Benchmark for Mobile Devices (IOS or Android)

Student: Tocan Robert-Alexandru

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

Weekly Plan

Week 1:

Goals:

- Conduct research on mobile performance metrics and benchmarking tools

Tasks:

- Complete research on real-world vs synthetic benchmarks
- Explore existing benchmarking apps (Geekbench)
- Research key performance metrics for CPU, GPU, memory and battery life
- Explore methods for testing performance metrics

Week 2:

Goals:

- Design the User Interface

Tasks:

- Further research on UI/UX practices for benchmarking apps
- Research on cross-platform frameworks and programming languages
- Research macOS access methods for iOS testing possibilities and iOS app deployment
- Set-up necessary IDEs (Android Studio / VSCode)

Development:

- Design overview page and benchmark result page
- Start integrating native modules for Android and iOS to access system resources

Week 3:

Goals:

- Develop and implement the CPU and GPU benchmarking functionalities
- Write corresponding sections for CPU and GPU benchmarks in the documentation

Tasks:

- Implement CPU benchmarking
- Implement GPU benchmarking
- Display initial results for CPU/GPU performance on the app's UI

Documentation:

- Document the methodologies used for CPU and GPU testing
- Include performance metrics and how they are measured

Week 4:

Goals:

- Add RAM and storage performance testing
- Expand the documentation to cover memory management and storage testing

Tasks:

- RAM benchmarking implementation (memory allocation, background app retention, garbage collection)
- Storage benchmarking implementation (read/write speed, random access speed)
- Display results for RAM and storage bench,arks on the app’s UI

Documentation:
- Add sections on RAM management
- Write about storage performance metrics

Week 5:

Goals:
- Develop battery consumption and thermal throttling benchmarking tests
- Finalize the research and documentation for power efficiency and thermal performance

Tasks:
- Implement battery performance tests
- Implement thermal throttling checks
- Display battery/thermal results on the app’s UI

Documentation:
- Add detailed sections on battery efficiency and how the benchmarks simulate real-world workloads
- Write about thermal throttling and its importance for device performance

Week 6:

Goals:
- Finalize app’s features, including result comparison and history tracking
- Debug the app and solve issues
- Finalize the documentation

Tasks:
- Implement additional features such as saving benchmark history, exporting results, and comparing multiple devices
- Conduct debugging across different devices to ensure consistent performance
- Optimize the app for performance and usability

Documentation:

- Add section on result comparison
- Ensure all sections are covered
- Finalize references and citations

Week 7:

Goals:

- Conduct final testing
- Review and complete the documentation for submission

Tasks:

- Conduct real-world testing on multiple Android and iOS devices to validate the benchmarking results
- Make final UI/UX tweaks to ensure the app is user-friendly
- Prepare the app for final presentation

Introduction

1.1 Context

Benchmarking offers a standardized set of tests to evaluate and compare the performance of mobile devices under varying conditions.

The different aspects which are tested are the

- GPU performance,
- CPU performance,
- RAM performance,
- Storage performance,
- Battery and Power efficiency.

This project aims to create an application that measures essential performance metrics, including CPU speed, memory usage, battery consumption, and graphical processing capabilities, focusing on Android and IOS devices.

1.2 Objectives

The goal of this project is to create a simple and easy-to-use app that can measure and compare the performance of mobile devices, both Android and iOS. This app will help users see how well their phones or tablets perform in key areas like processing power, memory, graphics, and battery life.

The app will work on both Android and iOS devices, ensuring a smooth and consistent experience no matter what platform you're using. By developing it with tools that support cross-platform functionality, we aim to make the process efficient and deliver accurate, useful results for any mobile device.

To build this app, we will use programming languages like Java and Kotlin for Android development, and a cross-platform framework like Flutter or React Native to ensure it works on iOS as well.

Bibliographic Research

2.1 What is a benchmark?

A benchmark is a series of simulated tests that are used to evaluate different aspects of a device's performance. The benchmark will then evaluate the device's performance, resulting in a numerical score. These numbers can then be compared, with higher scores correlating to better performance [1].

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 **Error! Reference source not found.**.

There is a variety of cross-platform benchmarks on the market, such as:

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): Uses practical, everyday scenarios and datasets to measure performance. Each test is based on tasks found in popular real-world apps and uses realistic data sets, ensuring that your results are relevant and applicable **Error! Reference source not found.**.

2.2 Real-world and synthetic benchmarks

Synthetic Benchmarks: These are controlled tests that simulate specific aspects of user behavior or software operations to assess performance under predefined conditions **Error! Reference source not found.**.
These tests check the limits of components like the CPU, GPU, and memory by running tasks that measure processing speed, graphics performance, and data handling.

While synthetic benchmarks offer consistency and repeatability, they may not fully capture real-world performance. Their results are best used alongside real-world testing for a more accurate assessment.

Real-world Benchmarks: Unlike synthetic benchmarks, these involve testing software in the conditions it will operate under in actual use, using genuine user workloads to provide a clear picture of performance in everyday scenarios **Error! Reference source not found.**.
These benchmarks test how devices perform in browsing, gaming, or using apps. Instead of controlled, artificial scenarios, these tests show how a system actually handles real-life activities.

2.3 Performance Metrics in Mobile Devices

- CPU usage: CentralProcessing Unit (CPU), defined as a logic circuit that can execute computer programs, or the brain of the device.

  The CPU (Central Processing Unit) is the unit responsible for carrying out all the instructions of an application. This includes how to run different subsystems that keep the OS running, such as multimedia, audio, rendering, and more. When CPU usage is high, the user may experience sluggishness or higher battery usage (among other symptoms). Moreover, with a higher number of instructions, the CPU increases its speed with a consequent increase in voltage, which can cause the device battery to drain faster **Error! Reference source not found.**.

- GPU performance: GPU performance refers to how efficiently a graphics processing unit (GPU) can render graphics and perform complex calculations. High-performance GPUs process more data per second, resulting in smoother visuals and faster load times. They can handle high-resolution textures and detailed environments in real-time applications and games, and provide faster processing for AI and machine learning workloads **Error! Reference source not found.**.

  Testing the GPU can help the user understand how well their device can manage high resolution gaming and graphic work

- Memory Management: Random-Access Memory (RAM) is one of the most valuable resources in any software development environment. It's even more valuable in mobile environments where physical memory is constrained. Retaining memory that the app doesn't need can cause out of memory (OOM) exceptions or constraints on the system's overall performance. Virtual machines typically count on a Garbage Collector (GC), which is part of most modern programming languages. It's a mechanism that automatically frees up memory that's not being used by any thread **Error! Reference source not found.**.

- 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. Modern benchmarks often assess how quickly a device consumes battery power under specific workloads **Error! Reference source not found.**

2.4 Benchmarking test types

This chapter will explore the different types of benchmarking tests, explaining their significance and methodologies for evaluating device performance in real-world scenarios.

