Benchmark program for a PC

Cadar Andrei Calin

Technical University of Cluj-Napoca

Group: 30432

Table of Contents

Benchmark program for a PC

1. Introduction

1.1 Context

The goal of this project is to design and implement a benchmark program for a PC. The program will be able to benchmark the most important resources of a PC and test their most sought-after parameter when talking about comparison of PCs, their speed.

Another important parameter would be power consumption over different loads, but just a few users are interested in power consumption since the PCs don't usually consume a lot, however I am sure that for network servers this parameter is as important as their performance. To test this, it would also mean to tear down my laptop and I do not want to do it.

1.2 Application type

The main application will be compiled as x86_64 executable for Microsoft Windows. This application will execute directly other executables as child processes or run a python script which will run those processes multiple times when **intensive mode** is used.

2. Bibliographic study

First of all, I needed to find how to make the CPU perform a certain number of operations. It may seem easy at first, but the compiler does a lot of optimizations to make the program run faster. So even if you have a 1 million loop, the final .exe may perform only a thousand iterations.

After reading the manual[1] for ***gcc*** and after following a discussion[2] on stack overflow about this topic I have found out the right options for gcc to compile with the least number of optimizations. Why least and not 0? Because even the lowest level has 52 optimizations according to this[12] Stack-Overflow's question's first answer.

Luckily, I have experience with reverse engineering and static analysis and I can check if my machine code was what I expected it to be.

After that I needed to settle on how I was going to make my GUI. After I have read about Qt, I thought it is too complex for my project so I have chosen .NET Windows Forms App. Windows has one of the largest and well-done documentations out there so this is also a plus.

3. Analysis

3.1 CPU frequency test

To measure CPU frequency accurately in the context of our benchmark project, I need to develop a custom benchmarking tool using C. This tool will specifically be designed to determine CPU frequency by analyzing the elapsed time for a set of CPU-intensive tasks. To have consistency, a for loop with a simple arithmetic operation inside should be ok. We will count the number of operations and the elapsed time in milliseconds. To get the frequency we divide the number of operations to the elapsed time.

3.2 Memory Read/Write speed

To measure memory's write speed I will write a lot of values to an already allocated array. Subtracting the start time from the final time will give as the duration during which we have written data. By dividing it to the memory size that we have written int the duration we will get a MB/s or KB/s rate, depending on the transfer speed.

To measure the read speed, we will perform assign operations in a loop between a variable with the size of a byte and the elements from an array. To get the memory read speed we will perform the same strategy as above.

3.3 Storage Read/Write speed

To measure storage read/write speed we will write a sequence of bytes to a file and read the contents of a file. To calculate the rate of bytes per period of time we perform the same operations as above, divide the memory size to the duration from start to finish.

4. Design

4.1 GUI Design

For the GUI I have chosen a simple grid design, with a purple oriented theme.

There will be three rectangles, in each of them the start button for the test and information about that resource will be present. The program's UI will accommodate 1 CPU, 1 or 2 banks of RAM and 1 or 2 storage units.

At the bottom last benchmark information can be visible also arranged in a grid rectangle design and a button to perform all benchmarks one after the other, to benchmark all resources at a click.

Figure 4.1 design of the app

4.2 GUI Interaction

After starting the main app executable, the user can press 4 buttons:

a) Start CPU Test
b) Start RAM Test
c) Start Storage Test
d) Start All Tests

During the execution of a test, or all test in case of the last button, the user cannot interact with the app until the test in progress is done (cannot start another test).

