Task1, Task2, Task3 andTask4(6CS005)

|  |  |
| --- | --- |
| University ID | 2407642 |
| Student Name | Pranjal Babu Koirala |
| Group | L6CG8 |
| Tutor | Manyak Baryal |
|  |  |

Table of Contents

[Task1 1](#_Toc219982969)

[WordOccurrence 1](#_Toc219982970)

[Main Part of Code: 8](#_Toc219982971)

[Includes and Defines 8](#_Toc219982972)

[Explanation: 8](#_Toc219982973)

[Structures 8](#_Toc219982974)

[Explanation: 9](#_Toc219982975)

[Hash Function 9](#_Toc219982976)

[Explanation: 9](#_Toc219982977)

[Add Word 9](#_Toc219982978)

[Explanation: 10](#_Toc219982979)

[Free Table 10](#_Toc219982980)

[Explanation: 10](#_Toc219982981)

[Thread Work 11](#_Toc219982982)

[Explanation: 11](#_Toc219982983)

[Merge Table 11](#_Toc219982984)

[Explanation: 12](#_Toc219982985)

[Explanation: 12](#_Toc219982986)

[Main Flow 12](#_Toc219982987)

[Explanation: 13](#_Toc219982988)

[Output: 14](#_Toc219982989)

[Explanation: 16](#_Toc219982990)

[Task2 17](#_Toc219982991)

[task2\_advance 17](#_Toc219982992)

[Main part of Code: 20](#_Toc219982993)

[Includes and Structure 20](#_Toc219982994)

[Explanation: 20](#_Toc219982995)

[Allocate Matrix 20](#_Toc219982996)

[Explanation: 21](#_Toc219982997)

[Free Matrix 21](#_Toc219982998)

[Explanation: 21](#_Toc219982999)

[Cap Threads 21](#_Toc219983000)

[Explanation: 21](#_Toc219983001)

[Print Matrix 22](#_Toc219983002)

[Explanation: 22](#_Toc219983003)

[Matrix Operation 23](#_Toc219983004)

[Explanation: 23](#_Toc219983005)

[Main Flow 24](#_Toc219983006)

[Explanation: 24](#_Toc219983007)

[Memory Deallocation 24](#_Toc219983008)

[Explanation: 25](#_Toc219983009)

[Close the file: 25](#_Toc219983010)

[Explanation: 25](#_Toc219983011)

[Output: 25](#_Toc219983012)

[Explanation: 27](#_Toc219983013)

[Task3 28](#_Toc219983014)

[Full Code: Password\_Task 28](#_Toc219983015)

[PasswordGeneratorToText.c 28](#_Toc219983016)

[EncryptSHA512.c 29](#_Toc219983017)

[CryptForCuda.c 30](#_Toc219983018)

[CudaMatchHashes.cu 31](#_Toc219983019)

[Main part of code: 32](#_Toc219983020)

[Process 1: Header Files and Constant Definitions 32](#_Toc219983021)

[Explanation 32](#_Toc219983022)

[Process 2: Device-side Memory Comparison Function 33](#_Toc219983023)

[Explanation 33](#_Toc219983024)

[Process 3: CUDA Kernel for Hash Matching 33](#_Toc219983025)

[Explanation 33](#_Toc219983026)

[Process 4: Thread Index Calculation and Bounds Check 34](#_Toc219983027)

[Explanation 34](#_Toc219983028)

[Process 5: Parallel Hash Comparison Loop 34](#_Toc219983029)

[Explanation 34](#_Toc219983030)

[Retrieving Original 4-Digit Password 34](#_Toc219983031)

[Explanation 35](#_Toc219983032)

[Raw Password Generation 35](#_Toc219983033)

[Explanation 35](#_Toc219983034)

[CUDA Kernel Launch (Parallel Execution) 35](#_Toc219983035)

[Explanation 35](#_Toc219983036)

[Final Output Storage 36](#_Toc219983037)

[Explanation 36](#_Toc219983038)

[Output: 36](#_Toc219983039)

[Task4 38](#_Toc219983040)

[Full code: Sobel Edge Detection Using CUDA 38](#_Toc219983041)

[Main part of the code: 40](#_Toc219983042)

[Reading PNG Image(s) into CPU Memory 40](#_Toc219983043)

[Explanation 40](#_Toc219983044)

[GPU Memory Allocation and Data Transfer 40](#_Toc219983045)

[Explanation 41](#_Toc219983046)

[Sobel Edge Detection CUDA Kernel 41](#_Toc219983047)

[Explanation 41](#_Toc219983048)

[CUDA Kernel Launch (Parallel Execution) 42](#_Toc219983049)

[Explanation 42](#_Toc219983050)

[Copy Result Back and Write Output Image 42](#_Toc219983051)

[Explanation 42](#_Toc219983052)

[Memory Deallocation 43](#_Toc219983053)

[Explanation 43](#_Toc219983054)

[Output: 43](#_Toc219983055)

Table of Figures

[Figure 1: Includes and Defines 8](#_Toc219982573)

[Figure 2: Structures 8](#_Toc219982574)

[Figure 3: Hash Function 9](#_Toc219982575)

[Figure 4: Add Word 9](#_Toc219982576)

[Figure 5: Free Table 10](#_Toc219982577)

[Figure 6: Thread Work 11](#_Toc219982578)

[Figure 7: Merge Table 12](#_Toc219982579)

[Figure 8: Sort Compare 12](#_Toc219982580)

[Figure 9: Main Flow 13](#_Toc219982581)

[Figure 10: result.txt 15](#_Toc219982582)

[Figure 11: Includes and Structure 20](#_Toc219982583)

[Figure 12: Allocate Matrix 20](#_Toc219982584)

[Figure 13: Free Matrix 21](#_Toc219982585)

[Figure 14: Cap Threads 21](#_Toc219982586)

[Figure 15: Print Matrix 22](#_Toc219982587)

[Figure 16: Matrix Operation 23](#_Toc219982588)

[Figure 17: Main Flow 24](#_Toc219982589)

[Figure 18: Memory Deallocation 24](#_Toc219982590)

[Figure 19: Close the file: 25](#_Toc219982591)

[Figure 20: Result.txt 26](#_Toc219982592)

[Figure 21: PasswordGeneratorToText.c 28](#_Toc219982593)

[Figure 22: EncryptSHA512.c 29](#_Toc219982594)

[Figure 23: CryptForCuda.c 30](#_Toc219982595)

[Figure 24: CudaMatchHashes.cu 32](#_Toc219982596)

[Figure 25: Header Files and Constant Definitions 32](#_Toc219982597)

[Figure 26: Device-side Memory Comparison Function 33](#_Toc219982598)

[Figure 27: CUDA Kernel for Hash Matching 33](#_Toc219982599)

[Figure 28: Thread Index Calculation and Bounds Check 34](#_Toc219982600)

[Figure 29: Parallel Hash Comparison Loop 34](#_Toc219982601)

[Figure 30: Retrieving Original 4-Digit Password 34](#_Toc219982602)

[Figure 31: Raw Password Generation 35](#_Toc219982603)

[Figure 32: CUDA Kernel Launch (Parallel Execution) 35](#_Toc219982604)

[Figure 33:Final Output Storage 36](#_Toc219982605)

[Figure 34: Cracked.txt Output 37](#_Toc219982606)

[Figure 35: Sobel Edge Detection Using CUDA 39](#_Toc219982607)

[Figure 36: Reading PNG Image(s) into CPU Memory 40](#_Toc219982608)

[Figure 37: GPU Memory Allocation and Data Transfer 40](#_Toc219982609)

[Figure 38: Sobel Edge Detection CUDA Kernel 41](#_Toc219982610)

[Figure 39: CUDA Kernel Launch (Parallel Execution) 42](#_Toc219982611)

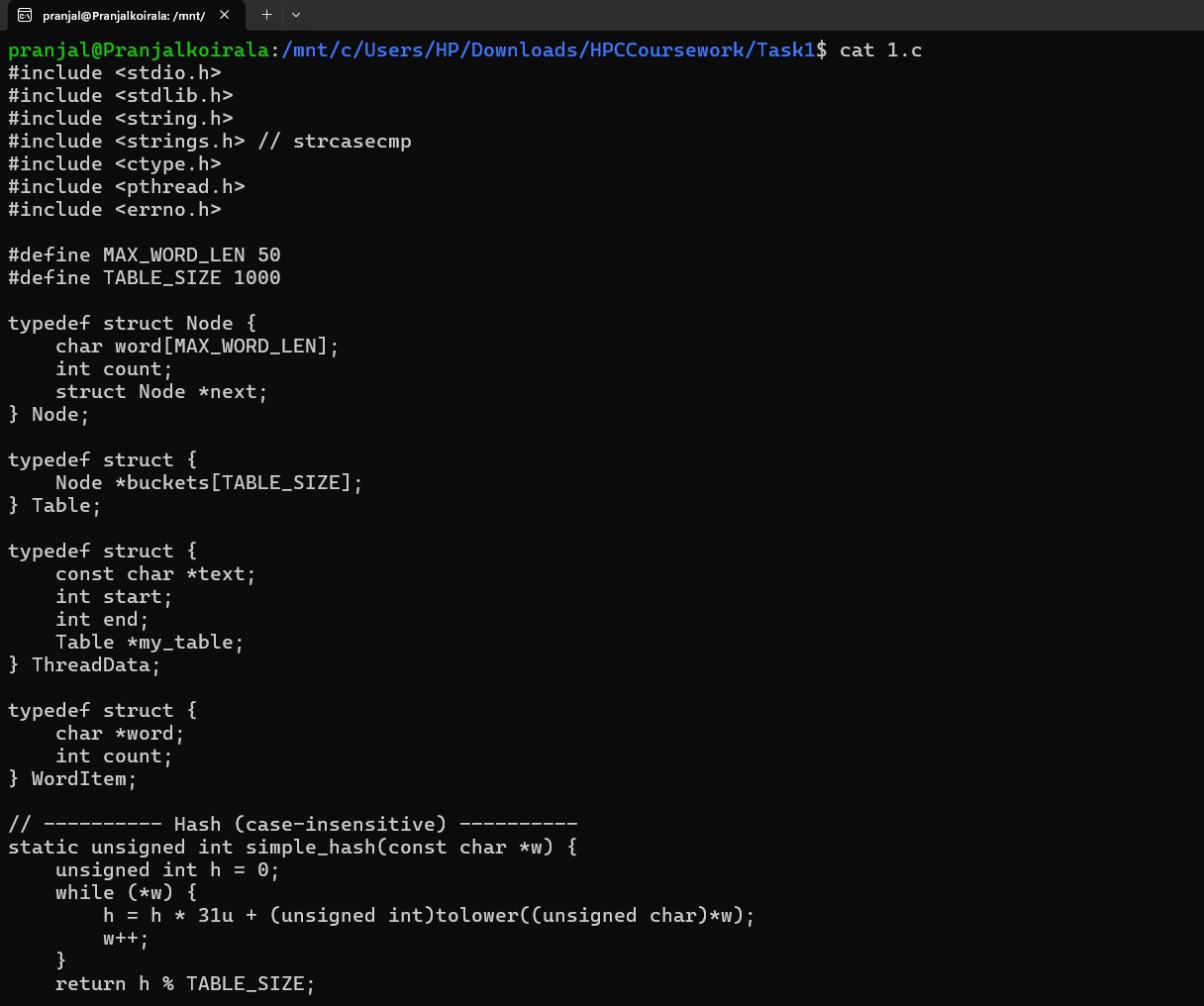
[Figure 40: Copy Result Back and Write Output Image 42](#_Toc219982612)