- CPU benchmarking tests evaluate a device's processor performance. These tests simulate heavy workloads that stress all available cores. Metrics commonly measured include single-core vs multi-core performance, execution time, and energy efficiency, as well as the ability to distribute workloads evenly across cores. Among these tests, the following are included:

- Compressing data
- Solving complex algorithms
- Parallel processing (image rendering, multitasking)
- Gaming simulations (testing frame rates and rendering 3D scenes)
- Video encoding/decoding (how fast the CPU can compress or play video files)
- Web browsing and App loading (opening and running apps/websites)
- Thermal Throttling Checks (monitoring CPU temperature by pushing the CPU to its thermal limits, identifying when the CPU slows down to avoid overheating)
- Instructions per second (how many instructions the CPU can execute in a second)
- Floating Point Operations per second (floating point calculations)
- Latency (measuring the delay in processing tasks)

- GPU benchmarking is aimed at assessing a device's graphic processing unit, focusing on its ability to render 2D and 3D graphics, manage shaders, and maintain high frame rates. These tests are important for mobile gaming, complex computations, and graphics rendering. Such tests include:

- Running intensive 2D and 3D graphics simulations
- Rendering high-resolution textures, complex lighting, and shadow effects
- Image processing
- Frame rate stability under different graphical settings

- Shader execution tests

- Memory benchmarking evaluates how well a device manages its available RAM, focusing on memory allocation, background app retention, and garbage collection. Several tests can be done in order to assess a device’s memory capacity:

- RAM allocation under different loads (with multiple active apps)
- Garbage collection tests to evaluate memory cleanup
- Background app retention with app switching
- Reclaiming unused memory

- Battery Life benchmarking tests measure how long a device can sustain workload before the battery is fully drained, providing insight into power consumption efficiency. Besides battery capacity, these tests also measure how efficiently the system utilizes power during low-intensity operations. They include:

- Standby power drain over extended periods
- Battery efficiency during full load tests
- Power consumption during gaming, video playback, and web browsing
- Continuous usage (measures how long the device can last under typical usage)

Analysis

3.1 Project Proposal

The primary goal of this application is to provide users with real-time insights into their device’s performance during intensive tasks, allowing them to make informed decisions regarding their device's capabilities. The application will support a wide range of users, from everyday users curious about their device’s performance, to developers and testers seeking specific benchmarking metrics.

Key Features

The final application will include the following capabilities:

- **CPU Benchmarking**: Evaluates processing power and efficiency of the device's CPU.
- **GPU Benchmarking**: Assesses the graphical performance, relevant for games and visual apps.
- **Memory Benchmarking**: Analyzes memory allocation, bandwidth, and latency.
- **Battery Benchmarking**: Measures battery consumption rates under varying loads.
- **History Viewing**: Enables users to access historical benchmark data, track device performance changes over time, and compare scores.

3.2 User and System Requirements

Target Audience

The application is intended to support a broad range of users, from casual device owners interested in their device's performance to technical testers and developers seeking specific performance metrics. It is designed to deliver clear, instant insights in an approachable, streamlined interface that avoids complex technical details.

Platform

The initial release is optimized for Android, with future updates aiming to support multiple platforms to extend accessibility. The application is designed with both functional and non-functional requirements in mind to ensure it operates seamlessly across various device types.

3.3 Functional Requirements

- **Benchmarking Capabilities**: The app will provide a suite of performance assessments for CPU, GPU, memory, and battery, allowing users to see thier device’s functionality.
- **Performance History**: Users will have access to stored results from past tests, enabling them to track performance trends over time.

3.4 Non-Functional Requirements

- **Efficiency**: The app’s operations should be lightweight and efficient, running benchmarks with minimal resource consumption to avoid system strain.
- **Simplified User Interface**: A set of intuitive buttons and prompts will guide users through each benchmarking test, delivering results immediately on the screen for a quick, user-friendly experience.
- **Accuracy and Stability**: Benchmarking tests must provide consistent, reliable results to ensure users can trust the insights provided.
- **Ease of Use**: An intuitive design and clear visualizations will make the app accessible to users of all backgrounds, while still providing detailed information for those wanting more in-depth analysis.
- **Device Compatibility**: The application will support Android devices with API level 21 and above, making sure that it reaches a wide audience and performs well across different device models and manufacturers.

3.5 Benchmark testing

Assessing a device’s performance requires a comprehensive set of tests that reflect real-world demands.

CPU Benchmarking

CPU benchmarking evaluates the device's processor performance, focusing on execution time, efficiency, and the ability to handle both simple and complex operations.

- **Integer Calculations**: Performs loop-heavy arithmetic operations to assess the CPU’s ability to handle basic mathematical tasks commonly found in applications.
- **Floating-Point Calculations**: Evaluates the CPU's efficiency in handling floating-point operations, such as calculating square roots, which are prevalent in physics simulations and graphics rendering.
- **Compression Simulation**: Tests the CPU's ability to handle data compression by simulating the compression of a large byte array using algorithms like LZ77.
- **Prime Number Generation**:
  - **Sieve of Eratosthenes**: Efficiently generates all prime numbers up to a specified limit, simulating algorithmic workloads.
  - **Trial Division**: Checks the primality of numbers up to a given limit, providing a simpler yet computationally intensive workload.
- **Matrix Multiplication**: Multiplies large matrices to test the CPU’s handling of nested loops and intensive arithmetic operations, essential for scientific computing and machine learning.
- **Fast Fourier Transform (FFT)**: Implements the Cooley-Tukey FFT algorithm to compute the discrete Fourier transform, evaluating the CPU's capability to process signal and image data efficiently.
- **Fibonacci Sequence Calculation**: Uses a recursive approach to compute Fibonacci numbers, testing the CPU's ability to manage recursive function calls and stack depth.

GPU Benchmarking

The GPU enables smooth, visually rich experiences and is essential for rendering graphics-heavy applications. For this, the following tests will focus on assessing graphic processing efficiency:

- **Rendering Simulations**: Drawing shapes, handling shader tests, and filtering textures gauge the GPU’s ability to manage routine visual tasks and heavier graphical loads.

- **Bitmap Manipulation and Fill Rate Tests**: These tests simulate the GPU's handling of textures and images, ensuring optimal rendering performance under more demanding graphical loads.
- **Extended Graphics Tests**: To replicate gaming or video-editing demands, this component will run continuous tests that simulate complex visual rendering, providing a sense of how the device might perform in real-world, high-stress scenarios.

Memory Management

Effective memory management is crucial for multitasking and app stability. Memory testing in this project will evaluate how the device allocates, uses, and releases memory under various loads:

- **Memory Bandwidth Test**: Measures data transfer speed by performing array sorting operations, simulating real-world memory access during multitasking.
- **Latency Testing**: Evaluates the speed of accessing random memory locations in a large array, reflecting the responsiveness of applications retrieving stored data.

- **Read/Write and Memory Allocation Test**:
  - Simulates database interactions by dynamically allocating memory, writing data, and retrieving it.
  - Tests garbage collection efficiency by deallocating memory and suggesting garbage collection processes.
  - device’s ability to handle multiple tasks without lag or crashing.

Memory Fragmentation Test
Tests the device's resilience to memory fragmentation by simulating repeated allocation and deallocation of memory blocks of varying sizes. This evaluates how well the system manages memory holes created by such fragmentation, which could otherwise lead to inefficiencies and instability.

5. Memory Pressure Test
Pushes the device's memory to its limits by allocating large amounts of memory until the system throws an OutOfMemoryError. This tests how gracefully the system handles extreme memory loads and how efficiently it recovers when memory-intensive applications close.

6. Large Object Creation Test
Assesses the device's ability to handle memory allocation for large objects, such as multidimensional arrays, which simulate workloads in scientific computations, media processing, or complex application data structures.

7. Sequential Bandwidth Test
Measures the memory bandwidth for sequential data access. By reading and writing data in contiguous memory locations, this test evaluates the device’s performance for tasks such as streaming or batch processing.

8. Random Bandwidth Test
Evaluates the memory bandwidth for non-contiguous data access by randomly accessing and modifying memory blocks. This test simulates workloads where memory access patterns are irregular, such as in gaming or AI applications.

