MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

PCAP: Mini Project Report on

Parallel Encryption, Cracking and Decryption: A CUDA Approach to the Vigenère Cipher

SUBMITTED BY

SUJEET SANJAY AMBERKAR 200905092

Under the Guidance of:
Mrs Neelima Bayyapu
Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal Karnataka – 576104

April 2024

Index

Abstract:

This project explores the encryption cracking and decryption of Vigenère cipher using the power of parallel computing. This study leverages the Compute Unified Device Architecture (CUDA) to implement and optimize the encryption, decryption, and cryptanalysis of the Vigenère cipher on NVIDIA GPUs. The results show a significant increase in the speed of cryptoanalysis when done using CUDA.

Introduction:

In the field of cryptography, the ability to process large volumes of data with high efficiency and speed is very important. CUDA's architecture is designed for Single Instruction Multiple Data. Cryptographic algorithms for Encryption, Decryption and cracking of ciphers often involve similar or identical operations across all data points making it an ideal fit for using CUDA to increase the speed. The Shared memory also helps us to resources between threads thus reducing the latency and increasing the bandwidth. Importantly, in cryptographic systems such as the Vigenère cipher, the size of the key space expands exponentially with the length of the key, following a 26^(size of the key) pattern. This immense key space underscores the need for CUDA's powerful parallel processing capabilities, as it enables the exploration of vast cryptographic landscapes within feasible timeframes,

Project Objective

The Vigenère cipher is a method of encrypting alphabetic text where each letter of the plaintext is encoded with a different Caesar cipher. It is an evolution of the Caesar cipher in which we conduct a series of different shifts or keys rather than just a single key in the Caesar cipher.

Core Objecetives

1) CUDA- Enabled Vigenère Encryption and Decryption: This involves developing a framework using CUDA for performing Encryption and Decryption taking advantage of GPU Architecture and Shared Memory in CUDA.

2) **Cryptanalysis Techniques:** Implement advanced CUDA-based implementation of the Kasiski algorithm to guess the key length and then find the key of Encrypted text using Frequency Analysis by parallel computing.

3) **Comparative Performance Analysis**: Conduct a thorough performance analysis comparing the CUDA-accelerated Vigenère cipher operations and cryptanalysis methods against traditional CPU-based implementations. This comparative study aims to highlight the efficiency and speed gains achieved through GPU parallelism**,**

Understanding CUDA

1) **Parallel Execution with CUDA Kernels:** The CUDA kernel such as “repeatString”, “vignere_encrypt_cuda”, ”crack_vigenere_kernel”, ”vignere_decrypt_cuda”, “calculate_matches” leverages the power of Single Program Multiple Data by performing similar instructions across multiple data points

2) Efficient Memory Usage: The use of `__constant__` memory types, as seen with d_inputString and d_key, optimizes memory. **cudaMemcpyToSymbol** is specifically used to copy the encryption key and decryption key to these memory spaces. Ensuring all threads have quick access to these constants.

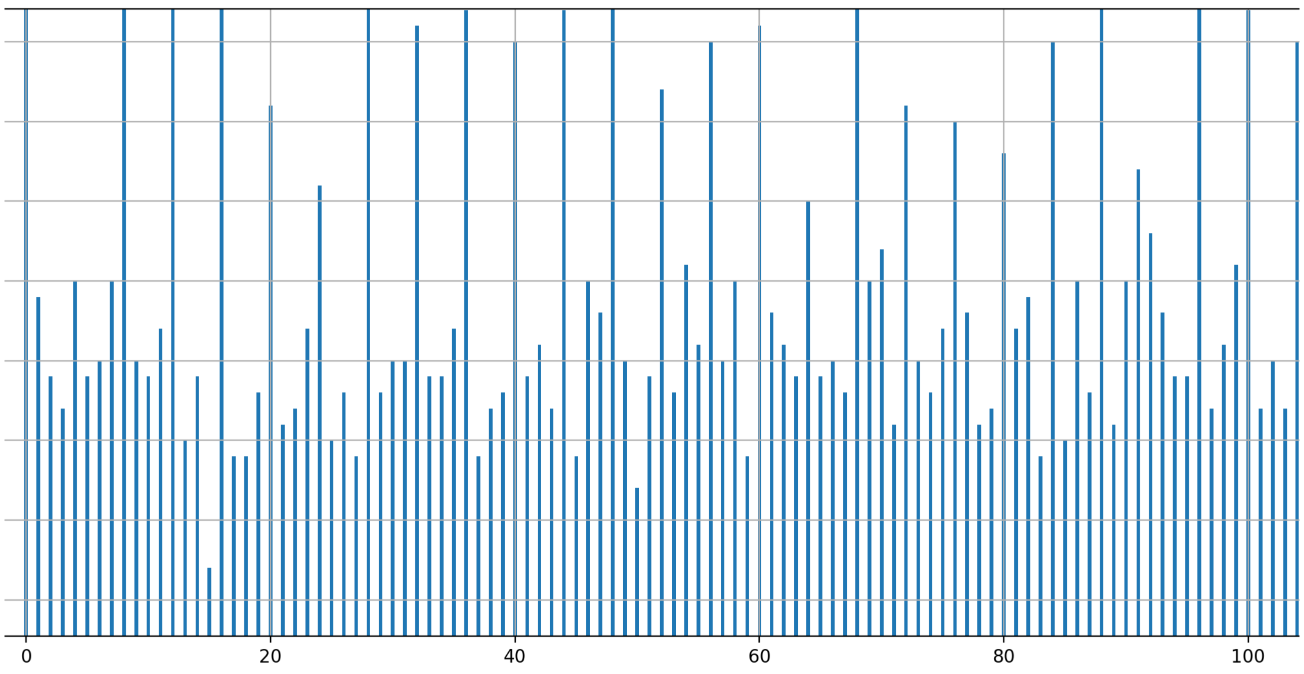
3) **Optimal Resource Utilization**: The Effective Kernel launch configurations (<<<numBlocks, blockSize>>>) for the CUDA process.

