PC BENCHMARK APPLICATION

Student Name: Cristea Tudor

Group: 30433

CONTENT

1) INTRODUCTION

1.A) Project Proposal

The principal objective of this project is to create a desktop application developed mainly in *Java* and *C/C++* programming languages that will make use of various libraries, methods and/or frameworks that are specific to the testing process of every component. The application is capable of displaying information about the hardware and software of the computer, running a benchmark on the system that consists of several tests which are meant to measure the performance of some of the components of the computer and finally presenting some scores that are obtained from the benchmark and that are indicative of the system's performance.

It mainly consists of one window in which details about the components of the system are shown. Below, the user can find a few buttons which, when pressed, will trigger the benchmark tests and obtain the corresponding score. These scores can be subsequently used in order to compare, contrast and analyze the performances of different computers or focus on a certain resource. The components that are benchmarked include the CPU, GPU, Memory, Disk and Network.

This application is intended to component manufacturers, computer building firms and regular users alike that wish to test the capabilities of their systems. This can assist any of the categories of users in the process of optimizing the entire system or a single, particular component or with finding potential bottlenecks that are negatively affecting the overall performance of the whole system.

In conclusion, this project aims to deliver a robust computer benchmarking application that provides users with valuable insights into their system's various performance metrics. By utilizing a combination of *Java* and *C/C++*, the aim is to achieve efficient and accurate benchmarking test results, thus enhancing the overall user experience.

1.B) Project Plan

The plan consists of the following steps:

a) Weeks 1-2
- Research and learn about PC benchmarking in general and about the methods which are used to benchmark several components of the system including the CPU, GPU, Memory, Disk and Network
- Set up the appropriate development environment (including necessary initial dependencies, libraries and/or frameworks)

b) Weeks 3-4
- Research and learn about how to obtain the details of the components of the PC in order to be displayed inside the application
- Begin the implementation of this part of the project

c) Weeks 5-6
- Finish the implementation of the part of the project that was researched and started in weeks 3-4
- Check if the information provided by the application regarding the system's components is correct and accurate
- Research and learn about how to create and run tests for each computer component and choose the most useful tools and libraries available for *C/C++*

d) Weeks 7-9
- Implement the part of the project that was researched in weeks 5-6
- Test if each benchmark test runs correctly and accordingly (with the algorithms implemented)

e) Weeks 10-11
- Research and learn how to measure the performance of these components during the tests (how many and which metrics are important) and establish a relevant and applicable scoring system, and subsequently, implement this part of the project

- Design and implement an intuitive and user-friendly UI (using *Java Swing*) for displaying details about the existing components of the system as well as the scores resulted from running the benchmarking tests

f) Weeks 12-13
- Check the ease of use and the conveniency of the application and fix visual bugs
- Run the application on a couple of computer systems and analyze and compare the scores of the benchmark

The documentation will be appended with new information and/or the existing information will be modified as the development of the application progresses in parallel. Rigorous testing and debugging will also be performed constantly, at each step, in order to ensure the correctness of the application and establish a smooth and risk-free development process.

2) BIBLIOGRAPHIC STUDY

A benchmark consists of a series of software tests that are ran on a piece of hardware. It replicates the kinds of tasks a PC would perform in real-world, everyday usage. It is supposed to give the user/customer an idea of the overall performance of the system. Some (tech-savvy) people take these benchmark results into consideration when they are faced with the decision of buying a computer, laptop or smartphone. They are particularly good at showing how performance improves from one generation to the next and can help in determining the value for money of a product since you can easily see how it compares to similarly-priced alternatives. Among the most well-known benchmarking applications, there is *AnTuTu*, *Geekbench*, *3DMark*, *HWMonitor*, *PCMark*, *Cinebench* and *AIDA64*. In addition, even some modern PC games offer the ability to run a specialized benchmark in order for the player to choose the right settings and play the game as efficiently as possible. Many components of a PC can be benchmarked, including the CPU, GPU, Memory, Networking capabilities, I/O elements and Storage, but some might matter more than others depending on the situation.

In benchmarking, there are seven vital principles/characteristics, namely ([4]):

1. **Relevance**: Benchmarks should measure relatively vital features
2. **Representativeness**: Benchmark performance metrics should be broadly accepted by industry and academia
3. **Equity**: All systems should be fairly compared
4. **Repeatability**: Benchmark results can be verified and should yield roughly the same results
5. **Cost-effectiveness**: Benchmark tests should be economical
6. **Scalability**: Benchmark tests should work across systems possessing a range of resources from low to high
7. **Transparency**: Benchmark metrics should be easy to understand and analyze

In addition, there are several types of benchmarking such as ([1]):

1. **Real program**: the list includes compilers, word processing software, video games
2. **Component Benchmark/Microbenchmark**: it consists of a relatively small and specific piece of code; it is used to measure the performance of a computer's basic components; it may also be used for automatic detection of a computer's hardware parameters including the number of registers, cache size or memory latency
3. **Kernel**: it contains codes that perform a specific, basic operation; it is normally abstracted from the actual program; the results are represented in Mflop/s
4. **I/O Benchmark**
5. **Database Benchmark**: it measures the throughput and response times of database management systems (DBMS)
6. **Parallel Benchmark**: it is used on machines with multiple cores and/or processors, or on systems consisting of multiple machines
7. **Synthetic Benchmark**: each operation of the application programs has a proportion associated to it; it was the first kind of benchmarks that became general purpose industry standard benchmarks; there are

two possible types: *Whetstone* (consists of a combination of integer instructions only) and *Drystone* (consists of a combination of integer instructions as well as floating point instructions);

Unfortunately, benchmarks are far from perfect and the main issue is that a lot of manufacturers optimize the processor's architecture for the sole purpose of having high scores in benchmark application leaderboards, but that degree of performance is not reflected in everyday usage, thus being misleading for the average customer.

When it comes to incorporating the benchmark results into a formula in order to obtain a single score indicative of the overall performance of the system, there are four possible such formulas, namely ([1]):

1. **Arithmetical Mean**

$$B_{AM} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

2. **Weighted Arithmetical Mean**

$$B_{AM} = \frac{1}{n}\sum_{i=1}^{n} w_i * t_i$$

3. **Geometrical Mean**

$$B_{GM} = \sqrt{\prod_{i=1}^{n} t_i}$$

4. **Weighted Geometrical Mean**

$$B_{GM} = \sqrt{\prod_{i=1}^{n} w_i * t_i}$$

(t_i – execution time of program i, w_i – weight of the program i, indicating the frequency of its execution, $i \in [1,n]$)

Usually, the weight is chosen such that on a reference computer, the execution time of each benchmark is equal. This is called NORMALIZATION. When referring to the geometric mean formulas, the advantages are that they are independent of the running times of individual programs and that it does not matter which of the machines is used for the process of normalization, but the main disadvantage is the fact that they do not predict the execution time.

3) ANALYSIS

PC benchmarking implies the testing of several computer components such as the CPU, GPU, Memory, Disk and Network. Moreover, for testing each and every one of these components, one needs a few algorithms that are specific for that component. In order to correctly and accurately measure the performance of these algorithms, one needs to use non-recursive implementations and those implementations need to use a minimal number (ideally, zero) of method calls to other functions, during the testing period of that particular algorithm (unless it is absolutely necessary). In this way, the overhead and the stack usage are minimized, thus offering a more precise performance measurement of the algorithm itself, since it excludes the execution time of other auxiliary operations.

For the application itself, an MVC inspired architecture is used in order to better separate the user interface which represents the front-end of the program from the implementation details and the underlying logic that represent the back-end of the program. This way, high cohesion and loose coupling are obtained. When it comes to the UI itself, it consists of a main window, separated into three sections, which are presented in more detail, below.

When the user opens the application, they are greeted with a friendly interface in which the details of their system's hardware components are presented in an organized way. This information is found in the first section of the application window.

When it comes to establishing a score that is representative of the system's performance, a weighted arithmetic mean is considered, but the weights are chosen by the user of the application (before the benchmark is launched), since they know better which components matter to them more than others. However, there are two restrictions/conditions that the user has to abide by. The first one is that the weights can only have a floating point value between 0 and 1, and can have, at most, two decimals. If the user chooses, for one of the components, a weight equal to zero, then that component's tests are not executed (since the resulting score would be multiplied by 0 anyway). The second one refers to the fact that the sum of these weights has to be exactly 1, in order to obtain a valid score. If the user does not respect any of these rules, then an error message is displayed on the screen, informing them of the mistake they committed and the way they can fix it. This represents the second section of the main application window.

Once the benchmark is launched, the user has to wait until it is finished and only then see the scores that were obtained for each component that was tested as well as the total score. This information is displayed in a new window, while the main window is going to be disabled (non-interactable) throughout the entire duration of the testing process. Moreover, the user is restricted from interacting with the main window in order to ensure a smooth run of the benchmark without any unwanted interruptions or having multiple instances of the same test happening simultaneously.

Additionally, a log of all the benchmark runs is saved in a *.txt* file, every line of it containing not only the obtained score, but also details about the components of the system, such as name, frequency or capacity as well as software information such as OS name, OS version and OS architecture. In this way, the user can identify possible bottlenecks or if they want to optimize a certain component, they can examine the evolution of that procedure, by comparing all the obtained results of the same benchmark. Accessing the contents of the log file directly from the application is done by pressing a button that is found in the third section of the application window. Once pressed, a new window is opened containing a table that includes minimal, but relevant information from the log file, while again, the main window is not going to be user-interactable as long as this new window stays open.

Taking everything that was just discussed into consideration, the use case diagram of the application is presented below:

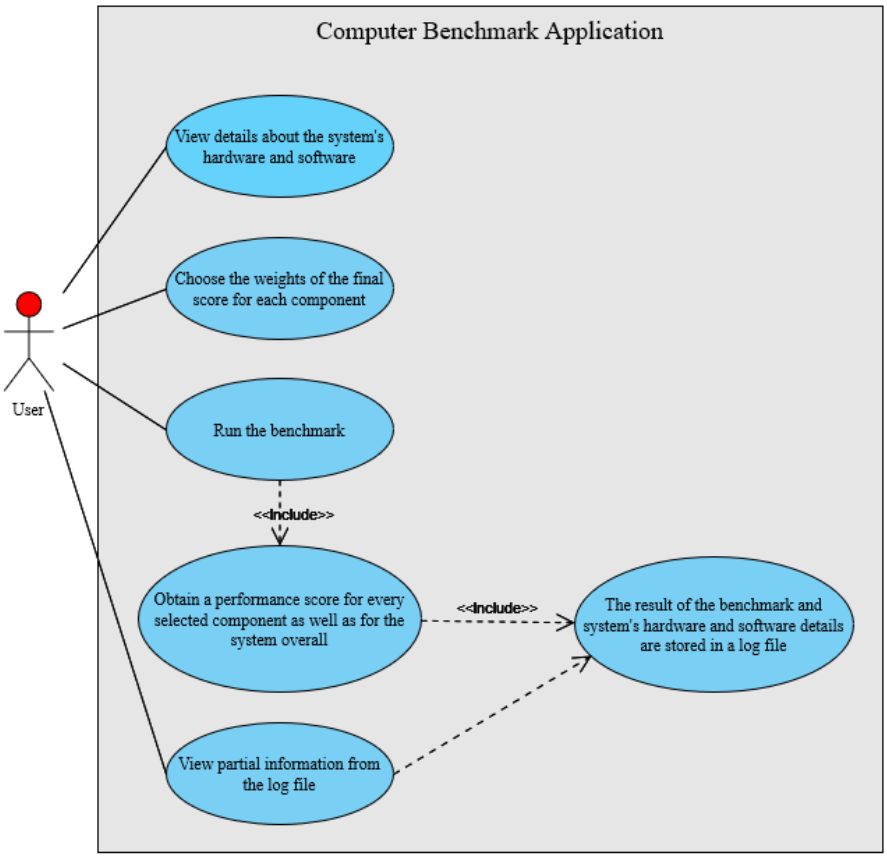


Fig. 1 - Use Case Diagram

a) Benchmarking the CPU

For benchmarking the CPU, one needs to benchmark several operations that work with different data types including *int*, *float/double* and *char/char[]*. Moreover, for the integer and floating point data types especially, one needs to test as many different operations as possible including addition/subtraction (+, -), multiplication (*), division (/, %), comparison operations (<, <=, >, >=, ==, !=) and assigning operations (=). Therefore, a number of algorithms which have different time complexities are carefully chosen to be implemented in this regard. All of these algorithms are run twenty times, the first few of which are considered warm-up and thus their scores are not taken into consideration when computing the CPU score while for the rest of them, their scores matter and are accounted for in the CPU result. This is called “Microbenchmarking”, whereas what is done on each primitive data type is called “Kernel benchmarking”, since the focus is on a certain operation(s).