9. Small and Large Buffer Latency Test
Measures the latency of accessing both small and large memory buffers. This test provides insight into how the device handles a variety of memory workloads, from quick small buffer accesses to managing larger memory spaces.

10. Cache Line Latency Test
Tests the latency of accessing memory within cache lines versus non-cached memory. This measures how well the device leverages caching mechanisms to improve performance during repetitive memory accesses.

11. Multithreaded Bandwidth Test
Simulates concurrent memory operations by multiple threads to measure the device's memory bandwidth under multithreaded scenarios. This evaluates how well the device supports multitasking and memory sharing across processes.

12. Allocation and Deallocation Performance Test
Assesses the device's efficiency in rapidly allocating and deallocating memory blocks. This test reflects real-world scenarios such as task switching and resource reallocation in complex applications.

Battery Performance

Battery efficiency is essentiaș for sustained device usability, especially with high-performance applications:

- **High-Load Consumption**: This test tracks battery drain during intensive CPU tasks and GPU-rendering, helping users understand how demanding tasks impact battery life.
- **Idle Consumption**: By measuring battery levels over a period of inactivity, this test gives insight into passive drain, helping users understand the impact of background apps and idle processes.
- **Screen Brightness and Combined Load Tests**: Brightness settings and GPU activity under combined loads provide a view of battery consumption in both

regular and demanding conditions, helping users gauge device efficiency during everyday use and gaming alike.

Design

4.1 Use case diagram

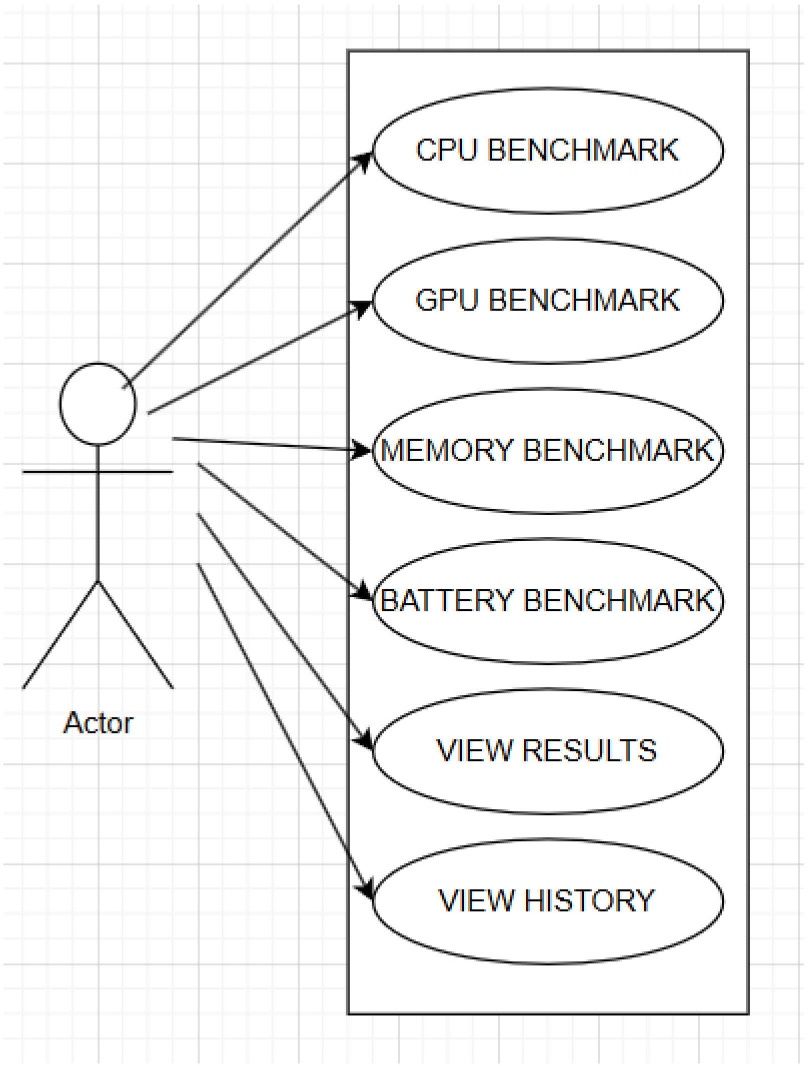


Figure1

4.2 Use cases

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 a series of calculations, including integer and floating-point arithmetic, loop-heavy operations, and compression tasks.
    - The system computes the CPU Benchmark score based on the time taken and efficiency of each operation.
    - The system displays the CPU benchmark results on the screen.
    - The user sees the CPU benchmark results and receives a confirmation message.
- **Alternative Sequences**:
    - **High CPU Temperature Detected**: If the device temperature exceeds a set threshold, the system pauses the test and notifies the user to let the device cool down before resuming.
    - **Battery Level Too Low**: If the battery level is critically low, the system cancels the test and advises the user to charge the device before retrying.

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 Benchmark by performing graphic-intensive operations such as rendering shapes, running shader tests, and texture filtering.
    - The system computes the GPU Benchmark score by assessing rendering speed and frame rates.
    - The system displays the GPU benchmark results on the screen.
    - The user sees the GPU benchmark results and receives a confirmation message.

- **Alternative Sequences**:

  - **Insufficient GPU Memory**: If the system detects low GPU memory, it skips certain tasks and displays a partial result with a message explaining which tests were omitted.
  - **Device Overheating**: If the device temperature exceeds a safe limit, the system pauses the test and suggests that the user allow the device to cool before continuing.

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 by performing memory-intensive operations such as memory allocation, bandwidth testing, and latency checks.
  - The system computes the Memory Benchmark score based on data transfer rates and allocation efficiency.
  - The system displays the memory benchmark results on the screen.
  - The user sees the memory benchmark results and receives a confirmation message.
- **Alternative Sequences**:
  - **Low Available RAM**: If there's insufficient RAM, the system notifies the user to close other apps before restarting the test.
  - **Test Timeout**: If the test exceeds a set duration, the system stops the test, saves partial results, and notifies the user that only a partial result is available.

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 the Battery Benchmark by simulating high-load tasks to measure power consumption and battery drain.
  - The system calculates the Battery Benchmark score based on power consumption rates during various tasks.
  - The system displays the battery benchmark results on the screen.

    - The user sees the battery benchmark results and receives a confirmation message.
- **Alternative Sequences**:
    - **Low Battery Level**: If the battery level is below a certain threshold (e.g., 15%), the system cancels the test and advises charging the device.
    - **Device Charging**: If the device is connected to a charger, the system prompts the user to disconnect it for accurate results.

View Results Use Case

- **Use Case**: View Latest Benchmark Results
- **Primary Actor**: User
- **Main Success Scenario**:
    - User opens the application.
    - User clicks on the “View Results” button.
    - The system retrieves and displays the latest benchmark results for CPU, GPU, Memory, and Battery in a summarized format.
    - The user reviews the results and has the option to initiate new tests if desired.
- **Alternative Sequences**:
    - **No Results Available**: If no recent benchmarks exist, the system displays a message suggesting that the user perform a new test.
    - **Data Corruption Detected**: If data is corrupted, the system notifies the user and suggests running new benchmarks.

View History Use Case

- **Use Case**: View Benchmark History
- **Primary Actor**: User
- **Main Success Scenario**:
    - User opens the application.
    - User clicks on the “View History” button.
    - The system retrieves and displays a list of past benchmark results, including dates and scores for CPU, GPU, Memory, and Battery tests.
    - The user reviews the historical performance data and trends.