Selecting Algorithm

1)Encryption:
$E_i(x_i) = (x_i + ki) \bmod 26$
Each character in the plaintext is shifted according to the corresponding character in the key. This operation, due to its simplicity and repetitive nature across the length of the plaintext, is ideally suited for GPU's SIMD. Allowing encryption of large texts in parallel with significant speed improvements

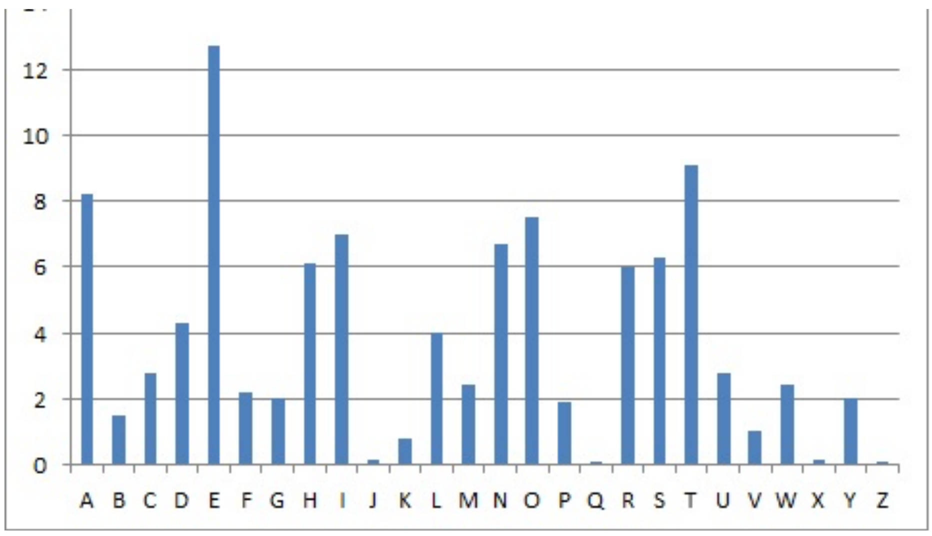
2) Decryption:
$E_i(x_i) = (x_i - ki) \bmod 26$
This is exactly the inverse of Encryption. Similar to encryption, decryption can be efficiently parallelized over the GPU, with each thread handling the decryption of individual characters or blocks of text, thus benefiting from CUDA's parallel processing capabilities.

3) **Key Length Determination (Kasiski Examination):** This method involves analyzing the ciphertext to find repeating sequences of characters and measuring the distances between these repetitions. Since we shift and repeat the process again, CUDA can significantly reduce the time required by performing these analyses in parallel across multiple segments of the text.



4) **Frequency Analysis:** Once the key length is known, the cipher can be treated as multiple Caesar ciphers. Frequency analysis is then applied to each segment in parallel to get the key.