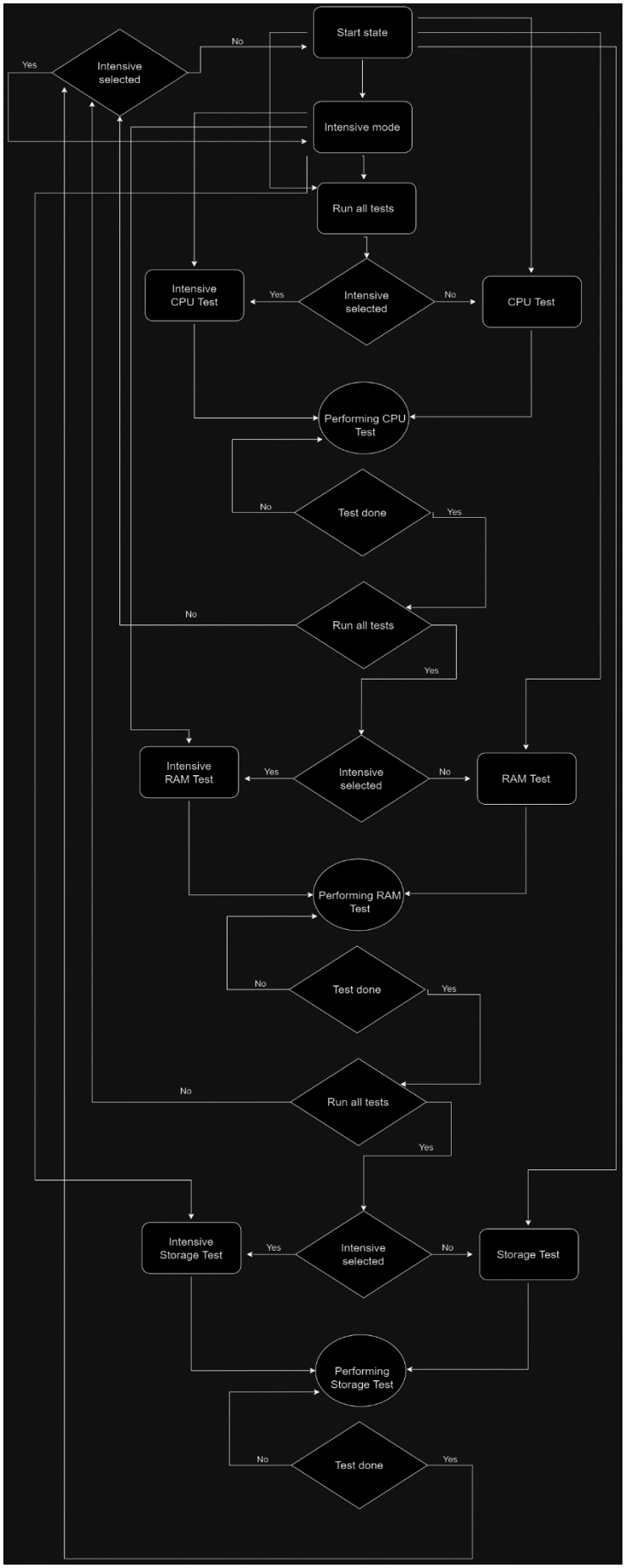

Figure 4.2 app state diagram

5. Implementation

5.1 CPU benchmark

First of all, we need to be able to count the number of operations that were done during the execution. For that we would use RDTSC[6] instruction in ASM x86 language.

Since the result is stored in two different registers as explained here[7], we have to combine them in a variable by performing an **or** logic operations between the lower part and the higher part logical shifted left by 32 bits.

```
static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

Figure 5.1 inline x86 assembly get current clock cycles

Now that we are able to calculate the clock cycles, we need to get the time. In order to get the elapsed time, we will use clock()[8] from standard library to give us a current system time, then we will subtract start time from the end time.

A problem that may arise at compile time might be that the compiler optimizes the code so much that it does not do the operations resulting in an oscillating result since the time interval is too small. To solve this, an if statement can be introduced inside the loop and also print out the variable that it is being incremented inside the loop. This way we will force the compiler to calculate it since it needs to be printed and also because it cannot deduce a formula between the number of steps and the final value.