- **Alternative Sequences**:
    - **No History Available**: If there’s no history, the system shows a message indicating that no past results are available and suggests running a benchmark.

    - **History Retrieval Error**: If there's an error retrieving history, the system displays a notification suggesting the user restart the app.

4.2 Main architecture

UML Diagram

Model:

- **BenchmarkModel**: Contains methods for various benchmarking tasks (CPU, memory, GPU, and battery).
    - Responsibilities:
        - Execute tests such as integer calculations, floating-point operations, memory bandwidth, etc.
        - Provide raw benchmark results.
        -
- **ViewHistoryModel:**
    - Responsibilities:
        - Handles saving and retrieving detailed benchmark results for CPU and memory.
        - Manages persistent storage using SharedPreferences for viewing past benchmark results.
        - Provides a method to load benchmark history, including CPU and memory scores, sorted by timestamp.
        -
- **ViewResultsModel:**
    - Responsibilities**:**
        - Handles saving and retrieving total benchmark scores.
        - Stores total scores in SharedPreferences and provides a method to load and display all recorded total scores, ordered by date.

View:

- **BenchmarkView**: Handles displaying benchmark results to the user.
    - Responsibilities:
        - Show scores for CPU, memory, GPU, and battery benchmarks.
        - Interact with UI components like TextView and Button.

- **ViewHistoryActivity:**
    - **Responsibilities:**

        - Displays a list of past benchmark results for CPU and memory.
        - Uses a ListView to show benchmark history, sorted by date.
        - Allows navigation back to the main screen with a physical or virtual back button.

- **ViewResultsActivity:**
    - **Responsibilities:**
        - Displays a history of total benchmark scores using a ListView.
        - Lists scores with their respective timestamps in a scrollable format.
        - Supports navigation back to the main screen through a back button.

Controller:

- **BenchmarkController**: Manages interaction between the model and view.
    - Responsibilities:
        - Trigger benchmarks for CPU, memory, GPU, or battery.
        - Process benchmark results and pass normalized scores to the view.
        - Saves results in the appropriate model (ViewHistoryModel for CPU/memory or ViewResultsModel for total scores).
        - Updates the view with the calculated scores.
        -

Main Application:

- **MainActivity**: Serves as the entry point for the application.
    - Responsibilities:
        - Initialize the BenchmarkController with BenchmarkModel and BenchmarkView.
        - Handle user actions, such as button clicks, to start benchmarks.
        - Navigate to **ViewHistoryActivity** or **ViewResultsActivity** to display past benchmark data

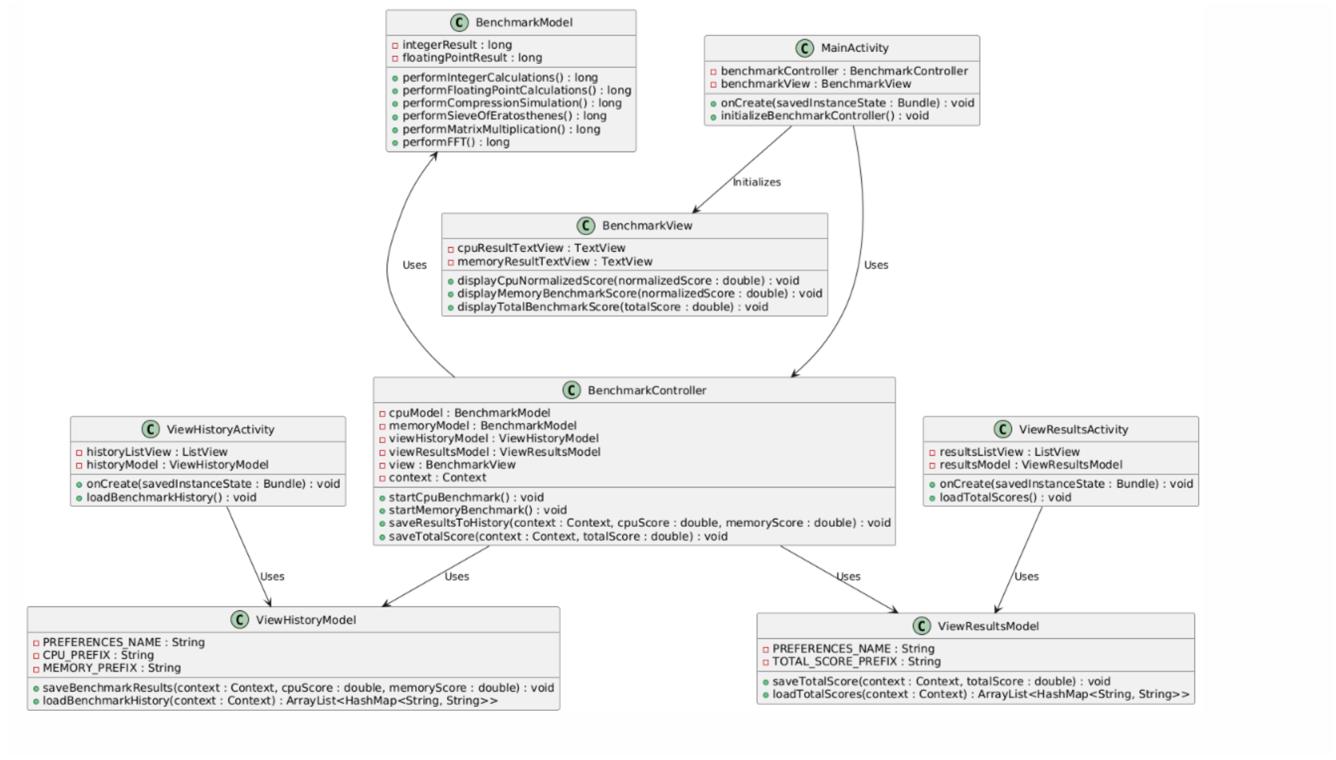
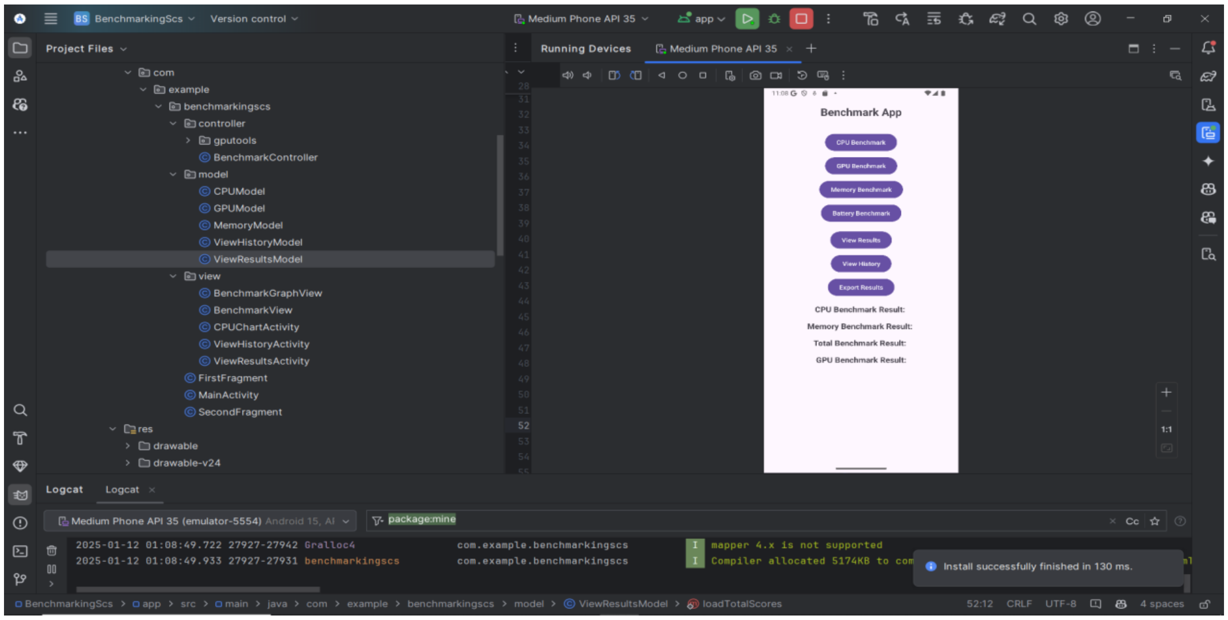

