/*Start Header
*****************************************************************/
/*!
\file main.cpp
\author Yin shuyu, yin.s, 1802075
\co-author Luo Yu Xuan, yuxuan.luo, 1802205
\par yin.s\@digipen.edu
\co-par yuxuan.luo\@digipen.edu
\date Apr 19, 2021
\brief  CS398 Final Project

*/
/* End Header
*****************************************************************/

## Team Member:

1. Yin shuyu, yin.s@digipen.edu
2. Luo Yu Xuan, yuxuan.luo@digipen.edu

# Final Write Up :
# Project on MD5 (Message-Digest algorithm 5) message recovery tool

**SUMMARY:**

Computer systems store user passwords.
However, storing passwords in plaintext is never recommended, as an attacker could simply find the password file and thus have access to all passwords stored on the machine!
Hence, passwords are first encrypted using strong crypto hash functions.
Given a generated crypto hash, it is hard to crack the hash to generate the original plaintext. Therefore they are the natural solution to password storage.
Most tools to recover passwords from crypto hashes are sequential and CPU-based. As we have more complex passwords, these CPU-based solutions fail to recover passwords in a reasonable amount of time. You may make such a password recovery tool for the MD5 hash using CUDA to see if you could do any better.

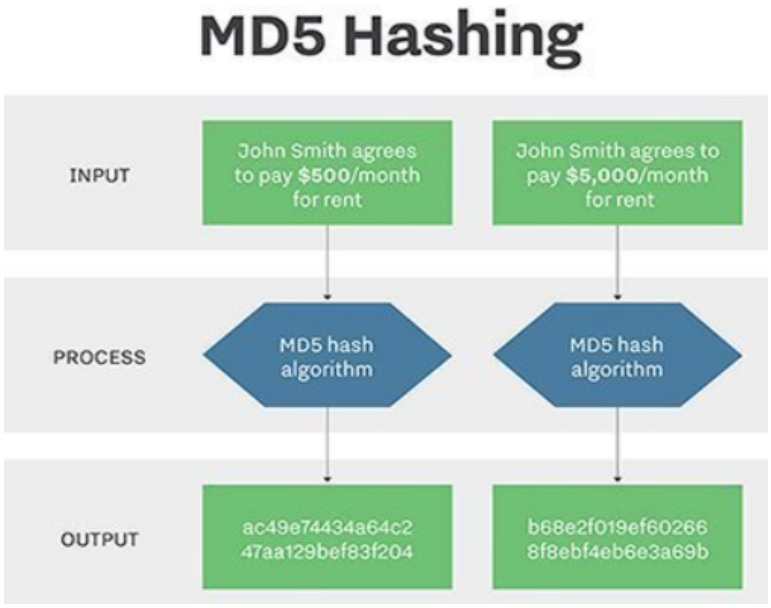
We implement an optimised password recovery tool for the MD5 hash using CUDA.
There are two sets of implementation on CPU and GPU.
(The Brute Force method and the Dictionary method.)
We compared the performance of these two sets of implementations.
Given the speed of our implementations, we demonstrate these effective GPU approaches.
As these CPU-based solutions fail to recover passwords in a reasonable amount of time.

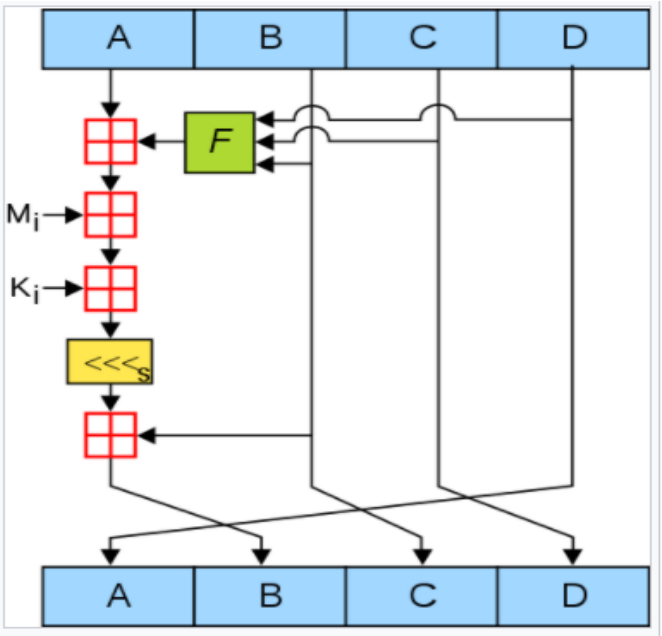
**BACKGROUND:**

**Application:**

Framework Build in Visual Studio 2019
Run on Window 10, x64 Release/Debug
CUDA device [GeForce GTX 1060 6GB] has 10 Multi-Processors, Compute 6.1

The program mainly focuses on Decoding MD5 hash.
However based on Research MD5 algorithm is a one way operation, not possible to reverse the algorithm and decode the hash.
Brute Force and Dictionary approach are the only 2 methods valid to decode MD5 hash. In these 2 algorithms/approaches, their main key operations are still MD5 hashing operations, as they are hashing messages and comparing them to the input hash from the user to find the password.



