Benchmark for Mobile Devices

Student: Danci Patricia Ioana

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

Contents 2
Introduction 7
1.1 Context 7
1.2 Objectives 7
List 1.2.1: Performance Metrics 7
List 1.2.2: Tools 8
Bibliographic Research 8
2.1 Mobile Device Benchmarks Overview 8
2.2 Synthetic vs Realistic Tests 8
2.3 Performance Metrics in Mobile Devices 9
2.4 Cross-Platform Development Considerations 9
Analysis 10
3.1 Project Proposal 10
List 3.1: App Features 10
3.2 Project Analysis 10
3.2.1 User Requirements 10
3.2.2 System Requirements 10
List 3.2.2.1: Functional Requirements 11
List 3.2.2.2: Non-Functional Requirements 11
3.2.3 Technology Stack 11
List 3.2.3: Libraries 11
3.2.4 Benchmark Test Analysis 12
3.2.4 Use Case Diagram 13
3.2.4 Main Use Case Diagram 13
3.2.4.1 CPU Use Case 13
3.2.4.2 GPU Use Case 14
3.2.4.3 Memory Use Case 14
3.2.4.4 Battery Use Case 14
3.4.4.5 View Results Use Case 15
3.4.4.6 History Use Case 15

Design 16
4.1 Overall Architecture 16
Figure 4.1.1: MCV Package Layout 16
Figure 4.1.2: General Architecture Design 16
4.1.1 Model 17
4.1.2 View 17
4.1.3 Controller 17
4.2. User Interface Design 17
4.2.1. Main Screen 18
Code Snippet 4.2.1: Layout SetUp in blank_layout.xml 18
4.2.1.1 Title Text 18
Code Snippet 4.2.1.1: Title Text in blank_layout.xml 18
4.2.1.2 Benchmark Buttons 18
Code Snippet 4.2.1.2: Benchmark Buttons in blank_layout.xml 19
4.2.1.3 Benchmarking History Button 19
Code Snippet 4.2.1.3: Benchmarking History Button in blank_layout.xml 19
4.2.1.4 Results Section 20
Code Snippet 4.2.1.4: Results Section in blank_layout.xml 20
4.2.2 History Screen 20
Code Snippet 4.2.2: History Layout SetUp in blank_layout.xml 20
4.2.2.1 History TextView 21
Code Snippet 4.2.2.1: History TextView in activity_history.xml 21
4.3 Data Models 21
List 4.3: Attributes for Data Models 21
4.4 Libraries Documentation 22
4.4.1 Room 22
4.4.2 Gson 22
4.4.3 Timber 22
4.4.4 MPAndroidChart 23
4.4.5 Android SDK (Core) 23
4.5 Class Design 23

4.5.1 Model 23
4.5.2 View 24
4.5.3 Controller 24
Figure 4.5.3.1: Sequence Diagram 27
Figure 4.5.3.2: User Interactions with the App 27
Figure 4.5.2: User Interactions with the App 28
Figure 4.5.4: UML Diagram 28
Implementation 29
5.1 View Package 29
5.1.1 MainActivity.java 29
5.1.2 HistoryActivity.java 30
5.1.3 InfoActivity.java 31
5.1.4 Info subpackage 31
5.1.4.1 BatteryInfoUI 31
5.1.4.2 CpuInfoUI.java 32
5.1.4.3 DeviceInfoUI.java 32
5.1.4.4 GpuInfoUI 32
5.1.4.5 MemoryInfoUI 32
5.1.4.6 OSInfoUI 33
5.2 Model Package 33
5.2.1 BenchmarkResult 33
5.2.2 Info Subpackage 34
5.2.2.1 BatteryInfo.java 34
5.2.2.2 CPUInfo.java 34
5.2.2.3 DeviceInfo.java 35
5.3.2.4 GPUInfo.java 35
5.3.2.5 MemoryInfo.java 36
5.3.2.6 OSInfo.java 36
5.3 Controller Package 37
5.3.1 BenchmarkController.java 37
5.3.2 Benchmarks subpackage 38

5.3.2.1 CpuBenchmark.java 38
5.3.2.2 CpuBenchmarkMulti.java 40
5.3.2.3 MemoryBenchmark.java 42
5.3.2.4 GpuBenchmark.java 44
5.3.2.5 *Gputools subpackage* 45
5.3.2.5.1 Model.java 45
5.3.2.5.2 Shader.java 45
5.3.2.5.3 VertexBuffer.java 46
Testing 47
6.1 Testing Objectives 47
6.2 Device Specifications 47
6.3 Test Cases 47
Figure 6.1 Application UI 48
Figure 6.2 Device information, CPU chart and benchmarking results for CPU and GPU 49
6.4 Limitations 49
6.5 Future testing plans 49
Conclusions 50
References 51

Introduction

1.1 Context

Benchmarking is a preset script or code to test a smartphone based on various parameters like CPU, GPU, battery life, and more. It can also test the performance of a processor for a better understanding. The higher the score, the better the performance [1].

Overall, benchmarking provides a standard set of tests designed to evaluate and compare the performance of mobile devices under different conditions.

The aim of this project is to develop an application that measures critical performance metrics such as CPU speed, memory usage, battery consumption and graphical processing capabilities. The primary focus will be both iOS and Android devices.

1.2 Objectives

This project will be designed using Java, Kotlin and Flutter (or React Native) through different IDEs such as Android Studio and Visual Studio Code, with the goal of creating a benchmark application for mobile devices that works on both iOS and Android platforms.

The main objective of this project is to implement tests that evaluate the following performance metrics:

List 1.2.1: Performance Metrics

- CPU performance through stress tests
- Memory Usage
- GPU performance
- Battery Consumption during intensive operations

Having a Windows computer, cross-platform frameworks such as Flutter will be used as they allow users to create a single codebase for both iOS and Android devices, despite development constraints. Since iOS apps cannot be compiled on Windows directly, either a Virtual Machine or a Cloud-Based macOS Service will be needed.

List 1.2.2: Tools

Android Development

- Android Studio
- Java/Kotlin

iOS Development

- Flutter or React Native (for cross-platform support)
- Access to macOS system (VM or cloud service)
- Xcode for compiling and testing

Additionally, a combination of both synthetic and realistic tests will be used to create a comprehensive assessment of mobile device performance.

Bibliographic Research

2.1 Mobile Device Benchmarks Overview

Almost every kind of tech product can be benchmarked, often in multiple ways. One of the most common types is the CPU benchmark, which directly impacts the speed of the device. There are a few ways to perform this benchmark as well, ranging from theoretical measurements like clock speed to real-world tests like video editing and gaming performance [2].

Currently, Geekbench and AnTuTu are amongst the most popular benchmark software.

AnTuTu (Android & iOS): Runs various tests like video playback, browser scrolling and gaming emulation. It provides an overall score with the breakdowns of CPU, GPU, memory (RAM & storage), and user experience (UX) [1].

Geekbench (Android & iOS): Goes deeper, focusing on CPU performance. It provides separate scores for single-core and multi-core tasks. Single-core reflects everyday activities like checking email, while multi-core tackles demanding tasks like video editing. Unlike AnTuTu, Geekbench uses realistic workloads to simulate real-world usage [1].

2.2 Synthetic vs Realistic Tests

Synthetic test data can offer several benefits for QA testing, such as the ability to create specific and customized scenarios, like edge cases or error conditions, that are difficult or impossible to find in real data [3]. These tests allow us to measure theoretical limits of device components, like CPU, GPU and

memory. They include mathematical calculations or data processing to evaluate processing speed, graphics rendering and data transfers.

On the other hand, real test data can provide several advantages for QA testing, such as validating the performance and functionality of your software in real-world conditions. It can also help you identify and fix issues that synthetic test data may miss, by exposing your software to the diversity of real data [3]. Such tests assess how devices perform under typical user scenarios. In this case, we can measure page loading times, video playback and app launch times.

By utilizing both synthetic and realist tests, raw capabilities and efficiency in everyday scenarios will be identified.

2.3 Performance Metrics in Mobile Devices

The most relevant performance metrics can be divided into several categories:

- CPU performance: A processor in a smartphone, often referred to as the Central Processing Unit (CPU), is the brain of the device. The processor's speed, efficiency, and number of cores (individual processing units within the CPU) significantly influence the smartphone's overall performance, affecting everything from app functionality and multitasking capability to battery life and the phone's ability to run complex or resource-intensive applications [4].
- Memory Management: Smartphones rely on RAM to hold the operating system (like Android and iOS), and to run apps and the data for those apps, as well as some caching and buffering data. The RAM needs to be organized and managed so that the apps can run smoothly. When a new app is launched, a free place in memory needs to be found to load the app and start it running [5].
- GPU performance: The GPU is responsible for rendering graphics and handling complex visual tasks, such as playing video games, rendering animations, and displaying high-resolution images [6]. The benchmark is done by running a workload that stress tests the GPU, which is similar to the content that the consumer will run on the device, such as gaming content.
- Battery Life: The average smartphone battery can last anywhere from 8 to 12 hours on a single charge, depending on usage and the device's specifications [7]. Modern benchmarks often assess how quickly a device consumes battery power under specific workloads [7]. We will measure battery consumption by monitoring the device’s power usage during specific tasks.

2.4 Cross-Platform Development Considerations

Multiplatform mobile development is an approach that allows you to build a single mobile application that runs smoothly on several operating systems. In cross-platform apps, some or even all of the source code can be shared. This means that developers can create and deploy mobile assets that work on both Android and iOS without having to recode them for each individual platform [8].

Frameworks such as Flutter or React Native enable developers to write code once and deploy it on multiple platforms. These multiplatform apps are designed to run identically on different mobile platforms.

This research will investigate whether these frameworks provide sufficient performance access to implement benchmarks for both Android and iOS.

Analysis

3.1 Project Proposal

The primary goal of the application is to give users insight into how their device handles intensive tasks, allowing them to make informed decisions about their device's capabilities. The final device will encompass the following features:

List 3.1: App Features

- CPU Benchmarking
- GPU Benchmarking
- Memory Benchmarking
- Battery Benchmarking
- History Viewing

3.2 Project Analysis

3.2.1 User Requirements

The application is designed to meet the needs of everyday users, as well as developers and technical testers. In order to meet such needs, the application provides a wide selection of benchmarking tests that are easy to use and understand, ranging from simple to more complex ones. Benchmarking results are displayed instantly on the main screen, providing the application with a clean, intuitive interface.

3.2.2 System Requirements

Currently, the application is mainly designed to be deployed on Android devices, with future considerations of a cross-platform implementation.

