Benchmark CPU and Memory

Sandor Marian

Technical Univeristy of Cluj Napoca

02.11.2020

Contents

1. Introduction

The goal of the project is to create a benchmark application for measuring CPU and Memory performance. To obtain a representation of the performance for the CPU there will be a number of tests consisting in integer operations, floating point operations and data compression. The final score will be computed based on each individual test. For testing the memory, we are interested in the speed for accessing data. Therefore, the tests will consist in reading and writing data into files sequentially as well as randomly.

The plan is to divide the implementation of the application into 4 stages. Each stage will have a two weeks deadline.

- First Stage

In the first stage we will prepare the working environment for the application. Implement the base structure of the program as well as the functionality of getting the specs of the hardware. Identify the CPU as well as the memory related information.

- Second Stage

During this stage we will focus on implementing the CPU benchmark. This consists in implementing all the tests proposed and generating the sample data with which the tests will work.

- Third Stage

This stage is meant for the memory benchmark functionality of the application. This is when the memory tests will be implemented.

- Fourth Stage

In the final stage, the goal is to process and display the recorded data. The focus will be mainly on creating a graphical user interface which will provide a friendly and easy to understand way of displaying the results to the user.

2. Bibliographic Research

In computing, a benchmark is the result of running a computer program, or a set of programs, in order to assess the relative performance of an object, by running a number of standard tests and trials against it. The term is also commonly used for specially-designed benchmarking programs themselves. Benchmarking is usually associated with assessing performance characteristics of computer hardware. [1]

Synthetic benchmarks are designed to have easily repeatable results for accurate comparisons and minimal bottlenecks across different isolated tests. This makes it easier to test individual parts -like and SSD or Processor- without other factors affecting results, but it also may not reflect a user's actual experience with the machine since it doesn't test everything working together. [2]

The tests for the CPU explained:

- Integer Test

The Integer Test aims to measure how fast the CPU can perform mathematical integer operations. An integer is a whole number with no fractional part. This is a basic operation in all computer software and provides a good indication of 'raw' CPU throughput. The test uses large sets of an equal number of random 32-bit and 64-bit integers and adds, subtracts multiplies and divides these numbers. [3]

- Floating Point Test

The Floating-Point Test performs the same operations as the Integer Test however with floating point numbers, using an equal amount of single precision (32-bit) and double precision (64-bit) values. A floating-point number is a number with a fractional part. These kinds of numbers are handled quite differently in the CPU compared to Integer numbers as well as being quite commonly used, therefore they are tested separately. [4]

- Compression Test

The Compression Test measures the speed that the CPU can compress blocks of data into smaller blocks of data without losing any of the original data. The result is reported in Kilobytes per Second. This test uses complex data structures and complex data manipulation techniques to perform a function that is very common in software applications. The compression test uses Crypto++ Gzip. [5]

The tests for the memory explained:

- Read Uncached

This test measures the time taken to read a large block of memory. Measured in GB/s, larger values are better. The test will use large block such that it will be too large to be held in cache. [6]

- Write

This test measures the time taken to write information into memory. Measured in GB/s, larger values are better. Similar to the read uncached test, except it tests the writing performance instead. [7]

3. Requirements & Features

The final product will be able to assess the computing power of the processor as well as the speed at which the memory manipulates data. All of the results will be displayed to the user along with a final score which can be used as a comparison reference between different machines.

❖ **IOPS Test**

For this test we are working with integer values and we want to find out the number of operations performed per second. There will be a number of variables initialized and then a loop that will process them. Each iteration of the loop performs a series of additions and multiplications with the variables. The number of iterations will be set to some big value (i.e 100.000) such that the overlay of the loop will be insignificant, thus the results will be more accurate.

❖ **FLOPS Test**

For this test we are working with floating point values and we want to find out the number of operations performed per second. There will be a number of variables initialized and then a loop that will process them. Each iteration of the loop performs a series of additions and multiplications with the variables. The number of iterations will be set to some big value (i.e 100.000) such that the overlay of the loop will be insignificant, thus the results will be more accurate.

❖ **Compression Test**

For this test we will use big random generated files (i.e 1GB) on which we will apply Deflate compression algorithm. The time needed to compress a number of such files will be measured and divided by the total size of the files. The result will be given as MB per second. We will use the Crypto++ library [8] for the compression functionalities.

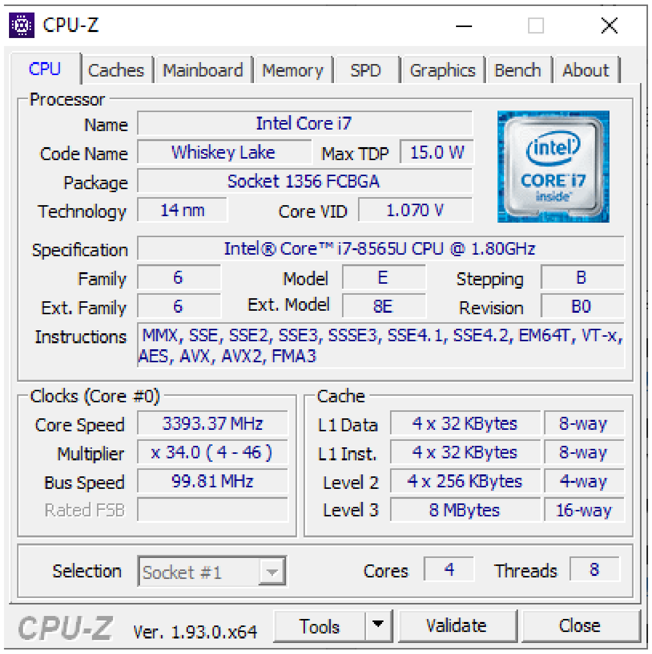
- ❖ **Read Uncached Tesst**

For this test we will dynamically create a big array of chars (1 MB) and then perform a sequence of buffer reads at random positions. The throughput is given by the number of bytes read per second.

- ❖ **Write Test**

For this test we will dynamically create a big array of chars (1 MB) and then perform a sequence of buffer writes at random positions. The throughput is given by the number of bytes written per second.

The application will also gather and display information about the CPU and RAM memory of the machine it runs on. The feature is similar to what CPU-Z does. For the CPU it will show the following information: CPU vendor, CPU brand, Stepping ID, Model, Family ID, Extended Model ID, Extended Family ID, Part of Instruction set, Number of Physical Processors, Number of Logical Processors and Number of Logical Processors per Core. As for the memory it will display the Capacity, In use memory percent and Free Memory.


CPU-Z
CPU | Caches | Mainboard | Memory | SPD | Graphics | Bench | About
intel CORE i7 inside
CPU-Z Ver. 1.93.0.x64
Tools
Validate
Close

Processor

Name	Intel Core i7				
Code Name	Whiskey Lake	Max TDP	15.0 W		
Package	Socket 1356 FCBGA				
Technology	14 nm	Core VID	1.070 V		
Specification	Intel® Core™ i7-8565U CPU @ 1.80GHz				
Family	6	Model	E	Stepping	B
Ext. Family	6	Ext. Model	8E	Revision	B0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3				

Clocks (Core #0)

Core Speed	3393.37 MHz
Multiplier	x 34.0 (4 - 46)
Bus Speed	99.81 MHz
Rated FSB	

Cache

L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	4-way
Level 3	8 MBytes	16-way

Selection Socket #1 Cores 4 Threads 8

Figure 1 CPU-Z Interface

4. Design

The application will be composed of 5 parts each having its own goal. A part refers to a module in which all the necessary structures and functions are implemented in order to achieve its goal. The modules will be CPU_info, MEM_info, CPU_tests, MEM_tests and GUI.

