CS4028 Assessment Report

Task 1:

The idea of the solution to task 1 was to generate every possible string over the given alphabet, generating them in shortlex order. This was achieved by looping through the list of hashes that needed to be cracked, trying to break one at a time. Then for each hash the programme enters a while loop that runs until the hash password pair is found. Within this loop is a for loop, generating every string of the alphabet in shortlex order using itertools.product() and a map to do this efficiently, an idea that came from a post on StackOverflow [1] and was modified for my use. Then it generated the related hash in sha512 for this string. If there is a match, it breaks the for loop, changes the condition of the while loop as to finish the while loop, then moves onto the next hash needing to be brute forced, then repeats the process. When the algorithm completes it returns a list containing the cracked passwords in plaintext.

Task 2:

Task 2 was to design an algorithm implementing a dictionary attack. For this, the function took in not only the hashes needing cracked, but also the dictionary of common passwords. The function then ensures that the file pointer is at the beginning, before initiating a python dictionary for password hash pairs that have already been generated to be stored in. The function then loops through the hashes given to crack, first checking if a matching password has already been generated by checking the python dictionary. If so, the function saves the plaintext password to a list. If not, the function runs through each password in the passed in dictionary generating the hash for it, starting from wherever the pointer is left from previous runs, ensuring no repetition in hash generation. It then saves this password hash combo to the python dictionary. If the hash generated is the same as the one trying to be cracked, the plaintext is saved to the output and the function moves to the next hash to crack, if not it continues to search through the dictionary.

Task 3:

Task 3 was designed to do the same as task 2, however now with salted passwords. To do this, the code for task 2 was initially copied across. This function takes in a list of tuples, with the hash, and the plaintext salt used for this password. As well as this, it also has a password dictionary input as in task 2. In the function, the python dictionary was removed, and any code checking its' contents. This was done because each password has a unique salt, so storing hash-password pairs would be meaningless. The function now takes a hash salt pair from the input, separates the two strings, and appends the hash to each password in the dictionary before finding its hash. If the hash matches then it is appended to the list of found passwords and resets the file pointer to the start for the next hash, if not, it continues to parse the dictionary.

Task 4:

For task 4, it was decided to try modifying the previous 3 tasks' code as to get GPU implementations for each of them. This was done as it seemed an interesting prospect and opportunity to learn about this area further. After research, it was discovered that the python package Numba could make this process more straightforward.

It was theorised that a GPU implementation would have the greatest effect on task 1 as this was the task that involved the most computation and repetitive generation of characters/strings, thus benefitting from the GPU compilation and acceleration most.

CS4028 Assessment Report

Initially, a tutorial from GeeksForGeeks [2] was followed. However, this soon led to issues arising from the demonstrated plug and play method used in this tutorial. It was learnt that the @jit (just in time [compilation]) decorator that could be used from the Numba package had restrictions on datatypes and functions used, especially when the nopython=True argument was passed to it. These limitations included only taking limited built in data types (dtypes), as well as only a handful of built in functions as well as NumPy.

These restrictions are most apparent when using the argument “nopython=True”, which forces Numba to skip the python interpreter entirely, and try and compile all code for GPU. Without this argument, Numba will try and compile for GPU but fall back to the python interpreter where it is unable to. With this enabled it throws an error and does not allow execution or compilation to go any further.

A lot of time was spent trying to reengineer functions into an acceptable format for Numba’s nopython mode. This included writing a variation of itertools.product() with help from Stackoverflow (see appendix)[3]. Another function needing to be written was on to take the input of a text file password dictionary and generate a python list with each entry being a single password, stripped of its’ newline char. This was written to be used for Task 2 and 3 due to Numba clashing with string.strip().

Both of these rewrites were required as Numba does not support the use of itertools, or string stripping. As such, significant time was spent researching how to work around these restrictions. The rewriting was found to be the best option for this. After fixing these errors, it was discovered that Numba did not support Hashlib, the library used for hashing in this project. This was an issue that there was not enough time remaining to overcome, however research did show that it may have been possible by creating a wrapper for the Hashlib module and adding @jit decorator to it. [4]

