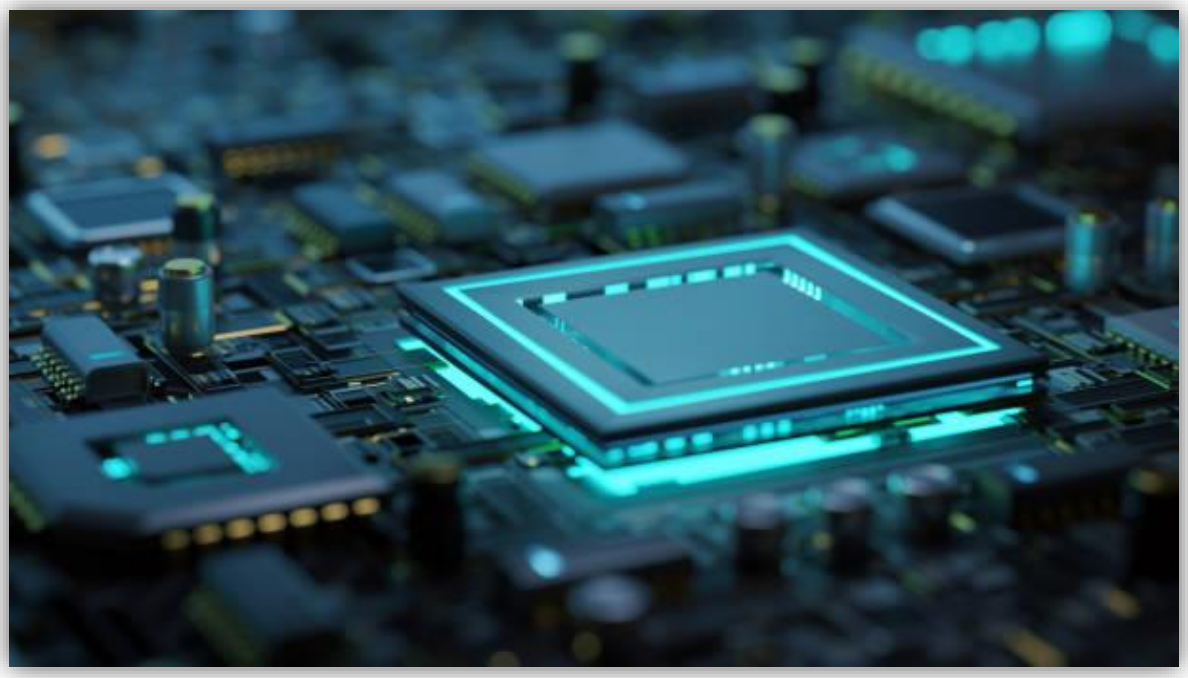
Program for testing the performance of a PC



Student: Cantemir Alexandru

Group: 30435

Table of Contents

1. Introduction

1.1 Project Proposal

Context

With the rapid evolution of computer hardware and the ever-increasing number of different types of CPUs, GPUs, RAM and others comes a high demand for determining the real performance of a PC beyond the factory specifications. This project aims to develop a simple and efficient program that can test and analyze the performance of the computer and display the information in an easy-to-understand manner. This establishes that the program is mainly designed to be accessible to the average, day-to-day computer user providing performance testing features without requiring advanced technical knowledge.

Problem

Users often lack an accessible and easy-to-use way to objectively measure the performance of their system. Without reliable testing tools, it is difficult to identify bottlenecks, detect performance degradation over time, or compare different hardware setups.

Motivation

Creating a lightweight and customizable PC performance testing program helps both enthusiasts and regular users understand how well their system performs. It can also assist technicians or students in evaluating how hardware changes influence system speed and stability.

Objectives

Measuring PC performance using a combination of arithmetic and logical operation and algorithms implemented in a low-level programming language that provides direct access to hardware and minimizes external dependencies. To promote accessibility and widespread use, the program will be distributed as free software, allowing any user to evaluate their system's performance without cost barriers. The benchmarking of the CPU will consist of different algorithms applied to different types of data like integers or floats alongside some basic information like the type of processor and its base frequency. Regarding the GPU, the application could compute the throughput and display the relevant information like FLOPS or OPS. The speed

of transferring different sizes of blocks of data will be the main parameter for benchmarking the RAM.

1.2 Project Timeline

Week 5-6

Implement a few rudimentary benchmark tests in order to check some basic capabilities of the computer. These tests would first target the CPU. The program would need to find the type of processor, the base frequency speed and the average computation time for some algorithms.

Week 7-8

Research and implement different algorithms and methods for testing the performance of the GPU especially regarding FLOPS/OPS.

Week 9-10

Managing a way to move dynamic blocks of data in order to test the performance of the RAM memory.

Week 11-12

Begin implementation of a basic GUI for the benchmark application. This version will focus on the early structure and layout, including initial interface elements and minimal functionality

Week 13-14

Extend and refine the GUI to make it fully functional. This includes integrating all core features of the benchmark system, enabling user interaction, displaying real benchmark results, and finalizing the visual design and responsiveness.

