# Program for testing the performance of a PC

***Student: Denis Chipirliu***

**Structure of Computer Systems Project**

***Technical University of Cluj-Napoca***

***2025***

# Contents

# *Introduction*

## 1.1 Context

This project is a C application designed to test and analyze the performance of a PC by measuring various hardware parameters and computational capabilities. By focusing on critical factors such as the type of processor, its operating frequency, memory size, data transfer speeds, and execution time of arithmetic and logical operations, the program provides a comprehensive overview of the system's performance.

By leveraging C for development, the program ensures low-level access to the system's resources, maximizing performance during testing and providing precise measurements of hardware capabilities.

## 1.2 Objectives

The key objectives of this C application for testing PC performance are:

1. **Processor Evaluation**:
    - Identify the type and model of the processor.
    - Measure the processor's clock frequency for performance analysis.
2. **Memory Analysis**:
    - Detect and report the size of available system memory (RAM).
3. **Data Transfer Benchmarking**:
    - Measure data transfer speeds between memory, storage, and CPU by benchmarking the transfer of data blocks.
4. **Execution of Arithmetic and Logical Operations**:
    - Benchmark the time it takes to perform arithmetic and logical operations using an encryption algorithm to evaluate CPU computational efficiency.

# *Bibliographic Research*

## 2.1 Computer Performance Testing and Benchmarking

The performance of a computer can be measured in different ways such as throughput, latency or efficiency. Throughput refers to the number of tasks completed in each time frame, while latency measures the time taken to complete a single task. Efficiency measures the ratio between the tasks performed and the resources used for them.

There are multiple ways of benchmarking a computer:

- Kernels
- Toy programs
- Synthetic benchmarking

Unfortunately, these methods are no longer widely used; instead, benchmarking suites such as SPEC have become standard. A benchmarking set provides a more accurate measure of performance because the processor is tested on multiple mini programs that give a more accurate and complex view of the performance of a CPU and it is also made in such a way that it minimizes the delay caused by I/O devices.

In practice, benchmarks provide a tool for the customer to choose between different products, the best one. But for the benchmark to be relevant for the user, it needs to perform real programs that the user does daily.

## 2.2 Processor architecture and Operation

Processor architecture is key to determining how efficiently a computer handles tasks. Factors such as the number of cores, clock speed, and cache size have a significant impact on performance. Modern processors utilize multiple cores to execute instructions in parallel, improving multitasking and throughput. Additionally, technologies like pipelining, branch prediction, and out-of-order execution further optimize performance by reducing delays and increasing instruction processing speed. Cache memory, particularly multi-level cache systems, plays an essential role in reducing memory access times, which is critical for tasks involving frequent data access, such as encryption.

## 2.3 Memory management and Performance

Memory performance is crucial in determining a computer's ability to handle large data sets and maintain high processing speeds. Key factors include memory size, bandwidth, and latency. Efficient memory hierarchies, such as the combination of cache, RAM, and virtual memory, allow for faster access to frequently used data, minimizing delays. Data transfer speed, or memory bandwidth, reflects how quickly data can be moved between memory and the processor, which becomes critical in tasks like encryption, where blocks of data are processed continuously. High bandwidth and low latency ensure optimal performance for both arithmetic operations and complex algorithms that rely heavily on memory access.

## 2.4 Encryption algorithms and Computational Performance

Encryption algorithms are essential for securing data by converting plaintext into ciphertext, thereby protecting sensitive information from unauthorized access. Their performance is critical, particularly in environments that demand fast and efficient data processing. Symmetric encryption algorithms, which utilize the same key for both encryption and decryption, are especially valued for their speed and effectiveness.

Among these symmetric-key algorithms, Blowfish, designed by Bruce Schneier, stands out for its exceptional performance and security. Operating on 64-bit blocks, Blowfish supports variable-length keys ranging from 32 bits to 448 bits, providing flexibility in its application. The algorithm emphasizes efficiency, making it ideal for scenarios where encryption speed is paramount. It employs a sophisticated key schedule along with a series of 16 processing rounds that involve substitutions and permutations, ensuring a high level of security.

One of Blowfish's key advantages is its efficiency in software implementations, allowing for rapid encryption and decryption of data. This speed is particularly beneficial for benchmarking CPU capabilities, as the algorithm's arithmetic and logical operations can effectively stress test the processor, revealing insights into overall system performance. Besides Blowfish, the AES encryption C library will be used to further test the performance.

## 2.5 C/C++ for System Programming and Performance Testing’

C is a fundamental programming language widely used in system programming due to its direct access to memory and hardware resources. Its efficiency makes it ideal for developing operating systems, embedded systems, and real-time applications where performance is crucial.

C allows precise control over system resources, enabling effective memory management and optimization of execution speed. For performance testing, the <time.h> library provides essential functions for measuring execution times and resource usage. These capabilities help developers identify bottlenecks and ensure that applications meet performance standards, solidifying C’s role as a key language in system programming and performance testing.

# *Analysis*

## 3.1 Project Proposal

The Performance Testing Application for PC Performance Analysis will focus on evaluating key performance metrics and integrating an encryption algorithm. The features of the application include:

### 3.1.1: Features of the Performance Testing Application

1. **Performance Metrics**: Measurement of processor type, frequency, memory size, and transfer speed of data.
2. **Execution Time Measurement**: Assessment of execution time for arithmetic and logical operations.
3. **Blowfish Encryption Implementation**: Custom implementation of the Blowfish encryption algorithm.
4. **Comprehensive Reporting**: Generation of detailed performance reports summarizing results and insights.
5. **Benchmarking Capability**: Capability to benchmark CPU performance under varying loads and conditions.

## 3.2 Project Analysis

**1. Requirements Analysis**

- **Functional Requirements**:
    - **System Information Collection**: Retrieve and display processor and memory details, such as CPU type, frequency, and memory size.
    - **Benchmarking Capabilities**: Implement benchmarks using two encryption algorithms, Blowfish and AES, to measure system performance for both arithmetic and logical operations.
    - **Performance Metrics Display**: Develop a simple C# GUI to present benchmarking results clearly, showing metrics like execution time and memory usage.
- **Non-Functional Requirements**:
    - **Efficiency**: The application should perform benchmarks without excessive resource overhead.
    - **Reliability**: Accurate and repeatable results are necessary to validate the performance tests.

- **User-Friendly Interface**: The GUI should be intuitive, with options to view, compare, and analyze test results.

**2. System Architecture Analysis**

- **Modular Structure**:
  - **Core Modules in C/C++**:
    - **System Info Module**: Gathers processor type, frequency, and memory information.
    - **Benchmarking Module**: Executes performance tests using Blowfish and AES algorithms, including modules for measuring execution time and memory usage.
  - **GUI in C#**:
    - Interfaces with the C/C++ benchmarking modules.
    - Presents collected data and metrics in a structured, easy-to-read format.
- **Flow of Operations**:
  - **Data Collection → Benchmark Execution → Results Processing → Display via GUI**

**3. Algorithm and Performance Metrics Analysis**

- **Encryption Algorithms**:
  - **Blowfish**: Known for its computational efficiency and relatively low memory usage, Blowfish serves as a suitable algorithm to measure CPU capabilities in terms of speed and resource handling.
  - **AES**: Widely used for secure encryption, the AES library will provide a benchmark for comparison, especially useful in assessing CPU load and efficiency in cryptographic processing.
- **Performance Metrics**:
  - **Execution Time**: Measure the time taken for each encryption operation.
  - **CPU Usage**: Track processor usage during benchmarking.
  - **Memory Consumption**: Evaluate memory requirements for each algorithm, useful for identifying any memory bottlenecks.