**For Brute Force algorithm:** **(pseudo code in the next few pages)**

Inputs:

- Input hash, 32 bytes hash string/char array
- Minimum length and Maximum Length, the message range to scan through for message combination
- Block Size and Tile Size, if you want do see the time in different setting

Output

- a decode string message, the message matching the hash or an empty string

It seems the algorithm is running through all the possible combinations in a sequential manner.
There are n^k possible combination strings of length k that can be formed from a set of n characters.
So the computation time increases exponentially as the length of the message increases.
It takes a very long time to compute through a large amount of combinations in a sequential manner.
So Cuda and run all combinations in parallelization to speed up the computation time.
The parallelization will show effect only for a large amount of combinations, if not just to initiate/prepare Cuda will be longer than CPU sequential execution.

And for my algorithm:

- shared data is not needed so data no need to be parallel.
- It depends on thread Index to compute the combination
- All is done locally in individual threads, only when the thread found the matching message it will update the result and inform it to stop further operations.

**CPU Brute Force pseudo code**

```
Input: H, 32 bytes hash and, MinLength and MaxLength, the message range
Output: a decode string message

for n <-- MinLength to MaxLength
    testMessge, length of n
    i = 0
    RecursiveLoop:
        if i == MinLength
            md5 hash testMessge into a 32 bytes hash, testHash
            if testHash is similar to input H, then
                return testMessge
        foreach ASCII hex in preDefine ASCII range do
            set testMessge[i] as ASCII hex
            Increment i
            Go to RecursiveLoop

return an empty string
```

**GPU Brute Force pseudo code**

```
Input: H, 32 bytes hash and, MinLength and MaxLength, the message range
Output: a decode string message

Cuda_kernel:
    Get tx, ty, bx, by
    col = tx + bx * blockDim.x
    rol = ty + by * blockDim.y
    index = rol * blockDim.x + col, the thread index in a single array

    // get the combination msg of this thread based on the thread index
    testMessge, length of n
    divisor = (preDefine ASCII range)^(n-1)
    for n <-- 0 to n-1
        testMessge[n] = ASCII hex at (index / divisor)
        Index %= divisor
        divisor /= n
    testMessge[n] = ASCII hex at index

    md5 hash testMessge into a 32 bytes hash, testHash
    if testHash is similar to input H, then
          return testMessge

GPU_BruteForce:
    for n <-- MinLength to MaxLength
        go to Cuda_kernel

    return an empty string
```

**For Dictionary  algorithm:**

Inputs:

- Input hash, 32 bytes hash string/char array
- Text file, the file that contains all the passwords/messages
- Block Size and Tile Size, if you want do see the time in different setting

Output

- a decode string message, the message matching the hash or an empty string

**CPU Sequential Dictionary pseudo code**

```
Input: H, 32 bytes hash and, file, dictionary text file
Output: a decode string message

load messages, String array, from file

foreach messge in messages
    md5 hash messge into a 32 bytes hash, testHash
    if testHash is similar to input H, then
        return testMessge

return an empty string
```

**GPU Threading Dictionary pseudo code**

```
Thread1:
Input:   msgMaxLgth, //max message length of each dictionary word
         dictionary_size, //size of dictionary
         dictionary_list, //host dictionary memory
         pinnedMemory_dictionary[3],  //transfer the host dictionary to the pinnedMemory
         numOfStreams, //number of streams
         uint tileSize// tile size
Output: transferred host to host pinned memory

For each tilesize to dictionary_size
    If not first loop
        Wait for semaphore[0][streamID];
        If resultFound
            return;
    Memcpy dictionary memory from the host to the pinned host memory;
    Signal semaphore[1][streamID];

Thread2:
Input:   msgMaxLgth, //max message length of each dictionary word
         dictionary_size, //size of dictionary
         device_hash, //User hash in device memory
         device_result[3],//password result in device memory
         device_dictionary[3], //dictionary data in device memory
         pinnedMemory_dictionary[3],  //stored dictionary in the host pinnedMemory
         final_result[3], //host memory for result
         Index, //index of the stream that found the passwordresult
```

```
        stream[3], //cuda streamIDs
        numOfStreams, //number of streams
        tileSize// tilesize
        Deviceresultfound //boolean for device - checks if result is found in the device
        resultFound //boolean for host- checks if result is found in the device
Output: password result of the Userhash

For each tilesize to dictionary_size
    Wait for semaphore[1][streamID];
    cudaMemcpyAsync from pinned dictionary memory to device dictionary memory in streamID;
    Call GPUScanDictionary Kernel;
    cudaMemcpy from Deviceresultfound to resultFound ;
    if  resultFound is true
        CudaMemcpy  from device_result to final_result
        for i = 0 to numOfStreams + 1
            Signal semaphore[1][i];
        return;
     Signal semaphore[0][streamID];
```