The chosen algorithms are inherently parallelizable, making them excellent candidates for CUDA's architecture



Implementing CUDA Kernels:

1) Encryption Kernel(vignere_encrypt_cuda**):**
This kernel encrypts input text using the Vigenère cipher. Each thread encrypts a single character of the input string, calculating the corresponding encrypted character by shifting it according to the key. The kernel is designed to handle each character independently, enabling the encryption process to scale efficiently with the length of the text.

```
    int blockSize = 256;

    int numBlocks = (msgLen + blockSize - 1) / blockSize;

    vignere_encrypt_cuda<<<numBlocks, blockSize>>>(d_message, d_encrypted, msgLen, keyLen);
```

2)Decryption Kernel(vignere_decrypt_cuda**):** The Decryption process in exactly inverse of the encryption process. Thus this follows the same kernel design as the Encryption

```
    int blockSize = 256;

    int numBlocks = (msgLen + blockSize - 1) / blockSize;

    vignere_decrypt_cuda<<<numBlocks, blockSize>>>(d_encrypted, d_decrypted, msgLen, keyLen);
```

3)Finding Length (calculate_matches):
The CUDA kernel calculate_matches and the function guess_key_length_cuda are central to the implementation of an efficient approach to determining the key length in the cryptanalysis of the Vigenère cipher.
The calculate_matches kernel is designed to calculate the number of matching characters for each possible shift value across the cipher text. Each thread handles a different shift value, iterating through the cipher text and incrementing a counter whenever a character matches its shifted counterpart.

```
    int threadsPerBlock = 256;

    int blocks = (length + threadsPerBlock - 1) / threadsPerBlock;

    calculate_matches<<<blocks, threadsPerBlock>>>(d_cipher, length, d_matches);
```

4)Generating Key(crack_vigenere_kernel):
Once the length of the key is known, the crack_vigenere_cuda function and crack_vigenere_kernel break the encrypted text into n Caesar Ciphers and Threads iterate through their assigned sections, counting the frequency of each letter. The frequency analysis aims to identify the most common letter in each segment, under the assumption that it represents the encrypted form of the letter 'E'—the most frequently occurring letter in English text.

```
int threadsPerBlock = 256;

int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

crack_vigenere_kernel<<<blocksPerGrid, threadsPerBlock>>>(dev_cipher, cipher_len, n, dev_keys);
```

Optimisation Strategies

1)Minimizing Global Memory Access Latency: The kernel accesses the cipher text stored in global memory. To reduce the latency, the text is accessed sequentially by threads, which is beneficial for memory coalescing, where the GPU can efficiently load contiguous memory segments into the cache, reducing the number of required memory transactions.

2)Balancing Load Across Threads: The workload is evenly distributed among threads to ensure all are actively computing matches for different shift values, minimizing idle time and maximizing hardware utilization.

3)Using Atomic Operations for Counting Matches: To avoid race conditions and ensure correct match counts in parallel executions, atomic operations could be utilized when incrementing the match counters in global memory.

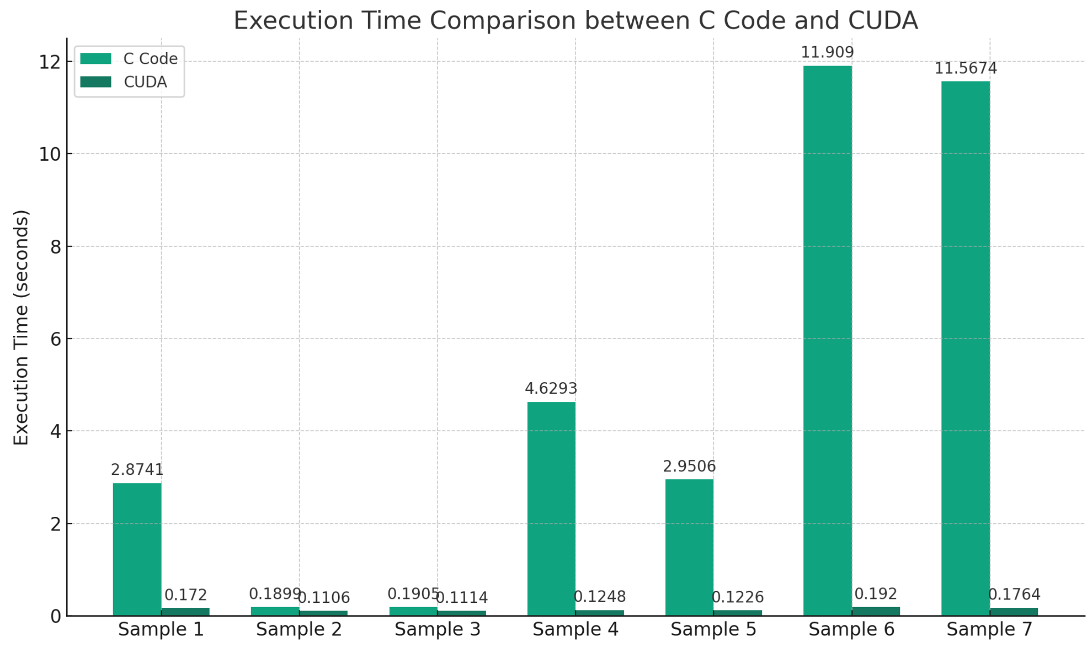
Memory Management:

1)Wherever Possible use global memory and cudaMemcpyToSymbol when the same resources need to be accessed by all threads, especially for data that needs to be accessed by all threads, such as the encryption key during encryption and decryption processes. This approach ensures efficient use of the GPU's memory hierarchy.

