# Device Specifications

MSI GE73 Raider RGB 8RF, running an Intel i7-8750H @2.20GHz with 6 cores and 12 logical processors. 23.8GB of usable RAM with a 64-bit Operating System. Operating System, Windows 11 Pro. The software used for this assignment: Visual Studio Code v1.59.1 and Ubuntu 18.04 from the Windows Store in order to make use of the Windows Subsystem for Linux.

# How the tests were conducted

## Setup:

Each program file was planned to execute the phrase, “The quick brown fox jumps over the lazy dog.”, a total of 25,000 times. To do this each program file had to be coded to execute said phrase. To execute the program files n about of times, a tester file was created to execute each program file after they have been compiled. How to compile the above-mentioned program files:

|  |  |
| --- | --- |
| hello.asm | yasm -f elf64 -g drawf2 -l hello.lst hello.asm  ld -o hello hello.o |
| hello.cpp | g++ -o hello hello.cpp |
| hello.java | javac hello.java |
| hello.py | python3 hello.py |

The program specific software needed to compile and run their respective programs:

|  |  |  |
| --- | --- | --- |
| Assembly | yasm 1.3.0 | Instillation: [*click here*](https://zoomadmin.com/HowToInstall/UbuntuPackage/yasm) |
| C++ | g++ 9.3.0 | Instillation: [*click here*](https://linuxconfig.org/how-to-install-g-the-c-compiler-on-ubuntu-20-04-lts-focal-fossa-linux) |
| Java | openjdk 11.0.11 2021-04-20 | Instillation: [*click here*](https://ubuntu.com/tutorials/install-jre#2-installing-openjdk-jre) |
| Python | Python 3.8.5 | Instillation: [*click here*](https://phoenixnap.com/kb/how-to-install-python-3-ubuntu) |

## The tester.sh

This file has a single task of calling the executable file of the different programs in a for-loop with iterates 50 times. This is skeleton of this file:

for((counter = 0; counter < 50; counter++))

do

    # ./hello for C++ and Assembly

    # java hello for Java

    # python3 hello.py for Python

done

Between the do and done is where the respective executable file will be called to execute the programs. The counter variable will iterate from 0 – 49, equating to 50 iterations.

## The Testing

After each program had been compiled, an associated *tester.sh* file was used to execute their associated executable programs a certain number of times. This was accomplished by calling the executable program in a for-loop which iterated several times. To get the time it took for *tester.sh* to execute, the command ***time bash tester.sh*** was used in the terminal in the respected directory. This was used to print out the system time of the file which executed the executable files several times. Each program executed a total amount of 25,000 times, executed 50 per single call on the ***bash tester.sh*** file. The times were recorded in seconds going to a thousandth of a second for more accuracy. Each program had three different phases for testing:

### Phase One: (100 x ***time bash tester.sh)*** = 5000 iterations

* 50 iterations upon one ***time bash tester.sh*** call.

### Phase Two: (10 x ***time bash tester.sh)*** = 5000 iterations

* 500 iterations upon one ***time bash tester.sh*** call.

### Phase Three: (3 x ***time bash tester.sh)*** = 15,000 iterations

* 5000 iterations upon one ***time bash tester.sh*** call.

The motivation behind this was to see how the system would respond upon executing light to heavy execution in each language.

# The Results

The results are in seconds.

## Java

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Average: seconds | Slowest: seconds | Quickest: seconds | Total: seconds |
| 50 | 0.842 | 1.084 | 0.63 | 84.255 |
| 500 | 8.179 | 8.623 | 7.794 | 81.792 |
| 5000 | 80.811 | 79.509 | 82.353 | 242.432 |

## C++

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Average: seconds | Slowest: seconds | Quickest: seconds | Total: seconds |
| 50 | 0.036 | 0.07 | 0.003 | 3.2 |
| 500 | 0.225 | 0.193 | 0.261 | 2.247 |
| 5000 | 1.990 | 2.005 | 1.972 | 5.971 |

## Assembly

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Average: seconds | Slowest: seconds | Quickest: seconds | Total: seconds |
| 50 | 0.015 | 0.024 | 0.000 | 1.591 |
| 500 | 0.123 | 0.161 | 0.098 | 1.23 |
| 5000 | 1.181 | 1.290 | 1.100 | 3.542 |

## Python

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Average: seconds | Slowest: seconds | Quickest: seconds | Total: seconds |
| 50 | 0.189 | 0.405 | 0.083 | 18.947 |
| 500 | 1.794 | 2.055 | 1.635 | 17.938 |
| 5000 | 16.54 | 17.090 | 15.68 | 49.620 |

## Average time(s) for each language

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Assembly | C++ | Java | Python |
| 50 | 0.015 | 0.036 | 0.842 | 0.189 |
| 500 | 0.123 | 0.225 | 8.179 | 1.794 |
| 5000 | 1.181 | 1.990 | 80.811 | 16.54 |

Based off the Averages table we can see Assembly had the quickest execution time for all three categories. C++ came in second, followed by Python and Java being the slowest.

The reason why Assembly was faster even though it had the most lines of code, is due to how it is interpreted by the computer. Assembly essentially serves as direct instructions for the CPU without having to undergo translation through compilers. It is a lower programming language which means it is closer to machine language than the other programming languages.

C++ came in second in execution time. It is faster than Java as Java is a language that requires synchronization upon ever execution. C++ is also more stringent regarding memory location. It is a language that requires more code to monitor and manage where and how data is stored. This becomes an easier conversion from C++ to machine as it is like Assembly in giving instruction to the CPU.

Python is slower than C++ as it is interpreted rather than being compiled. This means that Python does not have primitive types, and everything is interpreted as objects and because of this, Python must store additional data for its Objects.

Java, by far, is the slowest out of the four tested languages. This is due to the language being multi-threaded and requires synchronization upon every execution. This means that the Java Virtual Machine must be set up and destroyed upon every execution. Java is a friendlier language that does not require the coder to pay as much attention to memory storage and allocation. Java handles memory allocation itself and because of this, causes the language to run slower as it is interpreting memory before executing. When executing the larger iterations, Java really struggled to run at optimal speeds as it needed to execute more at greater densities.

It is cleared that in terms of batch processing , the faster execution ranks Assembly, C++, Python, Java (best to worst).