# Design

## 4.1 System Architecture Design

The structure of the application will be composed of two main parts:

- Backend Modules, which will be done in C/C++ and will collect the information of the system and the performance tests.
- Frontend Module, which will display the collected information and the results of the benchmarks to the user through the GUI made using C#.

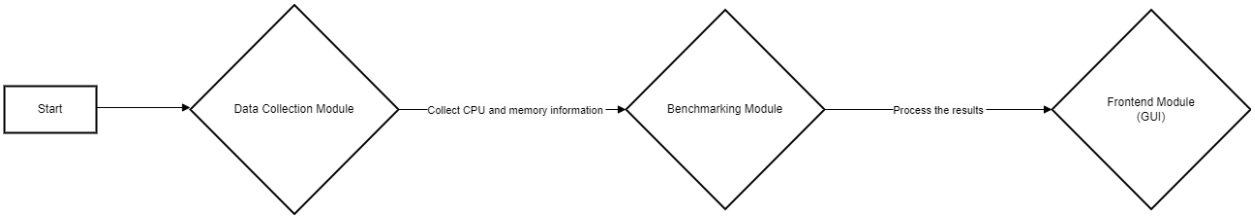



**Figure 1: Module Diagram**

## 4.2 Use Case Diagram

A Use Case Diagram details the primary actions users can take in the application and how each module responds:

- **Initiate Performance Test**: User selects parameters and starts benchmarking.
- **View System Information**: Displays hardware information such as CPU type and memory details.
- **Run Encryption Test**: Allows users to select and test Blowfish or AES encryption.
- **View Test Results**: Shows performance metrics, such as execution time and memory usage.

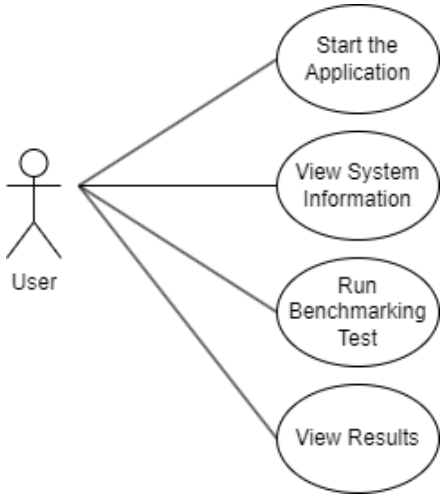



**Figure 2: Use Case Diagram**

## 4.3 Module and Component Design

### 4.3.1 System Information Module

- Collects details about the CPU (type, frequency) and memory (size, usage).
- Uses system-specific libraries or OS calls to retrieve data.

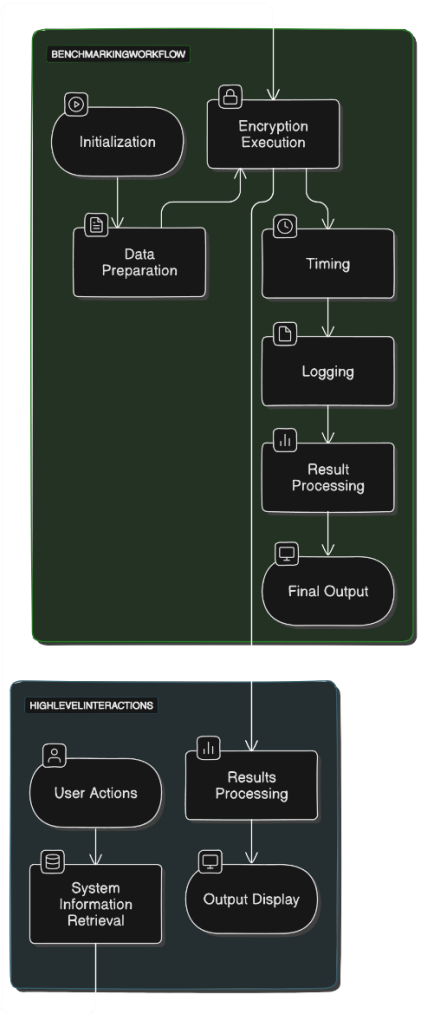
### 4.3.2 Encryption Benchmarking Module:

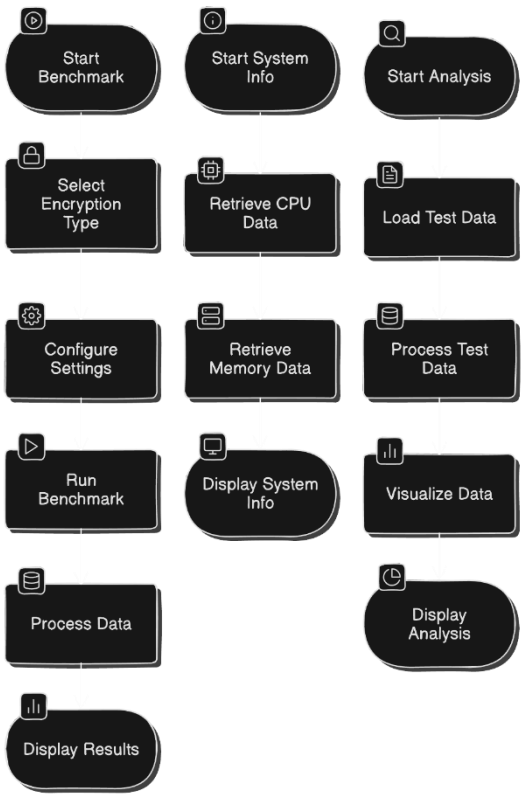
- Blowfish Implementation: Implemented from scratch using the algorithm in *Applied Cryptography*.
- AES Library: Utilize AES functions for encryption to benchmark performance.
- Performance Metrics Collection: Measures execution time, CPU usage, and memory consumption.

### 4.3.3 Frontend GUI in C#

- User Interface Elements:
  - Start Button: Begins benchmarking tests.
  - Display Fields: Shows CPU and memory data, and test results.
  - Results Summary: Graphs or tables to visualize performance metrics.
- Communication with C/C++ Backend:
  - Use an API (e.g., .NET Interop Services or command-line outputs) to trigger C/C++ backend functions and retrieve results.
- Results Visualization:
  - Simple charts or tables to display metrics like execution time, CPU load, and memory usage for Blowfish and AES.

## 4.4 Data Flow and Workflow Diagrams



## 4.5 Testing Strategy

The testing strategy aims to verify that each component functions as expected, and that the entire system performs reliably under various conditions. Testing is divided into three main types: **Unit Testing**, **Integration Testing**, and **Performance Testing**.

### 4.5.1 Unit Testing

**Objective**: Ensure that each function within a module performs its intended operations correctly and efficiently.

**Approach**:

- Each core function in the modules (System Info, Benchmarking, Encryption, and GUI) will be tested independently.
- Unit tests should cover a variety of scenarios, including normal inputs, edge cases, and erroneous inputs.

