Universiteit van Amsterdam

Lab 2

Lab 2: Cuda

28 november 2023

Student:
Naeem Almawladi
13887793

Cursus:
Distributed and Parallel Programming

1 Design and Implementation

1.1 Wave Equation

1.1.1 Implementation

The implementation of the 1D wave equation simulation using CUDA involves several key components, designed to leverage the parallel processing capabilities of a GPU. We start with CUDA Kernel (simulateKernel): This function, executed on the GPU, calculates the wave's amplitude at each point in space for the next time step. It uses the grid and block dimensions to determine the unique index of each thread (i). For each point i (excluding boundary points), the new amplitude is calculated based on the current and previous amplitudes at i, and the amplitudes at neighboring points (i-1 and i+1). Boundary points are set to zero.

Further more the code allocates memory on the GPU for three arrays - deviceOld, deviceCurrent, and deviceNext - which represent the wave at two consecutive time steps and the next time step, respectively. Data is transferred between the host (CPU) and the device (GPU) using cudaMemcpy.

The main loop iterates over time steps, calling the simulateKernel. After each iteration, the pointers for the wave arrays are rotated to prepare for the next time step.

To handel errors we use the utility function checkCudaCall checks for and handles CUDA errors, ensuring robust execution

1.1.2 Important Choices

their is 2 important choices the first one using three arrays to represent different time steps and rotating them, memory usage is optimized, avoiding unnecessary data duplication. the second one is implementation of error handling to catch and report any issues during CUDA operations, which is crucial for debugging and stability.

```
Function simulateKernel(oldWave, currentWave, nextWave, i_max, constant):
    i = unique_thread_index
    if i within boundaries:
        nextWave[i] = compute_new_amplitude(oldWave, currentWave, i, constant)
    else if i at boundary:
        nextWave[i] = 0

Function simulate(i_max, t_max, block_size, old_array, current_array, next_array):
    Allocate memory on GPU
    Copy initial data to GPU

    for t from 0 to t_max:
        Call simulateKernel in parallel
        Synchronize CUDA device
        Rotate wave arrays for next time step

    Copy final data back to host
    Free GPU memory
    Return final wave state
```

The pseudocode outlines the key steps in the simulation process. Each iteration of the simulation loop involves calling the simulateKernel function on the GPU, which calculates the next state of the wave at each point in parallel. Post-calculation, the wave state arrays are rotated to prepare for the next iteration. This design effectively utilizes the parallel processing power of GPUs, making it well-suited for simulations involving large datasets and computations.

1.2 Parallel cryptography in CUDA

1.2.1 Implementation

The implementation encompasses both sequential and parallel approaches for encryption and decryption, utilizing CUDA for efficient parallel processing. In this system, output at each point is computed by adding the key to the corresponding input; for the Vigenère cipher, where the key is a string, it's indexed modulo the key's length. Decryption involves subtracting the key. The parallel implementation processes each output index in a separate thread. Specifically, CUDA is employed for its GPU parallelization capabilities, suitable for large text handling through Shift and Caesar ciphers. Key components include encryption and decryption kernels (encryptKernel and decryptKernel), which execute on the GPU, processing each character individually. Additionally, sequential implementations (EncryptSeq and DecryptSeq) are provided for comparison, running on the CPU. The system also manages memory allocation and data transfer between CPU and GPU efficiently and includes robust error checking with checkCudaCall. The main function serves as the hub for managing inputs, file operations, and orchestrating the overall encryption and decryption processes.

```
Function encryptKernel(deviceDataIn, deviceDataOut, deviceKey, keyLength):
    i = blockIdx.x * blockDim.x + threadIdx.x
    keyIndex = i % keyLength
    c = deviceDataIn[i]
    key = deviceKey[keyIndex]
    Encrypt character c using key

Function decryptKernel(deviceDataIn, deviceDataOut, deviceKey, keyLength):
    Same structure as encryptKernel, but performs decryption

Function EncryptSeq(n, data_in, data_out, key_length, key):
    For each character in data_in:
        Encrypt using Shift cipher
    Output encrypted data to data_out

Function DecryptSeq(n, data_in, data_out, key_length, key):
    For each character in data_in:
        Decrypt using Shift cipher
    Output decrypted data to data_out

Function EncryptCuda(n, data_in, data_out, key_length, key):
    Allocate memory on GPU
    Copy data and key to GPU
    Call encryptKernel
    Copy encrypted data back to host
    Free GPU memory

Function DecryptCuda(n, data_in, data_out, key_length, key):
    Same structure as EncryptCuda, but for decryption

Function main():
    Parse command-line arguments for key
    Read input data
    Perform encryption and decryption using both sequential and CUDA methods
    Write output data to files
```

This pseudocode simplifies the essential operations in the code, focusing on the major functions and their roles in the encryption and decryption processes. The CUDA kernels (encryptKernel and decryptKernel) are the central elements where parallel processing takes place. The sequential functions (EncryptSeq and DecryptSeq) provide a baseline for comparison, while the EncryptCuda and DecryptCuda functions orchestrate the entire process of encryption and decryption using CUDA. The main function acts as the controller, orchestrating the overall workflow, including file handling and key management.