The following system requirements must be fulfilled for the application to function effectively: Functional and Non-Functional. They are described below:

List 3.2.2.1: Functional Requirements

1. Benchmarking Tests:
    - CPU Benchmark
    - GPU Benchmark
    - Memory Benchmark
    - Battery Benchmark
2. Benchmark History:
    - Store results in a structured format
    - Display previous data
3. User Interface:
    - Buttons for each benchmarking tests
    - Display result on the main screen

List 3.2.2.2: Non-Functional Requirements

1. Performance: Benchmarks should execute efficiently without significant delays or system slowdowns
2. Reliability: Results should be consistent and accurate
3. Usability: Design should be user-friendly
4. Compatibility: Support devices with Android API level 21 and above, ensuring wide device coverage

3.2.3 Technology Stack

The project will primarily utilize Java as the main programming language, although Flutter framework is also considered as an alternative for future cross-platform implementation. The Android SDK will be used to interact with the device's hardware and perform benchmarking tasks.

Besides that, several libraries will support the project's functionality:

List 3.2.3: Libraries

1. Room: for managing and storing data
2. Gson: JSON data handling
3. Timber: logging and debugging
4. MPAndroidChart: visualization

3.2.4 Benchmark Test Analysis

Performance metrics play a critical role in device efficiency. To achieve an evaluation of such data, the project will implement a set of designed tests, each chosen for its relevance to real-world usage.

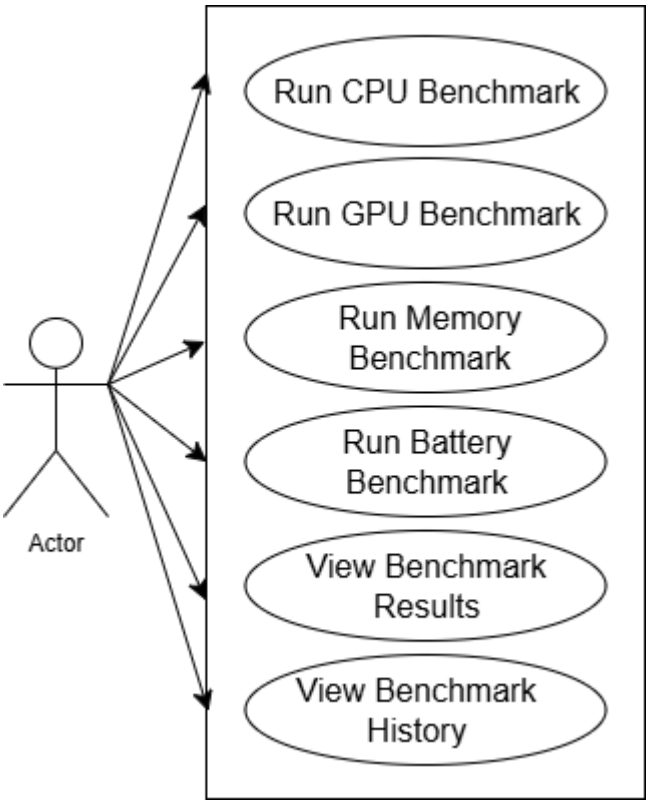
First, the **CPU** is responsible for executing the app's instructions, its performance directly impacting the app's speed and responsiveness. The CPU tests will involve Integer and Floating-Point Arithmetic calculations, such as Loop Intensity or Prime Number Calculation test. Their purpose is relevant for everyday app functionality. Additionally, Single-Core and Multi-Core performance tests can be done, such as basic web browsing, social media apps, complex data processing, or gaming. Lastly, tests can simulate compression and decompression tasks, which are common in file handling.

Next, the **GPU** handles rendering and is crucial for graphics-intensive applications like games or apps with complex visuals. Testing this metric includes mostly simulation of rendering operations, namely drawing shapes on a canvas. Moreover, shader tests, texture filtering and fil rate tests, as well as bitmap manipulation tests will assess the efficiency of the GPU.

After that, the **Memory** management plays a significant role in app stability and multitasking performance. The memory test will measure the device's ability to allocate, use, and release memory efficiently. Methods of performing such analysis include Memory Bandwidth tests (e.g.: Array Sorting), which measure the speed at which data is transferred between the CPU and memory, Latency tests, that assess the time taken to access stored data, and Database Read/Write tests, which simulate reading and writing from a database. Furthermore, memory allocation and garbage collection tests will evaluate the device's ability to ensure smooth multitasking and reduce the possibility of app slowdowns.

Finally, **Battery** performance is essential for maintaining prolonged device usage. Tests here will monitor the rate of battery consumption during intensive operations, such as continuous calculations or data processing. Possible implementations cover Power consumption during high CPU usage and battery drain under combined load. Moreover, idle battery consumption tests will record battery levels before and after a period of inactivity, while screen brightness and GPU battery tests will measure battery drain by increasing screen brightness to maximum and running GPU operations.

3.2.4 Use Case Diagram



3.2.4 Main Use Case Diagram

3.2.4.1 CPU Use Case

- Use Case: Run CPU Benchmark
- Primary Actor: User
- Main Success Scenario:
    - User opens the application
    - User clicks on the CPU Benchmark button
    - The system initiates the CPU Benchmark by performing several calculations
    - The system computes the CPU Benchmark score based on the time taken for all calculations
    - The system displays the result on the screen
    - The user sees the benchmark result and receives a confirmation message
- Alternative Sequence:
    - If the system encounters an error during the benchmark, it displays a message

3.2.4.2 GPU Use Case

- Use Case: Run GPU Benchmark
- Primary Actor: User
- Main Success Scenario:
    - User opens the application
    - User clicks on the GPU Benchmark button
    - The system initiates the GPU Benchmar, simulation different rendering tasks
    - The system calculates the benchmark score based on rendering performance
    - The system displays the result on the screen
    - The user sees the benchmark result and receives a confirmation message
- Alternative Sequence:
    - If the device's GPU doesn't support certain tasks, the system skips those tests and continues with the rest
    - If there is insufficient memory for rendering tasks, the system displays an error message and aborts the test

3.2.4.3 Memory Use Case

- Use Case: Run Memory Benchmark
- Primary Actor: User
- Main Success Scenario:
    - User opens the application
    - User clicks on the Memory Benchmark button
    - The system initiates the memory benchmark, conducting tests like memory bandwidth, latency, and database read/write operations
    - The system computes the memory benchmark score based on the speed and efficiency of memory handling
    - The system displays the result on the screen
    - The user sees the benchmark result and receives a confirmation message
- Alternative Sequence:
    - If there is insufficient memory to run the test, the system notifies the user and aborts the test
    - If memory results are inconclusive, the system suggests retrying

3.2.4.4 Battery Use Case

- Use Case: Run Battery Benchmark
- Primary Actor: User
- Main Success Scenario:

    - User opens the application
    - User clicks on the Battery Benchmark button
    - The system initiates battery usage tests by performing continuous calculations and monitoring the battery drain
    - The system calculates battery performance based on the rate of battery consumption during the test
    - The system displays the result on the screen
    - User sees the benchmark result and receives a confirmation message
- Alternative Sequence:
    - If the battery level is below a minimum threshold (e.g.: 15%), the system warns the user and cancels the test
    - If the device is charging, the system suggests disconnecting from power to run an accurate test
    - If the battery test cannot complete due to power constraints, it provides partial results or suggests a retry

3.4.4.5 View Results Use Case

- Use Case: View Benchmark Results
- Primary Actor: Users
- Main Success Scenario:
    - User completes a benchmark (CPU, GPU, Memory, or Battery)
    - The system automatically displays the benchmark results on the main screen under Benchmark Results section
    - User sees the results for each test conducted
- Alternative Sequence:
    - If the user hasn't run a test yet, the system shows a placeholder text ("Not tested")
    - If the user wants to view past results, they can navigate to Benchmarking History

3.4.4.6 History Use Case

- Use Case: View Benchmarking History
- Primary Actor: User
- Main Success Scenario:
    - User opens the application and clicks on the Benchmarking History button
    - The system navigates to a new screen displaying a list of past benchmark results with timestamps
    - User scrolls through the results to view past performance data for CPU, GPU, Memory, and Battery tests
- Alternative Sequence:
    - If no benchmarking history exists, the system displays a message ("No benchmarking history available")

- If the app cannot receive past results, the system shows an error message

Design

4.1 Overall Architecture

The Benchmark Application is structured using the Model-View-Controller (MVC) architectural pattern. This approach separates the logic and presentation layers, making the application easier to manage.

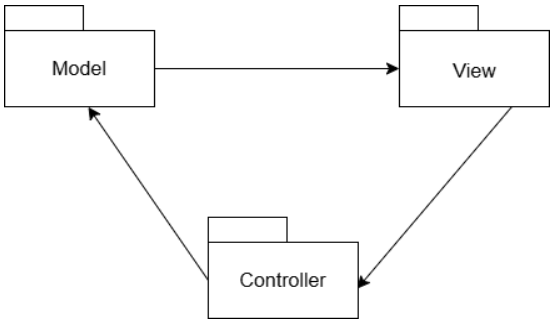


Figure 4.1.1: MCV Package Layout

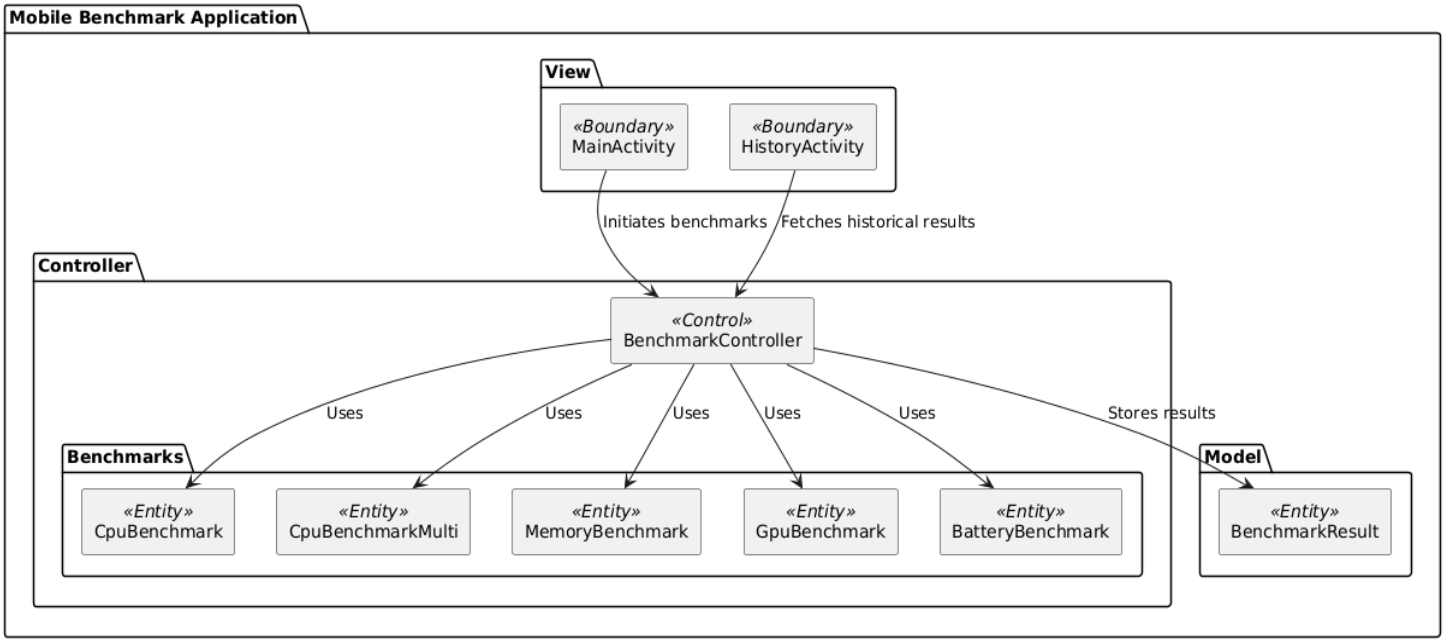


Figure 4.1.2: General Architecture Design

4.1.1 Model

The Model component manages the data and logic of the application. Currently, the BenchmarkResult.java class represents the model, which holds data related to each benchmarking test, including both timestamp and metrics score. This class is designed to support serialization, allowing data to be easily stored and retrieved. In order to ensure that data is independent of the UI, this package only interacts with the controller.