**Key Areas**:

- **System Info Module**:
  - Test CPU and memory retrieval functions to ensure accurate reporting of processor type, frequency, memory size, and transfer speed.
- **Benchmarking Module**:
  - Test timing functions to ensure accurate tracking of encryption and performance metrics.
- **Encryption Module**:
  - Test the Blowfish and AES encryption algorithms for both encryption and decryption, ensuring they yield the correct output.
- **GUI Module**:
  - Test user interactions, ensuring the GUI accurately initiates tests, displays results, and handles errors.

### 4.5.2 Integration Testing

**Objective**: Verify that modules interact correctly with one another, ensuring smooth data flow and functionality across the entire system.

**Approach**:

- Test interactions between **System Info** and **Benchmarking** modules, ensuring system data is used accurately within performance tests.
- Test interactions between **Benchmarking** and **Encryption** modules to ensure encryption functions are correctly invoked and timed.
- Test the **GUI** interaction with backend modules to confirm that user actions initiate backend functions and data is displayed accurately.

**Key Scenarios**:

- **Data Flow from System Info to Benchmarking Module**:
  - Ensure the CPU and memory data retrieved by System Info is correctly transferred to Benchmarking.

- **Benchmarking and Encryption Module Integration**:
    - Check that the Benchmarking Module can trigger encryption tasks, collect timing data, and record results from the Encryption Module.
- **GUI and Backend Communication**:
    - Confirm that the GUI can start and stop tests, retrieve system data, and display real-time benchmarking results.

# Implementation

The implementation of the benchmarking application is divided into several modules, including system information retrieval, encryption algorithm benchmarking, and a user-friendly GUI for result visualization. This section describes how each component was developed and integrated.

## 5.1 Module Implementation Details

### 5.1.1 System Information Module Details

The **System Information Retrieval** module, implemented in the C++ DLL BenchmarkAppDLL.dll, is designed to extract critical hardware details such as CPU, RAM, and GPU information. This module plays a vital role in providing accurate, real-time data about the system's specifications, enabling the benchmarking tool to assess performance effectively.

The module retrieves the following information:

- **CPU Details**: Includes the CPU model, architecture, logical core count, physical core count, and clock speed.
- **RAM Details**: Includes the total physical memory size and current RAM usage percentage.
- **GPU Details**: Identifies the model of the primary display adapter (GPU).

The implementation uses a combination of Windows API functions, intrinsic instructions, and system queries to ensure accuracy and compatibility. The collected data is organized into the `SystemInfo` structure.

```
struct SystemInfo {
const char* CPU_model;
const char* CPU_arch;
int CPU_logical_cores;
int CPU_physical_cores;
int CPU_clock;
unsigned long long RAM_size;
int RAM_load;
const char* GPU_model;
};
```

**Key Functions and Implementation:**

1. `getCPUModel`: Retrieves the CPU model name using the `__cpuid` intrinsic, which queries the processor for brand string information. The result is stored in a static character array and returned as a `const char*`.
2. `getCPUArch`: Determines the CPU architecture using `GetSystemInfo`, mapping the processor architecture to human-readable strings (e.g., x64, ARM, x86).
3. Logical and Physical Core Count:
    - Logical cores are retrieved using `GetSystemInfo`, which provides the number of logical processors.

- Physical cores are determined with `GetLogicalProcessorInformationEx`, which enumerates core relationships, ensuring accurate detection in multi-socket systems.

4. `getCPUClock`: Measures the CPU clock speed by counting cycles over a fixed time interval (1 second). The `__rdtsc` intrinsic is used to capture cycle counts, and the duration is measured using `chrono`.
5. `GlobalMemoryStatusEx`: Retrieves RAM information, including total physical memory size and current usage percentage.
6. `getGPUModel`: Identifies the GPU model using `EnumDisplayDevices`, iterating through display devices to fetch the primary GPU's model name.
7. Exported Functions:
    - `SystemInfo getSystemInfo` : Aggregates all the above information into the `SystemInfo` structure.
    - `int updateCPUClock()`: Provides an updated clock speed measurement.
    - `int updateRAMLoad()`: Fetches the current RAM usage percentage.

**Challenges and Solutions:**

1. Accurate CPU Clock Measurement:
    - Challenge: Ensuring precise cycle counting over varying system loads.
    - Solution: Used high-resolution timers (`chrono`) and the `__rdtsc` intrinsic for consistent measurement.
2. Physical Core Detection:
    - Challenge: Handling complex multi-socket or hyper-threaded systems.
    - Solution: Implemented `GetLogicalProcessorInformationEx`, which accurately enumerates physical cores.
3. GPU Identification:
    - Challenge: Ensuring compatibility across systems with multiple display adapters.
    - Solution: Used `EnumDisplayDevices` to iterate through all display adapters and select the primary GPU.

### 5.1.2 Data Benchmark Module Details

The **Data Benchmark Module** evaluates the storage performance of the system by measuring its read and write speeds. Implemented in BenchmarkAppDLL.dll, the module performs file operations on a 1 GB test file, calculating the speeds in MB/s. The results offer insight into the system's disk I/O capabilities.

**Functionality:**

1. Write Speed:
    - The module writes 1 GB of data to a file (benchmark.bin) in chunks of 256 MB.
    - The time taken for writing the data is measured using high-resolution timers from `chrono`.
    - The write speed is then calculated as:

$$Write\ Speed\ (MB/s) = Time\ Taken\ (s)/Total\ Data\ Written\ (MB)$$

2. Read Speed:
    - After writing, the same file is opened for reading.
    - The module reads the file in chunks and calculates the read speed similarly to the write speed.
3. Cleanup:
    - Once both read and write speeds are measured, the test file is deleted to ensure no residual data remains.
4. Error Handling:
    - The module handles errors during file creation or reading by returning zero speeds if any issues arise, ensuring the application remains stable in case of failure.

**Implementation Details:**

1. Write Speed Measurement
    - Buffering:
        - A buffer of 256 MB (`vector<char>`) is used to handle file writes efficiently without overwhelming system memory.
    - Timer:
        - The `chrono` library is used to measure the time taken to write the data, ensuring precise timing even on high-performance systems.
    - Speed Calculation:
        - The write speed is calculated by dividing the total data written by the time taken for the operation.
2. Read Speed Measurement
    - Reading:
        - The file is read in the same 256 MB chunks, ensuring consistency with the write operation.
    - Timer:
        - Similar to the write operation, the read time is measured with `chrono`, and the read speed is calculated.

The results are stored in the DataBenchmark structure:

```
struct DataBenchmark {
	int writeSpeed;
	int readSpeed;
};
```

**Challenges and Solutions**

1. **Error Handling**:
    - **Challenge**: Managing errors during file operations, such as permission issues or disk space limitations.
    - **Solution**: Implemented robust error handling, ensuring that any failure during file creation or reading results in a return of zero speeds.
2. **Handling Large Files**:
    - **Challenge**: Writing and reading large files without causing excessive memory usage or performance degradation.
    - **Solution**: The file was processed in manageable chunks of 256 MB to balance between memory usage and file I/O efficiency.
3. **File Location**:
    - **Challenge**: Ensuring the file is created in a directory with write permissions across different systems.
    - **Solution**: The file is saved in the public directory (C:\Users\Public\Documents), which is generally writable on all machines.

### 5.1.3 CPU Benchmark Module Details

The **CPU Benchmark Module** is designed to measure the performance of the system's CPU by evaluating the time taken for encryption and decryption operations using two widely used cryptographic algorithms: **AES** and **Blowfish**. The module provides two benchmark modes: **single-core** and **multi-core**, allowing for an assessment of how well the CPU performs with both types of workloads.