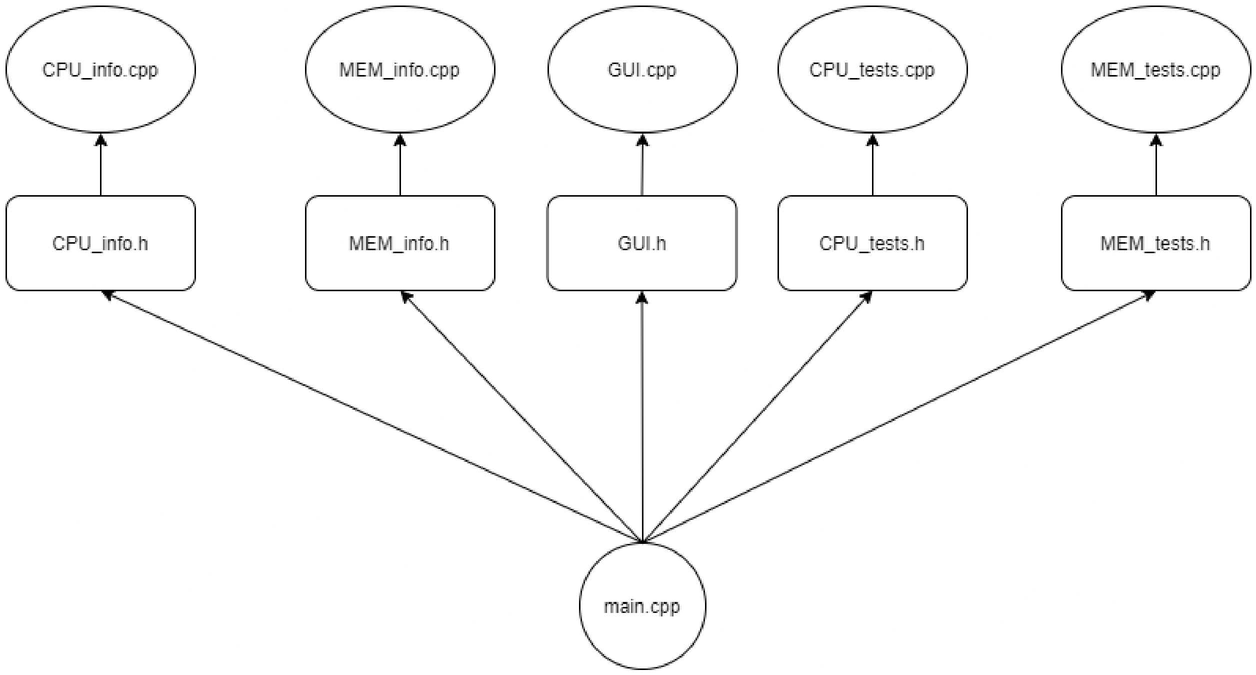
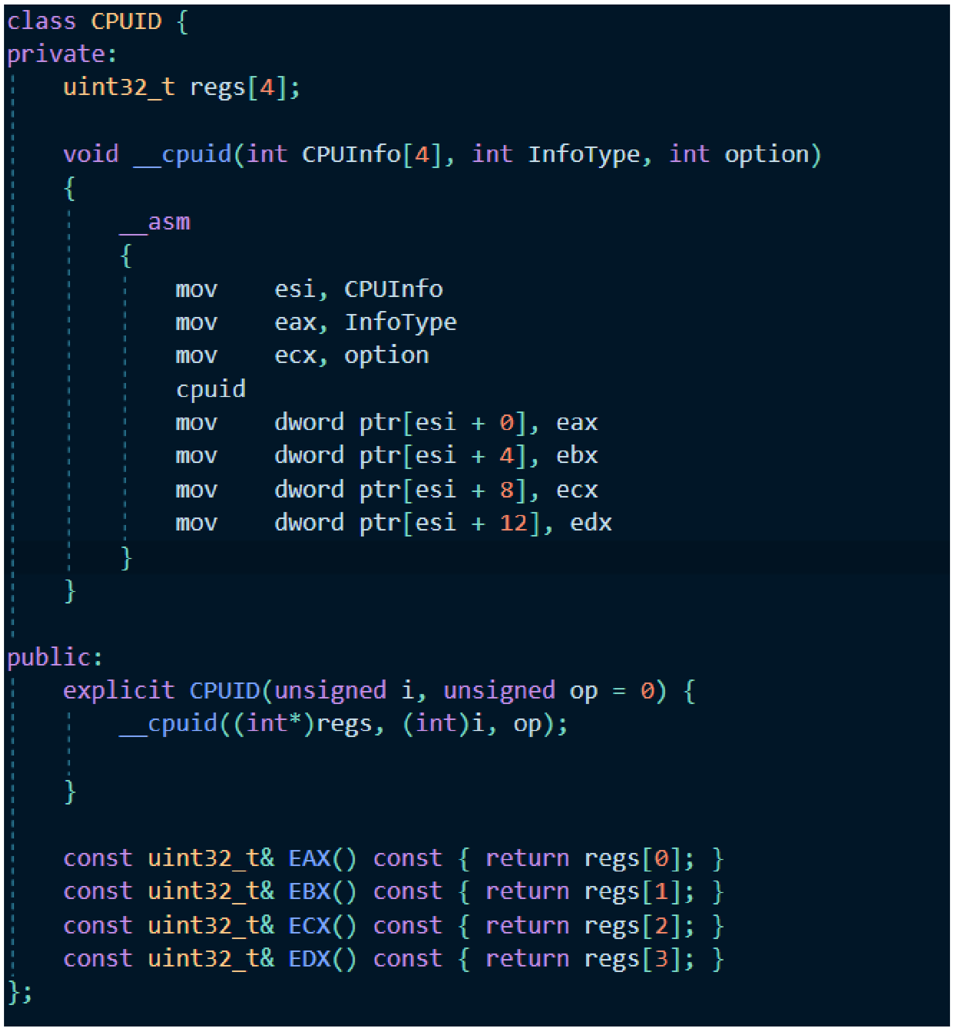


Figure 2 Project Structure

5. Implementation

First, we will take care of fetching the CPU and memory information to the user. Taking a look to the project structure presented above (Figure 2), we can see that the two modules that provide these functionalities reside in the CPU_info and MEM_info headers along with their implementation files. We start off by implementing the CPU_info module. Since we are extracting the data using the cupid assembly instruction, we need to write an interface through which the code will call this instruction with different parameters. This is al covered by the class CPUID implemented in the CPU_info.h (Figure 3). The class constructor takes two arguments, first one is the value of the EAX register and the second is optional for calls that require a second parameter to be specified in the ECX register. The configurations with which cupid can be called are found here [9] [10]. Further on, there are 5 functions that query information about the processor, mainly: CPUID_getVendor(), CPUID_getBrand(), CPUID_getProcessorVersion(), CPUID_getInstructions() and CPUID_getTopology() all taking use of the CPUID class mentioned. The memory information is obtained through calls to a function provided in the windows.h namely GlobalMemoryStatusEx (). It returns the data in a variable of type MEMORYSTATUSEX which can be further queried using the dot notation. Additional information about the GlobalMemoryStatusEx() can be found here [11].

