Concurrent and Parallel Systems

40600630 - Joshua Mendez

Coursework 1 Report

This report examines the acceleration of a word occurrence detection algorithm using GPU parallelisation, achieved by porting a given CPU-based algorithm to the GPU and optimising kernel configurations in CUDA. Performance comparisons between the serial CPU, unoptimised GPU and optimised GPU versions will be provided. Results demonstrate that the GPU implementation significantly reduces execution time, with notable improvements after optimisation. The results highlight the effective use of GPU parallelism for this task.

1 Porting the Algorithm to GPU

1.1 CPU

The original algorithm implemented performs word occurence detection by searching text files for specific target words. This CPU version involves looping through each character in the text data to check if it matches the beginning of a target word. If a match is detected, further checks are performed to ensure it is a stand-alone word, confirming that it is not part of a larger word. This algorithm processes one position at a time and has a runtime that increases linearly with text length.

1.2 Porting Strategy

To exploit GPU parallelism, the algorithm was ported to the GPU using CUDA which enables parallel execution of operations across many threads. CUDA supports programming in a style similar to c++, making it easier to adapt a GPU-based algorithm. The primary goal was to assign each character position to a different thread on the GPU, allowing each thread to compare a unique position in the text to the target word, thereby enabling simultaneous comparisons.

1.3 Implementation of the CUDA Kernel

The **find_occurrences** kernel function was designed to parallelise the word occurrences detection algorithm by distributing the work across many threads on the GPU. This parallelisation is achieved by launching a large number of threads, with each thread responsible for examining a specific position in the text data to check for the target word. The key steps and techniques used in this kernel include thread indexing, substring matching, shared memory usage and efficient use of atomic operations to handle concurrent writes safely. Each thread in the kernel calculates its unique index based on the block ID and thread ID using the formula:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

This formula enables the distribution of data across threads by assigning a specific character position in the text data to each thread. With each thread responsible for its own index, all threads can can concurrently work on different parts of the text, reducing the time taken to process the entire dataset. This formula is important because it determines each thread's unique position within the data, allowing each thread to access and process a specific character in the text independently.

Once a thread has its starting index, it performs a substring match to check if the characters at this index match the target word. If the substring matches the target word, boundary checks are performed to ensure that the match is a stand-alone word:

- **Prefix Check**: Verifies that the character before the start of that match is not an an alphabetical character ensuring that the match is not part of a larger word.
- **Suffix Check**: Checks that the character after the end of the match is not an alphabetical character.

The first thread in each block initialises a shared memory variable, **'local_count'**, to zero. Each thread that finds a valid match increments **'local_count'** using an atomic operation. After all threads in the block have finished one designated thread writes **'local_count'** to the result in global memory, this minimises global memory access and reducing contention.

The kernel uses atomic operations to safely update counts without conflicts when multiple threads attempt to increment the same value concurrently. Atomic operations ensure that only one thread can modify a variable at a time. The **atomicAdd** function allows threads to safely update a shared variable in global or shared memory. This is essential in situations where multiple threads might try to increment the same counter simultaneously, which leads to race conditions. **atomicAdd** can introduce performance bottlenecks, especially when used excessively, as it forces threads to wait for exclusive access to the variable, which can impact kernel execution time. **_syncthreads** is a thread that synchronisation barrier that ensures all threads within a block reach the same point in the kernel before any thread proceeds. In this kernel, **_syncthreads** is used to synchronise the threads after initialising **local_count** and before the final write to global memory. This is critical when threads within the block share data in shared memory, as it also prevents race conditions by ensuring all threads complete one stage of computation before moving to the next.

By launching thousands of threads, each responsible for checking a different character position, the kernel allows the GPU to leverage its parallel architecture fully. In the original approach checking each position one at a time results in a linear increase in runtime as the file size grows. In contrast the GPU kernel scales well with file size, as additional threads can be launched to handle larger datasets without a big increase in runtime.

2 Optimisation

2.1 Optimisation target

The main goals of optimising the GPU were to reduce overhead from repetitive memory allocations, minimise contention in global memory and enhance the utilisation of GPU threads. The optimisation specifically targeted efficient memory management and the use of shared memory within the GPU , both of which are critical in achieving faster execution.