b) Benchmarking the GPU

For benchmarking the GPU, one needs to benchmark a 3D scene, in which one or multiple objects are performing the basic graphical operations, i.e., rotation, translation and scaling. Another aspect that can be tested and measured is the lighting manipulation, and interaction with the object(s) of a scene. In addition, a peculiar aspect that can also be tested and measured refers to a particle system (that respects some physics laws). For this, the OpenGL rendering pipeline is used, since it is well-known, programmer-friendly and accessible. In order to assess the performance of these algorithms, a certain number of frames has to be rendered. This happens five times, the first few of which are considered warm-up, while the others are accounted for in the GPU score. This type of benchmarking is part of the “Real Program” benchmark category since it renders objects in a similar way as it would be done inside a video game or other graphical applications.

c) Benchmarking the Memory

For benchmarking the Memory, one needs to benchmark dynamical allocation (done in several different ways) and deallocation of memory. These algorithms are run twenty times, the first few of which are considered warm-up and thus their scores are not taken into consideration when computing the Memory score while for the rest of them, their scores matter and are accounted for in the Memory result. This type of benchmarking is called "Microbenchmarking", but it can also be considered to be a part of the "Real Program" benchmark category if the use of custom memory pools is introduced.

d) Benchmarking the Disk

For benchmarking the Disk, one needs to benchmark reading and writing from and to a (text) file that is located on the disk. These algorithms are run twenty times, the first few of which are considered warm-up and thus their scores are not taken into consideration when computing the Disk score while for the rest of them, their scores matter and are accounted for in the Disk result. This type of benchmarking is called "Real Program" benchmarking since it interacts with actual files found on the computer's hard drive (or solid state drive) or it could be considered as a form of "I/O Benchmark".

e) Benchmarking the Network

For benchmarking the Network, one needs to benchmark the download speed of the current internet connection. This algorithm is run twenty times, the first few of which are considered warm-up and thus their scores are not taken into consideration when computing the Network score while for the rest of them, their scores matter and are accounted for in the Network result. This type of benchmarking is called "Microbenchmarking", since it won't actually measure the real download speed, because that would imply a very complex algorithm that sends requests to multiple servers located in different places and also takes into account other aspects such as network congestion.

4) DESIGN

For the application itself, the following package diagram and class diagrams represent a starting point of how the application's structure is going to look like:

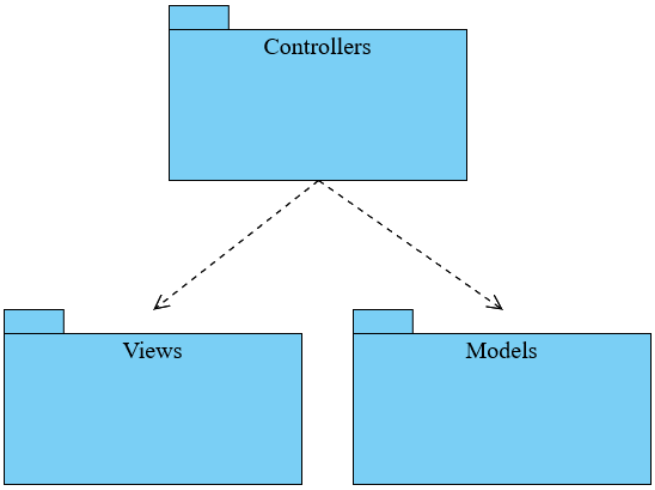


Fig. 2 - Package diagram

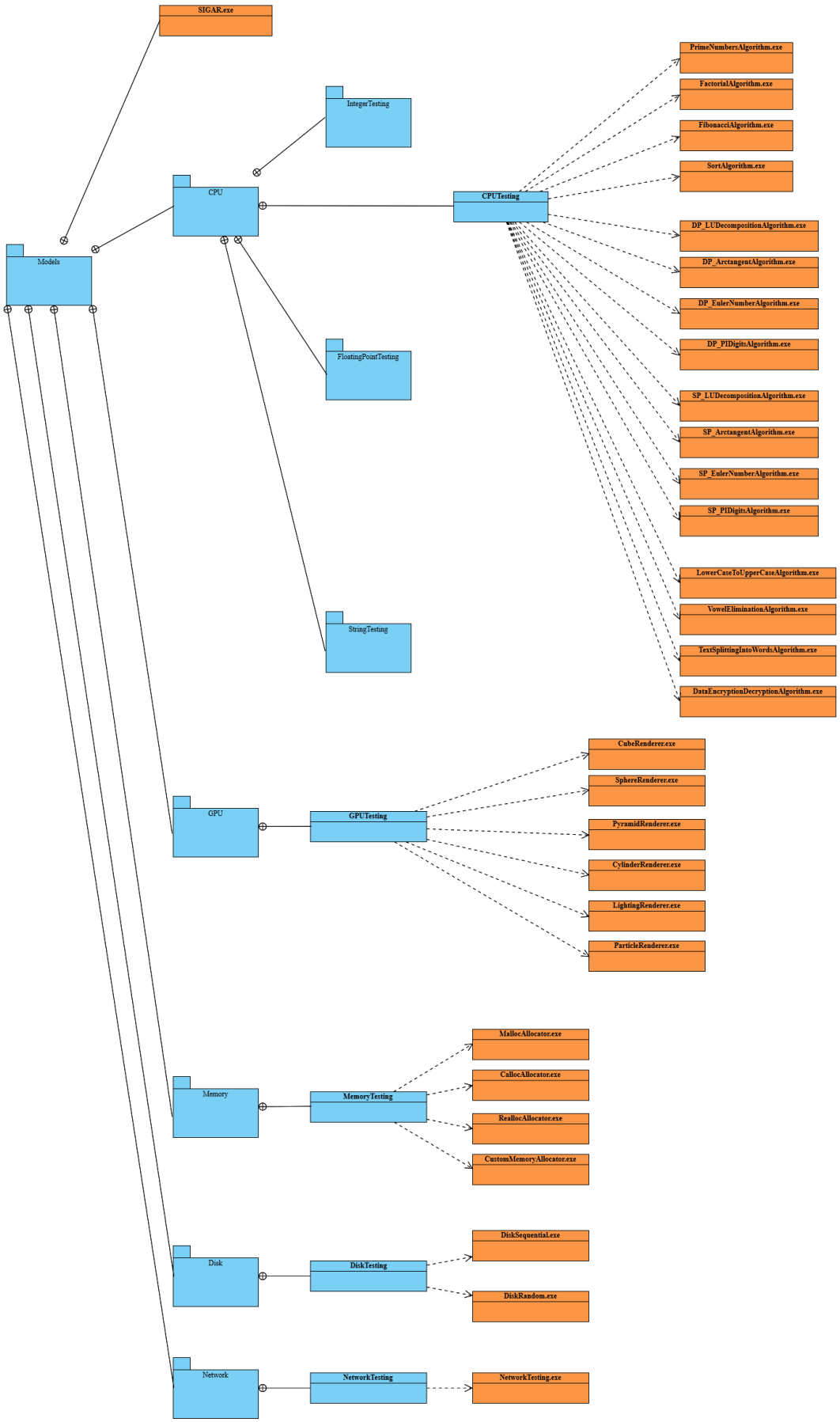


Fig. 3 - Models Class Diagram

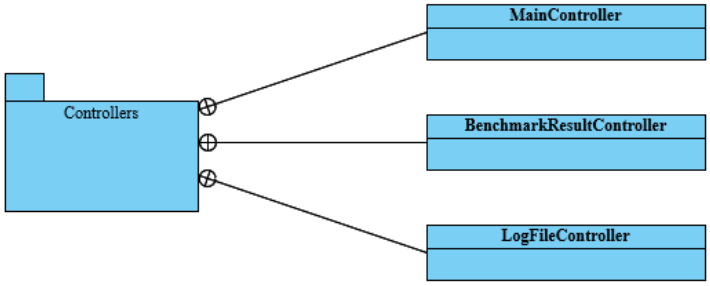


Fig. 4 - Controllers Class Diagram

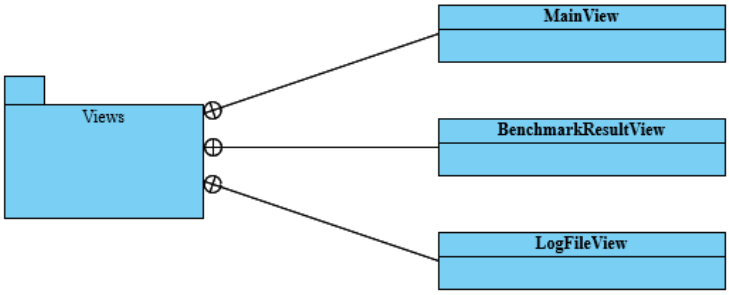


Fig. 5 - Views Class Diagram

The measurement of all the following algorithms is done by using the *chrono::high_resolution_clock::now()* function found in the *chrono* library from *C/C++* in order to memorize the approximate processor time that is consumed by the program of two different moments, i.e., the starting time and the ending time, the difference of these two timestamps representing the so-called elapsed time of the algorithm, to which another function, namely *count()* (from the same library) is applied ([12]). Moreover, the algorithm will repeat the same operations on the same data sets until a second has passed (for CPU, Memory, Disk and Network) or until five seconds have passed, and the result is divided by five (for GPU). Thus, in the end, a score number that does not have few, nor many digits, is obtained. In this way, it is easier for the user to compare the performances of two different systems and allows for the inevitable further growth of future systems to be able to be represented by this score number.

When it comes to establishing a score, obtaining the median of all the scores is a more accurate result and representation of the performance of a component than the average, since, in the computation process of the latter, warm-up scores, as well as, "spike" values (up spikes and/or down spikes) are also included. The process of eliminating these erroneous values from the computation such that one obtains an accurate value of the average, is unnecessarily complicated and rather tedious. That's why using the median approach is a very simple and elegant way to obtain a good result (the values are inserted into a list that is sorted and then, the middle value is taken). The median generally gives a more accurate value when it works with skewed data, i.e., when there are "spikes" among the results. The only situation in which the median does not give a correct result, even on skewed data, is if there are multiple clusters of values, which, I believe, is not the case here. Another metric that could have been used for the score is represented by the minimum of all scores, since this would give the "best case" score for a system. However, I believe this metric is a bit extreme and does not really portray the real performance of the computer that is tested ([13]).

a) Benchmarking the CPU

The following remark needs to be made in advance: the following integer algorithms' time complexities are mentioned by using the "Big Oh" notation (*O*). This is done because each of the scores for integer processing, but also floating point processing and string processing have a separate formula which is calculated by using the weighted arithmetical mean in order to be able to put more emphasis on the algorithms that have a higher time

complexity than the others. In addition, the measurement of these algorithms is done in MIPS (millions of instructions per second) by counting the number of assignments and comparisons that are made throughout the algorithm and then, dividing that number by a million.