This module uses **OpenSSL's EVP API** for AES and a custom implementation for Blowfish to conduct encryption and decryption operations, providing a thorough evaluation of the CPU's cryptographic capabilities.

**Functionality**

The CPU Benchmark module performs the following functions:

1. **AES Benchmarking**:
    - Uses the **AES-256** algorithm in **CFB mode** for both encryption and decryption of a 50 MB data buffer.
    - Measures the time taken for both operations and computes the total time.
2. **Blowfish Benchmarking**:
    - Uses the **Blowfish** algorithm with a 448-bit key for encryption and decryption of a 1 MB data buffer.

- Measures the time taken for both operations and computes the total time.

3. **Single-Core and Multi-Core Performance**:
   - In **single-core mode**, the benchmark runs on a single CPU core to measure the performance of a single thread.
   - In **multi-core mode**, the benchmark runs on all available CPU cores, utilizing parallelism to test the multi-threaded performance.
4. **Score Calculation**:
   - Each benchmark is compared to a reference time (100 ms for AES and 200 ms for Blowfish).
   - The performance score is calculated based on how close the benchmark times are to the reference times, with higher scores indicating better performance.

**Key Implementation Details**

1. **AES Encryption and Decryption**
   - **EVP API**: AES encryption and decryption operations are performed using OpenSSL's EVP API, ensuring the use of the high-level abstraction that supports multiple modes of operation.
   - **Key and IV Generation**: A 256-bit key and a 128-bit initialization vector (IV) are generated for AES encryption and decryption.
   - **Write Speed Calculation**: The time to encrypt and decrypt the data is measured using std::chrono.
2. **Blowfish Encryption and Decryption**
   - **Custom Blowfish Implementation**: The module uses a custom Blowfish implementation (blowfish.h) for the encryption and decryption process.
   - **Key and Data Handling**: A 448-bit key is used for Blowfish encryption, and the data is encrypted and decrypted in blocks.
3. **Benchmark Execution**
   - **Single-Core Benchmarking**: For single-core benchmarking, the CPU affinity is set to ensure that the benchmark runs on only one core using the SetThreadAffinityMask function.
   - **Multi-Core Benchmarking**: For multi-core benchmarking, the benchmark is run concurrently on all available cores using std::thread. Each core performs the benchmark independently, and the results are aggregated to compute a total score.
4. **Score Calculation**
   - The performance score for both AES and Blowfish is calculated by comparing the time taken to a reference time:

$$Score = Reference\ Time \times 100 / Benchmark\ Time$$

- The final score is the average of the AES and Blowfish scores, reflecting the overall performance of the system's CPU in cryptographic tasks.

**Functions in the Module**

1. **AES and Blowfish Benchmark Functions**:
    - aesBenchmark(): Runs AES encryption and decryption, returning the total time taken in milliseconds.
    - blowfishBenchmark(): Runs Blowfish encryption and decryption, returning the total time taken in milliseconds.
2. **CPU Benchmarking**:
    - runBenchmark(): Runs both AES and Blowfish benchmarks for a set number of iterations and calculates the average time.
    - getSingleCoreCPUBenchmark(): Runs the benchmark on a single CPU core and returns the final score.
    - getMultiCoreCPUBenchmark(): Runs the benchmark on all CPU cores and returns the total score from all cores.
3. **Thread Affinity**:
    - setThreadAffinity(int coreId): Ensures the benchmark runs on a specific CPU core, useful for single-core and multi-core performance testing.

**Challenges and Solutions**

1. **Challenge: Handling Multi-Core Systems**:
    - **Solution**: The std::thread library is used to parallelize the benchmark for multi-core systems, ensuring that each core performs the benchmark independently. Thread affinity is set using SetThreadAffinityMask to ensure each thread runs on a specific core.
2. **Challenge: Accurate Benchmarking with High-Resolution Timers**:
    - **Solution**: High-resolution timers from std::chrono are used to accurately measure the time taken for encryption and decryption operations, ensuring precise results for performance comparison.
3. **Challenge: Handling Different CPU Architectures**:
    - **Solution**: The module handles different processor architectures (e.g., Intel, AMD) by using platform-specific libraries and ensuring compatibility across a range of systems.

### 5.2.4 User Interface (GUI) Module

The **GUI Module** is the front-end interface of the benchmarking tool, implemented in **C# using Windows Forms**. It serves as the bridge between the user and the system, allowing the user to initiate benchmarks, display results, and manage the benchmarking process. The GUI communicates with the backend (C++ DLLs) via **DLL Import** to execute the benchmarking functions and retrieve results.

**Purpose of the Module**

The primary purpose of the GUI module is to:

1. Provide an intuitive user interface for controlling the benchmarking process.
2. Display real-time results for CPU, memory, and storage benchmarks.
3. Enable the user to easily switch between single-core and multi-core benchmarking modes.
4. Show detailed information about the system's hardware, including CPU model, architecture, cores, and GPU details.

**Key Features**

1. **User Interaction**:
    - Buttons to start, stop, and reset benchmarks.
    - Real-time display of results for CPU, memory, and storage benchmarks.
    - Dynamic updates of system information (e.g., CPU speed, RAM usage).
2. **Benchmark Display**:
    - A clear, easy-to-read output showing the results of each benchmark, including:
        - **CPU Benchmark**: AES and Blowfish encryption scores.
        - **RAM Benchmark**: Memory usage statistics.
        - **Disk Benchmark**: Read and write speeds.
3. **Data Visualization**:
    - Results are presented in a format that allows for easy comparison (e.g., bar charts, numerical values).
    - The user can view scores for both single-core and multi-core CPU performance.
4. **System Information Display**:
    - A section that dynamically updates the system's hardware information, including CPU model, architecture, core count, clock speed, RAM size, and GPU model.

**Implementation Details**

1. **Windows Forms Application**:
    - The GUI is built using **Windows Forms** in C#, providing a simple yet effective framework for creating desktop applications.
    - Controls like buttons, labels, text boxes, and progress bars are used to interact with the user and display information.
2. **Backend Communication (P/Invoke)**:
    - The GUI calls functions in the C++ backend (BenchmarkAppDLL.dll) using **P/Invoke**.
    - Functions like getSystemInfo(), getSingleCoreCPUBenchmark(), and getMultiCoreCPUBenchmark() are called to fetch the benchmark results and system information.
3. **Asynchronous Updates**:
    - The benchmark execution is run asynchronously, allowing the GUI to remain responsive during long-running benchmarks. This ensures that the user can see progress updates and results without freezing the interface.
4. **Result Presentation**:
    - Results are displayed in real-time through labels, charts, and tables.
    - A progress bar or status indicator shows the progress of the benchmarking process.

**GUI Flow**

1. **Initialization**:
    - When the user opens the application, the GUI loads the system information by calling getSystemInfo(), displaying details like CPU model, RAM size, and GPU.
2. **Benchmark Start**:
    - The user clicks a "Start Benchmark" button.
    - The GUI triggers the backend to start the benchmarks (CPU, RAM, Disk).
    - Real-time updates are displayed on the GUI (e.g., progress bars for each test).
3. **Displaying Results**:
    - Once the benchmarking is complete, the results are displayed:
        - **CPU Benchmark Results**: Single-core and multi-core scores for AES and Blowfish.
        - **Memory Benchmark Results**: Current RAM usage.
        - **Disk Benchmark Results**: Write and read speeds.