2.2 Kernel Optimisation

In the initial unoptimised version, memory for storing the text data, the target word and results was allocated each time the kernel executed a new search. This repetitive allocation created a significant overhead, as memory allocation and deallocation are relatively time-consuming on the GPU. The optimised approach improves efficiency by allocation memory for these elements once outside the loop and reuse this memory for multiple searches. The result buffer is reset between searches instead of being reallocated, which reduces setup time and ensures that each word search can start without waiting for memory operations. This alone saves considerable time, enabling faster processing of large text datasets.

Shared memory was used to store partial results. Each block uses shared memory to maintain a local counter for word matches. By accumulating matches within each block the need to access slower global memory is minimised. This improves performance reducing global memory access.

Each block was set to contain a high number of threads, which allows a large number of character positions in the text to be checked simultaneously.

2.3 Serial CPU version

The CPU version processes each character position sequentially, one by one, to determine if it matches the start of the target word. This version is straightforward but inherently limited by the CPU's single thread nature. This can only handle one position at a time, as a result the CPU version has a long execution time for large files. This linear increase in runtime with text size makes the CPU approach inefficient for handling large datasets.

3 Unoptimised GPU

In the unoptimised GPU version, multiple threads on the GPU simultaneously check different parts of the text for the target word, greatly increasing processing speed. Each thread writes its results to global memory, while faster than the CPU, creates contention issues. Many threads attempt to access and modify global memory at the same time, delays can arise. Additionally, memory allocation and deallocation for each search create further overhead, limiting performance.

Despite these limitations, the unoptimised version shows a speed-up over the CPU. This demonstrates the advantage of parallel processing, through further optimisation is required to reduce contention and improve memory handling.

3.1 Optimised GPU

In the optimised GPU version, memory allocation for the file data, target word and result buffer is moved outside the innermost loops. This change reduces the setup time associated with memory allocation by allocating GPU memory

for the entire set of searches rather than for each individual word search, For each word search, the result buffer is reset using **cudaMemset** to clear previous counts, rather than reallocating memory for each new search. This setup minimises allocation and deallocation cost, this enables the GPU to focus more on computation.

Shared memory is utilised within each block of threads to store a temporary counter, **local_count** which trachs word matches locally within that block. Threads that find a match increment this local counter using an atomic operation, which is faster and less costly in shared memory than in global memory. A single designated thread writes the **local_count** to global memory, adding it to the final result.

The kernel is configured with a **blockDim** value of 1024 threads per block, which is generally the maximum allowed on modern GPU's. This setup ensures a high degree of parallelism, as each block processes multiple character positions in the text simultaineously. The number of blocks (**num_blocks**) is dynamically calculated based on the size of the text data.

4 Hardware setup and Results

4.1 Hardware Setup

The experiment were done on a system using an Intel I7-9700KF CPU, NVIDIA RTX 2080 GPU and 32 GB of DDR4 RAM operating at 3000MHz. This hardware configuration is more than capable for evaluating GPU-accelerated algorithms.

4.2 Results

The experiment compared the performance of three implementations, CPU, unoptimised GPU and optimised GPU across multiple datasets. As shown in Figures 1-3. The GPU implementation significantly reduced execution time compared to the CPU implementation, with the optimised GPU version performing slightly faster in most cases.

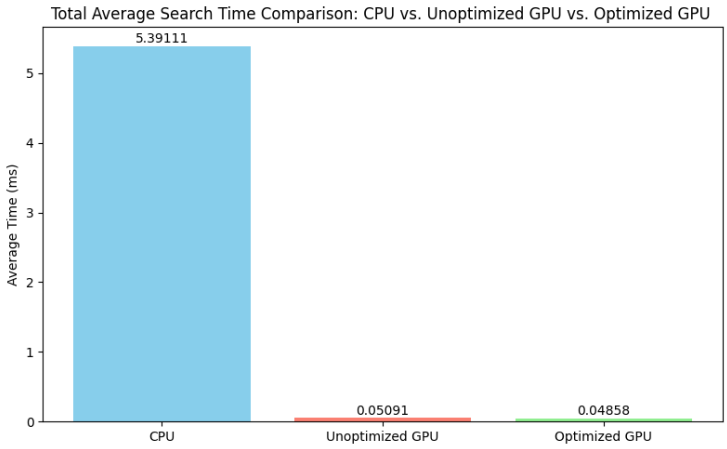

Figure 1: Total Average Time for All Datasets.