The algorithms that are chosen for testing the CPU's performance of working with integer operators are the following:

- **Prime Numbers Generation**: this algorithm generates and counts how many prime numbers exist from zero to an upper limit number which is specified by the programmer (in this case, it is 100,000); this algorithm focuses more on the integer division and modulo operations; this algorithm has a time complexity of $O(n* \sqrt{n})$ and a weight of 0.3 in the integer formula;
- **BubbleSort**: this algorithm sorts an array of 10,000 elements which were randomly generated (this operation was not taken into account during the measurement procedure); this algorithm focuses more on the comparison and assigning operations; this algorithm has a time complexity of $O(n^2)$ and a weight of 0.4 in the integer formula;
- **Fibonacci Numbers Generation**: this algorithm generates a certain number of Fibonacci elements, specified by an upper limit parameter, that is a constant set by the programmer (in this case, it is 100,000); this algorithm focuses more on the addition operation; this algorithm has a time complexity of $O(n)$ and a weight of 0.15 in the integer formula;
- **Factorial Number Computation**: in order to be somewhat on par with the number of computations of the other algorithms, this one computes the factorial until the maximum value of the "*unsigned long long"* data type is exceeded, thus triggering an overflow and obtaining the value zero, which corresponds to the moment that the algorithm stops; this algorithm focuses more on the multiplication operation; this algorithm also has a time complexity of $O(n)$ and a weight of 0.15 in the integer formula;

When it comes to algorithms that primarily work on floating point numbers, the focus is not on covering all of the basic operations, but rather on the algorithms themselves, since they are now more involved with the field of mathematics, namely mathematical analysis. The algorithms that are chosen for testing the CPU's performance of working with floating point operators (both single precision and double precision) are the following ([3]):

- **LU Decomposition**: this algorithm, which is based on the Doolittle algorithm ([25]), makes the LU decomposition of a given square matrix which has a dimension of *n*, a constant parameter that is chosen by the programmer (in this case, it is 100); this algorithm has a weight of 0.25 in the floating point formula;
- **Arctangent Angle Computation**: this algorithm takes a tangent (which has to be equal or larger than 1) and by using the arctangent function which is expressed as an approximation formula that uses a Maclaurin series, computes the angle of that given tangent [(26)]; the approximation becomes more accurate as the number of terms in the series, which is a constant parameter that is set by the programmer (in this case, it is 100,000) increases; this algorithm has a weight of 0.25 in the floating point formula;
- **Euler Number Computation**: this algorithm computes an approximation of the Euler number (e) by using a Taylor series expansion ([27]); again, the approximation becomes more accurate as the number of terms in the series, which is a constant parameter that is set by the programmer (in this case, it is 100,000) increases; this algorithm has a weight of 0.25 in the floating point formula;
- **PI Number Computation**: this algorithm uses Leibniz's formula ([28]) in order to compute an approximation of the PI number (π), that implies another summation of terms of a series and as the number of terms in the series, which, again, is a constant parameter that is set by the programmer (in this case, it is 100,000) increases, so does the accuracy of the computed result; this algorithm has a weight of 0.25 in the floating point formula;

When it comes to algorithms that test the CPU's capability of working with strings, their focus is on text and individual character manipulation. Obviously, string specific functions found in the standard *C/C++* library are NOT used in any of the implemented algorithms. In this regard, the following algorithms are chosen in order to assess this performance metric:

- **Lower Case To Upper Case Text Transformation**: this algorithm takes a randomly generated 100,000 character text from a *.txt* file, which is only composed of letters and converts every lower case letter into its upper case counterpart by subtracting a constant (32) from it; this algorithm has a weight of 0.2 in the character string formula;
- **Vowel Elimination**: this algorithm takes a randomly generated 100,000 character text from a *.txt* file, which is only composed of letters and every time a vowel character is detected by the algorithm, it is deleted; this algorithm has a weight of 0.2 in the character string formula;
- **Text Splitting Into Words**: this algorithm takes a randomly generated 100,000 character text from a *.txt* file, which is only composed of letters and spaces, and splits it into singular words; this algorithm has a weight of 0.2 in the character string formula;
- **Data Encryption/Decryption**: this algorithm takes a randomly generated 100,000 character text from a *.txt* file and by using an encryption key, applies a formula to it and writes it back to the same file; afterwards, that text file is accessed again and decrypted back to its original form; this algorithm has a weight of 0.4 in the character string formula;

b) Benchmarking the GPU

The measurement of these algorithms is done in "number of rendered frames per second" by counting the number of frames that are rendered in five seconds and then, dividing that number by five. In this regard, the following algorithms are chosen in order to assess this performance metric ([7]):

- **Cube Rotation**: in this scene, a cube that has different color faces is continuously rotated in several different ways (around both the X-axis and the Y-axis), but its position does not change from the center of the rendering window, nor does it scale; this algorithm has a weight of 0.1 in the GPU formula;
- **Sphere Scaling**: in this scene, a sphere continuously grows and shrinks in size on some or all axes simultaneously, but it does not change its position from the center of the rendering window, nor does it rotate; this algorithm has a weight of 0.1 in the GPU formula;
- **Cylinder Translation**: in this scene, a cylinder is continuously changing its position in the rendering window, but it does not rotate nor does it scale; this algorithm has a weight of 0.1 in the GPU formula;
- **Pyramid All Transformations**: in this scene, the GPU is more intensively tested since all three transformations are applied to the same object simultaneously; thus, a pyramid is rotating (only around the Y-axis), and is also changing both its size (in all axes) and location in the rendering window all at the same time; this algorithm has a weight of 0.35 in the GPU formula;
- **Torus Lighting**: in this scene, a static torus (donut) is continuously rendered alongside a moving light that is alternatively rotating around the X-axis or the Y-axis of the object/scene; this algorithm has a weight of 0.2 in the GPU formula;
- **Particle System Generation**: in this scene, a particle system is generated and rendered in which a sizeable number of little 3D spheres are bouncing around (colliding with other particles or with the "walls"); this algorithm has a weight of 0.05 in the GPU formula;

c) Benchmarking the Memory

All of the algorithms that are presented next, work based on the following framework: allocation of a large amount of memory of a single size and its deallocation, followed by the allocation of a large amount of memory of different sizes and its deallocation and ended by the allocation of a smaller amount of memory and its deallocation in a loop. The measurement of these algorithms is done in "millions of memory operations per second" by counting the allocation, deallocation, reading and writing operations and the, dividing that number to a million. The following methods are chosen in order to measure the performance of this component ([5], [6]):

- **Allocation Using *malloc()***: in this algorithm, four arrays, one of the *char* data type (one byte), one of the *short* data type (two bytes), one of the *int* data type (four bytes) and one of the *long long* data type (eight bytes) are used inside the aforementioned framework; this algorithm has a weight of 0.15 in the Memory formula;
- **Allocation Using *calloc()***: in this algorithm, the same four arrays are used as before, with the only difference being in the function that is used to allocate the memory; this algorithm has a weight of 0.15 in the Memory formula;

- **Allocation Using *realloc()***: in this algorithm, the same four arrays are used as before with the exception that, for each step of the framework, the memory is allocated and then reallocated, the time being measured only for the reallocation and deallocation part of the code; this algorithm has a weight of 0.15 in the Memory formula;
- **Allocation Using A Custom Memory Pool**: in this algorithm, a custom pool of memory blocks is used inside of the aforementioned framework instead of relying solely on general-purpose memory allocation functions (*malloc()* and *free()*); this algorithm has a weight of 0.55 in the Memory formula;

d) Benchmarking the Disk

The two algorithms that are presented next, work based on the following framework: a text file is opened or created (if it does not already exist) and, for one second, blocks of data of a fixed size are written to it and then for a certain amount of time (which might not be exactly one second), the contents of the file are read (in the same sized blocks) until the end of file is reached. In order to ensure that the reading and writing operations are correct, the file that is used for them, is opened in "binary mode". This way, characters such as newline ("\n") are counted as a byte, as opposed to opening a file in "text mode" where the newline character is counted as two bytes, because carriage return ("\r") is also counted. Therefore, the results that are obtained, are more accurate. The measurement of these algorithms is done in "megabytes per second" by multiplying the number of writes/reads with the size of the block. The following methods are chosen in order to measure the performance of this component ([18], [19]):

- **Sequential Access**: in this algorithm, the writing and the reading are done in a sequential manner, meaning from the beginning of the file until the end of it (the size of the file is variable); this algorithm has a weight of 0.5 in the Disk formula;
- **Random Access**: in this algorithm, the writing and reading are done in a random manner, meaning that random locations in the file are chosen for both these operations ([29], [30]), but for this purpose, the size of the file has to be fixed; this algorithm has a weight of 0.5 in the Disk formula;

e) Benchmarking the Network

The only algorithm that is chosen for testing the performance of the Network component consists of sending an HTTP request to a mock-up server from which a download operation is performed. The measurement of this algorithm is done in "bytes per second" by dividing the total size of the download to the number of HTTP requests in which these multiple downloads were performed.

5) IMPLEMENTATION

Each of the algorithms presented in the previous section is implemented separately into its own source file, and thus, for each of them, a separate executable file is generated and used for the project.

In order to run these executable files from *Java*, the *Runtime.getRuntime().exec()* method is used to create one instance of a *Process* object (and of an actual process) for every executable that is used, as if the executables were run from the command line. As an argument, the absolute path to the executable file is given when one of these objects is created. Moreover, in order to run every algorithm for a number of times in a sequential manner, the *waitFor()* method is used such that to ensure these process instances do not run in parallel ([22]). In addition, in order to ensure a means of communication between the executables and the *Java* program, text files are used. These are created (if they do not already exist) and written to by the *Java* program using the *FileWriter* class or appended by the executable files, while the process of reading the execution scores is done using the *FileReader* class.

For obtaining information about the hardware components of the system that are tested, WMI is used. Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems ([17]). A program is written in *C/C++* and the relevant information about the CPU, GPU, Memory, Disk and Network components, as well as details about the operating system, is extracted/queried from the WMI and then written inside two separate text files (one for hardware and the other for software), from which the Java program extracts it and places it into the UI of the application and/or in the log file that is appended at the end of each benchmark ([8], [9], [10], [11], [12], [13], [14]).

In order to display the information of the log file in a nice and neat manner, a neat separator is used for the text file itself in order to split all the information that is recorded, such as: timestamp (including date and time), details about the hardware components that were also displayed in the main window of the application, details about the software, the weights that were chosen by the user for each component in the benchmark, as well as all the scores of the benchmark (individual component scores and the total score). However, for the application itself, only a few of these aspects are shown inside a *JTable*, namely the timestamp, the weights and the scores, since all the other details can be seen either in the log file or in the main window of the application (or both).

For the GPU testing algorithms, the *GLUT* library is used since it is a more simplified and programmer-friendly version of the *OpenGL* library that greatly facilitates the development of graphical applications by hiding/masking the complexity of the *OpenGL* library into easy-to-use functions ([23], [24]).

For the custom memory pool allocator algorithm, a special structure is defined. It includes the size of a block, the maximum number of blocks (capacity), the data itself and a list of pointers that point to the blocks of memory that are not yet allocated (they are available/free). Therefore, the way this works is the following: when a block is allocated, it is removed from the "free list", and when it is deallocated, it is added to the "free list". In this way, the true deallocation takes place only when the memory pool is destroyed.

For the network testing algorithm, the *libcurl* library is used in order to simplify the socket creation as well as the HTTP request generation. In addition, it uses a callback function, the content of which can be defined by the programmer. This function is called as soon as there is data available (in this case, it just computes and returns the size of the data that is available) ([20], [21]).

It is highly recommended to disable all antivirus software programs before launching the application in order to obtain accurate results (this happens because the executables that are ran, are considered to be "suspicious" and thus the results of the benchmark could be affected during the time in which the antivirus software checks the safeness of the executable files). In addition, if the system has a dedicated graphics card from *Nvidia* or *AMD*, in order to obtain accurate results, the Vertical Sync (VSync) option needs to be turned off. Furthermore, the user should not interact with the pop-up windows that appear during the GPU testing phase of the benchmark (resizing or moving the window) as this behavior can affect the score of this test.