4.1.2 View

The View component is responsible for the user interface and presentation layer. In this application, the MainActivity.java and HistoryActivity.java classes, along with their corresponding XML layout files, currently serve as the view, although further implementations might come with additional UIs.

Therefore, the MainActivity.java class displays the main screen where users can initiate each benchmarking test (CPU, GPU, Memory, Battery) and view results instantly.

In addition, the HistoryActivity.java class provides a historical view of past benchmark results, allowing users to see previous performance metrics.

The View package also interacts only with the controller, avoiding any direct manipulation of data and focusing mainly on the presentation.

Each XML layout file defines the structure of its respective screen, using RelativeLayout for flexibility and convenience.

4.1.3 Controller

The Controller component handles the application's logic and acts as an intermediary between the Model and the View. Here, classes are responsible for managing the execution of benchmark tests.

This separation allows the controller to handle any changes in data without directly impacting the view.

Overall, the Controller manages user interactions and processes these actions by performing the corresponding benchmark and updating the model. In addition, it ensures that the data generated is stored in history.

4.2. User Interface Design

The Benchmark Application's UI is designed to be straightforward and user-friendly. The layout is structured using Android's RelativeLayout, which allows each element to be positioned relative to others. Alternative layout include LinearLayout and ConstraintLayout, but the Relative option was chosen for its balance between simplicity and flexibility.

At present, the UI is divided into two main screens:

4.2.1. Main Screen

Designed inside the blank_layout.xml file, this UI includes primary options for initiating benchmark tests and displays the results directly beneath each option for a quick view. The design utilizes simple buttons and text views for each test type.

A placeholder text “Not tested” appears until each button is pressed. If a performance metric measuring test is yet to be implemented, pressing its corresponding button leads to the display of “Not implemented” text.

XML Structure: the main screen’s XML layout is structured with RelativeLayout and includes buttons and text views arranged in a vertical stack, centered horizontally.

Code Snippet 4.2.1: Layout SetUp in blank_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

Displays the title of the application (“Device Benchmark”), centered at the top of the screen.

Code Snippet 4.2.1.1: Title Text in blank_layout.xml

```
<!-- Title Text -->
<TextView
    android:id="@+id/titleText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Device Benchmark"
    android:textSize="24sp"
    android:textStyle="bold"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="64dp"
    android:layout_marginBottom="16dp" />
```

- CPU Benchmark: triggers a CPU performance test
- GPU Benchmark: triggers a GPU performance test

- Memory Benchmark: triggers a memory performance test
- Battery Benchmark: triggers a battery performance test

Code Snippet 4.2.1.2: Benchmark Buttons in blank_layout.xml

```
<!-- CPU Benchmark Button -->
<Button
    android:id="@+id/cpuBenchmarkButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="CPU Benchmark"
    android:layout_below="@id/titleText"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="16dp" />

<!-- GPU Benchmark Button -->
<Button
    android:id="@+id/gpuBenchmarkButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="GPU Benchmark"
    android:layout_below="@id/cpuBenchmarkButton"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="8dp" />

…<! -Same for rest>
```

4.2.1.3 Benchmarking History Button

Located down below on the screen, this button navigates the user to a separate screen displaying past benchmark results.

Code Snippet 4.2.1.3: Benchmarking History Button in blank_layout.xml

```
<!-- Benchmarking History Button -->
<Button
    android:id="@+id/historyButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Benchmarking History"
    android:layout_below="@id/batteryBenchmarkButton"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="256dp" />
```

4.2.1.4 Results Section

After running a benchmark, the results are displayed in text views. Each performance metric has its own result text view, where the app updates the result immediately after completing each test.

Code Snippet 4.2.1.4: Results Section in blank_layout.xml

```
<!-- Results Title Text -->
<TextView
    android:id="@+id/resultsTitleText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Benchmark Results"
    android:textSize="18sp"
    android:textStyle="bold"
    android:layout_below="@id/batteryBenchmarkButton"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="24dp" />

<!-- CPU Result Text -->
<TextView
    android:id="@+id/cpuResultText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="CPU: Not tested"
    android:layout_below="@id/resultsTitleText"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="8dp" />

…<! -Same for rest>
```

4.2.2 History Screen

The History Screen displays a list of past benchmarking results in a text view. This screen allows users to review previous test results in a simple, scrollable format. Each entry includes the timestamp, CPU score, GPU score, Memory score, and Battery score.

XML Structure: the history screen also uses a RelativeLayout with a single TextView at the top of the display.

Code Snippet 4.2.2: History Layout SetUp in blank_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
    <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp">
```

4.2.2.1 History TextView

This Text View displays the benchmark history, its content being updated each time a benchmark is completed and added to the history.

Code Snippet 4.2.2.1: History TextView in activity_history.xml

