2:1666657/FULLTEXT01.pdf [Pages 17, 18, and 19.]

[44] G. Tollin and M. Lidekrans, "React native vs. flutter: A performance comparison between cross-platform mobile application development frameworks," Bachelor's Thesis, Linköping University, 2023. [Online]. Available: https://liu.diva-portal.org/smash/get/diva2:1768521/FULLTEXT01.pdf [Pages 17 and 18.]

[45] S. Ametova and T. Lindström, "Exploring the performance gap: How animation implementation affects the cpu and ram usage in mobile applications: Among cross-platform and native development approaches," Bachelor's Thesis, Jönköping University, 2023. [Online]. Available: https://hj.diva-portal.org/smash/record.jsf?pid=diva2%3A1783215&dswid=1770 [Pages 17, 19, and 53.]

[46] W. Oliveira, B. Moraes, F. Castor, and J. a. P. Fernandes, "Analyzing the resource usage overhead of mobile app development frameworks," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, 2023. doi: 10.1145/3593434.3593487 p. 152–161. [Pages 17 and 19.]

[47] M. Guśpiel, "Mobile app development using kotlin multiplatform," Bachelor's thesis, Laurea University of Applied Sciences, 2024. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-202404237210 [Pages 18, 19, 20, 21, and 48.]

[48] V. Ionzon and S. Jägstrand, "A company case study: Examining criteria in cross-platform evaluation frameworks," Bachelor's Thesis, Linnaeus University, 2022. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1673341 [Page 20.]

[49] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000. ISBN 9780521646703 [Pages 24 and 25.]

[50] Apple, "Testing a release build," https://developer.apple.com/documentation/xcode/testing-a-release-build, n.d., accessed: 2025-04-08. [Page 29.]

[51] D. Yates, D. S. Moore, and D. S. Starnes, *The Practice of Statistics*, 3rd ed. W. H. Freeman and Company, 2008. ISBN 9780716773092 [Page 30.]

[52] Gavrilov Daniil, "Gdperformanceview-swift," https://github.com/dani-gavrilov/GDPerformanceView-Swift/tree/master, 2020, accessed: 2025-04-23. [Pages 33 and 34.]

[53] Apple, "systemuptime," https://developer.apple.com/documentation/foundation/processinfo/systemuptime, n.d., accessed: 2025-05-14. [Page 33.]

[54] ——, "Phphotolibrary," https://developer.apple.com/documentation/photos/phphotolibrary, n.d., accessed: 2025-05-14. [Page 37.]

[55] ——, "Filehandle," https://developer.apple.com/documentation/foundation/filehandle, n.d., accessed: 2025-05-14. [Page 38.]

Appendix A

Source Code and Results

Kotlin and Compose Multiplatform Benchmark App

https://github.com/VanjaVidmark/master-thesis/tree/main/kmp-benchmarks

Native Swift Benchmark App

https://github.com/VanjaVidmark/master-thesis/tree/main/native-benchmarks

Raw Performance Data

https://github.com/VanjaVidmark/master-thesis/tree/main/data

Full GitHub Repository

https://github.com/VanjaVidmark/master-thesis

TRITA-EECS-EX-2025:640
Stockholm, Sweden 2025