When it comes to the benchmark itself, the duration of a complete version of it (testing all five components) is estimated to be somewhere between 10 and 15 minutes (depending on the system). Below, is a list of durations for every component's separate benchmark:

- **CPU**: 5-6 minutes
- **GPU**: 2-3 minutes
- **Memory**: 1-2 minutes
- **Disk**: <1 minute
- **Network**: <1 minute

6) TESTS/EXPERIMENTS

For testing the benchmarking application, three different systems are analyzed. For each one, 30 runs are performed, the first 10 of which are with the default version of the benchmark, the middle 10 are with a light version of the benchmark (ex: integer algorithm arrays are smaller, 3D objects to be rendered are simpler meaning that they have fewer vertices), while the remaining 10 are with a heavy version of the benchmark (which is essentially the opposite of the light version). In addition, the weight of every component is chosen to be 0.2. Moreover, while the benchmarks were run on the systems, they were not used for anything else. Below, the specifications of the three systems are presented:

A. Old Laptop
    1. CPU:
        - Name: Intel Core i5-8250U
        - Architecture: x64
        - Clock Frequency: 1.8 GHz
        - Number of Cores: 4
        - Number of Logical Processors: 8
    2. GPU:

- Name: Intel UHD Graphics 620
- Memory: 1 GB
- Architecture: VGA

3. Memory:
- Manufacturer: Samsung
- Frequency: 2400 MHz
- Capacity: 8 GB

4. Disk:
- Name: Micron_1100_MTFDDAV256TBN
- Capacity: 256 GB
- File System Type: NTFS

5. Network:
- Name: Intel Dual Band Wireless-AC 8265

6. Software:
- OS Name: Microsoft Windows 11 Pro
- OS Architecture: 64-bit
- OS Version: 10.0.22631

B. Gaming Laptop

1. CPU:
- Name: Intel Core i7-12700H
- Architecture: x64
- Clock Frequency: 2.3 GHz
- Number of Cores: 14
- Number of Logical Processors: 20

2. GPU:
- Name: Intel Iris Xe Graphics
- Memory: 1 GB
- Architecture: VGA

3. Memory:
- Manufacturer: Micron Technology
- Frequency: 4800 MHz
- Capacity: 16 GB

4. Disk:
- Name: NVMe SAMSUNG MZVLQ1T0HBLB-00B00
- Capacity: 1 TB
- File System Type: NTFS

5. Network:
- Name: Intel Wi-Fi 6 AX201 160MHz

6. Software:
- OS Name: Microsoft Windows 11 Pro
- OS Architecture: 64-bit
- OS Version: 10.0.22631

C. Gaming PC

1. CPU:
- Name: Intel Core i7-10700KF
- Architecture: x64
- Clock Frequency: 3.8 GHz
- Number of Cores: 8
- Number of Logical Processors: 16

2. GPU:
- Name: NVIDIA GeForce RTX 2080 Ti
- Memory: 4 GB
- Architecture: VGA

3. Memory:
- Manufacturer: G Skill Intl
- Frequency: 2133 MHz
- Capacity: 16 GB

4. Disk:

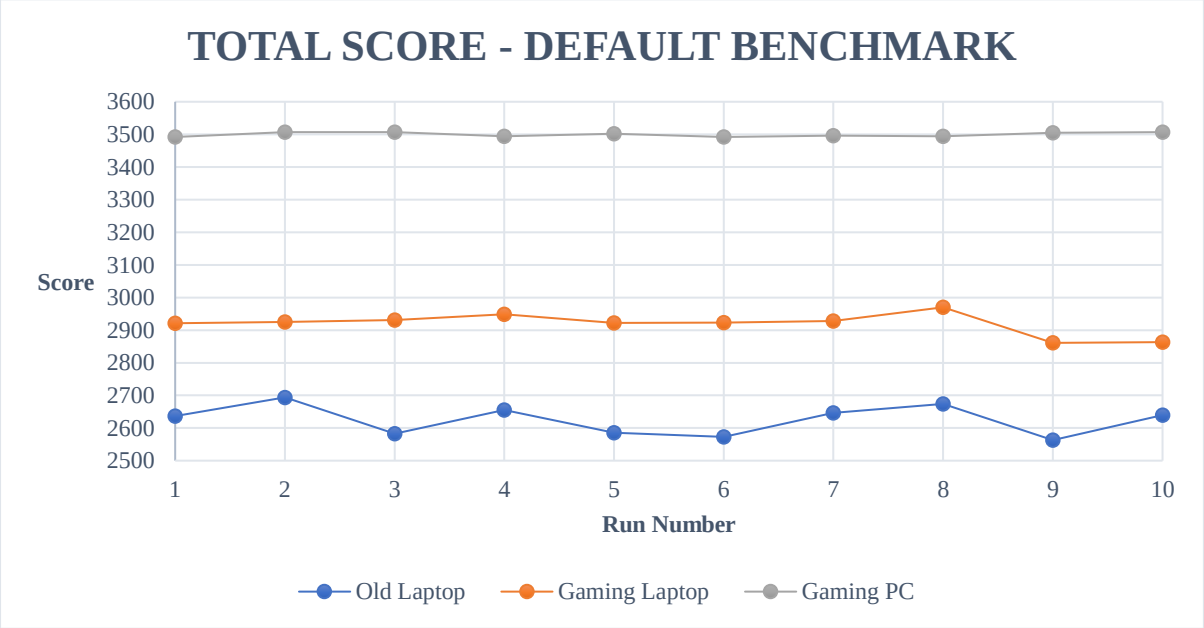
- Name: KINGSTON SHSS37A240G
- Capacity: 256 GB
- File System Type: NTFS

5. Network:
   - Name: Intel Wi-Fi 6E AX210 160MHz
6. Software:
   - OS Name: Microsoft Windows 10 Enterprise
   - OS Architecture: 64-bit
   - OS Version: 10.0.19045

Below, the results that were obtained after performing all benchmarks are presented:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	2637	2921	3492
2	2694	2925	3507
3	2583	2931	3507
4	2655	2949	3494
5	2586	2922	3502
6	2573	2923	3492
7	2646	2928	3496
8	2674	2970	3494
9	2563	2861	3505
10	2640	2863	3507

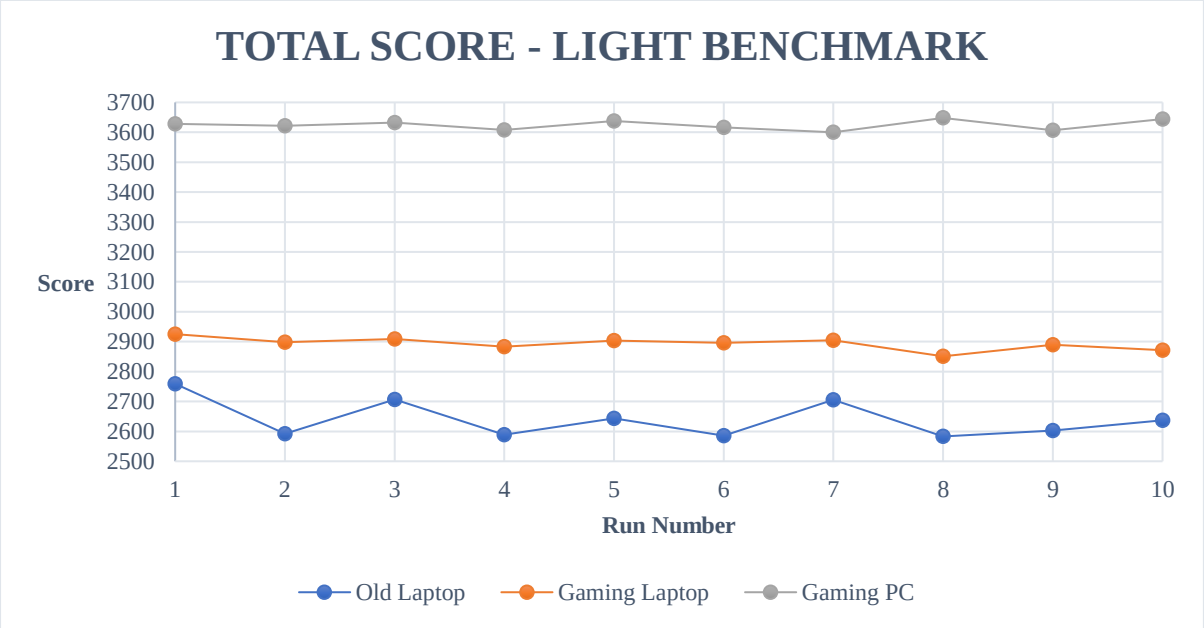
Table 1 - Total Scores/Default Benchmark



Graph 1

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	2759	2925	3628
2	2592	2898	3622
3	2707	2909	3633
4	2589	2883	3608
5	2643	2904	3638
6	2586	2896	3616
7	2705	2905	3600
8	2583	2851	3648
9	2603	2890	3607
10	2637	2871	3644

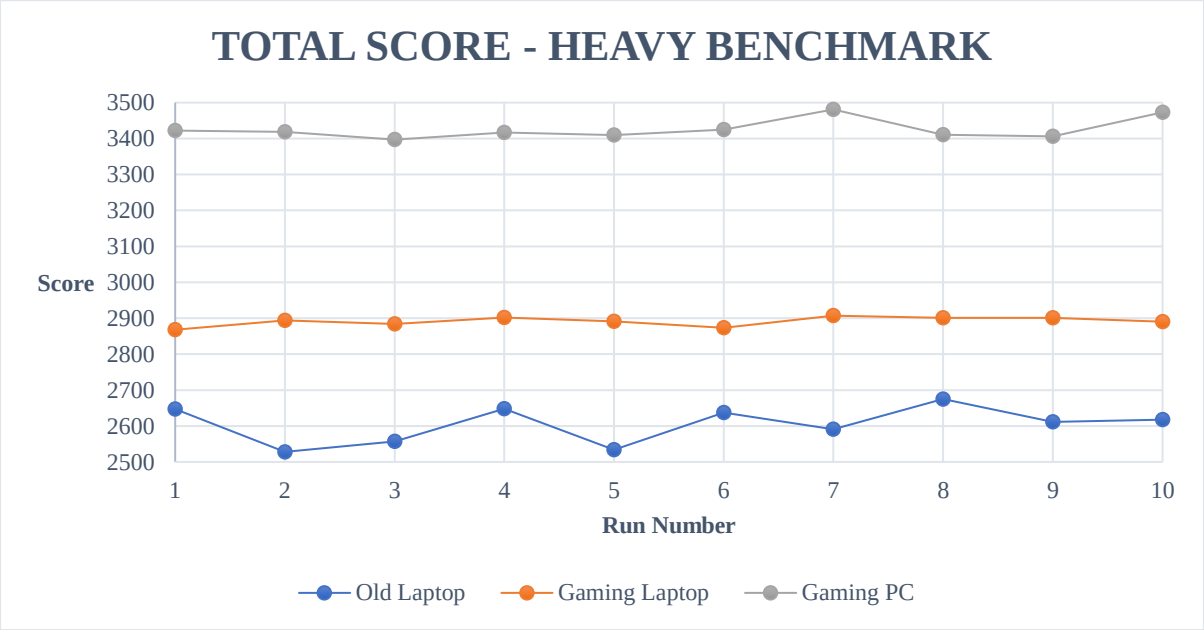
Table 2 - Total Scores/Light Benchmark



Graph 2

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	2647	2868	3422
2	2528	2894	3419
3	2557	2884	3397
4	2648	2902	3417
5	2534	2891	3410
6	2637	2873	3425
7	2591	2907	3481
8	2675	2901	3411
9	2611	2901	3406
10	2618	2890	3473