1.3 Resutlts

1.3.1 Wave Equation

i (Amplitude)	Execution Time
1,000	252 ms
10,000	276 ms
100,000	283 ms
1,000,000	439 ms
10,000,000	2.07 s

(a) Block size of 512

i (Amplitude)	Execution Time
1,000	265 ms
10,000	211 ms
100,000	172 ms
1,000,000	384 ms
10,000,000	2.1 s

(b) Block size of 32

i (Amplitude)	Execution Time
1,000	200 ms
10,000	210 ms
100,000	219 ms
1,000,000	383 ms
10,000,000	2.14 s

(c) Block size of 64

i (Amplitude)	Execution Time
1,000	208 ms
10,000	233 ms
100,000	253 ms
1,000,000	399 ms
10,000,000	2.3 s

(d) Block size of 128

i (Amplitude)	Execution Time
1,000	192 ms
10,000	239 ms
100,000	267 ms
1,000,000	377 ms
10,000,000	2.14 s

(e) Block size of 256

i (Amplitude)	Execution Time
1,000	189 ms
10,000	212 ms
100,000	284 ms
1,000,000	369 ms
10,000,000	2.13 s

(f) Block size of 1024

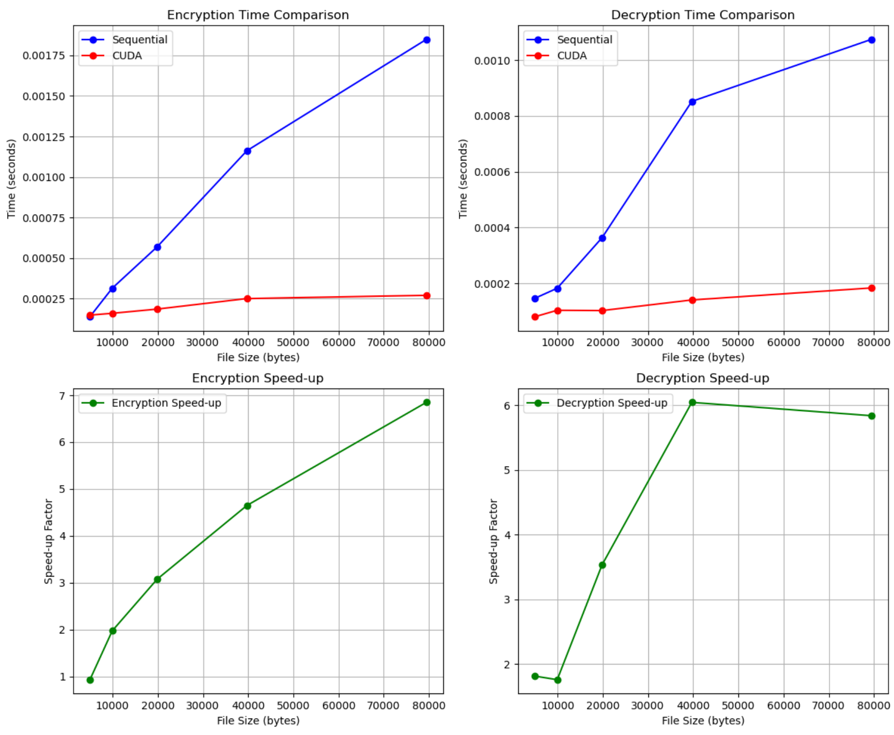
Figuur 1: Execution times for different problem sizes with various block sizes.

1.3.2 Conclusion

CUDA outperforms OpenMP and pthreads in several aspects, particularly in parallelism granularity, hardware utilization, memory bandwidth, and specialized computation. It excels in fine-grained parallelism, making it ideal for tasks like wave simulation where operations can be divided into thousands of smaller, independent tasks. CUDA's design is tailored for GPUs, which possess more cores than CPUs, allowing for a greater number of parallel threads and significantly faster computation. Additionally, GPUs have higher memory bandwidth compared to CPUs, facilitating quicker data transfers, crucial for handling large data sets in applications like wave simulation. While OpenMP and pthreads offer flexibility for general-purpose parallel computing on CPUs, CUDA is specifically optimized for high-throughput computing tasks on GPUs, providing enhanced performance for such applications.

1.3.3 Parallel cryptography in CUDA

To test it i duplicate the input file orginal.data 5 times sizes (4986, 9927, 19853, 39707, 79416)



Figuur 2: Caption for the image.

As the file size increases, the speed-up factor for encryption also increases. This trend indicates that the GPU version performs increasingly better relative to the sequential version as the file size grows. The largest file size (79,416 bytes) shows the highest speed-up, suggesting that the GPU's parallel processing capabilities are more effectively utilized with larger data.

1.3.4 Conclusion

GPU-accelerated encryption and decryption (CUDA) offer significant speed-ups over sequential processing, especially as file sizes increase. The performance gains are more pronounced for larger files, highlighting the effectiveness of parallel processing in GPUs for handling large-scale data encryption and decryption tasks. This analysis suggests that for applications dealing with large files, implementing GPU-based encryption and decryption can lead to substantial performance improvements.