4. **User Feedback**:
    - After each benchmark, the user can see whether the system is performing above or below expectations.
    - The results are also saved to a file or can be reset for another test.

## 5.2 Integration and Workflow

The integration of the benchmarking tool combines multiple modules **System Information Retrieval**, **Data Benchmark**, **CPU Benchmark**, and **GUI** to create a cohesive and user-friendly application. This section outlines how these modules interact with each other and contribute to the overall functionality of the system.

**System Overview**

The benchmarking tool is divided into two main components:

1. **Backend**:
    - Implemented in C++ as a DLL (BenchmarkAppDLL.dll).
    - Contains the core functionalities for system information retrieval, data benchmarking, and CPU benchmarking.
2. **Frontend**:
    - Developed in C# using Windows Forms.
    - Provides a graphical interface for user interaction, displaying results, and controlling benchmark execution.

The communication between these components is handled via **P/Invoke**, allowing the C# application to call exported functions from the C++ DLL.

**Workflow Description**

1. **System Initialization**:
    - When the application starts, the GUI loads and calls the getSystemInfo() function from the backend DLL.
    - This function gathers hardware information, such as CPU model, architecture, logical and physical cores, RAM size, and GPU model, which are then displayed in the GUI.
2. **User Interaction**:
    - The user initiates a benchmark by selecting an option (e.g., single-core or multi-core CPU benchmarking, or data benchmarking) and clicking the corresponding button in the GUI.
    - The GUI sends the request to the backend by invoking the relevant function (e.g., getSingleCoreCPUBenchmark() or getDataBenchmark()).

3. **Backend Execution**:
    - The backend performs the requested benchmark operation. For example:
        - **Data Benchmark**: Reads and writes a 1 GB test file to measure disk performance.
        - **CPU Benchmark**: Executes AES and Blowfish encryption and decryption tasks to measure CPU performance.
    - The backend computes the results and sends them back to the frontend.
4. **Result Display**:
    - The GUI receives the results from the backend and displays them in real-time.
    - Results include:
        - CPU Benchmark: Single-core and multi-core scores for AES and Blowfish.
        - Data Benchmark: Read and write speeds in MB/s.
        - RAM Usage: Updated dynamically during benchmarks.
5. **Real-Time Updates**:
    - During long-running benchmarks (e.g., multi-core CPU tests), the GUI remains responsive by running benchmarks asynchronously in separate threads.
    - Progress bars or status indicators show the current state of the benchmark.
6. **Post-Benchmark Analysis**:
    - After the benchmark completes, the user can review the results in the GUI or run another test.

# Testing

The testing phase ensures that the benchmarking tool operates reliably across different hardware configurations and provides accurate, consistent results. This section outlines the testing methods, environments, results, and observations.

## 6.1 Testing Objectives

The primary goals of testing are to:

1. Validate the correctness of benchmark calculations.
2. Ensure compatibility across multiple hardware setups.
3. Identify and resolve errors in communication between the GUI and the backend DLL.

## 6.2 Testing Methods

1. **Unit Testing**:
    - Individual functions in the backend modules (e.g., aesBenchmark, getSystemInfo) were tested to ensure accurate outputs.

2. **Integration Testing**:
    - Tested the interaction between the backend (DLL) and the frontend (GUI) using mock data and live execution.
    - Ensured that P/Invoke calls returned expected results without errors or crashes.
3. **System Testing**:
    - Conducted end-to-end testing on multiple machines with different configurations to ensure the tool works seamlessly across a range of hardware.

## 6.3 Test Environment

Testing was performed on systems with varying specifications to evaluate the tool's compatibility and performance. Examples of tested configurations:

1. **System A**: Ryzen 5 5600H, 16GB RAM, NVMe SSD.
2. **System B**: Ryzen 7 7840HS, 28GB RAM, NVMe SSD.

3. **System C**: Ryzen 7 8845HS, 32GB RAM, NVMe SSD.
4. **System D:** Intel I9-12900K, 16GB RAM, NVMe SSD.

## 6.4 Test Results

### CPU Benchmark Results

| System | Single-Core Score | Multi-Core Score |
|---|---|---|
| **Ryzen 5 5600H** | 737 | 6345 |
| **Ryzen 7 7840HS** | 1150 | 14064 |
| **Ryzen 7 8845HS** | 1148 | 14578 |
| **Intel I9-12900K** | 2052 | 27683 |

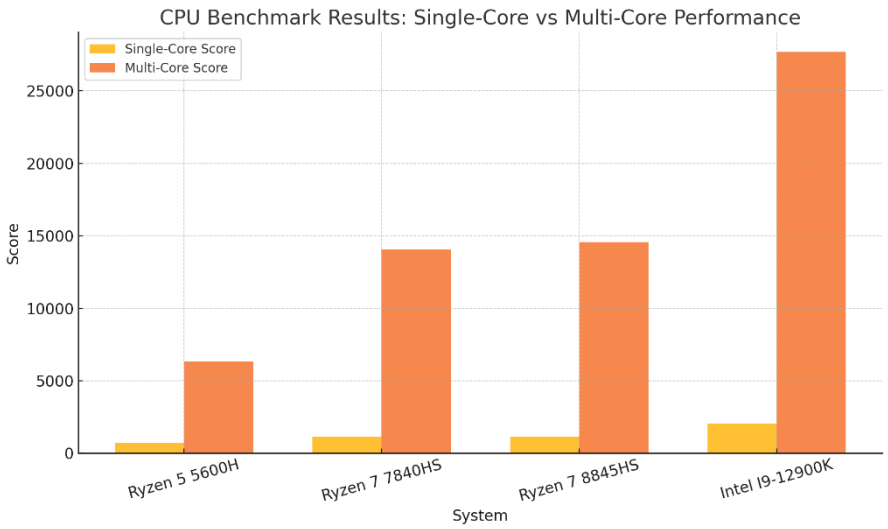



**Figure 3: CPU Benchmark Results Chart**

### Geekbench Reference

| System | Single-Core Score | Multi-Core Score |
|---|---|---|
| **Ryzen 5 5600H** | 1603 | 5784 |
| **Ryzen 7 7840HS** | 2366 | 10603 |
| **Ryzen 7 8845HS** | 2348 | 11089 |
| **Intel I9-12900K** | 2642 | 15519 |