2.Bibliograpgic Study

2.1 Critical analysis of similar existing solutions

There are a lot of benchmarking programs that try to solve the described problems each with their advantages and disadvantages. Some of them are:

- **CPU-Z** - provides detailed hardware information and a basic CPU benchmark.
  Advantage: Simple interface and reliable CPU performance comparison.
  Drawback: Limited to processor tests; no memory or disk benchmarks.
- **Cinebench** - evaluates CPU performance through 3D rendering tasks.
  Advantage: Real-world workload simulation using multi-threading.
  Drawback: Focused mainly on rendering performance, not general computing.
- **SuperPI** - calculates the digits of π to test CPU and memory performance.
  Advantage: Very precise measurement of single-thread CPU performance and stability.
  Drawback: Outdated methodology; does not reflect multi-core or real-world workloads.
- **NovaBench** - a free, all-in-one benchmarking utility for Windows, macOS, and Linux.
  Advantage: Simple interface and quick execution; provides composite scores for CPU, GPU, RAM, and disk.
  Drawback: Limited testing depth and fewer configuration options compared to professional tools.
- **SiSoftware Sandra (System ANalyser, Diagnostic and Reporting Assistant)** - a comprehensive suite that benchmarks and analyzes all major system components.
  Advantage: Extremely detailed; includes both synthetic and real-world benchmarks.
  Drawback: Complex interface and many advanced features that can overwhelm average users; some modules require paid licenses.

2.2 Theoretical concepts

In [1] the authors raise the concern of reading the system clock at the start and end of a run and calculating the difference between. Because the system clock can jump due to time adjustments and even change its pace, it is important to use a time source that is guaranteed to be strictly monotonic and of constant rate. Many operating systems and programming languages offer such a time source with high precision specifically for benchmarking. Regarding this information, the main requirement of this program begins with understanding the usage of the RDTSC (Read Time-Stamp Counter) [2] function. The Time Stamp Counter (TSC) is a 64-bit register present on all x86 processors since the Pentium. It counts the number of CPU cycles since its reset. The instruction RDTSC returns the TSC in EDX:EAX. In x86-64 mode, RDTSC also clears the upper 32 bits of RAX and RDX. Programs like SuperPI or NovaBench use the

built-in Windows/Linux high precision functions instead of RDTSC because they focus more on specialized tasks and cross-platform compatibility rather than extreme precision.

3.Analysis

3.1 Performance parameters

3.1.1 CPU Performance

The first information the application will gather about the CPU will be the make and model, number of cores and threads and the clock frequency. Digging deeper, the application can use some benchmark tests to compute execution time and instructions per second. When we talk about benchmarking the CPU we have to take into consideration the clock frequency and its fluctuations that affect the final result of any kind of computation.

Integer Performance (CoreMark)

CoreMark is a Micro Benchmark (*Figure 1*) that tries to simulate real life workloads with integer arithmetic that uses different types of operations. These operations are based on linked lists, matrices and finite state machines.

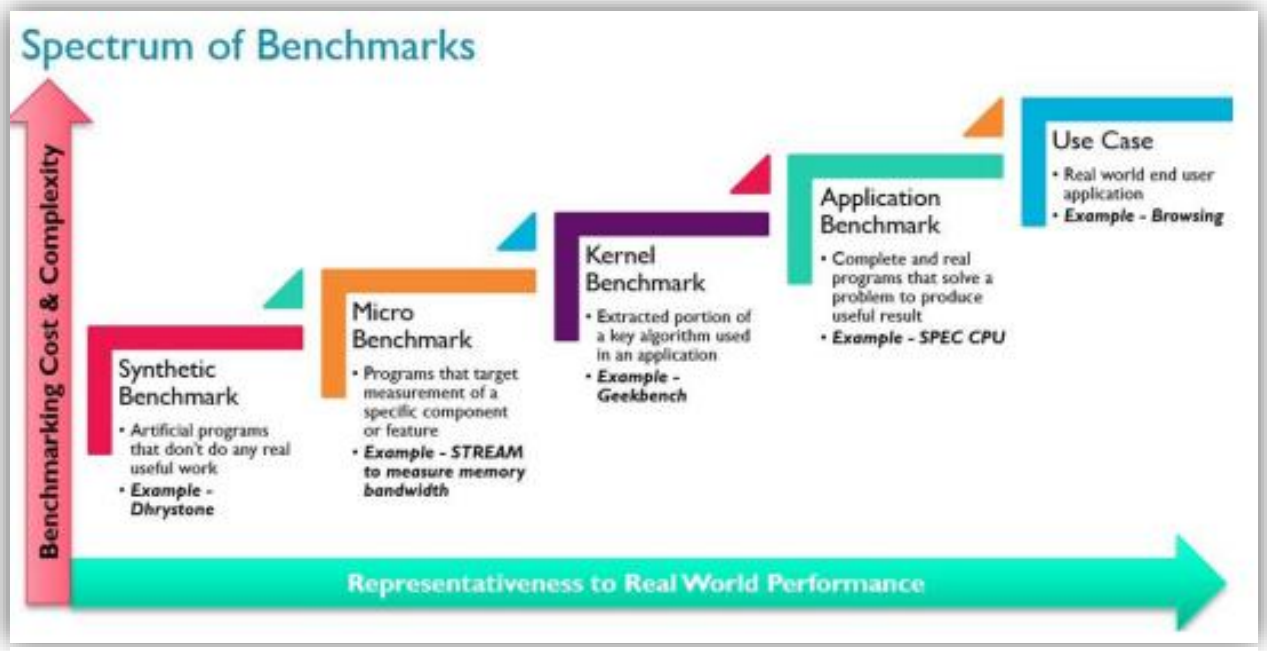


Figure 1. Spectrum of Benchmarks.

