UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

# Part -3: Password Cracking with Files using CUDA


| | |
|---|---|
| Student Id | : 2407710 |
| Student Name | : Dhadkan K.C. |
| Group | : L6CG4 |
| Lecturer | : Mr.Yamu Poudel |

# Table of Contents

Table of Figures

## Overview

This work executes a full password cracking pipeline based on a CPU and a CUDA. The encryption passwords are produced and hashed on the CPU and parallel hash matching and all possible encrypted passwords are generated by the CUDA. The aim is to decode the initial 4-character passwords (aa00-zz99) out of a file of hashed passwords encrypted. Its implementation is correct in file operations, using dynamic blocks and threads to execute CUDA kernel operations, in host-device memory transfers, and memory deallocation.

## Header Files Used

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <crypt.h>
#include <unistd.h>
#include <cuda_runtime.h>
```

*Figure 1 Header file*

- stdio.h – File and console input/output
- stdlib.h – Dynamic memory allocation
- string.h – String operations
- time.h – Random number seeding
- crypt.h – SHA-512 hashing
- cuda_runtime.h – CUDA runtime support

# Password Cracking Implementation

## Password Generation

### Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char wrap_letter(char c) {
    return 'a' + (c - 'a' + 26) % 26;
}

char wrap_digit(char c) {
    return '0' + (c - '0' + 10) % 10;
}

char* cudaCrypt(char* rawPassword) {
    static char newPassword[11];

    newPassword[0] = rawPassword[0] + 2;
    newPassword[1] = rawPassword[0] - 2;
    newPassword[2] = rawPassword[0] + 1;
    newPassword[3] = rawPassword[1] + 3;
    newPassword[4] = rawPassword[1] - 3;
    newPassword[5] = rawPassword[1] - 1;
    newPassword[6] = rawPassword[2] + 2;
    newPassword[7] = rawPassword[2] - 2;
    newPassword[8] = rawPassword[3] + 4;
    newPassword[9] = rawPassword[3] - 4;
    newPassword[10] = '\0';

    for (int i = 0; i < 10; i++) {
        if (i < 6)
            newPassword[i] = wrap_letter(newPassword[i]);
        else
            newPassword[i] = wrap_digit(newPassword[i]);
    }
    return newPassword;
}

int main(int argc, char* argv[]) {

    if (argc != 2) {
        printf("Usage: %s <number_of_passwords>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    if (n < 10000) {
        printf("Password count must be at least 10000\n");
        return 1;
    }

    FILE* fp = fopen("passwords.txt", "w");
    if (!fp) {
        printf("Error opening output file\n");
        return 1;
    }

    srand((unsigned)time(NULL));

    char raw[5];
    raw[4] = '\0';

    for (int i = 0; i < n; i++) {
        raw[0] = 'a' + rand() % 26;
        raw[1] = 'a' + rand() % 26;
        raw[2] = '0' + rand() % 10;
        raw[3] = '0' + rand() % 10;
        fprintf(fp, "%s\n", cudaCrypt(raw));
    }

    fclose(fp);
    return 0;
}
```

*Figure 2 PasswordGeneratorToText.c*

At this stage, the random raw passwords are created in the CPU. The passwords are in a standard format of two lower case letters and then two numbers, where the first one starts with aa00 and the last is zz99. This restricted format will have a limited password space that is vulnerable to brute force cracking.

Every generated raw password is encrypted by means of the offered cudaCrypt () function. The amount of passwords to be generated is entered by the user in the command line, and the amount must be at least 10,000 passwords as indicated in the task specification. The coded passwords are then written in a text file in line by line to be processed further.

```
int main(int argc, char* argv[]) {

    if (argc != 2) {
        printf("Usage: %s <number_of_passwords>\n", argv[0]);
        return 1;
    }
```

it ensures that the user provides the number of passwords to generate. It allows the dataset size to be controlled dynamically and enforces correct program usage.

```
for (int i = 0; i < n; i++) {
    raw[0] = 'a' + rand() % 26;
    raw[1] = 'a' + rand() % 26;
    raw[2] = '0' + rand() % 10;
    raw[3] = '0' + rand() % 10;
    fprintf(fp, "%s\n", cudaCrypt(raw));
}
```

It generates raw passwords using valid ASCII ranges. The first two characters are lowercase letters, and the last two characters are digits, forming passwords in the required format.