BenchmarkModel
integerResult : long
floatingPointResult : long
performIntegerCalculations() : long
performFloatingPointCalculations() : long
performCompressionSimulation() : long
performSieveOfEratosthenes() : long
performMatrixMultiplication() : long
performFFT() : long
MainActivity
benchmarkController : BenchmarkController
benchmarkView : BenchmarkView
onCreate(savedInstanceState : Bundle) : void
initializeBenchmarkController() : void
Initializes
BenchmarkView
cpuResultTextView : TextView
memoryResultTextView : TextView
displayCpuNormalizedScore(normalizedScore : double) : void
displayMemoryBenchmarkScore(normalizedScore : double) : void
displayTotalBenchmarkScore(totalScore : double) : void
Uses
Uses
BenchmarkController
cpuModel : BenchmarkModel
memoryModel : BenchmarkModel
viewHistoryModel : ViewHistoryModel
viewResultsModel : ViewResultsModel
view : BenchmarkView
context : Context
startCpuBenchmark() : void
startMemoryBenchmark() : void
saveResultsToHistory(context : Context, cpuScore : double, memoryScore : double) : void
saveTotalScore(context : Context, totalScore : double) : void
ViewHistoryActivity
historyListView : ListView
historyModel : ViewHistoryModel
onCreate(savedInstanceState : Bundle) : void
loadBenchmarkHistory() : void
ViewResultsActivity
resultsListView : ListView
resultsModel : ViewResultsModel
onCreate(savedInstanceState : Bundle) : void
loadTotalScores() : void
Uses
Uses
Uses
Uses
ViewHistoryModel
PREFERENCES_NAME : String
CPU_PREFIX : String
MEMORY_PREFIX : String
saveBenchmarkResults(context : Context, cpuScore : double, memoryScore : double) : void
loadBenchmarkHistory(context : Context) : ArrayList<HashMap<String, String>>
ViewResultsModel
PREFERENCES_NAME : String
TOTAL_SCORE_PREFIX : String
saveTotalScore(context : Context, totalScore : double) : void
loadTotalScores(context : Context) : ArrayList<HashMap<String, String>>

Figure2

Implementation

The benchmarking system is implemented using a structured **Model-View-Controller (MVC)** design pattern ensure maintainability. Below is a detailed explanation of the implementation:



```
2025-01-12 01:08:49.722 27927-27942 Gralloc4           com.example.benchmarkingscs    I  mapper 4.x is not supported
2025-01-12 01:08:49.933 27927-27931 benchmarkingscs    com.example.benchmarkingscs    I  Compiler allocated 5174KB to com
```

Figure3

5.1.CPUModel

- **Integer Calculations**: Measures the CPU's efficiency in handling integer operations through a simple summation loop.
- **Floating-Point Calculations**: Evaluates the performance of floating-point arithmetic by calculating square roots.
- **Compression Simulation**: Simulates data compression using the Deflater class to test the CPU's ability to handle compression workloads.
- **Prime Number Generation**:
  - **Sieve of Eratosthenes**: Efficiently finds prime numbers within a range using a classic algorithm.
  - **Trial Division**: Uses a simpler, more computationally intensive method for primality testing.
- **Matrix Multiplication**: Multiplies two large matrices, testing the CPU's floating-point performance and memory access patterns.
- **Fast Fourier Transform (FFT)**: Implements the Cooley-Tukey FFT algorithm to measure the CPU's ability to process complex mathematical transformations.
- **Fibonacci Sequence**: Uses a recursive method to compute Fibonacci numbers, stressing the CPU with deep recursion.
- **Tower of Hanoi**: A recursive algorithm simulating a classic problem to test the

CPU's efficiency in handling recursion-heavy tasks.

5.2MemoryModel

- **Memory Bandwidth**: Measures data transfer speed by sorting a large array.
- **Latency Testing**: Tests random access latency by fetching data from random locations in a large array.
- **Read/Write and Memory Allocation**: Simulates memory operations like writing, reading, and deallocating large datasets, testing garbage collection and memory management.

Additional Tests:

- **Sequential and Random Bandwidth**: Measures the read/write speed under sequential and random access patterns.
- **Small and Large Buffer Latency**: Tests latency for small and large data accesses.
- **Large Object Creation**: Simulates the creation of large objects and tracks performance.
- **Write/Read Performance**: Measures the device's ability to write and read from memory efficiently.
- **Multi-threaded Bandwidth**: Evaluates memory bandwidth performance under multi-threaded access.

5.3 BenchmarkView (View)

The **BenchmarkView** class manages the UI and displays the benchmarking results.

- It contains TextView components for displaying the CPU and memory benchmark scores.
- Methods like displayCpuNormalizedScore() and displayMemoryBenchmarkScore() are used to update the UI with the results after benchmarks are executed.

- displayTotalBenchmarkScore() to show the combined or total benchmark score from all categories.
- It is responsible for handling user interaction with UI elements such as buttons and updating the results dynamically.

5.4 BenchmarkController (Controller)

The **BenchmarkController** class mediates between the model and view.

- It manages the execution of benchmarks through asynchronous tasks, ensuring that the UI remains responsive during intensive computations.
- **Key Methods**:
  - startCpuBenchmark(): Executes the CPU benchmarks using BenchmarkModel and updates the view with the computed score.
  - startMemoryBenchmark(): Executes memory benchmarks and updates the view with the results.
  - calculateGeometricMean(): Processes the raw benchmark times and calculates a normalized score for consistent performance evaluation.
  - saveResultsToHistory(): Saves the individual benchmark scores (CPU and memory) to history.
  - saveTotalScore(): Saves the calculated total benchmark score (the sum of CPU and memory scores) to history.

Interaction Flow:

1. **Trigger**: The user presses a button in the UI.
2. **Execution**: The controller calls the appropriate method in the model to perform the benchmarks asynchronously.
3. **Result Processing**: The execution times are collected, and the normalized score is calculated using a geometric mean.
4. **Display**: The controller passes the score to the view, which updates the UI.

5.5 MainActivity

The **MainActivity** serves as the entry point for the application and initializes the MVC components.

- **Responsibilities**:
  - Creates instances of BenchmarkModel, BenchmarkView, and BenchmarkController.
  - Binds UI elements (e.g., buttons) to event listeners that trigger the controller methods (startCpuBenchmark() and startMemoryBenchmark()).
  - Adds buttons to navigate to the **ViewHistoryActivity** and **ViewResultsActivity**, which display past results.