**GPU Kernel Dictionary Pseudo code**

```
Input:  const char* __restrict__ hash,  //constant memory as all threads only need to read this value
        char * list,
        unsigned listSize,
        char* result,
        unsigned msgMaxLgth,
        bool * resultfound
Output: result //password result

For every thread in each block
    If thread is within listSize
        Get Password from list;
        Hash the password;
        If passwordhash is same as Userhash
            result= password;
            resultfound= true;
```

Depending on the dictionary size, the sequential algorithm will spend a lot of time hashing each dictionary password one by one hashing is the key operation in each of the for loop.
Hence, this part of the sequential algorithm can be optimised by CUDA device where each thread hashes a dictionary password simultaneously,

For Cuda, Constant data is being utilised. The hashed user input is restricted to being constant as there is no need to modify the data and is being read by all threads in the grid.
There is no need for locality as 1 password in the dictionary is only accessed by 1 thread.

As the dictionary size is can be very huge, we have also decided to use pinned memory and task parallelism (multiple streams) to divide the task in batches of tiles of tile_size x tile_size

**APPROACH:**

**For Brute Force algorithm:** **(pseudo code in the previous few pages)**

Base CPU Brute Force attack reference code for MD5 hash:
https://github.com/rossmacarthur/md5-brute-forcer

We use this as an example and build our CPU Brute Force algorithm.
And for the GPU Brute Force algorithm, We just used parallelization technologies.
As the algorithm only requires one constant string hash, to compare all the algorithm generator hashes. So no need to transfer a lot of memory from the host to the device.

We split the combinations in algorithms to threads, so multiple combinations can be computed in parallelization. So each thread alone will have to compute a combination, by MD5 hashing it and comparing with the input hash.

The original serial algorithm is modified to fit for cuda.
The original serial algorithm makes use or recursive functionality to get each combination,
As for the cuda side we make use of the thread Index compute the combination.
A simple algorithm is created to map indexes to a combination.

As for the process of acquiring current optimization, we just try to convert the original serial algorithm into cuda. But make use of the threads to compute combinations, take us some time to create the new algorithm for each thread map to a unique combination.
The next optimization is to handle different message length. But our algorithm in cuda cannot map combinations for different length at the same time, so we get different message length combinations executed in different batches. And this avoids the problem.
Furthermore when there is a longer message length, we split the combinations in batches as well to fit into the tile size we allocation.
Lasly, we can stop the remaining batches of kernel execution when a matching message is found.

And this is the approach we did for GPU Brute Force algorithm.

**For Dictionary  algorithm:** **(pseudo code in the next previous pages)**

The technologies used are Cuda parallelism model, task parallelism and pinned host memory.

The original sequential serial algorithm is not changed for the Cuda kernel.
In the cuda kernel, each thread hashes a password and compares with user input hash. If it is the same, result is found and a boolean is modified to true to inform the CPU to stop the multi-threading process.

For big dictionaries, where memory cannot be allocated all at once into the device, task parallelism is utilised.For task parallelism in the CPU, the tasks are split into 2 threads to enable cuda kernel for big dictionaries where Pinned host memory is tilised for fast transfer from host to device.
The diagram below shows how the task are delegated among the 2 threads.

| Thread 1 | Transfer host memory to pinned host memory for stream 1 | Transfer host memory to pinned host memory for stream 2 | Transfer host memory to pinned host memory for stream 1 | Transfer host memory to pinned host memory for stream 2 | |
|---|---|---|---|---|---|
| Thread 2 | Wait for pinned host memory | Stream 1<br>- Transfer pinned host memory to device memory<br>- Kernel execution<br>- Transfer device memory to host memory | Stream 2<br>- Transfer pinned host memory to device memory<br>- Kernel execution<br>- Transfer device memory to host memory | Stream 1<br>- Transfer pinned host memory to device memory<br>- Kernel execution<br>- Transfer device memory to host memory | Stream 2<br>- Transfer pinned host memory to device memory<br>- Kernel execution<br>- Transfer device memory to host memory |

