Master's thesis

Master's Programme in Computer Science

On-Device Machine Learning Inference using Cross-Platform Frameworks

Ossi Lehtonen

May 31, 2024

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Ossi Lehtonen

Työn nimi — Arbetets titel — Title

On-Device Machine Learning Inference using Cross-Platform Frameworks

Ohjaajat — Handledare — Supervisors

Prof. Jukka K. Nurminen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 31, 2024	52 pages, 7 appendix pages

Tiivistelmä — Referat — Abstract

Machine learning (ML) and mobile applications are two major technology trends in the past decade. Arguably, developing a successful mobile application is difficult, but ML can help with the challenges. ML enables new capabilities for an application, but there are multiple obstacles to be passed before publishing a ML feature to production. One of them is the inconstancy in mobile devices' hardware. A developer might be able to achieve fast and accurate enough inference in their own sandbox, but actual end-users might observe a bad user experience (UX) using an application due to inefficient inference because their device hardware varies from the developer's setup. Cross-platform frameworks are intended to provide consistent UX across operating systems (OSs) with a single codebase, which in turn shortens the development time compared to creating individual native applications for different OSs. Even though applications built using a cross-platform framework are generally less performant than those built with native languages, they can achieve near-native performance in certain tasks through the use of native modules. This makes it interesting to combine ML inference and cross-platform development and to compare the inference capabilities of the currently most common cross-platform frameworks: React Native and Flutter. The results of this thesis indicate that running hardware-accelerated ML inference is possible in both frameworks using open-source libraries. And the inference is efficient when it comes to execution time, especially on newer devices. But to achieve generally good inference time and accuracy across different devices without sacrificing UX, one should most likely lean towards React Native.

ACM Computing Classification System (CCS)

Software and its engineering → Software notations and tools → Software libraries and repositories

Computing methodologies → Machine learning → Machine learning approaches

Avainsanat — Nyckelord — Keywords

cross-platform development, machine learning, inference

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction **1**

2 Cross-platform Mobile Development **3**

2.1 Approaches . . . 3

2.2 React Native and Flutter . . . 5

2.2.1 React Native . . . 5

2.2.2 Flutter . . . 9

3 Machine learning inference in Mobile applications **13**

3.1 On-Device Inference . . . 14

3.1.1 Model Design and Compression . . . 14

3.1.2 Inference Acceleration . . . 15

3.1.3 High-level Machine Learning Libraries . . . 16

3.1.4 Using Cross-Platform Frameworks . . . 17

4 Methods **20**

4.1 Earlier Work of Experiments . . . 20

4.2 Experiment Setup . . . 21

4.2.1 Tasks and Models . . . 21

4.2.2 Devices . . . 22

4.2.3 React Native . . . 22

4.2.4 Flutter . . . 24

4.3 Performance Evaluation . . . 25

4.3.1 Metrics . . . 25

4.3.2 Test system . . . 25

5 Results **28**

5.1 Image Classification . . . 28

5.2 Object Detection . . . 31

5.3 Semantic Image Segmentation 32
5.4 Overview of AITs 33
5.4.1 Libraries 33
5.4.2 Delegates 34

6 Discussion **37**
6.1 Inference Speed 37
6.1.1 React Native vs Flutter 37
6.1.2 Library AIT Performance 38
6.1.3 Library Performance Stability 40
6.1.4 Delegate AIT Performance 40
6.2 Other Observations 42
6.2.1 Summary 43
6.3 Threats to Validity 43

7 Conclusions **46**

Bibliography **47**

A All Test Results **i**

B Code of the Test System

1 Introduction

In today's world, mobile devices have a significant role in people's lives. In 2022, there were already 8.36 billion mobile cellular subscriptions across the world (The World Bank, 2024). Mobile devices do not serve only as communication channels anymore; they are also essential devices for a wide range of daily activities. The embedding of machine learning has further expanded the capabilities of mobile applications. This thesis explores the integration of machine learning inference into mobile applications using popular cross-platform frameworks, specifically React Native and Flutter.

The reasons for the emergence of machine learning are multifaceted. There have been improvements in machine learning algorithms (Fradkov, 2020), and open-source machine learning software has made it possible for a wide range of users to build machine learning features. One of the key enablers for the emergence of machine learning are the advancements in computing power. Particularly the advances in graphics processing units (GPUs) and neural processing units (NPUs). With modern hardware, the processing of large datasets, machine learning model training, and inference have become significantly faster and more efficient. Not only traditional computers, but also mobile devices, have taken huge leaps in computational power. These advantages of mobile devices' computational power make it possible to run machine learning inference on-device. The main benefits of running inference on-device compared to performing it on a server are: offline inference capabilities, no network overhead, reduced server maintenance, and enhanced privacy by keeping user data on the device (Lee et al., 2019).

Despite the leaps in the processing power of hardware, the difficulties of applying efficient on-device inference originate from the fact that mobile devices' computing resources are limited compared to traditional computers (Xu et al., 2020; Lee et al., 2019). It has been shown that efficient inference can be run on-device, but it is difficult to achieve efficient inference across different devices due to a lack of programmability when it comes to accessing inference accelerators (Wu et al., 2019). One of the reasons for this is that the inference acceleration interfaces are OS- or even device vendor-specific.

Luckily, this trend is changing. Apple, for example, has introduced Metal (Apple, 2024b) and CoreML (Apple, 2024a) APIs. And Android provides Neural Network API (Android Developers, 2024) for performing efficient inference on-device. Moreover, open-source

cross-platform machine learning libraries that are capable of utilizing OS-specific inference acceleration through native modules have been published in the past few years. Cross-platform frameworks typically introduce at least a slight performance overhead (Biørn-Hansen et al., 2020). But the combination of a cross-platform framework with a machine learning inference library utilizing native modules should, in theory, not only perform efficient inference but also help with the programmability problem addressed by (Wu et al., 2019).

Hence, in this thesis, I will evaluate on-device machine learning inference using cross-platform mobile application development tools, with a focus on React Native and Flutter. These frameworks are chosen for the evaluation, as they are currently the most popular tools (AppStudio, 2024; Redwerk, 2024; ScaleupAlly, 2024) and are both open-source. Both of the frameworks support Android and iOS, which hold the biggest market share across operating systems (Stats, 2024). The research questions are:

1. What is the current state of open-source machine learning libraries in on-device libraries for React Native and Flutter?
2. How do React Native and Flutter compare in terms of machine learning inference performance across different devices, and what are the underlying technical factors contributing to these performance differences?

To answer the research questions, a custom test system is built. The test system consists of Flutter and React Native applications and a Data API. In both applications, two different ML libraries are used to run inference, and the results are gathered by the Data API. Four different devices (2 iOS and 2 Android) are used to collect the results. The inference is tested for various different models with different inference accelerators. The main focus is on evaluating the inference speed and the technical factors behind the results. Additionally, I will briefly discuss inference accuracy, developer experience with the ML libraries, and the limitations of the libraries.

The rest of the thesis is sectioned as follows: In Sections 2 and 3, I give relevant background on cross-platform mobile development and on-device machine learning. The details of the test system are introduced in Section 4. Section 5 gives an overview of the results, and the reasonings for the results are discussed in Section 6. And finally, I will conclude the thesis in Section 7.

2 Cross-platform Mobile Development

Mobile applications can be built using various programming languages and frameworks. These tools can be divided into two main branches: cross-platform development and native development. In cross-platform development, a single codebase is used to create an application for multiple operating systems. This can be achieved with various different approaches (Xanthopoulos and Xinogalos, 2013). Cross-platform development leads to faster development time compared to native development, as native applications require different codebases for different operating systems. In native development, the target platform's native programming languages and tools are used to create an application. Generally speaking, native applications provide better performance (Biørn-Hansen et al., 2020; Oliveira et al., 2023) compared to cross-platform tools. And hence the user experience (UX), as they can take full advantage of the device's hardware and software capabilities. However, sometimes the faster development time outweighs the performance effects, making cross-platform development more suitable for a project.

Cross-platform mobile development frameworks have evolved significantly in the past 10 years. In 2012, the most popular frameworks were Rhodes, PhoneGap, DragonRad, and MoSync (Palmieri et al., 2012). Out of the four, only PhoneGap, now known as Apache Cordova, is still a somewhat relevant cross-platform mobile development framework. Modern cross-platform development frameworks provide support for integrating native code into a project. There is usually at least a minimal overhead in calling native code, but it has been shown that cross-platform frameworks can perform equally well with native applications in some tasks (Biørn-Hansen et al., 2020).

2.1 Approaches

Cross-platform mobile development has four main approaches: Web apps, Hybrid apps, Interpreted apps, and Generated apps (Xanthopoulos and Xinogalos, 2013).

Web Apps are not installed on a device but are accessed through web browsers. They are typically built using HTML and JS, making them portable across operating systems. As of today, Web apps can utilize some of the device's native features. For example, geolocation features, camera, microphone, and vibration. Still, a lot of features are unavailable for

Web apps, such as advanced Bluetooth capabilities, Near-field communication, and full file system access. Some popular frameworks for Web apps are React.JS, Angular, and Vue.JS. The downsides, in addition to the lack of native features, are performance and the lack of native feel in UX (Xanthopoulos and Xinogalos, 2013). This is because Web apps, by nature, are subject to the performance of the web browser and internet speed. And native user interface (UI) components are unavailable in the browser.

Progressive Web Apps (PWAs) are a step closer to native apps compared to Web Apps (Sheppard and Sheppard, 2017). They use service workers to allow the running of code in the background and provide offline support (Biørn-Hansen et al., 2020). Still, the problem of a lack of platform-specific features remains. An example of a PWA framework is Angular.JS.

Hybrid apps are applications that try to leverage Web apps' and native apps' benefits (Xanthopoulos and Xinogalos, 2013). The applications are built using HTML and JS and are then wrapped inside native containers. A hybrid app can be installed on-device but it still lacks some of the native features. Ionic is an example of a Hybrid app framework.

Generated apps are highly performant and usually have the feel of native apps, as they are compiled to native code (Xanthopoulos and Xinogalos, 2013). In theory, they have access to all device features, including hardware accelerators, as long as the development frameworks provide support for them. Xamarin (Microsoft, 2024) is a development tool that compiles C#-code code into native iOS and Android code. A Xamarin application might still require some platform-specific code for complex UIs. Kotlin Multiplatform (Kotlin, 2024) is a relatively new development tool that compiles Kotlin into native OS code. Flutter compiles to native code and hence falls into the category of Generated Apps.

Interpeted apps offer a balance between performance and versatility (Xanthopoulos and Xinogalos, 2013). They can leverage most of the native features and provide a near-native user experience while still maintaining a single codebase for multiple platforms and a good developer experience. The performance is high but usually cannot reach that of a native application. React Native, due to its unique architecture, fits somewhere between Generated and Interpeted apps.

2.2 React Native and Flutter

React Native and Flutter are chosen for the comparison, as they are currently the top two cross-platform development frameworks (Redwerk, 2024; AppStudio, 2024; ScaleupAlly, 2024) and they provide good support for Android and iOS. Both of the frameworks are mature open-source projects with wide community support. The UI is built using reusable code in both of the frameworks: components in React Native and widgets in Flutter. Both Flutter and React Native introduce overhead in terms of performance and resource usage (Oliveira et al., 2023; Biørn-Hansen et al., 2020). The code execution architecture varies between the frameworks due to the different programming languages. To optimize the calculation of heavy computational tasks or to offload the calculation to hardware accelerators by calling native modules, the frameworks provide different solutions.

2.2.1 React Native

React Native is developed by Facebook, and it enables building mobile apps using JavaScript and React (Meta Platforms, Inc., 2024). React Native has similar syntax to React, making it easy for React developers to adopt React Native. In the core of React are reusable components; see Figure 2.1. React Native has a unique code execution architecture, and it differs from React due to the execution environment. It combines JavaScript with native platform capabilities. A JS engine is used to run JS code, and a JavaScript Interface (JSI) is used to communicate with modules written in PHP with specific native features.

The components of React Native are written using Javascript XML (JSX). Unlike in React, no HTML is used to write components, but the consept of components still remains; see Figure 2.2. Compared to the React code, the <button> html element is a <Button> component from the React Native library, and the <div> html element is a <View> component. The <Button> and <View> components are examples of built-in native components in React Native. The behavior of some native components slightly differs between OSs, but with the use of them, React Native's UI can be built to feel native.

The architecture of code execution in React Native is unique. Figure 2.6 illustrates the architecture. React Native uses two different threads to execute code: the JS thread and the Native/UI thread. The code executed in the JS thread is executed by a JS engine. Using the default React Native JS engine, Hermes, the developer-written JS code is compiled to bytecode at build time. At runtime, Hermes executes this pre-compiled

```
export const ExampleButton = ({ title }) => {
    <button>{title}</button>
}
```

```
const App = () => {
    return (
        <div>
            <h1>An Example Application</h1>
            <ExampleButton title="Click me" />
        </div>
    )
}
```

Figure 2.1: React's reusable component. The component ExampleButton can be used across a React application.

```
import { Button } from 'react-native';

const ExampleButton = ({ title }) => {
  return <Button title={title} />
}
```

```
const App = () => {
  return (
    <View>
      <ExampleButton title="Click me" />
    </View>
  )
}
```

Figure 2.2: React Native's reusable component. The component ExampleButton can be used across a React Native application.

```
const ExpensiveComponent = () => {
  const expensiveResult = expensiveJSFunction();
  return (
    <View>
      {
        // Display the result of the expensive function
      }
    </View>
  );
};
```

Figure 2.3: A component which is resource expensive for JS engine. Component cannot be rendered before the function *expensiveJSFunciton* is ready.

bytecode, so no just-in-time compilation (JIT) is needed to execute JS.

The Native/UI thread is responsible for and optimized for rendering platform-specific UI components. However, other resource-consuming tasks can also be run on the thread. Using this approach, the single-threaded JS engine is not blocked, which would lead to poor performance and UX. To execute native code on the Native/UI thread, for example, Swift on iOS, the JS engine communicates through JSI with the native code (Meta Platforms, Inc., 2024). The JSI exposes native method references directly to the JS code. This is accomplished with the use of C++ Host Objects, making the overhead of communicating with the Native/UI thread minimal.

For example, in Figure 2.3, a resource-expensive JS function is run before rendering an UI component. This can make the UI feel unresponsive, as the rendering needs to wait for a function result. Note that by initializing the result to null and running the *expensive-JSFunction* after the initial render to display the result, one can make the component's initial state render faster, but the feel of sluggish UI still remains while the expensive function is running. See Figure 2.4. To solve the problem, the expensive function should make a call to the native module that runs the function to not block the JS thread for the calculation of the result (Figure 2.5).

```
const ExpensiveComponent2 = () => {
  const [expensiveResult, setExpensiveResult] = useState(null);

  useEffect(() => {
    const result = expensiveJSFunction();
    setExpensiveResult(result);
  }, []);

  return (
    <View>
      {
        // Display the result of the expensive function
      }
    </View>
  );
};
```

Figure 2.4: A component which is resource expensive for JS engine. The initial render is faster, but the expensiveJSFunction still blocks the JS thread, leading to unresponsive UI.

```
import { NativeModules } from 'react-native';

const { ExampleModule } = NativeModules;

const expensiveJSFunction = async () => {
  const result = await ExampleModule.expensiveFunction();
  return result;
};
```

Figure 2.5: A expensive JS function that calls native module to perform calculation. This offloads the calculation away from JS thread.