Table 3 - Total Scores/Heavy Benchmark



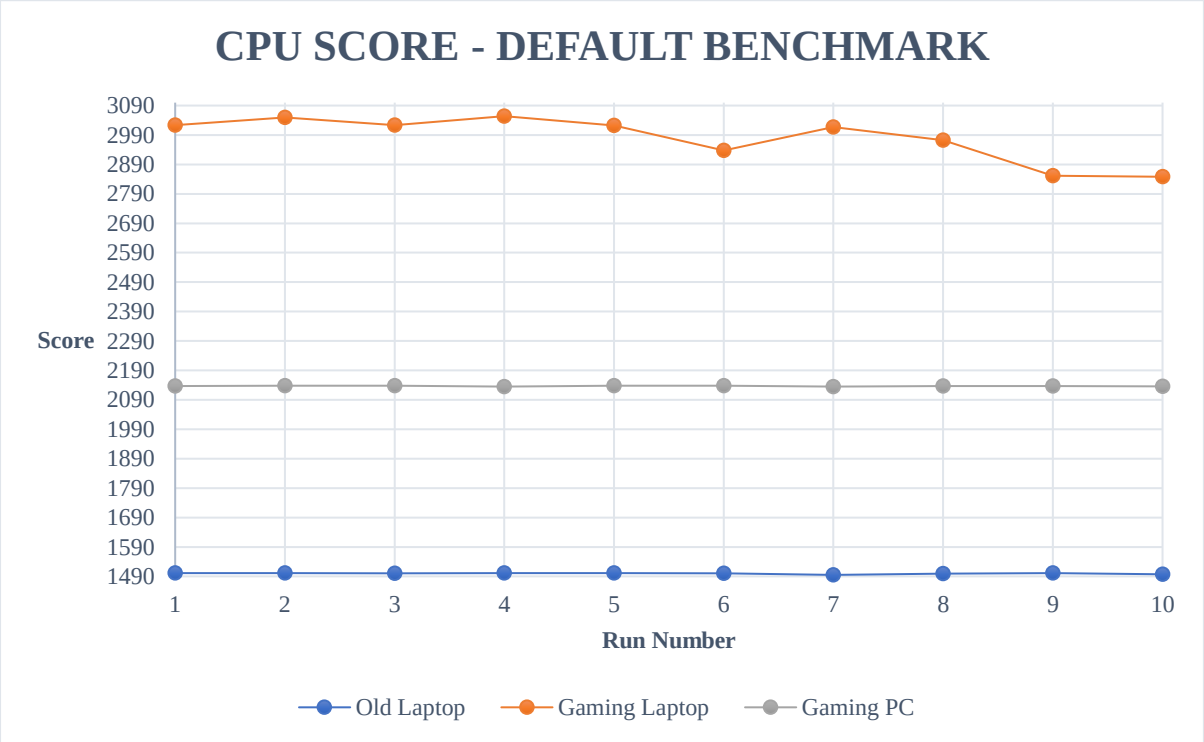
Graph 3

The comments that can be drawn from these three tables and their respective graphs is the fact that no matter the type of benchmark, the Gaming PC is the best performer while the worst performer is the Old Laptop (which was expected to be the case). The gap is mostly due to the major difference in GPUs between the Gaming PC and the other two systems. Another thing that was expected to happen refers to the fact that all of the devices obtained better scores for the light benchmark than the other two. However, the scores obtained for the heavy benchmark and the ones obtained for the default one are very similar. This tells me that the default benchmark is well made and it does not need to be made computationally heavier since it is already pushing the systems that were tested, to their limits.

The scores for the central processing unit are presented below:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1501	3023	2137
2	1501	3050	2138
3	1500	3023	2138
4	1501	3054	2135
5	1501	3022	2138
6	1500	2938	2138
7	1495	3017	2135
8	1499	2972	2137
9	1501	2852	2137
10	1497	2848	2136

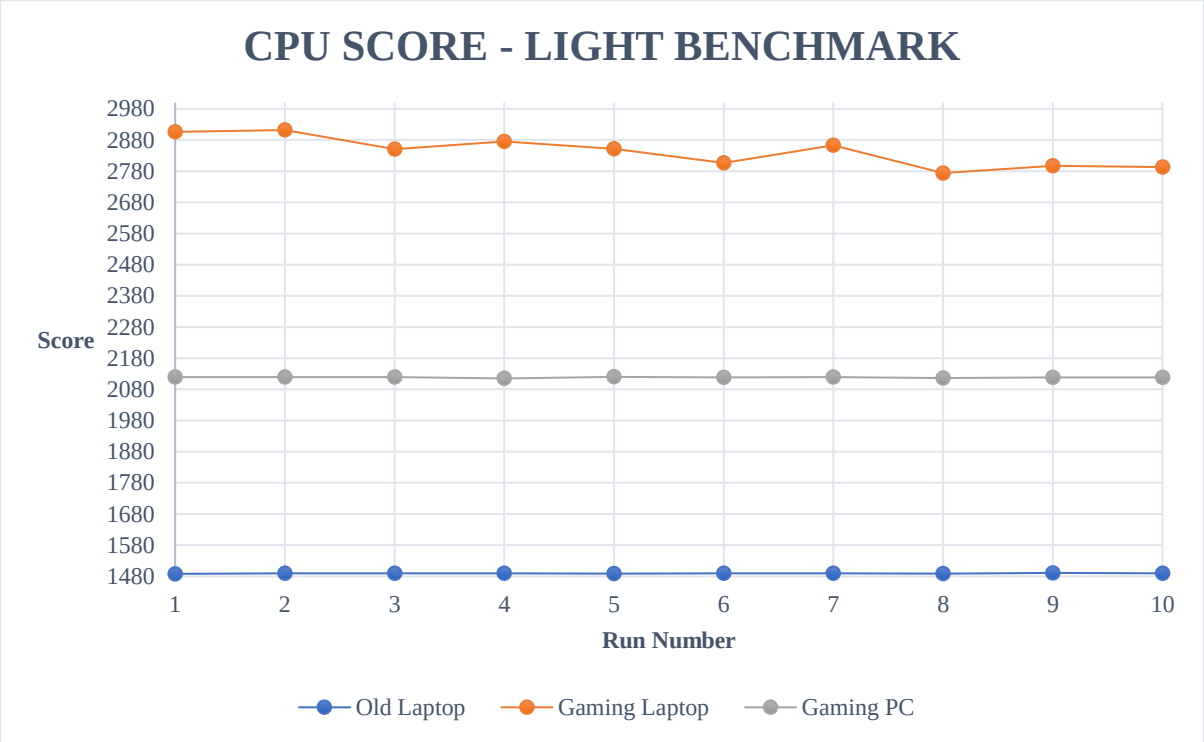
Table 4 - CPU Scores/Default Benchmark



Graph 4

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1488	2906	2119
2	1490	2912	2119
3	1490	2851	2119
4	1490	2875	2115
5	1489	2852	2120
6	1490	2807	2118
7	1490	2863	2119
8	1489	2774	2116
9	1491	2797	2118
10	1490	2793	2118

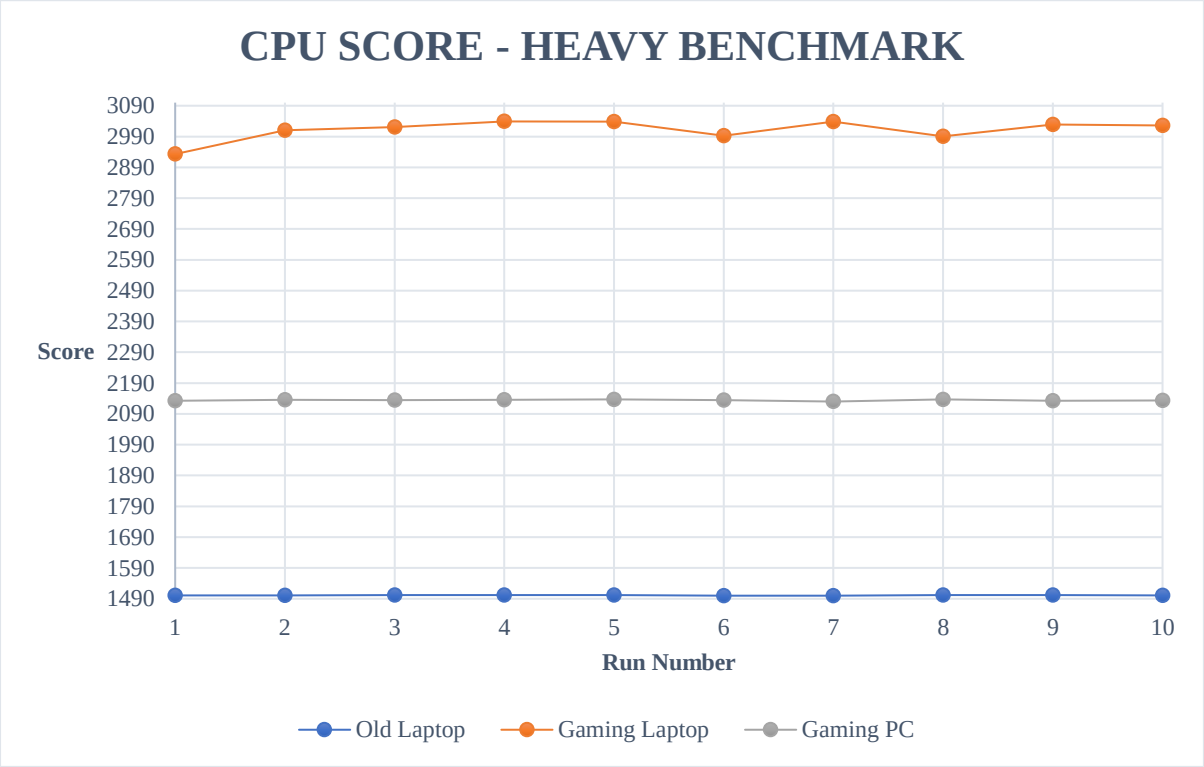
Table 5 - CPU Scores/Light Benchmark



Graph 5

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1501	2933	2132
2	1501	3010	2136
3	1502	3021	2135
4	1502	3039	2136
5	1502	3038	2137
6	1500	2993	2135
7	1500	3038	2130
8	1502	2991	2137
9	1502	3029	2132
10	1501	3026	2134

Table 6 - CPU Scores/Heavy Benchmark



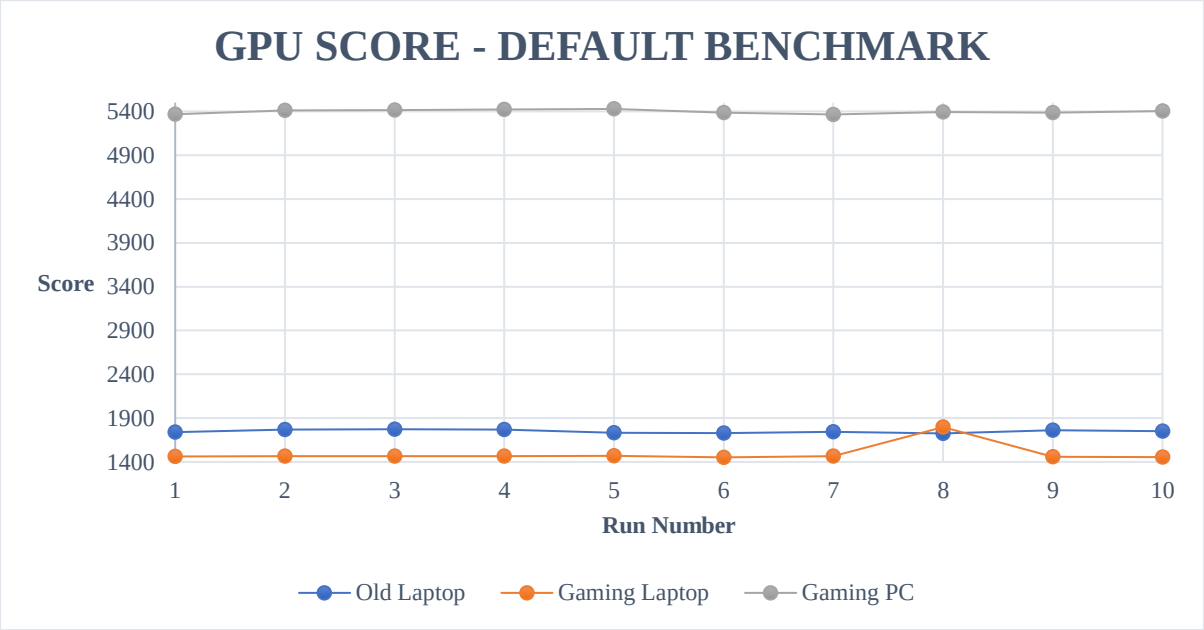
Graph 6

These results are interesting to say the least. Looking at the specifications on paper, the initial assumption was that the best performing CPU corresponds to the best "on paper" CPU, which is not necessarily the case. Well, actually, it depends at what aspect of the CPU we are looking when we compare them and their respective performance. It seems like the frequency of the CPU is not that important when it comes to benchmarking, but actually what matters is the number of cores as well as the number of logical processors, because the results are directly proportional to these two parameters. Another aspect that is worth to be noted refers to the fact that the performances of all three CPUs remain almost the same no matter the complexity of the benchmark.