```
for(unsigned long long i = 0; i <7500000000; i++)
{
    if(i%2)
        x++;
    else
        x+=2;
}
```

Figure 5.2 perform basic operation

```
printf("%llu", x);
```

Figure 5.3 print variable to force compiler to calculate it

We can verify that indeed it does all the expected operations by disassembling and decompiling the executable with Ghidra[9] and looking at the instructions. Ghidra is a free and open-source reverse engineering tool developed by the National Security Agency (NSA) of the United States.

Figure 5.4 Ghidra view to see if every expected instruction is done

5.2 RAM benchmark

For RAM I decided to allocate 512MB to get consistent results. Here it was trickier because I wanted to measure exactly and only the read and write time durations. But in a for loop there are operations running in the background like the increment of the iterator and the **CMP** instruction. In order to exclude these from our calculations an empty loop was needed to measure the time is takes for all the instructions to complete except the memory read/write ones. Then the actual read/write duration will be the *full loop time – empty loop time.*

In order to have only one executable the program will receive an argument **'1'** or **'2'** to know if it will perform the write operation or the reading operation respectively.

```
int main(int argc, char *argv[])
{
    double theSpeed = 0;

    if(argv[1][0] == 49)
    {
        theSpeed = getWriteSpeed();
    }
    else
    {
        theSpeed = getReadSpeed();
    }
```

Figure 5.5 check argument passed to the executable

At the reading operation I encountered another problem. To actually read the data from the memory I need a C instruction like the following:

```
char value = data[i];
```

Figure 5.6 actual read operation

But if **value** is not used, the compiler is smart enough not to the operation at all. My solution was to use the **value** variable in calculating a sum so the empty loop for the read operation will also contain this addition. Also printing the sum was required in order to make the compiler add the instructions we want to measure in the executable.

```
double getReadSpeed()
{
    char* data = (char*)malloc(DATA_SIZE);
    if (data == NULL)
    {
        perror("Memory allocation failed");
        return 1;
    }

    clock_t start_time, end_time;

    for (size_t i = 0; i < DATA_SIZE; i++)
    {
        data[i] = 3;
    }
    unsigned long long x = 0;

    start_time = clock();
    for (size_t i = 0; i < DATA_SIZE; i++)
    {
        x+=2;
    }

    end_time = clock();

    clock_t duration1 = end_time - start_time;

    start_time = clock();
    for (size_t i = 0; i < DATA_SIZE; i++)
    {
        char value = data[i];
        x+=value;
    }
    end_time = clock();
    printf("%ull", x);
    duration1 = (end_time-start_time) - (duration1);

    double readSpeed = (double)DATA_SIZE / (duration1) * CLOCKS_PER_SEC;

    return readSpeed;
}
```

Figure 5.7 code which calculated RAM reading speed as exactly as possible

We can check with Ghidra[9] that indeed this is what the compiled executable does and we see that it is ok.

```
pvVar5 = malloc(0x20000000);
if (pvVar5 == (void *)0x0) {
  perror("Memory allocation failed");
  dVar6 = 1.0;
}
else {
  for (i = 0; i < 0x20000000; i = i + 1) {
    *(undefined *)(i + (longlong)pvVar5) = 3;
  }
  x = 0;
  cVar1 = clock();
  for (i_1 = 0; i_1 < 0x20000000; i_1 = i_1 + 1) {
    x = x + 2;
  }
  cVar2 = clock();
  cVar3 = clock();
  for (i_2 = 0; i_2 < 0x20000000; i_2 = i_2 + 1) {
    x = x + (longlong)*(char *)(i_2 + (longlong)pvVar5);
  }
  cVar4 = clock();
  printf("%ull",x);
  dVar6 = (536870912.0 / (double)((cVar4 - cVar3) - (cVar2 - cVar1))) * 1000.0;
}
return dVar6;
```

Figure 5.8 Ghidra screenshot to check if everything is done as expected

When displaying RAM details, we have to take into consideration the number of RAM slots used on the computer. For example, my PC has 2 slots used, both of them completely identical. This is the best way to achieve maximum performance and avoid compatibility issues as described here[13]. One person said that: *"Generally speaking, best practice is to use the same brand (Samsung, etc.) / type (DDR3, etc.) / speed (PC3 10700, etc.) / timings (CL7 7–7–7–19, etc.), for best results."*, but that does not mean they cannot be different so we have to treat each RAM bank individually.