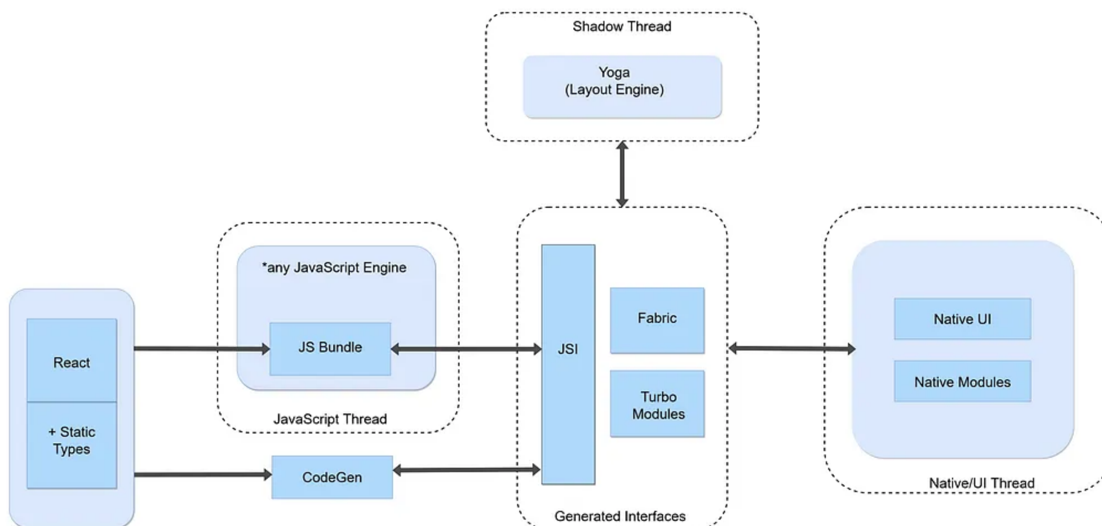

Figure 2.6: Overview of React Native's internal architecture (Coox Tech, 2023).

2.2.2 Flutter

Flutter (Google, 2024a), created by Google, is a UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase. Flutter apps are built using the Dart programming language, and the Dart code is compiled into native code, which ensures high performance. UI elements are built using reusable components called widgets. The UI of a Flutter app is drawn by a high-performance rendering engine.

The equivalent of React Native's components are the widgets in Flutter. See Figures 2.7 and 2.8. Unlike in React Native, no native UI components are used. Flutter compiles all of its code ahead-of-time and Flutter's rendering engine is responsible for drawing the UI. The initial Flutter rendering engine is called Skia. However, due to the "early-onset jank" issue on Skia, a new rendering engine called Impeller has been developed (Google, 2024b) which should solve the problem.

Even though Dart is single-threaded, Flutter is not strictly single-threaded. While the main UI thread (or platform thread) handles rendering and gesture detection, Flutter also supports using multiple threads for executing Dart code. Dart's approach to concurrency is Isolates. They run on their own memory space and communicate asynchronously. This architecture allows Flutter to perform complex tasks without blocking the UI, though the UI itself is built and rendered on a single thread. For example, if a heavy computational task is run without the Isolates, the code is run on the main UI thread (Figure 2.9), possibly making the UI unresponsive as there is a race condition between UI rendering and the task. In Figure 2.10 Isolates are used to solve the problem. Much like in React Native, one can call platform-specific modules in Flutter. The messages are passed from Dart code to platform specific code using platform channels (Flutter, 2024). When running resource-intensive tasks with native code, one can call an Isolate which then uses a platform channel to communicate with native code. This ensures the UI is not blocked while the task is running.

```
class ExampleButton extends StatelessWidget {
  final String title;

  ExampleButton({super.key, required this.title});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      child: Text(title),
      onPressed: () {
        // Define your action here
      }
    ); // ElevatedButton
  }
}
```

Figure 2.7: Example of a reusable Flutter Widget.

```
class MyComponent extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: ExampleButton(title: 'Click Me'),
    ); // Center
  }
}
```

Figure 2.8: A reusable Flutter Widget is used in a separate Widget.

```
class NoIsolateExpensiveWidget extends StatefulWidget {
  @override
  _NoIsolateExpensiveWidget createState() => _NoIsolateExpensiveWidget();
}

class _NoIsolateExpensiveWidget extends State<NoIsolateExpensiveWidget> {

  @override
  void initState() {
    super.initState();
    _performComputation();
  }

  void _performComputation() {
    // Perform your heavy computation here
    String result = heavyComputation();

    // do something with the result
  }

  String heavyComputation() {
    // Dummy heavy computation
    int sum = 0;
    // performs a heavy computation
    return "Computation result: $sum";
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('No Isolate Computation Example')),
      body: Center(
        child: Text("No Isolate Computation Example"),
      ), // Center
    ); // Scaffold
  }
}
```

Figure 2.9: Example of a Flutter expensive widget. No Isolates are used, making the *heavyComputation* run on the main thread, possibly blocking the UI.

```
class _ExpensiveWidget  extends State<ExpensiveWidget> {

  @override
  void initState() {
    super.initState();
    _startHeavyComputation();
  }

  void _startHeavyComputation() async {
    ReceivePort receivePort = ReceivePort();
    await Isolate.spawn(_heavyComputation, receivePort.sendPort);

    receivePort.listen((data) {
      // do something with the result
    });
  }

  static void _heavyComputation(SendPort sendPort) {
    // Perform heavy computation here, e.g., ML inference
    String result = "Computation result";
    sendPort.send(result);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Heavy Computation Example')),
      body: Center(
        child: Text("Heavy Computation Example"),
      ), // Center
    ); // Scaffold
  }
}
```

Figure 2.10: Example of a Flutter expensive widget. An Isolate is used, offloading the computation for another thread from the main thread. Hence, keeping the UI more responsive.

3 Machine learning inference in Mobile applications

Machine learning (ML) is a subset of artificial intelligence that uses statistical algorithms to learn from data. ML is about understanding and finding patterns in data. The patterns are then used to make predictions or decisions. ML algorithms are designed to learn from and make decisions based on input data. Usually, the learning process involves large amounts of data because it allows for the discovery of patterns and features in a phenomenon. The four main types of learning are:

1. **Supervised**. Training is done using labeled dataset, where the expected output is known. The goal of a supervised learning learning algorithm is to use inputs to predict outputs (Hastie et al., 2009).

2. **Unsupervised**. The training is done using unlabeled dataset. Unsupervised algorithms learn and present structures from data (Mahesh, 2020).

3. **Semi-Supervised**. Both labeled and unlabeled data is used for the training (Mahesh, 2020). It can be useful e.g. if there is no fully labeled dataset available.

4. **Reinforcement**. A agent is used to perform actions of a particular task (Kaelbling et al., 1996). The agent improves in performing the task by trial-and-error intercations in a dynamic environment.

This thesis focuses on models achieved by supervised learning. After a supervised learning model has been trained on a dataset, it can be used to make ”inference”. Inference refers to the process of feeding a machine learning model input data and receiving an output. This is the practical use of ML models, where they are applied to real-world scenarios. The challenges of applying performant on-device inference are mainly related to the hardware of mobile devices, but different approaches have been developed to tackle the problems.

3.1 On-Device Inference

The focus of this thesis is on inference, which is performed on-device, specifically defined as the process of running inference with the hardware of a mobile device. Running machine learning inference on a mobile device is sometimes impossible or extremely slow due to the low computational power on the device (Xu et al., 2020; Lee et al., 2019). By offloading the inference to a server, the mobile application is only responsible for a possible pre-processing of the data and rendering the server response result for the user. However, offloading the inference to a server behind the internet is obviously impossible if the device is offline. In addition, offloading computing to a server can introduce data privacy concerns (Dzombeta et al., 2014; Lee et al., 2019) and it might be costly network-wise, affecting the inference performance.

Additionally, Wu et al. (2019) recognize the diversity of smartphones' hardware as a challenge for running efficient machine learning inference on a wide range of devices. Making unique optimizations for inference performance is not feasible for thousands of different chipsets. It's not only important to optimize inference speed but also to take into consideration the stability of performance, model sizes, and the effect of inference on battery consumption. Three high-level concepts for improving on-device inference are model design, model compression, and inference acceleration (Xu et al., 2020). The most relevant concept regarding this thesis is inference acceleration, as different accelerators are utilized with different machine learning libraries.

3.1.1 Model Design and Compression

Researchers have made significant work on introducing efficient mobile neural network (NN) models (Sun et al., 2020; Howard et al., 2017; Sandler et al., 2018). These approaches focus on reducing model size through careful re-design of network architecture and weight quantization. Additionally, using these techniques, faster computation and a lower memory footprint can be achieved in resource-constrained devices (Shao and Zhang, 2020)

Modern ML models have significant amounts of parameters. For example, LlaMa models have paremeters up to 75 billion (Touvron et al., 2023). The increase in model sizes not only leads to slower-performing models but also results in increased memory requirements, meaning that careful model design is beneficial for efficient on-device inference. So-called

mobile models such as MobileBERT (Sun et al., 2020) and Mobilenets models (Howard et al., 2017; Sandler et al., 2018) have been designed for on-device inference.

Models not designed for on-device inference can be improved with model compression (Shao and Zhang, 2020). With the help of model compression, model sizes and resource consumption can be lowered. The process should not affect the accuracy of a model in a significant way. Two of the main approaches to compression are quantization and model pruning. In quantization, the model parameters are stored in a less accurate format, leading to a lower memory footprint and possibly faster computation. For example, 32-bit floating numbers are converted to 8-bit integers. This has been shown to accelerate inference when running on the CPU without dramatically affecting accuracy (Vanhoucke et al., 2011). Model pruning is the process of reducing the number of neurons and connections in a neural network model. It is also shown to be effective in improving inference time and lower energy consumption without dramatically affecting model accuracy (Zhu and Gupta, 2017).

3.1.2 Inference Acceleration

Inference acceleration can be divided into two subsets: hardware acceleration and software acceleration (Xu et al., 2020). Hardware acceleration means the process of running inference fully or partly on a dedicated computing resource, such as a GPU or NPU. Software acceleration, on the other hand, tries to optimize resource management, pipeline design, and compilers. Both Android and iOS support hardware and software acceleration through their own hardware and interfaces. In addition, there exists accelerators that are universal, regardless of the OS.

Wu et al., 2019 recognizes that optimizing ML inference across different hardware is challenging due to the hardware variety and lack of programmability. This is particularly the case with Android phones. Due to the challenges, a lot of ML inference is run on central processing units (CPUs), leading to slower inference times than what the use of GPUs or other hardware accelerators would offer. In recent years, the landscape has been changing. Apple has introduced Core ML (Apple, 2024a) and Neural Engine for its devices, and Android's Neural Networks API (NNAPI) (Android Developers, 2024) has been developed. Both APIs try to provide interfaces for utilizing hardware efficiently in the respective OSs. A common term used in this context is 'delegate,' which refers to software mechanisms that offload inference tasks to optimized hardware or software libraries. For example, Core

ML delegate utilizes Apple's hardware accelerators efficiently, while Android's NNAPI is able to tap into the available accelerators on Android devices.

Core ML and Android NNAPI have significantly advanced ML inference on mobile devices. Core ML is designed to integrate machine learning models into iOS apps efficiently. It optimizes performance by leveraging CPUs, GPUs, and Neural Engines, thus ensuring seamless operation across various Apple devices. There is also a possibility to use Apple's Metal API (Apple, 2024b) to accelerate on-device inference with GPUs on iOS devices, even though the Metal API was originally meant for graphics rendering.

NNAPI, on the other hand, is an alternative for Android devices. It provides a base layer of functionality for higher-level machine learning frameworks, enabling them to execute across a variety of Android devices more efficiently. NNAPI aims to harness the power of different hardware accelerators, including GPUs and specialized NPUs, to enhance the speed and efficiency of ML inference operations. Some Android phone manufacturers, such as Samsung, have built their own SDKs for accessing hardware acceleration features in their hardware. However, using NNAPI is recommended to be used when possible, as the interface for developers remains the same (Sipola et al., 2022).

Additionally to the advancements made through Core ML and Android NNAPI, WebGL (Mozilla Developer Network, 2024) plays a crucial role in facilitating ML inference, especially in web-based applications. WebGL was originally a JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins. However, it has also been leveraged to be used in machine learning. It allows for the utilization of a device's GPU for computations typically required in machine learning tasks. OpenGL is similar technology to WebGL but meant to be used in native applications; it is also used as a backend for some React Native ML libraries.

Lastly, XNNPACK (Google, 2024c) is a library supported by Google that is optimized to support neural network inference on CPU architectures, specifically on ARM, x86, WebAssembly, and RISC-V platforms.

3.1.3 High-level Machine Learning Libraries

Higher-level open-source on-device ML libraries such as Tensorflow Lite (TFLite) (TensorFlow, 2024c) and ONNX Runtime (ONNX Runtime, 2024) have been improved to support different inference accelerators such as CoreML, Metal, NNAPI, OpenGL, and XNNPACK, in addition to the possibility of running inference on a CPU. The purpose

of these high-level libraries is that developers do not need to directly communicate with the delegates, which makes it more feasible to use hardware acceleration when performing on-device inference.

TFLite (TensorFlow, 2024c) is Google's lightweight solution for on-device machine learning. Normal Tensorflow models can be converted to .tflite format using TFLite SDKs. The SDKs also provide support for model compression through quantization. These models can then be deployed to a mobile device and run using TFLite using platform-specific TFLite APIs.

TFLite provides support for CoreML, NNAPI, OpenGL, and XNNPACK (TensorFlow, 2024d). However, Core ML and NNAPI are not available for all devices, especially older models. TFLite also provides an interface to use a GPU delegate for hardware acceleration on both iOS and Android devices. For Android, one can also use Hexagon delegate for older devices without NNAPI support (TensorFlow, 2024a).

ONNX is a format used to represent machine learning models in a standardized way. Popular model formats such as .tflite or PyTorch's .pt models can be converted to ONNX models. ONNX Runtime (ONNX Runtime, 2024) is an engine used to run inference using ONNX models. There exists ONNX Runtime implementations for various different programming languages, such as Python, C++, Java, JS, and Objective-C. Similar to TFLite, ONNX Runtime provides support for CoreML, NNAPI, and XNNPACK.

3.1.4 Using Cross-Platform Frameworks

Model design and compression are still both relevant when it comes to running on-device ML inference using cross-platform frameworks; however, this thesis primarily focuses on the acceleration of ML inference. The models used in the evaluation are pre-trained and intended for mobile use.

Both ONNX Runtime and TFLite are available for Flutter and React Native through different open-source ML libraries. These ML libraries add another layer of abstraction on top of TFLite and ONNX Runtime; see figure 3.1. The cross-platform ML library calls a native module (TFLite or ONNX Runtime), which then uses a delegate to perform inference on the hardware of a mobile device.