[Figure 41: Memory Deallocation 43](#_Toc219982613)

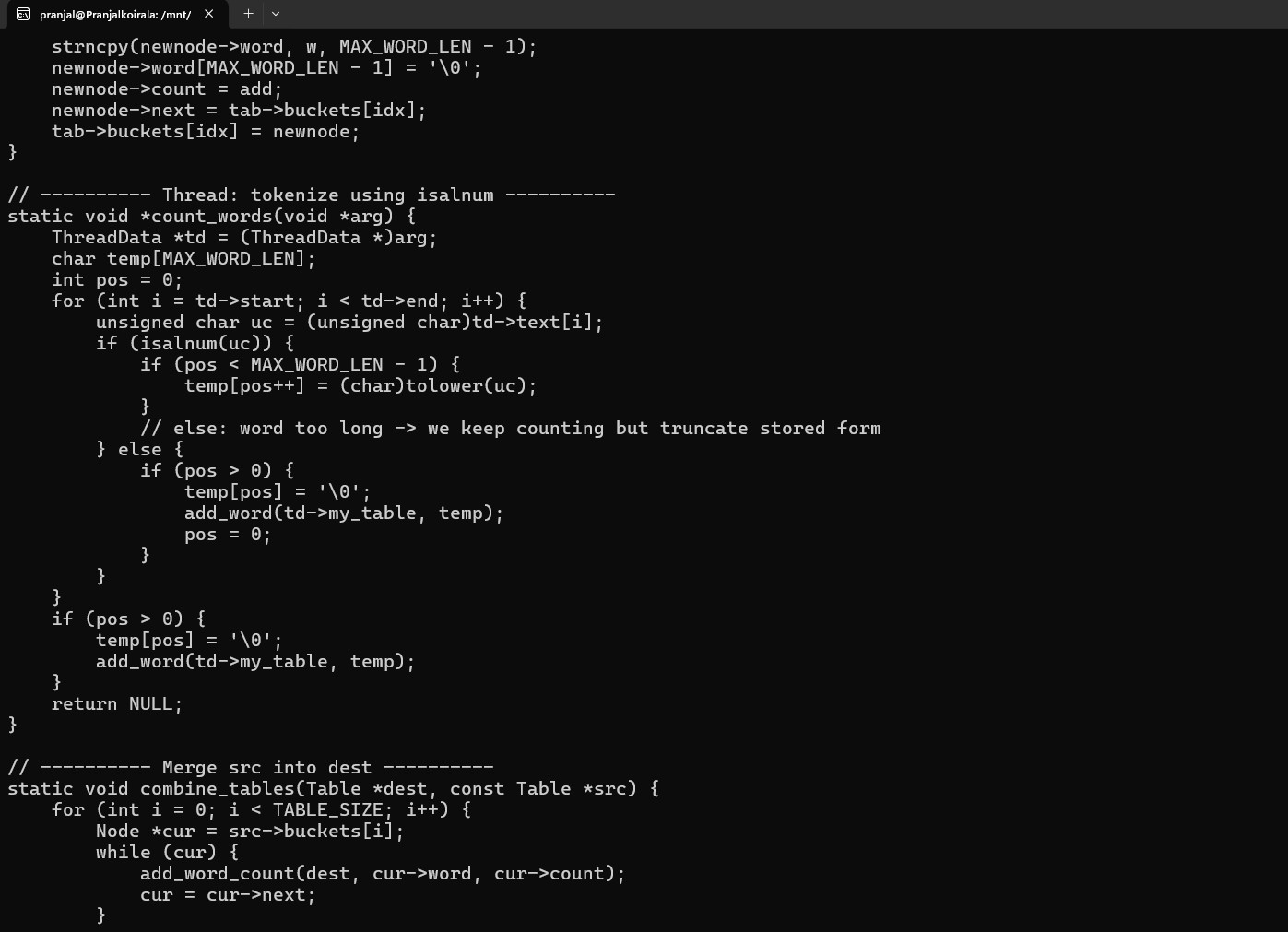
[Figure 42: Output of image hck.png 43](#_Toc219982614)