```
var ram = new ManagementObjectSearcher("select * from Win32_PhysicalMemory")
.Get()
.Cast<ManagementObject>()
.ToArray();

if (ram.Length == 1)
{
    RAMDetailsText.Text = "";
    ulong capacity = (ulong)ram[0]["Capacity"] / 1024 / 1024 / 1024;
    string type = ((ushort)ram[0]["MemoryType"] == 0) ? "DDR4" : "DDR3";
    RAMDetailsText.Text += "Slot 1" + "\n  Type: " + type + "\n  Capacity " + capacity.ToString() + "GB\n  Speed " + ram[0]["Speed"] + "Mhz\n";
    RAMDetailsText.Visible = true;
}
else if (ram.Length == 2)
{
    RAM1DetailsText.Text = "";
    RAM2DetailsText.Text = "";
    ulong capacity1 = (ulong)ram[0]["Capacity"] / 1024 / 1024 / 1024;
    ulong capacity2 = (ulong)ram[1]["Capacity"] / 1024 / 1024 / 1024;
    string type1 = ((ushort)ram[0]["MemoryType"] == 0) ? "DDR4" : "DDR3";
    string type2 = ((ushort)ram[0]["MemoryType"] == 0) ? "DDR4" : "DDR3";
    RAM1DetailsText.Text += "Slot 1" + "\n  Type: " + type1 + "\n  Capacity " + capacity1.ToString() + "GB\n  Speed " + ram[0]["Speed"] + "Mhz\n";
    RAM2DetailsText.Text += "Slot 2" + "\n  Type: " + type2 + "\n  Capacity " + capacity2.ToString() + "GB\n  Speed " + ram[1]["Speed"] + "Mhz\n";
    RAM1DetailsText.Visible = true;
    RAM2DetailsText.Visible = true;
}
```

During the testing of the RAM speeds, I had a surprise, my values were about **3GB/s** for writing and about **4GB/s** for reading, I did not know what the values were supposed to be so I googled what those values should be. What a surprise was when I found out on every site that the speeds should be around **20 – 25 GB/s** for a DDR4 RAM with a frequency of **3200 MHz** like mine.

Fortunately, I realized quickly what the problem was. Firstly, I performed operations on individual bytes, I allocated an array of **char**, when in practice pages of at least **4KB** are used, even 4MB. Then I tried with an array of **short**, **int** and **long long**. Of course, the results were different because more data was allocated in parallel in a single iteration of the loop. Here is a table with the size in bytes of each mentioned data type in C and the speed to write 256 MB in blocks of that size.

