Matriculation Number: 40646460

Concurrent & Parallel Systems

Coursework: 1

Contents

[Introduction 2](#_Toc181788913)

[1. Converting the original program 3](#_Toc181788914)

[1.1 Determining the need for parallelism 3](#_Toc181788915)

[1.2 Transforming the program with CUDA 3](#_Toc181788916)

[1.2.1 Changing the token search function 3](#_Toc181788917)

[1.2. Creating the kernel function 4](#_Toc181788918)

[2. Optimising performance 5](#_Toc181788919)

[3. Performance 6](#_Toc181788920)

[References 7](#_Toc181788921)

[Appendix 8](#_Toc181788922)

# Introduction

This report explores the procedure of converting a C++ program for graphics processing unit (GPU) parallel processing using Computer Unified Device Architecture (CUDA). Arguably, GPU offers performance advantages for objectives with high levels of parallelism, like text search algorithms. In this report, performance times of the central processing unit (CPU) and GPU are compared, highlighting the beneficial usage of parallelism. Additionally, a detailed explanation of how the C++ algorithm was ported to CUDA is shown, breaking down each step of the process. Finally, figures and test cases are demonstrated to show the performance improvement.

# Converting the original program

## 

## 1.1 Determining the need for parallelism

The CPU is not efficient with performance for repeatable tasks. The task of searching text for words is one that can be done in parallel using the GPU, and to perform this conversion, CUDA would suffice.

## 

## 1.2 Transforming the program with CUDA

The first step was figuring out which part of the program could be used for a GPU Kernal. The answer was the token search function because it repeats with the usage of a control loop. This means the read file function will not be changed.

### 

### 1.2.1 Changing the token search function

The token search function again takes in two parameters, one   
 for a vector containing the file contents and a second for the character tokens   
 (words). The length of the two parameters is stored.

Next, the data is initialised with cudaMalloc which allocates memory on the   
 device memory for the file data, word token, and the result. This is due to the   
 GPU not having access to data stored on host memory, which means the data   
 needs to be copied from the CPU (host) to the GPU (device). To achieve this,   
 cudaMemcpy is used to copy each host data to the device. This is declared by   
 using the cuda function: cudaMemcpyHostToDevice.

To utilise the GPU, the program must now call a GPU kernel to perform the search. First, the block size and number of threads must be defined. To start, a thread size of 256 is set as this is a standard size. The number of blocks is then done through the calculation (data length + thread size – 1) % number of threads. This calculation ensures that there are the right number of blocks for the data length. The kernel is then called using the CUDA kernel call parameters <<< >>> with the block size and thread number inside the parameters. The device copies of the data are also sent as typical C++ pointers.

Once the kernel has finished, the result is returned from the GPU stored in the device memory. Because the result is stored in the device it must be copied back to the host to be shown on the CPU. This is again performed through cudaMemcpy, plus the cuda function: cudaMemcpyDeviceToHost. For good practice, once there is no need for the data to be held on the device memory, cudaFree function is used to free up the memory. Finally, the result that has now been copied to the host is returned to the main function.

### Creating the kernel function

To create a CUDA kernel, the keyword \_\_global\_\_ tells the complier the function is a GPU kernel. The kernel takes in the following parameters: the file containing the data, the length of the data, the tokens (words), the length of the tokens, and an integer for the result.

The ability to perform the search in parallel is done through threads having access to unique parts of the data at the same time, therefore each thread needs a unique index. Consequently, the CUDA function .x is used to track the block dimension, the block index, and the thread index. The calculation is blockIdx.x \* blockDim.x + threadIdx.x. This index allows each thread to work on a unique index allowing parallel processing. Before starting, the program checks to see if the index plus the token length is greater than the file length. If true, the program terminates, as this would result in threads operating out of bounds. To compare tokens to the file, the code was split into different functions to maintain modularity. Unlike CPU (C) programming, normal functions cannot simply be declared, as the GPU cannot access them. To fix this, two device functions (declared \_\_device\_\_) are used, acting as GPU functions.

The first device function checks for non-letters, and returns true if the character is not lower case. The second device function is the method of comparing tokens. This function loops around the length of the current token and performs an if statement to see if the data and token do not match. If true, the program returns a false statement. As this function is repeatedly called by threads, containing it inside a device function stops duplicated code. Due to the kernel not having the implementation, an if statement calls the compare token device with the condition of returning true if a match is found. If true, then a boundary check runs to ensure the match is valid (in bounds) against potential errors such as blank space or the word actually being apart of a larger word (e.g., “cat” being found in “catalogue”). This boundary check is done by calling the second device function. If the kernel finds a match in the data and that match is valid, the result is stored. Consequently, atomic add (atomicAdd) was used.  
The need to use atomic operations is because multiple threads will try and update the counter at the same time, and this could be problematic if left unsupervised; however, atomic add allows for management of this by only letting a single thread access the variable at one given time. This results in the counter being updated correctly, which results in a correct counter.