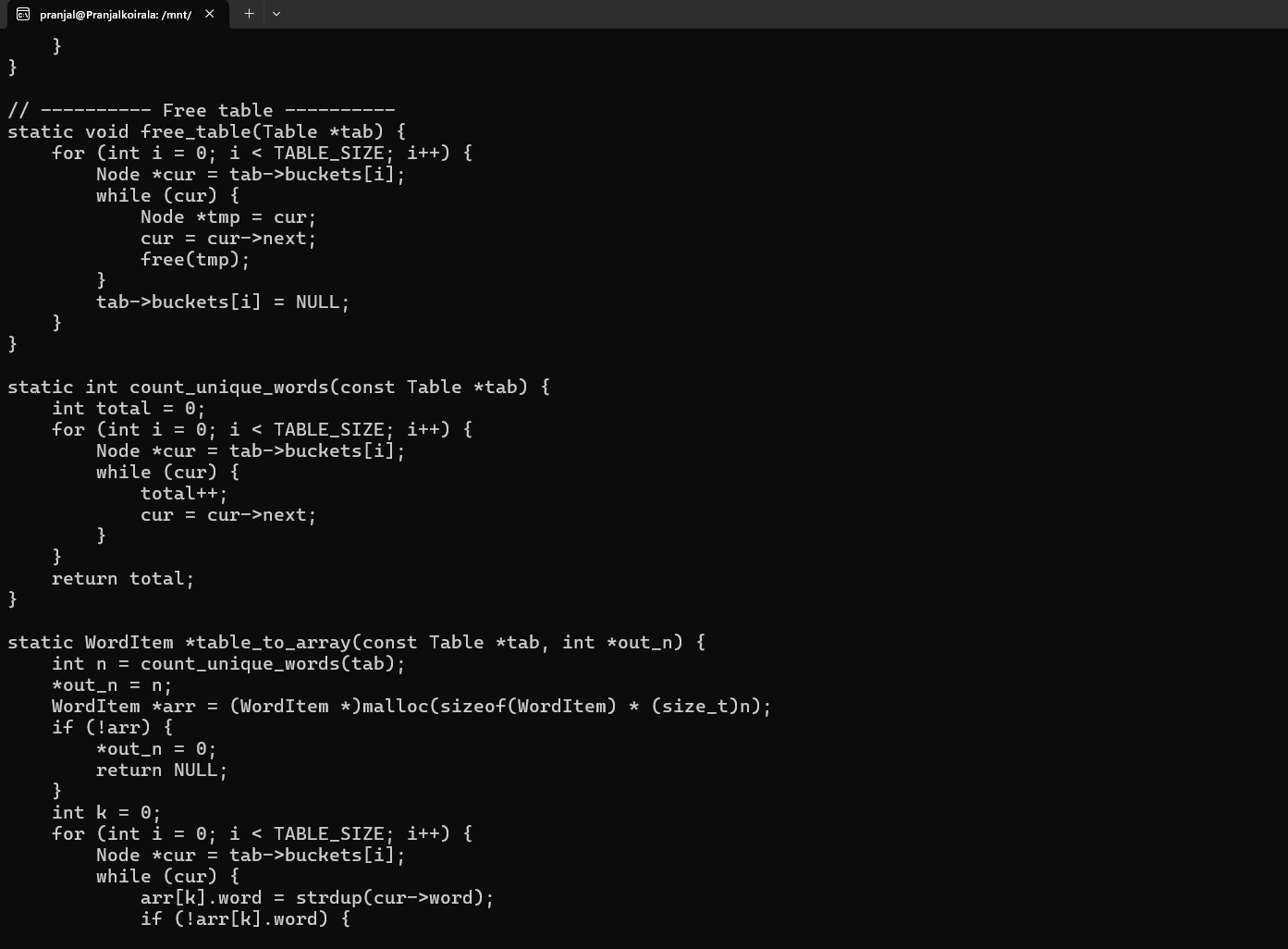
## Task1

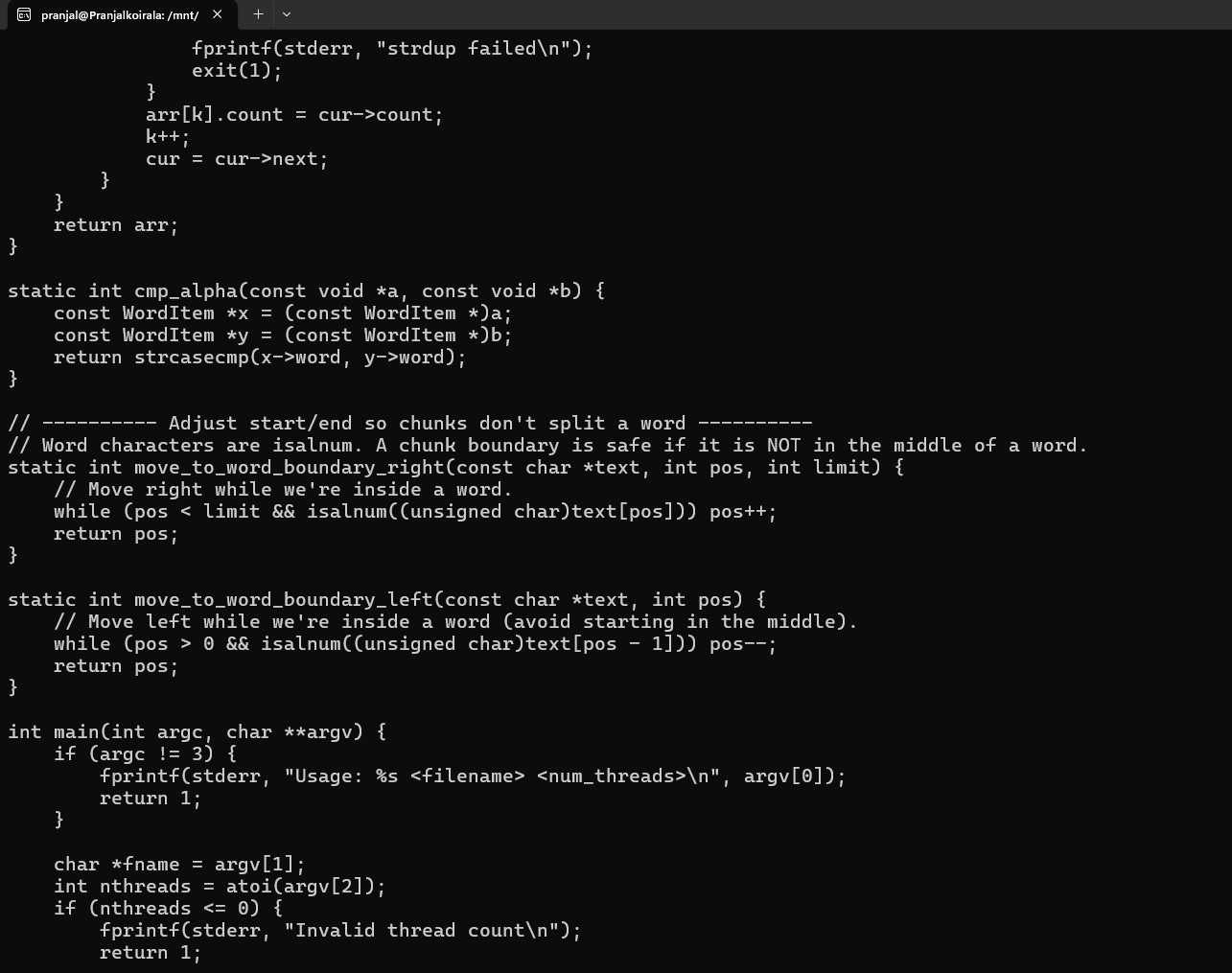
## WordOccurrence















## Main Part of Code:

## Includes and Defines

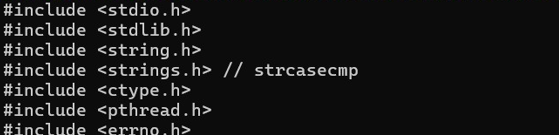


Figure : Includes and Defines

## Explanation:

These lines bring in everything the program needs. stdio.h is for printing and files, stdlib.h for memory and string.h for text functions, strings, strings.h for case-insensitive compare, ctype.h for tolower, pthread.h for threads, errno.h for error messages. MAX\_WORD\_LEN 50 limits word size so no overflow. TABLE\_SIZE 1000 is number of buckets in hash table enough to hold many words without too many collisions.

## Structures

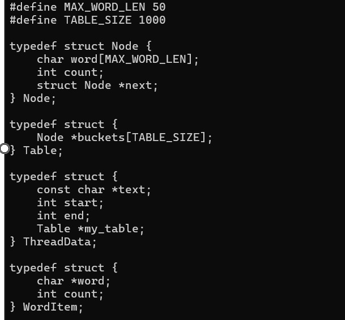


Figure : Structures

## Explanation:

This section defines the data structures used throughout the program. The Node structure stores a word, its count, and a pointer to the next node for collision handling using chaining. The Table structure represents the hash table with multiple buckets. ThreadData stores information needed by each thread, such as text range and private hash table. WordItem is used later to store words and count together for sorting and output.

## Hash Function

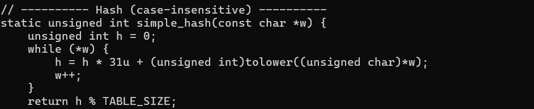


Figure : Hash Function

## Explanation:

The hash function converts a word into a numeric index for the hash table. It processes each character of the word by converting it to lowercase and combining it using multiplication by 31, which helps distribute values evenly. Taking modulo with TABLE\_SIZE ensures the index stays within valid bucket limits. This approach minimizes collisions and ensures fast word insertion and lookup. Case-insensitive hashing ensures words like “Text” and “text” are treated the same.

## Add Word

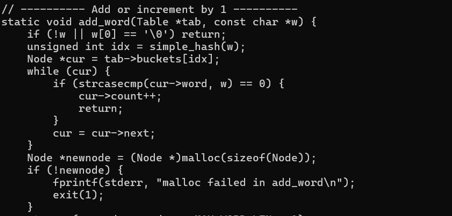


Figure : Add Word

## Explanation:

This function inserts words into the hash table or updates their count if they already exist. First, it ignores empty strings to avoid invalid entries. It calculates the hash index and checks the linked list at that bucket for an existing word. If found, the word count is increased. Otherwise, a new node is created and added to the front of the linked list, efficiently handling collisions.

## Free Table

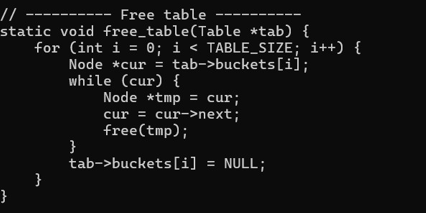


Figure : Free Table

## Explanation:

This part is responsible for freeing all dynamically allocated memory in the hash table. It loops through each bucket and traverses the linked list stored in it. Each node is freed one by one to prevent memory leaks. After clearing all nodes, the bucket pointers are set to NULL. This ensures the program exits cleanly without wasting system memory.

## Thread Work

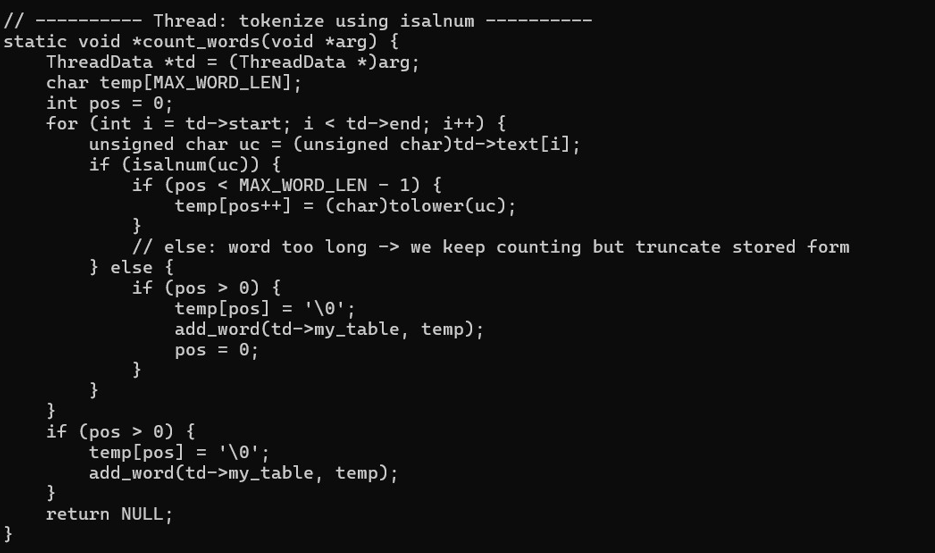


Figure : Thread Work

## Explanation:

This function is executed by each thread independently. Each thread processes a specific portion of the text assigned to it. It skips non-alphanumeric characters and builds words character by character while converting them to lowercase. Once a word is complete, it is added to the thread’s private hash table. Using separate hash tables avoids race conditions and removes the need for locks.

## Merge Table

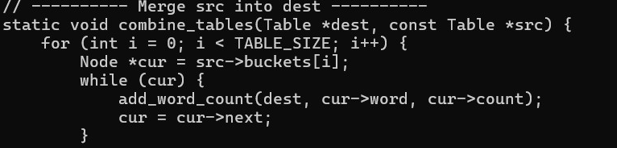


Figure : Merge Table

## Explanation:

After all threads finish execution, this function merges individual hash tables into a single main table. It loops through each bucket of the thread’s table and inserts every word into the main table. If a word already exists, its count is added. This step combines all partial results into one final accurate word count. It ensures correctness while still benefiting from parallel execution.

**Sort Compare**

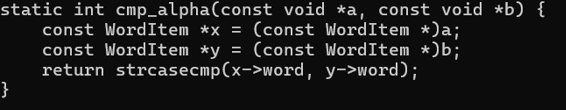
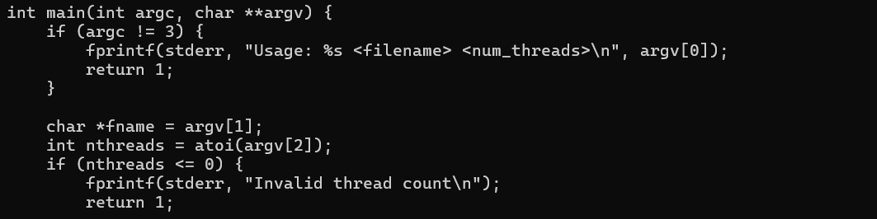
****

Figure : Sort Compare

## Explanation:

This function is used with qsort to sort words alphabetically. It compares words using a case-insensitive comparison method, so sorting is consistent. This ensures that uppercase and lowercase words appear together. Sorting improves readability of the final output file. It prepares the data for clean and organized presentations.

## Main Flow



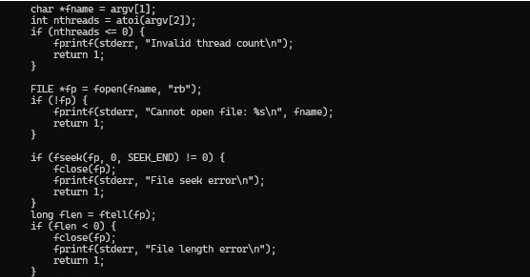


Figure : Main Flow

## Explanation:

The main function controls the entire program execution. It checks command-line arguments, opens the input file, and reads its content into memory. The text is divided into segments and assigned to multiple threads without splitting words. Threads are created and executed in parallel to process the text. After completion, results are merged, sorted, written to a file, and memory is freed.