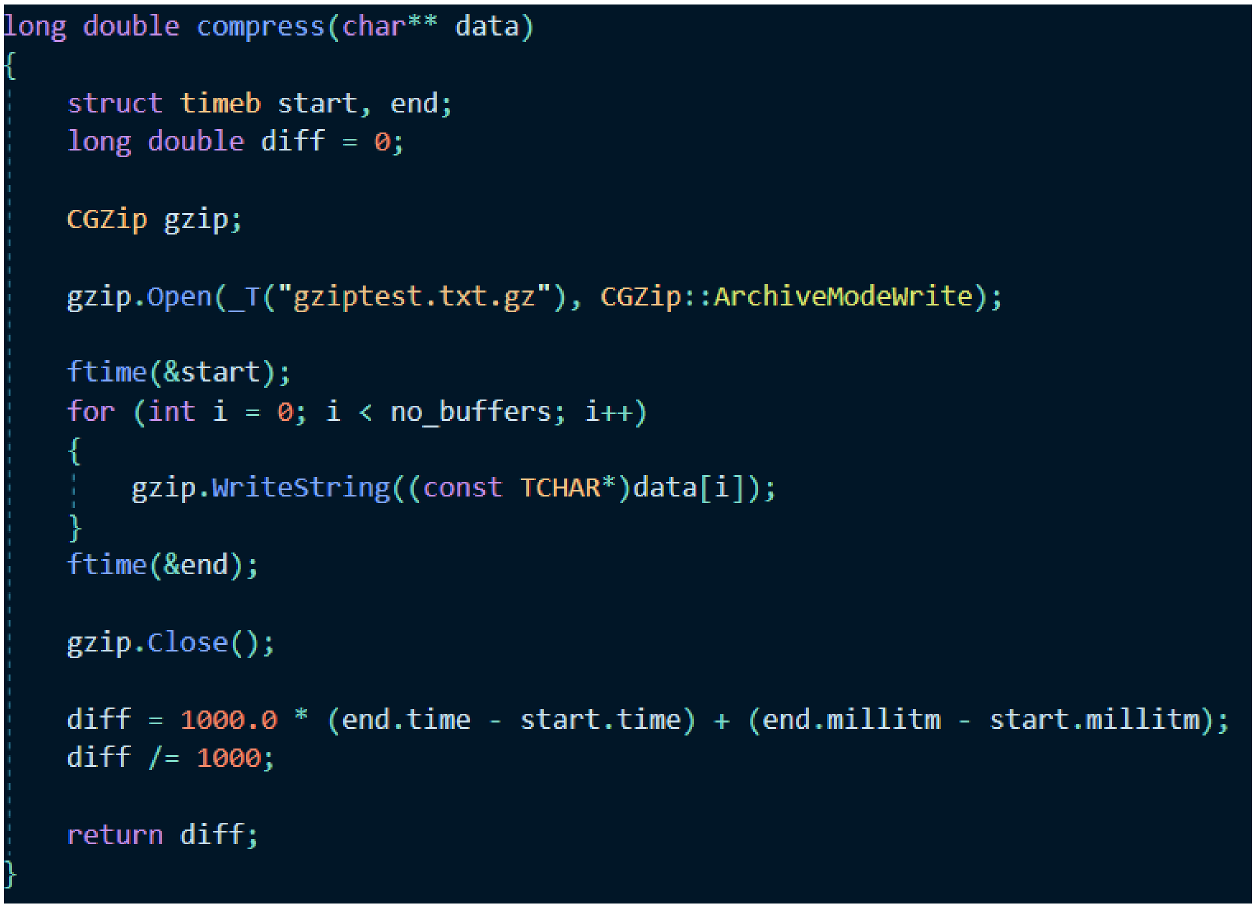


```
class CPUID {
private:
    uint32_t regs[4];

    void __cpuid(int CPUInfo[4], int InfoType, int option)
    {
        __asm
        {
            mov    esi, CPUInfo
            mov    eax, InfoType
            mov    ecx, option
            cpuid
            mov    dword ptr[esi + 0], eax
            mov    dword ptr[esi + 4], ebx
            mov    dword ptr[esi + 8], ecx
            mov    dword ptr[esi + 12], edx
        }
    }

public:
    explicit CPUID(unsigned i, unsigned op = 0) {
        __cpuid((int*)regs, (int)i, op);

    }

    const uint32_t& EAX() const { return regs[0]; }
    const uint32_t& EBX() const { return regs[1]; }
    const uint32_t& ECX() const { return regs[2]; }
    const uint32_t& EDX() const { return regs[3]; }
};
```

Figure 3 CPUID class code

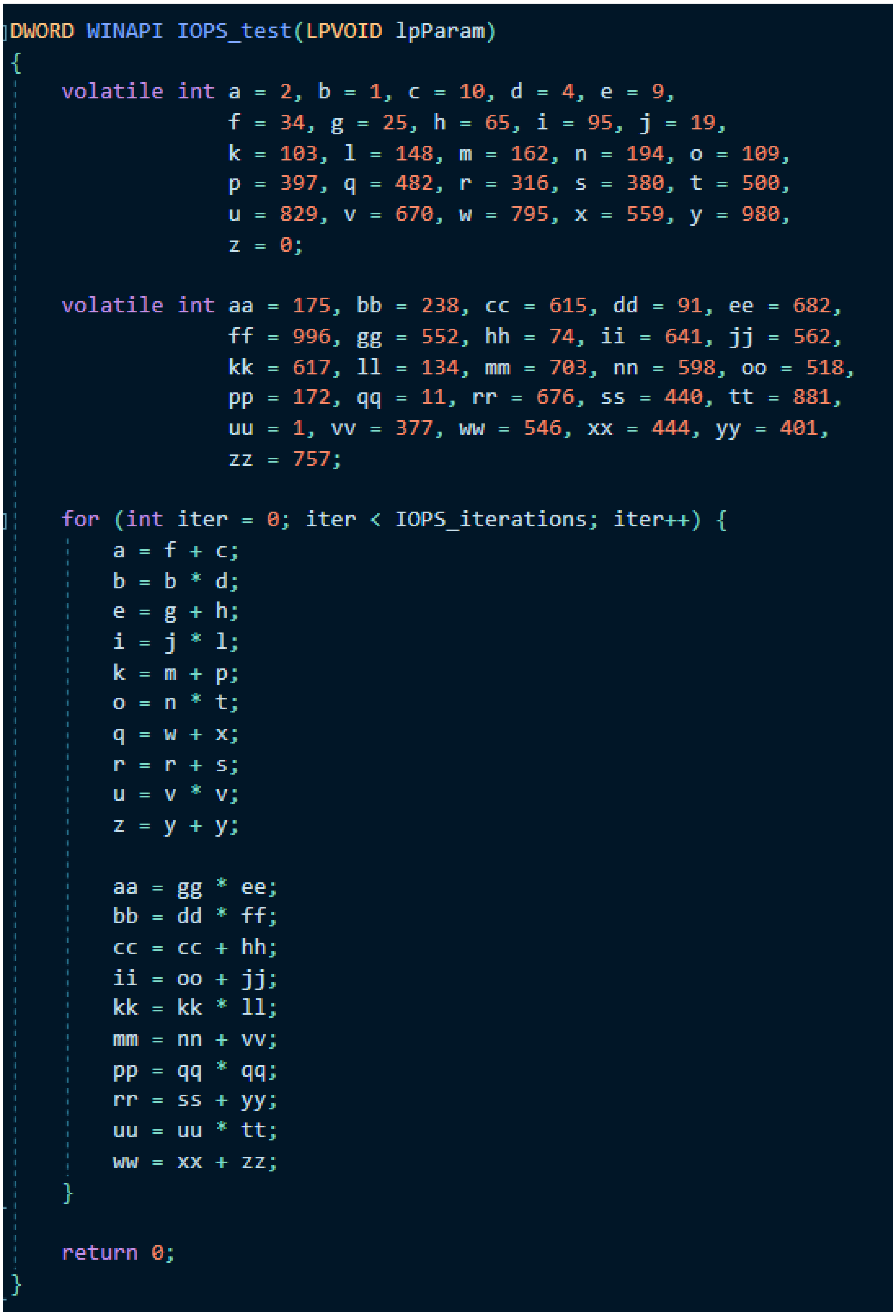
We move on to the implementation of the CPU tests. These can be found in the CPU_test.cpp file and accessed through the header CPU_test.h. As mentioned before, there are three tests that are done independently for the cpu. The first two are the IOPS (figure 5) and FLOPS (figure 6) tests. These tests work by giving a big workload to the processor. In both cases, the workload is composed of a large number of operations (additions and multiplications) and this is achieved by iterating over a set of operations (there are 20 operations executed at each iteration). What makes the difference between the two functions is that one works with integer numbers and the other with floating point numbers which are more demanding for the processor to work with. These two tests are run multiple times with a different number of threads. The more threads are assigned the more operations are to be performed because each of them executes the function once. The number of iterations in the FLOPS and IOPS function is fixed by a constant (500.000) thus we can compute the total number of operations executed when running one of these functions is 500.000 * 20. Taking in considerations the number of threads too, the final number of operations is given by 500.000 * 20 * no_threads. In order to obtain the Gflops and Giops value (giga floating point/integer operations per second) we need to measure the time in which the operations are performed. For this, we use the ftime function provided in the sys\timeb.h header [12]. The last step to obtain the desired value is to divide the number of operations performed by a conversionfactor of 10^9 (bytes to gigabytes) and then by the total time in seconds. In the end we perform the two tests for 1 thread, 2, 4, 8, 16, 32 and 64 and we obtain 7 Gflops values and 7 Iops values.