**RESULTS:**

## Results For Brute Force algorithm:

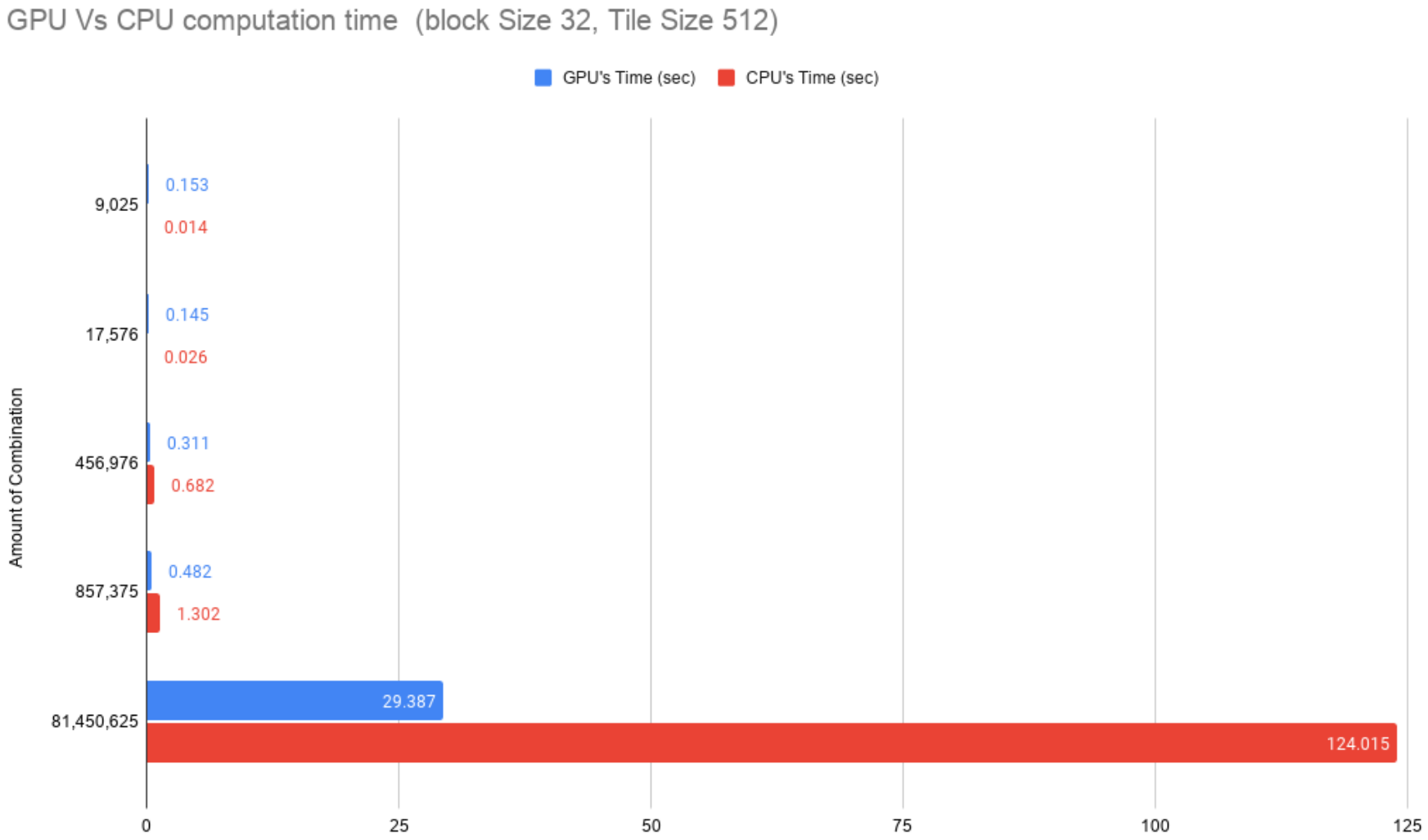
(***raw data result in BruteForceComputationResults.xlsx***)

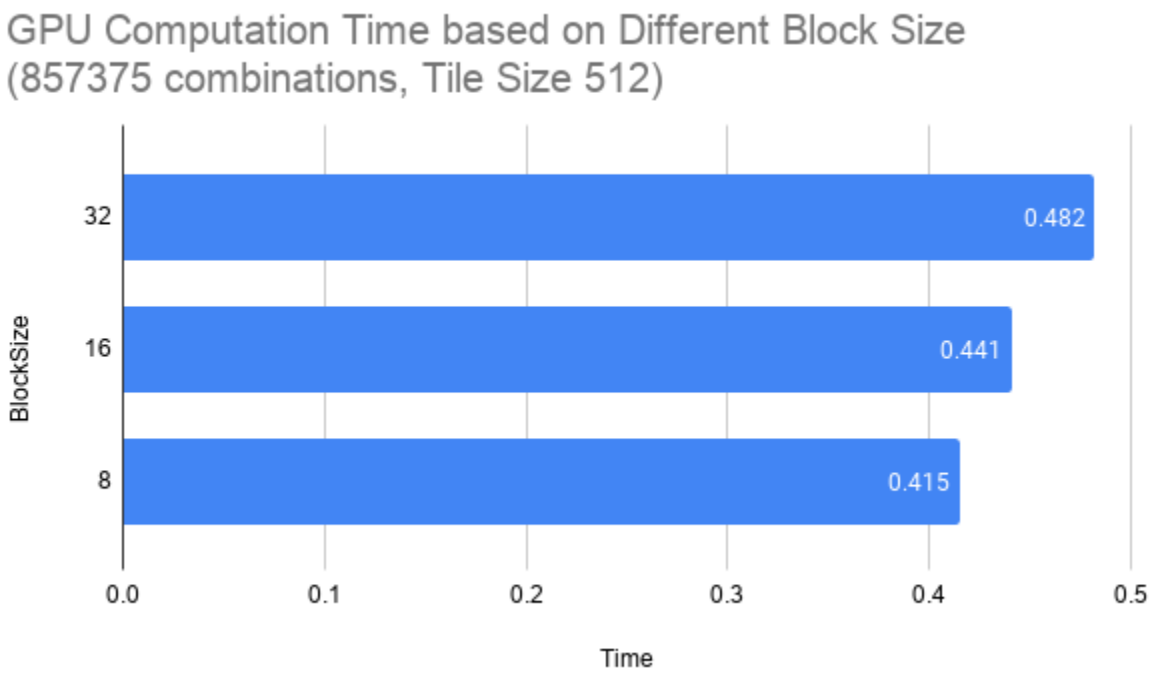
**Different GPU Brute Force Approach Timing**