### Compile and run

Compile:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ gcc PasswordGeneratorToText.c  -o PasswordGeneratorToText
```

Run:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ ./PasswordGeneratorToText 10000
```

### Output:

Filename: Passwords.txt

```
rnqysu9579
zvypjl0657
tpswqs5157
tpsmgi0657
zvynhj1735
tpshbd3902
eadhbd2891
wsvmgi4080
iehgac1713
rnqxrt3991
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$
```

*Figure 3 passowrd.txt*

## SHA-512 Hashing

### Source Code

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ cat EncryptSHA512.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crypt.h>
#include <unistd.h>

#define SALT "$6$AS$"
#define MAX_LEN 256

int main(int argc, char *argv[]) {

    if (argc != 3) {
        printf("Usage: %s <input_file> <output_file>\n", argv[0]);
        return 1;
    }

    FILE *in = fopen(argv[1], "r");
    FILE *out = fopen(argv[2], "w");

    if (!in || !out) {
        printf("Error opening files\n");
        return 1;
    }

    char line[MAX_LEN];

    while (fgets(line, MAX_LEN, in)) {
        line[strcspn(line, "\n")] = '\0';
        char *hash = crypt(line, SALT);
        if (hash)
            fprintf(out, "%s\n", hash);
    }

    fclose(in);
    fclose(out);
    return 0;
}
```

*Figure 4 EncryptSHA512.c*

Here, cryptographic hashing is done on the encrypted passwords formed in Step 1. The program deciphers encrypted passwords in an input file known through command line arguments. All the

passwords are hashed by the crypt() function using the fixed salt and hashing algorithm SHA-512.

The resultant hashes are appended in a new output file, where the output has one hash per line. This file is the password dataset hashed which will subsequently be broken with the help of CUDA.

```
FILE *in = fopen(argv[1], "r");
FILE *out = fopen(argv[2], "w");
```

This block opens the input and output files passed through the command line, allowing the same hashing program to be reused for multiple steps.

```
while (fgets(line, MAX_LEN, in)) {
    line[strcspn(line, "\n")] = '\0';
    char *hash = crypt(line, SALT);
    if (hash)
        fprintf(out, "%s\n", hash);
}
```

The crypt() function applies SHA-512 hashing using a fixed salt. This produces a secure, fixed-length hash for each encrypted password.

## Compile and run

Compile:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ gcc EncryptSHA512.c -o EncryptSHA512 -lcrypt
```

Run:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ ./EncryptSHA512 passwords.txt file_hashes.txt
```

## Output:

Filename: file_hashes.txt

