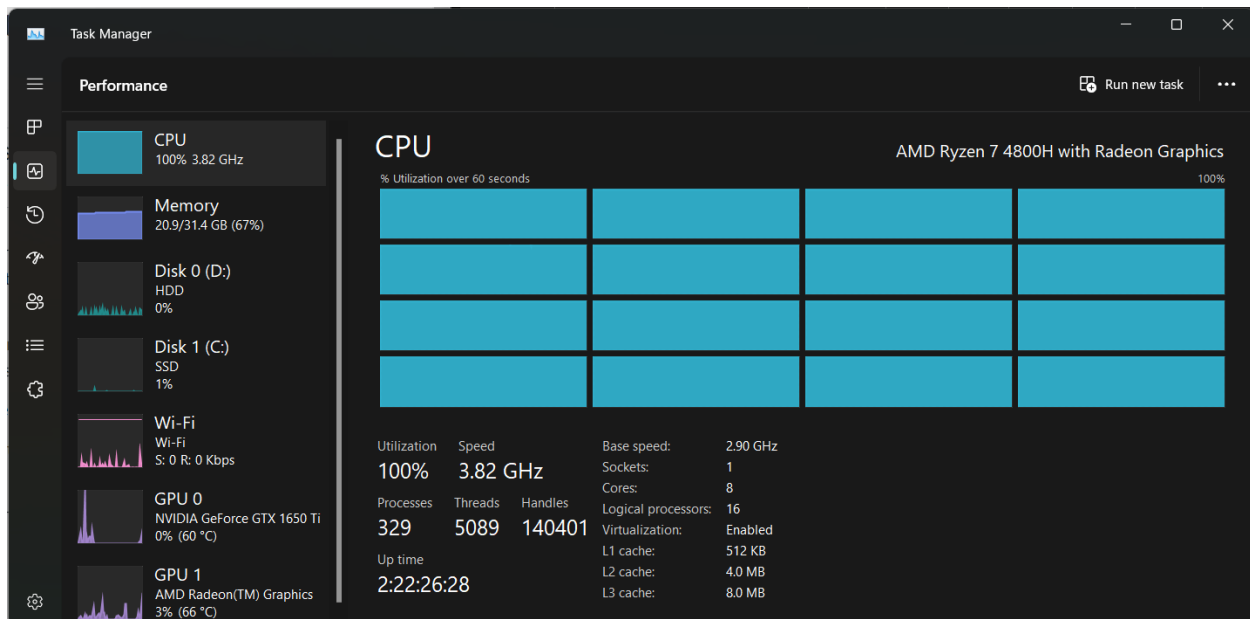


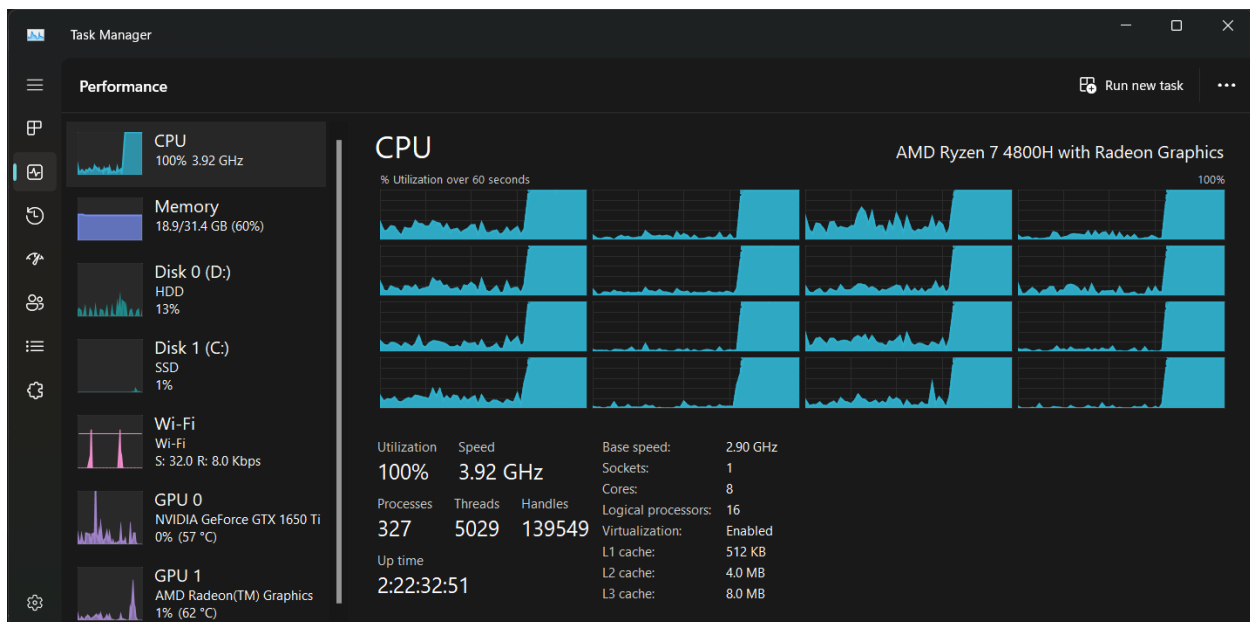
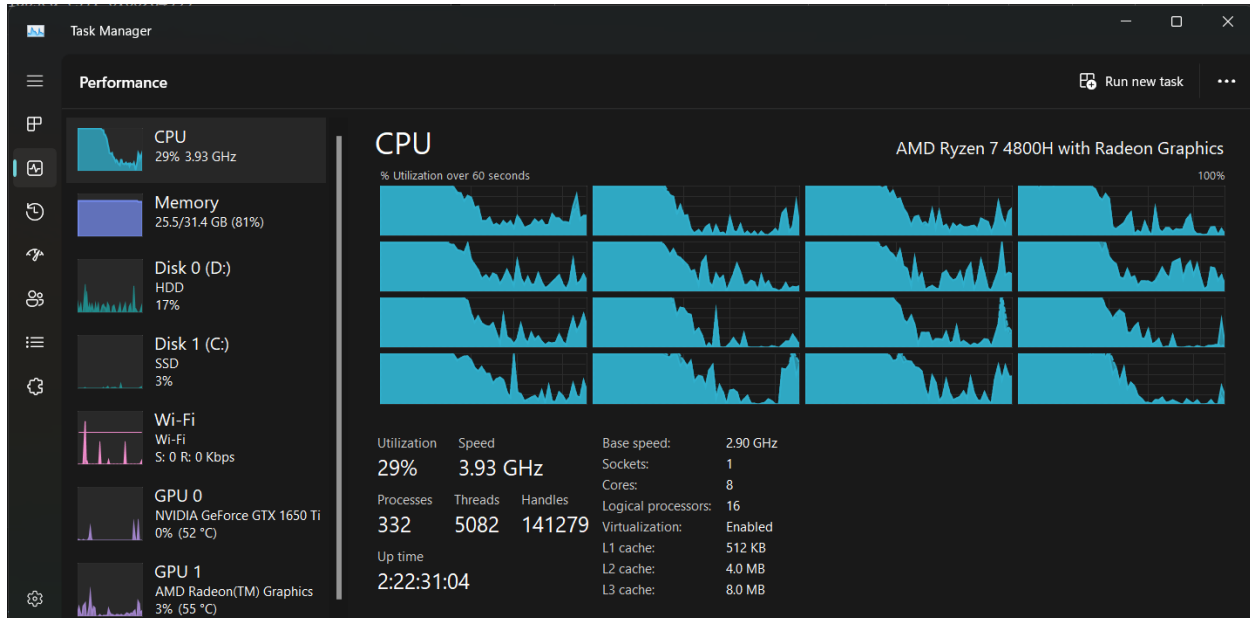
The performance metric I decided to use was the number of hashes I could calculate per second. The reason I chose this metric because of the nature of the problem I decided to try solving, namely that it is largely a brute-force problem, so concurrency optimizations would far outstrip most other optimizations I could think of.

I first tried some rather niche optimizations, where I turned the most common calculations into macros, figuring that the function overhead would be eliminated, and the results would compound and add up as more and more calculations were made. I did not run tests with this optimization with messages longer than three characters, because the number of guesses required to guess messages of increasing length would increase exponentially, and I was optimizing to run on my local machine only. To be honest, I was mostly just afraid of melting my computer, but I realize that my fears are somewhat unfounded, considering the other optimization I made.

I then decided to use threads to increase the number of concurrent calculations being made. I settled on using 16 threads, as my machine has 16 logical processors to it. I also figured that the thread initialization and resolving overhead would be minimal compared to the amount of work each thread would be doing. I figured the same would apply when adding up the number of hashes each thread individually calculated, because for the thousands (up to potentially millions) of guesses each thread would make, it would only need to lock and unlock a mutex one time and make a single addition to properly count the total number of hashes calculated across the threads.