Next up, the scores for the graphical processing units are presented:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1741	1463	5367
2	1770	1467	5410
3	1773	1464	5413
4	1768	1467	5423
5	1733	1470	5428
6	1729	1452	5386
7	1745	1464	5365
8	1725	1797	5394
9	1761	1457	5384
10	1751	1455	5402

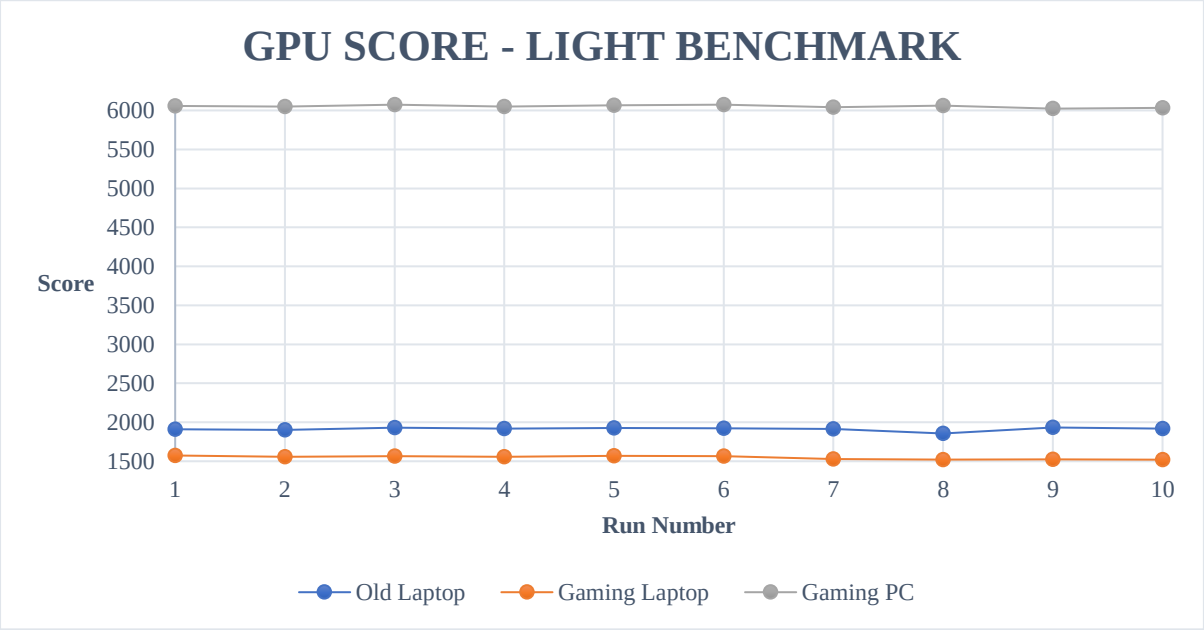
Table 7 - GPU Scores/Default Benchmark



Graph 7

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1912	1574	6058
2	1901	1558	6049
3	1931	1567	6074
4	1917	1559	6050
5	1925	1571	6067
6	1921	1565	6074
7	1914	1527	6043
8	1856	1521	6061
9	1933	1524	6023
10	1920	1522	6031

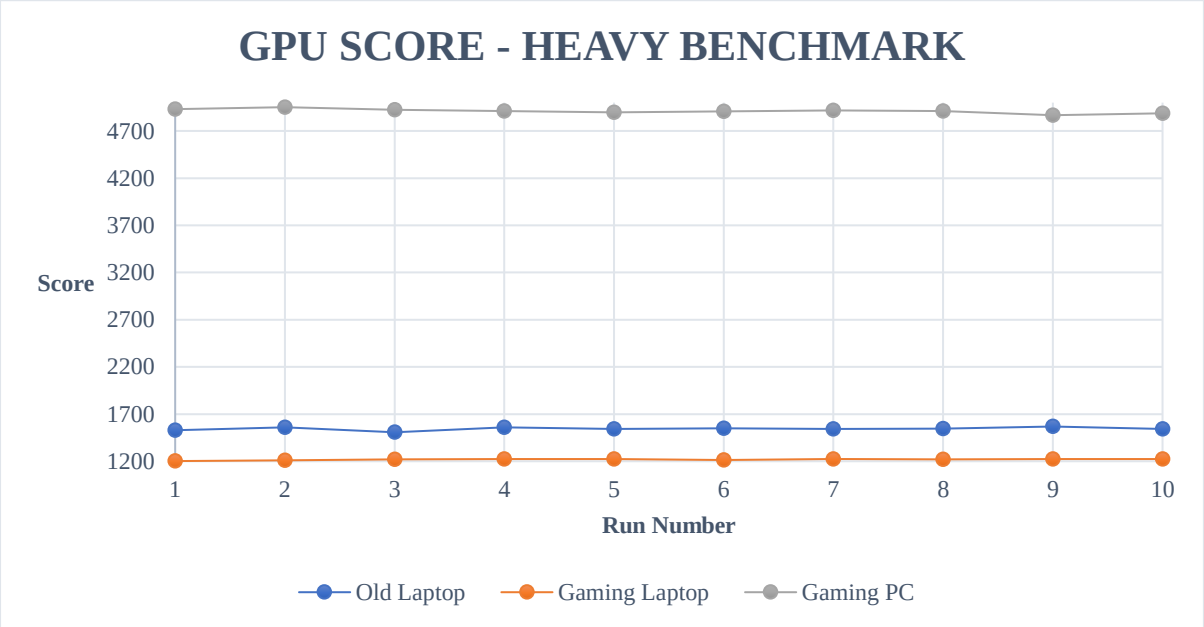
Table 8 - GPU Scores/Light Benchmark



Graph 8

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1528	1202	4932
2	1559	1209	4952
3	1507	1220	4923
4	1558	1223	4909
5	1543	1223	4898
6	1549	1214	4908
7	1541	1224	4917
8	1544	1220	4911
9	1568	1223	4867
10	1542	1223	4888

Table 9 - GPU Scores/Heavy Benchmark



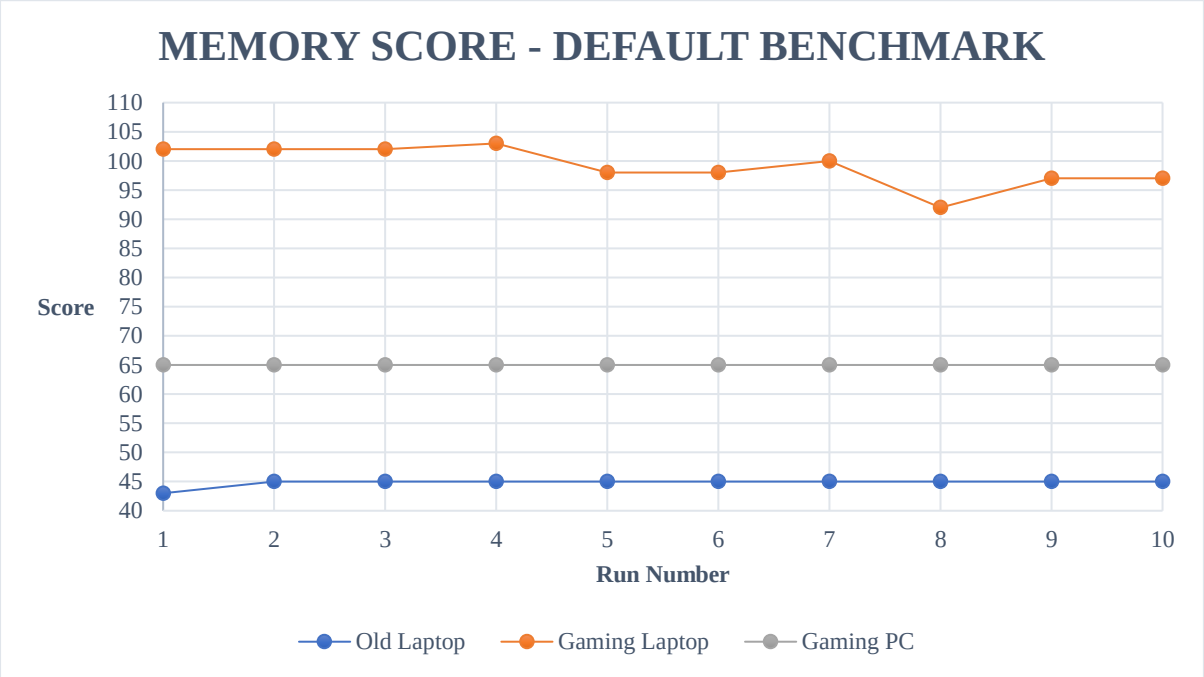
Graph 9

Looking at the results for the GPU, they are representative of what was expected, meaning that the two laptop's GPUs, which are integrated graphics cards performed worse than the PC's dedicated graphics card, having only between a fourth and a third of its performance (which is consistent with the difference of GPU memory between them).

Moving on, the scores of the Memory component are presented next:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	43	102	65
2	45	102	65
3	45	102	65
4	45	103	65
5	45	98	65
6	45	98	65
7	45	100	65
8	45	92	65
9	45	97	65
10	45	97	65

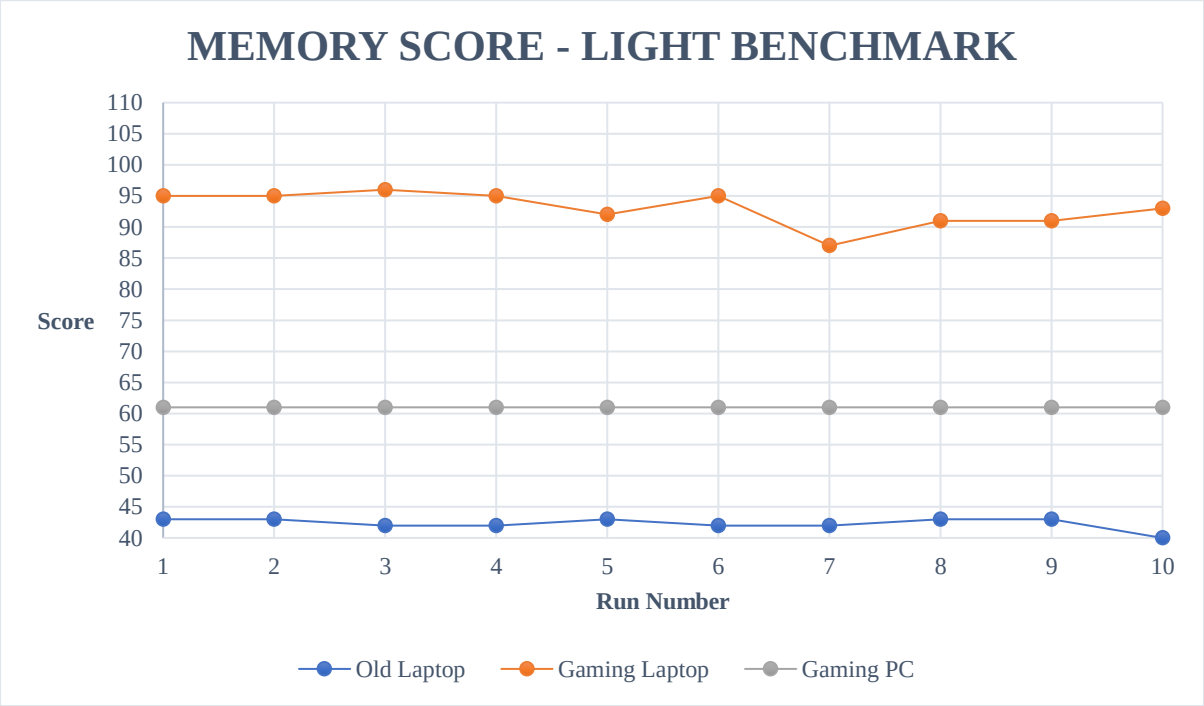
Table 10 - Memory Scores/Default Benchmark



Graph 10

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	43	95	61
2	43	95	61
3	42	96	61
4	42	95	61
5	43	92	61
6	42	95	61
7	42	87	61
8	43	91	61
9	43	91	61
10	40	93	61