Related work

A lot of the work regarding machine learning inference in cross-platfrom mobile applications is focused on implementations in which the inference is done at the server-side (Roopashree and Anitha, 2021; Wijesinghe et al., 2021; Kumar et al., 2022; Elhassan et al., 2023; Maitriboriruks et al., 2022; Lei et al., 2023). The studies are more focused on the whole system implementation and use cases than the inference part of the system. The impact of the cross-platform development tool on the inference performance is not critical if the inference is offloaded, as the device is only responsible for possible data preprocessing and sending and receiving of the input for the server.

The inference of a model can also be partly run on the device and the cloud (Huang et al., 2023). The method allows flexible architecture that can adapt to different devices' capabilities, and privacy-sensitive data does not need to be sent to the server. However, it adds some additional complexity to the system. The authors use a web application for the on-device inference. No relevant work using Generated or Interpreted cross-platform mobile development frameworks seems to have been done using this approach.

Sun (2020) and Cui et al. (2022) also use Web apps to perform cross-platform on-device inference. The machine learning inference is run using JavaScript (JS), making it portable for any device with a browser that has the capability of running JS. As discussed earlier, the major downside of the approach is the performance of a browser application compared to a native application (Xanthopoulos and Xinogalos, 2013).

Mohzary et al. (2023) present an application that detects DeepFake images. The application is built using React Native and performs on-device inference. A rapid detection speed (under 200 ms) is achieved by deploying a Deep Neural Network model on-device using Tensorflow.js library.

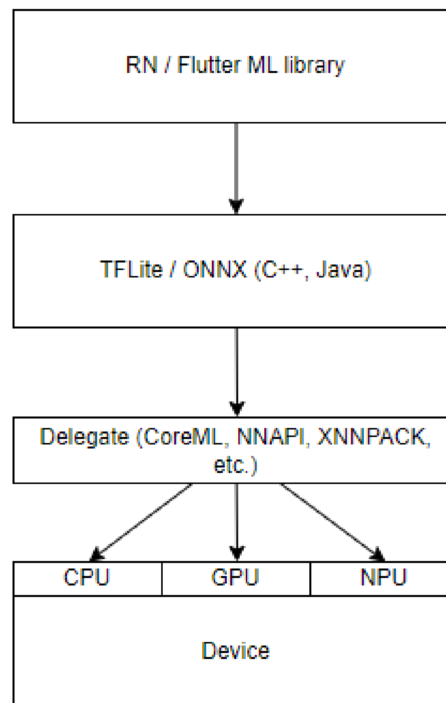

Figure 3.1: Simplified image of the layers when performing on-device inference using cross-platform frameworks.

4 Methods

In this Section, an overview of the test setup for the evaluation of the ML inference on React Native and Flutter is covered. First, an overview of previous work comparing Flutter and React Native performance is given. Then, the selected ML libraries, models, devices, and the architecture of the test system are introduced.

4.1 Earlier Work of Experiments

There has been no significant work done on evaluating on-device inference using cross-platform mobile development frameworks. The performance evaluations made for Flutter and React Native are focused on different on-device tasks than machine learning inference.

Biørn-Hansen et al. (2020) compares Ionic, NativeScript, MAML/MD2, Flutter, and React Native with a baseline Java application. The work is similar to this thesis as it focuses on calls made from a cross-platform context to native modules. The benchmark tasks contain calls to the geolocation API, contacts API, file system integration, and accelerometer integration. There is no clear winner between Flutter and React Native, as they excel at different tasks and measurements. However, React Native had a significantly slower time-to-completion compared to the baseline on most of the tasks. Flutter, on the other hand, compares a lot better with the baseline on time-to-completion.

The paper from Oliveira et al. (2023) compares the resource usage overhead of Flutter, React Native, and Ionic to a native Android app written in Java. They have similar results to the other papers: Flutter seems to generally have less overhead in execution time and energy usage than React Native. However, they find that React Native comes on top when it comes to rendering native animations. This is because React Native can efficiently use native modules to perform the animations.

CPU usage, memory usage, response time, frame rate, and application size of Flutter and React Native are compared in the work from Mahendra and Anggorojati (2020). The baseline native Android app outperforms Flutter in CPU and memory usage, while Flutter outperforms React Native in measurements. The tests are performed on device simulators.

User interaction performance of React Native is compared to the native Android baseline

in work by Huber and Demetz (2019). Similarly to other papers, CPU and memory usage of React Native is high in this use case.

4.2 Experiment Setup

The selected ML tasks and models, devices, framework versions and configurations, and ML libraries are presented in this Section.

4.2.1 Tasks and Models

The selected tasks and models for performance evaluation are partly derived from (Janapa Reddi et al., 2022). Other benchmark suites, such as the work from (Ignatov et al., 2018) have been introduced for mobile inference testing. Both suites include image classification, object detection, and semantic image segmentation. However, the models and test data from (Janapa Reddi et al., 2022) are easily accessible and open-source (MLCommons, 2024a). All of the models are designed for mobile environments. Table 4.1 shows an overview of the models used in this thesis.

The image classification task is done using both MobileNetEdgeTPU and MobileNetV2 (Sandler et al., 2018). MobileNetEdgeTPU should provide good performance on various SoCs (Janapa Reddi et al., 2022). MobileNetEdgeTPU is modified from MobileNet-v2 and is explicitly designed for mobile devices. The input images for both models are from the ImageNet dataset and of size 224x224.

SSD-mobilenet v2 is used for object detection. In the original benchmark specification, two different object detection models were used, but only SSD-mobilenet v2 is selected for this evaluation as the input data is easier to access. The image resolution is 300x300. The images are from the Coco 2017 dataset.

For semantic image segmentation, DeepLab v3+ with MobileNet v2 backbone is used. The ADE20K dataset is used with 32 classes. The image resolution is 512x512.

The input data and every model except MobileNetv2 are from the Github repository (ML-Commons, 2024b), which is also used by (Janapa Reddi et al., 2022). The MobilenetV2 model is from Keras library (Keras, 2024). Each of the models contains a single-precision floating-point 32 format (FP32) and an 8-bit unsigned integer (UINT8) version. The original models are in .tflite format, but for the ONNX libraries, all models were converted

	Input dimesions	FP32 Size	UINT8 Size
MobileNetEdgeTPU	224x224	15.9 MB	4.1 MB
MobileNetV2	224x224	3.7 MB	3.9 MB
SSD-mobilenet v2	300x300	65.8 MB	6 MB
DeepLab v3+	512x512	8.9 MB	2.3 MB

Table 4.1: The used models and corresponding disk sizes (in TFlite format). The sizes of the Onnx versions of the models are all within 0.25 MB of their TFlite equivalents.

to .onnx format using an open-source library tf2onnx (Open Neural Network Exchange (ONNX), 2024). Note that the MobileNetV2 UINT8 version size is bigger than the FP32 version by 200KB, possibly due to inefficient conversion.

4.2.2 Devices

For both Android and iOS, two different devices were selected: an old model and a newer model. This makes it possible to validate the inference performance across different generations of hardware. The newer devices provide further hardware acceleration through NPUs. Table 4.2 shows an overview of the selected devices.

The selected iOS devices are iPhone 7 and iPhone 12 mini. iPhone 7 contains an A10 Fusion chip, which contains a quad-core CPU and a six-core GPU. The device has LPDDR4 RAM chips with 2 GB of RAM. The iPhone 12 Mini, on the other hand, contains 4 GB of RAM and an A14 Bionic chip. The A14 Bionic chip has a 6-core CPU and a 4-core GPU. Notably, the chip also contains a 16-core Neural Engine. A Neural Engine is a type of NPU dedicated to Apple's devices.

Samsung Galaxy A40 and Samsung Galaxy S20 FE are selected for Android devices. Samsung Galaxy A40 contains an Octa-core CPU, a Mali-G71 MP2 GPU, and 4 GB of RAM. The latter device contains a newer version of the Octa-core CPU, an Adreno 650 GPU, 6 GB of RAM, and a NPU.

4.2.3 React Native

Expo (Expo, 2024) version 50 is used for React Native. The React Native version used is 0.73.4. Expo eases the development of React Native applications and was thus chosen for use instead of React Native CLI.

Table 4.2: Devices used, operating systems, systems on chip (SoC), and SoC details. Note that the Android devices have different versions of Octa-core.

	OS	**SoC**	**CPU**	**GPU**	**RAM**	**NPU**
iPhone 12 Mini	iOS 17.4.1	A14 Bionic	Hexa-core	Apple GPU (4-core)	4 GB	16-Core Neural Engine
iPhone 7	iOS 15.8.2	A10 Fusion	Quad-core 2.34 GHz	PowerVR Series7XT Plus (6-Core)	2 GB	-
Samsung Galaxy A40	Android 11	Exynos 7904	Octa-core	Mali-G71 MP2	4 GB	-
Samsung Galaxy S20 FE	Android 13	Snapdragon 865	Octa-core	Adreno 650 (2-Core)	6 GB	Contains NPU, but details not available

Both React Native and Flutter have open-source libraries that utilize TFLite and ONNX Runtime for efficient inference. These libraries are partly able to utilize inference acceleration through different delegates such as Core ML, Metal API, and NNAPI. This should make the inference close to what is achievable through native development. The frameworks were selected based on three main criteria: sufficient documentation, the ability to use custom models, and the fact that the library should provide some OS-specific inference acceleration.

The architecture of React Native, discussed in 2.2.1, allows the development of ML libraries on top of native code. It would be possible to run inference using only the JS engine. However, the selected React Native libraries have optimized inference with the use of native code.

Selected machine learning libraries

The library react-native-fast-tflite (Rousavy, 2024) is a relatively new library that promises to support hardware acceleration through WebGL, CoreML, and Metal APIs. But the Metal API did not work on the test system built. Currently, it does not support hardware acceleration for Android.

ONNX Runtime is utilized by the onnxruntime-react-native library (Microsoft and contributors, 2024). It is an open-source project backed by Microsoft. The library is part of the bigger onnxruntime library.

A library called @tensorflow/tfjs-react-native (Github, 2024) is relatively popular, but it does not contain comprehensive documentation for custom model usage and only supports WebGL delegate. It is hence omitted from the thesis.

Name	Version	Backed by	Community support / popularity
react-native-fast-tflite	1.2.0	Individual developers only	Growing
onnxruntime-react-native	1.17.1	Microsoft	Low

Table 4.3: Selected React Native ML inference libraries.

4.2.4 Flutter

The used Flutter version is 3.13.9. A production build (IPA for iOS and APK for Android) was built using Flutter SDK. The Impeller engine discussed in section 2.2.2 is not yet fully supported on Android and is hence omitted in this thesis, and the initial engine, Skia, is used in the performance analysis for both iOS and Android. The usage of the rendering engine should not affect the results, as there is no UI rendering when performing the inference tests. The ML libraries were selected with similar criteria as the libraries chosen for React Native. The selected Flutter libraries are able to utilize platform-specific hardware acceleration through the Flutter platform channels introduced in Section 2.2.2.

Selected machine learning libraries

Tensorflow provides tflite_flutter plugin (TensorFlow, 2024b) for integrating TFLite with Flutter. Among other things, the library supports Android NNAPI and other Android GPU delegates. On iOS, both Metal and CoreML delegates are supported.

The plugin onnxruntime_flutter provides an API to integrate and run inference on ONNX models in Flutter applications. Similarly to tflite_flutter, it is able to provide acceleration using multi-threading. The library authors promise that the inference speed should not be slower than native Android or iOS apps that use the native ONNX Runtime Java/Objective-C API.

Name	Version	Backed by	Community support / popularity
tflite_flutter	0.10.4	Google	High
onnxruntime_flutter	1.4.1	Individual developers only	Low

Table 4.4: Selected Flutter ML inference libraries.

4.3 Performance Evaluation

In this Section, the selected performance evaluation metrics are introduced. Additionally, an overview of the implemented test system is given, along with limitations related to it.

4.3.1 Metrics

The selected metrics for the tests are inference speed and accuracy. Model accuracy is only calculated for the image classification task. The ground truth labels for the other models were not easily accessible.

For each combination of library, delegate, device, and model, the tests were conducted at least three times. There was a cooldown period between the runs to keep the device from overheating. Subsequently, the inference times and outputs of the models were collected for each run. The average inference time (AIT) was then computed for each combination. The AIT was determined by calculating the mean of the inference times obtained across the runs for each input index. This method ensured that the typical performance (as indicated by the AIT) was comprehensively assessed for each combination of library, delegate, device, and model.

4.3.2 Test system

Custom React Native and Flutter applications were built for the performance evaluation of the ML libraries using the four different devices. The applications are extremely simple UI-wise, only focusing on the inference performance. In addition, a data API was built to serve input data for the applications and to save the inference results for later analysis. The links for the source code can be found from Appendix B.

The high-level architecture of running the inference tests is similar for both React Native and Flutter; see Figure 4.1. The input data is fetched from the mobile application using the data API. The data is then preprocessed. The preprocessed data and function for running inferences on the model for an input are given to a performance tester module. The module runs inference for each input one by one, capturing the inference time (and output from image classification tasks) and then sending the results to the data API. The data API saves the results for later analysis.

For React Native, a production build (IPA for iOS and APK for Android) was built using

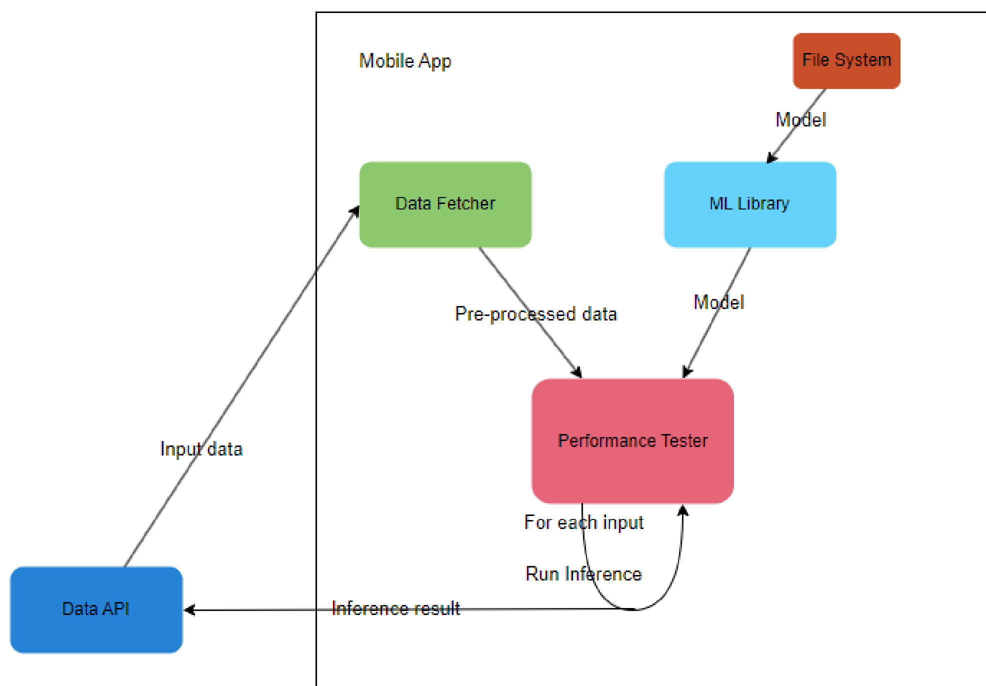

Figure 4.1: Simplified high-level architecture of the test setup. The Flutter and React Native applications use similar internal architectures.