2) **Resource Cleanup:** Free the Cuda Memory when the job is done

Execution Time Measurements:

C Code	Nvidia	Key	Length of Sample Text
2.874124	0.171985	SUJEET	3059
0.189914	0.110613	SUJEETAM	745
0.190495	0.111438	SUJEET	745
4.629269	0.124841	SUJEET	3737
2.950582	0.122647	SUJEET	2983
11.909002	0.191981	SUJEET	5971
11.567394	0.176444	SUJEETAM	5971



Bottlenecks and Limitations:

1)**GPU Utilization**: For smaller workloads, the overhead of initializing CUDA kernels and managing GPU resources may not be fully offset by the gains in parallel execution, leading to smaller speedups.
2) **Limitation of Kaiski-algorithm** : The project depends on finding the Kaiski-algorithm for finding the length of the key but as the length of the key increases the efficiency of the Kaiski-algorithm decreases. So all the limitations of Kaiski algorithm are also inherited in this code.

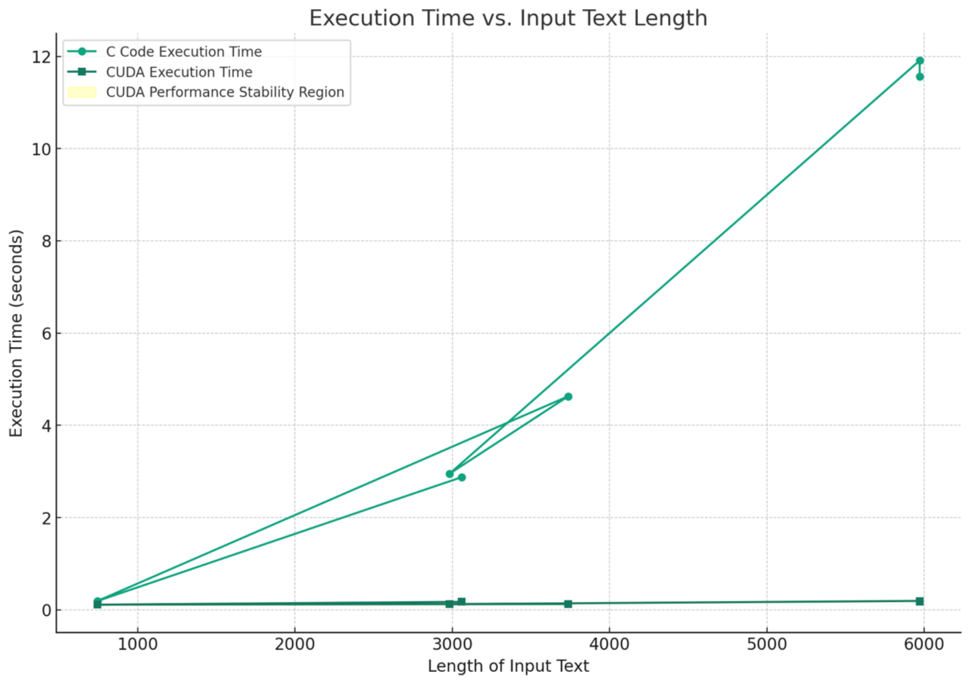
References:

1) Impact of Computational Power on Cryptography by Bhat, M.I., Giri, K.J. (2021) DOI: 10.1007/978-981-15-8711-5_4
2) Analyzing the Kasiski Method Against Vigenere Cipher by April Lia Hananto DOI: 10.48550/arXiv.1912.04519
3) https://youtu.be/QgHnr8-h0xI

Conclusion

C Code Execution Time: There's a noticeable trend that as the length of the input text increases, the execution time also increases, particularly for larger input sizes. This showcases the linear or super-linear scaling behaviour typical of CPU-based sequential processing

CUDA Execution Time**: In contrast, the CUDA execution times remain relatively stable across different input sizes. Despite minor fluctuations, the execution time does not significantly increase even as the input size grows**


Execution Time vs. Input Text Length
C Code Execution Time
CUDA Execution Time
CUDA Performance Stability Region
Execution Time (seconds)
Length of Input Text