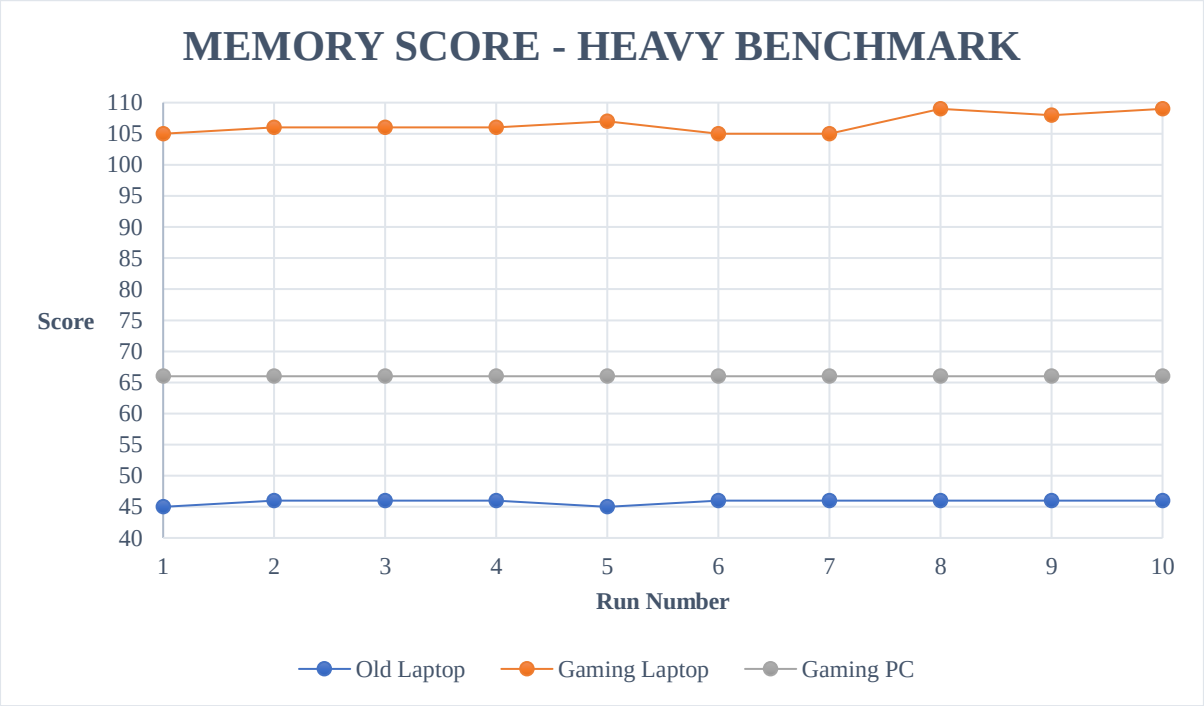
Table 11 - Memory Scores/Light Benchmark



Graph 11

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	45	105	66
2	46	106	66
3	46	106	66
4	46	106	66
5	45	107	66
6	46	105	66
7	46	105	66
8	46	109	66
9	46	108	66
10	46	109	66

Table 12 - Memory Scores/Heavy Benchmark



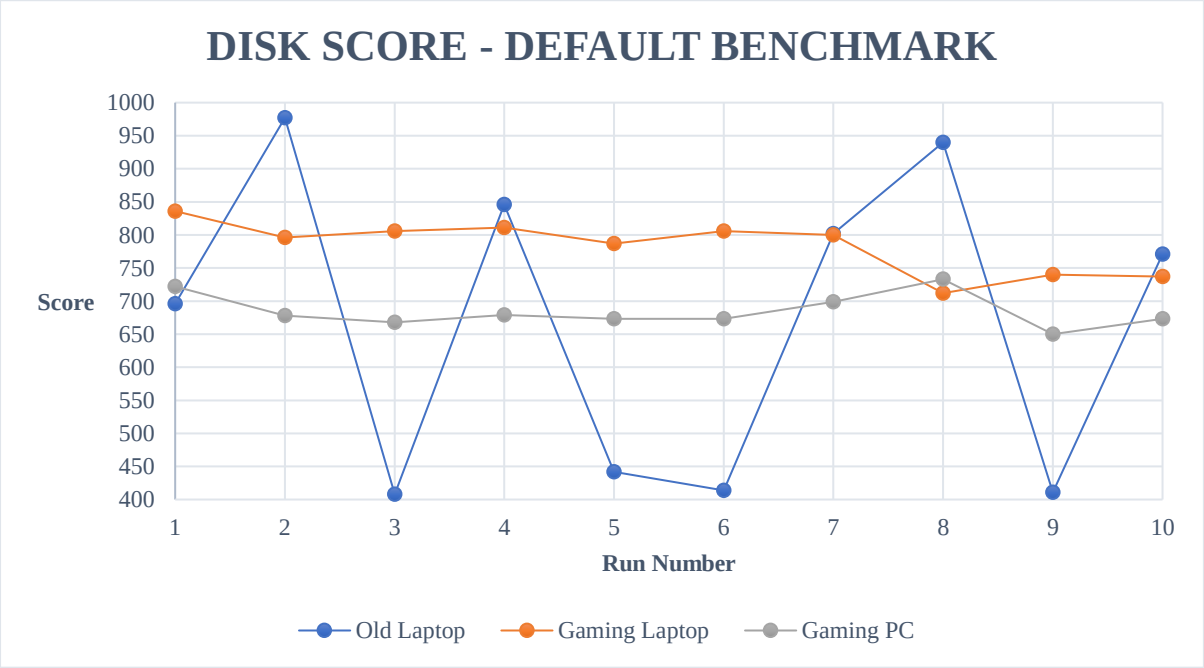
Graph 12

Looking at the results, we can safely say that both the frequency and the capacity are important when it comes to the performance of the memory, but they do not matter equally. Even though the Gaming Laptop and the Gaming PC have the same memory capacity, the frequency is more than doubled on the laptop and this metric is somewhat translated to the benchmark score. Likewise, if we compare the Old Laptop which has half the capacity of the Gaming PC but close frequencies, we can, again, see a 50% difference between the two. However, one thing that is odd refers to the fact that the performance scores that were obtained by the Gaming Laptop's memory component are a lot more unstable compared to the other two systems, which could signal a problem with the driver of the component, a potential factory defect or some other background process was also using the memory at the same time as the benchmark (further testing and analysis is advised in this case). In addition, the scores remain mostly unchanged no matter the complexity of the benchmark.

The penultimate component that was tested is the Disk, and below, the results that were obtained, are presented:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	696	836	722
2	977	796	678
3	408	806	668
4	846	811	679
5	442	787	673
6	414	806	673
7	802	800	699
8	940	712	733
9	411	740	650
10	771	737	673

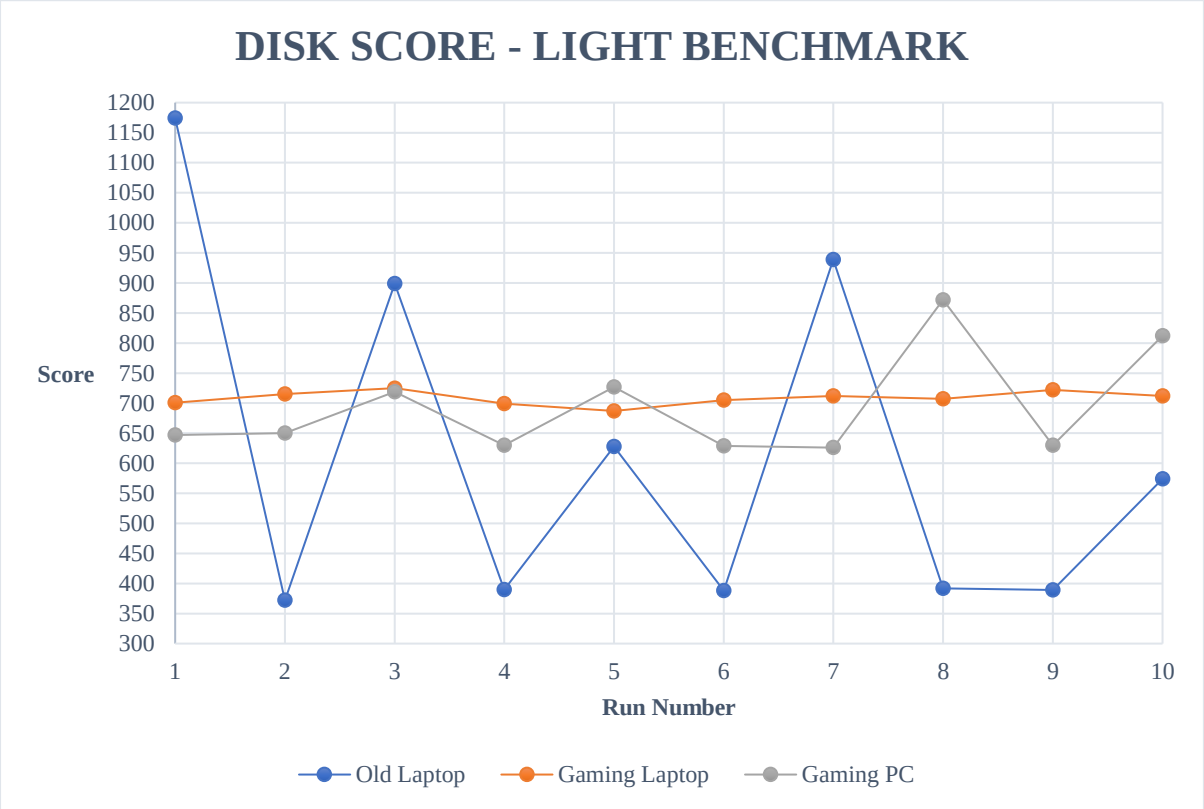
Table 13 - Disk Scores/Default Benchmark



Graph 13

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1174	701	647
2	372	715	650
3	899	725	719
4	390	699	630
5	628	687	727
6	388	705	629
7	939	712	626
8	392	707	872
9	389	722	630
10	574	712	812

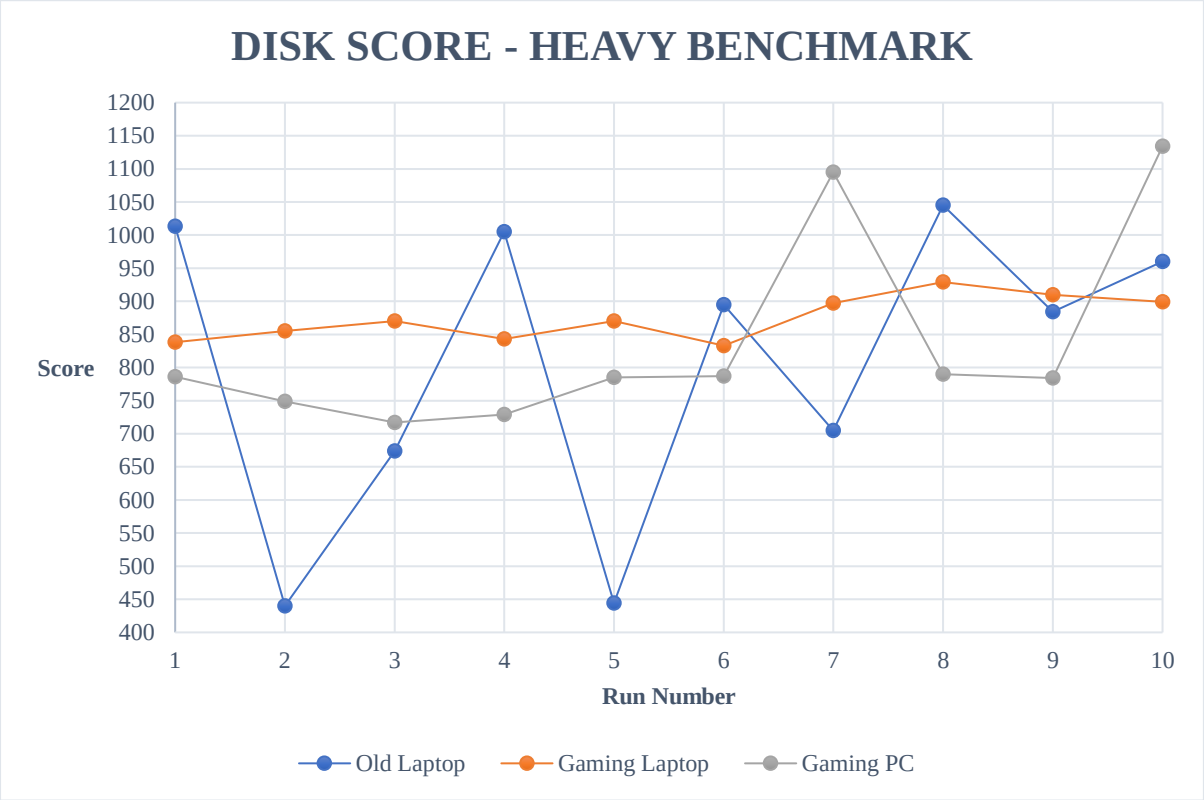
Table 14 - Disk Scores/Light Benchmark



Graph 14

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	1013	838	786
2	440	855	749
3	674	870	717
4	1005	843	729
5	444	870	785
6	895	833	787
7	705	897	1095
8	1045	929	790
9	884	910	784
10	960	899	1134