```
$6$AS$lzLoIu6AI0siX9qaIoS6R9hVokkDl6NxwQYYdBuS9AmaFpoq2VHujQo3futAxNOLVhdQEyOOgvehL2G3HhSIr/
$6$AS$3ATU9JSTOovZG.pAYAtjCzTcqMw2bWJI5.jwH/IFBr1nwqZ4xeC0cY3irK0.DmhRQwr9/sVi8S.JzK9ClfjZf0
$6$AS$TaftDQE89JuuO9edXxNEMHISSJI6.nDjWwK9jwWxBFFK6NLbvjqmWLzsbsn3B67ergoyWKiJ/Ur8K3g73DjZu0
$6$AS$h680fXAk0y0IFfhX6ATL88HTeXFy2oRMpXgWzTvY9zsNrZE33PDQAZrVzNKGmMrsuxVXD1bGAG9tDcMP0K9U2.
$6$AS$kTTq5SJd8SxhZopfFGNnWUOjyEMR2eiJks3kwKAWmLAI4VbdawmoqXHpfVsSzORcbm8w4dkN6GAxa8JIbj5T01
$6$AS$Om6k0gms0DwXMwcsg2eZ2xA85CnOjgLbgng8jDteosfXz80txGiCmnNYgC5lU4dys.tY15G9boS0gzr4Yfjys.
$6$AS$/7W65t5pJ7cPpeZbcORhKWELH6N4irPZnoK1SsuVFKNpjjBpAA8HRt5zjvHppJTQXSSzLTc/k6Itu5TXDsW.q1
$6$AS$Ci3PQvRtbPOV2brTFf9.6fsPbVFzKyDj4Ii/J/Kp0BB27xWB3Zw8AxivFLmmETACoQOJxVptEd7aNgXzDdhwp0
$6$AS$rO7KOBLjK7PgsnMFyyRkC3f7a1A0G46axXYDwkWwrntEgLRs9k3x2zdJOMQqKEwtAzbscH1btSEFdKRCMob7h/
$6$AS$aXRwlQEcO444KaQzAtg7In8EJ9qEghoUcue06Mp2vmKKVWRC2MYp.rQ2OkYyR0/TXxkNWgwn8iYVUz4cdd9Dk.
$6$AS$v4emVSu4NJHnky3NbePfgozOviLN0MsnzTwiDfN/EABxmcWxAkotrQzQr.DU88FMMg9joO3Qzk/CAXTcTLgrj0
$6$AS$4WhJCduGh5VBHUb/wv3TBUB3W75FccoSfmarYXvl1LY6LkGeeJb68pZhN8cNrVset.SkkyPyogfQfTaw17Y../
$6$AS$oxQsYMzAVSF0l721jNfzfdSa6eFSt.YYbe9PHrx0VuOu2AxlEC9BdDRk/2E38038/s.2wFQGxdhDNUTYAqn1p/
$6$AS$HRQHNjBKwMycdXSl3A7fDpJ0IcNmDSUBHqxjwo.k4xM5MZwHQHTs.fUyNpyJokfsNEDmhtUGXjfQONh0uSaaY.
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$
```

*Figure 5 file_hashes.txt*

# CUDA (GPU) Password Generation

## Source Code

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ cat CryptForCuda.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define TOTAL 67600
#define PASS_LEN 11

__device__ char wrap_letter(char c) {
    return 'a' + (c - 'a' + 26) % 26;
}

__device__ char wrap_digit(char c) {
    return '0' + (c - '0' + 10) % 10;
}

__device__ void cudaCrypt(char* raw, char* out) {

    out[0] = raw[0] + 2;
    out[1] = raw[0] - 2;
    out[2] = raw[0] + 1;
    out[3] = raw[1] + 3;
    out[4] = raw[1] - 3;
    out[5] = raw[1] - 1;
    out[6] = raw[2] + 2;
    out[7] = raw[2] - 2;
    out[8] = raw[3] + 4;
    out[9] = raw[3] - 4;
    out[10] = '\0';

    for (int i = 0; i < 10; i++) {
        if (i < 6)
            out[i] = wrap_letter(out[i]);
        else
            out[i] = wrap_digit(out[i]);
    }
}

__global__ void generate(char* d_out) {

    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id >= TOTAL) return;

    char raw[4];
    raw[0] = 'a' + (id / 2600);
    raw[1] = 'a' + (id / 100) % 26;
    raw[2] = '0' + (id / 10) % 10;
    raw[3] = '0' + (id % 10);

    cudaCrypt(raw, &d_out[id * PASS_LEN]);
}

int main() {

    char *h_out = (char*)malloc(TOTAL * PASS_LEN);
    char *d_out = NULL;

    if (!h_out) return 1;
    if (cudaMalloc((void**)&d_out, TOTAL * PASS_LEN) != cudaSuccess) return 1;

    int threads = 256;
    int blocks = (TOTAL + threads - 1) / threads;

    generate<<<blocks, threads>>>(d_out);
    if (cudaDeviceSynchronize() != cudaSuccess) return 1;

    if (cudaMemcpy(h_out, d_out, TOTAL * PASS_LEN,
                   cudaMemcpyDeviceToHost) != cudaSuccess)
        return 1;

    FILE* fp = fopen("Cuda_passwords.txt", "w");
    if (!fp) return 1;

    for (int i = 0; i < TOTAL; i++)
        fprintf(fp, "%s\n", &h_out[i * PASS_LEN]);

    fclose(fp);
    cudaFree(d_out);
    free(h_out);

    return 0;
}
```