## Output:

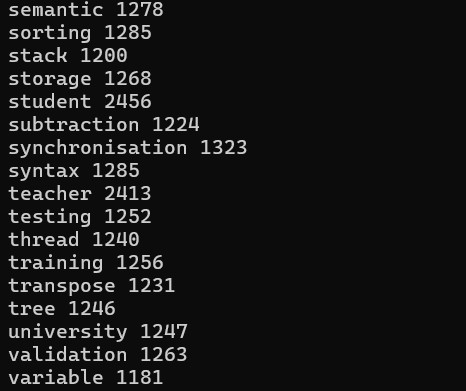
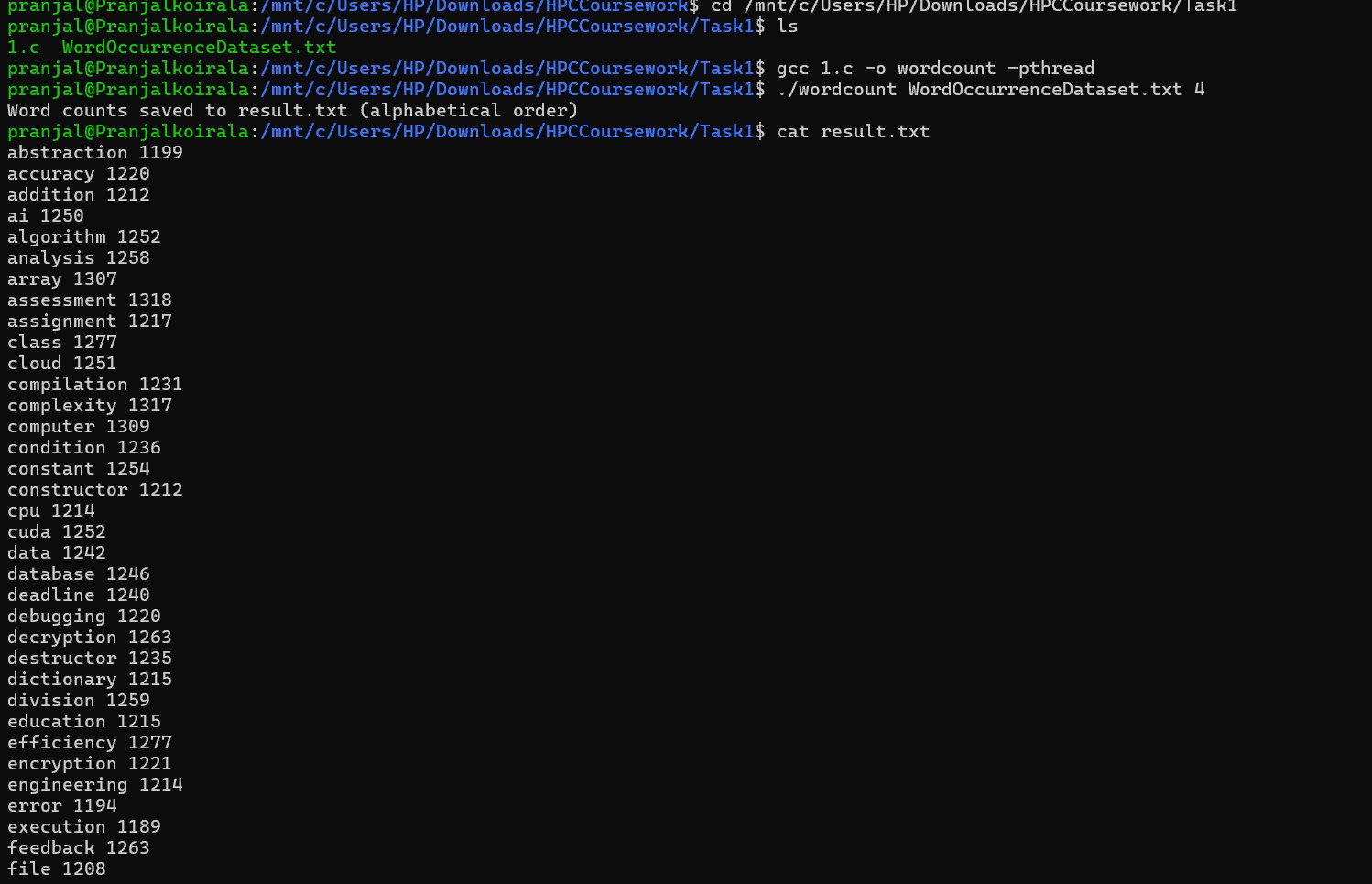
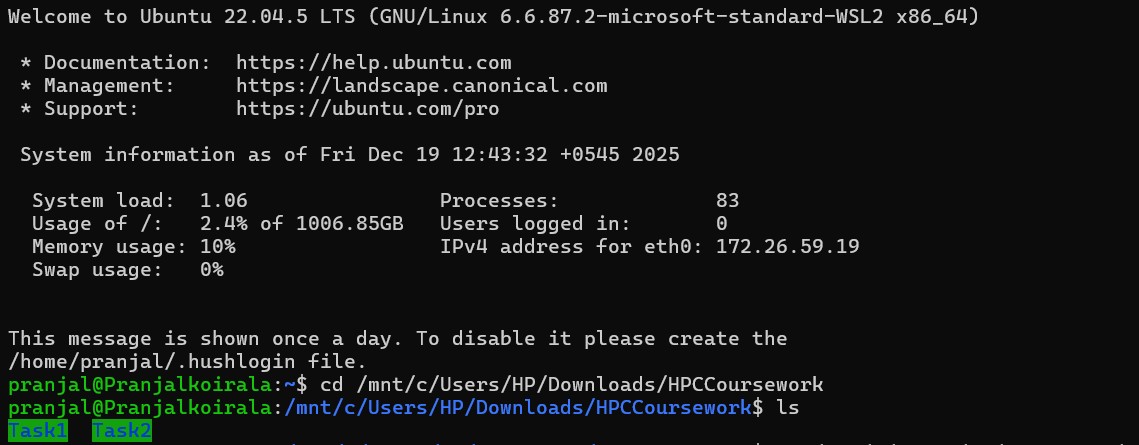


Figure : result.txt

## Explanation:

In this task, I wrote a program to count how many times each word appears in a large text file using multiple threads to make the process faster. When the program runs, I give the file name and the number of threads as input. The program first opens the file and reads the whole content into memory. After that, it divides the text into parts based on the number of threads, but I made sure that no word is split between two threads by adjusting the start and end positions properly. Each thread then processes its own part of the text at the same time. While reading characters one by one, the thread forms words using letters and numbers, converts them to lowercase, and stores them in its own hash table. If a word already exists, the count is increased, otherwise a new entry is created. Using separate hash tables for each thread avoids conflicts and makes the program safe without needing locks. Once all threads finish execution, the main program waits for them and then combines all the individual hash tables into one final table by adding the counts together. After merging, all unique words are sorted alphabetically and written to a file called result.txt. Finally, all allocated memory is freed and the program exits successfully. This approach makes the word counting fast, accurate, and efficient even for very large files.

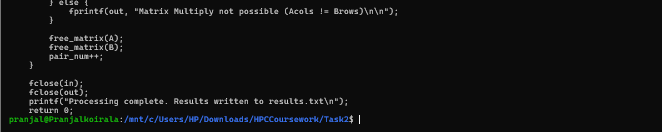
## Task2

## task2\_advance

****

****

****

****

## Main part of Code:

## Includes and Structure



Figure : Includes and Structure

Explanation: This section includes required libraries for matrix operations and parallel execution. stdio.h and stdlib.h handle file input, output, and memory allocation. omp.h enables OpenMP-based parallel processing to speed up computations. math.h provides support for special values like NaN during division. The Matrix structure stores row count, column count, and a dynamic 2D array for matrix values

Allocate Matrix

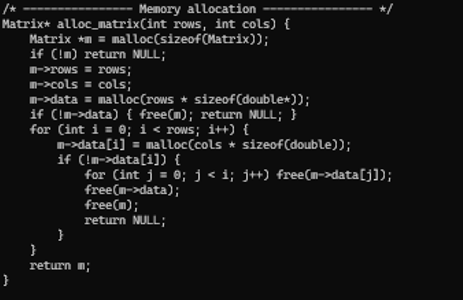


Figure : Allocate Matrix

## Explanation:

This function dynamically allocates memory for a matrix of given size. First, memory is allocated for the matrix structure itself. Then an array of row pointers is created, followed by memory allocation for each row. If allocation fails at any point, previously allocated memory is safely freed. This careful approach prevents memory leaks and ensures stability.

## Free Matrix

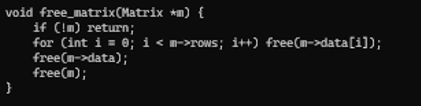


Figure : Free Matrix

## Explanation:

This function releases all memory used by a matrix. Each row is freed individually, followed by the array of row pointers. Finally, the matrix structure itself is freed. This ensures no unused memory remains allocated after matrix operations. Proper deallocation is important when working with large matrices.

## Cap Threads

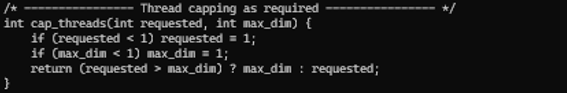


Figure : Cap Threads

## Explanation:

This part limits the number of OpenMP threads used during execution. It ensures that at least one thread is used and prevents creating more threads than matrix rows. This avoids unnecessary overhead and idle threads. Capping threads improves performance efficiency. It also makes the program behave predictably on different systems.

## Print Matrix

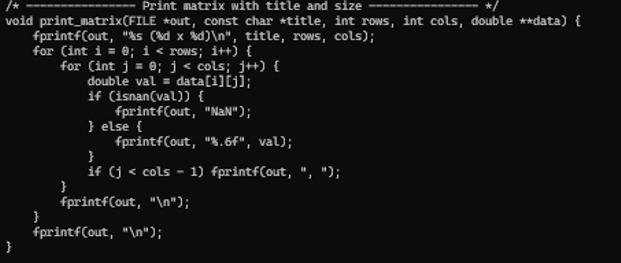


Figure : Print Matrix

## Explanation:

This function prints a matrix neatly to the output file. It displays the matrix name and dimensions before printing values. Each element is printed with fixed decimal precision for clarity. Rows and columns are formatted properly for readability. This makes the output easy to understand and verify.