Table 4.5: Supported delegates for a device and library pair

	react-native-fast-tflite	**onnxruntime-react-native**	**tflite_flutter**	**onnxruntime_flutter**
iPhone 12 Mini	CoreML, OpenGL	CoreML, CPU, XNNPACK	CoreML, Metal, CPU, XNNPACK	CoreML, XNNPACK
iPhone 7	OpenGL	CoreML, CPU, XNNPACK	Metal, CPU, XNNPACK	CoreML, XNNPACK
Samsung Galaxy A40	-	NNAPI, CPU, XNNPACK	NNAPI, GPU, CPU, XNNPACK	NNAPI, CPU, XNNPACK
Samsung Galaxy S20 FE	-	NNAPI, CPU, XNNPACK	NNAPI, GPU, CPU, XNNPACK	NNAPI, CPU, XNNPACK

Expo's EAS Build. The Flutter applications were built using Flutter SDK locally. Both React Native and Flutter applications have the same features. Both contain individual screens for each framework-specific ML library. On these screens, there is a button that starts the inference tests. The tests are run for all of the models and available delegates on the device.

It is important to note that the available delegates are not the same for each device and library pair. Naturally, NNAPI is only available for Android devices. And CoreML and Metal delegates are only available for iOS devices. The react-native-fast-tflite only supports iOS. It does not support CoreML delegate for iPhone 7 because TFLite do not support CoreML on devices without a Neural Engine. The same holds for the tflite_flutter library. Additionally, the tflite_flutter library has support for Android GPU delegate. The onnxruntime_flutter package supports CPU delegate only for Android, and the XNNPACK delegate only works for UINT8 models. Table 4.5 shows the working delegates for each device and library.

The inferences of the Flutter libraries (onnxruntime_flutter and tflite_flutter) were both run without the use of Isolates. This is because the performance with Isolates was discovered to be significantly worse than without Isolates. For example, SSD-Mobilenet v2 FP32 with an iPhone 12 mini and onnxruntime_flutter XNNPACK delegate achieves a 9 ms average inference time without the use of Isolates. With Isolates, the corresponding average inference time is around 65 ms. A significant increase in the use of Isolates was present with both of the Flutter libraries; hence, the usage of Isolates was omitted.

On the other hand, the React Native tests were run asynchronously, fully offloading inference away from the main thread, as onnxruntime-react-native does not have an option to run inference synchronously. And there were no significant differences in inference speed between synchronous and asyncronous executions with react-native-fast-tflite.

5 Results

In this Section, I present the results of the experiments comparing the AIT across React Native and Flutter apps, considering the different devices, models, data types, ML libraries, and delegates. The experiments were conducted using the test system discussed in Section 4. The underlying causes for the results are discussed in Section 6.

First, for each setup, the best-performing and worst-performing ones regarding AIT are reported in this Section in textual format. Additionally, the differences in accuracy for the image classification models are presented. Lastly, the summaries of the AIT results are given at the end of this Section. For detailed results of the tests, see Appendix A.

5.1 Image Classification

The specification from Janapa Reddi et al. (2022) requires top-1 accuracy of 74% from the MobileNetEdgeTPU model. However, the required accuracy is based on the full validation dataset accuracy. In these performance tests, only a subset (300 images) of the validation dataset was used, explaining the lower accuracy.

It should be noted that the accuracies between MobileNetEdgeTPU and MobilenetV2 are not directly comparable, as the validation dataset represents more of MobileNetEdgeTPU's training data. The maximum difference in Top-1 accuracy observed with the MobileNetEdgeTPU model is 0.33% in both FP32 and UINT8 models and 1.66% with the MobileNetV2 model. The difference of 1.66% is only observed when using iPhone 7. The key takeaway regarding accuracy is that there are some, but not huge, differences in accuracy between the library, delegate, and device used.

Additionally, no conclusions should be drawn from the accuracy differences between the FP32 and UINT8 versions of a model, as the preprocessing steps in the test setups might not be optimal due to a lack of information on effective preprocessing for the pretrained models from Janapa Reddi et al. (2022). However, the preprocessing is the same for each device, library, and delegate pair, meaning that cautious conclusions can be drawn from the effects of a device, library, and delegate for accuracy.

MobileNetEdgeTPU FP32

iPhone 12 Mini. The top 2 AITs (1 ms) were obtained with onnxruntime_flutter and react-native-fast-tflite. The onnx-runtime-react-native library achieves an AIT of 2 ms. tflite_flutter with the XNNPACK delegate performed worst, with a 121 ms AIT. Despite big differences in the inference speeds, there is no variance in the accuracy across the setups.

iPhone 7. The onnxruntime_flutter library with CoreML delegate is at the top when it comes to AIT (22 ms). Like with iPhone 12 Mini, the tflite_flutter with XNNPACK delegate performed worst (316 ms), but the inference accuracy remains the same with all the setups.

Samsung Galaxy A40. The top two performers are onnxruntime_flutter and onnx-runtime-react-native with CPU delegates (158 ms and 167 ms AITs). The slowest-performing setup is tflite_flutter with an XNNPACK delegate. Again, there is no variance in the accuracy between the used library and delegate setup.

Samsung Galaxy S20 FE. With onnxruntime_flutter and the NNAPI delegate, the best AIT of 25 ms is achieved. The worst AITs are with the tflite_flutter library, with the CPU delegate having as high as 281 ms AIT. All of the libraries with the NNAPI delegate have 0.33 % better accuracy than the rest of the setups.

MobileNetEdgeTPU UINT8

iPhone 12 Mini. The fastest AIT (7 ms) is achieved with onnxruntime_flutter library using the XNNPACK delegate. The onnxruntime-react-native with the same delegate and react-native-fast tflite with OpenGL and CoreML delegates are close seconds with 8 ms AITs. The tflite_flutter with Metal delegate performs the worst with 61 ms AIT. There is a maximum difference of 0.33% in accuracy between the setups.

iPhone 7. AIT of 40 ms with react-native-fast-tflite and OpenGL delegate is the fastest, outperforming the second setup by 10 ms. The slowest setup is tflite_flutter with the XNNPACK delegate (111 ms AIT). What is interesting is that tflite_flutter with metal delegate has 0.67% worse accuracy than the highest performing setups.

Samsung Galaxy A40. onnxruntime_flutter with CPU and NNAPI delegate perform the best, having 65 ms and 67 ms AITs. tflite_flutter with NNAPI and XNNPACK delegates both have 296 ms AIT, performing worst with the device. However, tflite_flutter has

0.33% better accuracy on the device compared to other libraries.

Samsung Galaxy S20 FE. CPU delegate with onnxruntime_flutter and onnxruntime-react-native are the top two AIT performers (10 ms and 12 ms AITs). With NNAPI delegates from the libraries having 13 ms and 15 ms AITs. Again, tflite_flutter has the worst performance but achieves 0.33% better accuracy compared to other libraries.

MobileNetV2 FP32

iPhone 12 Mini. The react-native-fast-tflite library with OpenGL and CoreML delegates perform almost identically (7 ms AIT), and both outperform other setups. tflite_flutter with a CPU delegate (104 ms AIT) performs the worst. The react-native-fast-tflite with both OpenGL and CoreML delegates achieve the worst accuracy. Having 1.33% worse accuracy than tflite_flutter with XNNPACK and 1% worse than the rest of the setups.

iPhone 7. Similarly to the iPhone 12 Mini, react-native-fast-tflite using OpenGL has the top AIT (23 ms). Close second being onnxruntime_flutter with 25 ms AIT when using XNNPACK delegate. And tflite_flutter CPU and XNNPACK delegates have as slow as 170 ms AIT. The accuracy results are similar to those of the iPhone 12 Mini, with the difference being that onnxruntime-react-native CoreML has 0.33% worse accuracy than what the setup has with the iPhone 12 Mini.

Samsung Galaxy A40. onnxruntime_flutter with XNNPACK, NNAPI, and CPU delegates (98, 100, and 100 ms AITs) have the best performance. tflite_flutter with CPU and NNAPI delegates perform the worst with 419 ms of AIT each. The tflite_flutter XNNPACK delegate has 0.33% better accuracy than the other setups. Accuracy results are the same as with the iPhone 12 Mini.

Samsung Galaxy S20 FE. XNNPACK, CPU, and NNAPI delegates with onnxruntime-react-native all have the best AITs (26 ms). onnxruntime_flutter with the same delegates achieves similar results having 29–30 ms AITs. XNNPACK with tflite_flutter achieves the best accuracy by 0.33% but performs worst with 212 ms AIT. The tflite_flutter with NNAPI delegate has 0 accuracy.

MobileNetV2 UINT8

iPhone 12 Mini. Multiple setups are able to achieve sub-6 ms AIT without fluctuation in accuracy. tflite_flutter with the Metal delegate is performing worst (49 ms AIT).

The onnxruntime_flutter with both available delegates and tflite_flutter with XNNPACK delegate performs worst regarding accuracy, having an accuracy of 62.67%, compared to the accuracy of 66.33% most of the setups have. onnxruntime-react-native with CoreML delegate has the best accuracy, 64.0%.

iPhone 7. onnxruntime_flutter with CoreML delegate has the best AIT of 16 ms, but the accuracy is 0.66% worse than with onnx-runtime-react-native with CPU delegate. The former has 17 ms AIT. Unsurprisingly, tflite_flutter performs worst, with metal delegate having only 93 ms AIT. The accuracy results are the same as with the iPhone 12 Mini, but the onnxruntime-react-native with CoreML delegate has 64.33% accuracy.

Samsung Galaxy A40. onnxruntime_flutter with CPU and NNAPI delegates perform the best (34 ms AITs). Again, tflite_flutter performs the worst, with NNAPI and CPU delegates having 226 ms of AIT each. The accuracy results are similar to those of the iPhone 12 Mini, but the tflite_flutter GPU delegate has an accuracy of 64%.

Samsung Galaxy S20 FE. onnxruntime_flutter with NNAPI has the best AIT (8 ms). However, various other setups, such as onnxruntime-react-native with NNAPI, have a 9-ms AIT. tflite_flutter performs worst regarding AIT, and CPU and GPU delegates with the library have the worst accuracies with the device.

5.2 Object Detection

For the SSD-mobilenet v2 model, there is no accuracy metric calculated.

SSD-mobilenet v2 FP32

iPhone 12 Mini. Using CoreML delegate, onnxruntime_flutter, react-native-fast-tflite, and onnxruntime-react-native are all able to perform with sub-6 ms AITs. The tflite_flutter library with the XNNPACK delegate performs as badly as 410 ms AIT.

iPhone 7. onnxruntime-react-native with CoreML delegate has the lowest AIT of 86 ms. Again, tflite_flutter performs worst, with 521 ms AIT.

Samsung Galaxy A40. The onnxruntime_flutter with CPU delegate performs the best, but with only 359 ms AIT. The tflite_flutter with CPU delegate performs the worst, having 1273 ms AIT.

Samsung Galaxy S20 FE. CPU delegate with onnxruntime_flutter with 61 ms AIT is the

best performer. tflite_flutter is again the slowest library with all of the delegates.

SSD-mobilenet v2 UINT8

iPhone 12 Mini. There are four setups performing similarly (9 ms AITs). These are: onnxruntime_flutter with XNNPACK delegate, react-native-fast-tflite with CoreML and OpenGL delegates, and onnxruntime-react-native with XNNPACK delegate. The tflite_flutter with Metal delegate performs the worst, with 89 ms AIT.

iPhone 7. react-native-fast-tflite with the OpenGL delegate performs the best with 47 ms AIT. onnx-runtime-react-native with XNNPACK delegate is close second with 52 ms AIT. And tflite_flutter with the XNNPACK delegate performs the worst, having 231 ms AIT.

Samsung Galaxy A40. onnxruntime-flutter and onnxruntime-react-native with CPU delegates perform the best, with 70ms and 78ms AITs. The onnxruntime-react-native with the NNAPI delegate has only 970 ms of AIT.

Samsung Galaxy S20 FE. CPU and NNAPI delegates with onnxruntime_flutter perform best regarding AIT (14 and 16 ms). onnxruntime-react-native and CPU delegate setup is a close third with 17 ms AIT. Unsuprisingly, tflite_flutter has the highest AITs.

5.3 Semantic Image Segmentation

DeepLab v3+ FP32

iPhone 12 Mini. The CoreML delegate with onnxruntime_flutter, react-native-fast-tflite, and onnxruntime-react-native libraries have the best AITs (33, 38, and 38 ms). The worst-performing setup when it comes to AIT is tflite_flutter with the XNNPACK delegate.

iPhone 7. onnxruntime_flutter with XNNPACK delegate comes on top with 274 ms AIT. Again, the tflite_flutter performs the worst on AIT, having 1166 ms of AIT with the CPU delegate.

Samsung Galaxy A40. The best-performing setup on AIT, onnxruntime_flutter with CPU delegate, performs relatively badly, with 992 ms AIT. Last on the list is tflite_flutter with the same delegate; it has only 9494 ms of AIT.

Samsung Galaxy S20 FE. With the onnxruntime_flutter NNAPI delegate, the AIT is 163 ms, which is the best setup. NNAPI with onnxruntime-react-native is the second on AIT

with 180 ms. All of the tflite_flutter delegates perform badly, with 1588 ms AIT being the worst.

DeepLab v3+ UINT8

iPhone 12 Mini. react-native-fast-tflite with OpenGL and CoreML delegates are clearly on top with 58 and 61 ms AITs, respectively. Third, onnxruntime-react-native with a CPU delegate has 115 ms AIT. The tflite_flutter using the XNNPACK delegate performs the worst (338 ms AIT).

iPhone 7. Similarly to the iPhone 12 Mini, the react-native-fast-tflite comes clearly on top, having 331 ms of AIT with an OpenGL delegate. tflite_flutter using the XNNPACK delegate performs the worst with 861 ms AIT.

Samsung Galaxy A40. onnxruntime_flutter and CPU delegate have the best AIT of 409 ms. onnxruntime_flutter an onnxruntime-react-native with NNAPI delegates perform the worst on AIT, with 4179 and 4228 ms AITs.

Samsung Galaxy S20 FE. Both onnxruntime-flutter and onnxruntime-react-native with CPU delegate perform relatively well with 63 ms and 76 ms AITs, respectively. All of the tflite_flutter delegates perform badly, up to 845 ms AIT with the GPU delegate.

5.4 Overview of AITs

An overview of the AIT results for individual libraries and delegates is given in this Section. TFLite generally performs badly on all of the devices and models. All of the other libraries perform relatively well across the setups.

5.4.1 Libraries

When comparing the AIT of the different libraries, the results vary based on the criteria used. In Table 5.1, the best libraries are listed with the count of the occurrence of the library as the best performing library regarding AIT for each model and device pair. In this naive comparison, onnxruntime_flutter is clearly a winner. However, as the differences in AITs are very minimal in some cases, this result does not necessarily highlight the fact that other libraries are able to achieve similar results.

Library	Top 1 AIT count
onnxruntime__flutter	23
onnxruntime-react-native	7
react-native-fast-tflite	2
tflite__flutter	0

Table 5.1: The top 1 AIT count of a library when each model and device pair is considered.