Data type	**Size in bytes**	**Speed GB/s (25 6 MB data)**
char	1	3.09
short	2	4.24
int	4	5.70
long long	8	6.50

$$\frac{x}{(1+x)}$$

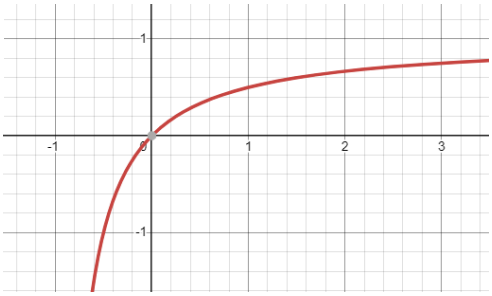

Figure 5.10 x/(1+x) graph

Plotting a graph with the x axis being the size in bytes of the data type and y axis being the speed confirms my assumption of a logarithmic dependency. Why logarithmic? We can see it as a **x/(a+x)** function where **a** is a constant. **x** is the time to actual transfer the data and **a** is the time for the operating system to manage the page, which is the same regardless the size.

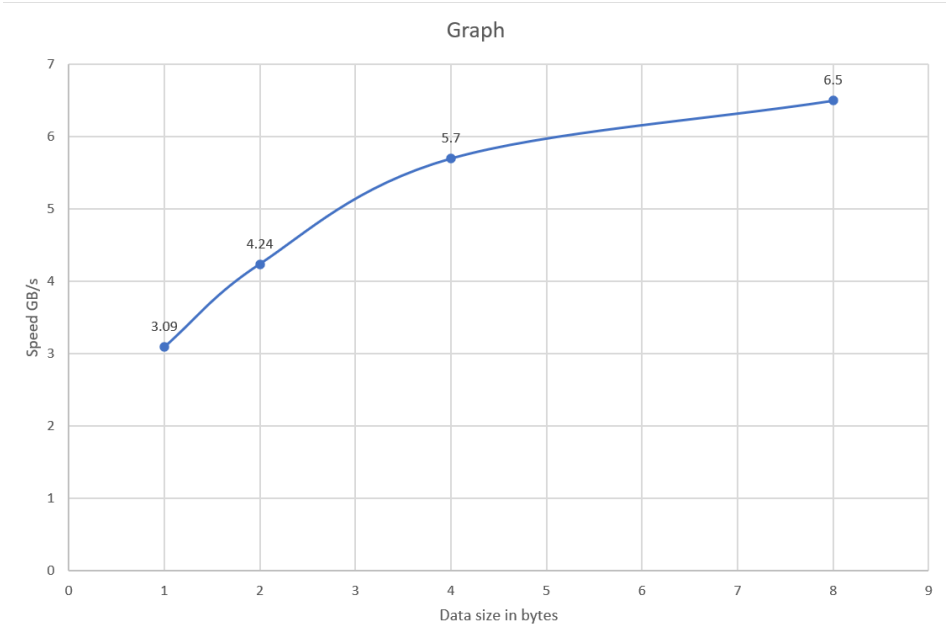

Figure 5.11 RAM speed plot by page size

Now how do we get to that **25GB/s**? The page size in **Windows** are a bit complicated so we will take the page size in **Linux** which is **4KB** or **4096 bytes**. We will assume this is the best size in terms of speed to work with RAM. Now we need to find the **y** such that ***f(4096) = y***.

To approximate the curve of the function and to extrapolate to higher values we can use the method **curve_fit()** from scipy library[14] in Python. This function returns us 2 parameters **a** and **b** from y = a * log(x) + b. Which in this case are:

- a = 1.686
- b = 3.129

Then it is very easy to calculate our y for **x = 4096** and it is:

- y = 17.157

If our data type would have had **4KB** we would get a write speed of approximately **17.15** GB/s, which is still a bit far from **20 GB/s** but closer than **3 GB/s**. But pages can vary in size on Linux from **4KB** to even **4MB** and for a size of **4MB**, by using the same method we get a speed of **28.64 GB/s** which is in line with our theory.

5.3 Storage benchmark