## Matrix Operation

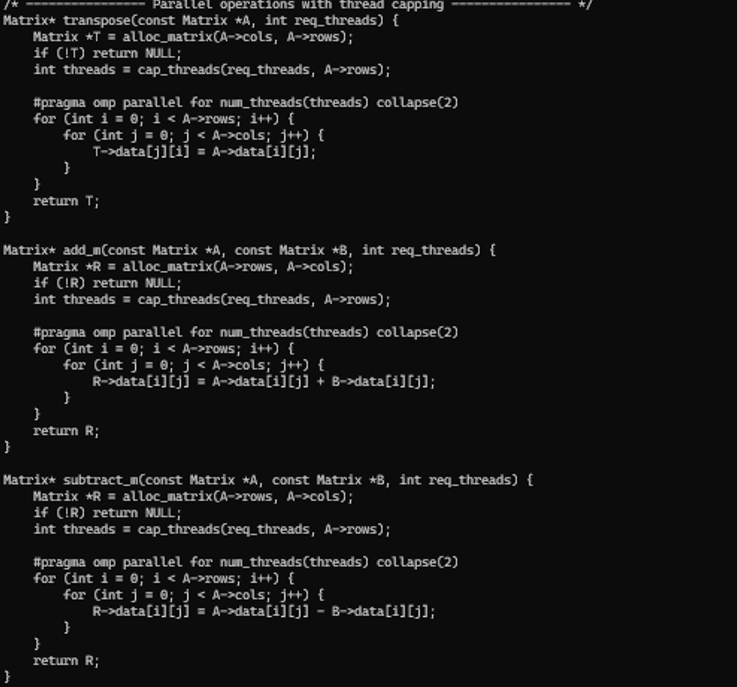


Figure : Matrix Operation

## Explanation:

This section performs matrix addition, subtraction, multiplication, and division using OpenMP. Parallel loops allow multiple elements to be processed at the same time. Dimensional checks are done before each operation to ensure validity. During division, division-by-zero is handled safely by producing NaN values. This ensures correctness and prevents runtime errors.

## Main Flow

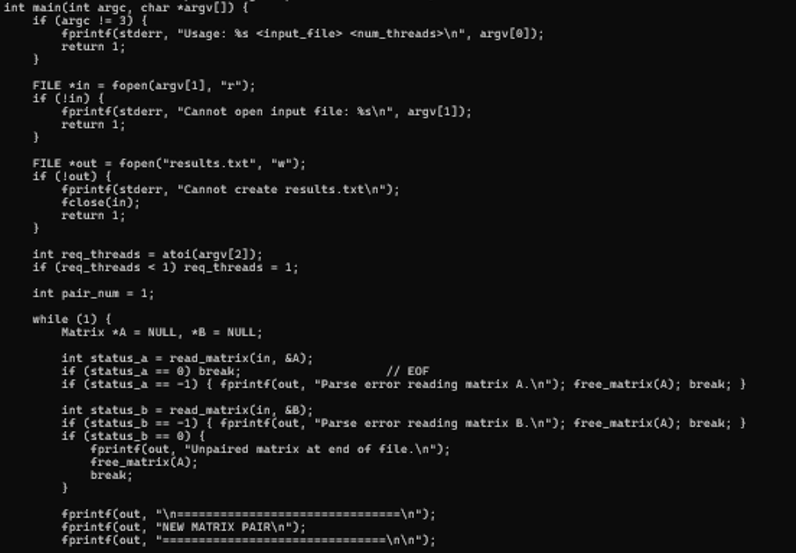


Figure : Main Flow

## Explanation:

The main function reads matrix data from the input file. It allocates matrices and fills them with values. Based on matrix dimensions, it decides which operations are possible. Results of each operation are written to an output file. After processing all matrix pairs, all memory is freed properly.

## Memory Deallocation

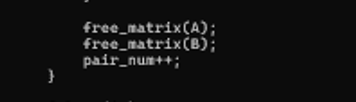


Figure : Memory Deallocation

## Explanation:

This section ensures all allocated matrices are freed after use. Temporary matrices created during operations are also released. This avoids memory leaks during long executions. Proper cleanup ensures the program runs efficiently even with large datasets. It is an essential part of safe memory management.

## Close the file:



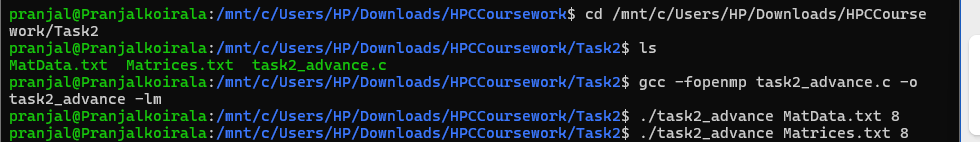
Figure : Close the file:

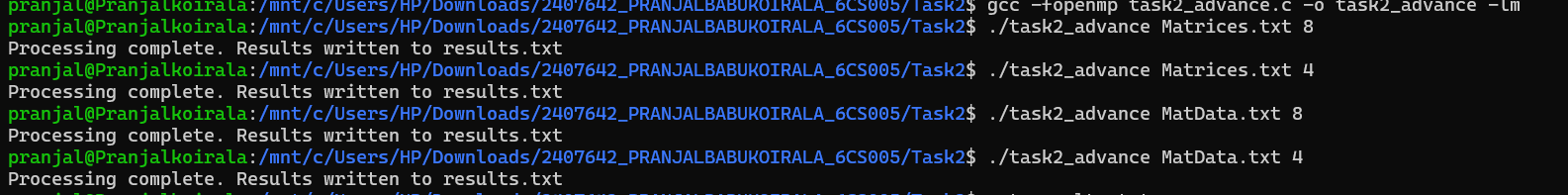
## Explanation:

Finally, all open files are properly closed. This ensures data is written completely to disk. Closing files also releases system resources. It marks successful completion of the program. The program then exits cleanly.

## Output:

****

****

****

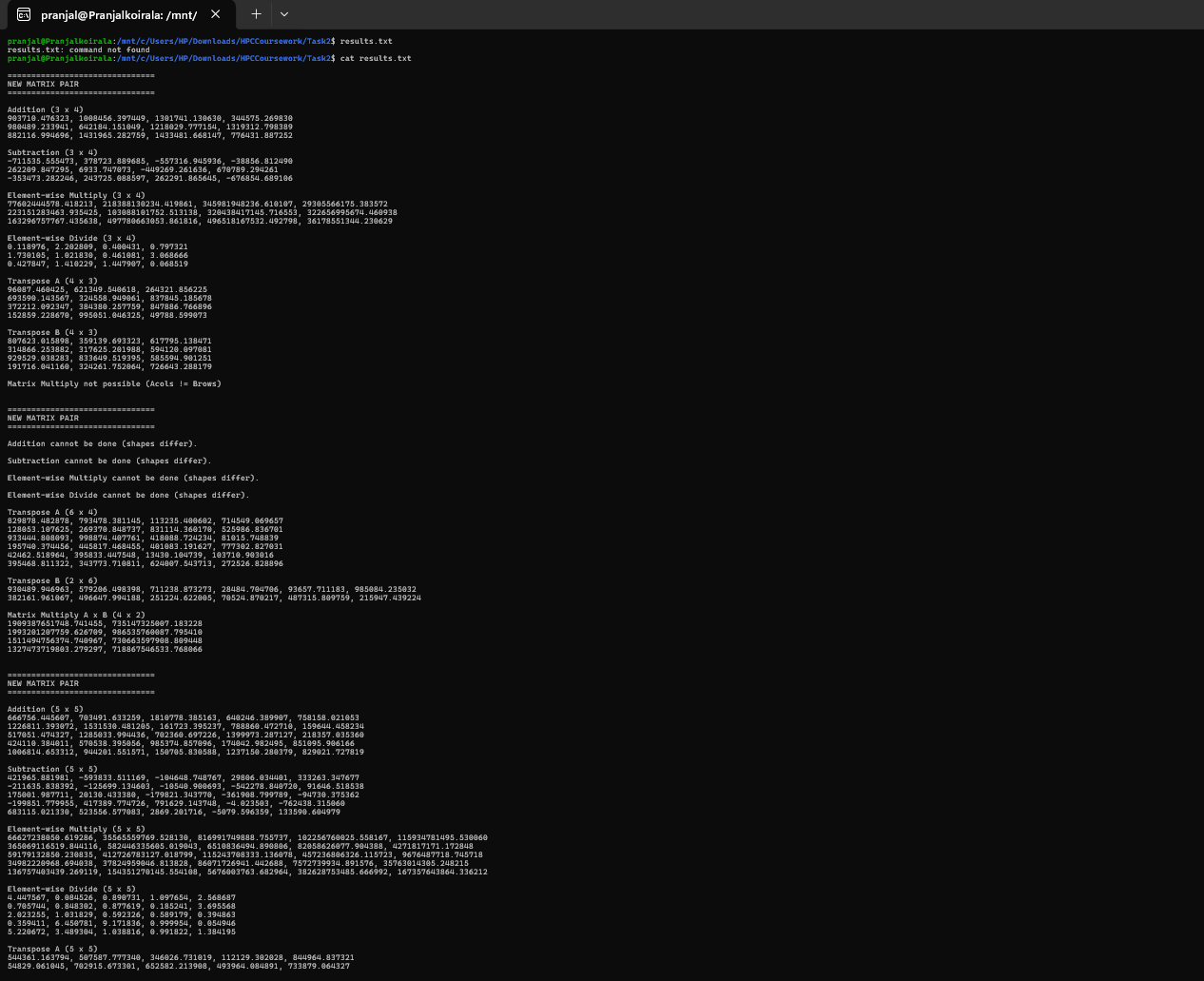
****

Figure : Result.txt

## Explanation:

In this task, I worked with two matrix data files named MatData and Matrices and performed different matrix operations using OpenMP for parallel processing. When the program runs, it reads matrix pairs from the input file along with their sizes and values. For each pair, I first check whether their dimensions allow certain operations. If both matrices are the same size, the program performs addition, subtraction, element wise multiplication, and division in parallel, so multiple elements are processed at the same time. During division, if any element is divided by zero, the program safely outputs NaN. The program also calculates the transpose of each matrix by swapping rows and columns using parallel loops. Matrix multiplication is only performed when the number of columns in the first matrix matches the number of rows in the second matrix, otherwise the program prints a clear message saying the operation is not possible. As each operation is completed, the results are written neatly into results.txt with proper headings. After processing all matrix pairs, the program frees all memory and finishes execution. This task shows how OpenMP helps speed up matrix computations while still handling different matrix sizes correctly.