```
<TextView
    android:id="@+id/historyTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="No history available"
    android:textSize="16sp"
    android:padding="8dp"
    android:layout_alignParentTop="true"
    android:layout_marginTop="16dp"/>
```

4.3 Data Models

The Data Models are designed to capture and pile performance metrics from benchmark tests. Each test's results are stored with specific attributes that make it easy to retrieve, display, and analyze.

The primary data model used in this application is the BenchmarkResult.java class inside the model package, which encapsulates the details of a single benchmark test result.

The following attributes are used:

List 4.3: Attributes for Data Models

- Timestamp (String): Records the date and time the benchmark was performed
- cpuScore (int): Holds the performance score for the CPU test
- gpuScore (int): Stores the GPU performance score
- memoryScore (int): Contains the Memory performance score
- batteryScore (int): Tracks the Battery performance, typically measured by power consumption

The BenchmarkResult class is designed to support *serialization*, making it possible to save and retrieve data from device storage efficiently. This is achieved by implementing the *Serializable* Interface.

Each time a benchmark test is completed, a BenchmarkResult instance is created with the test's results and added to a list.

The list of BenchmarkResult objects is used to display a history of tests in the HistoryActivity screen.

In the current application, only the BenchmarkResult model is implemented to store data. However, future extensions could introduce other data models, such as UserSettings or DeviceInfo, designed to store user preferences and capture detailed information about the device, like model, OS version, and hardware specifications.

4.4 Libraries Documentation

To enhance functionality, simplify development, and improve app performance, several key libraries will be used. Each library serves a specific purpose, helping with tasks like data management, as well as potential future data visualization.

4.4.1 Room

The Purpose of this library is to provide an API for data persistence, allowing for simple queries and data storage without the complexity of handling raw SQL.

In this application, the Usage of Rooms could be to store historical benchmark results in a persistent database. This would ensure that data is available even after the app is closed and reopened. Additionally, Room supports offline storage, meaning users can access their benchmark history without an internet connection.

This library is preferred over plain SQLite because it integrates with LiveData and ViewModel, offering a more developer-friendly API for database interactions.

4.4.2 Gson

Gson is a Java library for converting Java objects into their JSON representation and vice versa. This is especially useful when dealing with data that needs to be shared or serialized for storage.

Although not fully implemented, Gson could be used to serialize the BenchmarkResult objects into JSON format before storing them. If the app were to add a cloud storage feature, Gson would help convert data between JSON and Java objects for API communication.

This library is simple, fast, and widely used in Android applications. It also integrates well with Retrofit, should future development include server communication.

4.4.3 Timber

Timber is a lightweight logging library specifically designed for Android. It provides simple API to log messages, making it easir to manage and debug the app.

This library could be used throughout the app to log errors and debugging information, especially during development and testing phases. Moreover, it helps in maintaining clean logs while still allowing debugging during development.

4.4.4 MPAndroidChart

MPAndroidChart is a library designed for creating charts and graphs on Android.
This library is considered for visualizing benchmarking results, as it could be used to display metrics performance over time, helping users compare scores.
In this application, the library could be integrated into the history screen to present benchmark data in a more interactive and insightful format.

4.4.5 Android SDK (Core)

The Android SDK provides the core libraries and APIs needed for Android development. It includes essential components like UI elements, background processing, and device hardware interaction.
Inside the application, it is used to access the device's hardware resources for benchmarking.
It also provides access to UI components and allows the app to use device features such as storage.
As a fundamental part of Android app development, the Android SDK is indispensable.

4.5 Class Design

4.5.1 Model

The Model package is designed to manage the application's data, specifically the benchmark results.
The primary class in this package is:

- BenchmarkResult.java:
  Designed to represent a single benchmark result (like a record), it encapsulates all performance metrics, along with a timestamp indicating when the benchmark was performed.
  This class also support serialization, allowing efficient storage and retrieval of historical records.
  In terms of DESIGN, this class holds only methods for getters and setters, since it acts as a "bridge" between the View and the Controller.
  It is instantiated both inside the View and the Controller package, in order to either introduce or retrieve results from benchmarkings.

4.5.2 View

The View package is responsible for the presentation layer, managing the user interface and user interactions. It contains three main classes:

- MainActivity.java
  Represents the main screen of the application, where users can initiate each benchmarking test and view the result on the screen.
  In terms of methods, it overrides onCreate, which is called when the activity is starting. Besides this method, it also contains action listeners for each of the 4 buttons: CPU, GPU, Memory and Battery;
  each action listener is designed to call the corresponding method inside the Controller class, while the ExecutorService is used in order to avoid running the benchmark tests directly on the UI Thread; this approach ensures that the tasks are ran in the background, in order to avoid ANRs.
- HistoryActivity.java
  Provides a historical view of past benchmarking results.
  The screen displays a list of previous performance metrics, enabling users to track and compare results over time.
  It is instantiated inside the MainActivity class with the help of an Intent (*Intent* intent = new *Intent*(*MainActivity*.this, *HistoryActivity*.class)) and opened with *startActivity*(intent);
  This class only holds the onCreate method, since it represents an additional UI that is created once the History Button is pressed; inside this method, the benchmark results are appended to a text view, in order to be displayed as a list.
- InfoActivity.java
  This class serves as the main interface for accessing detailed information about the device. It provides a user-friendly menu with buttons that lead to specific information sections, including CPU, GPU, memory, battery, operating system, and general device information. Each button navigates the user to a corresponding activity where the requested details are displayed.

In addition to these three classes, the view package contains an *info* subpackage.
This subpackage holds classes such as BatteryInfoUI, CpuInfoUI, GpuInfoUI, DeviceInfoUI, MemoryInfoUI, and OSInfoUI, each one responsible for retrieving corresponding data regarding selected parameters.

4.5.3 Controller

The Controller package coordinates between the View and the Model packages. It manages application logic, processing user actions and benchmarking tests. This package includes:

- BenchmarkController.java:

This class serves as the central controller, responsible for managing benchmark execution.
It initializes a BenchmarkResult object *currentResult*, which udpates as each benchmark method gets called.
In addition, it contains methods of type BenchmarkResult for performing each benchmark test; these methods each call their corresponding benchmark class, aggregate the result, and compute the final score for each performance metric.
Besides these methods, this class holds a list of BenchmarkResult objects for tracking results history and uses Context to access resources like image files and raw data for tasks such as image processing or file compression/decompression.
Ultimately, its main two methods *finalizeBenchmark()* and *computeOverallScore(BenchmarkResult result)* are responsible for saving a full benchmark into history, as well as computing the overall score relative to a threshold of 1000.

- Benchmarks (subpackage): Contains individual classes for each benchmark test type
    - ***CpuBenchmark.java***
      This class executes CPU tests, both synthetic and real-life;
      It is responsible for the single-core CPU performance computation
      For each of its methods, it computes the time (given by *System.nanoTime*();) then converts it to milliseconds.
      When instantiated inside the main Controller class, each method's time is assigned to a value, then the score is computed as the geometric mean of all results.
      The lower the score, the higher the performance.
    - ***CpuBenchmarkMulti.java***
      This class uses a similar approach to the previous one, except it tests for multi-core performance.
      Inside this class, most of the methods of the single-core are kept, with few adjustments where needed. This class primarly uses ExecutorService in order to divide the workload amongst available cores, while computing the final result using *Future* and *invokeAll*.
      The score is computed in the same manner as the single-core one.
    - ***MemoryBenchmark.java***
      This class measures memory efficiency through tests on bandwidth, latency, and memory management.
      It contains several methods, providing both synthetic and real-life tests.
      Since these methods return results in different measurement units, the score computation is divided into two main categories, based on tests returning time and tests returning MB/s.
      When instantiated inside the main Controller class, latency and bandwidth values are calculated using the geometric mean, with both scores being displayed as the result. Since a higher latency signifies a

worse performance, whereas a higher bandwidth results in higher performance, these results are displayed separately. Generally, the lower the latency score, the higher the bandwidth score will be.

- ***GpuBenchmark.java with subpackage gputools***