The third test is the compression test. Here we are interested to find the speed (mb/sec) at which the processor can compress a chunk of 1GB of data. We first generate the data containing random characters and then we apply the compression algorithm. (figure 4) The speed is computed by dividing the size (1024MB) of the data by the time it took to compress it.



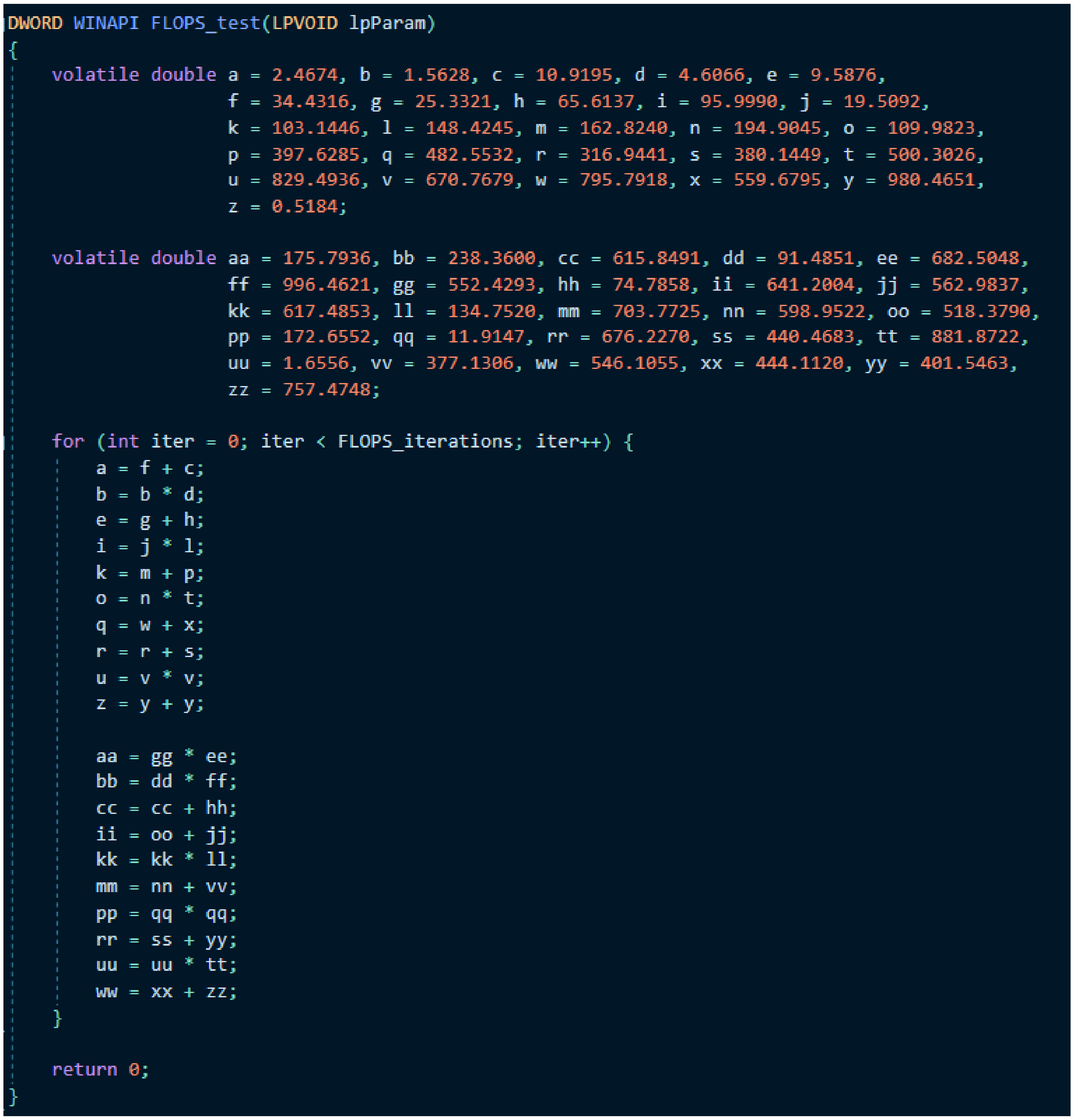
```
long double compress(char** data)
{
    struct timeb start, end;
    long double diff = 0;

    CGZip gzip;

    gzip.Open(_T("gziptest.txt.gz"), CGZip::ArchiveModeWrite);

    ftime(&start);
    for (int i = 0; i < no_buffers; i++)
    {
        gzip.WriteString((const TCHAR*)data[i]);
    }
    ftime(&end);

    gzip.Close();

    diff = 1000.0 * (end.time - start.time) + (end.millitm - start.millitm);
    diff /= 1000;

    return diff;
}
```

Figure 4 Compress test code



```
DWORD WINAPI IOPS_test(LPVOID lpParam)
{
    volatile int a = 2, b = 1, c = 10, d = 4, e = 9,
                 f = 34, g = 25, h = 65, i = 95, j = 19,
                 k = 103, l = 148, m = 162, n = 194, o = 109,
                 p = 397, q = 482, r = 316, s = 380, t = 500,
                 u = 829, v = 670, w = 795, x = 559, y = 980,
                 z = 0;

    volatile int aa = 175, bb = 238, cc = 615, dd = 91, ee = 682,
                 ff = 996, gg = 552, hh = 74, ii = 641, jj = 562,
                 kk = 617, ll = 134, mm = 703, nn = 598, oo = 518,
                 pp = 172, qq = 11, rr = 676, ss = 440, tt = 881,
                 uu = 1, vv = 377, ww = 546, xx = 444, yy = 401,
                 zz = 757;

    for (int iter = 0; iter < IOPS_iterations; iter++) {
        a = f + c;
        b = b * d;
        e = g + h;
        i = j * l;
        k = m + p;
        o = n * t;
        q = w + x;
        r = r + s;
        u = v * v;
        z = y + y;

        aa = gg * ee;
        bb = dd * ff;
        cc = cc + hh;
        ii = oo + jj;
        kk = kk * ll;
        mm = nn + vv;
        pp = qq * qq;
        rr = ss + yy;
        uu = uu * tt;
        ww = xx + zz;
    }

    return 0;
}
```