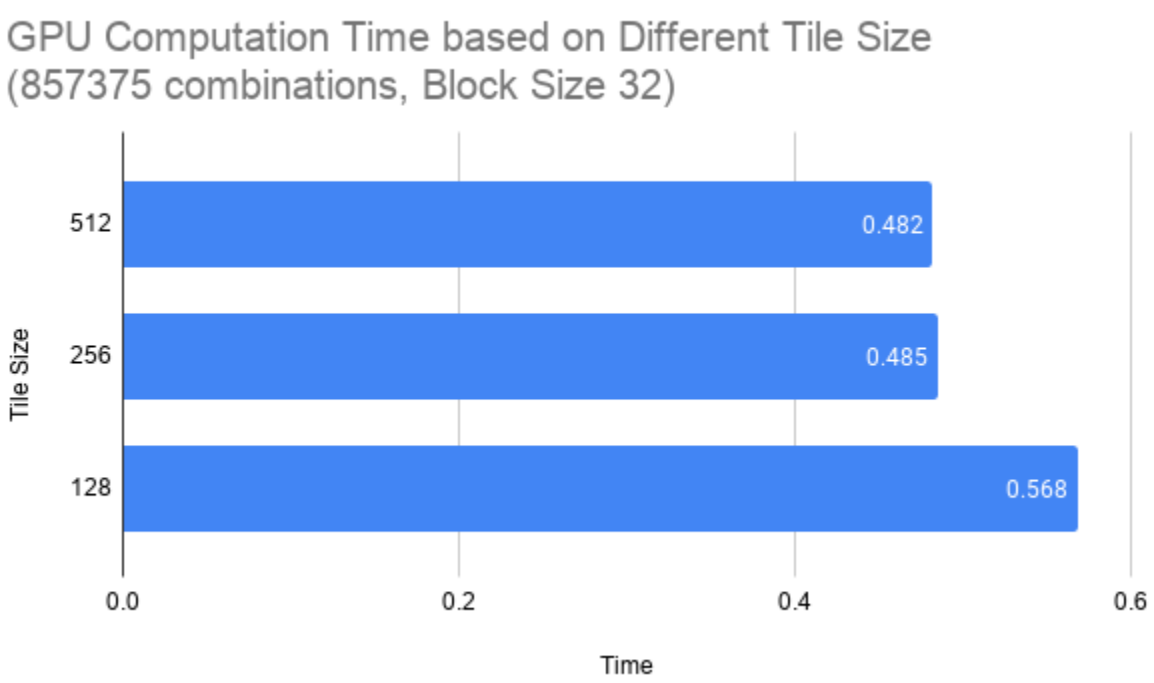
| ASCII Range | Message Length | Total Combinations | Block Size | Tile Size | Total Latency (sec) |
|---|---|---|---|---|---|
| 95 | 2 | 9,025 | 32 | 512 | 0.153 |
| 95 | 3 | 857,375 | 32 | 512 | 0.482 |
| 95 | 4 | 81,450,625 | 32 | 512 | 29.387 |
| | | | | | |
| 26 | 3 | 17,576 | 32 | 512 | 0.145 |
| 26 | 4 | 456,976 | 32 | 512 | 0.311 |
| | | | | | |
| 95 | 3 | 857,375 | 32 | 512 | 0.464 |
| 95 | 3 | 857,375 | 32 | 256 | 0.485 |
| 95 | 3 | 857,375 | 32 | 128 | 0.568 |
| | | | | | |
| 95 | 3 | 857,375 | 32 | 512 | 0.482 |
| 95 | 3 | 857,375 | 16 | 512 | 0.441 |
| 95 | 3 | 857,375 | 8 | 512 | 0.415 |