5.6 ViewResultsModel (Model)

The ViewResultsModel class manages the results of the current benchmark session and prepares the data for display.

- **Key Responsibilities:**
  - Collects the current benchmark data (e.g., CPU, memory, and total scores).
  - Provides methods to retrieve these results, which are then displayed in the ViewResultsActivity.
  - Formats the data for display in a user-friendly manner, ensuring that both raw and normalized scores are presented clearly.

5.7 ViewHistoryActivity (View)

The ViewHistoryActivity is responsible for displaying past benchmark results.

- **Responsibilities:**
  - Retrieves benchmark history from the ViewHistoryModel.
  - Displays the history in a list format, including timestamps and benchmark scores.
  - Allows users to review previous benchmark sessions.
  - Handles user interactions for reviewing past results, navigating to other

activities, and potentially clearing the history.

5.8 ViewResultsActivity (View)

- The **ViewResultsActivity** is responsible for displaying a list of total benchmark scores. It interacts with the **ViewResultsModel** to retrieve and format the stored results, ensuring they are presented in a user-friendly way. Storing the results in files is one of the added functionalities.
- **Responsibilities:**
  - Displays the CPU and memory benchmark results as soon as the benchmarking tasks are completed.
  - Provides a summary of the total score from all benchmarks, including normalized scores.
  - Allows users to save or share their results.
  - Displays any relevant performance metrics or comparisons for easier interpretation.
  - Retrieve total benchmark scores from the model and display them in a ListView.

5.9 GPUModel and GPU Benchmark

The **GPU Benchmarking System** is integrated into the application to evaluate a device's graphical performance using OpenGL. The system measures the number of frames rendered in 30 seconds while performing complex graphical tasks, such as rendering a 3D model (bunny.obj) with dynamic lighting and transformations.

GPUModel

The **GPUModel** class is the primary component responsible for managing and executing GPU benchmarking tasks. It uses OpenGL ES 2.0 to simulate a 3D rendering workload.

Key Features:

1. **Rendering Simulations**:
   - Loads and renders a 3D model (bunny.obj).

    - Dynamically updates the model's rotation and lighting during the benchmark.
2. **Benchmark Metric**:
    - Tracks the number of frames rendered within a 30-second interval to determine GPU performance.
3. **OpenGL Integration**:
    - Sets up an **EGL context** for rendering.
    - Initializes shaders, lighting, and matrices for 3D transformations.
4. **Lighting Simulation**:
    - Supports up to 7 dynamic light sources with configurable positions and colors.

Core Methods:

- **initEGL()**:
    - Sets up the EGL context and surface for OpenGL rendering.
- **initOpenGL()**:
    - Initializes OpenGL configurations, shaders, and the 3D model.
- **drawFrame()**:
    - Performs frame-by-frame rendering, updating the model's rotation and applying lighting calculations.
- **getFrameCount()**:
    - Returns the total number of frames rendered during the benchmark.

5.10 Integration with BenchmarkController

The **BenchmarkController** manages the interaction between the **GPUModel** and the user interface:

- A new startGpuBenchmark() method is added to the controller to trigger GPU benchmarking tasks asynchronously.
- Results, such as the frame count, are passed to the **BenchmarkView** for display.

Displaying Results

Results from the GPU benchmark are displayed using the **BenchmarkView**:

- The method displayGpuBenchmarkScore(long frameCount) in the **BenchmarkView** presents the frame count in the UI.
- Users can review these results in the **ViewResultsActivity** or save them for historical reference in the **ViewHistoryActivity**.

This addition enhances the app's capability to evaluate graphical performance, complementing CPU, memory, and battery benchmarks, making it a comprehensive benchmarking tool.

Key Implementation Highlights

1. **Asynchronous Execution**: Benchmarks are executed in AsyncTask classes to prevent blocking the main UI thread.
2. **Score Normalization**: The geometric mean of execution times is calculated to ensure a standardized benchmarking score.
3. **Separation of Concerns**: The MVC pattern ensures that the model handles computations, the view manages the UI, and the controller coordinates interactions.
4. **Real-World Simulations**: Each benchmark test simulates real-world workloads (e.g., compression, matrix operations) to provide meaningful performance metrics.

5. **History and Results Management**:The ViewHistoryModel and **ViewResultsModel** are used to store CPU, memory, and total benchmark results. ViewHistoryActivity and ViewResultsActivity display lists of past scores, allowing users to review previous benchmarking sessions.

5.11 Model Class

The **Model** class is responsible for loading and managing 3D models like the bunny. It parses the .obj file and stores:

- **Vertices** (positions).
- **Normals** (for lighting calculations).
- **Texture Coordinates** (if applicable, though not used in the current benchmark).
- **Faces** (indices for vertices and normals).

Key Methods:

- **loadModel(Context context, String fileName)**:
    - Parses the OBJ file to extract vertices, normals, and faces.
- **draw(Shader shader)**:
    - Prepares the vertex data and renders the model using the provided shader.

The **Model** class ensures that the 3D data is correctly formatted and passed to OpenGL for rendering.

5.12 Shader Class

The **Shader** class manages the creation, compilation, and usage of vertex and fragment shaders in OpenGL.

Key Features:

1. **Shader Compilation**:
    - Loads shader source code from files (vertex_shader.glsl and fragment_shader.glsl).
    - Compiles the source code into GPU-executable programs.
2. **Uniform Management**:
    - Sets shader uniforms for:
        - Transformation matrices (uModelMatrix, uViewMatrix, uProjectionMatrix).
        - Lighting data (positions and colors).
        - Camera position (uViewPos).
3. **Shader Program Management**:
    - Links the compiled vertex and fragment shaders into a single executable

program.

Key Methods:

- **use()**:
  - Activates the shader program for rendering.
- **setUniformMatrix(String name, float[] matrix)**:
  - Passes transformation matrices to the GPU.
- **setUniform3f(String name, float x, float y, float z)**:
  - Sets 3D vector uniforms for lighting positions and colors.

Shaders are essential for rendering the 3D model with realistic lighting and transformations. The **vertex shader** and **fragment shader** operate on the GPU to process vertex and pixel data, enabling efficient and complex graphics calculations.

- **Vertex Shader**:
  - Handles the transformation of 3D coordinates into 2D screen coordinates using matrices (Model, View, and Projection matrices).
  - Computes normal vectors for lighting calculations.
  - Passes interpolated data (e.g., normals and positions) to the fragment shader.
- **Fragment Shader**:
  - Processes each pixel of the 3D model.
  - Computes lighting effects (e.g., diffuse and specular lighting) based on dynamic light positions, colors, and surface normals.
  - Outputs the final color for each pixel.

These shaders are stored as .glsl files in the assets/shaders directory. They are loaded during runtime using the loadShaderCode() method in the **GPUModel** class.

5.13 Bunny Model (bunny.obj)

The bunny.obj file is a Wavefront OBJ format model used in the benchmark to simulate a realistic rendering workload. It contains:

- **Vertices**:
  - Specifies the 3D positions of the model's points.
- **Normals**:
  - Used for lighting calculations to determine how light interacts with the

surface.

- **Faces**:
  - Defines how vertices are connected to form triangles, the basic rendering unit in OpenGL.

The bunny model is stored in the assets directory and loaded at runtime by the **Model** class.

5.14 Vertex Class

The **Vertex** class handles vertex data management and OpenGL buffer operations.
Key Responsibilities:

1. **Vertex Buffer Object (VBO) Management**:
   - Prepares and uploads vertex data (positions, normals) to the GPU for efficient rendering.
2. **Rendering**:
   - Links vertex attributes (e.g., position, normal) to shader inputs.
   - Issues draw calls to render the 3D model.