Floating-Point Performance (Whetstone)

Whetstone is a synthetic benchmark (*Figure 1*) for evaluating the performance of the CPU. Instead of measuring the performance of the CPU using integer arithmetic, it is instead based on floating-point operations. The basic form of Whetstone implements 8 procedures:

1. Floating point 1
2. Floating point 2
3. Branch (if-then-else)
4. Fixed point
5. Trigonometry (sin, cos, atan)
6. Floating point 3
7. Assignments (=)
8. Other maths (log, exp, sqrt)

3.1.2 GPU Performance

Same as the CPU, the first information displayed about the GPU will be the make and model, followed by some algorithms that will compute the FLOPS and memory bandwidth. When testing the performance of the GPU the number of threads needs to be optimized in such a way that the GPU is not underutilized because that can skew the results in a bad way.

The application will first compute the theoretically estimated value of the GPU's FLOPS with the formula: FLOPS = Cores * Clock Frequency * Operations per Cycle. This value will then be compared with the calculated value. To calculate this real-life value of the FLOPS the application will use multiply-and-add operations which consist of 2 FLOPS inside a loop of N iterations. A pseudocode of this algorithm would look like this:

```
Set up GPU kernel
Launch kernel with many threads:
   for i = 1 to N
      a = a * b + c
Synchronize
```

The main focus points of applying this algorithm are: making sure the kernel runs long enough to avoid timing noise, using GPU timers instead of CPU timers for accuracy, using registers to avoid memory bottlenecks and using enough threads to occupy all cores.

Memory bandwidth can be thought of in two separate ways. Firstly, we can think about the internal memory bandwidth which refers to the VRAM inside the GPU. The basic algorithm to measure the VRAM bandwidth is:

- Allocate two large buffers in GPU memory (large enough to saturate the cache)
- Warm up the GPU with a few basic memory operations
- Launch a kernel that performs a pure memory transfer like read-only, write-only and copy tests
- Time the transfer using high-resolution GPU timers

This algorithm will of course be implemented inside a loop of N iterations to evaluate average result.

3.1.3 RAM Performance

As discussed with the other components, the same remains for the RAM benchmarking, where the first information to be displayed is the model and the capacity/available capacity of the RAM. Other parameters regarding the RAM's performance are: frequency, bandwidth and latency. To measure the bandwidth of the RAM, 2 arrays will be used with a size large enough to avoid caching. With these 2 arrays the application will perform repeated copy operations and the time elapsed during these operations will be inserted into a formula to compute the bandwidth.

In the case of latency, the same tactic is used in which an array of large enough size is used to bypass caching. Accessing this data and measuring the time it takes to read it will provide the latency of the RAM. The app will also have to take into account the type of access because sequential access is faster due to prefetching while random access is slower, but it better approximates the true RAM latency.

4. Design

In this chapter, the design of the application is described in detail. The concepts introduced in Chapter 3 (Analysis) are applied to define the system's architecture, components, and interactions. We will translate these theoretical aspects into a modular implementation. Each hardware component is represented by a dedicated module responsible for its own information

retrieval and performance testing. As depicted in *Figure 2*, the architecture follows a modular layered model which is divided into 4 main layers.

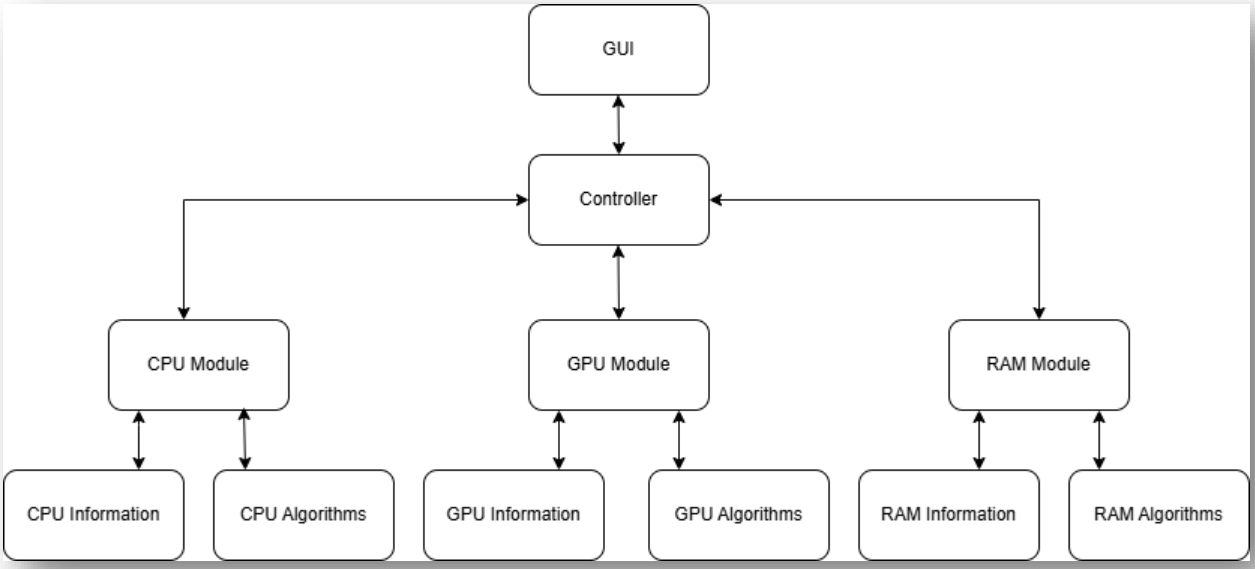


Figure 2. General Architecture Diagram.

The first and outmost layer is the GUI which will collect the information from the Controller and processes and displays benchmark outputs in a structured format. Any test request from the GUI will be transmitted to the Controller which will propagate said request to the appropriate module. The selected module will use its utility classes to compute a result and return it to the controller to be processed. This result is transmitted in turn to the GUI to be displayed.