Library	AIT within 20 ms	AIT within 40 ms	AIT within 80 ms
onnxruntime__flutter	30	30	30
onnxruntime-react-native	25	29	31
react-native-fast-tflite	13	15	15
tflite__flutter	0	2	7

Table 5.2: Count of a library being within 20 ms, 40 ms, and 80 ms from the best-performing setup for each model and device pair. There are a total of 32 model and device pairs. react-native-fast-tflite is only available for iOS, making the maximum count 16 for the library.

Table 5.2 shows the count of each library being within tresholds of 20 ms, 40 ms, and 80 ms from the best performing library for a device and model pair. Maximum count being 32 (4 devices $\times$ 8 models = 32). Tables 5.3 and 5.4 show the results for iOS and Android devices only.

5.4.2 Delegates

In Tables 5.5-5.8, a similar summary of results is given for delegates as was given for libraries.

Library	AIT within 20 ms	AIT within 40 ms	AIT within 80 ms
onnxruntime__flutter	14	14	14
onnxruntime-react-native	12	13	15
react-native-fast-tflite	13	15	15
tflite__flutter	0	2	5

Table 5.3: iOS devices only. There are 16 device and model pairs in total.

Library	AIT within 20 ms	AIT within 40 ms	AIT within 80 ms
onnxruntime_flutter	16	16	16
onnxruntime-react-native	13	16	16
react-native-fast-tflite	Not available	Not available	Not available
tflite_flutter	0	0	2

Table 5.4: Android devices only. There are 16 device and model pairs in total.

Delegate	Top 1 AIT count
CPU	11
CoreML	9
XNNPACK	5
OpenGL	4
NNAPI	3
Others	0

Table 5.5: All devices, Top 1 AIT count of a delegate for a device and model pair.

Delegate	AIT within 20 ms	AIT within 40 ms	AIT within 80 ms
CPU	21	25	29
CoreML	13	14	15
XNNPACK	15	18	21
NNAPI	9	10	10
OpenGL	10	13	14
Others	0	0	4

Table 5.6: Count of a delegate being within 20 ms, 40 ms, and 80 ms from best performing setup for each model and device pair. There are in total 32 model and device pairs.

Delegate	AIT within 20 ms	AIT within 40 ms	AIT within 80 ms
CPU	6	9	13
CoreML	13	14	15
XNNPACK	10	12	13
OpenGL	10	13	14
Others	0	0	4

Table 5.7: iOS devices only.

Delegate	**AIT within 20 ms**	**AIT within 40 ms**	**AIT within 80 ms**
CPU	15	16	16
XNNPACK	5	6	8
NNAPI	9	10	10
Others	0	0	0

Table 5.8: Android devices only.

6 Discussion

When looking into the accuracy results from the image classification models, there seems to be no significant variability. Meaning that all of the libraries are capable of providing accurate inferences. However, it should be noted that the image classification tests only contain 300 images, and therefore no definitive conclusions should be drawn from the libraries' accuracies. Hence, the discussion focuses only on inference speed when it comes to performance metrics. Moreover, library usability, other observations, and threats to validity are discussed.

6.1 Inference Speed

The test setup contains multiple dimensions: model, delegate, ML library, and the device used. In addition to comparing React Native and Flutter, the insights from the effects of each of these on the results are discussed in this Section.

6.1.1 React Native vs Flutter

What is interesting is that Flutter generally outperforms React Native when it comes to resource usage (Oliveira et al., 2023; Mahendra and Anggorojati, 2020; Biørn-Hansen et al., 2020; Huber and Demetz, 2019), but the same cannot be said for ML inference execution time. The results of this thesis show that Flutter is usually able to achieve slightly better inference times with a CPU delegate compared to React Native. But React Native is able to achieve very similar and even better results using both CPU and other delegates.

When looking into the results from Tables 5.1 and 5.2, the onnxruntime_flutter seems to be the most consistent library across different devices. However, it should be again noted that there is a major drawback to using Flutter for inference, which is that despite the fact that the ML libraries support Flutter Isolates, using them makes the inference speed significantly slower compared to running inference without them. The exact reasons behind this are difficult to interpret, but speculation about Isolates' slow performance in a GitHub issue (GitHub, 2024) might provide answers. Firstly, Isolates might pass on the

execution to a slow core of a device CPU, which explains why the execution on the CPU delegate gets slower. Or if there is no available core, it might execute on the same core as the UI, leading to a race condition between the UI rendering and ML inference. One possible reason is related to the I/O handling and message passing. It might be that this is slower when using an Isolate compared to the main thread. Regardless of the reason, not using Isolates is unoptimal UX-wise, as the UI thread can get blocked for a long time if the inference execution time is high.

The results from this thesis indicate that the JSI of React Native might provide a more efficient way to call native modules compared to the platform channels of Flutter, especially without blocking the UI. And it is also shown in (Oliveira et al., 2023) and (Biørn-Hansen et al., 2020) that React Native can achieve better performance than Flutter in certain tasks requiring native module calls. Hence, React Native might be a better fit for running on-device ML inference when optimized inference execution time with a non-blocking UI is necessary.

Table 6.1: Comparison of Flutter and React Native in Mobile Development and ML Inference using Open-Source libraries. Synchronously means that the inference call is made so that the main thread waits for the ML job to finish, blocking the UI for the execution time.

Metric	Flutter	React Native
Accuracy of ML Models	Good	Good
AIT - Synchronously	Good	Good
AIT - Asynchronously	Slower (due to Isolates)	Good

6.1.2 Library AIT Performance

The results of this thesis indicate that both Flutter and React Native are capable of efficiently calling native modules, at least synchronously. Therefore, both the role of the native module's implementation and the way of calling the module in a library are important, regardless of the cross-platform framework used.

Regarding performance, the tflite_flutter library performs generally the worst across all devices and models. The AITs achieved using the library are consistently slow compared to other libraries, regardless of the model and device used. By observing the source code of the library, the inefficiencies seem to stem at least partly from inefficient code. There seem to be unefficient list operations for input and output tensors called before running native code, possibly leading to unefficient CPU usage. Figure 6.1 illustrates

this overhead. The onnxruntime_flutter library, on the other hand, directly calls native methods when starting the inference. Additionally, on the Samsung A40, the use of GPU and NNAPI delegates using tflite_flutter results in significantly higher inference times, especially noticeable in complex models like SSMobileNet v2 and DeepLab v3+. Additionally, high standard deviations of AITs suggest performance instability, possibly due to delegate utilization inefficiencies in the library.

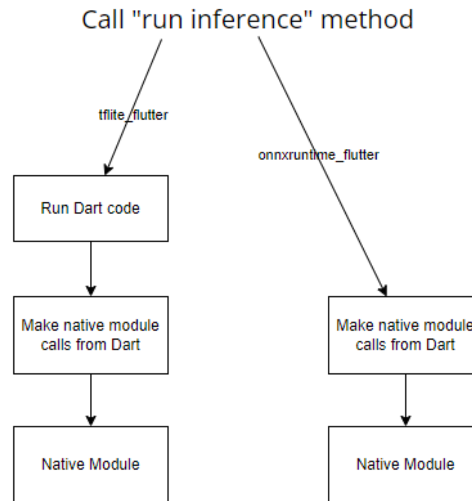

Figure 6.1: tflite_flutter (on the left) runs excessive Dart code before invoking Native Modules, partly explaining poor performance. onnxruntime_flutter directly invokes Native Modules.

The react-native-fast-tflite performs well on iOS devices; see Table 5.7. This is especially true for iPhone 12 Mini, as the library is able to achieve AIT within 20 ms from the best performing library with all of the 8 models on the device. It evenly competes with onnxruntime libraries, which highlights its effective use of iOS hardware optimizations. Much like onnxruntime_flutter, the library does not contain any unnecessary framework-specific (JS) code before delegating the inference for the Native Module.The major downside is that it does not support Android devices.

The library onnxruntime-react-native exhibits lower inference times on iOS devices when using optimized delegates like Core ML, indicating good integration with Apple's hardware optimizations. And as can be seen from Table 5.4 the library is able to provide relatively good AITs on the Android devices as well.

The library onnxruntime_flutter shows competitive performance across all of the device and model pairs, especially when it comes to Android devices. It has similar results to onnxruntime-react-native in terms of performance on iOS, showing strong performance

with CoreML but generally even higher and more consistent performance than its react-native counterpart. Table 5.2 shows that the library has the most consistent AITs accross devices when 20 ms treshold from top AIT is used, but onnxruntime-react-native and react-native-fast-tflite achieve similar consistency with the higher tresholds. Indicating that onnxruntime_flutter can generally achieve the lowest AITs, but both onnxruntime-react-native and react-native-fast-tflite are as viable options due to the low difference in AITs.

6.1.3 Library Performance Stability

Wu et al., 2019 underline the importance of stable execution times as it provides a consistent user experience. I will analyze this through standard deviations (STDs) of the inference times of test setups (device, library, model). It seems that better performing setups regarding AIT tend to have lower STDs as well.

Looking into the STDs of the different libraries, one can draw the following insights: The onnxruntime-react-native seems to provide moderate to high STDs, particularly with older devices like the Samsung Galaxy A40. For newer devices, the library shows relatively low STDs for most of the models. The heavier models tend to show bigger STDs.

Using onnxruntime_flutter, the STDs are moderate, but again higher with older devices. The simpler models tend to show less variability in the inference times.

The library tflite_flutter performs generally poorly in the AIT, so it is no surprise that the STDs are high as well. This is especially true for older devices.

The best-performing library regarding STDs is react-native-fast-tflite. The STDs are generally low across devices and models, but it should be taken into account that the results are only from iOS devices and the library only uses OpenGL and CoreML delegates, which are well optimized for iOS.

6.1.4 Delegate AIT Performance

NNAPI shows bad AIT on the Samsung Galaxy A40 and with more demanding models such as SSD-mobilenet v2 and DeepLab v3+ FP32 versions. It might be due to the fact that the GPU of the Galaxy A40 does not have shared memory, leading to slower computation on the GPU with more complex models. Or the NNAPI is trying to offload the work on the NPU but constantly fallsbacks on the CPU or GPU as the NPU is

not found on the device, leading to overhead in execution. This suggests that NNAPI's performance might be less predictable, at least for lower-end devices, which could be problematic for applications requiring consistent processing times. On the Galaxy S20 FE, NNAPI is generally able to achieve good AITs.

Across the runs, CoreML not only provides very good performance but also shows lower variability in inference times, making it reliable for iOS decices. CoreML especially works well on the iPhone 12 Mini, as it is constantly able to achieve almost optimal AIT. It especially competes well with the FP32 models. For example, with onnxruntime_flutter and the CoreML delegate, the MobileNetEdgeTPU FP32 model runs with 1 ms AIT, but the UINT8 version has 15 ms AIT. The Neural Engine only supports FP and needs to convert UINTs to FP before code execution, at least partly explaining the difference in performance. The fact that the iPhone 7, which does not contain a Neural Engine, does not have a similar trend when it comes to FP32 and UINT8 model AITs using CoreML delegate supports this explanation. CoreML works on par with OpenGL and XNNPACK on iPhone 7; see Table 5.3.

The GPU and Metal delegates, only available from tflite_flutter, generally show slower performance across devices, with high AITs especially noticeable for more complex models like DeepLab v3+. On the Samsung Galaxy A40, the GPU delegate frequently exceeds 500 ms AIT for these models. However, the tflite_flutter library generally performs badly, so no conclusions should be drawn for the delegates.

The CPU delegate generally provides consistent AITs across devices, showing stability across different libraries and models. On iOS devices, the CPU delegate typically shows higher AITs compared to Core ML, but the performance is still very similar to CoreML, especially on less complex models. On Android, the CPU delegate is every time except one within 20 ms of the optimal inference time with both of the Android devices on all of the models; see Table 5.8. The only exception is on the Samsung Galaxy S20 FE and DeepLab v3+ FP32 model, when the NNAPI delegate of onnxruntime_flutter is able to achieve 24 ms better AIT than any library with a CPU delegate. All in all, the CPU delegate seems to provide very consistent AITs on Android devices.

The XNNPACK delegate exhibits relatively good AITs on iOS devices; see Table 5.3. The results stem from the fact that the delegate performs relatively well especially on the iPhone 7. On other devices, the CPU delegate seems to provide more consistent AIT across the models.

OpenGL delegate, only available in the react-native-fast-tflite library, shows promising

performance on iOS devices. It seems to effectively leverage OpenGL's GPU acceleration, resulting in reasonably fast inference times. On the iPhone 12 Mini, the OpenGL delegate shows good AITs for models like MobileNet v2. This performance suggests that the OpenGL delegate can efficiently utilize GPU capabilities for lightweight models, offering an alternative to the CoreML delegate on iOS devices. Despite its strengths, the OpenGL delegate is not available on Android through react-native-fast-tflite, making it less usable. And the variability in performance seems to increase for more demanding models, where CoreML provides more stable performance. For example, using DeepLab v3+, the inference times increase considerably.

6.2 Other Observations

The tflite_flutter seems to be the most stable when developer experience is considered. However, the slow inference times across the devices make it almost unusable. The rest of the libraries seem to have some limitations or bugs that affect their usability. The react-native-fast-tflite only works for iOS devices. And the CoreML delegate does work for older devices, as TFLite only supports CoreML for iOS devices with an A12 SoC or higher (TensorFlow, 2024a).

The onnxruntime-react-native supports the major delegates per platform. However, on iOS, it does seem to have a bug when trying to load a local model. In onnxruntime_flutter, the CPU delegate throws an error when used on iOS devices. Additionally, when using the ONNX Runtime libraries, the output structure is an object with named keys derived from the model output names, which complicates the process of deploying new models. It was possible to not hardcode output processing by utilizing the libraries' tools, but it felt more time-consuming than when using the TFLite libraries, which provided the outputs as number arrays.

The variability of the best-performing setup for a model can be high when it comes to different devices. For example, the Galaxy A40 is able to achieve a minimum of 992 ms AIT with the DeepLab v3+ FP32 model, but the iPhone 12 Mini has a 33 ms AIT. However, using the UINT8 version of the model, the AIT with the Galaxy A40 drops but grows for the iPhone 12 Mini. This outlines the fact that different versions of models are needed if optimal inference time per device is necessary. And at least with complex models, inference performance varies significantly across devices.

6.2.1 Summary

If the Top 1 AIT count of a library, shown in Table 5.1, is considered, onnxruntime_flutter outperforms all the other libraries. But in a real-life use case, a developer must take into consideration other aspects as well. One of them is the slow inference time using Flutter Isolates compared to React Native. With React Native, one can achieve very similar inference times with onnxruntime-react-native and react-native-fast-tflite. The tradeoff that React Native offers slightly slower AITs but performs the inference without blocking the UI is most likely worth taking in real-world use cases.