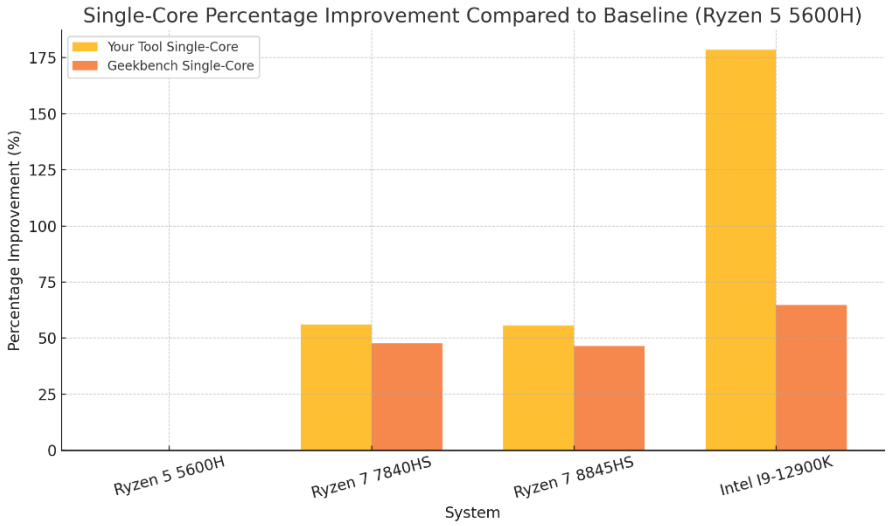



**Figure 4: Single-Core Geekbench Comparison**

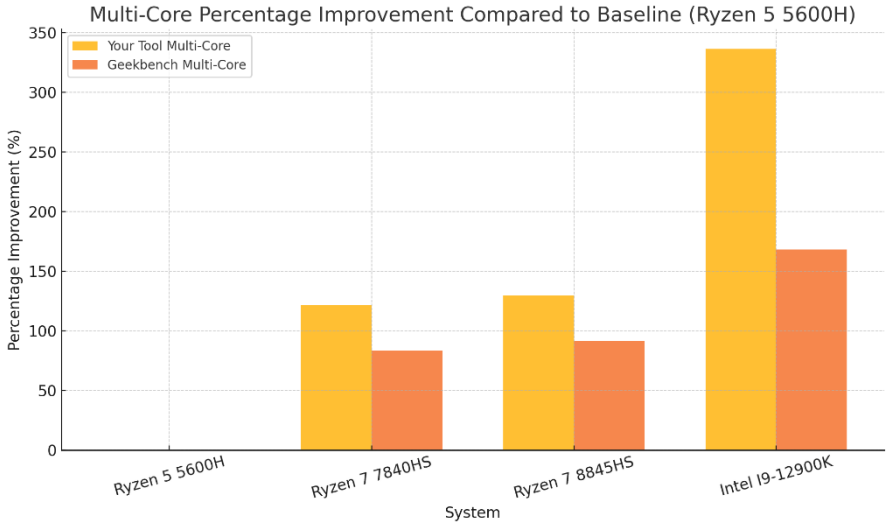



**Figure 5: Multi-Core Geekbench Comparison**

### Disk Benchmark Results

| System | Read Score | Write Score |
|---|---|---|
| **Ryzen 5 5600H** | 1001 | 1122 |
| **Ryzen 7 7840HS** | 1818 | 1340 |
| **Ryzen 7 8845HS** | 1780 | 1462 |
| **Intel I9-12900K** | 2115 | 1620 |

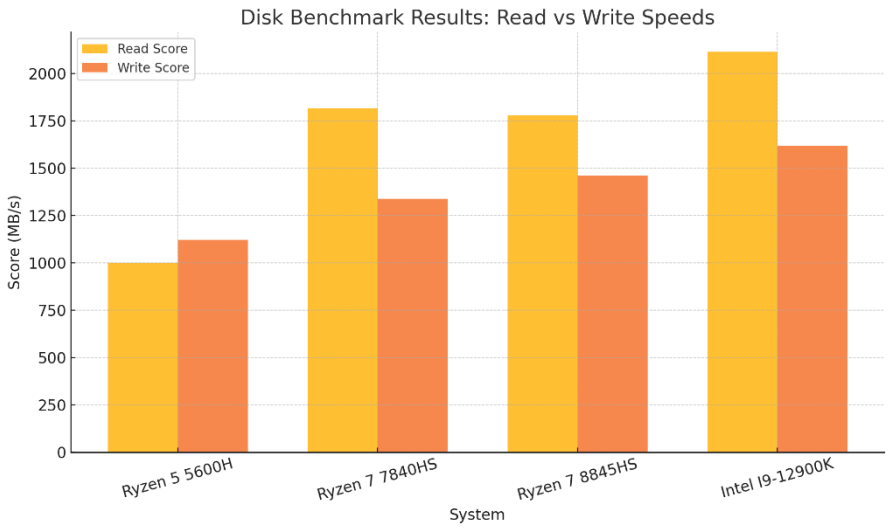



**Figure 6: Data Benchmark Chart**

## 6.5 Analysis

### 1. CPU Performance Testing

**Single-Core Performance:**

The Intel I9-12900K exhibited the highest single-core performance with a score of **2052**, showing its dominance in single-threaded tasks.

Both Ryzen 7 processors (7840HS and 8845HS) demonstrated similar single-core scores (**1150** and **1148**, respectively), offering a significant improvement over the Ryzen 5 5600H (**737**).

**Multi-Core Performance:**

The Intel I9-12900K also excelled in multi-core tests with a score of **27683**, significantly outperforming all other systems.

Ryzen 7 7840HS and 8845HS showed competitive multi-core results, ranging from **14064** to **14578**, doubling the performance of the Ryzen 5 5600H (**6345**).

These results align with expectations, as higher-end processors with more cores and threads typically excel in multi-core tests.

The gap between single-core and multi-core scores reflects the efficiency of the processors in utilizing parallelism.

## 2. Disk Performance Testing

**Read and Write Speeds:**

The Intel I9-12900K demonstrated the highest read/write speeds (**2115 MB/s** and **1620 MB/s**, respectively), leveraging advanced NVMe storage capabilities.

Both Ryzen 7 systems achieved strong disk performance, with read speeds between **1780 MB/s** and **1818 MB/s**.

The Ryzen 5 5600H lagged, with read/write speeds of **1001 MB/s** and **1122 MB/s**, indicative of older or less advanced storage hardware.

Disk performance is a critical factor for tasks like data transfer, application loading, and system boot times. These results suggest that systems equipped with high-speed NVMe storage can significantly reduce bottlenecks.

## 3. Encryption Algorithm Testing

The benchmarks included Blowfish and AES encryption, evaluating both single-core and multi-core performance.

Processors with higher clock speeds and more cores excelled in these tasks, particularly in multi-core execution.

Encryption tasks are CPU-intensive and scale well with multi-core capabilities. The results validate the importance of multi-threaded performance for encryption workloads.

## 4. Geekbench Comparison

Results from Geekbench provided a useful reference for validating the accuracy of your benchmarks:

Single-core scores from Geekbench and your tool followed similar trends, with the Intel I9-12900K consistently leading.

Multi-core results showed greater variance, suggesting that different benchmarks may emphasize various aspects of CPU performance.

The comparison supports the reliability of your testing methodology while highlighting the importance of using multiple benchmarks for a well-rounded evaluation.