In the case of this benchmark there will be only one Actor, that being the User using the application. Depicted in *Figure 3* are all the actions a user can perform.

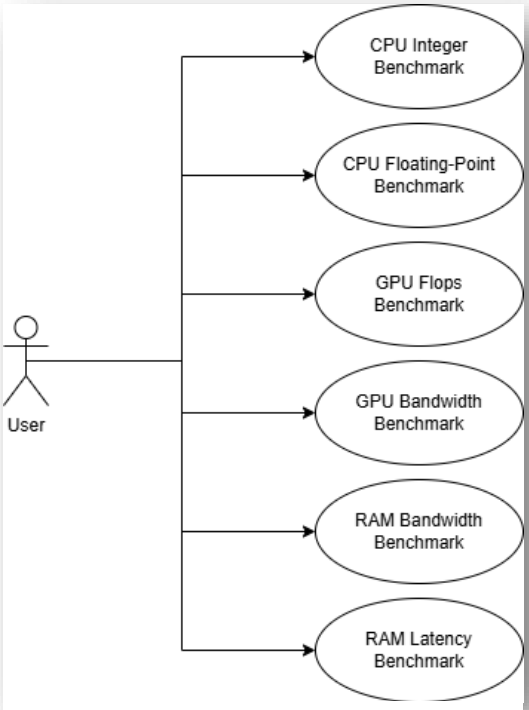


Figure 3. Use Case Diagram.

4.1 Graphical User Interface (GUI)

The GUI represents the topmost layer and serves as the main interaction point for the user. It is designed using a simple and intuitive layout that displays the available benchmark categories and their respective results. These results are computed with a weighted formula that will provide a final score for the computer.

Responsibilities:

- Display available benchmark tests (CPU, GPU, RAM)
- Collect user commands and send them to the Controller
- Display benchmark results in a clear and structured form
- Handle user feedback (loading indicators, progress bars, etc.)

The GUI communicates directly with the Controller using predefined event handlers. This design ensures that the GUI remains independent of the benchmarking logic.

4.2 Controller

The Controller acts as the communication bridge between the GUI and the hardware benchmark modules. It receives user requests (e.g., “Start Benchmark”) from the GUI, delegates them to the appropriate module, and then sends the processed results back.

Responsibilities:

- Handle all benchmark execution requests
- Manage module initialization and cleanup
- Coordinate result collection and error handling
- Ensure synchronization between GUI updates and ongoing tests

4.3 Benchmarking Modules

The benchmarking process is divided into specialized modules, each responsible for evaluating the performance of a particular hardware component. Each module operates as a self-contained package (*Figure 4*), capable of performing tests, gathering information, and returning results through the Controller layer.

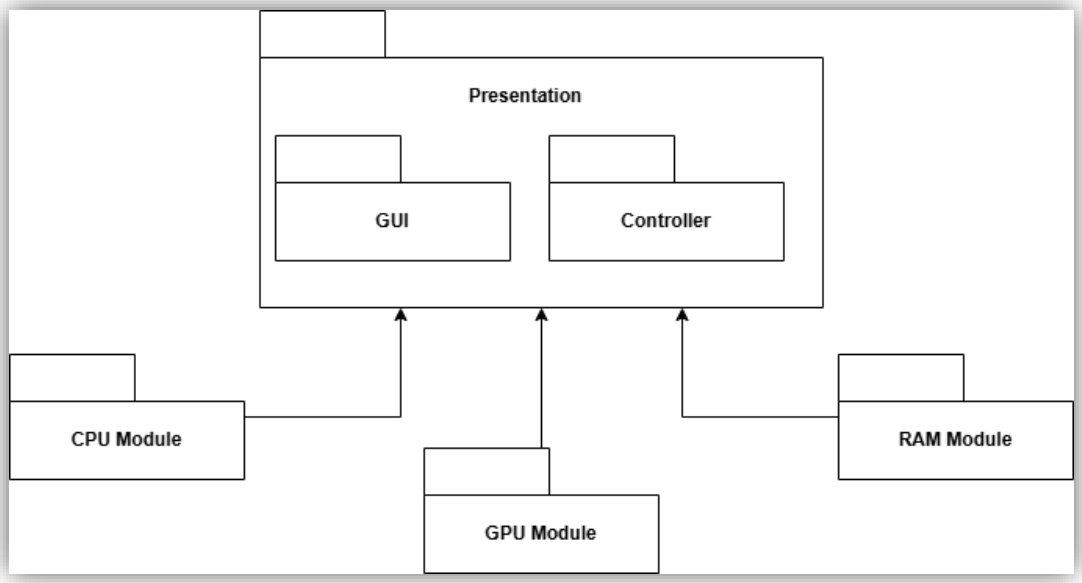


Figure 4. Package Diagram

All benchmarking modules follow a similar structure consisting of:

- A main entry file (e.g., cpu_main.c, gpu_main.c, ram_main.c) which acts as the module's interface and exposes a single public function (e.g., cpu_main(), gpu_main(), ram_main()).
- Utility and helper classes/functions, which handle data processing, mathematical computations, or hardware communication.
- Benchmark core logic, which executes performance tests and records metrics such as execution time, throughput, or bandwidth.

Each module can be executed independently or through the centralized main.c file, which provides a unified entry point for testing multiple hardware components.

CPU Module

The CPU Module focuses on evaluating the processor's computational performance. It contains two main categories of tests:

- The integer benchmark focuses on measuring the CPU's performance in executing algorithmic workloads involving data structure traversal, matrix manipulation, and state machine processing. These tasks represent a balanced mix of control flow, memory access, and computational intensity, providing a realistic estimate of integer performance in typical software applications.
- Floating-point benchmark, which evaluates the processor's ability to handle mathematical calculations, functions and branches involving real numbers.