*Figure 6 CryptForCuda.cu*

This step involves changing the cudaCrypt function written in this case by a CPU-based cudaCrypt into a CUDA device function meaning this can be executed on the GPU. A CUDA kernel is created to generate all combinations of raw passwords between aa00 and zz99 which give rise to 67 600 unique combinations. One password is encrypted by each CUDA thread. On

the x-dimension, several blocks and threads are employed to make sure the workload is spread effectively on the GPU. The passwords that are encrypted in the GPU are sent back to the host and stored in an output file.

```
__device__ void cudaCrypt(char* raw, char* out) {

    out[0] = raw[0] + 2;
    out[1] = raw[0] - 2;
    out[2] = raw[0] + 1;
    out[3] = raw[1] + 3;
    out[4] = raw[1] - 3;
    out[5] = raw[1] - 1;
    out[6] = raw[2] + 2;
    out[7] = raw[2] - 2;
    out[8] = raw[3] + 4;
    out[9] = raw[3] - 4;
    out[10] = '\0';

    for (int i = 0; i < 10; i++) {
        if (i < 6)
            out[i] = wrap_letter(out[i]);
        else
            out[i] = wrap_digit(out[i]);
    }
}
```

It is the GPU analog of the CPU encryption functionality. It provides the same logic in order to make sure that encrypted passwords are created on the GPU are the same as those created on the CPU.

```
__global__ void generate(char* d_out) {

    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id >= TOTAL) return;

    char raw[4];
    raw[0] = 'a' + (id / 2600);
    raw[1] = 'a' + (id / 100) % 26;
    raw[2] = '0' + (id / 10) % 10;
    raw[3] = '0' + (id % 10);

    cudaCrypt(raw, &d_out[id * PASS_LEN]);
}
```

Each CUDA thread computes a unique password combination based on its thread ID. Bounds checking ensures that threads do not access invalid memory.

```
generate<<<blocks, threads>>>(d_out);
if (cudaDeviceSynchronize() != cudaSuccess) return 1;

if (cudaMemcpy(h_out, d_out, TOTAL * PASS_LEN,
               cudaMemcpyDeviceToHost) != cudaSuccess)
    return 1;
```

kernel is launched using multiple blocks and threads. Synchronisation ensures that all GPU computations finish before results are copied back to the CPU.

## Compile and run

Compile:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ nvcc CryptForCuda.cu  -o CryptForCuda
```

Run:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ ./CryptForCuda
```

## Output:

Filename: cuda_passwords.txt

```
bxacwy0679
bxacwy0680
bxacwy0691
bxacwy0602
bxacwy0613
bxacwy0624
bxacwy0635
bxacwy1746
bxacwy1757
bxacwy1768
bxacwy1779
bxacwy1780
bxacwy1791
bxacwy1702
bxacwy1713
bxacwy1724
bxacwy1735
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$
```

*Figure 7 cuda_passwords.txt*

## SHA-512 Hashing of CudaCrypt

This step involves changing the cudaCrypt function written in this case by a CPU-based cudaCrypt into a CUDA device function meaning this can be executed on the GPU. A CUDA kernel is created to generate all combinations of raw passwords between aa00 and zz99 which give rise to 67 600 unique combinations. One password is encrypted by each CUDA thread. On the x-dimension, several blocks and threads are employed to make sure the workload is spread effectively on the GPU. The passwords that are encrypted in the GPU are sent back to the host and stored in an output file.

### Compile and run

Compile:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ gcc EncryptSHA512.c -o EncryptSHA512 -lcrypt
```

Run:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ ./EncryptSHA512 Cuda_passwords.txt cuda_hashes.txt
```

It is the GPU analog of the CPU encryption functionality. It provides the same logic in order to make sure that encrypted passwords are created on the GPU are the same as those created on the CPU.