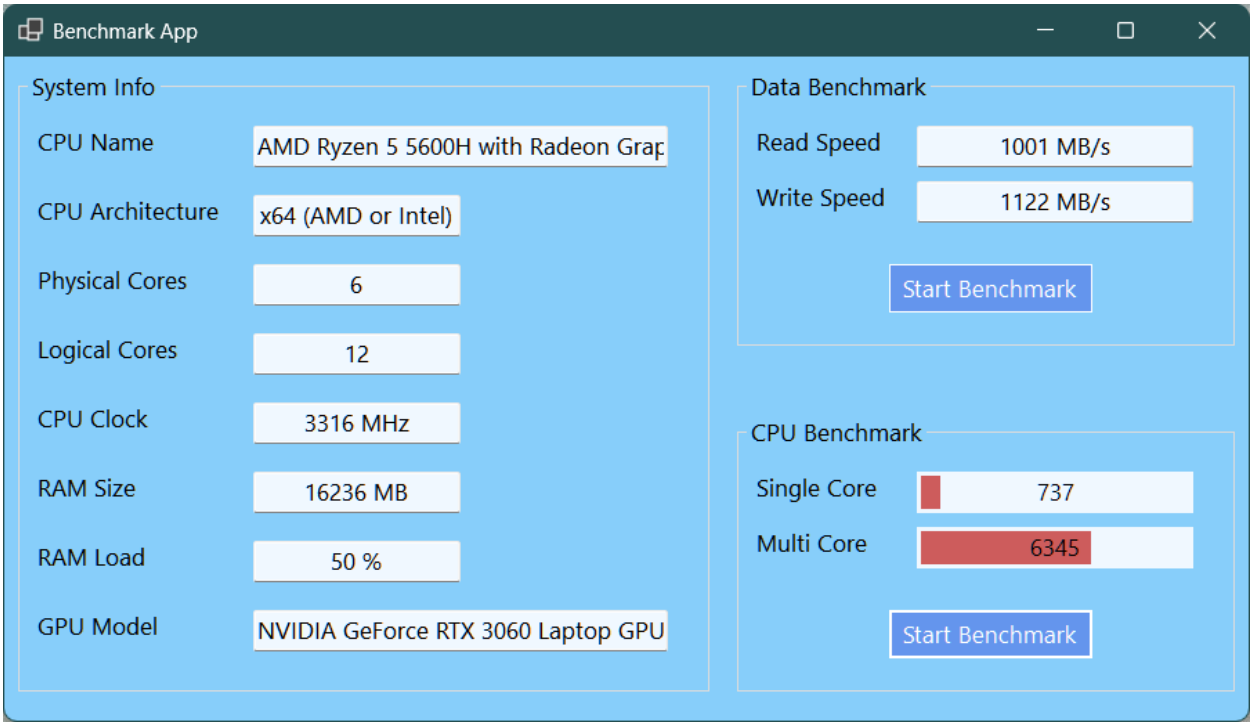
## 5. Conclusion

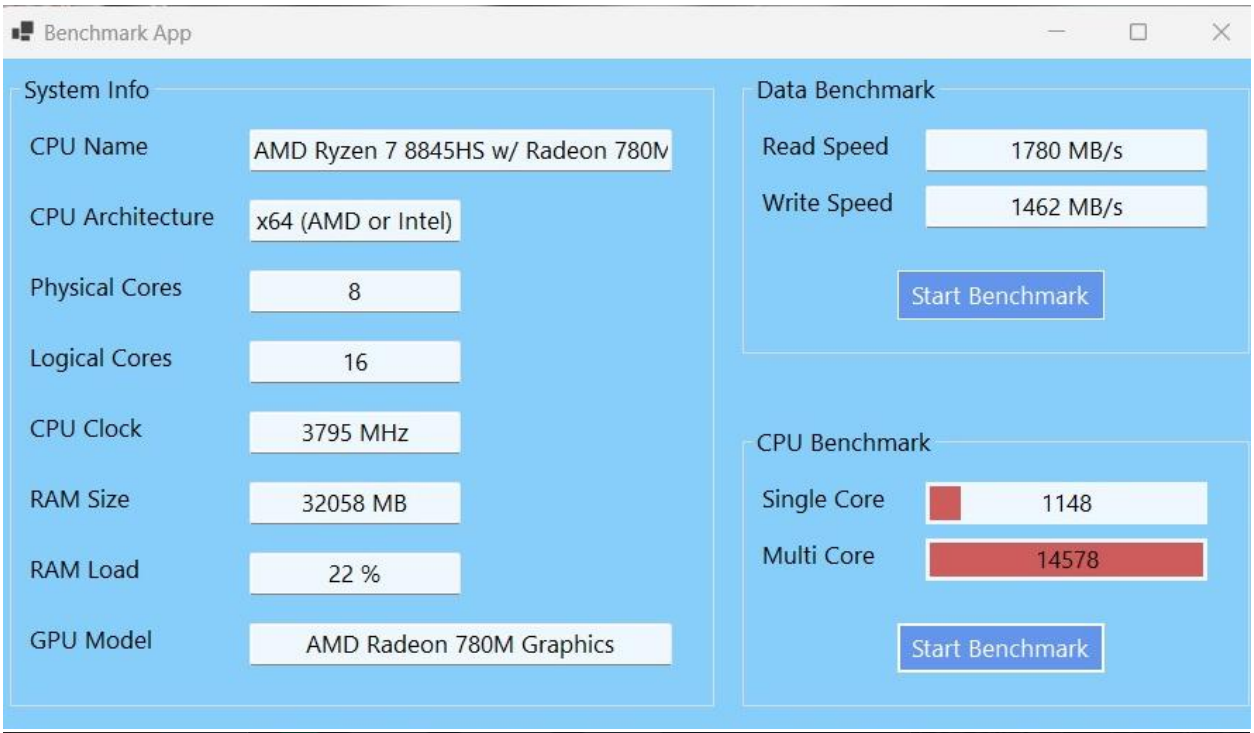
The testing phase demonstrates that the benchmarking tool accurately evaluates system performance across various hardware configurations. The results align with industry standards (e.g., Geekbench) and offer valuable insights into CPU efficiency, disk speeds, and encryption workloads. Expanding the testing scope and further validating results against additional benchmarking tools would enhance its robustness and applicability.