This module also gathers system information such as CPU model, number of cores, and clock speed. The benchmark is implemented using CoreMark, a widely recognized industry standard for integer performance, and Whetstone, which evaluates floating-point computational efficiency, providing a comprehensive measure of CPU performance.

GPU Module

The GPU Module gathers information about the graphics processor, including its model, number of cores, and clock frequency. It then measures performance using two complementary approaches:

Throughput (FLOPS):

The module estimates theoretical floating-point operations per second (FLOPS) using the formula:

$$FLOPS = GPU\ cores * Clock\ frequency * Operations\ per\ cycle$$

To measure real performance, the module executes multiply-and-add operations in a parallel GPU kernel across many threads. This approach ensures the GPU is fully utilized, avoids underloading cores, and uses GPU timers to minimize timing noise. Synchronization and sufficient kernel runtime ensure accurate measurement of throughput.

Memory Bandwidth:

The module allocates large buffers in GPU memory and performs repeated memory transfer operations such as read-only, write-only, and copy tests. The time taken to complete these transfers is measured over multiple iterations to compute the average VRAM bandwidth. This process helps identify memory bottlenecks and evaluates the GPU's efficiency in handling large data transfers.

This design ensures that both computational and memory performance of the GPU are evaluated in a consistent, repeatable manner.

RAM Module

The RAM Module focuses on evaluating system memory performance and gathers information such as the total capacity and memory frequency. The performance is measured using two primary metrics:

Bandwidth:

Two large arrays are allocated to prevent caching effects. Repeated copy operations are performed between these arrays, and the elapsed time is recorded. Bandwidth is calculated based on the size of the arrays and the total time taken, providing an estimate of the memory's data transfer rate.

Latency:

Using a similar approach, the module measures the time required to access individual elements in a large array. Random-access patterns are tested, as sequential access benefits from prefetching and represents ideal throughput, while random access better approximates true memory latency under real-world conditions.

By measuring both bandwidth and latency, the RAM Module provides a comprehensive view of memory performance, helping identify potential bottlenecks affecting overall system speed.

5. Implementation

The application is implemented in the C programming language using the GCC compiler in order to be very close to the hardware for the best results of the benchmark. Windows API calls (e.g., __cpuid()) are used to gather CPU information like number of cores, threads and clock frequency. To be able to implement these functionalities, libraries “Windows.h” and “intrin.h” are mainly used. In order to have an accurate and easy to use timing method, the application uses a combination of windows functions like clock() and QueryPerformanceCounter().

5.1 CPU Benchmark

The CPU module is implemented in the file cpu_main.c. It retrieves system information such as processor model, number of cores and logical threads using the Windows System Information API (GetSystemInfo) and CPUID instructions.

The benchmarking logic is divided into two distinct workloads: integer performance (CoreMark) and floating-point performance (Whetstone).

The integer benchmark workload is implemented using the CoreMark benchmark. CoreMark executes three algorithmic tests: linked list processing, matrix manipulation, and finite state machine simulation. Each test tries to simulate real life workloads through its operations.

The linked list benchmark evaluates the CPU's ability to handle pointer indirection and random memory access. The linked list data structure is composed of a 16 bit integer(data16) and a 16 bit index(idx). The linked list nodes have a structure named list_head_s with a *next* pointer and *info* pointer. The operations performed on this linked list are: find, mergesort, reverse and remove. These are implemented in the core_list_join.c file.

Matrix manipulation consists of adding constants to a matrix, multiplication with a constant, vector and matrix and it is implemented in the core_matrix.c file. These matrices are constructed out of 16 bit integers.

The simulation of a finite state machine is aimed at testing the flow control and branch prediction. It uses a small Moore machine that parses through a string and determines if it’s a number or something else. It also introduces some corruption and tests for it.

These tests are integrated into a big iteration that runs in the core_main.c file. Their results are combined to produce an overall performance score in iterations per second.

The floating-point behnchmark is implemented in the whetstone.c file and it tests 7 distinct modules:

1. Array Access
2. Conditional Jumps
3. Trigonometry Computations
4. Procedure Calls
5. Array References
6. Transcendental Functions
7. Standard FP Operations

Both benchmarks are combined in the cpu_main.c file that displays the necessary informations and the scores computed as Iterations/Second and Whetstone GIPS.

5.2 GPU Benchmark

To enable hardware-accelerated computation within the benchmarking application, the GPU is initialized using the OpenCL framework provided by Intel oneAPI. During the startup process, the application automatically discovers the available OpenCL platforms in the system and attempts to select a valid GPU device.

The OpenCL API functions clGetPlatformIDs() and clGetDeviceIDs() are used to enumerate platforms and query the first available GPU device on the selected platform. If no compatible GPU is detected, the system notifies the user and exits the application.

Once the GPU device is identified, the application gathers several key hardware parameters that inform the user of the current hardware and also influence performance during parallel computation with the use of the clGetDeviceInfo() function. Alongside this function, the constants CL_DEVICE_NAME, CL_DEVICE_GLOBAL_MEM_SIZE, CL_DEVICE_MAX_COMPUTE_UNITS, CL_DEVICE_MAX_CLOCK_FREQUENCY are used to retrieve the needed information. These parameters are: GPU make and model, memory size, nr. of compute units, max clock frequency.

The performance of the GPU is computed with 2 parameters: the FLOPS and memory bandwidth.