**Different CPU Brute Force Approach Timing**

| ASCII Range | Message Length | Total Combinations | Total Latency (sec) |
|---|---|---|---|
| 95 | 2 | 9,025 | 0.014 |
| 95 | 3 | 857375 | 1.302 |
| 95 | 4 | 81450625 | 124.015 |
| | | | |
| 26 | 3 | 17576 | 0.026 |
| 26 | 4 | 456976 | 0.682 |


GPU Vs CPU computation time (block Size 32, Tile Size 512)
GPU's Time (sec)
CPU's Time (sec)
Amount of Combination
9,025
0.153
0.014
17,576
0.145
0.026
456,976
0.311
0.682
857,375
0.482
1.302
81,450,625
29.387
124.015
0
25
50
75
100
125


GPU Computation Time based on Different Block Size
(857375 combinations, Tile Size 512)
BlockSize
32
16
8
0.482
0.441
0.415
0.0
0.1
0.2
0.3
0.4
0.5
Time


GPU Computation Time based on Different Tile Size
(857375 combinations, Block Size 32)
Tile Size
512
256
128
0.482
0.485
0.568
0.0
0.2
0.4
0.6
Time

As you can see from the Table/Graph above:

- For Performance between sequential CPU and Cuda GPU
  (Default block Size of 32, tile Size of 512)
  - For **9,025** and **17,576** message combinations:
    - There is no speedup show on the Cuda GPU side.
  - Starting from **456,976** message combinations:
    - A speedup of **220%** on the Cuda GPU (0.682/0.311)
  - For **857,375** message combinations:
    - A speedup of **270%** on the Cuda GPU (1.302/0.482)
  - For **81,450,625** message combinations:
    - A Speedup of **422%** on the Cuda GPU (124.015/29.387)

- As Performance between Different Tile size on Cuda GPU
  (Default block Size of 32, 857,375 message combinations)
  - For **128** vs **256** tile Size:
    - A speedup of **117%** was shown (0.568/0.485)
  - For **256** vs **512** tile Size:
    - A speedup of **100.6%** was shown (0.485/0.482)

- As Performance between Different Block size on Cuda GPU
  (Default tile Size of 512, 857,375 message combinations)
  - For **8** vs **16** and **16** vs **32** block Size:
    - A slow down is shown in both cases.

Observations based on the table results are Cuda GPU will only just to improve the performance when a large amount of different message combinations need to be executed (around 200 000, i think). So if cracking/decoding a small message and message ASCII range, It is better to use a sequential CPU.

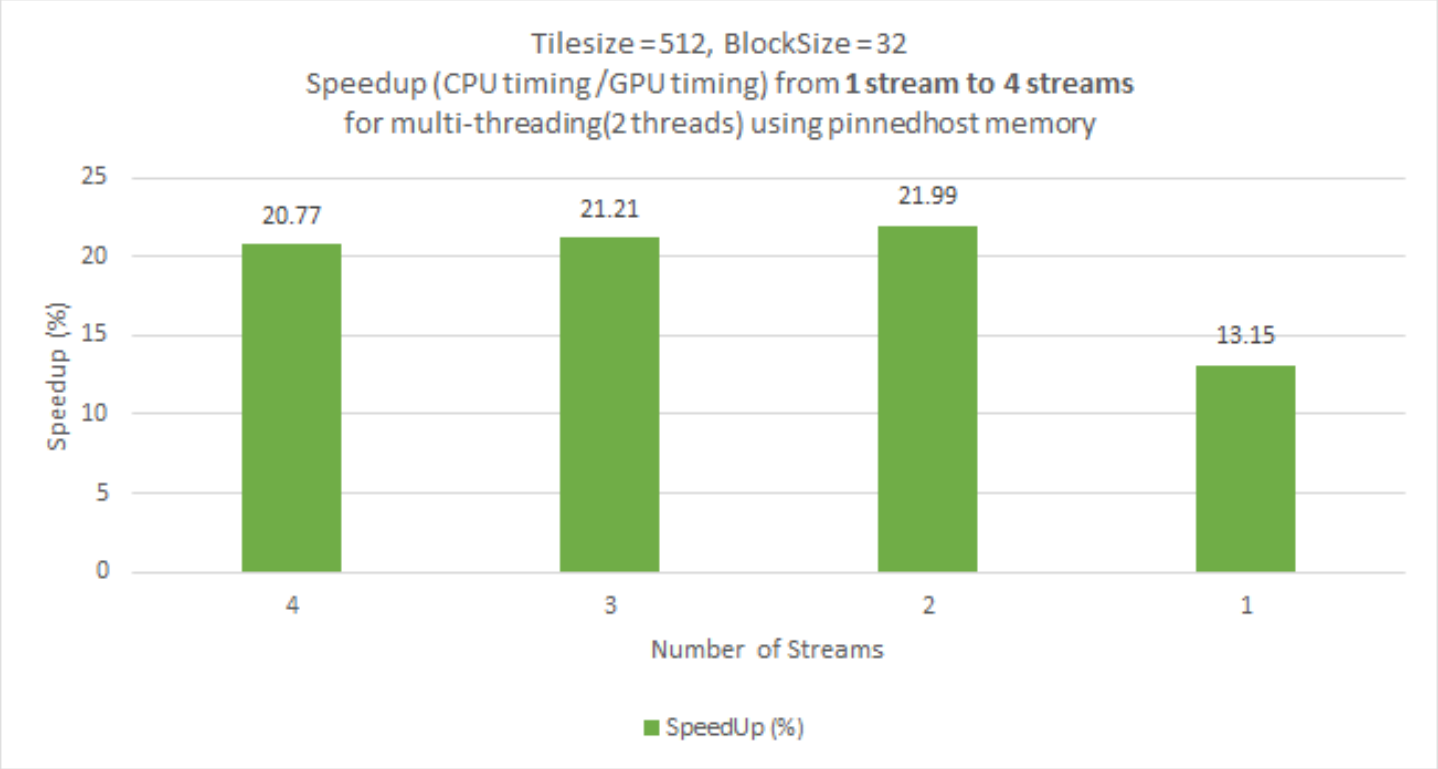
As for different tile sizes on Cuda GPU, it seems that tile size does not significantly improve the performance, It seems to hit a Limit at 256 tile size.