### Output:

Filename: cuda_hashes.txt

```
$6$AS$F18BUi/R6l8WRCIDCvdKfUXGPXVulA9XufmwZSoeHy26c6OgYbY2JjoaUihtGaGiMDuyaV8Q3HM8vaNKRX3q4.
$6$AS$fwSykLM8.IHAyo9y/dVMUKo5uVwrFhm1XUKm9hTtT0NaEG6zs2yEDRWTI181hjzZhV.9pp3iA/VPW2ZYfUSQI.
$6$AS$4MtdxvZwhz3Lhv5L4BbxJqMCBc9TjyUEhzSVG87sCrFXk/VPXBVC83x6xjP32cKAkQv3RFCzF9WiygHGAqDA/.
$6$AS$12IhIVAutGfANySmrEWDMs7zwCfhcfQmKYGRkF6Kd/zXxI.MMdHh0NVPSJyYBKgvGkN7D2FJc0hHKoq2PNueN0
$6$AS$xlOeFMuUhXoHy7nNjbIxtCmdmCVOsXviUWhrO7r099qz4rBuwsWyO0Xb76rA18dnnLWt2I81xyhcrnt/tDq4m/
$6$AS$h81Bho.MZghi.6jHrQkTG63FLC6.k3uw8Ri646ylUwV8shcZ/3ykdnU/xm/eGTJCLkT.NIrBvKHyfExrsMPSE.
$6$AS$Phqhs3NnDVYj.WdnyQxG30s8i3SKaOzJNIzAJilHiNjjoNW7NT6f.z//oTIL5gKy9Ipvx99KPh.g8lBdNG2yk/
$6$AS$leU7rUtwL79nbvnaYoOLgTyhX2ACJ2votXbbBaL/8b.MhFmDJFKXU4VP6aE9b0pXH05AXbRCehndT5P18TS260
$6$AS$KiACY8nMo2cLh70RcUvo7cKFPeKHhPYwS3ti7qyAvpXwuYANQF8e4sf3rSyzHBuTEu9BGZU/lWam9F/IyG9eV1
$6$AS$W2Q6a7FAe5pBJrZIjITC6oMujOI33GcWXS6liyaUwoWFfOhjieRzvvaRil2Zwu99M1hlpfQ3Sobi06uQ.0LXb1
$6$AS$6LefSchak4z/lDT2L4Ygmu9y7Ffk27tX6JbsMdkgof2JLOItPa4NDmWbLABnQY.irhd6FYgW7E2D4KJ/b8lbj0
$6$AS$iBVG7zSoQiV..0uKrqtDwZ.HZbAVkYlINL3Y11A8yXG30MFJmwLXqek6TqSJW/JBZFpsrhWG4eD7iI0F2/Ig./
$6$AS$2UCVhnGHnimElgCs9948I..XSNpjCokHhUv2e53t3giWus1UVNtvNqouHqPoX.QKbFMX1BlxQ9E0KDeEzMh/.0
$6$AS$dGeReJ1MHa60.mRDOqQlkZx6TSQANYxDG/3TQVlS9ns2t.aC7ZtO7GQqK9syzUe55q/NMmHYiLNUsP.Ddv9NJ/
$6$AS$YgzFbhEGWN1W6DkNwwKJKIKSMB0AjqaAuoBONkUceG7t2Rsm.AvbfAdvjQMaRNPc.g.Ij2l51qrwuwe3hm003.
$6$AS$kKDYr5kOlxCBZ1yvLrK0M9e/yZJXKnB0Y4nFlzpNM4JDlQpzpomgEp3tj5sR6cLuGFpU7uQ9nukLDcTFVUO2Z/
$6$AS$1HSWbUfo5Rlt4I4NDqyrubIuEWMaEPyR6YNzAW.C71CtIGDGUxPm/Tx.roOAZ.h.2hdyVLfsGjPYq657T5Ywo1
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$
```

*Figure 8 cuda_hashes.txt*

# Password Cracking and Matching

## Source Code

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ cat CudaCrackPasswords.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>

#define MAX_HASH_LEN 130
#define TOTAL_CUDA_PASSWORDS 67600

__device__ int device_strcmp(const char *a, const char *b) {
    for (int i = 0; i < MAX_HASH_LEN; i++) {
        if (a[i] != b[i]) return 0;
        if (a[i] == '\0' && b[i] == '\0') return 1;
    }
    return 1;
}