The application measures the number of FLOPS with a kernel that compiles and executes a loop containing floating-point calculations:

```
a = a * b;
a = a + c;
sum += a;
```

b = b + c;

The timing for these operations are calculated with the use of the OpenCL library, specifically with the CL_PROFILING_COMMAND_START and CL_PROFILING_COMMAND_END macros. The number of iterations will be chosen by the user to best suit the hardware. The aim of the benchmark runtime for each test should be about 10 seconds.

For the bandwidth part, the application again launches a kernel with 3 functions of read, write and read+write. A buffer size is chosen in such a manner that it doesn't get cached, but it also doesn't surpass the GPU's memory. It is computed as a quarter of the GPU's memory with a lower bound at 256MB and a higher bound at 1GB.

5.3 RAM Module

The RAM module is implemented in the file ram_main.c. This module gets the basic RAM information and evaluates the performance of the system's main memory by bypassing the CPU caches to measure bandwidth and latency.

After the size and frequency of the RAM are displayed the benchmark tests start. The application uses 512MB blocks in order to bypass caching. The first 2 tests compute the write and copy bandwidth using memset and memcpy respectively.

Measuring RAM latency is challenging because modern CPUs use Hardware Prefetching to guess what data will be needed next. If a benchmark simply reads memory in a predictable pattern (array[0], array[1], array[2]), the CPU will fetch the data into the cache before the program asks for it, resulting in artificially low latency numbers. To solve this, the application implements a Pointer Chasing algorithm. The allocated memory block is treated as an array of integers, where each element holds the index of the next element to visit. The array is initialized with sequential indices and then shuffled using the Fisher-Yates algorithm. This creates a randomized path through memory. With the loop current_idx = list_buffer[current_idx] the application makes sure that the CPU cannot know the address of the next load until the current load is complete.

The score is computed using a non-linear normalization method. A linear often fails to represent the user experience accurately. In real-world usage, the difference between a low-end system and a mid-range system is felt much more intensely than the difference between a high-end and an ultra-high-end system. The application uses an exponent of 0.6 to create a curve of diminishing returns. The formula for the score is:

$$\text{Score} = S_{cap} * \left(\frac{S_{measured}}{S_{reference}} \right)^{0.6}$$

Where:

S_{cap}: The maximum possible points assigned to this specific test (e.g., 4000 points).

$S_{measured}$: The actual result obtained from the specific benchmark.

$S_{reference}$: The theoretical maximum value defined for current high-end hardware.

6. Tests and Experiments

This chapter aims to detail the experimental validation of the benchmark application. With the help of experiments and comparison, the accuracy, reliability and responsiveness of the benchmark across different hardware configurations can be confirmed.

6.1 Performance Core vs Efficiency Core

Since the CPU benchmark is implemented in such a way that it tests a single CPU core, we can test the differences between P-Cores and E-Cores on hybrid CPU architectures. To eliminate skewed results, I ran the benchmark 5 times on the P-Core and 5 times on the E-Core. The CPU that this test was run on is an Intel i7-1255U.

The first big difference we see is that the P-Core has a max frequency of 4.7 GHz while the E-Core has a max frequency of 3.5 GHz.

The average scores for these tests are as follows:

- E-Core CoreMark: 18,806
- P-Core CoreMark: 20,391
- E-Core Whetstone: 6.7 GIPS
- P-Core Whetstone: 10 GIPS

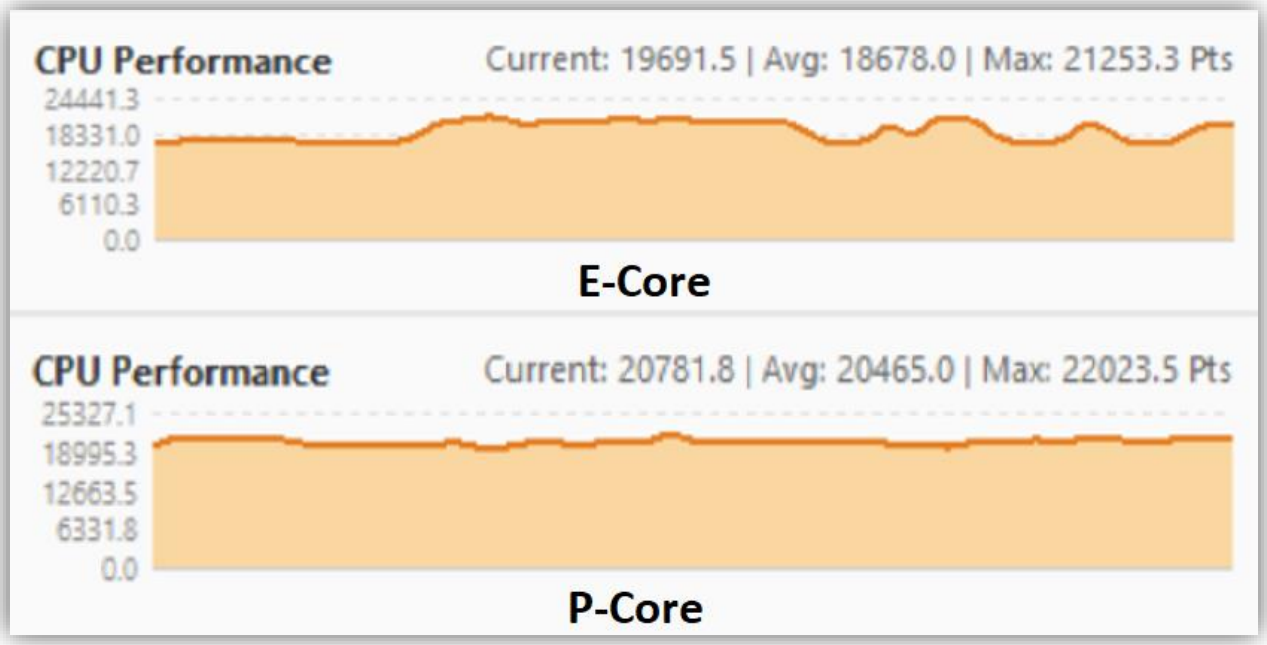


Figure 5. E-Core vs P-Core consistency graph

Looking at *Figure 5* confirms that the P-Cores are not just faster than E-Cores, but also more stable and consistent through heavy computation.