For the storage benchmark mostly is the same as for the RAM benchmark, we have to measure write and read speeds. The size of the file to be read and written will be **50MB**, will get us a good average of MB/s without using too much storage. If multiple units of storage are present, the one on which the program runs will be used.

Here we do not have to subtract the time needed for the empty loops because we need to pass the buffer to fwrite()[10] and fread()[11] and they will take care of performing the operations as fast as possible by writing entire pages of size set by the operating system.

Firstly, the write test will be performed, because we can keep that file and read it in the read test. But after the read test is completed, the file used will be removed from the file system, because nobody wants a 50MB file just staying on their computers. For my tests I settled with a “page” of 8 bytes and I will let the extrapolation to a page of 4KB as a future improvement.

Intensive mode

The user has a toggle button in the top right corner of the app which switches between the standard mode and an intensive mode.

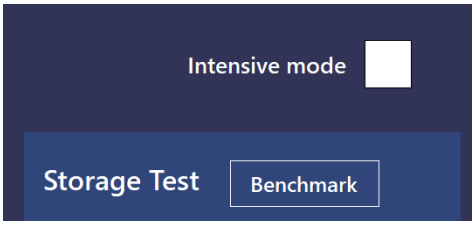

Figure 5.12 Intensive mode toggle switch unselected

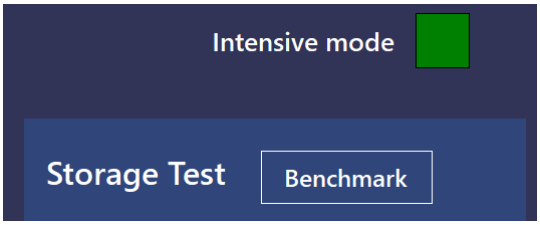

Figure 5.13 Intensive mode toggle switch unselected

When the **intensive mode** is selected each benchmark test will perform ten times, this number of tests allows us to get a good approximation of the real performance without waiting too much.

The displayed result will be an average of the values from the multiple tests. In intensive mode a graph for each test will be available for viewing to see if some test results were a lot less or greater than the average. This will tell the user that this test was not conclusive and should repeat it. The button is between the current test results and last results panels of each test.

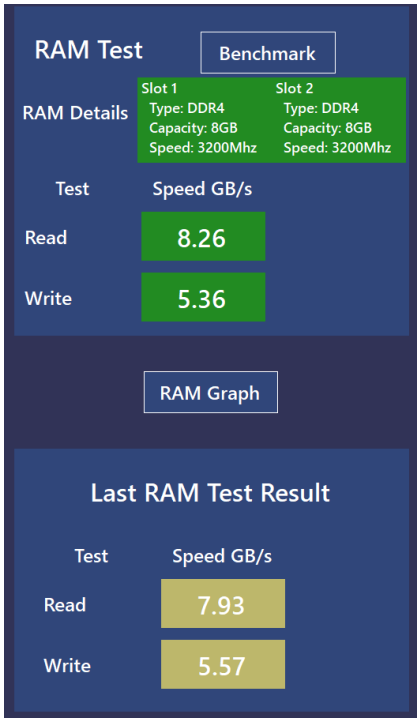

Figure 5.14 Button to show the graph

After pressing the button, a pop-up window opens with the graph.