Figure 1 shows the CPU has a significant higher average time, reflecting the limitations of single-threaded processing when handling large datasets. The unoptimised GPU implementation offers a substantial improvement, using parallel threads to reduce processing time. The optimised GPU implementation further reduces the average time by using shared memory and reducing memory allocation overhead. This comparison shows the effectiveness of GPU-based parallel processing and the added value of optimisation for handling intensive tasks efficiently

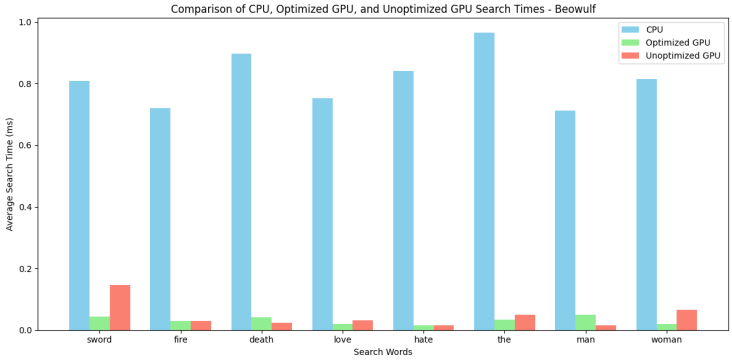

Figure 2: Beowulf average time.

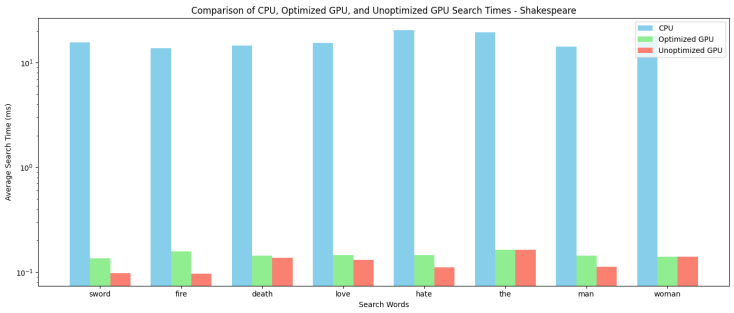

Figure 3: Shakespeare average time.

In the case of Shakespeare dataset, it was observed that the unoptimised GPU performed marginally faster than the optimised version. This unexpected result can be attributed to several factors. The added complexity and synchronisation overhead in the optimised version, particularly due to shared memory usage and atomic operations, may counteract the benefits of optimisation. When threads in the optimised version attempts to access shared memory concurrently, synchronisation barriers(**__syncthreads**) ensure data integrity but may introduce delays. Additionally, the optimised version's atomic operations, while reducing global memory contention, can cause bottlenecks if used frequently, as threads wait for exclusive access to update shared memory. (NVIDIA, 2013)

5 Conclusion

This report demonstrates the significant benefits of GPU parallelisation for word occurrence detection, transforming single-threaded CPU algorithm into an efficient parallelised GPU solution. The optimised GPU implementation outperforms both CPU and unoptimised GPU versions, with a substantial reduction in average execution time due to improved memory management and shared memory usage. However, results from specific datasets, such as the Shakespeare text, show that optimisation strategies may yield variable performance gains depending on dataset characteristics.

These findings show the importance of tailoring optimisation to the workload and hardware constraints. While atomic operations and shared memory improve parallel efficiency, they introduce complexity that may impact performance under certain conditions. Overall, this illustrates the potential of GPU parallelism to accelerate compute-intensive tasks, making it an invaluable approach for applications in text processing and other data heavy tasks.

6 References

NVIDIA. (2013). *Using Shared Memory in CUDA C/C++.* Retrieved from https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/