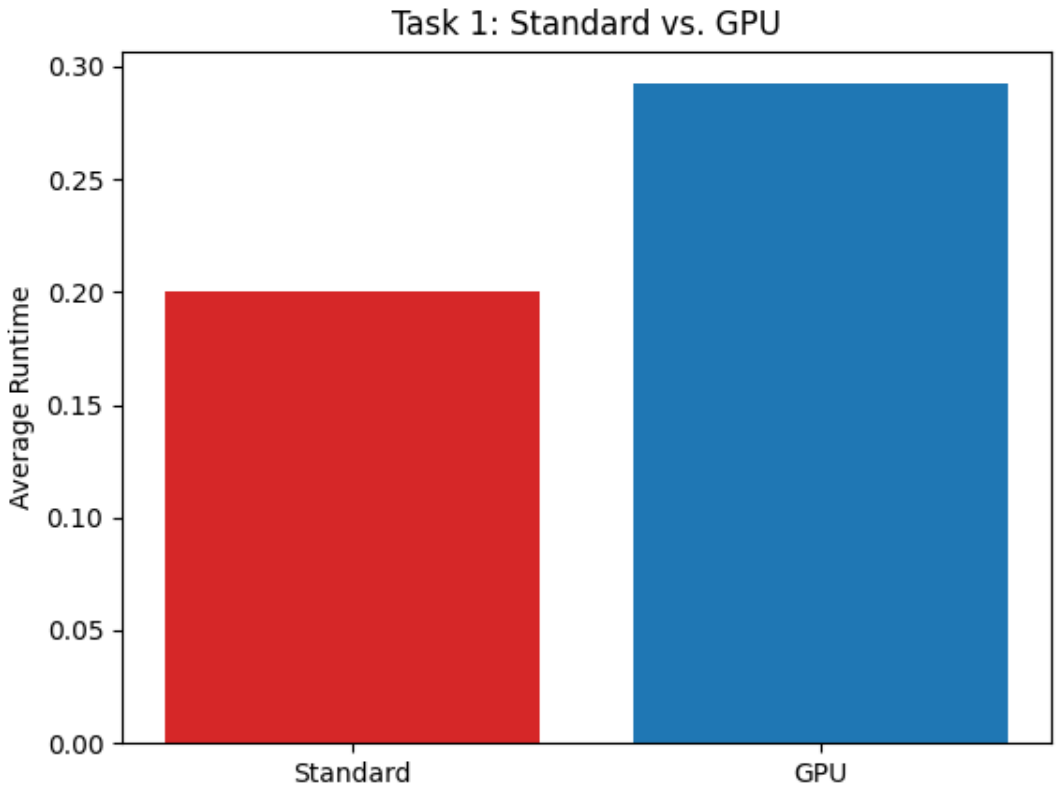
After these struggles the code was run with the @jit decorator but notpython was not set to True. This meant that it tries to compile for GPU but falls back to python interpreter for when it is not able to do this. The standard and GPU version of each function were then run 100 times to find the average run time, generating comparative bar charts for each (See Appendix). They showed that the GPU implementation was on average slower than the standard. However, most of this slow down came from the first run as that is when the Numba compiler tries to compile the code, causing an initial slowdown. (See “generated_data” folder for raw data.)

I learned a lot from trying to implement this with my code. First of all, I learned about Numba, and its usefulness, as well as its constraints. Another lesson I learned was to do deeper research into the topic earlier on, as to not spend so much time looking into the wrong solutions. I also learnt about generator functions and how to use them, and the yield key word in python when working to reengineer functions.