And for different block sizes on Cuda GPU, it seems that block size does not show any improved performance. Yet it showed a slight deterioration on the performance.

**Summary for Brute Force algorithm on CPU and Cuda GPU:**

- The more combinations needed to execute the better the performance on Cuda GPU than CPU.
- But only when the number of combinations are over 456,976(maybe lower), otherwise CPU has a better performance on small amount of combinations.
- 256 Tile Size can considerat the optimal tile size, as tile size doesn't affect significantly on the performance,
- Block size doesn't affect the performance significantly, however it showed a slight deterioration as the block size increased.

**Results For Dictionary  algorithm:**

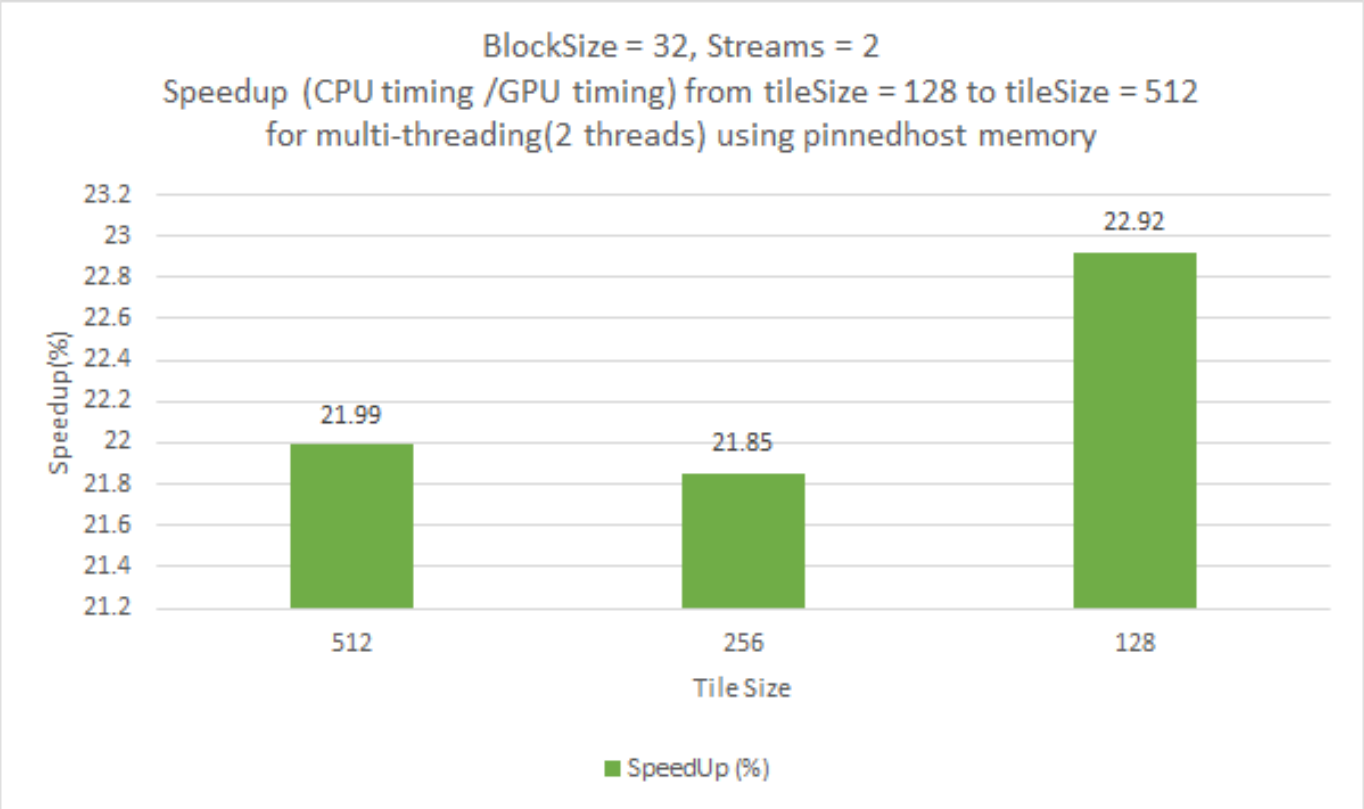



| Total Messages | Block Size | Tile Size | Streams | Memory Allocation Time (sec) | Algorithm Execution Time (sec) | GPU's Total Latency (sec) | CPU's Total Latency (sec) | SpeedUp (%) |
|---|---|---|---|---|---|---|---|---|
| 14,341,564 | 32 | 512 | 4 | 0.21999 | 0.81957 | 1.0396 | 21.594 | 20.77 |
| 14,341,564 | 32 | 512 | 3 | 0.15971 | 0.85852 | 1.0182 | 21.594 | 21.21 |
| 14,341,564 | 32 | 512 | 2 | 0.17392 | 0.80816 | 0.9821 | 21.594 | 21.99 |
| 14,341,564 | 32 | 512 | 1 | 0.15964 | 1.48197 | 1.6416 | 21.594 | 13.15 |

The timing results of the GPU method was recorded separately with the allocation time which includes cudaMalloc, cudaHostAlloc and the thread execution timing which include the execution of the kernel, cudaMemCpyAsync and memcpy(transferring of host memory to pinned host memory).
The size of the inputs were based on a dictionary downloaded from online which has 14,341,564 passwords in total.
For the dictionary multithreading and cuda parallelism model, we were able to speedup the CPU sequential method up to about 13X if using 1 stream and up to the limit of about 21X for 2 or more streams.

The limit for only being able to optimise up to 2 streams is due to the task only being able to be split up to 2 threads with transferring of host memory to pinned host memory for one thread and the transfer of pinned host memory to device memory and vice versa and the execution of the kernel in another thread. Hence, the increase in more than 2 streams does not allow for more concurrency to happen in the CPU.