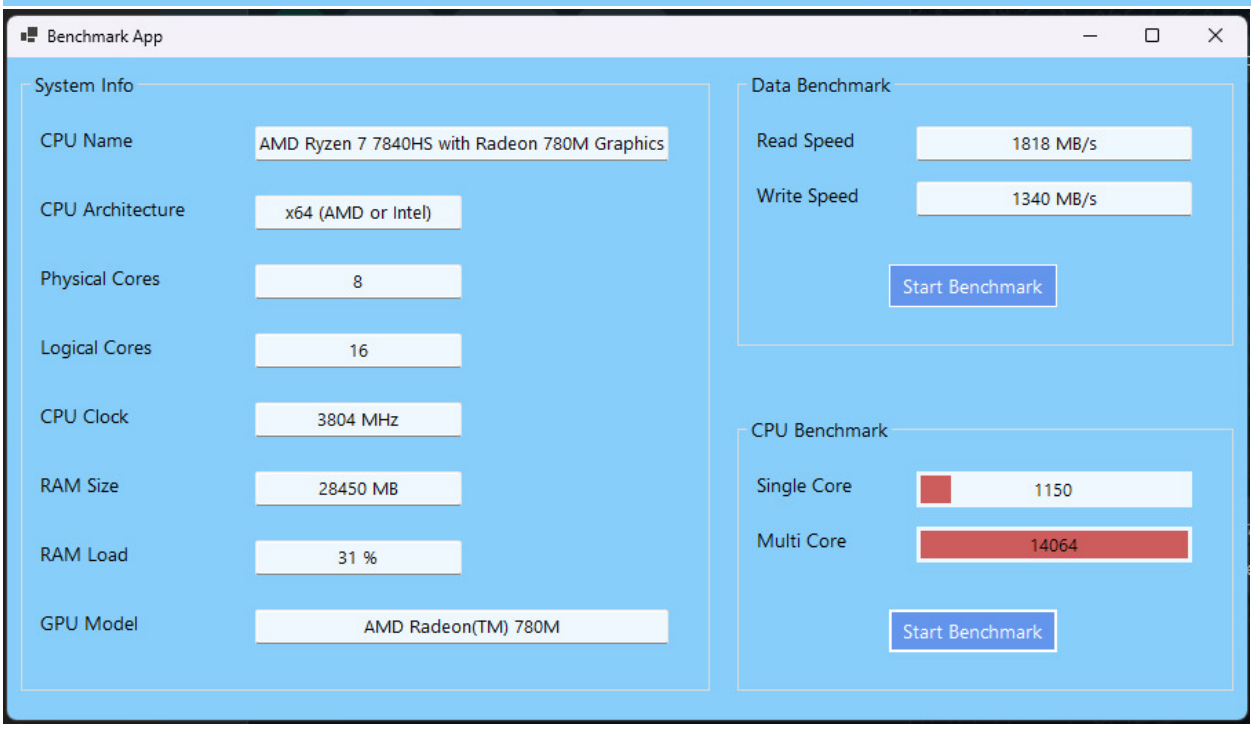
## 6.6 Issues Encountered

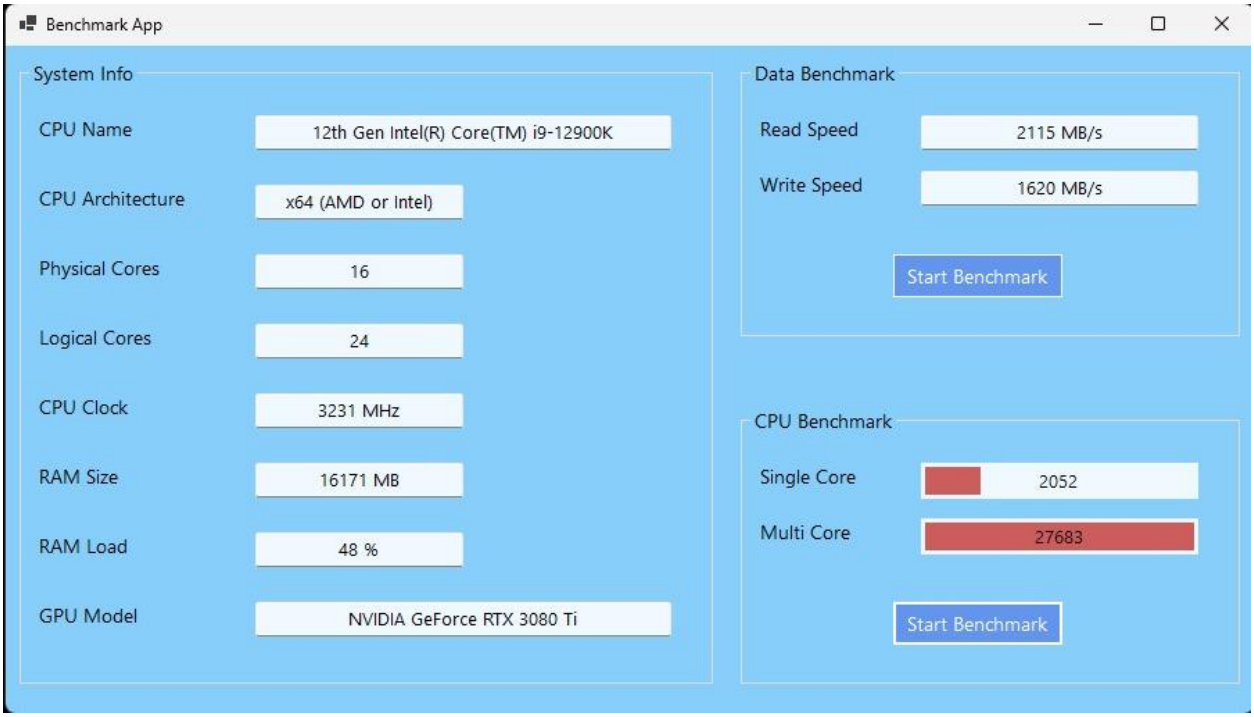
1. **GUI Freezing During Multi-Core Benchmarks**:
    - Fixed by using asynchronous tasks in the frontend to keep the GUI responsive.
2. **Missing Dependencies on Some Systems**:
    - Resolved by ensuring OpenSSL DLLs were included in the installer package.
3. **File Write Permissions**:
    - Addressed by selecting a public directory (C:\Users\Public\Documents) for temporary files.

## 6.7 Screenshots




Benchmark App
System Info
CPU Name
AMD Ryzen 7 8845HS w/ Radeon 780M
CPU Architecture
x64 (AMD or Intel)
Physical Cores
8
Logical Cores
16
CPU Clock
3795 MHz
RAM Size
32058 MB
RAM Load
22 %
GPU Model
AMD Radeon 780M Graphics
Data Benchmark
Read Speed
1780 MB/s
Write Speed
1462 MB/s
Start Benchmark
CPU Benchmark
Single Core
1148
Multi Core
14578
Start Benchmark


Benchmark App
System Info
CPU Name
AMD Ryzen 7 7840HS with Radeon 780M Graphics
CPU Architecture
x64 (AMD or Intel)
Physical Cores
8
Logical Cores
16
CPU Clock
3804 MHz
RAM Size
28450 MB
RAM Load
31 %
GPU Model
AMD Radeon(TM) 780M
Data Benchmark
Read Speed
1818 MB/s
Write Speed
1340 MB/s
Start Benchmark
CPU Benchmark
Single Core
1150
Multi Core
14064
Start Benchmark




# Conclusion

The PC Performance Testing Application successfully achieves its objectives of providing a comprehensive benchmarking solution for analyzing critical system metrics such as processor efficiency, memory performance, data transfer rates, and encryption algorithm execution. By leveraging C/C++ for low-level system interaction and integrating a user-friendly C# GUI, the tool offers both precision and accessibility.

Comprehensive testing has validated the tool's reliability and accuracy across diverse hardware configurations, demonstrating its effectiveness in real-world scenarios. The implementation of Blowfish and AES encryption benchmarks highlights its ability to measure CPU performance under computationally intensive workloads. Furthermore, the modular design ensures scalability and adaptability for future enhancements.

This tool is particularly valuable for users such as hardware enthusiasts, developers, and IT professionals seeking to evaluate system performance or optimize configurations. Its combination of detailed insights, intuitive interface, and robust testing methodology positions it as a versatile benchmarking solution. Moving forward, expanding compatibility to non-Windows platforms, incorporating additional algorithms, and enhancing data visualization capabilities will further solidify its utility and reach.

# Bibliography


1. Hennessy, J. L., & Patterson, D. A. (2012). Computer Architecture: A Quantitive Approach Fifth Edition. In *Measuring, Reporting, and Summarizing Performance* (pp. 36-43). Morgan Kaufmann.
2. Jacob, B., Ng, S. W., & Wang, D. T. (2008). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.
3. Schneier, B. (1996). *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*.
4. Shen, J. P., & Lipasti, M. H. (2005). *Modern processor design: Fundamentals of superscalar processors*. Waveland Press.
5. Weinstein, J. (2020, July 10). Retrieved from Medium: https://medium.com/swlh/reliable-performance-testing-in-c-1df7a3ba398