  This class is designed to assess GPU rendering capabilities by running a 3D rendering benchmark. It uses OpenGL ES 2.0 to simulate real-world rendering workloads, rendering 3D frames for a fixed duration of 30 seconds. The number of frames rendered during the benchmark (frameCount) serves as the primary performance metric, where a higher frame count indicates better GPU performance. This class employs custom shaders, 3D models (Stanford Armadillo), and lighting configurations to create a realistic GPU workload. It integrates with the BenchmarkController to provide detailed insights into GPU efficiency.
  The ***gputools subpackage*** includes helper classes used by GpuBenchmark. The **Model.java** class is responsible for loading and parsing 3D models into vertex positions, normals, and texture coordinates. It assembles triangles for rendering and binds the data to shader attributes. The **Shader.java** class manages OpenGL ES shader programs, compiling and linking vertex and fragment shaders while providing utilities for setting shader uniforms and attributes. It also handles error reporting for issues encountered during compilation or linking. The **VertexBuffer.java** class stores vertex data in a GPU-friendly format and connects it to the shaders for rendering. It ensures efficient rendering of 3D models by mapping parsed model data directly to the GPU for processing.

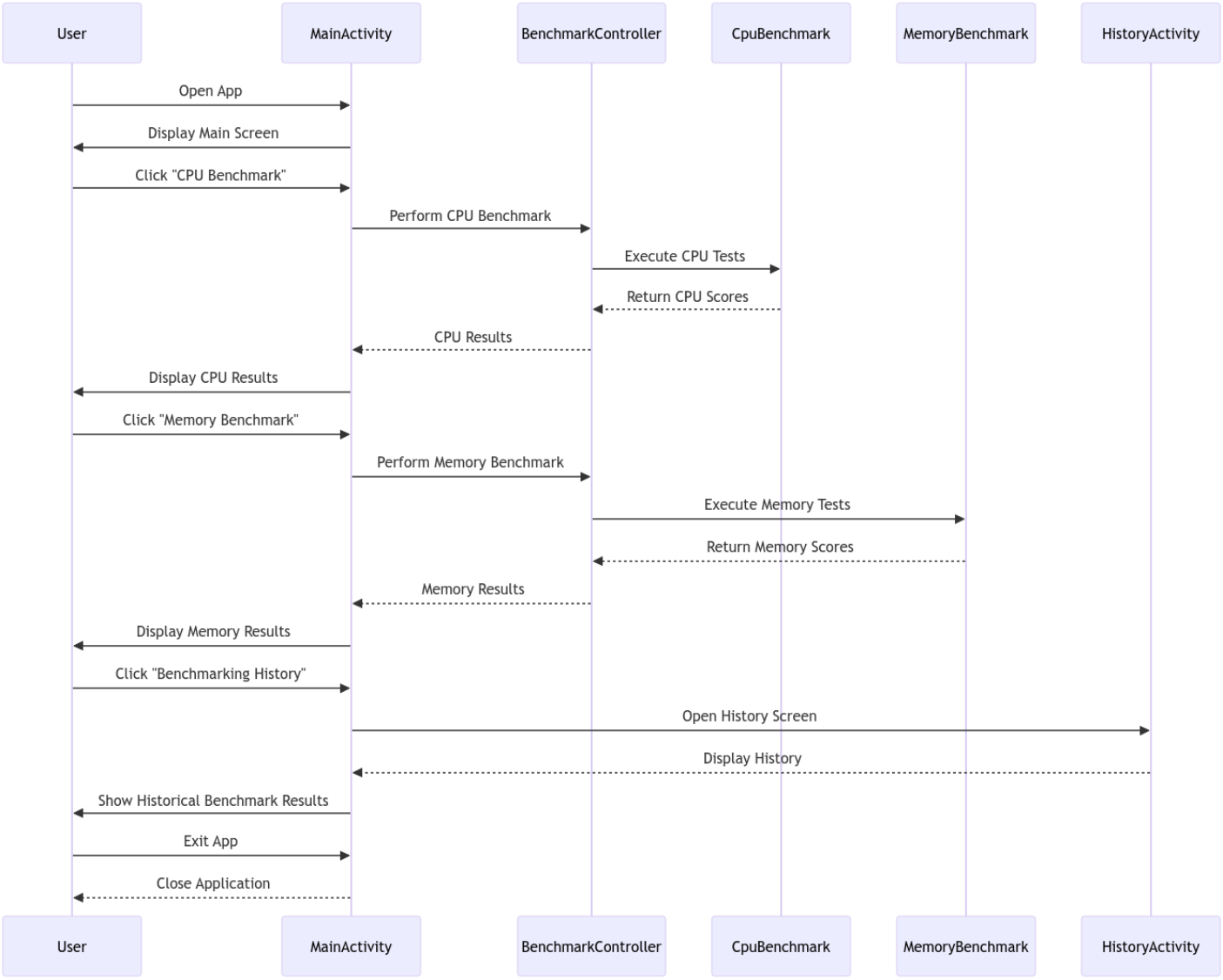


Figure 4.5.3.1: Sequence Diagram

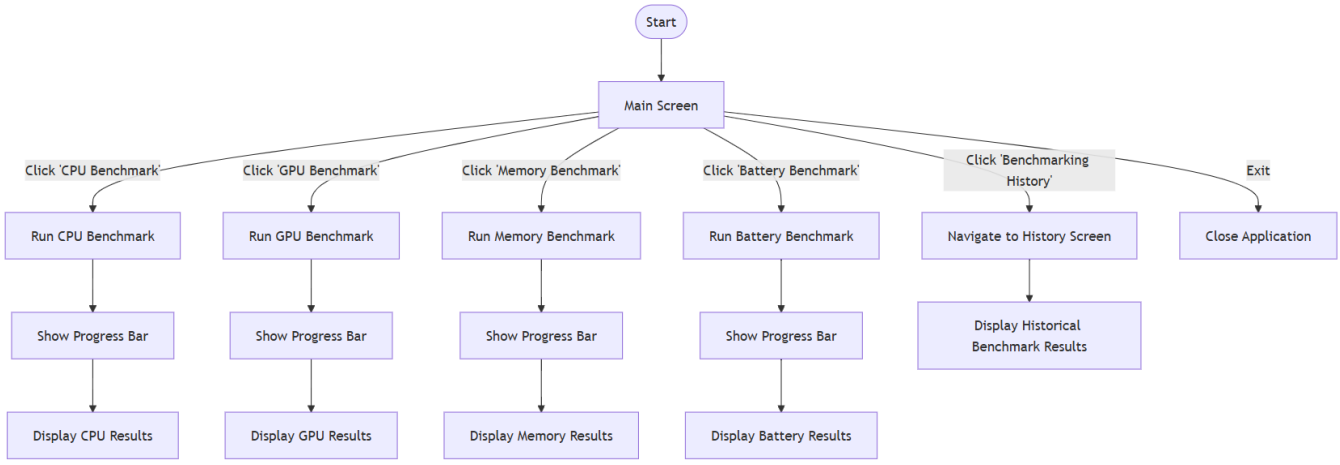


Figure 4.5.3.2: User Interactions with the App

4.5.4 UML Diagram

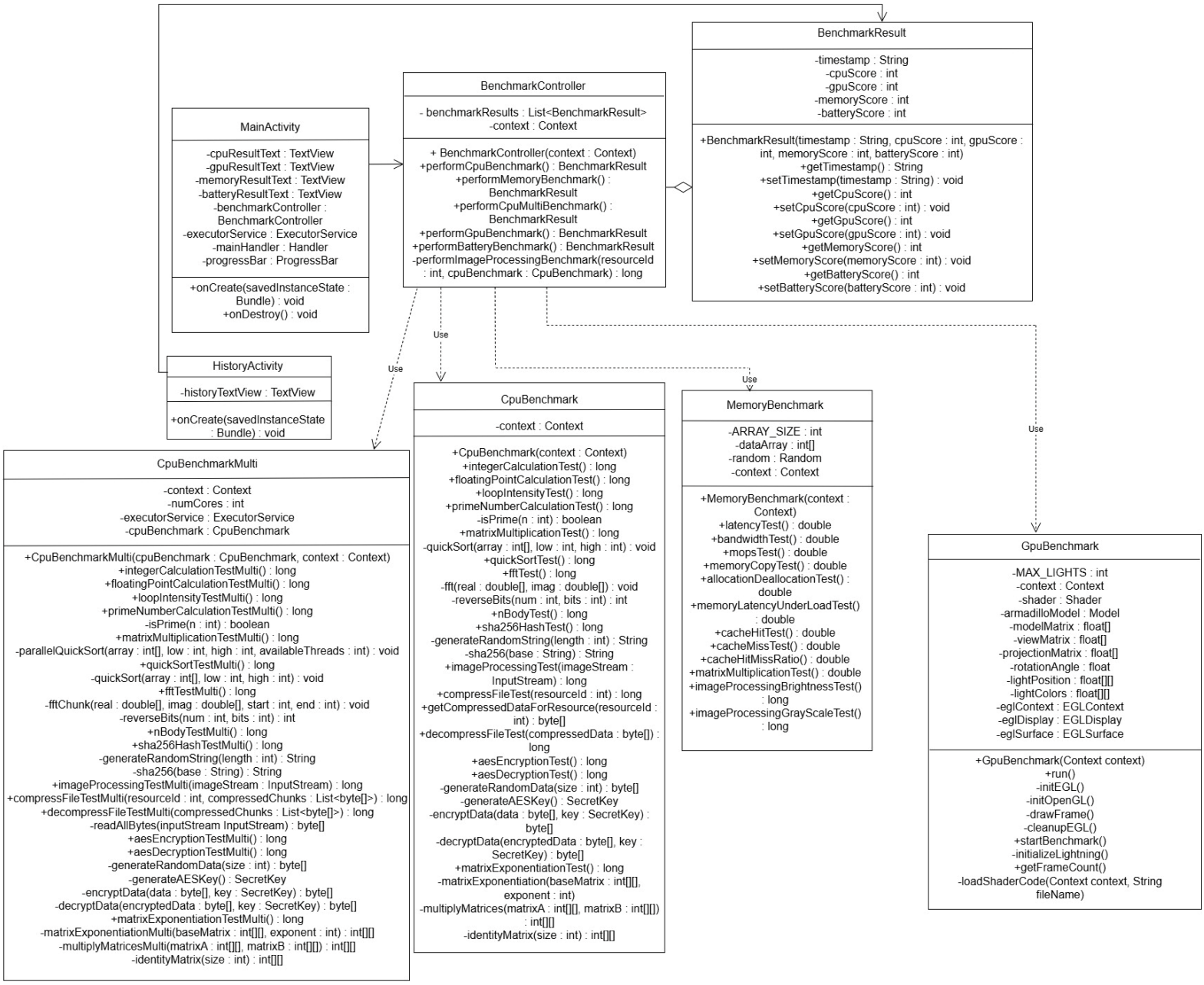


Figure 4.5.4: UML Diagram

Implementation

5.1 View Package

5.1.1 MainActivity.java

The MainActivity.java class serves as the main screen of the application. It extends AppCompActivity, which is necessary to provide lifecycle support for an Android activity.

Attributes:

- **TextView cpuResultText, gpuResultText, memoryResultText, overallScoreText**: These are UI elements used to display the benchmark results for CPU, GPU, Memory, and the overall score.
- **BenchmarkController benchmarkController**: This is the main controller used to manage and execute benchmarks. It interacts with various benchmark classes.
- **ExecutorService executorService**: A single-threaded executor that handles tasks in the background, ensuring that benchmarking tasks do not block the main UI thread.
- **Handler mainHandler:** This handler posts results from the background thread to the main thread, allowing UI updates after a benchmark completes.
- **ProgressBar progressBar**: A progress indicator shown while a benchmark is in progress, giving feedback to the user.
- **Button cpuBenchmarkButton, gpuBenchmarkButton, memoryBenchmarkButton, historyButton, infoButton, finalizeBenchmarkButton**: Buttons to start benchmarks, view history, and finalize benchmarks.

Methods:

- **onCreate(Bundle savedInstanceState)**: This method initializes the UI and components when the activity is created. It sets up buttons for each benchmark type and associates click listeners to start benchmarks or show history.
    - Initializes benchmarkController, executorService, and mainHandler.
    - Finds and links buttons and TextView elements from the layout.
    - Sets up button click listeners for each type of benchmark (CPU, GPU, Memory, Battery) and history view.

- **Button Click Listeners**:
    - CPU Benchmark Button:
        - Sets the CPU result text to "Benchmark in progress..." and shows the progress bar.
        - Executes the performCpuBenchmark and performCpuMultiBenchmark methods in BenchmarkController on a background thread.

        - Once the benchmark completes, updates the UI with the CPU score and hides the progress bar.
    - GPU Benchmark Button:
        - Sets the GPU result text to "Benchmark in progress..." and shows the progress bar.
        - Executes the performGpuBenchmark method in BenchmarkController on a background thread.
        - Once the benchmark completes, updates the UI with the GPU score and hides the progress bar.
    - Memory Benchmark Button:
        - Sets the Memory result text to "Benchmark in progress..." and shows the progress bar.
        - Executes the performMemoryBenchmark method in BenchmarkController on a background thread.
        - Updates the UI with the Memory score and hides the progress bar after completion.
    - History Button:
        - Opens the HistoryActivity to view the history of benchmark results. It passes the list of benchmark results to HistoryActivity via an Intent.
    - Info Button:
        - Opens the InfoActivity to view additional information.
    - Finalize Benchmark Button:
        - Calls benchmarkController.finalizeBenchmark() to finalize the benchmark.
        - Computes the overall score using benchmarkController.computeOverallScore().
        - Updates the overallScoreText with the computed overall score.
        - Resets the individual benchmark result texts to "Not benchmarked".
        - Displays a toast message with the overall score.
        - Disables the finalizeBenchmarkButton and resets the benchmark completion flags.

- **onDestroy()**: This method is called when the activity is destroyed. It shuts down the executorService to free up resources and avoid memory leaks.

5.1.2 HistoryActivity.java

The HistoryActivity class provides a screen that displays the history of previous benchmark results. It extends AppCompatActivity, which is necessary for managing the Android activity lifecycle.

Attributes:

- **TextView historyTextView**: A UI element used to display benchmark history in text format.

Methods:

- **onCreate(Bundle savedInstanceState)**: This method is called when the activity is created. It initializes the layout and retrieves benchmark data to display.
    - Sets the content view to activity_history.
    - Finds the TextView for displaying history by its ID.
    - Retrieves the list of BenchmarkResult objects passed from MainActivity via an Intent.
    - If benchmark results are available, iterates over each result and appends its details (timestamp, CPU score, GPU score, Memory score) to a StringBuilder. This text is then set in the historyTextView to display the results.
    - If no benchmark results are available, sets the historyTextView to display "No benchmarking history available."

5.1.3 InfoActivity.java

The InfoActivity class provides a user interface for accessing various information about the device, such as CPU, GPU, memory, battery, OS, and device details. It includes buttons that navigate to different activities displaying the respective information.

Methods:

- **onCreate(Bundle savedInstanceState):** Initializes the activity, sets the content view, and sets up button click listeners for navigating to different information activities.

5.1.4 Info subpackage

5.1.4.1 BatteryInfoUI

The BatteryInfoUI class is an activity that displays detailed information about the device's battery. It initializes a BatteryInfo object to fetch battery details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState)**: Initializes the activity, sets the content view, fetches battery details using BatteryInfo, and displays the information in a TextView.

5.1.4.2 CpuInfoUI.java

The CpuInfoUI class is an activity that displays detailed information about the device's CPU. It initializes a CPUInfo object to fetch CPU details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState):** Initializes the activity, sets the content view, fetches CPU details using CPUInfo, and displays the information in a TextView.

5.1.4.3 DeviceInfoUI.java

The DeviceInfoUI class is an activity that displays detailed information about the device. It initializes a DeviceInfo object to fetch device details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState)**: Initializes the activity, sets the content view, fetches device details using DeviceInfo, and displays the information in a TextView.

5.1.4.4 GpuInfoUI

The GpuInfoUI class is an activity that displays detailed information about the device's GPU. It initializes a GPUInfo object to fetch GPU details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState):** Initializes the activity, sets the content view, fetches GPU details using GPUInfo, and displays the information in a TextView.

5.1.4.5 MemoryInfoUI

The MemoryInfoUI class is an activity that displays detailed information about the device's memory. It initializes a MemoryInfo object to fetch memory details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState):** Initializes the activity, sets the content view, fetches memory details using MemoryInfo, and displays the information in a TextView.

5.1.4.6 OSInfoUI

The OSInfoUI class is an activity that displays detailed information about the device's operating system. It initializes an OSInfo object to fetch OS details and displays this information in a TextView.

Methods:

- **onCreate(Bundle savedInstanceState):** Initializes the activity, sets the content view, fetches OS details using OSInfo, and displays the information in a TextView.

5.2 Model Package

5.2.1 BenchmarkResult

This package contains the BenchmarkResult.java class, a simple data structure that stores the results of each benchmarking test, including scores for CPU, GPU, memory, and battery. It provides getter and setter methods to access and modify these values. By implementing Serializable, this class allows instances to be easily passed between activities, making it suitable for storing and sharing benchmarking data within the app.

Attributes:

- **timestamp** (String): Stores the date and time when the benchmark was performed.
- **cpuScore** (int): Holds the score result for the CPU benchmark.
- **gpuScore** (int): Holds the score result for the GPU benchmark.
- **memoryScore** (int): Holds the score result for the memory benchmark.
- **batteryScore** (int): Holds the score result for the battery benchmark.

Constructor:

- **BenchmarkResult(String timestamp, int cpuScoreSingle, int cpuScoreMulti, int gpuScore, int memoryLatencyScore, int MemoryBandwidthScore)**: Initializes the object with the timestamp and scores for CPU, GPU, and Memory benchmarks.

Methods:

- **getTimestamp()**: Returns the timestamp of the benchmark result.
- **setTimestamp(String timestamp)**: Sets a new timestamp.
- **getCpuScoreSingle()**: Returns the CPU benchmark score.
- **setCpuScoreSingle(int cpuScoreSingle)**: Sets a new CPU benchmark score.

- **getCpuScoreMulti()**: Returns the CPU benchmark score.
- **setCpuScoreMulti(int cpuScoreMulti)**: Sets a new CPU benchmark score.
- **getGpuScore()**: Returns the GPU benchmark score.
- **setGpuScore(int gpuScore)**: Sets a new GPU benchmark score.
- **getMemoryLatencyScore()**: Returns the memory benchmark score.
- **setMemoryLatencyScore(int memoryLatencyScore)**: Sets a new memory benchmark score.
- **getMemoryBandwidthScore()**: Returns the memory benchmark score.
- **setMemoryBandwidthScore(int memoryBandwidthScore)**: Sets a new memory benchmark score.