6.2 PC vs Laptop

When comparing a PC to a Laptop we can observe some key differences. By looking at *Figure 6* and comparing it with *Figure 7* we reach some interesting conclusions. First of all, even though the PC's CPU is much older than the Laptop's, the performance of the cores are almost the same. By this information alone it cannot be said that they have the same computing power. If we dig deeper, we see that the Laptop's CPU has almost triple the amount of cores and threads meaning that parallelism is much more efficient on the Laptop than on the PC. Second of all, which is also the elephant in the room is the comparison between graphics chips. It is very clear from these results that even a 10 year old dedicated GPU is about 4 times as faster as an integrated Laptop GPU. Of course, as in any benchmarking application, we must take this with a grain of salt, because these results do not exactly mean the PC GPU is 4 times better than the Laptop's GPU because the Laptop's GPU is designed in such a way to reduce energy consumption and space. Lastly, looking at the RAM performance, we see something unexpected. Even though the Laptop's RAM works at a higher frequency, the write speed and latency are slower. This does not necessarily mean that the Laptop's RAM is worse because if we investigate what could cause this, we reach the conclusion that the integrated GPU does not have its own

separate VRAM and it uses the available RAM for its storage which causes this decline in performance.

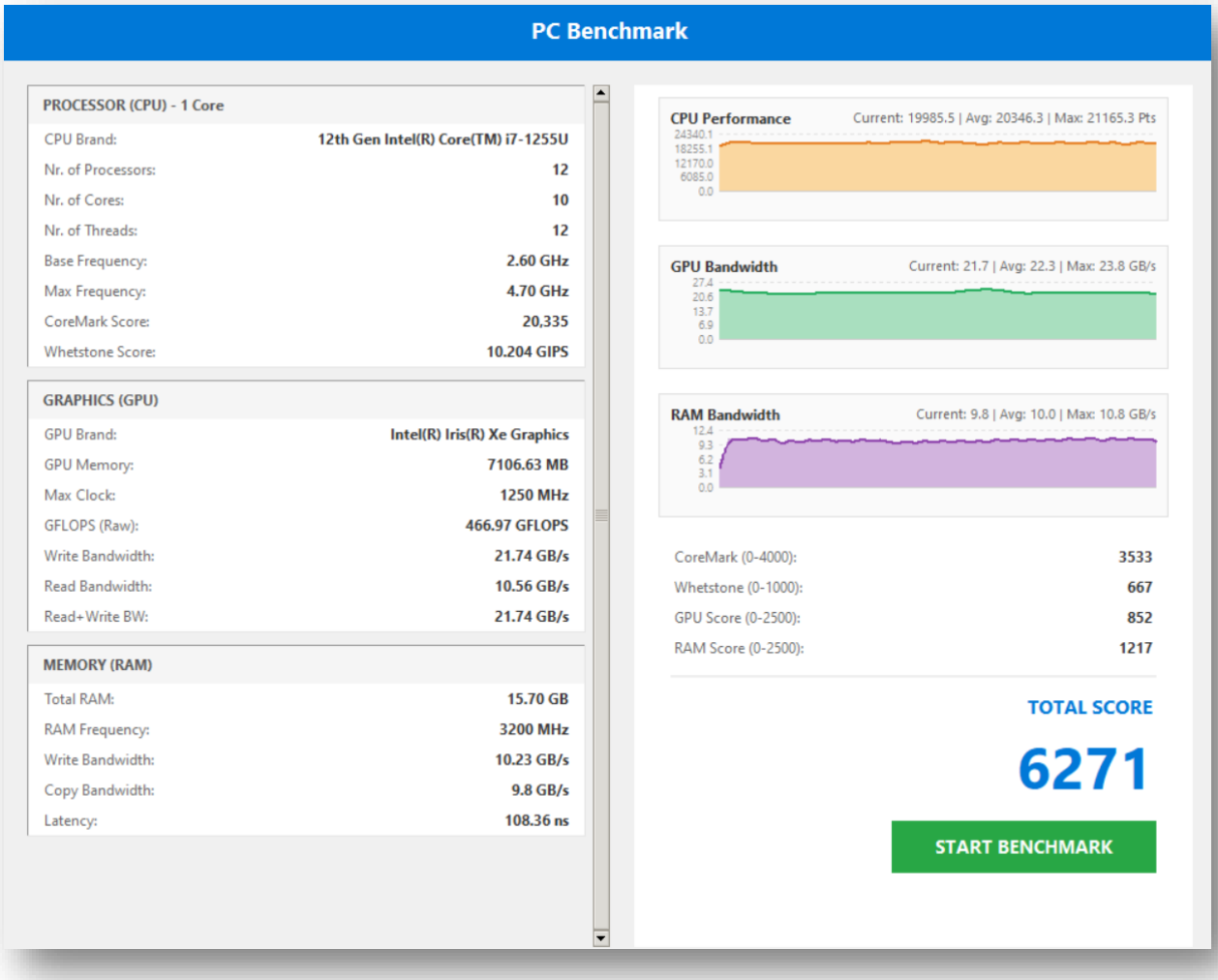


Figure 6. Laptop Benchmark Results (P-Core)