| Total Messages | Block Size | Streams | Tile Size | Memory Allocation Time (sec) | Algorithm Execution Time (sec) | GPU's Total Latency (sec) | CPU's Total Latency (sec) | SpeedUp (%) |
|---|---|---|---|---|---|---|---|---|
| 14,341,564 | 32 | 2 | 512 | 0.17392 | 0.80816 | 0.9821 | 21.594 | 21.99 |
| 14,341,564 | 32 | 2 | 256 | 0.12686 | 0.86153 | 0.9884 | 21.594 | 21.85 |
| 14,341,564 | 32 | 2 | 128 | 0.11764 | 0.82469 | 0.9423 | 21.594 | 22.92 |




The change in tileSize does not seem to affect the speedup of the algorithm.
From tileSize of 128 to 512, ranges from speedup of 21X to 23X.

**REFERENCES:**

Please provide a list of references used in the project if any.


Research On MD5 encoding & Cracking:

https://www.youtube.com/watch?v=53O9J2J5i14
http://cs.indstate.edu/~fsagar/doc/paper.pdf
https://crackstation.net/hashing-security.htm#attacks

Frameworks resources that we reference from:

https://github.com/CommanderBubble/MD5 (encoding)
https://github.com/Ex094/HashCrackerV.2 (Dictionary Decoding)
https://github.com/rossmacarthur/md5-brute-forcer (brute force Decoding)

Dictionary Resources:

https://www.kaggle.com/wjburns/common-password-list-rockyoutxt


**LIST OF WORK BY EACH STUDENT:**

If your project is a team project, please list the work performed by each partner.

| Tasks: | What's Done: | Contribution: |
|---|---|---|
| Research of MD5 hash encoding algorithm | Google on what's MD5 and how its implemented, and search for sample frameworks | Shuyu, yu xuan |
| Research of MD5 hash decoding algorithm | Google on ways to decode MD5 hash, and search for sample frameworks | Shuyu, yu xuan |
| Project Framework | Use assignment 1 framework as base, and added text interface to menu | Shuyu :p |
| MD5 hash encoding algorithm | We sample the research hash algorithm implemented it into our framework, applied cuda into it later on | Shuyu, yu xuan |
| CPU MD5 hash decoding algorithm<br>● Brute Force<br>● Dictionary | Understanding the the 2 decoding algorithm from research, we implemented it in. | Shuyu, yu xuan |
| Optimization of the MD5 hash Dictionary algorithm using CUDA | | yu xuan |
| Optimization of the MD5 hash Brute Force algorithm using CUDA | I parallelize the combinations that get for Brute Force algorithm into the cuda threads, performance will be improve if the message is very long | Shuyu |
| Final Write Up Report | Write up report, graph, and pseudo code | Shuyu, yu xuan |