Figure 5 IOPS test code

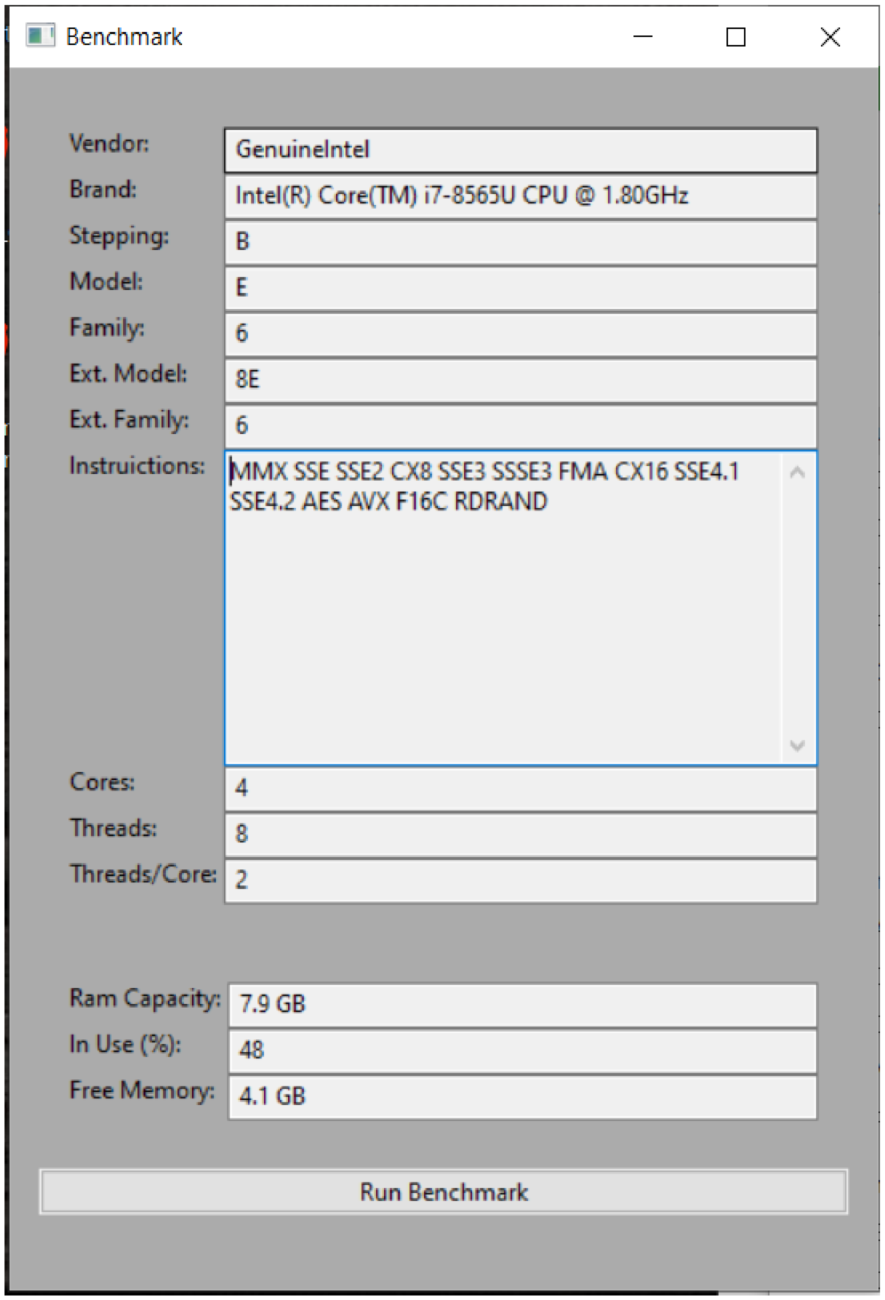


```
DWORD WINAPI FLOPS_test(LPVOID lpParam)
{
    volatile double a = 2.4674, b = 1.5628, c = 10.9195, d = 4.6066, e = 9.5876,
                    f = 34.4316, g = 25.3321, h = 65.6137, i = 95.9990, j = 19.5092,
                    k = 103.1446, l = 148.4245, m = 162.8240, n = 194.9045, o = 109.9823,
                    p = 397.6285, q = 482.5532, r = 316.9441, s = 380.1449, t = 500.3026,
                    u = 829.4936, v = 670.7679, w = 795.7918, x = 559.6795, y = 980.4651,
                    z = 0.5184;

    volatile double aa = 175.7936, bb = 238.3600, cc = 615.8491, dd = 91.4851, ee = 682.5048,
                    ff = 996.4621, gg = 552.4293, hh = 74.7858, ii = 641.2004, jj = 562.9837,
                    kk = 617.4853, ll = 134.7520, mm = 703.7725, nn = 598.9522, oo = 518.3790,
                    pp = 172.6552, qq = 11.9147, rr = 676.2270, ss = 440.4683, tt = 881.8722,
                    uu = 1.6556, vv = 377.1306, ww = 546.1055, xx = 444.1120, yy = 401.5463,
                    zz = 757.4748;

    for (int iter = 0; iter < FLOPS_iterations; iter++) {
        a = f + c;
        b = b * d;
        e = g + h;
        i = j * l;
        k = m + p;
        o = n * t;
        q = w + x;
        r = r + s;
        u = v * v;
        z = y + y;

        aa = gg * ee;
        bb = dd * ff;
        cc = cc + hh;
        ii = oo + jj;
        kk = kk * ll;
        mm = nn + vv;
        pp = qq * qq;
        rr = ss + yy;
        uu = uu * tt;
        ww = xx + zz;
    }

    return 0;
}
```

Figure 6 FLOPS test code

Last, we have to implement the memory tests. Here, there are two tests, for reading and for writing, but each of them should be performed in two situations. These two situations are when the data is cached and when it's uncached. The cached situation ca be easily simulated by performing the operations of read/write sequentially. For the uncached situation we could choose to read/write from random locations in our allocated memory zone or we can choose a stride with which we can iterate over our memory zone. We prefer the second variant. Now the question is how big the stride should be and the first intuitive answer is that is should be bigger than the cache memory of the configuration on which the tests are ran. Thus, we chose a stride of 8MB which is fairly big for most of the current configurations available. The next thing we need to take care of is to allocate a memory zone with which we can work. We define the size of it to be of 1 GB. Then we define 4 functions for each test and situation and we make sure that we are timing each function independently. For writing tests, we only write to the memory location pointed by the data array and for reading we simply read each byte to a buffer variable. The results we want are the speed of each of the mentioned operation so we divide the 1GB amount of memory to the time taken to perform that test in seconds. We finally obtain four speeds (GB/s) for each of the situations: rd_cached_speed, rd_uncached_speed, wr_cached_speed and wr_uncached_speed.

The final thing we need to implement is the graphical user interface in order to display the information to the user and provide a way to run the tests by pressing a button. In order to create a graphical interface in c++ we need to use a third-party library. The one I chose is wxWidgets [13]. It is fairly easy to create a simple graphical user interface with it and the code is straight forward (figure 7 presents the gui). One more thing to be done is to provide the user a score that can be used to compare his configuration against other. We compute this score by a adding the weighted average of the Gflops, Iops and compression test with the weighted average of the writing and reading tests.