# Optimising performance

## 2.1 Tracking performance

### 2.1.1 CPU Tracking

To track the timing of the CPU performance, chrono library’s high\_resolution\_clock function is used to record the start and end time of the program. These timestamps are then subtracted to calculate the overall duration of the program, which is then displayed in milliseconds to the console.

### 2.1.2 GPU Tracking

To track the GPU timings, CUDA events are needed. CUDA events provide precise tracking capabilities. Similar to the CPU, two events are needed, one to track the start time and one to track the end time. The function cudaEventCreate and cudaEventRecord are used to create these. After the GPU completes its operations, cudaEventSynchronize ensures that all GPU activities are finished before the end time is recorded. Finally, to calculate the total time taken by the GPU, the recorded start and end time is added by cudaEventElapsedTime.

### 2.1.3 Comparing the times

To compare the two saved times, an if statement is used to determine which recorded time is shorter, indicating the faster method. Additionally, the difference in speed is also calculated and shown.

## 2.2 First iteration of the kernel

Originally, the kernel was unoptimized with a default thread and block size. These were set to 1 block with 100 threads per block. This configuration did lead to a fast GPU search time of 39ms; however, the search failed to pick up the correct number of occurrences. This failure was due to the block size not being large enough to search the whole file, leading to most of the words not being searched for.

## 2.3 Configuration for optimal block size

The issue that presented itself was the block size not being large enough to search through the whole file. To fix this, a calculation was used to dynamically set the block size in accordance with the length of the data it would be searching. The calculation was as follows:

Block Size = data length + number of threads - 1   
 % number of threads

This formula ensured that the block size was large enough to search through the whole file, resulting in all occurrences being tracked. The benefit of having a dynamically assigned block size is the result of using only what is required, thereby saving memory and optimising performance.

2.4 Configuration for optimal thread number

As the block size was then dynamically assigned, finding a thread number that best suited the hardware came from rigorous testing of different values. Following a standard thread allocation, the test values were all powers of 2 (128, 256, 512). All these values worked well because the block size was optimal; however, 256 proved to be the best for performance due to 256 being balanced value in terms of memory usage and occupancy.

# 3. Performance analysis

## 3.1 Hardware setup

Hardware specifications: testing was done on a NVIDIA GeForce RTX 3060 Laptop GPU, with a memory size of 6143MB, Warp Size of 32, and 30 Multiprocessors.

## 3.2 Test Cases

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case** | Thread Size | Block Size | Average (of three) CPU Time (ms) | Average (of three) GPU Time (ms) | Notes: |
| **1:** | 100 | 1 | 466ms | 39ms | The GPU version did not find most of the correct occurrences |
| **2:** | 1012 | 10 | 416ms | 38ms | The GPU version did not find most of the correct occurrences, but more than test case 1 did. |
| **3:** | 256 | data length + number of threads - 1 % number of threads | 445ms | 43ms | Occurrence count matches CPU version  (fastest) |
| **4:** | 128 | data length + number of threads - 1 % number of threads | 447ms | 45ms | Occurrence count matches CPU version |
| **5:** | 512 | data length + number of threads - 1 % number of threads | 441ms | 45ms | Occurrence count matches CPU version |

## 3.3 Figures

A graph of a graph

Description automatically generated with medium confidence

A graph with a blue line

Description automatically generated

## 3.4 Discussion of findings

From this performance assessment, the optimal configuration was a thread size of 256 alongside a dynamically set block size based on the size of the data. This not only generated the correct number of occurrences, but also provided the most optimal performance speed.

The testing process followed a rule of three speeds to be recorded for each thread size (see Appendix), then an average ms speed was calculated to get the most accurate recording. As the charts created using Microsoft Excel show, the difference between the different thread sizes are not significant due to the block size being configured in a way to dynamically generate the optimal size regardless of thread count. The discovery of 256 being the best for performance was merely an added benefit; the real optimisation comes from the block size.

# Appendix

***Test Cases (1):***

1. CPU = 428ms GPU = 41ms

2. CPU = 457ms GPU = 34ms

3. CPU = 512ms GPU = 42ms

*CPU average of three =* 466ms

*GPU average of three = 39ms*

***Test Cases (2):***

1. CPU = 408ms GPU = 40ms

2. CPU = 422ms GPU = 37ms

3. CPU = 418ms GPU = 38ms

*CPU average of three = 416ms*

*GPU average of three = 38ms*

***Test Cases (3):***

1. CPU = 416ms GPU = 40ms

2. CPU = 497ms GPU = 43ms

3. CPU = 423ms GPU = 46ms

*CPU average of three = 445ms*

*GPU average of three = 43ms*

***Test Cases (4):***

1. CPU = 429ms GPU = 44ms

2. CPU = 443ms GPU = 44ms

3. CPU = 470ms GPU = 48ms

*CPU average of three = 447ms*

*GPU average of three = 45ms*

***Test Cases (5):***

1. CPU = 457ms GPU = 46ms

2. CPU = 426ms GPU = 43ms

3. CPU = 440ms GPU = 46ms

*CPU average of three = 441ms*

*GPU average of three = 45ms*