__global__ void crackKernel(
    char *fileHashes,
    char *cudaHashes,
    int *results,
    int totalFileHashes
) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= totalFileHashes) return;

    char *target = &fileHashes[tid * MAX_HASH_LEN];

    for (int i = 0; i < TOTAL_CUDA_PASSWORDS; i++) {
        char *candidate = &cudaHashes[i * MAX_HASH_LEN];
        if (device_strcmp(target, candidate)) {
            results[tid] = i;
            return;
        }
    }
    results[tid] = -1;
}

void indexToPassword(int index, char *out) {
    out[0] = 'a' + (index / 2600);
    out[1] = 'a' + (index / 100) % 26;
    out[2] = '0' + (index / 10) % 10;
    out[3] = '0' + index % 10;
    out[4] = '\0';
}

int main() {

    FILE *f1 = fopen("file_hashes.txt", "r");
    FILE *f2 = fopen("cuda_hashes.txt", "r");
    if (!f1 || !f2) return 1;

    int count = 0;
    char line[MAX_HASH_LEN];
    while (fgets(line, MAX_HASH_LEN, f1)) count++;
    rewind(f1);

    char *h_fileHashes = (char*)malloc(count * MAX_HASH_LEN);
    char *h_cudaHashes = (char*)malloc(TOTAL_CUDA_PASSWORDS * MAX_HASH_LEN);

    for (int i = 0; i < count; i++) {
        fgets(&h_fileHashes[i * MAX_HASH_LEN], MAX_HASH_LEN, f1);
        h_fileHashes[i * MAX_HASH_LEN +
            strcspn(&h_fileHashes[i * MAX_HASH_LEN], "\n")] = '\0';
    }

    for (int i = 0; i < TOTAL_CUDA_PASSWORDS; i++) {
        fgets(&h_cudaHashes[i * MAX_HASH_LEN], MAX_HASH_LEN, f2);
        h_cudaHashes[i * MAX_HASH_LEN +
            strcspn(&h_cudaHashes[i * MAX_HASH_LEN], "\n")] = '\0';
    }

    fclose(f1);
    fclose(f2);
```

*Figure 9 CudaCrackPasswords.cu*

```
    char *d_fileHashes, *d_cudaHashes;
    int *d_results;

    cudaMalloc((void**)&d_fileHashes, count * MAX_HASH_LEN);
    cudaMalloc((void**)&d_cudaHashes, TOTAL_CUDA_PASSWORDS * MAX_HASH_LEN);
    cudaMalloc((void**)&d_results, count * sizeof(int));

    cudaMemcpy(d_fileHashes, h_fileHashes,
               count * MAX_HASH_LEN, cudaMemcpyHostToDevice);
    cudaMemcpy(d_cudaHashes, h_cudaHashes,
               TOTAL_CUDA_PASSWORDS * MAX_HASH_LEN, cudaMemcpyHostToDevice);

    int threads = 256;
    int blocks = (count + threads - 1) / threads;

    crackKernel<<<blocks, threads>>>(
        d_fileHashes, d_cudaHashes, d_results, count);
    cudaDeviceSynchronize();

    int *h_results = (int*)malloc(count * sizeof(int));
    cudaMemcpy(h_results, d_results,
               count * sizeof(int), cudaMemcpyDeviceToHost);

    FILE *out = fopen("decrypted_passwords.txt", "w");
    char raw[5];

    for (int i = 0; i < count; i++) {
        if (h_results[i] >= 0) {
            indexToPassword(h_results[i], raw);
            fprintf(out, "%s\n", raw);
        } else {
            fprintf(out, "NOT FOUND\n");
        }
    }

    fclose(out);

    cudaFree(d_fileHashes);
    cudaFree(d_cudaHashes);
    cudaFree(d_results);
    free(h_fileHashes);
    free(h_cudaHashes);
    free(h_results);

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$
```

This is the last process of the password cracking pipeline. The hashed passwords produced by the original dataset (Step 2) are compared with the hashed passwords produced with CUDA (Step 4) in this step. The matching of hashes and recovery of original raw passwords is efficiently done using CUDA parallelism. Both hash files are initially loaded into the memory of the CPU and transmitted to the GPU memory. Parallel hash comparison is done by a CUDA kernel. Upon the identification of a match on a hash, the index is transformed back to the raw password form (aa00-zz99). The retrieved passwords are then loaded back into the CPU and stored in an output file.