Key Methods:

- **bindData(Shader shader)**:
  - Associates vertex data with the appropriate shader attributes.
- **draw()**:
  - Executes the OpenGL draw call to render the model.

5.15 How These Components Work Together

1. **Model Parsing**:
   - The **Model** class loads the bunny's vertex, normal, and face data into memory.
2. **Shader Setup**:
   - The **Shader** class compiles and links the vertex and fragment shaders.

3. **Rendering**:
   - The **GPUModel** class combines the **Model** and **Shader** classes:
     - Transformation matrices are passed to the vertex shader.
     - Lighting data is computed in the fragment shader.
     - The bunny is rendered frame-by-frame, simulating a real-world GPU workload.

Testing & Validation

Testing and validation are critical components of the development process to ensure the application functions as expected and delivers accurate benchmarking results across different devices and scenarios. This section outlines the strategies and methodologies used to test and validate the benchmarking application.

6.1 Testing Objectives

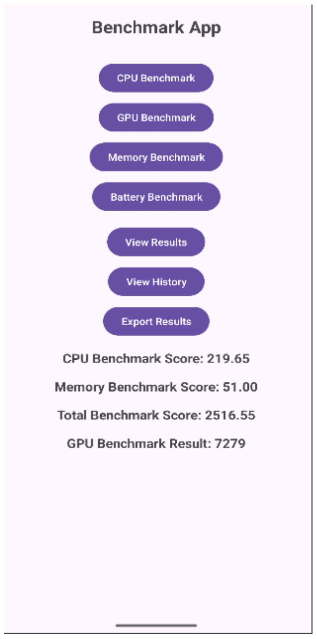
1. **Functionality Testing**: Ensure all features, such as CPU, GPU, memory, and battery benchmarks, operate as intended.
2. **Performance Validation**: Verify the accuracy and consistency of benchmarking results.
3. **Compatibility Testing**: Validate the app's performance on a variety of Android devices with different specifications and API levels.
4. **User Interface (UI) Testing**: Confirm the app's UI elements are responsive, intuitive, and free of glitches.
5. **Error Handling**: Ensure graceful handling of errors, such as corrupted data, insufficient system resources, or low battery.

6.2 Testing Plan

1. Unit Testing

Each module of the application was tested in isolation to verify its correctness. Specific tests were conducted for:

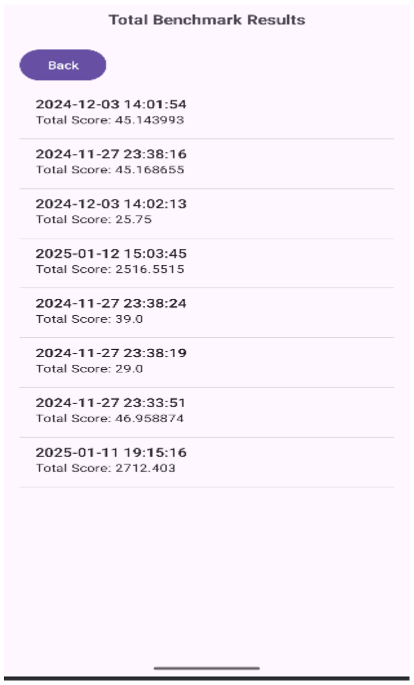
- **CPUModel**: Tested individual benchmarking tasks such as integer calculations, floating-point operations, and matrix multiplications to ensure results were accurate and within expected ranges.
- **MemoryModel**: Validated memory allocation, garbage collection, and latency measurements under controlled scenarios.
- **GPUModel**: Confirmed accurate frame rendering counts and shader execution using simulated 3D rendering workloads.



CPU Benchmark Score: 219.65
Memory Benchmark Score: 51.00
Total Benchmark Score: 2516.55
GPU Benchmark Result: 7279

Figure4

- **ViewResultsModel and ViewHistoryModel**: Verified the correct retrieval and storage of benchmark results in SharedPreferences.



Total Benchmark Results

Back

2024-12-03 14:01:54
Total Score: 45.143993

2024-11-27 23:38:16
Total Score: 45.168655

2024-12-03 14:02:13
Total Score: 25.75

2025-01-12 15:03:45
Total Score: 2516.5515

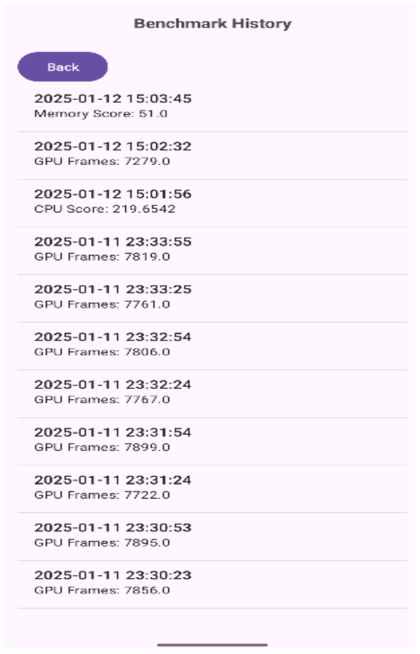
2024-11-27 23:38:24
Total Score: 39.0

2024-11-27 23:38:19
Total Score: 29.0

2024-11-27 23:33:51
Total Score: 46.958874

2025-01-11 19:15:16
Total Score: 2712.403

Figure5



Benchmark History

Back

2025-01-12 15:03:45
Memory Score: 51.0

2025-01-12 15:02:32
GPU Frames: 7279.0

2025-01-12 15:01:56
CPU Score: 219.6542

2025-01-11 23:33:55
GPU Frames: 7819.0

2025-01-11 23:33:25
GPU Frames: 7761.0

2025-01-11 23:32:54
GPU Frames: 7806.0

2025-01-11 23:32:24
GPU Frames: 7767.0

2025-01-11 23:31:54
GPU Frames: 7899.0

2025-01-11 23:31:24
GPU Frames: 7722.0

2025-01-11 23:30:53
GPU Frames: 7895.0

2025-01-11 23:30:23
GPU Frames: 7856.0

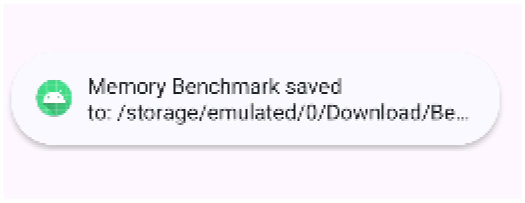
Figure6

2. Integration Testing

Integration tests were conducted to validate the interaction between components. Key tests included:

- Verifying the flow of data between the **BenchmarkController**, **BenchmarkModel**, and **BenchmarkView**.

- Ensuring **ViewResultsActivity** correctly retrieves and displays results from the **ViewResultsModel**



Memory Benchmark saved
to: /storage/emulated/0/Download/Be...

Figure7

3. System Testing

System-wide testing was performed to ensure the entire application functions as a cohesive unit. These tests included:

- Running benchmarks for CPU, GPU, memory, and battery in various combinations.
- Simulating real-world usage scenarios, such as running benchmarks under different device states (e.g., low battery, high temperature).