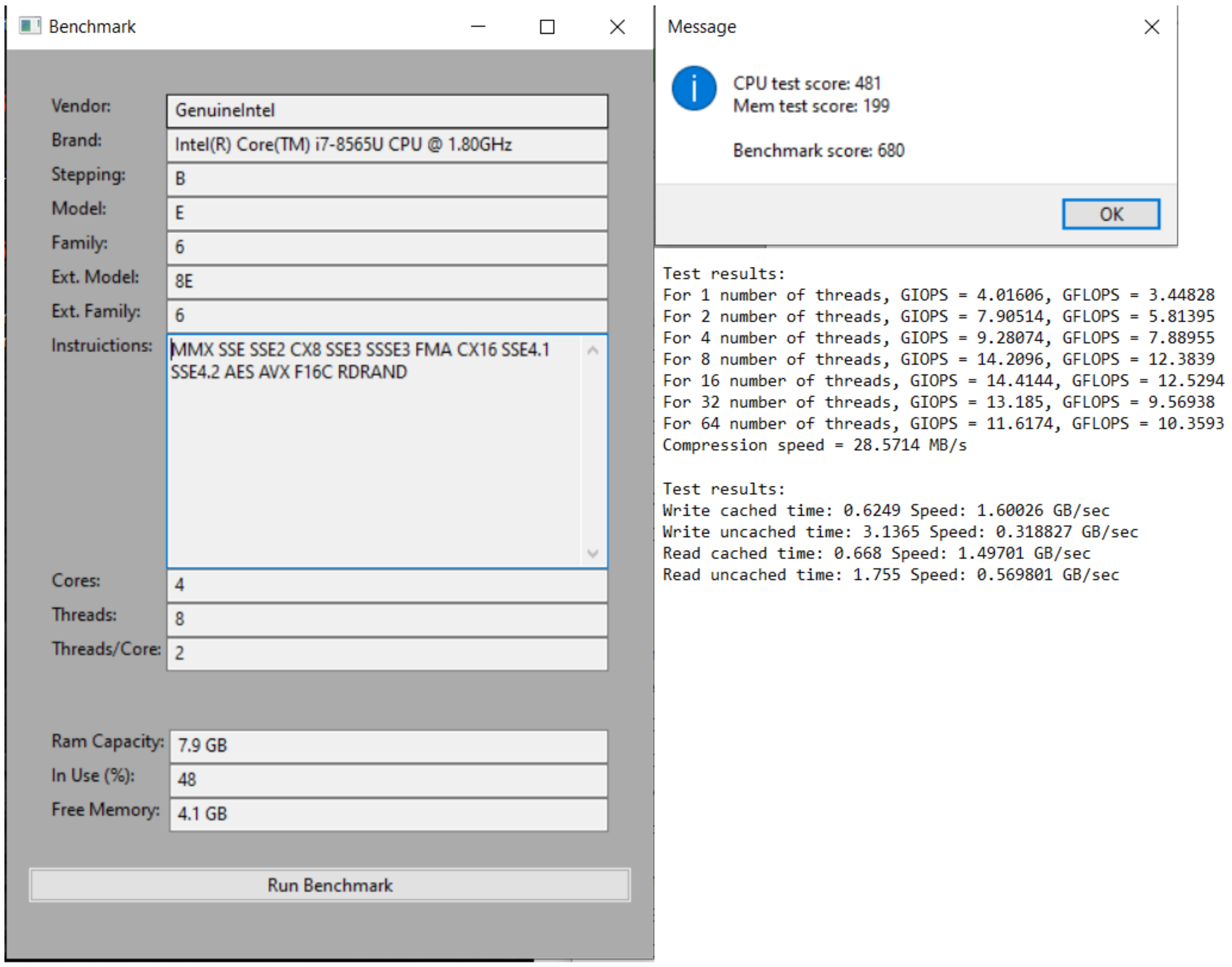

Benchmark

Vendor:	GenuineIntel
Brand:	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:	B
Model:	E
Family:	6
Ext. Model:	8E
Ext. Family:	6
Instruictions:	MMX SSE SSE2 CX8 SSE3 SSSE3 FMA CX16 SSE4.1 SSE4.2 AES AVX F16C RDRAND
Cores:	4
Threads:	8
Threads/Core:	2
Ram Capacity:	7.9 GB
In Use (%):	48
Free Memory:	4.1 GB

Figure 7 Graphical User Interface

6. Tests/Experiments

For tests, I aimed to run the benchmark on different configurations and to see how realistic the results are. I have run the benchmark tests on four different configurations (3 intel and 1 amd based) and the results were very pleasing. The results along with the logs can be seen below.



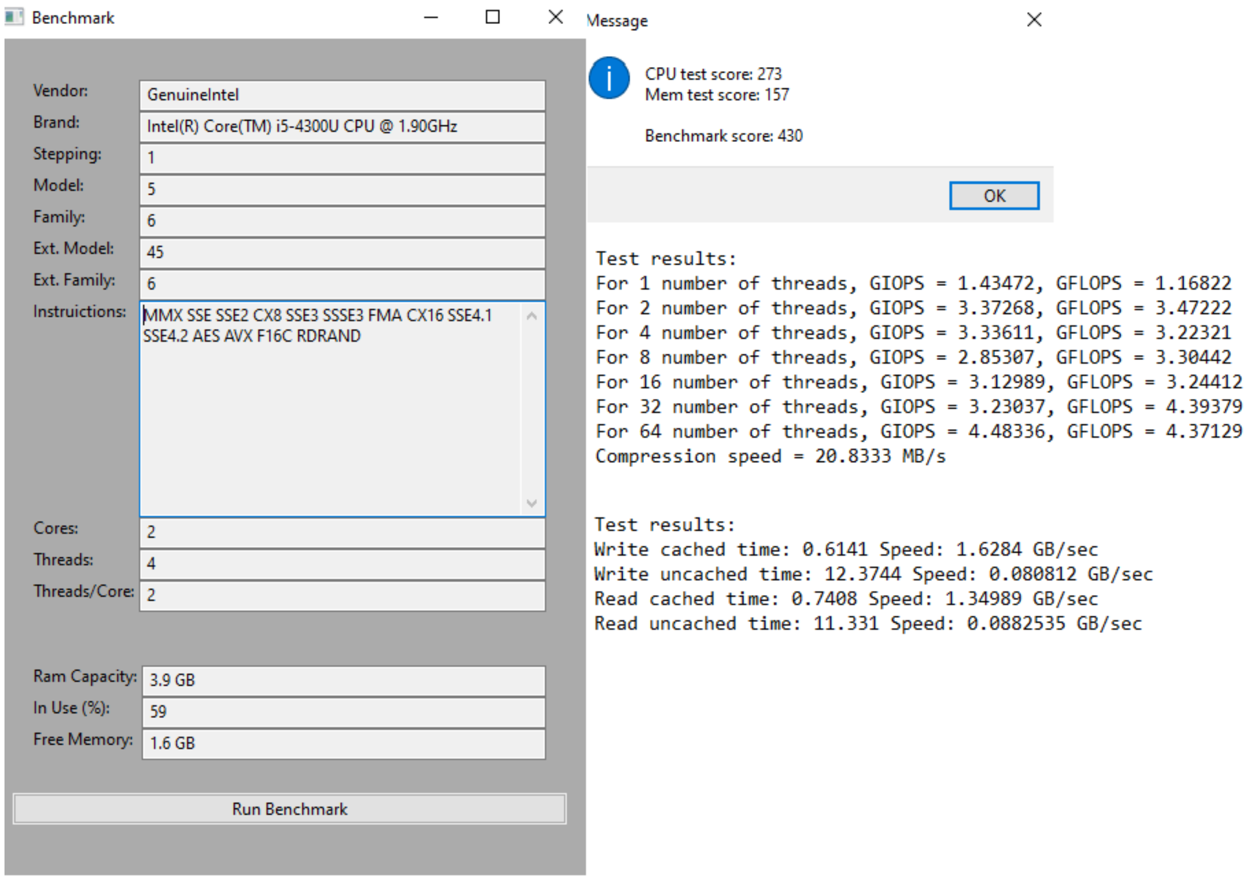
Vendor:	GenuineIntel
Brand:	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:	B
Model:	E
Family:	6
Ext. Model:	8E
Ext. Family:	6
Instruictions:	MMX SSE SSE2 CX8 SSE3 SSSE3 FMA CX16 SSE4.1 SSE4.2 AES AVX F16C RDRAND
Cores:	4
Threads:	8
Threads/Core:	2
Ram Capacity:	7.9 GB
In Use (%):	48
Free Memory:	4.1 GB