```
FILE *f1 = fopen("file_hashes.txt", "r");
FILE *f2 = fopen("cuda_hashes.txt", "r");
```

This block opens the two input files with the hashes of the SHA-512. The first file has the hash of passwords produced in Step 1 and the second file has the hash of all the possible passwords produced by CUDA.

```
cudaMalloc((void**)&d_fileHashes, count * MAX_HASH_LEN);
cudaMalloc((void**)&d_cudaHashes, TOTAL_CUDA_PASSWORDS * MAX_HASH_LEN);
cudaMalloc((void**)&d_results, count * sizeof(int));
```

The processor allocates the memory of the GPUs to store the hash datasets and an array to store matching indices. The allocations are calculated using the number of hashes read on the input files and this defines an efficient use of memory.

```
__device__ int device_strcmp(const char *a, const char *b) {
    for (int i = 0; i < MAX_HASH_LEN; i++) {
        if (a[i] != b[i]) return 0;
        if (a[i] == '\0' && b[i] == '\0') return 1;
    }
    return 1;
}
```

This operation does a direct hash string comparison in the GPU. It is necessary since C library functions of dealing with strings are not applicable within CUDA device code.

```
__global__ void crackKernel(
    char *fileHashes,
    char *cudaHashes,
    int *results,
    int totalFileHashes
) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= totalFileHashes) return;

    char *target = &fileHashes[tid * MAX_HASH_LEN];

    for (int i = 0; i < TOTAL_CUDA_PASSWORDS; i++) {
        char *candidate = &cudaHashes[i * MAX_HASH_LEN];
        if (device_strcmp(target, candidate)) {
            results[tid] = i;
            return;
        }
    }
    results[tid] = -1;
}
```

Every CUDA thread works with one hashed password of the original dataset. This hash is compared by the thread to all the CUDA produced hashes. When a match is discovered, the

index of the matching CUDA password is saved.

```
int threads = 256;
int blocks = (count + threads - 1) / threads;

crackKernel<<<blocks, threads>>>(
    d_fileHashes, d_cudaHashes, d_results, count);
cudaDeviceSynchronize();
```

The x-dimension involves the use of several blocks and threads to launch the kernel. The threads operate independently and perform one hash comparison task each, making them very efficient to execute in parallel.

## Compile and run

Compile:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ nvcc CudaCrackPasswords.cu -o CudaCrackPassword
```