Because **CPU** Frequency measurement is very exact, we see that not much changes between the tests. Also, we execute very simple instructions, addition and subtraction which are all done in an ALU, there is no data going to or coming from the Floating Point Unit (FPU) which could induce some random very small delays. After running more tests in a short period of time we can see a slight drop in the frequency of about 10 – 1kHz, this is due to Thermal Throttling, also called Dynamic frequency scaling[15]. The CPU heats up and protects itself by lowering the temperature. The maximum internal temperature supported by my CPU is 100°C, according to the official Intel documentation[16].

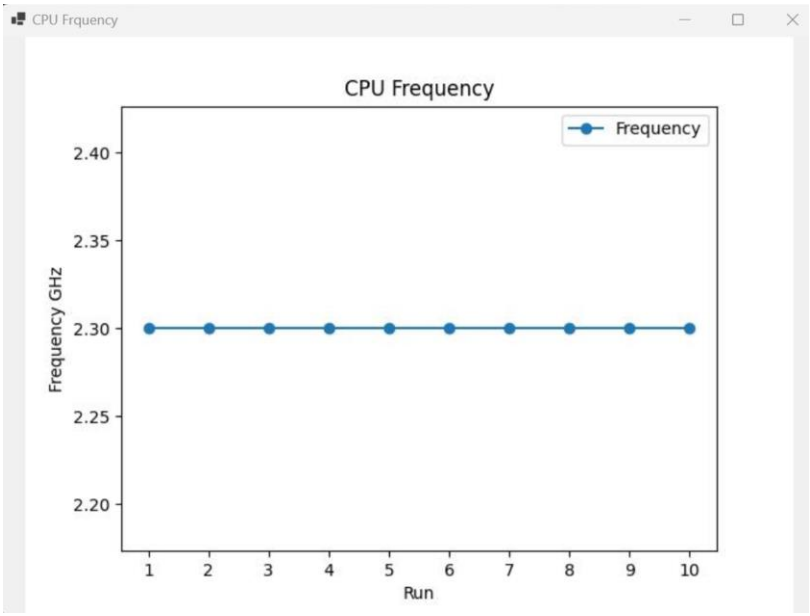

Figure 5.15 CPU Graph

On the other hand, on the **RAM** write and read tests, or on the **Storage write** and **read** tests, small variations between the tests are present, which is normal.

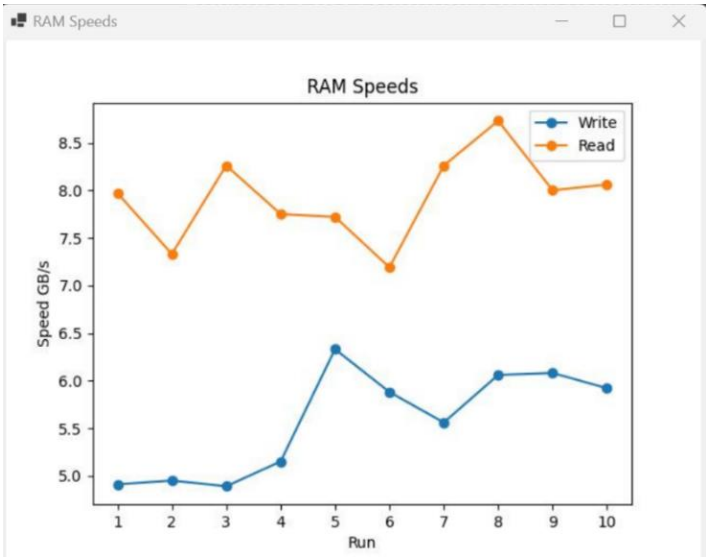

Figure 5.16 RAM Graph

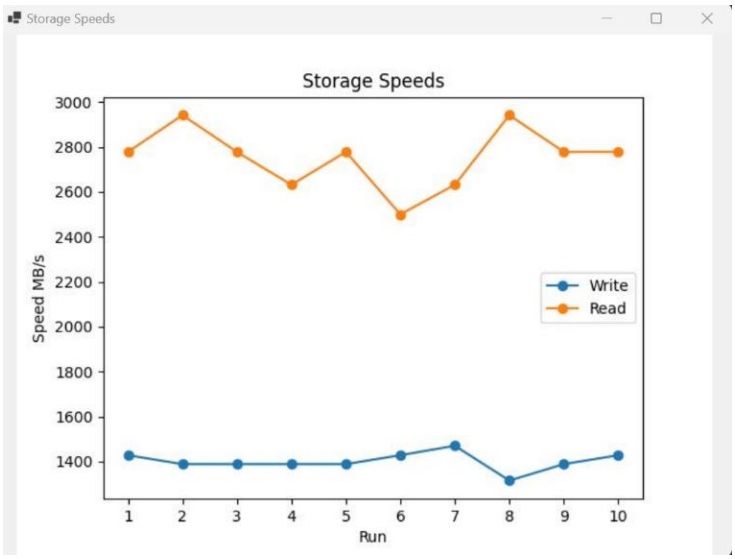

Figure 5.17 Storage Graph

In all cases on the X-Axis are represented the iterations of the test and on the Y axis the corresponding speeds. In all cases we have a legend, which is more useful in the **RAM** and **Storage** tests where we have two lines corresponding to **read** and **write**, each one with a different color on the graph.

Behind the scenes this graph plotter mechanism was implemented using a script in python which is executed every time a test is run using intensive mode.

The script takes as argument a string which can be “cpu”, “ram” or “storage” and executes the corresponding test script 10 times (20 times in case of ram and storage, 10 times for read and 10 for write) and plots a plot using Matplotlib[17]. The plot’s legend, name and title are set according to the parameter received.

The y axis is auto sized thanks to Matplotlib but the x axis needed to be set manually. By default, Matplotlib introduced on the X axis non integer numbers like 1.5 or 2.5 but of course we do not want them since we do not have the run 1.5. To solve this we had to use the method xticks()[18].