## Task3

## Full Code: Password\_Task

## PasswordGeneratorToText.c



Figure : PasswordGeneratorToText.c

## EncryptSHA512.c

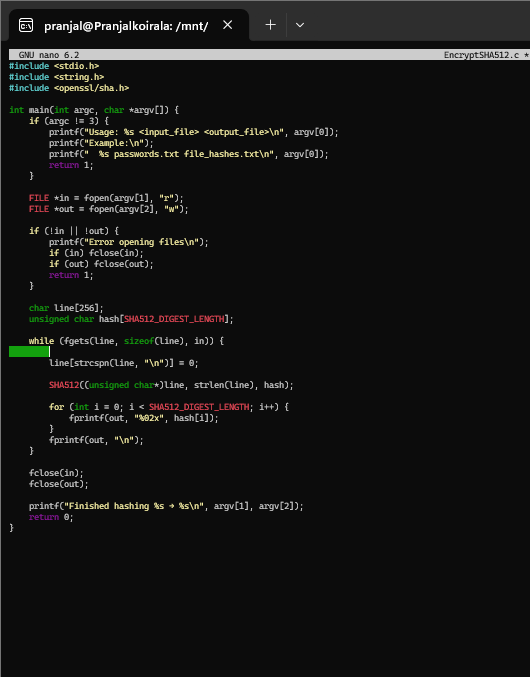


Figure : EncryptSHA512.c

## CryptForCuda.c

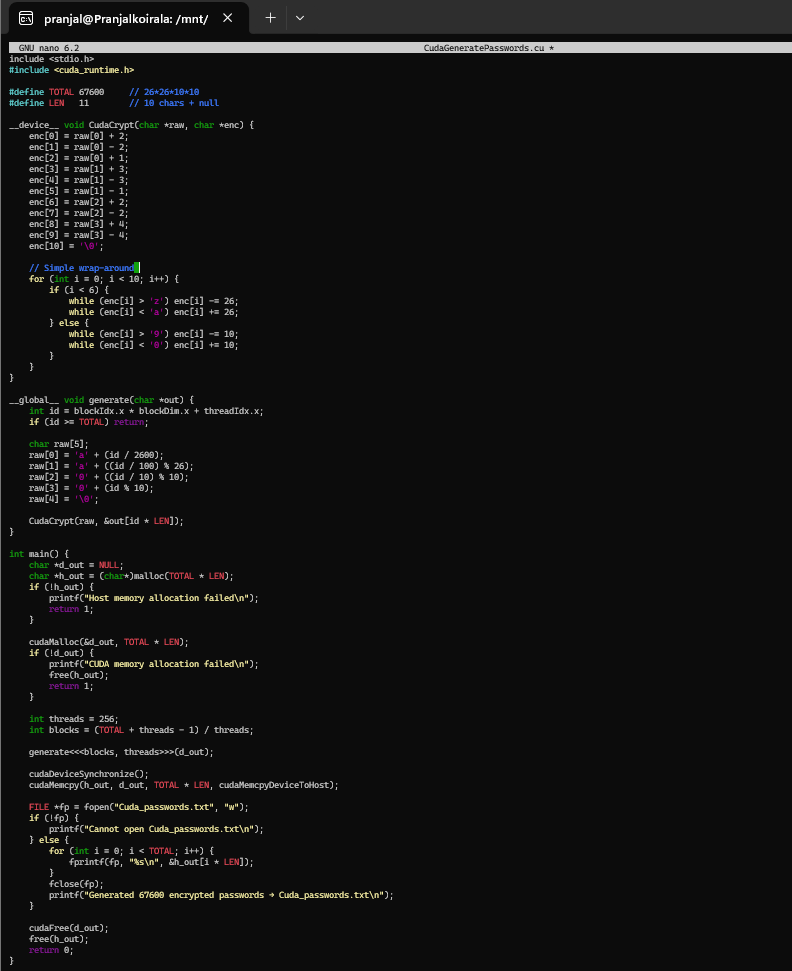


Figure : CryptForCuda.c

## CudaMatchHashes.cu



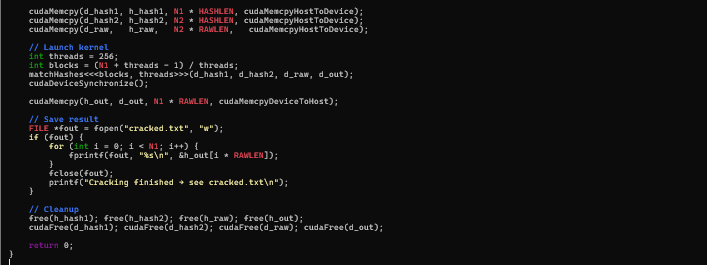


Figure : CudaMatchHashes.cu

## Main part of code:

## Process 1: Header Files and Constant Definitions

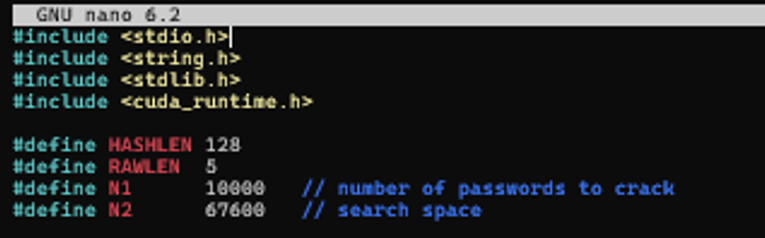


Figure : Header Files and Constant Definitions

Explanation**:** We start by pulling in key libraries like stdio.h for basic input and output, string.h for handling text like copying or trimming strings, and cuda.h to work with GPU features. Then we set up some fixed numbers: HASHLEN at 128 for the size of each hash string, RAWLEN at 5 for the short original passwords plus an end marker, N1 at 10000 for how many passwords we're trying to crack, and N2 at 67600 for the full list of possible guesses. These numbers help plan out memory use and loops, keeping things straightforward for the demo without needing changes later.

## Process 2: Device-side Memory Comparison Function

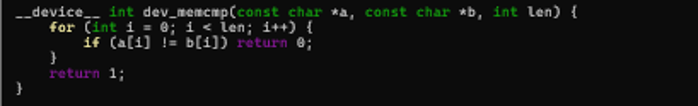


Figure : Device-side Memory Comparison Function

Explanation**:** This bit creates a simple function that runs only on the GPU to check if two strings match. It takes two pointers to text and loops through 128 characters, seeing if each spot is the same. If anything differs, it gives back 0 for no match; if all matches, it returns 1. We made this because the usual string check doesn't work on GPU hardware, so it lets us compare hashes quickly right where the work is happening, without sending data back and forth.

## Process 3: CUDA Kernel for Hash Matching



Figure : CUDA Kernel for Hash Matching

Explanation**:** Here's the core GPU function that does the heavy lifting. It gets pointers to the target hashes, the guess hashes, the original short passwords, and a place to store results. Each thread figures out its job number, checks if it's needed, and then tries every guess against its target. When it finds a match, it grabs the short password and stops. If nothing fits, it puts question marks as a note. This setup lets thousands of checks run at the same time on the GPU.

## Process 4: Thread Index Calculation and Bounds Check



Figure : Thread Index Calculation and Bounds Check

Explanation**:** Inside the GPU function, we calculate a unique number for each worker thread using the block and thread positions. Then we quickly check if that number is too big for our list of targets—if it is, the thread just quits early. This keeps everything safe and stops extra threads from wasting time or causing errors when we have an odd number of jobs.

## Process 5: Parallel Hash Comparison Loop



Figure : Parallel Hash Comparison Loop

Explanation**:** For each target, the thread runs a loop through all 67,600 guesses, using our custom check to see if the hashes line up. If they do, it moves on to save the short password. This repeating check is what makes it a search, but since it's spread across many threads, the whole thing finishes much faster than on a regular computer.

## Retrieving Original 4-Digit Password

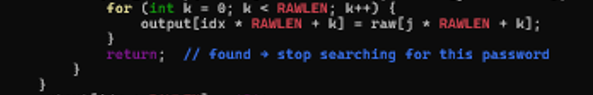


Figure : Retrieving Original 4-Digit Password

Explanation**:** When a match pops up, we copy the short password letter by letter into the results spot, using a small loop for the 5 characters including the end marker. Then the thread stops looking to save time. This step turns the found match into something useful we can read later.

## Raw Password Generation

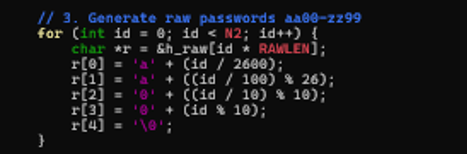


Figure : Raw Password Generation

Explanation**:** Here, we build the list of all possible short passwords in a loop from 0 to 67,599. For each spot, we figure out the letters and numbers based on math like dividing by 2600 for the first letter. We add the end marker too. This list gets sent to the GPU so it can pair matches with the right originals.

## CUDA Kernel Launch (Parallel Execution)



Figure : CUDA Kernel Launch (Parallel Execution)

Explanation**:** We decide on 256 workers per group, then figure out how many groups we need by dividing the targets and rounding them up. We start the GPU function with those settings, passing all the data pointers. After that, we wait for everything to finish with a sync call. This kicks off the parallel work and makes sure it's done before we grab the results.

## Final Output Storage

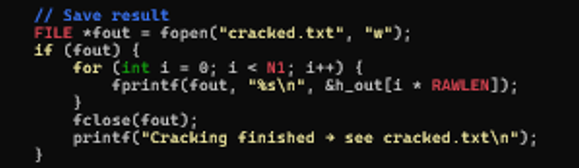
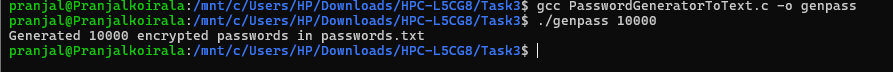
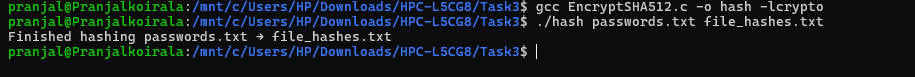


Figure :Final Output Storage

Explanation**:** Once the GPU is done, we copy the results back to the main computer. We open a file called cracked.txt to write in, then loop through each result, printing it with a new line. Close the file and clean up all the memory we used. This saves the cracked passwords in a simple text file for easy checking.