As react-native-fast-tflite only works on iOS, the most viable option seems to be onnxruntime-react-native. Additionally, when looking into the AIT performance of the library, it can be recommended to be used as it provides nearly similar AITs across the devices. The most performant delegates across the libraries in general seem to be CPU for Android and CoreML for iOS. The results of onnxruntime-react-native are somewhat in line with the general results; see Tables 6.2 and 6.3. The only exception is that the CPU delegate on lower-end iOS devices (iPhone 7) seems to work very similarly to CoreML. Table 6.4 shows that the NNAPI delegate performs generally well on newer devices. This is promising, as offloading the inference away from the CPU might free up computation space for other tasks. So looking into the result, to provide good inference and accuracy in general across different devices, I would suggest using onnxruntime-react-native with:

- CPU delegate on Android devices without an NPU
- NNAPI on Android devices equipped with an NPU
- CoreML on all iOS devices.

It is important to note that these results do not fully generalize to other libraries.

6.3 Threats to Validity

Understanding potential issues with validity helps to know this study's limits and interpret its results correctly. As mentioned before, the accuracy of the image classification models is only calculated by using 300 input images, meaning that more variability might be noticed if using a comprehensive validation set.

onnxruntime_flutter (CPU delegate)			
Device	**AIT within 20 ms**	**AIT within 40 ms**	**AIT within 80 ms**
iPhone 12 Mini	4	5	7
iPhone 7	2	4	6
Samsung Galaxy S20 FE	7	8	8
Samsung Galaxy A40	5	8	8

Table 6.2: Count of onnxruntime_flutter with CPU delegate being within 20 ms, 40 ms, and 80 ms from best performing setup for each model on the devices. There are in total 8 models.

onnxruntime_flutter (CoreML delegate)			
Device	**AIT within 20 ms**	**AIT within 40 ms**	**AIT within 80 ms**
iPhone 12 Mini	6	7	7
iPhone 7	3	3	6

Table 6.3: Count of onnxruntime_flutter with CoreML delegate being within 20 ms, 40 ms, and 80 ms from best performing setup for each model on the devices. There are in total 8 models. CoreML is iOS specific, hence no Android devices shown.

onnxruntime_flutter (NNAPI delegate)			
Device	**AIT within 20 ms**	**AIT within 40 ms**	**AIT within 80 ms**
Samsung Galaxy S20 FE	6	7	7
Samsung Galaxy A40	2	3	3

Table 6.4: Count of onnxruntime_flutter with NNAPI delegate being within 20 ms, 40 ms, and 80 ms from best performing setup for each model on the devices. There are in total 8 models. NNAPI is Android specific, hence no iOS devices shown.

All of the devices are able to provide relatively fast inference speeds for all of the models, meaning that the models aren't necessarily very complex. Using even more complex models for the evaluations would possibly provide more comprehensive results. And as Android manufacturers tend to have very divergent hardware, it would be beneficial to test devices from other manufacturers as well.

Regarding the test system, it's important to mention that the tests for each of the models are run one by one. In the real world, a mobile application might need multiple ML models to be run at the same time. Or the application could perform other resource-intensive tasks at the same time. The tests made do not directly consider how a situation like this would affect the results. One could speculate that because Flutter tends to use less resources in general than React Native, it could handle a situation like this better. But as it is possible to offload the inference on dedicated hardware on both iOS and Android, the differences should not be too significant if there is only one ML task running at the same time.

Lastly, it should be noted that the author did not have any previous Flutter experience before implementing the Flutter application for the performance evaluation. A more experienced developer could be able to optimize the test system for Flutter.

7 Conclusions

In this thesis, I presented the results of running ML inference on four different real devices using both React Native and Flutter. The results were gathered by creating a custom test bed, which consists of a data API, and React Native and Flutter applications, which utilize open-source libraries to run the inference.

With both React Native and Flutter, it is possible to achieve efficient ML inference accuracy when using open-source libraries. The most relevant technical factors contributing to the ML inference speed using cross-platform frameworks seem to be in the JS/Dart ML libraries, TFLite/ONNX implementations used in these libraries, the delegate in use, and the device. React Native's JavaScript Interface (JSI) may handle native module calls more efficiently than Flutter's platform channels; at least the inference slowdown using Flutter Isolates currently makes React Native potentially a better choice for on-device machine learning inference when optimized execution time is essential.

Despite using a cross-platform framework with efficient ML libraries, the variability of devices' hardware seems to still turn out to be a problem if optimal inference speed across all devices is necessary. One library might provide optimal inference for device A and model X, but it does not provide similar results for device B for the same model. However, if it is possible to take a small tradeoff from the inference speed for some devices, one can achieve generally good inference speed across devices with a single ML library. But developers should still use different delegates for inference depending on the model, the OS, and the hardware of a device.

In the future, the ML inference speed and accuracy using Flutter and React Native should be compared to native iOS and Android applications. This would give more insight into the overheads that the frameworks introduce when running ML inference. Additionally, to provide the best possible developer experience, ML libraries for both React Native and Flutter are needed that provide optimal inference for various different devices out of the box.

Bibliography

Android Developers (2024). *Neural Networks API*. https://developer.android.com/ndk/guides/neuralnetworks. Accessed: 2024-03-12.

Apple (2024a). *Core ML*. https://developer.apple.com/documentation/coreml. Accessed: 2024-03-12.

– (2024b). *Metal Overview*. https://developer.apple.com/metal/. Accessed: 2024-03-12.

AppStudio (2024). *Top 10 Cross-Platform App Frameworks*. https://www.appstudio.ca/blog/top-10-cross-platform-app-frameworks/. Accessed: 2024-02-02.

Biørn-Hansen, A., Rieger, C., Grønli, T.-M., Majchrzak, T. A., and Ghinea, G. (2020). "An empirical investigation of performance overhead in cross-platform mobile development frameworks". In: *Empirical Software Engineering* 25, pp. 2997–3040. DOI: 10.1007/s10664-020-09827-6.

Coox Tech (2023). *Deep Dive into React Native's New Architecture*. https://medium.com/coox-tech/deep-dive-into-react-natives-new-architecture-fb67ae615ccd. Accessed: 2024-02-05.

Cui, Y., Sun, Y., Luo, J., Huang, Y., Zhou, Y., and Li, X. (2022). "MMPD: A Novel Malicious PDF File Detector for Mobile Robots". In: *IEEE Sensors Journal* 22.18, pp. 17583–17592. DOI: 10.1109/JSEN.2020.3029083.

Dzombeta, S., Stantchev, V., Colomo-Palacios, R., Brandis, K., and Haufe, K. (2014). "Governance of Cloud Computing Services for the Life Sciences". In: *IT Professional* 16.4, pp. 30–37. DOI: 10.1109/MITP.2014.52.

Elhassan, G. E., Yasser, I., Faizal, M. O., and Zia, H. (2023). "optimizing Furniture Assembly: A CNN-based Mobile Application for Guided Assembly and Verification". In: *2023 9th International Conference on Optimization and Applications (ICOA)*, pp. 1–6. DOI: 10.1109/ICOA58279.2023.10308815.

Expo (2024). *Expo Documentation*. https://docs.expo.dev/. Accessed: 2024-03-12.

Flutter (2024). *Writing Custom Platform-Specific Code*. https://docs.flutter.dev/platform-integration/platform-channels. Accessed: 2024-03-12.

Fradkov, A. L. (2020). "Early History of Machine Learning". In: *IFAC-PapersOnLine* 53.2. 21st IFAC World Congress, pp. 1385–1390. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2020.12.1888.

GitHub (2024). *Widget rendering problem on iOS*. https://github.com/flutter/flutter/issues/145450. Accessed: 2024-05-20.

Github (2024). *TensorFlow.js for React Native*. https://github.com/tensorflow/tfjs/tree/master/tfjs-react-native. Accessed: 2024-02-21.

Google (2024a). *Flutter - Build apps for any screen*. https://flutter.dev/. Accessed: 2024-02-08.

– (2024b). *Impeller rendering engine | Flutter*. https://docs.flutter.dev/perf/impeller. Accessed: 2024-02-08.

– (2024c). *XNNPACK: High-performance floating-point neural network inference operators for small devices*. https://github.com/google/XNNPACK. Accessed: 2024-04-08.

Hastie, T., Tibshirani, R., Friedman, J., Hastie, T., Tibshirani, R., and Friedman, J. (2009). "Overview of supervised learning". In: *The elements of statistical learning: Data mining, inference, and prediction*, pp. 9–41.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861*. DOI: 10.48550/arXiv.1704.04861.

Huang, Y., Qiao, X., Tang, J., Ren, P., Liu, L., Pu, C., and Chen, J. (2023). "An Integrated Cloud-Edge-Device Adaptive Deep Learning Service for Cross-Platform Web". In: *IEEE Transactions on Mobile Computing* 22.4, pp. 1950–1967. DOI: 10.1109/TMC.2021.3122279.

Huber, S. and Demetz, L. (2019). "Performance Analysis of Mobile Cross-platform Development Approaches based on Typical UI Interactions." In: *ICSOFT*, pp. 40–48.

Ignatov, A., Timofte, R., Chou, W., Wang, K., Wu, M., Hartley, T., and Van Gool, L. (Sept. 2018). "AI Benchmark: Running Deep Neural Networks on Android Smartphones". In: *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*.

Janapa Reddi, V., Kanter, D., Mattson, P., Duke, J., Nguyen, T., Chukka, R., Shiring, K., Tan, K.-S., Charlebois, M., Chou, W., et al. (2022). "MLPerf Mobile Inference Benchmark: An Industry-Standard Open-Source Machine Learning Benchmark for On-Device AI". In: *Proceedings of Machine Learning and Systems*. Ed. by D. Marculescu, Y. Chi, and C. Wu. Vol. 4, pp. 352–369. URL: https://proceedings.mlsys.org/paper_files/paper/2022/file/a2b2702ea7e682c5ea2c20e8f71efb0c-Paper.pdf.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4, pp. 237–285.

Keras (2024). *Keras Documentation: MobileNet*. https://keras.io/api/applications/mobilenet/. Accessed: 2024-04-08.

Kotlin (2024). *Kotlin Multiplatform*. https://kotlinlang.org/docs/multiplatform.html. Accessed: 2024-02-02.

Kumar, V. R., Shrishti, V., and Sridhar, P. (2022). "Corn Plant Disease Classification using a combination of Machine Learning and Deep Learning". In: *2022 International Conference on Futuristic Technologies (INCOFT)*. IEEE, pp. 1–4. DOI: 10.1109/INCOFT55651.2022.10094326.

Lee, J., Chirkov, N., Ignasheva, E., Pisarchyk, Y., Shieh, M., Riccardi, F., Sarokin, R., Kulik, A., and Grundmann, M. (2019). "On-device neural net inference with mobile gpus". In: *arXiv preprint arXiv:1907.01989*. DOI: 10.48550/arXiv.1907.01989.

Lei, Y., Wang, Y., Caslin, T., Wisowaty, A., Zhu, X., Khamis, M., and Ye, J. (2023). "DynamicRead: Exploring Robust Gaze Interaction Methods for Reading on Handheld Mobile Devices under Dynamic Conditions". In: *Proceedings of the ACM on Human-Computer Interaction* 7.ETRA, pp. 1–17. DOI: 10.1145/3591127.

Mahendra, M. and Anggorojati, B. (2020). "Evaluating the performance of android based cross-platform app development frameworks". In: *Proceedings of the 6th International Conference on Communication and Information Processing*, pp. 32–37. DOI: 10.1145/3442555.3442561.

Mahesh, B. (2020). "Machine learning algorithms -A review". In: *International Journal of Science and Research (IJSR).[Internet]* 9.1, pp. 381–386.

Maitriboriruks, R., Piya-Aromrat, P., and Limpiyakorn, Y. (2022). "Smart Conveyor Belt Sushi Bill Payment with a Mobile Shot". In: *Proceedings of the 4th International Conference on Information Technology and Computer Communications*, pp. 1–8. DOI: 10.1145/3548636.3548637.

Meta Platforms, Inc. (2024). *React Native*. https://reactnative.dev/. Accessed: 2024-02-05.

Microsoft (2024). *Xamarin | .NET*. https://dotnet.microsoft.com/en-us/apps/xamarin. Accessed: 2024-02-02.

Microsoft and contributors, O. R. (2024). *ONNX Runtime for React Native*. https://github.com/microsoft/onnxruntime/tree/main/js/react_native. Accessed: 2024-02-21.

MLCommons (2024a). *MLCommons*. https://github.com/mlcommons. Accessed 2024-02-27.

MLCommons (2024b). *MLCommons Mobile Models*. https://github.com/mlcommons/mobile_models. Accessed: 2023-03-13.

Mohzary, M., Almalki, K. J., Choi, B.-Y., and Song, S. (2023). "MobiDeep: Mobile DeepFake Detection through Machine Learning-based Corneal-Specular Backscattering". In: *2023 IEEE 20th Consumer Communications & Networking Conference (CCNC)*, pp. 1104–1109. DOI: 10.1109/CCNC51644.2023.10059841.

Mozilla Developer Network (2024). *WebGL API*. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. Accessed: 2024-03-13.

Oliveira, W., Moraes, B., Castor, F., and Fernandes, J. P. (2023). "Analyzing the Resource Usage Overhead of Mobile App Development Frameworks". In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pp. 152–161. DOI: 10.1145/3593434.3593487.

ONNX Runtime (2024). *ONNX Runtime*. https://onnxruntime.ai/. Accessed: 2024-03-12.

Open Neural Network Exchange (ONNX) (2024). *TensorFlow-ONNX*. https://github.com/onnx/tensorflow-onnx. Accessed: 2024-04-08.

Palmieri, M., Singh, I., and Cicchetti, A. (2012). "Comparison of cross-platform mobile development tools". In: *2012 16th International Conference on Intelligence in Next Generation Networks*, pp. 179–186. DOI: 10.1109/ICIN.2012.6376023.

Redwerk (2024). *Best Cross-Platform Mobile Development Tools*. https://redwerk.com/blog/best-cross-platform-mobile-development-tools/. Accessed: 2024-02-02.

Roopashree, S. and Anitha, J. (2021). "DeepHerb: A Vision Based System for Medicinal Plants Using Xception Features". In: *IEEE Access* 9, pp. 135927–135941. DOI: 10.1109/ACCESS.2021.3116207.

Rousavy, M. (2024). *React Native Fast TFLite*. https://github.com/mrousavy/react-native-fast-tflite. Accessed: 2024-02-21.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 4510–4520.

ScaleupAlly (2024). *Top 12 Popular Cross-Platform App Development Frameworks 2024*. https://scaleupally.io/blog/cross-platform-app-development-frameworks/. Accessed: 2024-02-02.

Shao, J. and Zhang, J. (2020). "Communication-Computation Trade-off in Resource-Constrained Edge Inference". In: *IEEE Communications Magazine* 58.12, pp. 20–26. DOI: 10.1109/MCOM.001.2000373.

Sheppard, D. and Sheppard, D. (2017). *Beginning progressive web app development*. Springer.

Sipola, T., Alatalo, J., Kokkonen, T., and Rantonen, M. (2022). "Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software". In: *2022 31st Conference of Open Innovations Association (FRUCT)*, pp. 320–331. DOI: 10.23919/FRUCT54823.2022.9770931.

Stats, S. G. (2024). *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide/. Accessed: 2024-01-25.

Sun, T. R. (2020). "FaceAUG: A Cross-Platform Application for Real-Time Face Augmentation in Web Browser". In: *2020 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pp. 290–293. DOI: 10.1109/AIVR50618.2020.00058.

Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., and Zhou, D. (2020). "Mobilebert: a compact task-agnostic bert for resource-limited devices". In: *arXiv preprint arXiv:2004.02984*. DOI: 10.48550/arXiv.2004.02984.

TensorFlow (2024a). *GPU acceleration delegate with Interpreter API*. https://www.tensorflow.org/lite/android/delegates/gpu. Accessed: 2024-03-12.

– (2024b). *TensorFlow Flutter TFLite*. https://github.com/tensorflow/flutter-tflite. Accessed: 2024-02-21.

– (2024c). *TensorFlow Lite*. https://www.tensorflow.org/lite. Accessed: 2024-02-21.

– (2024d). *TensorFlow Lite Delegates*. https://www.tensorflow.org/lite/performance/delegates. Accessed: 2024-03-15.

The World Bank (2024). *Mobile Cellular Subscriptions*. https://data.worldbank.org/indicator/IT.CEL.SETS. Accessed: 2024-01-25.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971*. DOI: 10.48550/arXiv.2302.13971.

Vanhoucke, V., Senior, A., Mao, M. Z., et al. (2011). "Improving the speed of neural networks on CPUs". In: *Proc. deep learning and unsupervised feature learning NIPS workshop*. Vol. 1. 2011, p. 4.

Wijesinghe, W., Amarasinghe, A., Bandara, T., Gamage, A. I., and Ganegoda, D. (2021). "VOYAGER–Smart Travel Guidance Cross Platform Mobile Application". In: *2021 3rd International Conference on Advancements in Computing (ICAC)*. IEEE, pp. 163–168. DOI: 10.1109/ICAC54203.2021.9671136.

Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., et al. (2019). "Machine learning at facebook: Understanding

inference at the edge". In: *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, pp. 331–344. DOI: 10.1109/HPCA.2019.00048.

Xanthopoulos, S. and Xinogalos, S. (2013). "A comparative analysis of cross-platform development approaches for mobile applications". In: *Proceedings of the 6th Balkan Conference in Informatics*, pp. 213–220. DOI: 10.1145/2490257.2490292.

Xu, D., Li, T., Li, Y., Su, X., Tarkoma, S., Jiang, T., Crowcroft, J., and Hui, P. (2020). "Edge intelligence: Architectures, challenges, and applications". In: *arXiv preprint arXiv:2003.12172*. DOI: 10.48550/arXiv.2003.12172.

Zhu, M. and Gupta, S. (2017). "To prune, or not to prune: exploring the efficacy of pruning for model compression". In: *arXiv preprint arXiv:1710.01878*. DOI: 10.48550/arXiv.1710.01878.

Appendix A All Test Results

Model	Device	Library	Delegate	AIT	STD	Accuracy
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime-react-native	cpu	166.640000	27.050000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	251.580000	24.640000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime-react-native	nnapi	530.950000	10.230000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime_flutter	cpu	158.140000	16.790000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	345.490000	159.900000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	onnxruntime_flutter	nnapi	482.490000	21.980000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	tflite_flutter	gpu	447.520000	27.100000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	tflite_flutter	nnapi	569.680000	27.190000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	tflite_flutter	cpu	577.550000	36.680000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy A40	tflite_flutter	xnnpack	579.420000	46.990000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	34.240000	2.500000	0.636700
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	44.110000	4.020000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	86.730000	2.190000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	24.560000	1.680000	0.636700
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	29.200000	5.910000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	70.990000	13.680000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	tflite_flutter	nnapi	217.760000	19.990000	0.636700
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	tflite_flutter	gpu	262.300000	19.990000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	265.690000	14.810000	0.633300
mobilenet_edgetpu_float32	Samsung Galaxy S20 FE	tflite_flutter	cpu	280.630000	22.740000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	onnxruntime-react-native	core_ml	1.280000	0.450000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	onnxruntime-react-native	xnnpack	32.820000	0.970000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	onnxruntime-react-native	cpu	34.810000	0.880000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	onnxruntime_flutter	core_ml	0.970000	0.210000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	onnxruntime_flutter	xnnpack	34.860000	1.220000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	react-native-fast-tflite	core_ml	1.010000	0.110000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	react-native-fast-tflite	opengl	25.100000	0.360000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	tflite_flutter	core_ml	98.380000	9.390000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	tflite_flutter	metal	111.700000	9.070000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	tflite_flutter	cpu	115.580000	8.100000	0.633300
mobilenet_edgetpu_float32	iPhone 12 mini	tflite_flutter	xnnpack	120.960000	8.240000	0.633300
mobilenet_edgetpu_float32	iPhone 7	onnxruntime-react-native	core_ml	26.010000	25.650000	0.633300
mobilenet_edgetpu_float32	iPhone 7	onnxruntime-react-native	xnnpack	53.050000	5.230000	0.633300
mobilenet_edgetpu_float32	iPhone 7	onnxruntime-react-native	cpu	66.980000	2.840000	0.633300
mobilenet_edgetpu_float32	iPhone 7	onnxruntime_flutter	core_ml	22.310000	4.520000	0.633300
mobilenet_edgetpu_float32	iPhone 7	onnxruntime_flutter	xnnpack	43.400000	0.640000	0.633300
mobilenet_edgetpu_float32	iPhone 7	react-native-fast-tflite	opengl	49.820000	1.210000	0.633300
mobilenet_edgetpu_float32	iPhone 7	tflite_flutter	metal	206.160000	45.680000	0.633300
mobilenet_edgetpu_float32	iPhone 7	tflite_flutter	cpu	294.980000	51.930000	0.633300
mobilenet_edgetpu_float32	iPhone 7	tflite_flutter	xnnpack	315.580000	42.090000	0.633300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime-react-native	cpu	70.910000	5.140000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime-react-native	nnapi	77.430000	5.850000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	176.340000	19.810000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime_flutter	cpu	64.480000	4.690000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime_flutter	nnapi	70.900000	4.310000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	164.010000	0.120000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy A40	tflite_flutter	gpu	274.750000	19.430000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy A40	tflite_flutter	cpu	295.000000	22.250000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy A40	tflite_flutter	xnnpack	295.700000	22.590000	0.686700

mobilenet_edgetpu_uint8	Samsung Galaxy A40	tflite_flutter	nnapi	295.940000	23.170000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	11.900000	2.040000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	14.600000	1.980000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	30.440000	2.710000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	9.610000	1.270000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	13.090000	1.220000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	26.140000	0.710000	0.683300
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	tflite_flutter	nnapi	88.550000	4.590000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	tflite_flutter	cpu	112.910000	4.930000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	113.100000	4.540000	0.686700
mobilenet_edgetpu_uint8	Samsung Galaxy S20 FE	tflite_flutter	gpu	156.580000	3.360000	0.680000
mobilenet_edgetpu_uint8	iPhone 12 mini	onnxruntime-react-native	xnnpack	8.010000	0.090000	0.683300
mobilenet_edgetpu_uint8	iPhone 12 mini	onnxruntime-react-native	cpu	16.790000	0.680000	0.683300
mobilenet_edgetpu_uint8	iPhone 12 mini	onnxruntime-react-native	core_ml	24.290000	0.570000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	onnxruntime_flutter	xnnpack	7.000000	0.030000	0.683300
mobilenet_edgetpu_uint8	iPhone 12 mini	onnxruntime_flutter	core_ml	15.320000	0.510000	0.683300
mobilenet_edgetpu_uint8	iPhone 12 mini	react-native-fast-tflite	opengl	8.010000	0.110000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	react-native-fast-tflite	core_ml	8.010000	0.140000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	tflite_flutter	cpu	45.640000	3.870000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	tflite_flutter	xnnpack	45.910000	3.870000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	tflite_flutter	core_ml	46.200000	4.170000	0.686700
mobilenet_edgetpu_uint8	iPhone 12 mini	tflite_flutter	metal	60.530000	4.010000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	onnxruntime-react-native	xnnpack	61.940000	4.990000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	onnxruntime-react-native	cpu	72.170000	4.090000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	onnxruntime-react-native	core_ml	102.560000	3.420000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	onnxruntime_flutter	xnnpack	49.720000	0.530000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	onnxruntime_flutter	core_ml	59.290000	0.500000	0.683300
mobilenet_edgetpu_uint8	iPhone 7	react-native-fast-tflite	opengl	40.120000	0.420000	0.686700
mobilenet_edgetpu_uint8	iPhone 7	tflite_flutter	metal	109.710000	7.350000	0.680000
mobilenet_edgetpu_uint8	iPhone 7	tflite_flutter	cpu	110.580000	8.820000	0.686700
mobilenet_edgetpu_uint8	iPhone 7	tflite_flutter	xnnpack	110.670000	8.310000	0.686700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	119.970000	30.550000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime-react-native	nnapi	121.870000	28.270000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime-react-native	cpu	123.280000	29.460000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	98.080000	6.970000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime_flutter	nnapi	100.100000	11.800000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	onnxruntime_flutter	cpu	100.380000	12.510000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	tflite_flutter	gpu	387.530000	27.010000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	tflite_flutter	xnnpack	408.490000	26.570000	0.650000
mobilenetv2_float32	Samsung Galaxy A40	tflite_flutter	cpu	418.630000	27.490000	0.646700
mobilenetv2_float32	Samsung Galaxy A40	tflite_flutter	nnapi	419.240000	27.190000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	25.590000	4.860000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	25.720000	5.050000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	25.900000	5.160000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	28.840000	0.780000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	29.540000	0.920000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	29.750000	1.840000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	tflite_flutter	nnapi	162.610000	21.400000	-
mobilenetv2_float32	Samsung Galaxy S20 FE	tflite_flutter	gpu	165.910000	20.450000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	tflite_flutter	cpu	185.290000	39.020000	0.646700
mobilenetv2_float32	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	212.310000	18.990000	0.650000
mobilenetv2_float32	iPhone 12 mini	onnxruntime-react-native	xnnpack	17.170000	0.600000	0.646700
mobilenetv2_float32	iPhone 12 mini	onnxruntime-react-native	cpu	18.620000	0.710000	0.646700
mobilenetv2_float32	iPhone 12 mini	onnxruntime-react-native	core_ml	40.430000	4.350000	0.646700
mobilenetv2_float32	iPhone 12 mini	onnxruntime_flutter	xnnpack	16.970000	0.500000	0.646700

mobilenetv2_float32	iPhone 12 mini	onnxruntime_flutter	core_ml	18.500000	0.980000	0.646700
mobilenetv2_float32	iPhone 12 mini	react-native-fast-tflite	core_ml	7.010000	0.130000	0.636700
mobilenetv2_float32	iPhone 12 mini	react-native-fast-tflite	opengl	7.190000	1.410000	0.636700
mobilenetv2_float32	iPhone 12 mini	tflite_flutter	metal	101.000000	7.850000	0.646700
mobilenetv2_float32	iPhone 12 mini	tflite_flutter	core_ml	101.190000	8.330000	0.646700
mobilenetv2_float32	iPhone 12 mini	tflite_flutter	xnnpack	103.050000	9.400000	0.650000
mobilenetv2_float32	iPhone 12 mini	tflite_flutter	cpu	103.520000	9.040000	0.646700
mobilenetv2_float32	iPhone 7	onnxruntime-react-native	xnnpack	27.730000	1.120000	0.646700
mobilenetv2_float32	iPhone 7	onnxruntime-react-native	cpu	29.710000	1.080000	0.646700
mobilenetv2_float32	iPhone 7	onnxruntime-react-native	core_ml	113.170000	15.630000	0.643300
mobilenetv2_float32	iPhone 7	onnxruntime_flutter	xnnpack	25.490000	0.700000	0.646700
mobilenetv2_float32	iPhone 7	onnxruntime_flutter	core_ml	27.270000	0.640000	0.646700
mobilenetv2_float32	iPhone 7	react-native-fast-tflite	opengl	22.720000	1.000000	0.636700
mobilenetv2_float32	iPhone 7	tflite_flutter	metal	169.100000	12.350000	0.646700
mobilenetv2_float32	iPhone 7	tflite_flutter	xnnpack	170.600000	9.860000	0.650000
mobilenetv2_float32	iPhone 7	tflite_flutter	cpu	171.600000	9.970000	0.646700
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime-react-native	nnapi	41.480000	4.740000	0.633300
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime-react-native	cpu	42.180000	5.740000	0.633300
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	71.480000	25.720000	0.633300
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime_flutter	cpu	33.500000	6.420000	0.626700
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime_flutter	nnapi	34.440000	5.070000	0.626700
mobilenetv2_uint8	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	58.570000	9.480000	0.626700
mobilenetv2_uint8	Samsung Galaxy A40	tflite_flutter	xnnpack	207.110000	19.860000	0.626700
mobilenetv2_uint8	Samsung Galaxy A40	tflite_flutter	gpu	223.860000	22.310000	0.640000
mobilenetv2_uint8	Samsung Galaxy A40	tflite_flutter	cpu	226.110000	22.910000	0.633300
mobilenetv2_uint8	Samsung Galaxy A40	tflite_flutter	nnapi	226.210000	22.020000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	8.820000	2.460000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	9.150000	1.990000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	18.620000	2.090000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	8.280000	0.920000	0.626700
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	9.130000	2.210000	0.626700
mobilenetv2_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	15.450000	0.710000	0.626700
mobilenetv2_uint8	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	59.330000	3.740000	0.626700
mobilenetv2_uint8	Samsung Galaxy S20 FE	tflite_flutter	nnapi	59.840000	6.280000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	tflite_flutter	cpu	62.410000	3.550000	0.633300
mobilenetv2_uint8	Samsung Galaxy S20 FE	tflite_flutter	gpu	94.300000	4.450000	0.623300
mobilenetv2_uint8	iPhone 12 mini	onnxruntime-react-native	xnnpack	4.250000	0.690000	0.633300
mobilenetv2_uint8	iPhone 12 mini	onnxruntime-react-native	cpu	5.040000	0.240000	0.633300
mobilenetv2_uint8	iPhone 12 mini	onnxruntime-react-native	core_ml	9.820000	0.630000	0.640000
mobilenetv2_uint8	iPhone 12 mini	onnxruntime_flutter	core_ml	4.010000	0.130000	0.626700
mobilenetv2_uint8	iPhone 12 mini	onnxruntime_flutter	xnnpack	4.020000	0.330000	0.626700
mobilenetv2_uint8	iPhone 12 mini	react-native-fast-tflite	opengl	6.010000	0.080000	0.633300
mobilenetv2_uint8	iPhone 12 mini	react-native-fast-tflite	core_ml	6.010000	0.140000	0.633300
mobilenetv2_uint8	iPhone 12 mini	tflite_flutter	xnnpack	40.810000	3.660000	0.626700
mobilenetv2_uint8	iPhone 12 mini	tflite_flutter	core_ml	42.740000	3.490000	0.633300
mobilenetv2_uint8	iPhone 12 mini	tflite_flutter	cpu	42.960000	3.610000	0.633300
mobilenetv2_uint8	iPhone 12 mini	tflite_flutter	metal	48.810000	3.640000	0.633300
mobilenetv2_uint8	iPhone 7	onnxruntime-react-native	cpu	18.200000	1.060000	0.633300
mobilenetv2_uint8	iPhone 7	onnxruntime-react-native	xnnpack	21.230000	0.900000	0.633300
mobilenetv2_uint8	iPhone 7	onnxruntime-react-native	core_ml	34.890000	2.430000	0.643300
mobilenetv2_uint8	iPhone 7	onnxruntime_flutter	core_ml	16.100000	0.330000	0.626700
mobilenetv2_uint8	iPhone 7	onnxruntime_flutter	xnnpack	17.930000	0.610000	0.626700
mobilenetv2_uint8	iPhone 7	react-native-fast-tflite	opengl	22.720000	0.610000	0.633300
mobilenetv2_uint8	iPhone 7	tflite_flutter	xnnpack	86.450000	6.550000	0.626700
mobilenetv2_uint8	iPhone 7	tflite_flutter	cpu	91.400000	6.260000	0.633300