Run:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/2407710_Dhadkan_KC_6CS005/Task_3$ ./CudaCrackPassword
```

It is the GPU analog of the CPU encryption functionality. It provides the same logic in order to make sure that encrypted passwords are created on the GPU are the same as those created on the CPU.

**Output:**

Filename: decrypted_passwords.txt

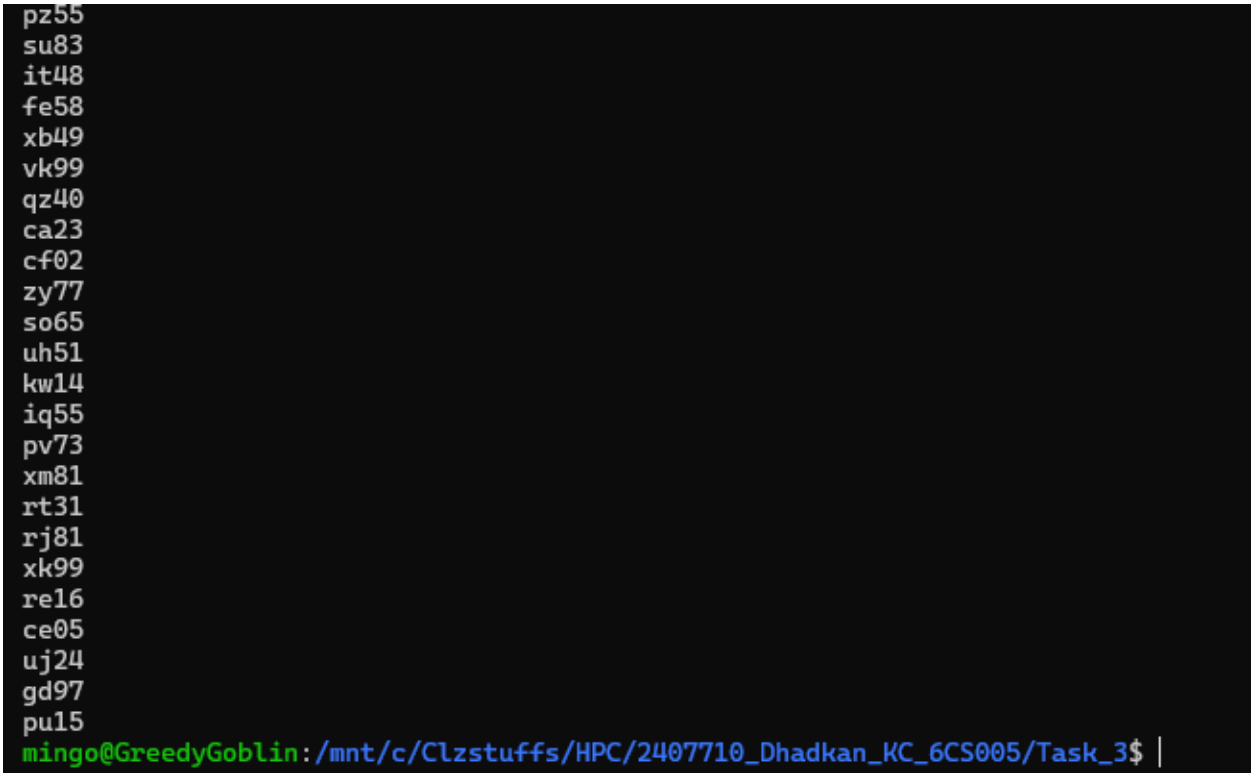


*Figure 10 decrypted_passwords.txt*

This step completes the password cracking process by leveraging CUDA parallelism to efficiently match SHA-512 hashes and recover original passwords. By distributing the workload across multiple GPU threads, the cracking process is significantly faster than a CPU-only approach.

## Comparison: CPU vs CUDA Implementation

Password generation and cracking and hashing are performed using a CPU-only method, and computationally inexpensive, with the operations occurring sequentially which is computationally expensive as the size of the password dataset expands. Conversely, the CUDA implementation uses massive parallelism to allocate a single task of password related operation to each thread on the GPU. Generation of passwords and hash matching are thus performed in thousands of threads, therefore cutting on the execution time considerably. Also, block and thread allocation in the C UDA implementation is dynamic which provides efficient utilization of the resources of the GPU. All said and done, CUDA strategy has better performance, scalability as well as efficiency when compared to purely CPU based solution especially in brute force password cracking.

## Limitations

- The password structure is only limited to two lower case letters, then two digits (aa00-zz99), which prevents the maximum address space of a password.
- SHA-512 hash is calculated solely on the CPU since cryptographic hash is not performed on the graphic card.
- Memory optimizations such as shared memory are not used and this could also be optimized to enhance performance.
- CUDA debugging is limited and can be extended to a stronger one.
- The implementation is sensitive to valid input files and does not deal with corrupted and malformed data largely.

## Conclusion

This assignment has managed to show how CUDA is used to speed up password cracking by using CPU preprocessing and parallel computation on the company of the GPUs. The passwords are encrypted and hashed on the CPU and the GPU is efficient in generating all the possible password combinations and parallel match hash. The implementation has properly handled file input/output, dynamic memory allocation and thread allocation, host-device memory transfers and memory deallocation. The retrieved passwords testify to the appropriateness of the method, and the solution fits all the requirements of the tasks and demonstrates the obvious benefits of the work in terms of performance of the use of GPUs.