## Output:









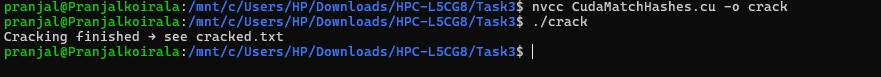


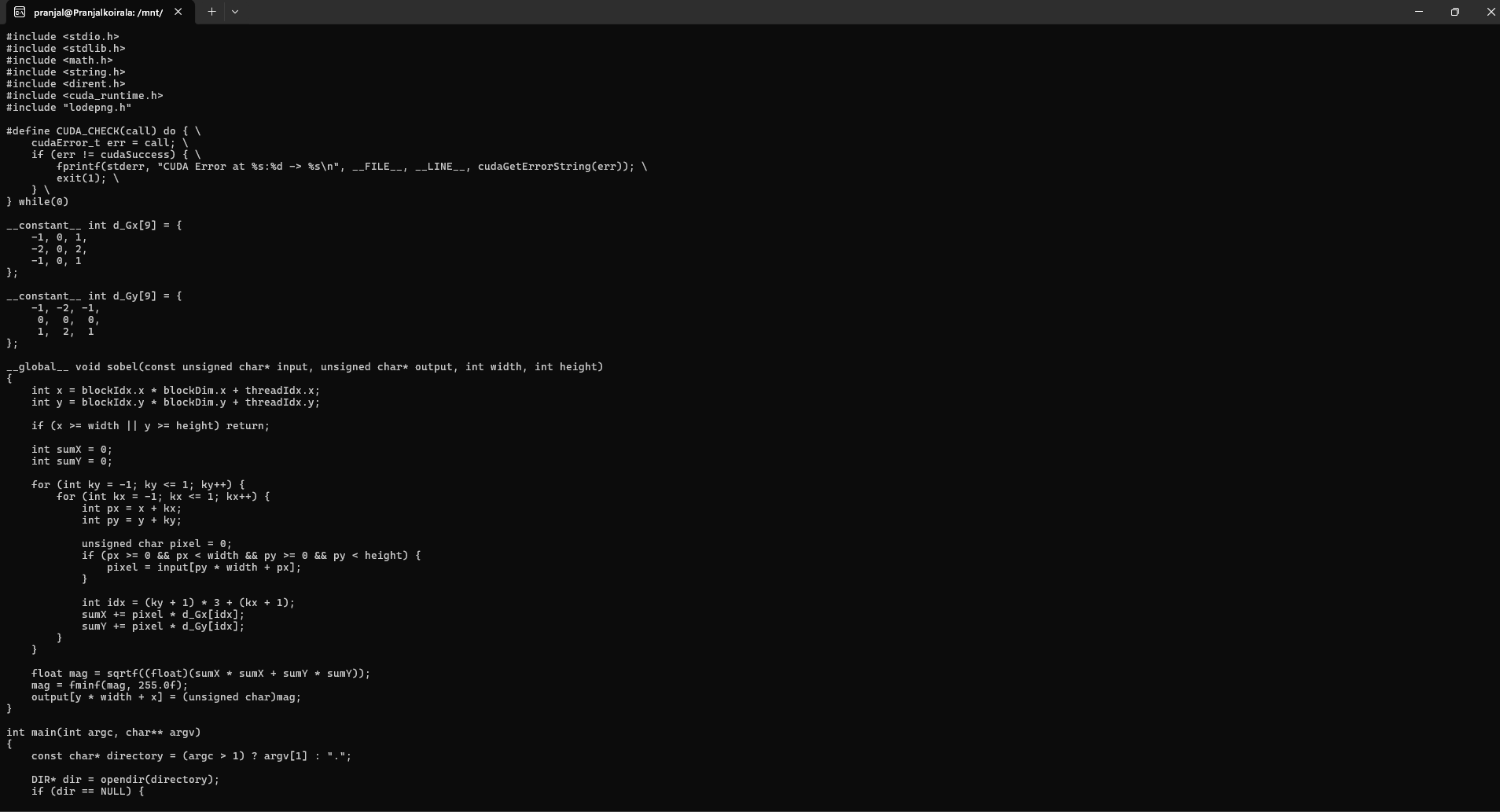


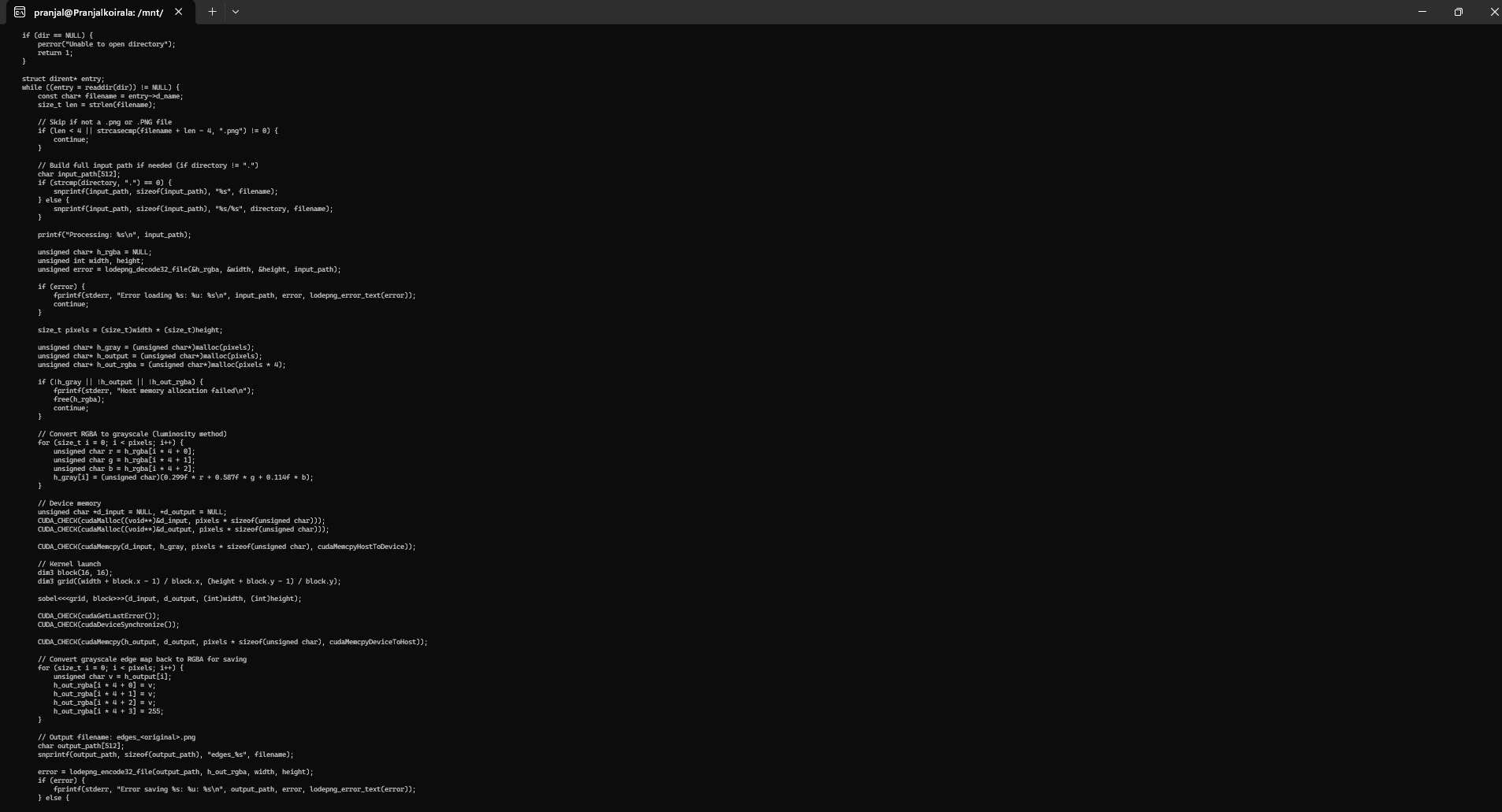


Figure : Cracked.txt Output

## Task4

## Full code: Sobel Edge Detection Using CUDA





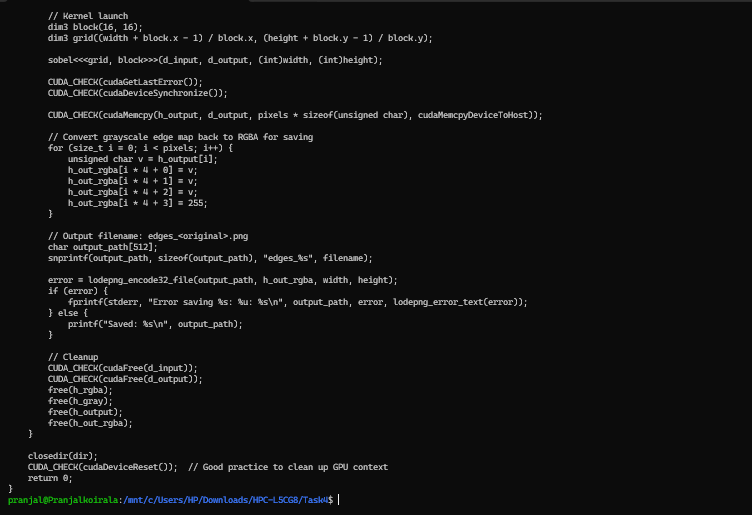
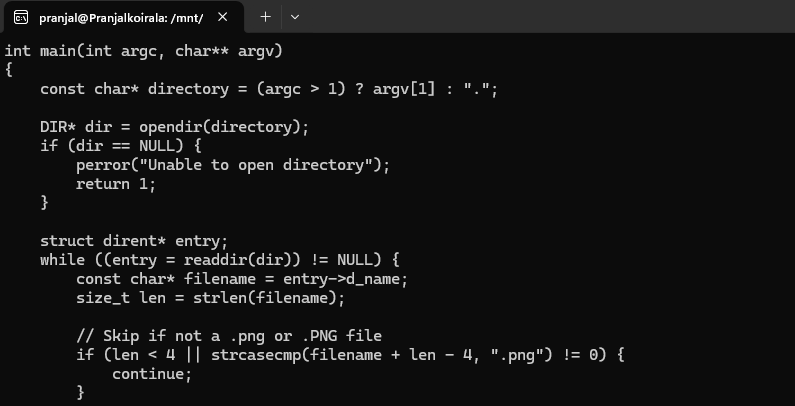