5.2.2 Info Subpackage

5.2.2.1 BatteryInfo.java

This class gathers data such as battery level, charging status, charging source, battery health, voltage, and temperature. This class is useful for monitoring battery status and displaying it within the app.

Attributes:

- **context (Context):** The application context used to access system services.

Constructor:

- **BatteryInfo(Context context):** Initializes the object with the provided context.

Methods:

- **getBatteryDetails():** Returns a formatted string containing detailed battery information.
- **getChargingSource(int chargePlug):** Returns a string representing the charging source based on the provided plug type.
- **getBatteryHealth(int health):** Returns a string representing the battery health based on the provided health status.

5.2.2.2 CPUInfo.java

This class provides methods to retrieve detailed information about the device's CPU. It gathers data such as CPU vendor, model, processor information, core count, clock speed, and supported ABIs. This class is useful for monitoring CPU status and displaying it within the app.

Methods:

- **getCPUDetails():** Returns a formatted string containing detailed CPU information.
- **getCPUVendor():** Returns the CPU vendor name from /proc/cpuinfo.
- **getCPUModel():** Returns the CPU model name from /proc/cpuinfo.
- **getCPUClockSpeed():** Returns the CPU clock speed in MHz from /proc/cpuinfo.
- **getProcessorInfo():** Returns the processor name from /proc/cpuinfo.
- **readFromCpuInfo(String key):** Reads specific information from /proc/cpuinfo based on the provided key.

5.2.2.3 DeviceInfo.java

This class provides methods to retrieve detailed information about the device. It gathers data such as device name, manufacturer, product name, host, build ID, serial number, screen resolution, device type, and kernel version. This class is useful for monitoring device status and displaying it within the app.

Attributes:

- **context (Context):** The application context used to access system services.

Constructor:

- **DeviceInfo(Context context):** Initializes the object with the provided context.

Methods:

- **getDeviceDetails():** Returns a formatted string containing detailed device information.
- **getSerialNumber():** Returns the device serial number.
- **getScreenResolution():** Returns the screen resolution as "width x height".
- **getDeviceType():** Determines the device type (Phone/Tablet) and returns "Tablet" or "Phone".
- **getKernelVersion():** Returns the kernel version from /proc/version.

5.3.2.4 GPUInfo.java

This class provides methods to retrieve detailed information about the device's GPU. It gathers data such as the renderer, vendor, and OpenGL version using EGL and OpenGL. This class is useful for monitoring GPU status and displaying it within the app.

Attributes:

- **renderer (String):** The name of the GPU renderer.
- **vendor (String):** The name of the GPU vendor.
- **glVersion (String):** The version of OpenGL supported by the GPU.

Constructor:

- **GPUInfo():** Initializes the object and retrieves GPU details.

Methods:

- **retrieveGPUInfo():** Retrieves GPU details using EGL and OpenGL.
- **getGPUDetails():** Returns a formatted string containing detailed GPU information.

5.3.2.5 MemoryInfo.java

This class provides methods to retrieve detailed information about the device's memory. It gathers data such as total RAM, available RAM, used RAM, low memory threshold, low memory state, app memory class, and whether the large heap is enabled. This class is useful for monitoring memory status and displaying it within the app.

Attributes:

- **context (Context):** The application context used to access system services.

Constructor:

- **MemoryInfo(Context context):** Initializes the object with the provided context.

Methods:

- **getMemoryDetails():** Returns a formatted string containing detailed memory information, such as RAM, available RAM, used RAM, low memory state etc.

5.3.2.6 OSInfo.java

This class provides methods to retrieve detailed information about the device's operating system. It gathers data such as OS version, API level, release version, codename, security patch level, and boot-loader version. This class is useful for monitoring OS status and displaying it within the app.

Methods:

- **getOSDetails():** Returns a formatted string containing detailed OS information.
- **getCodename(int apiLevel):** Returns the codename of the Android version based on the provided API level.

5.3 Controller Package

5.3.1 BenchmarkController.java

The BenchmarkController class serves as the central control unit for executing benchmark tests, calculating scores, and storing results. It relies on different benchmark classes (such as CpuBenchmark and MemoryBenchmark) to gather metrics and evaluates overall performance scores. Here, only the methods for performing the whole tests are implemented, by instantiating the corresponding class for each performance metric and using its methods to calculcate the score. Overall, this class is designed to return the score for each performance metric.

Attributes:

- **benchmarkResults** (List<BenchmarkResult>): Stores a list of BenchmarkResult objects, each containing benchmark data like CPU, GPU, memory, and battery scores.
- **context** (Context): Holds a reference to the application context, which is required for accessing resources like files and images for testing from the raw package.

Constructor:

- **BenchmarkController(Context context)**: Initializes a new BenchmarkController with the given context and an empty list for benchmark results.

Methods:

- **performCpuBenchmark()**: Runs a series of CPU benchmark tests, including integer calculations, floating-point operations, loop intensity, prime number calculations, matrix multiplication, quick sort, FFT, N-body simulation, SHA-256 hashing, and file compression/decompression. It calculates a total CPU score using a geometric mean of the times for each test and returns a BenchmarkResult with the calculated score. Each individual result can be printed on the console for debugging.
- **performMemoryBenchmark()**: Runs memory benchmarks to measure latency (e.g., cache hits and misses) and bandwidth (e.g., memory copy speed, MOPS, and matrix multiplication). It also includes image processing tasks for real-life simulations. The total memory score is calculated using a geometric mean of latency and bandwidth metrics and returned as a BenchmarkResult.
- **performCpuMultiBenchmark()** : Runs a series of CPU benchmark tests, the same ones as in the Single-Core class, except the workload is dispatched on all available cores through threads.
- **performGpuBenchmark()**: Runs the GPU benchmark on a background thread which waits for the main method to finalize before retrieving the result. We use an additional thread for synchronization since OpenGL uses its own thread (EGL).

- **getBenchmarkResults()**: Returns the list of all stored BenchmarkResult instances.
- **performImageProcessingBenchmark(int resourceId, CpuBenchmark cpuBenchmark)**: A helper method that performs image processing on a specified resource (image file) by calling imageProcessingTest in CpuBenchmark. If the resource is not found, it logs an error.
- **finalizeBenchmark()**: Generates the timestamp corresponding to the overall benchmark and computes the result based on all performance metrics returned by the benchmark methods.
- **computeOverallScore(BenchmarkResult benchmarkResult)**: Normalizes the values corresponding to a result against a threshold and computes the overall score.

5.3.2 Benchmarks subpackage

5.3.2.1 CpuBenchmark.java

The CpuBenchmark class provides a series of synthetic tests to measure the CPU performance of a device. It includes tests for various operations such as integer calculations, floating-point calculations, and matrix multiplications, as well as more complex tasks like quicksort, FFT, and SHA-256 hashing.

Attributes:

- **context** (Context): Holds a reference to the application context, which is used to access resources like files for compression/decompression and images for processing.

Constructor:

- **CpuBenchmark(Context context)**: Initializes the CpuBenchmark with a given context for accessing resources.

Methods:

- **integerCalculationTest()**: Measures the time taken to perform a series of integer calculations using BigInteger. It simulates computational workload through repeated multiplications and additions.
- **floatingPointCalculationTest()**: Measures the time for a series of floating-point operations, involving calculations with square roots and the value of π. This test provides insight into the CPU's floating-point performance.
- **loopIntensityTest()**: Executes nested loops that perform multiplications and additions with BigInteger, simulating a heavy computational workload. The time taken indicates how well the CPU handles intensive loops.
- **primeNumberCalculationTest()**: Checks for prime numbers up to a specified limit to measure CPU performance under load. The isPrime helper method performs the actual prime check.

- **matrixMultiplicationTest()**: Performs matrix multiplication on two 2D integer arrays, measuring the time taken. This test simulates workloads common in machine learning and scientific computations.
- **quickSortTest()**: Sorts a randomly generated array using the QuickSort algorithm. The quickSort helper method implements the sorting algorithm, which tests CPU speed in sorting operations.
- **fftTest()**: Performs the Fast Fourier Transform (FFT) on an array, simulating a workload used in signal processing. The fft helper method implements the FFT algorithm.
- **nBodyTest()**: Simulates an N-body gravitational interaction among particles, calculating forces and updating positions. This test measures CPU performance in handling physics simulations.
- **sha256HashTest()**: Hashes a randomly generated string multiple times using the SHA-256 algorithm. The sha256 helper method generates each hash, while generateRandomString creates random input data.
- **imageProcessingTest(InputStream imageStream)**: Adjusts the brightness of each pixel in a bitmap image, simulating a real-life image processing workload.
- **compressFileTest(int resourceId)**: Compresses a file using GZIP, measuring the time taken. It tests the CPU's performance in file compression tasks.
- **getCompressedDataForResource(int resourceId)**: Compresses a file and returns the compressed data as a byte array. It is used in conjunction with decompression tests.
- **decompressFileTest(byte[] compressedData)**: Decompresses GZIP-compressed data and measures the time taken, testing CPU efficiency in handling decompression.
- **aesEncryptionTest()**: Measures the time taken to encrypt random data using AES (Advanced Encryption Standard). It generates a 256-bit AES key, then performs 1000 encryption operations.