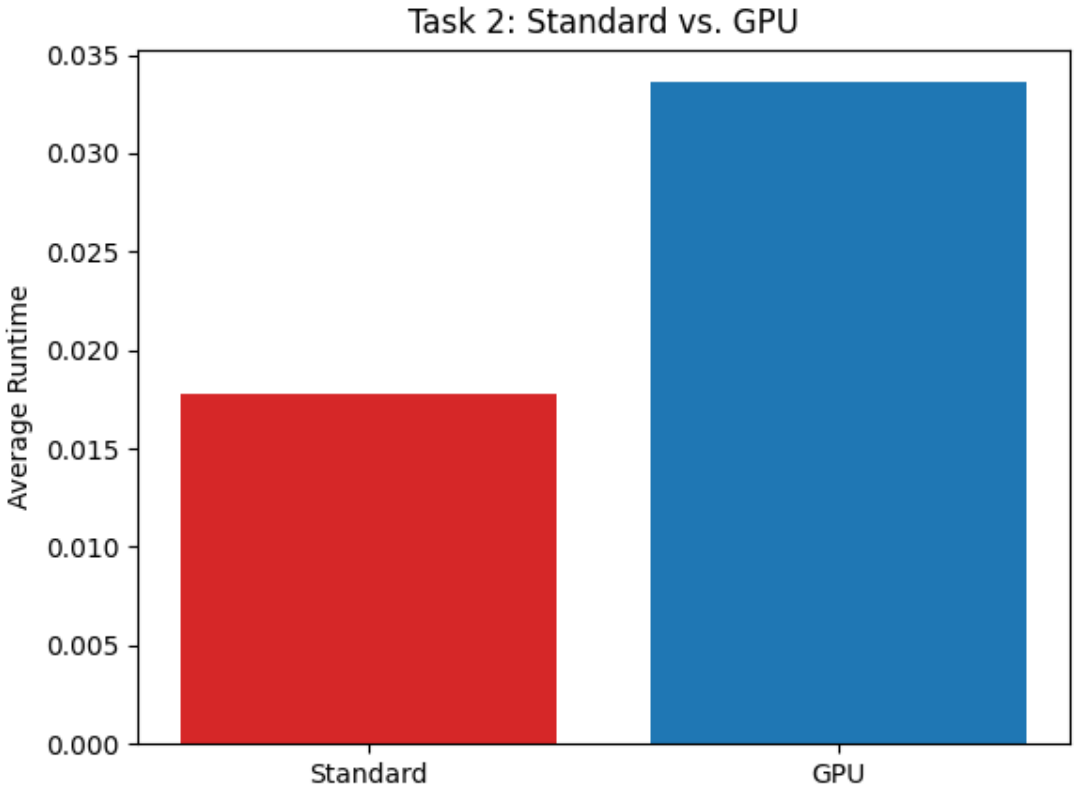
Task 4 was challenging but rewarding, although ultimately not completely successful. In future this is an area I plan to research into further to understand how this can be used to greater usefulness, and how to design code from scratch to work with Numba.

CS4028 Assessment Report

Appendix: [Figure 1]

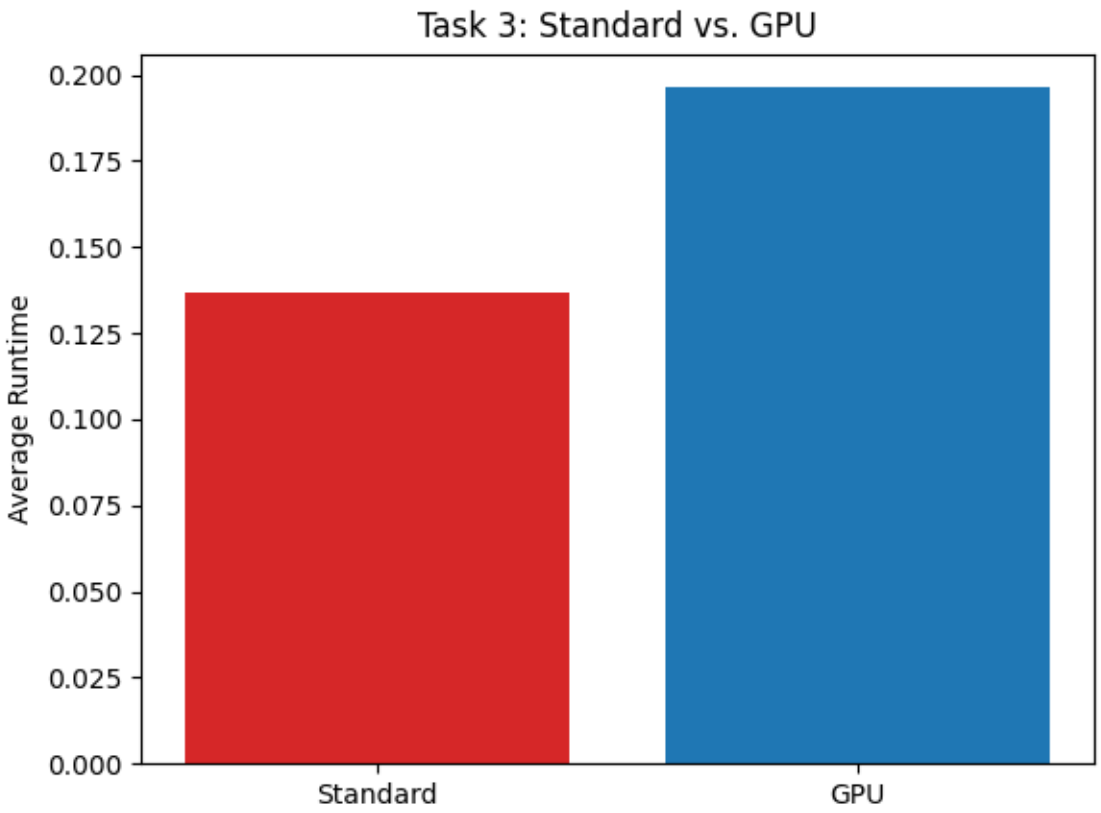


[Figure 2]