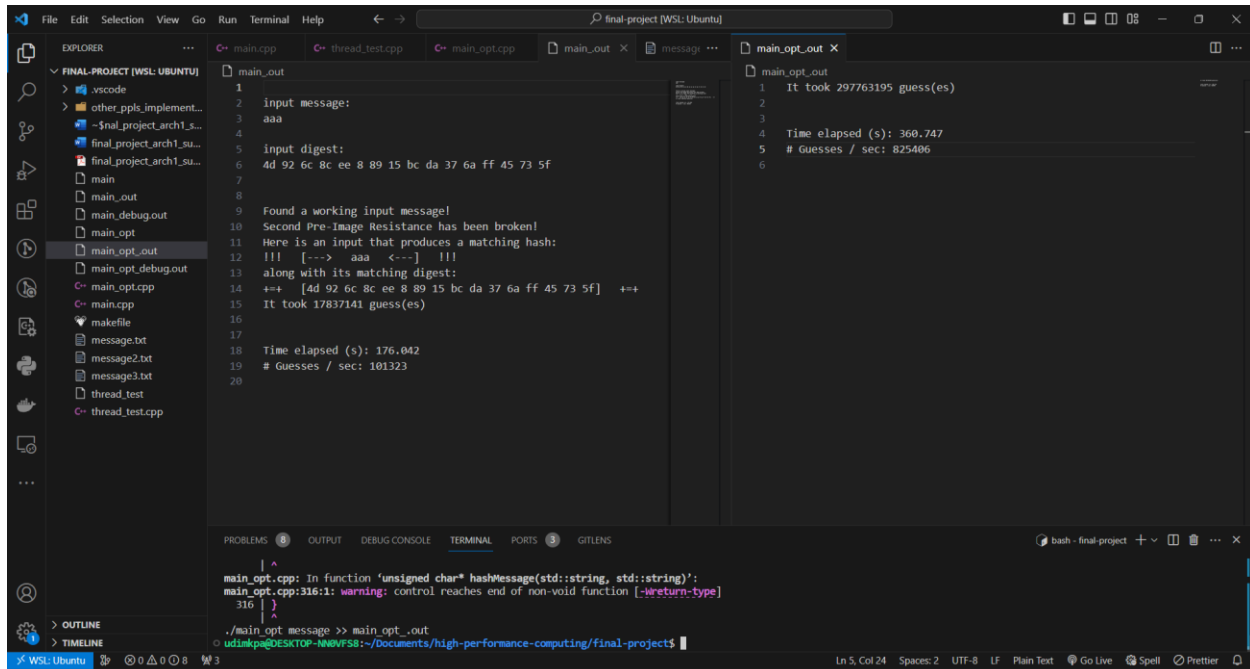
The results of such an optimization can be seen in some of my CPU resource utilization screenshots below.





As you can tell, my computer survived being maxed out across every single core. I was actually rather glad it did not begin overclocking, but rather remained at its maximum normal speed.

The calculations per second I get as a result of these optimizations went to over 800% that of my original maximum.



```
main_out
1  Input message:
2  aaa
3
4  Input digest:
5  4d 92 6c 8c ee 8 89 15 bc da 37 6a ff 45 73 5f
6
7
8
9  Found a working input message!
10 Second Pre-Image Resistance has been broken!
11 Here is an input that produces a matching hash:
12 !!! [---> aaa <---] !!!
13 along with its matching digest:
14 ++ [4d 92 6c 8c ee 8 89 15 bc da 37 6a ff 45 73 5f] ++
15 It took 17837141 guess(es)
16
17
18 Time elapsed (s): 176.042
19 # Guesses / sec: 101323
20
```

```
main_opt_out
1  It took 297763195 guess(es)
2
3
4  Time elapsed (s): 360.747
5  # Guesses / sec: 825406
6
```

```
main_opt.cpp: In function 'unsigned char* hashMessage(std::string, std::string)':
main_opt.cpp:316:1: warning: control reaches end of non-void function [-Wreturn-type]
316 | }
    | ^
```

```
./main_opt message >> main_opt_out
udinkpa@ESKTOP-100VFS8:~/Documents/high-performance-computing/final-project$
```

I attempted to do a little bit of digging into how to make the message generation faster by studying out whether strings in C++ were slow to work with. The information I came up with was a mixed bag, but I'm sure that was because I did not dig deep enough into the issue for long enough.

I'll stop here for now.