- **aesDecrytpionTest()**: Measures the time taken to decrypt AES-encrypted data. The test decrypts the data 1000 times.
- **matrixExponentiationTest()**: Computes the time taken to perform matrix exponentiation on a randomly generated matrix. This test involve raising a square matrix to a specific power using efficient algorithms.
- **convolutionTest()**: measures the time taken to compute a convolution operation on a large input array using a small kernel. It simulates the process of applying a convolution filte, which is commonly used in image processing and neural networks. The convolution computes a weighted sum of values in the input array based on the kernel.

Helper Methods:

- **isPrime(int n)**: Checks if a given number is prime.
- **quickSort(int[] array, int low, int high)**: Sorts an array using the QuickSort algorithm.
- **fft(double[] real, double[] imag)**: Performs the FFT on real and imaginary components of a signal.
- **reverseBits(int num, int bits)**: Reverses the bits of an integer.
- **generateRandomString(int length)**: Generates a random string of a specified length.
- **sha256(String base)**: Computes the SHA-256 hash of a string.

- **computeConvolutionSingle(int[] input, int[] kernel, int[] output):**
  Computes the convolution of an input array with a given kernel. This helper method iterates through the input array and calculates the weighted sum for each position based on the kernel.
- **generateRandomData(int size):**
  Generates an array of random bytes of the specified size. Used as input data for encryption and other byte-based operations.
- **generateAESKey():**
  Generates a 256-bit AES (Advanced Encryption Standard) key using the Java Cryptography Architecture. The key is used for encryption and decryption tasks.
- **encryptData(byte[] data, SecretKey key):**
  Encrypts the input data using the provided AES key. This method utilizes the `Cipher` class with AES encryption in ECB mode.
- **decryptData(byte[] encryptedData, SecretKey key):**
  Decrypts the input data using the provided AES key. Complements the `encryptData` method and measures decryption performance.
- **matrixExponentiation(int[][] baseMatrix, int exponent):**
  Raises a square matrix to a specified power using an efficient algorithm based on repeated squaring.
- **multiplyMatrices(int[][] matrixA, int[][] matrixB):**
  Multiplies two square matrices and returns the resulting matrix. Used in operations like matrix exponentiation.
- **identityMatrix(int size):**
  Generates an identity matrix of the specified size. Used as the initial value for matrix exponentiation operations.

5.3.2.2 CpuBenchmarkMulti.java

The **CpuBenchmarkMulti** class extends the functionality of **CpuBenchmark** by implementing multi-core parallelism for synthetic CPU tests. It utilizes all available processor cores to distribute tasks, providing a better understanding of the device's multi-threading capabilities.

Attributes:

- **cpuBenchmark (CpuBenchmark):** A reference to the single-threaded benchmark class for accessing utility methods.
- **numCores (int):** The number of available CPU cores on the device.
- **executorService (ExecutorService):** A thread pool that manages the execution of tasks across multiple cores.
- **context (Context):** Holds a reference to the application context, used for accessing resources such as images and files.

Constructor:

- **CpuBenchmarkMulti(CpuBenchmark cpuBenchmark, Context context):**
  Initializes the CpuBenchmarkMulti class by setting up the thread pool and referencing the context for resource access.

Methods:

- **integerCalculationTestMulti():**
  Distributes a series of integer calculations across multiple cores. Each core performs a portion of the workload involving repeated multiplications and additions using BigInteger.
- **floatingPointCalculationTestMulti():**
  Divides a set of floating-point operations, such as square root and π calculations, among all cores. This test evaluates the floating-point computation efficiency of the device.
- **loopIntensityTestMulti():**
  Executes nested loops performing multiplications and additions with BigInteger across multiple threads, simulating an intensive computational workload.
- **primeNumberCalculationTestMulti():**
  Splits the task of finding prime numbers within a specified range among multiple cores. Each thread processes a separate range using the helper method isPrime.
- **matrixMultiplicationTestMulti():**
  Performs matrix multiplication by dividing rows among available threads. This test simulates workloads common in machine learning and scientific computations.
- **quickSortTestMulti():**
  Implements a parallel version of the QuickSort algorithm. The array is divided into sections, and sorting tasks are executed concurrently.
- **fftTestMulti():**
  Distributes the Fast Fourier Transform (FFT) computations across cores. Each thread processes a chunk of data for signal processing simulations.
- **nBodyTestMulti():**
  Simulates N-body gravitational interactions by dividing particles into groups and calculating forces and positions in parallel. This test evaluates the device's ability to handle physics simulations.
- **sha256HashTestMulti():**
  Hashes multiple randomly generated strings using the SHA-256 algorithm. The hashing workload is distributed among threads for multi-core performance analysis.
- **imageProcessingTestMulti(InputStream imageStream):**
  Processes an image by adjusting the brightness of its pixels. The workload is divided into chunks for concurrent execution by all cores.
- **compressFileTestMulti(int resourceId, List<byte[]> compressedChunks):**
  Compresses a file using GZIP by dividing it into chunks, compressing each chunk in parallel, and returning the compressed data.
- **decompressFileTestMulti(List<byte[]> compressedChunks):**
  Decompresses GZIP-compressed chunks concurrently and combines them into the final output.
- **aesEncryptionTestMulti():**
  Encrypts random data using the AES algorithm in parallel. Each thread handles a portion of the encryption workload.
- **aesDecryptionTestMulti():**
  Decrypts AES-encrypted data using parallel processing. Threads divide the decryption workload.
- **matrixExponentiationTestMulti():**
  Performs matrix exponentiation by dividing matrix multiplications among available threads. This test evaluates the performance of multi-threaded linear algebra computations.

Helper Methods:

- **isPrime(int n):** Checks if a given number is prime. Used for the primeNumberCalculationTestMulti method.
- **fftChunk(double[] real, double[] imag, int start, int end):** Processes a portion of the FFT computation for a given range of indices.
- **reverseBits(int num, int bits):** Reverses the binary representation of a number up to the specified number of bits. Utilized in FFT computations.
- **computeConvolutionChunk(int[] input, int[] kernel, int[] output, int start, int end):** Computes the convolution of a segment of the input array with a given kernel. This method is used for the convolutionTestMulti.
- **parallelQuickSort(int[] array, int low, int high, int availableThreads):** Implements a multi-threaded QuickSort algorithm by recursively dividing sorting tasks among threads.
- **partition(int[] array, int low, int high):** Partitions an array for the QuickSort algorithm.
- **generateRandomString(int length):** Creates a random string of the specified length. Used for generating input for the SHA-256 hashing tests.
- **sha256(String base):** Computes the SHA-256 hash of a string and returns it in hexadecimal format.
- **readAllBytes(InputStream inputStream):** Reads all bytes from an input stream. Used for handling files during compression and decompression.
- **generateRandomData(int size):** Generates an array of random bytes for encryption and other operations.
- **generateAESKey():** Creates a 256-bit AES key using Java's cryptography libraries.
- **encryptData(byte[] data, SecretKey key):** Encrypts the given data using AES with the specified key.
- **decryptData(byte[] encryptedData, SecretKey key):** Decrypts AES-encrypted data with the given key.
- **matrixExponentiationMulti(int[][] baseMatrix, int exponent):** Performs matrix exponentiation using multi-core processing.
- **multiplyMatricesMulti(int[][] matrixA, int[][] matrixB):** Multiplies two matrices in parallel. Each thread processes a subset of rows.
- **identityMatrix(int size):** Generates an identity matrix of the specified size. Used in matrix operations.

5.3.2.3 MemoryBenchmark.java

The MemoryBenchmark class provides a range of memory performance tests, covering latency, bandwidth, and cache efficiency. It includes both synthetic and real-life scenarios, such as matrix multiplication and image processing, to provide a comprehensive memory benchmark. The methods return values that represent various memory performance metrics, such as latency in milliseconds, bandwidth in MB/s, and processing speed in GFLOPS, which can be used to assess and compare memory performance.

Attributes:

- **ARRAY_SIZE** (int): Defines the size of the main data array used in latency and cache tests.

- **dataArray** (int[]): An array of integers used for memory tests such as latency and cache hit/miss.
- **random** (Random): A Random object used for generating random indices in tests.
- **context** (Context): Stores the application context, used to access resources for image processing.

Constructor:

- **MemoryBenchmark(Context context)**: Initializes the dataArray with random integers and stores the context for accessing resources.

Methods:

- **latencyTest()**: Measures the average time taken to access a random index in dataArray. This test evaluates memory latency by averaging the time over multiple accesses.
- **bandwidthTest()**: Tests the bandwidth by writing and reading a large data block to/from memory. It calculates the bandwidth by dividing the data size by the total time taken.
- **mopsTest()**: Measures the number of memory operations per second (MOPS) by performing additions in a loop. This test is useful for evaluating computational performance in memory-bound scenarios.
- **memoryCopyTest()**: Measures the time taken to copy a data block from one array to another. It calculates memory copy speed, giving an indication of memory bandwidth.
- **allocationDeallocationTest()**: Measures the time taken for multiple allocation and deallocation operations. The average time for a single allocation/deallocation is calculated.
- **memoryLatencyUnderLoadTest()**: Measures memory latency while another thread performs a heavy load in the background. This test provides an understanding of memory latency under stress.