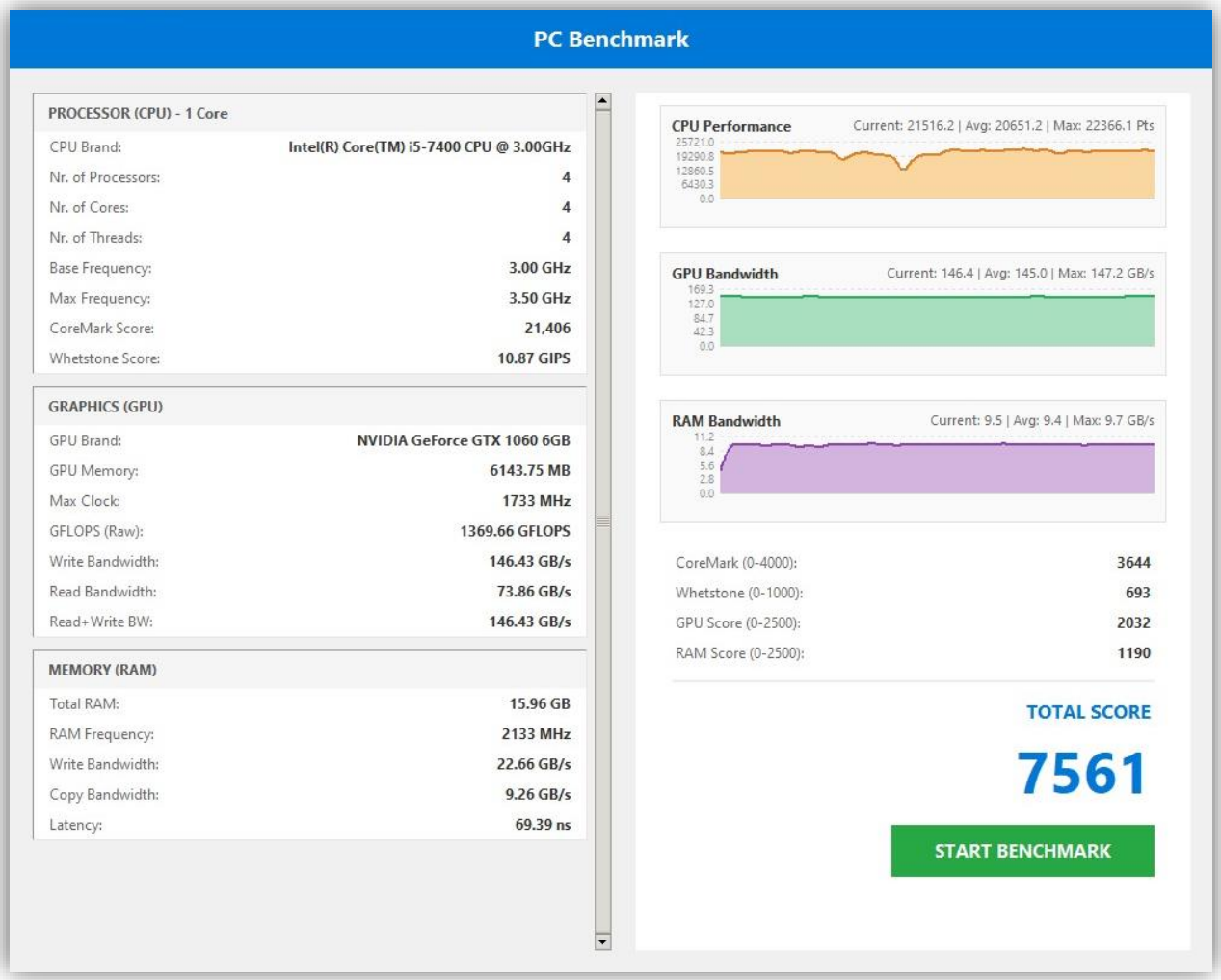


Figure 7. PC Benchmark Results

7. Conclusions

The application successfully incorporates the primary objectives of creating an accessible, lightweight and accurate testing tool for the average user while maintaining a low-level, in-depth implementation. By utilizing the C programming language and Windows API calls, the project achieved a high degree of hardware proximity, ensuring precise measurement of performance metrics. By analyzing the tests, we can see than an above-average laptop has ~6000 score out of 10000 which is exactly what one would expect out of a benchmark.

7.1 Further Improvements

For the application to reach completeness, there are several key upgrades that need to be implemented.

First, multi-core benchmarking for the CPU would be one of the first improvements that need to be introduced. A benchmark that has single-core and multi-core tests can offer a broader picture about the hardware.

Another key feature would dynamically change the score caps for the benchmark results based on what category the user wants to benchmark (e.g. Low-End, Mid-End, High-End, Ultra-High-End).

Lastly, a good quality of life feature would be portability. With the current implementation, the benchmark only works on Windows systems. In order to appeal to a bigger userbase, the application should be able to run properly on all popular operating systems.

References

[1] Dirk Beyer, Stefan Löwe, Philipp Wendler, Reliable benchmarking: requirements and solutions, 2019, [https://www.sosy-lab.org/research/pub/2019-STTT.Reliable_Benchmarking_Requirements_and_Solutions.pdf]

[2] Time-Stamp Counter: https://en.wikipedia.org/wiki/Time_Stamp_Counter [accessed Oct. 2025]