Figure : Sobel Edge Detection Using CUDA

## Main part of the code:

## Reading PNG Image(s) into CPU Memory

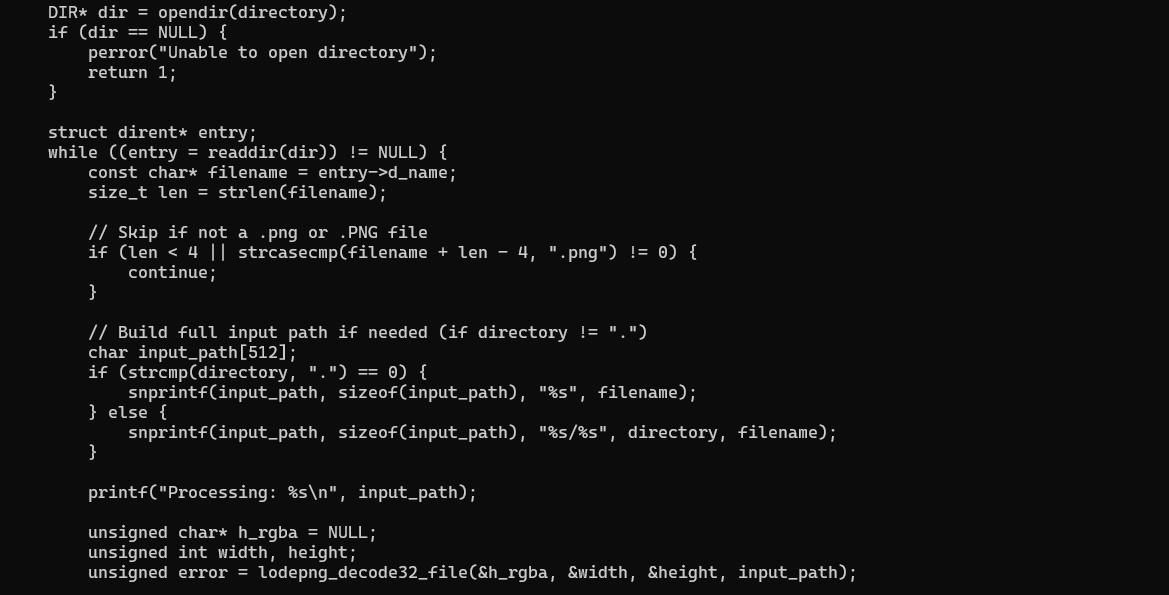


Figure : Reading PNG Image(s) into CPU Memory

Explanation**:** We use a library called libpng to load pictures. First, set up structures for reading the file and its details. Open the image file in binary mode, link it to the reader, and pull in the header info like size. Grab the width and height, then make space in memory for the pixel data. Read the whole image into rows, handling colors or gray as needed. This gets the picture ready on the regular computer before sending it to the GPU.

## GPU Memory Allocation and Data Transfer



Figure : GPU Memory Allocation and Data Transfer

Explanation**:** We ask the GPU for space to hold the input picture and the output, using sizes based on width, height, and colors. Copy the picture data from the computer to the GPU. If the image has colors, we might turn it to gray first either here or with a quick GPU step. This moves everything over so the GPU can work on it without waiting.

## Sobel Edge Detection CUDA Kernel

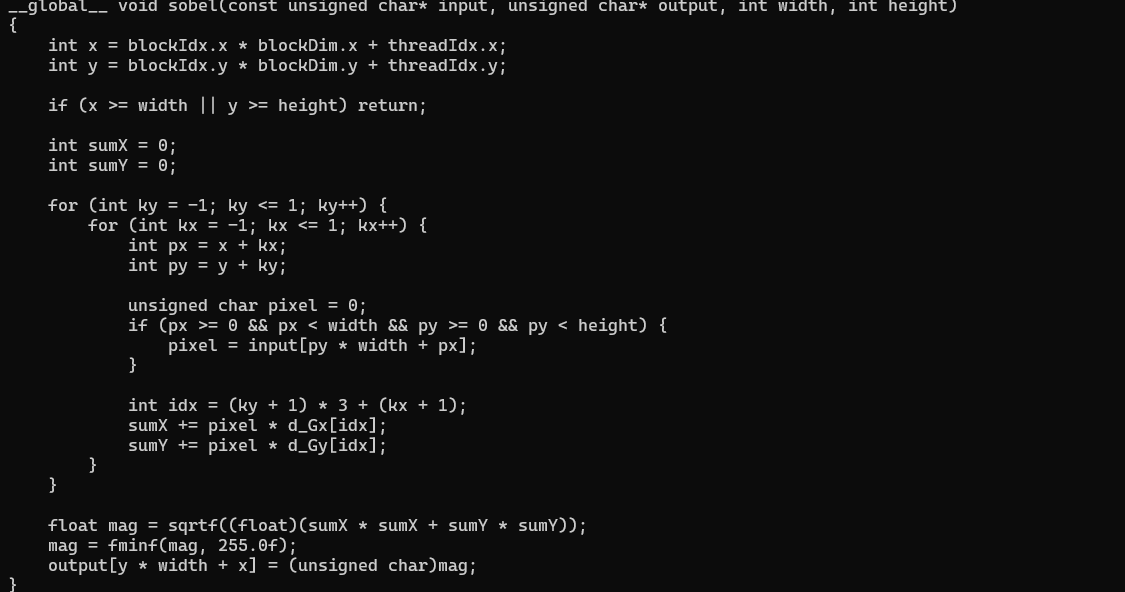
****

Figure : Sobel Edge Detection CUDA Kernel

Explanation**:** This GPU function finds edges in the picture. Each thread gets its spot from block and thread numbers, skips if outside the image. For every pixel, it looks at nearby ones using a pattern for horizontal and vertical changes, like weighting left and right differently. Calculate the strength from those, then save it as the new pixel value. Handle edges of the picture by ignoring or filling with zeros.

## CUDA Kernel Launch (Parallel Execution)

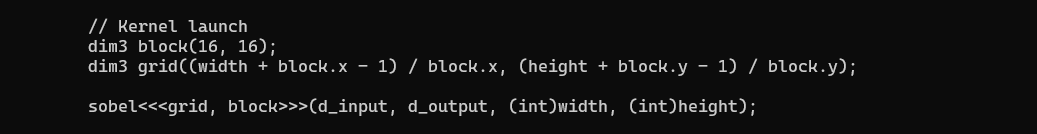


Figure : CUDA Kernel Launch (Parallel Execution)

Explanation**:** Set up groups of 16 by 16 threads, then calculate how many groups cover the whole picture size, rounding up. Start the edge-finding function on the GPU with input, output, and sizes. Wait for it to finish with a sync. This spreads the work across the GPU, so each part of the picture gets processed at the same time.

## Copy Result Back and Write Output Image

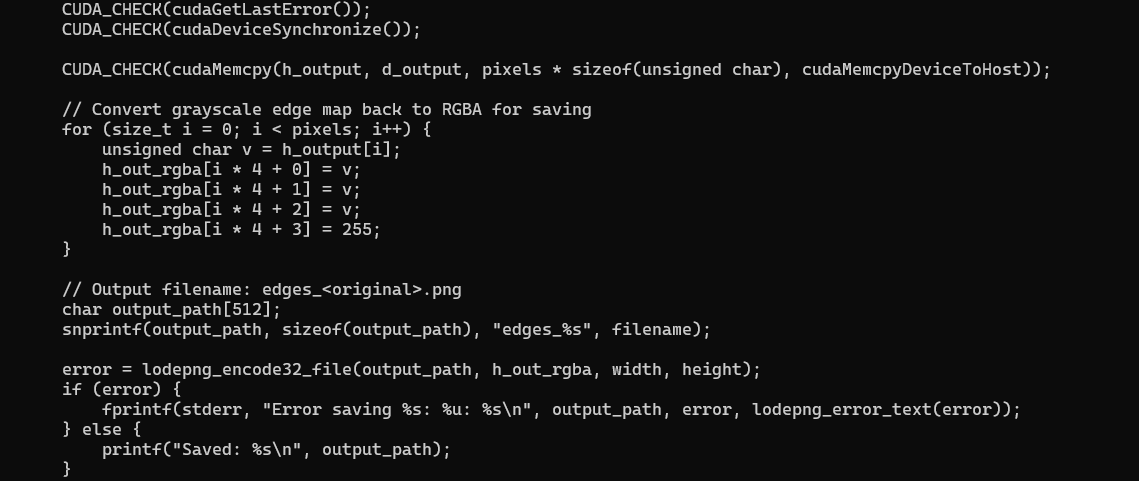


Figure : Copy Result Back and Write Output Image

Explanation**:** Pull the finished edge map from GPU back to the computer. Set up a writer for the new PNG, copy over size and type info. Open a file to save, write the header, then the pixel rows, and finish up. This turns the GPU results into a viewable picture file.

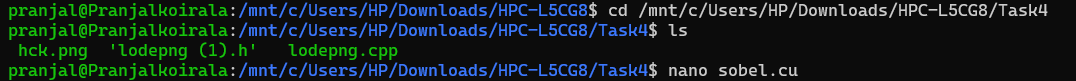
## Memory Deallocation

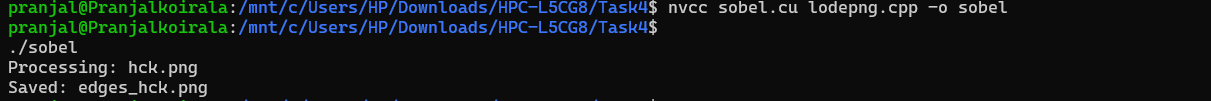


Figure : Memory Deallocation

Explanation: Free up the GPU space for input and output. Release the computer memory for the pictures. Clean up the library structures for reading and writing. Shut the files if they are still open. This clears everything to avoid using extra resources or causing leaks.

## Output:





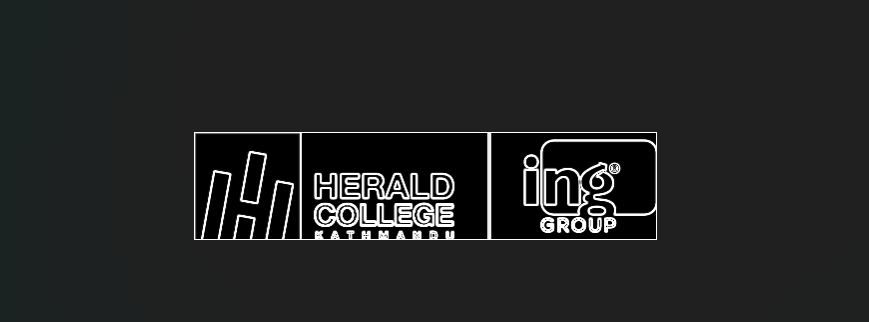


Figure : Output of image hck.png