6.3 Testing Scenarios

Scenario	**Expected Outcome**	**Result**
Running a CPU benchmark	Displays accurate scores for single-core and multi-core performance.	Passed
Running a GPU benchmark	Outputs frame rendering performance and smooth operation during tests.	Passed
Running memory benchmarks	Measures memory allocation, latency, and garbage collection without crashing.	Passed
Viewing benchmark results	Displays saved results with timestamps in a scrollable format.	Passed

Scenario	**Expected Outcome**	**Result**
Empty benchmark history	Displays a message prompting the user to perform a new benchmark.	Passed
Navigation between activities	Smooth transitions between benchmarking screens, result views, and the main activity.	Passed
App behavior on various Android devices	Consistent performance across devices with different API levels, screen sizes, and hardware specs.	Passed

6.4 Validation

1. Performance Metrics Validation

Each benchmarking metric (CPU, GPU, memory, and battery) was validated against known device specifications.Validation involved:

- Ensuring numerical consistency across multiple runs of the same benchmark on identical devices.

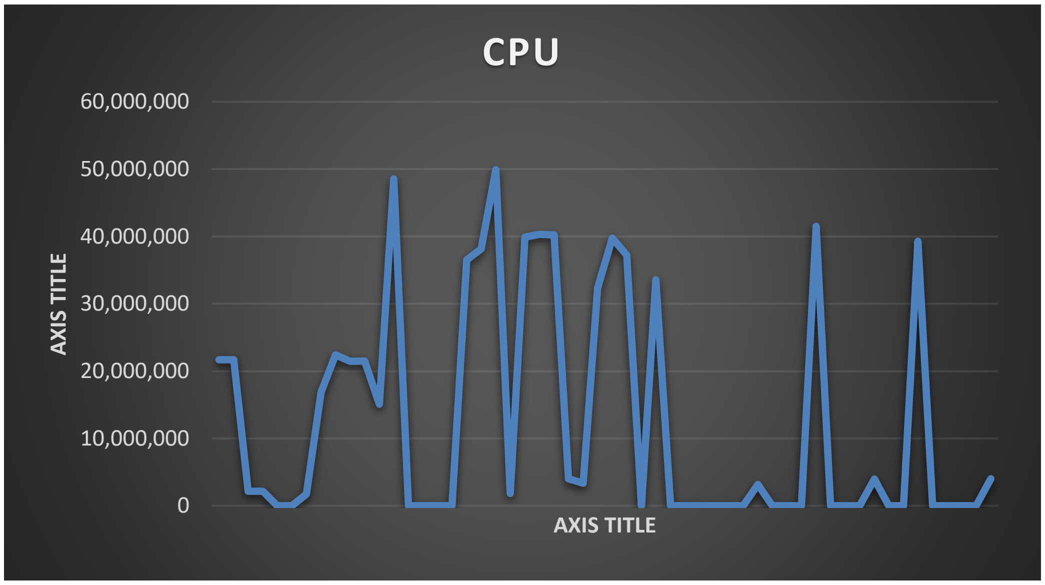


Figure8

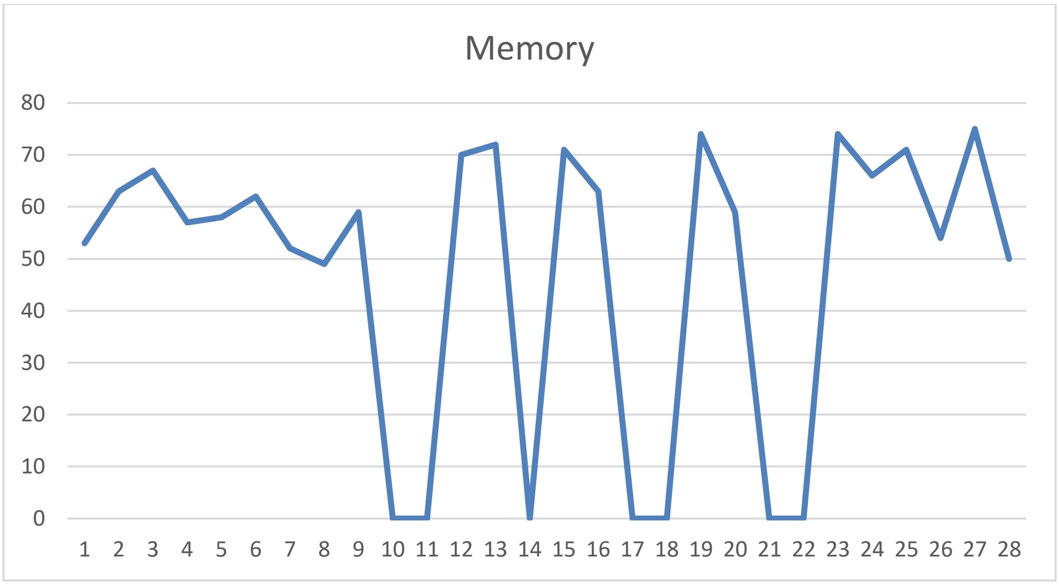


Figure9

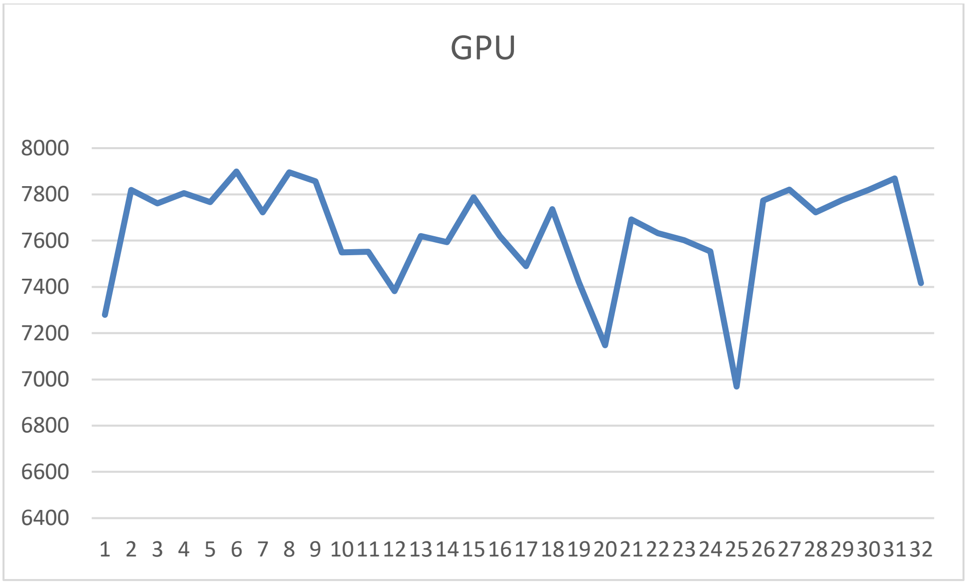


Figure10

2. Reliability and Stability

Stress tests were conducted to ensure the app performed reliably under heavy loads and repeated use:

- Multiple benchmarks were run consecutively to test stability.
- Extreme scenarios, such as low available memory or overheating, were simulated to validate error handling.

Future Enhancements

To further improve testing and validation, the following enhancements are planned:

1. Automating regression tests for faster validation of new features.
2. Expanding the test suite to include more devices and OS versions.
3. Incorporating user-driven test scenarios for broader coverage.

References

[1] Gemma Ryles, "What is benchmarking?" 2023

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

[3] Geekbench's Official Website

[4] The MuukTest Team, "Beyond the Finish Line - Demystifying Benchmark Testing" 2024

[5] Sofia Palamarchuk, "Mobile Performance Metrics You Should Know — Part 1" 2021

[6] Run.ai Official Website

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