```
plt.xlabel('Run')
plt.xticks(range(1, num_runs + 1))  # Set x-axis ticks to integers
plt.legend()  # Add legend
```

Figure 5.18 Storage Graph

6. Conclusion

I have chosen this project because since I was a kid, I was very interested in computer components, how do they connect to each other and their specifications. Since then, every 2-3 months I simulate a PC that I would buy at that time to keep me up to date about new components, their specifications and to see what is worth to buy. To compare them and to know which one to choose, of course I have to search benchmarks for them. These benchmarks almost every time are proprietary software and do not expose the code behind the actual tests and usually, they output a score, not a quantity and a unit of measure like MB/s (mega-bytes / second). More than often these scores are biased to the ones who pay the most, in this case tech-giants, Nvidia and Intel. As websites[19] or discussions on multiple forums reveal that there is an *“Anti-AMD stance”* by multiple benchmark websites and applications.

Why not then build my own benchmark tool? It was a really fun experience and learned a lot of new things while doing this project:

- How heavily optimized **gcc** can be (119 optimizations).
- Connect 3 layers of applications together (Windows Form application -> python script -> benchmark C code compiled with **gcc**.
- RAM speed logarithmic dependency on page size.
- Used reverse engineering knowledge to check if the compiled tests actually do what they are expected to do.
- Challenging for me to subtract time of execution of loop operations like increment and compare to measure just the read/write operations from memory/storage.

I have run the application on 5 different laptops running Windows 10 or 11. In all cases the results were as expected. The frequency of CPU matches exactly the one which is advertised with a precision of 2 decimals, storage speeds match those advertised and RAM speeds are a bit slower than maximum rated speeds of the dims.

What other features can be added or what can be improved? CPU benchmark can include tests that measure execution time of various types of operations like integer addition subtraction, division or operations with floating point numbers. Since the CPU will operate at the same frequency, the number of instructions per second would be the same, these tests will test the data speed transfer between the main CPU and the FPU and show how costly is a floating-point operation compared to an integer one.

Also at least 1000 tests should be run to test RAM read/write speeds with a page size of 1, 2, 4 and 8 bytes, make their average and input those values in the script that calculates logarithmic function's parameters. This is important because those values will stay the same for every RAM test that is performed. At the moment I think this function's parameters are dependent on the stick of ram the benchmark is run on, that's why the RAM is not calculated that way in the app yet. More testing should be done on other computers to test this theory. If it is RAM stick dependent then at the first run of the benchmark on a PC a calibration needs to be done. A lot of tests should be performed in order to get a good approximation of **a** and **b**. Otherwise these parameters can be hardcoded in the benchmark app.

7. Bibliography

[1] "GCC 8.1 Manual" [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Optimize-Options.html.

[2] "Optimization discussion stack overflow" [Online]. Available: https://stackoverflow.com/questions/8299643/how-to-compile-c-program-without-any-optimization.

[3] "Windows Forms documentation" [Online]. Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/?view=netdesktop-7.0.

[4] "Windows Form design ideas" [Online]. Available: https://en.idei.club/22013-winform-design.html.

[5] "UML Diagrams" [Online]. Available: https://app.diagrams.net/#.

[6] "RDTSC" [Online]. Available: https://community.intel.com/t5/Intel-C-Compiler/How-To-Read-The-RDTSC-With-Intel-C-Compiler-v-11-0/td-p/873887.

[7] "RDTSC return value registers" [Online]. Available: https://www.felixcloutier.com/x86/rdtsc.

[8] "Clock function cpp" [Online]. Available: https://en.cppreference.com/w/cpp/chrono/c/clock.

[9] "Ghidra" [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/releases.

[10] “fwrite” [Online]. Available: https://en.cppreference.com/w/c/io/fwrite.

[11] “fread” [Online]. Available: https://en.cppreference.com/w/c/io/fread.

[12] “StackOverflow question about number of optimizations” [Online]. Available: https://stackoverflow.com/questions/8299643/how-to-compile-c-program-without-any-optimization.

[13] “Quora RAM discussion” [Online]. Available: https://www.quora.com/Do-I-need-the-same-type-of-RAM-on-all-slots#:~:text=Yes.,on%20DDR4%20slot%2C%20etc).&text=Originally%20Answered%3A%20Do%20I%20need,of%20RAM%20in%20both%20slots%3F.

[14] “Scipy” [Online]. Available: https://scipy.org/.

[15] “Dynamic frequency scaling” [Online]. Available:

https://en.wikipedia.org/wiki/Dynamic_frequency_scaling.

[16] “Intel I7-11800H” [Online]. Available:

https://www.intel.com/content/www/us/en/products/sku/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz/specifications.html?wapkw=i7%2011800h.

[17] “Matplotlib” [Online]. Available: https://matplotlib.org/stable/index.html.

[18] “Matplotlib xticks()” [Online]. Available: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.xticks.html#matplotlib.pyplot.xticks.

[19] “Benchmark bias” [Online]. Available:

https://spearblade.com/userbenchmark-is-run-by-a-whining-crybaby/.