CPU test score: 481
Mem test score: 199

Benchmark score: 680

```
Test results:
For 1 number of threads, GIOPS = 4.01606, GFLOPS = 3.44828
For 2 number of threads, GIOPS = 7.90514, GFLOPS = 5.81395
For 4 number of threads, GIOPS = 9.28074, GFLOPS = 7.88955
For 8 number of threads, GIOPS = 14.2096, GFLOPS = 12.3839
For 16 number of threads, GIOPS = 14.4144, GFLOPS = 12.5294
For 32 number of threads, GIOPS = 13.185, GFLOPS = 9.56938
For 64 number of threads, GIOPS = 11.6174, GFLOPS = 10.3593
Compression speed = 28.5714 MB/s

Test results:
Write cached time: 0.6249 Speed: 1.60026 GB/sec
Write uncached time: 3.1365 Speed: 0.318827 GB/sec
Read cached time: 0.668 Speed: 1.49701 GB/sec
Read uncached time: 1.755 Speed: 0.569801 GB/sec
```

Figure 8 Results for Configuration 1



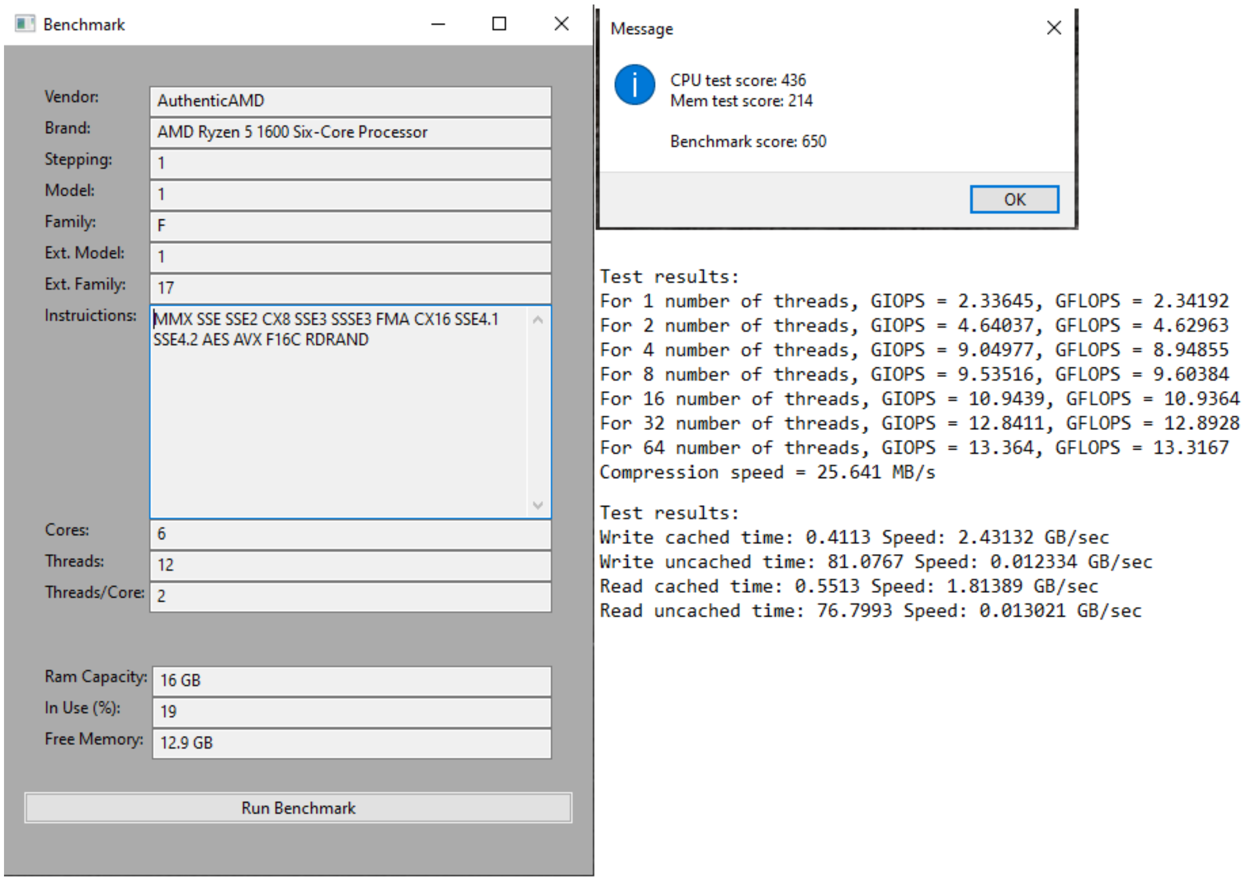
Vendor:	GenuineIntel
Brand:	Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz
Stepping:	1
Model:	5
Family:	6
Ext. Model:	45
Ext. Family:	6
Instruictions:	MMX SSE SSE2 CX8 SSE3 SSSE3 FMA CX16 SSE4.1 SSE4.2 AES AVX F16C RDRAND
Cores:	2
Threads:	4
Threads/Core:	2
Ram Capacity:	3.9 GB
In Use (%):	59
Free Memory:	1.6 GB

CPU test score: 273
Mem test score: 157

Benchmark score: 430

```
Test results:
For 1 number of threads, GIOPS = 1.43472, GFLOPS = 1.16822
For 2 number of threads, GIOPS = 3.37268, GFLOPS = 3.47222
For 4 number of threads, GIOPS = 3.33611, GFLOPS = 3.22321
For 8 number of threads, GIOPS = 2.85307, GFLOPS = 3.30442
For 16 number of threads, GIOPS = 3.12989, GFLOPS = 3.24412
For 32 number of threads, GIOPS = 3.23037, GFLOPS = 4.39379
For 64 number of threads, GIOPS = 4.48336, GFLOPS = 4.37129
Compression speed = 20.8333 MB/s

Test results:
Write cached time: 0.6141 Speed: 1.6284 GB/sec
Write uncached time: 12.3744 Speed: 0.080812 GB/sec
Read cached time: 0.7408 Speed: 1.34989 GB/sec
Read uncached time: 11.331 Speed: 0.0882535 GB/sec
```