Table 15 - Disk Scores/Heavy Benchmark



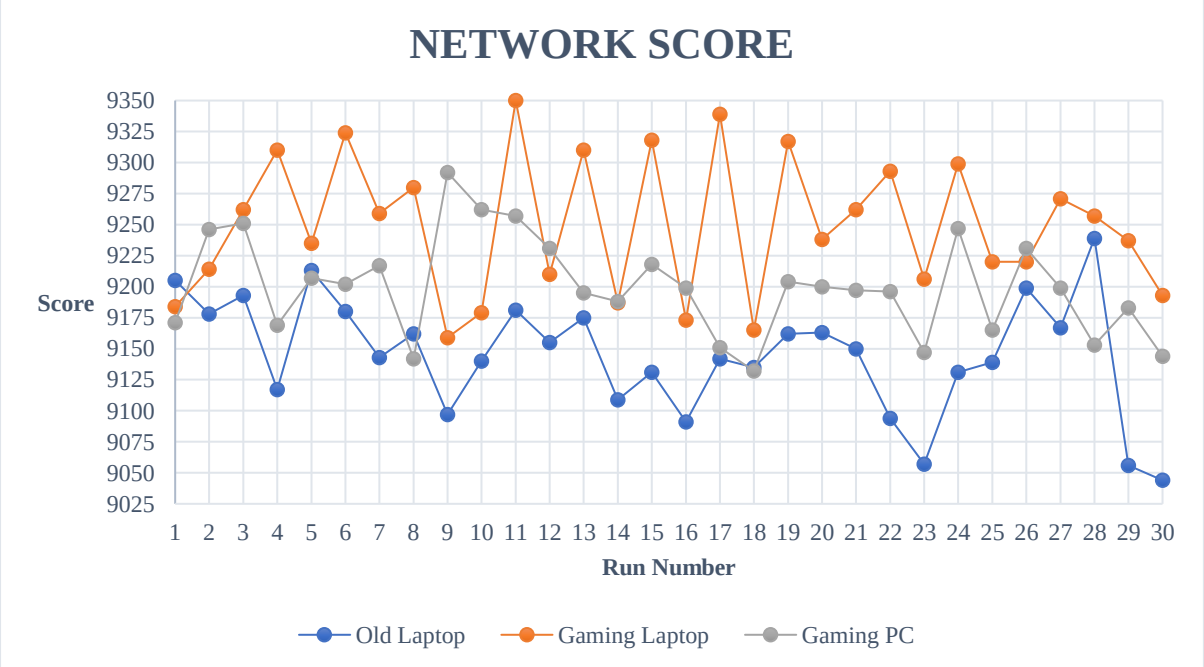
Graph 15

Analyzing these results, one can say that the most unstable disk component out of the three now belongs to the Old Laptop, while the other two remain pretty stable, with minor exceptions coming mostly from the Gaming PC's disk component which seems to obtain more volatile scores during the second half of the tests. One possible explanation could be that on the Old Laptop, the disk is a hard drive, while each of the other two systems sport a solid state drive. What is even more curios is represented by the fact that the results are directly proportional with the complexity of the benchmark, meaning that the components performed better under tighter conditions, which was definitely not expected (but a positive surprise). Another aspect that is worth mentioning is the fact that the capacity of the disk only plays a minor role in the performance of this component, with the Gaming Laptop's 1TB SSD performing ever so slightly better than the Gaming PC's 256 GB SSD.

Lastly, the results of the Network component are shown below. However, one thing must be mentioned beforehand and that is the fact that, for this component, all benchmarks (default, light and heavy) are all the same (there was not anything that could be changed in order to make the test easier or harder). Having said that, only one table and one corresponding graph (containing 30 tests) will be presented:

Run No.	Old Laptop	Gaming Laptop	Gaming PC
1	9205	9184	9171
2	9178	9214	9246
3	9193	9262	9251
4	9117	9310	9169
5	9213	9235	9207
6	9180	9324	9202
7	9143	9259	9217
8	9162	9280	9142
9	9097	9159	9292
10	9140	9179	9262
11	9181	9350	9257
12	9155	9210	9231
13	9175	9310	9195
14	9109	9187	9188
15	9131	9318	9218
16	9091	9173	9199
17	9142	9339	9151
18	9135	9165	9132
19	9162	9317	9204
20	9163	9238	9200
21	9150	9262	9197
22	9094	9293	9196
23	9057	9206	9147
24	9131	9299	9247
25	9139	9220	9165
26	9199	9220	9231
27	9167	9271	9199
28	9239	9257	9153
29	9056	9237	9183
30	9044	9193	9144

Table 16 - Network Scores



Graph 16

To make the competition as fair as possible, all three systems were connected, through Wi-Fi, to the same network. Investigating the results, they are unsurprising since the worst performer here is the Old Laptop, while the Gaming systems are more capable and better equipped for online, multiplayer games, which is the reason why they obtained a higher score. It is important to keep in mind that these scores are not fully representative of the actual networking capabilities of these components since the benchmark itself is not very complex and does not take many aspects into consideration. However, one aspect which can be explained is represented by the instability of the results, which is to be expected from a wireless connection to a network that becomes more or less congested depending on the time of day and that can fluctuate based on many factors that are not in our (the users) control.

7) CONCLUSIONS

In conclusion, the PC benchmarking application presented within this documentation stands as an invaluable tool for assessing and optimizing a system's performance. By offering a comprehensive suite of tests, useful (but mostly general) metrics, and user-friendly interface, it enables users to make informed decisions so as to enhance their computer's capabilities. This application not only simplifies the benchmarking process but also serves as a reliable guide in identifying hardware strengths and weaknesses. Its ability to generate fairly detailed log entries aids in troubleshooting and fine-tuning PCs for optimal performance. Moreover, a PC benchmarking application can be useful not only for testing the performance of a system or of a particular component, but it can also assist the user in finding potential problems/defects with certain components or discover various peculiarities between components of the same type.

Obviously, the application, in its current state, can be developed further with additional features, such as:

- Writing all the benchmarking algorithms in assembly language (lower level code) in order to obtain an even higher accuracy of the performance scores
- Developing the application for other operating systems (ex: Linux, Mac), so that it becomes more portable
- Developing the application for even more complex systems such as GRID architectures, multi-processor architectures (parallel computers), cloud architectures or a combination of all of the above
- Taking into consideration other data types or various structures/objects when testing the CPU, as well as, more complex operations (ex: mathematical operations such as the square root or the power)
- Testing both single core and multi core performances of the CPU (also include algorithms that explicitly work with threads and/or processes)
- Developing a GPU benchmark that is closer to a game by having a scene of multiple 2D and 3D, dynamic and static objects that have various textures applied and that interact with one another, as well as, multiple dynamic and static light sources (both directional and point) that also generate shadows for the objects that they "touch"
- Testing, for Memory, custom pools of various sizes that are implemented in different ways
- Making, for the Disk benchmark, custom algorithms that are better suited for hard disk drives and other ones that are better suited for solid state drives
- Testing, for the Disk component, other, more complex operations such as moving a file (which implies both reading and writing at the same time)
- Testing, for Network, other metrics such as the upload speed
- Creating benchmarks for other components such as Bluetooth, USB, I/O or Database
- Creating a wider variety of benchmarks by introducing other types of benchmarks such as synthetic, parallel, database and kernel
- Implementing a more complex scoring system that uses a combination of ways that compute the score (geometrical mean, arithmetical mean)
- Measuring the temperatures of all the components that are tested
- Give, the user, access (from the application) to a database of performance scores for each of his components, as well as, a leaderboard of the top 10/50/100 systems that scored the highest benchmarking results using this program
- Developing a more useful and professional user interface that:
    - ❖ displays, in more detail, information regarding the hardware and the software of the system

❖ gives the user even more flexibility when it comes to choosing which component to test and how to test it (stress test, regular test, change test parameters)
❖ shows in real-time, the progress of the benchmark by displaying a progress bar, as well as some text labels that inform the user about the current component that is under testing as well as, the algorithm that is currently tested for that particular component
❖ gives more than one score for every component, as well as, the total one
❖ informs the user of a possible bottleneck (and its cause, if it can be determined)
❖ creates a graph of the performance scores of each component and the overall performance when it has multiple benchmark results (at least two); in addition, it could inform the user of the percentage difference between two consecutive scores or between the smallest and the largest scores (the percentage can be both positive or negative)
❖ gives, the user, the possibility of exporting the results in a tabular form for further analysis

8) REFERENCE LIST

- [1] Gheorghe Sebestyen, "*Structure of Computer Systems"* course lecture slides (courses 1-2) [Online] https://moodle.cs.utcluj.ro/mod/folder/view.php?id=46903
- [2] "ChatGPT" [Online] chat.openai.com/chat
- [3] About the EEMBC FPMark™ Floating-Point Benchmark Suite [Online] https://www.eembc.org/fpmark/
- [4] "Benchmarking" [Online] https://en.wikipedia.org/wiki/Benchmarking
- [5] Rishabh Prabhu, "Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()" [Online] https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
- [6] "Benchmarks used to test a C and C++ allocator?" [Online] https://stackoverflow.com/questions/2560114/benchmarks-used-to-test-a-c-and-c-allocator
- [7] "OpenGL Programming" [Online] https://en.wikibooks.org/wiki/OpenGL_Programming
- [8] "Win32_VideoController class" [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-videocontroller
- [9] "Win32_Processor class" [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-processor
- [10] "Win32_PhysicalMemory class" [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-physicalmemory
- [11] "Win32_OperatingSystem class " [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-operatingsystem
- [12] "Win32_OperatingSystem class " [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-operatingsystem
- [13] "Win32_OperatingSystem class " [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-operatingsystem
- [14] "Win32_OperatingSystem class " [Online] https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-operatingsystem
- [15] "std::chrono::high_resolution_clock::now" [Online] https://cplusplus.com/reference/chrono/high_resolution_clock/now/
- [16] "Mean or median? Choose based on the decision, not the distribution", Tyler Buffington [Online] https://towardsdatascience.com/mean-or-median-choose-based-on-the-decision-not-the-distribution-f951215c1376
- [17] "Windows Management Instrumentation" [Online] https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page
- [18] "What is sequential read speed?" [Online]

https://www.userbenchmark.com/Faq/What-is-sequential-read-speed/44
- [19] “What is 4K random write speed?” [Online] https://www.userbenchmark.com/Faq/What-is-4K-random-write-speed/29
- [20] “libcurl - the multiprotocol file transfer library” [Online] https://curl.se/libcurl/
- [21] “Using libcurl” [Online] https://everything.curl.dev/usingcurl
- [22] “Class Runtime” [Online] https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html
- [23] “GLUT - The OpenGL Utility Toolkit” [Online] https://www.opengl.org/resources/libraries/glut/
- [24] “The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3”, Mark J. Kilgard [Online] https://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- [25] “Doolittle Algorithm : LU Decomposition” [Online] https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/
- [26] “Maclaurin Series of Arctanx” [Online] https://www.emathzone.com/tutorials/calculus/maclaurin-series-of-arctanx.html
- [27] “Deriving the famous Euler’s formula through Taylor Series” [Online] https://muthu.co/deriving-the-famous-eulers-formula-through-taylor-series/
- [28] “Leibniz formula for π” [Online] https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80
- [29] “std::uniform_int_distribution” [Online] https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution
- [30] “std::mt19937” [Online] https://cplusplus.com/reference/random/mt19937/