mobilenetv2_uint8	iPhone 7	tflite_flutter	metal	93.000000	5.510000	0.633300
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime-react-native	cpu	368.520000	34.660000	-
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	472.920000	25.260000	-
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime-react-native	nnapi	1164.260000	16.730000	-
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime_flutter	cpu	359.090000	31.400000	-
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	489.440000	3.700000	-
ssd_mobilenet_float32	Samsung Galaxy A40	onnxruntime_flutter	nnapi	1095.400000	23.030000	-
ssd_mobilenet_float32	Samsung Galaxy A40	tflite_flutter	gpu	913.510000	53.750000	-
ssd_mobilenet_float32	Samsung Galaxy A40	tflite_flutter	xnnpack	1179.850000	55.020000	-
ssd_mobilenet_float32	Samsung Galaxy A40	tflite_flutter	nnapi	1190.700000	86.120000	-
ssd_mobilenet_float32	Samsung Galaxy A40	tflite_flutter	cpu	1273.130000	54.620000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	78.300000	16.180000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	187.960000	4.350000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	320.360000	5.090000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	61.480000	4.220000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	142.230000	32.440000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	299.330000	4.150000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	tflite_flutter	gpu	421.150000	58.320000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	tflite_flutter	cpu	494.990000	69.710000	-
ssd_mobilenet_float32	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	512.500000	18.490000	-
ssd_mobilenet_float32	iPhone 12 mini	onnxruntime-react-native	core_ml	5.190000	0.590000	-
ssd_mobilenet_float32	iPhone 12 mini	onnxruntime-react-native	cpu	66.450000	1.340000	-
ssd_mobilenet_float32	iPhone 12 mini	onnxruntime-react-native	xnnpack	68.770000	1.580000	-
ssd_mobilenet_float32	iPhone 12 mini	onnxruntime_flutter	core_ml	4.220000	0.520000	-
ssd_mobilenet_float32	iPhone 12 mini	onnxruntime_flutter	xnnpack	114.600000	82.710000	-
ssd_mobilenet_float32	iPhone 12 mini	react-native-fast-tflite	core_ml	4.130000	0.350000	-
ssd_mobilenet_float32	iPhone 12 mini	react-native-fast-tflite	opengl	59.950000	2.040000	-
ssd_mobilenet_float32	iPhone 12 mini	tflite_flutter	core_ml	230.710000	76.110000	-
ssd_mobilenet_float32	iPhone 12 mini	tflite_flutter	metal	254.440000	88.960000	-
ssd_mobilenet_float32	iPhone 12 mini	tflite_flutter	cpu	306.780000	96.820000	-
ssd_mobilenet_float32	iPhone 12 mini	tflite_flutter	xnnpack	409.770000	183.680000	-
ssd_mobilenet_float32	iPhone 7	onnxruntime-react-native	core_ml	86.420000	22.790000	-
ssd_mobilenet_float32	iPhone 7	onnxruntime-react-native	xnnpack	125.220000	8.870000	-
ssd_mobilenet_float32	iPhone 7	onnxruntime-react-native	cpu	154.950000	9.490000	-
ssd_mobilenet_float32	iPhone 7	onnxruntime_flutter	xnnpack	88.240000	1.590000	-
ssd_mobilenet_float32	iPhone 7	onnxruntime_flutter	core_ml	92.860000	23.010000	-
ssd_mobilenet_float32	iPhone 7	react-native-fast-tflite	opengl	123.250000	12.840000	-
ssd_mobilenet_float32	iPhone 7	tflite_flutter	xnnpack	360.480000	22.690000	-
ssd_mobilenet_float32	iPhone 7	tflite_flutter	metal	380.010000	83.970000	-
ssd_mobilenet_float32	iPhone 7	tflite_flutter	cpu	520.880000	88.130000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime-react-native	cpu	78.200000	6.200000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	162.570000	22.080000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime-react-native	nnapi	970.610000	21.110000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime_flutter	cpu	70.350000	5.250000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	148.710000	2.970000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	onnxruntime_flutter	nnapi	941.370000	19.790000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	tflite_flutter	gpu	395.310000	22.490000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	tflite_flutter	nnapi	419.420000	20.840000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	tflite_flutter	xnnpack	420.300000	20.430000	-
ssd_mobilenet_uint8	Samsung Galaxy A40	tflite_flutter	cpu	420.460000	20.960000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	17.090000	4.510000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	31.650000	3.950000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	37.320000	2.630000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	13.790000	1.680000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	16.220000	3.020000	-

ssd_mobilenet_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	33.160000	0.380000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	tflite_flutter	nnapi	156.250000	8.180000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	tflite_flutter	cpu	187.720000	8.400000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	187.750000	8.140000	-
ssd_mobilenet_uint8	Samsung Galaxy S20 FE	tflite_flutter	gpu	234.600000	12.670000	-
ssd_mobilenet_uint8	iPhone 12 mini	onnxruntime-react-native	xnnpack	9.580000	0.510000	-
ssd_mobilenet_uint8	iPhone 12 mini	onnxruntime-react-native	cpu	16.230000	0.530000	-
ssd_mobilenet_uint8	iPhone 12 mini	onnxruntime-react-native	core_ml	21.550000	0.620000	-
ssd_mobilenet_uint8	iPhone 12 mini	onnxruntime_flutter	xnnpack	8.970000	0.200000	-
ssd_mobilenet_uint8	iPhone 12 mini	onnxruntime_flutter	core_ml	20.010000	0.510000	-
ssd_mobilenet_uint8	iPhone 12 mini	react-native-fast-tflite	opengl	9.130000	0.340000	-
ssd_mobilenet_uint8	iPhone 12 mini	react-native-fast-tflite	core_ml	9.130000	0.350000	-
ssd_mobilenet_uint8	iPhone 12 mini	tflite_flutter	core_ml	78.190000	5.400000	-
ssd_mobilenet_uint8	iPhone 12 mini	tflite_flutter	cpu	79.420000	5.850000	-
ssd_mobilenet_uint8	iPhone 12 mini	tflite_flutter	xnnpack	80.590000	6.050000	-
ssd_mobilenet_uint8	iPhone 12 mini	tflite_flutter	metal	89.480000	4.960000	-
ssd_mobilenet_uint8	iPhone 7	onnxruntime-react-native	xnnpack	57.130000	3.270000	-
ssd_mobilenet_uint8	iPhone 7	onnxruntime-react-native	cpu	72.380000	2.900000	-
ssd_mobilenet_uint8	iPhone 7	onnxruntime-react-native	core_ml	93.190000	3.980000	-
ssd_mobilenet_uint8	iPhone 7	onnxruntime_flutter	xnnpack	51.620000	1.090000	-
ssd_mobilenet_uint8	iPhone 7	onnxruntime_flutter	core_ml	76.740000	4.180000	-
ssd_mobilenet_uint8	iPhone 7	react-native-fast-tflite	opengl	47.020000	3.510000	-
ssd_mobilenet_uint8	iPhone 7	tflite_flutter	cpu	175.510000	12.520000	-
ssd_mobilenet_uint8	iPhone 7	tflite_flutter	metal	208.680000	58.880000	-
ssd_mobilenet_uint8	iPhone 7	tflite_flutter	xnnpack	230.840000	22.070000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime-react-native	cpu	1027.000000	49.990000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime-react-native	xnnpack	1358.560000	23.870000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime-react-native	nnapi	3701.530000	32.970000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime_flutter	cpu	992.210000	47.060000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime_flutter	xnnpack	1441.130000	7.850000	-
deeplabv3_float32	Samsung Galaxy A40	onnxruntime_flutter	nnapi	4038.260000	31.270000	-
deeplabv3_float32	Samsung Galaxy A40	tflite_flutter	xnnpack	3109.510000	112.490000	-
deeplabv3_float32	Samsung Galaxy A40	tflite_flutter	gpu	5430.120000	639.930000	-
deeplabv3_float32	Samsung Galaxy A40	tflite_flutter	nnapi	6256.320000	2193.500000	-
deeplabv3_float32	Samsung Galaxy A40	tflite_flutter	cpu	6493.780000	2189.130000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	179.860000	14.240000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	198.370000	11.090000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime-react-native	xnnpack	438.630000	87.870000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	162.510000	14.090000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	186.620000	13.550000	-
deeplabv3_float32	Samsung Galaxy S20 FE	onnxruntime_flutter	xnnpack	388.060000	87.670000	-
deeplabv3_float32	Samsung Galaxy S20 FE	tflite_flutter	nnapi	1178.490000	51.910000	-
deeplabv3_float32	Samsung Galaxy S20 FE	tflite_flutter	gpu	1349.100000	49.290000	-
deeplabv3_float32	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	1486.740000	48.610000	-
deeplabv3_float32	Samsung Galaxy S20 FE	tflite_flutter	cpu	1588.350000	46.860000	-
deeplabv3_float32	iPhone 12 mini	onnxruntime-react-native	core_ml	38.450000	24.940000	-
deeplabv3_float32	iPhone 12 mini	onnxruntime-react-native	xnnpack	197.560000	3.930000	-
deeplabv3_float32	iPhone 12 mini	onnxruntime-react-native	cpu	210.910000	3.090000	-
deeplabv3_float32	iPhone 12 mini	onnxruntime_flutter	core_ml	33.340000	9.610000	-
deeplabv3_float32	iPhone 12 mini	onnxruntime_flutter	xnnpack	265.370000	110.730000	-
deeplabv3_float32	iPhone 12 mini	react-native-fast-tflite	core_ml	38.180000	12.040000	-
deeplabv3_float32	iPhone 12 mini	react-native-fast-tflite	opengl	175.120000	4.700000	-
deeplabv3_float32	iPhone 12 mini	tflite_flutter	metal	620.100000	32.300000	-
deeplabv3_float32	iPhone 12 mini	tflite_flutter	cpu	768.200000	40.020000	-
deeplabv3_float32	iPhone 12 mini	tflite_flutter	core_ml	793.730000	237.700000	-

deeplabv3_float32	iPhone 12 mini	tflite_flutter	xnnpack	834.770000	168.850000	-
deeplabv3_float32	iPhone 7	onnxruntime-react-native	core_ml	316.990000	14.180000	-
deeplabv3_float32	iPhone 7	onnxruntime-react-native	xnnpack	392.350000	19.400000	-
deeplabv3_float32	iPhone 7	onnxruntime-react-native	cpu	726.300000	74.290000	-
deeplabv3_float32	iPhone 7	onnxruntime_flutter	xnnpack	274.060000	3.600000	-
deeplabv3_float32	iPhone 7	onnxruntime_flutter	core_ml	334.330000	10.690000	-
deeplabv3_float32	iPhone 7	react-native-fast-tflite	opengl	390.560000	43.910000	-
deeplabv3_float32	iPhone 7	tflite_flutter	metal	1016.310000	53.210000	-
deeplabv3_float32	iPhone 7	tflite_flutter	xnnpack	1133.170000	44.260000	-
deeplabv3_float32	iPhone 7	tflite_flutter	cpu	1166.210000	32.590000	-
deeplabv3_uint8	Samsung Galaxy A40	onnxruntime-react-native	cpu	441.360000	11.860000	-
deeplabv3_uint8	Samsung Galaxy A40	onnxruntime-react-native	nnapi	4216.120000	35.170000	-
deeplabv3_uint8	Samsung Galaxy A40	onnxruntime_flutter	cpu	409.490000	7.570000	-
deeplabv3_uint8	Samsung Galaxy A40	onnxruntime_flutter	nnapi	4178.780000	35.430000	-
deeplabv3_uint8	Samsung Galaxy A40	tflite_flutter	gpu	1498.980000	66.560000	-
deeplabv3_uint8	Samsung Galaxy A40	tflite_flutter	cpu	1841.220000	68.550000	-
deeplabv3_uint8	Samsung Galaxy A40	tflite_flutter	xnnpack	1841.580000	69.380000	-
deeplabv3_uint8	Samsung Galaxy A40	tflite_flutter	nnapi	1944.790000	128.610000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	cpu	75.770000	7.090000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	onnxruntime-react-native	nnapi	101.910000	72.870000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	cpu	62.890000	4.830000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	onnxruntime_flutter	nnapi	119.520000	71.050000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	tflite_flutter	nnapi	535.670000	24.230000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	tflite_flutter	xnnpack	771.910000	22.390000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	tflite_flutter	cpu	772.140000	23.870000	-
deeplabv3_uint8	Samsung Galaxy S20 FE	tflite_flutter	gpu	844.800000	25.770000	-
deeplabv3_uint8	iPhone 12 mini	onnxruntime-react-native	cpu	115.360000	2.560000	-
deeplabv3_uint8	iPhone 12 mini	onnxruntime-react-native	core_ml	144.250000	22.990000	-
deeplabv3_uint8	iPhone 12 mini	onnxruntime_flutter	core_ml	140.710000	10.040000	-
deeplabv3_uint8	iPhone 12 mini	react-native-fast-tflite	core_ml	58.290000	0.640000	-
deeplabv3_uint8	iPhone 12 mini	react-native-fast-tflite	opengl	60.950000	1.700000	-
deeplabv3_uint8	iPhone 12 mini	tflite_flutter	core_ml	329.870000	22.180000	-
deeplabv3_uint8	iPhone 12 mini	tflite_flutter	cpu	331.680000	22.240000	-
deeplabv3_uint8	iPhone 12 mini	tflite_flutter	metal	331.950000	18.160000	-
deeplabv3_uint8	iPhone 12 mini	tflite_flutter	xnnpack	337.950000	21.630000	-
deeplabv3_uint8	iPhone 7	onnxruntime-react-native	cpu	475.140000	48.530000	-
deeplabv3_uint8	iPhone 7	onnxruntime-react-native	core_ml	653.880000	31.420000	-
deeplabv3_uint8	iPhone 7	onnxruntime_flutter	core_ml	573.190000	89.830000	-
deeplabv3_uint8	iPhone 7	react-native-fast-tflite	opengl	330.840000	35.050000	-
deeplabv3_uint8	iPhone 7	tflite_flutter	metal	594.760000	48.560000	-
deeplabv3_uint8	iPhone 7	tflite_flutter	cpu	714.220000	52.720000	-
deeplabv3_uint8	iPhone 7	tflite_flutter	xnnpack	861.220000	99.590000	-

Appendix B Code of the Test System

React Native Application: https://github.com/lehtoneo/rn-performance

Flutter Application: https://github.com/lehtoneo/flutter-inference-performance

Data API: https://github.com/lehtoneo/cross-platform-inference-data-api