- **cacheHitTest()**: Measures the time taken to sequentially access elements in dataArray. This test simulates cache hits, where adjacent memory accesses are likely to be in the cache.
- **cacheMissTest()**: Measures the time taken to access elements in dataArray with a stride, simulating cache misses. This test helps understand how the CPU handles non-adjacent memory accesses.
- **cacheHitMissRatio()**: Calculates the ratio between cache hit and cache miss times to provide insight into cache efficiency.
- **matrixMultiplicationTest()**: Performs matrix multiplication on two 2D arrays, calculating gigaflops (GFLOPS) based on the operations performed. This test simulates workloads in fields like machine learning and scientific computing.
- **imageProcessingBrightnessTest()**: Adjusts the brightness of an image by modifying each pixel's RGB values. The test is performed on a sample image to evaluate memory performance in image processing tasks.
- **imageProcessingGrayScaleTest()**: Converts an image to grayscale by modifying the RGB values for each pixel, simulating a real-life memory-intensive task.

5.3.2.4 GpuBenchmark.java

The GpuBenchmark class provides a comprehensive GPU performance test by rendering a 3D model with multiple light sources. It measures the total number of frames rendered over a fixed period, which can be used to assess and compare GPU performance.

Attributes:

- **MAX_LIGHTS (int):** Defines the maximum number of light sources used in the benchmark.
- **context (Context):** Stores the application context, used to access resources.
- **shader (Shader):** The shader program used for rendering.
- **armadilloModel (Model):** The 3D model used in the benchmark.
- **modelMatrix (float[]):** The model transformation matrix.
- **viewMatrix (float[]):** The view transformation matrix.
- **projectionMatrix (float[]):** The projection transformation matrix.
- **normalMatrix (float[]):** The normal transformation matrix.
- **ratio (float):** The aspect ratio of the rendering surface.
- **rotationAngle (float):** The rotation angle of the model.
- **isBenchmarkRunning (boolean):** Indicates whether the benchmark is currently running.
- **startTime (long):** The start time of the benchmark.
- **frameCount (long):** The total number of frames rendered.
- **lightPositions (float[][]):** The positions of the light sources.
- **lightColors (float[][]):** The colors of the light sources.
- **eglContext (EGLContext):** The EGL context used for rendering.
- **eglDisplay (EGLDisplay):** The EGL display used for rendering.
- **eglSurface (EGLSurface)**: The EGL surface used for rendering.

Constructor:

- **GpuBenchmark(Context context):** Initializes the object with the provided context.

Methods:

- **run():** The main method that runs the benchmark, initializing EGL and OpenGL, starting the benchmark, and rendering frames until the benchmark duration is reached.
- **initEGL():** Initializes the EGL context, display, and surface.
- **initOpenGL():** Initializes OpenGL settings, loads shaders and the model, and sets up lighting.
- **drawFrame():** Renders a single frame, updating the model's rotation and swapping the EGL buffers.
- **cleanupEGL():** Cleans up the EGL context, display, and surface after the benchmark is complete.
- **startBenchmark():** Starts the benchmark by setting the start time and resetting the frame count and rotation angle.
- **initializeLighting()**: Initializes the positions and colors of the light sources.

- **getFrameCount():** Returns the total number of frames rendered.
- **loadShaderCode(Context context, String fileName):** Loads shader code from the specified file in the assets directory.

5.3.2.5 *Gputools subpackage*

5.3.2.5.1 Model.java

The Model class is responsible for loading and storing 3D model data, including vertex positions, normals, texture coordinates, and faces. It also provides a method to draw the model using a given shader.

Attributes:

- **positions (List<float[]>)**: A list of vertex positions.
- **normals (List<float[]>):** A list of vertex normals.
- **texCoords (List<float[]>):** A list of texture coordinates.
- **faces (List<int[]>):** A list of faces, each containing indices for positions, normals, and texture coordinates.

Constructor:

- **Model(Context context, int resourceId):** Initializes the object and loads the model data from the specified resource.

Methods:

- **loadModel(Context context, int resourceId):** Loads the model data from the specified resource file.
- **getPositions():** Returns the list of vertex positions.
- **getNormals():** Returns the list of vertex normals.
- **getTexCoords():** Returns the list of texture coordinates.
- **getFaces():** Returns the list of faces.
- **draw(Shader shader):** Draws the model using the specified shader.

5.3.2.5.2 Shader.java

The Shader class is responsible for compiling vertex and fragment shaders, linking them into a shader program, and providing methods to use the program and set uniform variables.

Attributes:

- **programId (int):** The ID of the compiled and linked shader program.

Constructor:

- **Shader(String vertexShaderCode, String fragmentShaderCode):** Compiles the vertex and fragment shaders, links them into a program, and checks for errors.

Methods:

- **use():** Activates the shader program for rendering.
- **getAttribLocation(String name):** Returns the location of an attribute variable in the shader program.
- **getUniformLocation(String name)**: Returns the location of a uniform variable in the shader program.
- **setUniformMatrix(String name, float[] matrix):** Sets a 4x4 matrix uniform variable in the shader program.
- **setUniform3f(String name, float x, float y, float z):** Sets a vec3 uniform variable in the shader program.
- **setUniform4f(String name, float x, float y, float z, float w):** Sets a vec4 uniform variable in the shader program.
- **compileShader(int type, String shaderCode):** Compiles a shader of the given type (vertex or fragment) and returns its ID.

5.3.2.5.3 VertexBuffer.java

The VertexBuffer class is responsible for storing vertex data in a buffer and providing methods to draw the vertices using OpenGL.

Attributes:

- **vertexBuffer (FloatBuffer)**: A buffer that stores the vertex data.
- **vertexCount (int):** The number of vertices in the buffer.

Constructor:

- **VertexBuffer(Model model):** Initializes the vertex buffer with data from the provided model.

Methods:

- **draw(Shader shader):** Draws the vertices using the specified shader.

Testing

6.1 Testing Objectives

The testing phase aimed to validate the application's performance, functionality, and compatibility across various scenarios to ensure that it meets the design and user requirements outlined in the project. Specifically, the objectives were:

- Verify the accuracy and reliability of the benchmark scores for CPU, GPU, and memory.
- Ensure the user interface is intuitive and responsive during interactions.
- Confirm compatibility with Android devices, starting with the Samsung Galaxy A11.
- Identify and resolve any errors, crashes, or performance bottlenecks.

6.2 Device Specifications

The app was deployed and tested primarily on a **Samsung Galaxy A11**. The specifications of this device include:

- Processor: Qualcomm SDM450 Aarch64
- Cores: 8
- OpenGL Version: OpenGL ES 3.2
- RAM: 1428 MB
- API Level: 31
- Screen Resolution: 720 x 1411

6.3 Test Cases

TEST	RESULT	WORKING
Launch the application	App launches without crashes	As expected
Execute CPU benchmark	Benchmark completes successfully and displays results, as well as plots	As expected
Execute GPU benchmark	Benchmark completes successfully and displays result	As expected
Execute Memory benchmark	Benchmark completes successfully and displays results, as well as plots	As expected
Compute overall score based on previous benchmarks	Overall score computed successfully, conditioned such that	As expected

	it can only be calculated after all parameters have been tested	
Navigate to benchmark history	History is updated after a complete set of benchmarks are done and the overall score is computed	As expected
Navigate to device information	All information is available	As expected

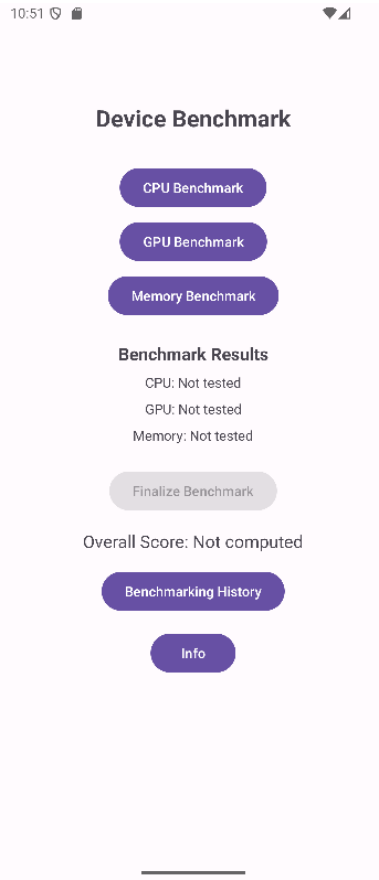


Figure 6.1 Application UI

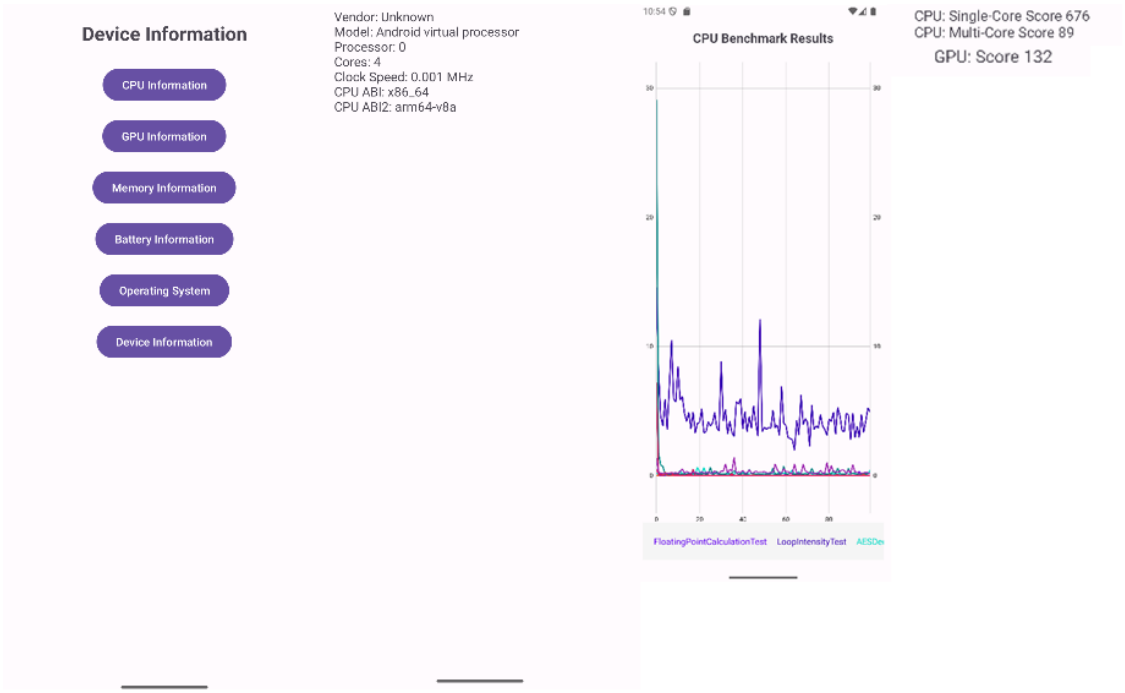


Figure 6.2 Device information, CPU chart and benchmarking results for CPU and GPU

6.4 Limitations

Battery benchmarking could not be validated, since the device is connected via USB, which implies charging. Therefore, a battery drain test could not be executed.

6.5 Future testing plans

- Perform cross-platform testing on iOS once the app is ported
- Incorporate battery testing
- Improve UI/UX by adding more detailed charts and rendering directly on screen
- Explore the possibility of integrating cloud-based analytics for comparative benchmarking across devices
- Implement a data base that would retain all past results even when the application is closed

Conclusions

The development and deployment of the benchmarking application have been a valuable learning experience, combining theoretical knowledge with practical implementation.

The application successfully integrates core features such as single-core and multi-core CPU benchmarks, memory bandwidth and latency tests, and GPU frame rendering performance. The benchmarks provide users with accurate and consistent results.

The interface was designed to be user-friendly, with features like chart visualizations, pop-up dialogs for additional options, and performance summaries that ensure an engaging user experience.

As for challenges, minimum problems were encountered, such as optimizing benchmarks to run efficiently on devices with limited hardware resources and integrating OpenGL for GPU testing on a background thread in order to avoid rendering on display directly.

The application's successful deployment and testing on a real device confirm its readiness for broader usage. With further testing and development, this project has the potential to become a comprehensive benchmarking tool for both casual users and developers seeking insights into device performance.

References

[1] Vanshika Malhotra, "MSP Explained: Here's How Mobile Benchmarks (Geekbench and AnTuTu) Really Work" 2024 .

[2] Calvin Wankhede, "What is benchmarking and why does it matter? Everything you need to know" 2023

[3] Arian Alijani, Trey Price and Neeraj Kumar, "What are the pros and cons of using synthetic vs. real test data for QA testing?" 2024.

[4] Honor, "A Complete Guide on What Is Processor in Mobile?" 2024

[5] Gary Sims, "Apple vs Android RAM management: Who does it better?" 2022

[6] Chetan Gaikwad, "Exploring the Role of Graphics Processing Unit (GPU) in Android Devices" 2023.

[7] Honor, "How Long Does a Phone Battery Last? Your Guide This 2023" 2023.

[8] Jetbrains, "What is cross-platform mobile development?" 2024