CS4028 Assessment Report

[Figure 3]



Task 1 Code:

```
def function_1(hash_list):
    start = time.time()
    alphabet = string.ascii_lowercase + string.digits
    answers = []
    for hash in hash_list:
        counter = 1          #Used for progressively increasing length of string
generated
        found = 0    # Create a variable for if item has been found to allow for
skipping of current run without breaking from entire loop
        while found != 1:    # While loop runs until password is found
            for string1 in map(''.join, itertools.product(alphabet, repeat=counter)):
# Use of itertools idea from: https://stackoverflow.com/questions/16347583/how-to-
generate-all-possible-strings-in-python
                if hashlib.sha512(bytes(string1, 'ascii')).hexdigest() == hash:    #
Computes hash of generated string, compares to given hash
                    answers.append(string1)
                    found = 1
                    break
            counter += 1

    print(answers)
    end = time.time()
    print(end - start)
    return answers
```

CS4028 Assessment Report

Task 2 Code:

```
# Task 2: Use given dictionary to generate hashes of common passwords and compare.
def function_2(passdict, hashes):
    passdict.seek(0)    # In case of prior access to file, reset pointer to start
    start = time.time()
    hashdict = {}   # Use Python Dictionary, keys: Hashes, contents: password
    answers = []
    for hash in hashes:
        if hash in hashdict:   # If hash is already computed from previous running,
then no need to compute
            answers.append(hashdict[hash])
        else:   # Strips newlines, calcs hash, adds to dict, and compares to see if it
matches given hash
            for password in passdict:
                password = password.strip()
                newhash = hashlib.sha512(bytes(password, 'ascii')).hexdigest()
                hashdict[newhash] = password

                if newhash == hash:
                    answers.append(password)
                    break

    print(answers)
    end = time.time()
    print(end-start)
    return answers
```

Task 3 Code:

```
# Task 3: To crack salted passwords given salt. Reused code from task 2, removing
hashdict
def function_3(passdict, hashes):
    start = time.time()
    answers = []
    for hash in hashes:
        passdict.seek(0)    # Resets pointer to start for each hash as no
precalculating because of unique salts
        for password in passdict:
            i_salt = password.strip()+hash[1]     # Strips newlines, adds salt
            newhash = hashlib.sha512(bytes(i_salt, 'ascii')).hexdigest()  # calcs hash

            if newhash == hash[0]:   # If hashes match, stores passwords, stripping
newlines
                answers.append(password.strip())
                break

    print(answers)
    end = time.time()
    print(end - start)
    return answers
```

CS4028 Assessment Report

Task 4 Code Attempts:

```
@jit(locals={'i' : str})
def product1(alphabet, repeat=1):       #modified version of code from:
https://stackoverflow.com/questions/61469388/how-can-i-replace-itertools-product-
without-itertools
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [alphabet] * repeat
    result = [[]]
    out = ""
    for pool in pools:
        #result = [x+[y] for x in result for y in pool]
        for x in result:
            for y in pool:
                temp = (x+[y])
                for i in temp:
                    out += i
                yield out
                out = ""
                result += [x+[y]]
```

```
def pass_list():
    passwordslocal = open("PasswordDictionary.txt", mode="r", encoding='ascii')
    output = []
    for i in passwordslocal:
        i = i.strip()
        output.append(i)
    return output


passwords_gpu = pass_list()
```

CS4028 Assessment Report

References:

[1]: *How to generate all possible strings in python?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/16347583/how-to-generate-all-possible-strings-in-python (Accessed: 22 October 2023).

[2]: *Running Python script on GPU.* (2023) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/running-python-script-on-gpu/ (Accessed: 22 October 2023).

[3]: *How can I replace itertools.product without itertools?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/61469388/how-can-i-replace-itertools-product-without-itertools (Accessed: 23 October 2023).

[4]: *Benchmark_of_the_SHA256_hash_function__Python_Cython_Numba*. Available at: https://perso.crans.org/besson/publis/notebooks/Benchmark_of_the_SHA256_hash_function__Python_Cython_Numba.html#The-SHA2_Numba-class (Accessed: 23 October 2023).