Figure 9 Results for Configuration 2



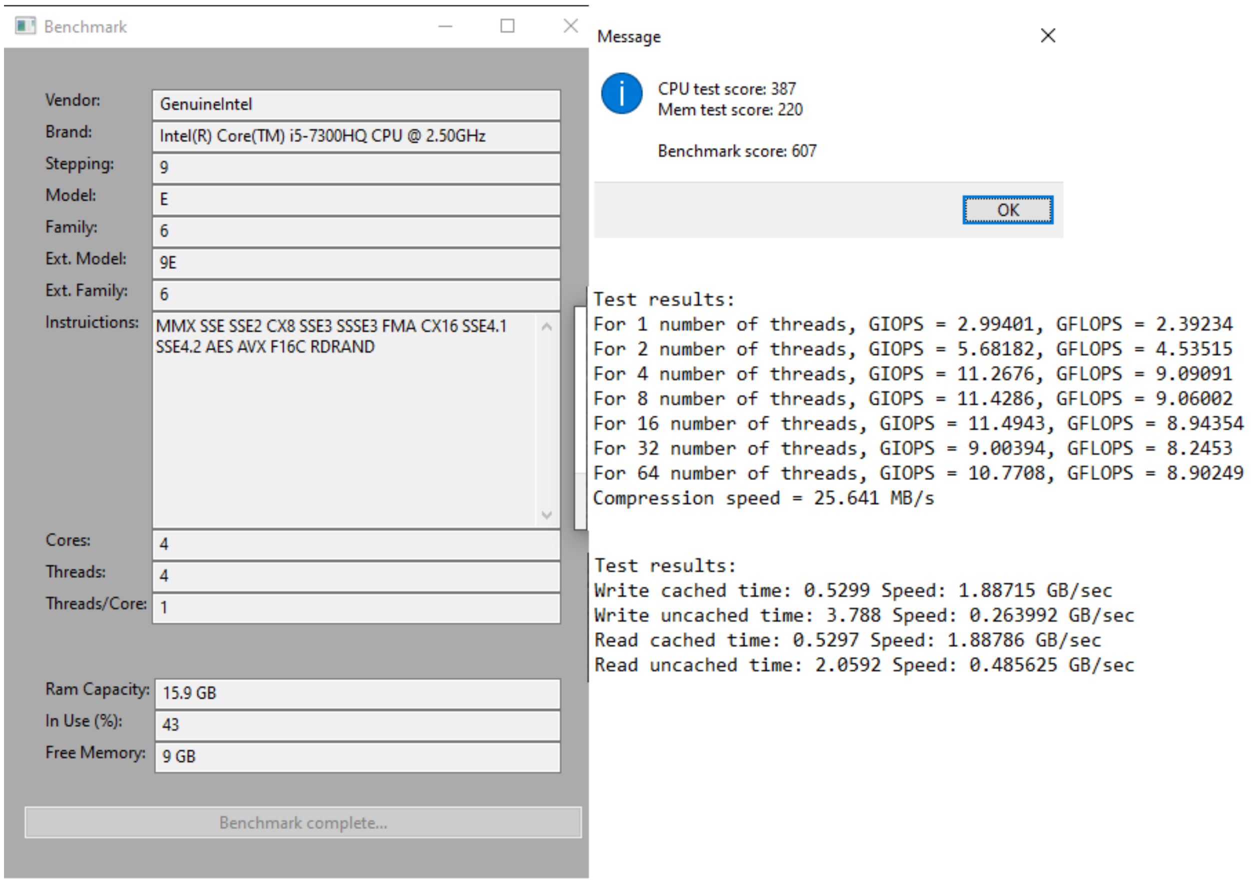
Vendor:	AuthenticAMD
Brand:	AMD Ryzen 5 1600 Six-Core Processor
Stepping:	1
Model:	1
Family:	F
Ext. Model:	1
Ext. Family:	17
Instruictions:	MMX SSE SSE2 CX8 SSE3 SSSE3 FMA CX16 SSE4.1 SSE4.2 AES AVX F16C RDRAND
Cores:	6
Threads:	12
Threads/Core:	2
Ram Capacity:	16 GB
In Use (%):	19
Free Memory:	12.9 GB

CPU test score: 436
Mem test score: 214

Benchmark score: 650

```
Test results:
For 1 number of threads, GIOPS = 2.33645, GFLOPS = 2.34192
For 2 number of threads, GIOPS = 4.64037, GFLOPS = 4.62963
For 4 number of threads, GIOPS = 9.04977, GFLOPS = 8.94855
For 8 number of threads, GIOPS = 9.53516, GFLOPS = 9.60384
For 16 number of threads, GIOPS = 10.9439, GFLOPS = 10.9364
For 32 number of threads, GIOPS = 12.8411, GFLOPS = 12.8928
For 64 number of threads, GIOPS = 13.364, GFLOPS = 13.3167
Compression speed = 25.641 MB/s

Test results:
Write cached time: 0.4113 Speed: 2.43132 GB/sec
Write uncached time: 81.0767 Speed: 0.012334 GB/sec
Read cached time: 0.5513 Speed: 1.81389 GB/sec
Read uncached time: 76.7993 Speed: 0.013021 GB/sec
```

Figure 10 Results for Configuration 3



Vendor: GenuineIntel
Brand: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Stepping: 9
Model: E
Family: 6
Ext. Model: 9E
Ext. Family: 6
Instruictions: MMX SSE SSE2 CX8 SSE3 SSSE3 FMA CX16 SSE4.1 SSE4.2 AES AVX F16C RDRAND
Cores: 4
Threads: 4
Threads/Core: 1
Ram Capacity: 15.9 GB
In Use (%): 43
Free Memory: 9 GB

CPU test score: 387
Mem test score: 220

Benchmark score: 607

```
Test results:
For 1 number of threads, GIOPS = 2.99401, GFLOPS = 2.39234
For 2 number of threads, GIOPS = 5.68182, GFLOPS = 4.53515
For 4 number of threads, GIOPS = 11.2676, GFLOPS = 9.09091
For 8 number of threads, GIOPS = 11.4286, GFLOPS = 9.06002
For 16 number of threads, GIOPS = 11.4943, GFLOPS = 8.94354
For 32 number of threads, GIOPS = 9.00394, GFLOPS = 8.2453
For 64 number of threads, GIOPS = 10.7708, GFLOPS = 8.90249
Compression speed = 25.641 MB/s

Test results:
Write cached time: 0.5299 Speed: 1.88715 GB/sec
Write uncached time: 3.788 Speed: 0.263992 GB/sec
Read cached time: 0.5297 Speed: 1.88786 GB/sec
Read uncached time: 2.0592 Speed: 0.485625 GB/sec
```

Figure 11 Results for Configuration 4

7. Conclusion

The goal of the project was to create a benchmark app that will test the CPU and memory of the configuration that it runs on. Additionally, it collects data about the CPU and memory and displays it to the user.

I found the project to be very interesting and it felt nice to get over all the challenging parts of it. One thing that I had troubles with was the cupid instruction. It took me a bit of time until I understand how to query the CPU with it and gather useful information. As for the tests, they were not something new for me so I didn't face many difficulties in implementing them.

As for the future, there are some things about the project that can be improved. The first one which is also quite important, is to make the application cross-platform as it for the moment it runs only on windows. Another improvement would be to add one more test that will take care of the graphical part of the configuration such that the test would be more complete.

8. Bibliographic Study

[1] “Category:Benchmarks(computing)” [Online]. Available: https://en.wikipedia.org/wiki/Category:Benchmarks_(computing)

[2] “The Differences Between Syntethic, Real World, and Hybrid Benchmarks” [Online]. Available:

https://lifehacker.com/the-differences-between-synthetic-real-world-and-hybr-1663435313

[3] “Integer Maths Test” [Online]. Available: https://www.cpubenchmark.net/cpu_test_info.html

[4] “Floating Point Math Test” [Online]. Available: https://www.cpubenchmark.net/cpu_test_info.html

[5] “Compression Test” [Online]. Available: https://www.cpubenchmark.net/cpu_test_info.html

[6] “Read Uncached” [Online]. Available: https://www.memorybenchmark.net/graph_notes.html

[7] “Write” [Online]. Available:

https://www.memorybenchmark.net/graph_notes.html

[8] “Crypto++ library” [Online]. Available: https://www.cryptopp.com

[9] “CPUID configurations source 1” [Online]. Available: https://www.felixcloutier.com/x86/cpuid

[10] “CPUID configurations source 2” [Online]. Available: https://c9x.me/x86/html/file_module_x86_id_45.html

[11] “GlobalMemoryStatusEx additional info” [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-globalmemorystatusex

[12] “sys/timeb.h” [Online]. Available: https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/ftime-ftime32-ftime64?view=msvc-160

[13] “wxwidgets library” [Online]. Available: https://www.wxwidgets.org/