## Introduction and Objective

The aim of this project is to improve the efficiency of the CPU-based C++ text search program by leveraging the computational power of a GPU. Originally, the CPU program sequentially counted occurrences of specific target words within large text files, this approach limits performance due to the CPU’s sequential processing nature.

In a CPU-based program, counting the number occurrences of specific words in large text files is done sequentially, which is done by scanning each character in the file one at a time, comparing it to the target words selected. While this method is straightforward, this sequential approach is limited by the CPU’s linear processing capabilities, which means, it processes each position in the text individually and in order. This limitation becomes especially apparent with large datasets, since each comparison has to wait for the previous one to finish, which causes delays. In contrast, parallel processing, such as on a GPU, could handle multiple sections of text simultaneously, dramatically improving performance by reducing the reliance on sequential processing.

While CPUs process in a limited and sequential manner, the GPU on the otherhand, uses thousands of cores to process many tasks in parallel, Thus, GPUs can process big datasets extremely quickly, therfore, it can produce an unparalleled speed advantage compared to the CPU process. This allows for an high throughput while ensuring efficient memory access and scalable resource usage across multiple threads. This level of parallelism makes this ideal finding occurrences of specific words within a text file, as each thread may handle its sub-part with total independence, and this notably accelerates the performance.

By utilizing the parallel processing capabilities of the GPU, the objective was to transform this algorithm into a highly efficient version capable of handling large data volumes rapidly.  
  
Counting word occurrences across large text files is a key task in applications such as document indexing, data analytics, and natural language processing. CPUs are effective at handling such tasks on smaller datasets but are constrained by limited cores and their inherently sequential operations. In contrast, GPUs, with their thousands of cores, offer a way to perform these operations in parallel, providing significant time savings when managing large datasets.

## Porting the Algorithm to the GPU

To port the text search algorithm to the GPU, several critical steps were taken to maximize the parallel processing and also to ensure accurate results:

**Firstlly the tasks were Divided across the threads**, the tasks were divided so that the text data for each GPU thread was responsible for checking specific segments, enabling multiple comparisons to be done simultaneously. This division ensured that each part of the text was analyzed concurrently, utilising the GPU’s extensive parallelism.

Secondly, the CUDA kernel was created, called calc\_token\_occurrences\_kernel, which is designed to process the data in parallel. By ensuring that each thread within the kernel is checking for the target words, starting at its assigned position, coordinating through atomic operations to the record occurrences.

**Thirdly,** the function gpu\_strncmp, was implemented to compare strings directly on the GPU. This allowed threads to determine if a target word matched within their assigned text segment, using a parallel version of standard string comparison.

**Finally, to handle the out of bounds conditions**, to ensure accuracy, each thread, checked that the words matches occurred at word boundaries (e.g., “the” vs. “their”). This check verified that found words were not part of larger words, refining results to include only the targeted-word matches.

## Optimizing the Algorithm and Kernel Setup

Optimization was critical to fully leverage the GPU's capabilities for this text search task. The following key strategies were explored and implemented to improve efficiency and reduce execution time:

**Global Memory Coalescing**

**For this first GPU port, global memory was used directly, but memory access patterns were optimized to promote coalescence. Coalesced memory access aligns the reads across threads in a warp such that the threads with consecutive thread IDs read from consecutive memory locations. This pattern minimized latency for global memory, which is normally higher than shared memory but allows a large amount of data to be read at once. It did this because it made threads capable of accessing memory in a predictable and aligned manner.**

## ****Shared Memory Usage****

## **Shared memory was used as another approach to minimizing memory latency. By loading in segments of text into shared memory, which is much faster than global memory. Threads within a block could potentially access the data much faster.**

## ****Optimization of Atomic Operations****

## **The reason for using atomic operations in the first place was to update shared data safely: namely occurrence counts of a target word might be found by multiple threads of execution. However, the more frequent an atomic operation was for high-frequency words, the more contention there was because of threads waiting to get a hold of the shared data to update it. This was partially alleviated by experimenting with different block sizes in order to balance the load across the threads better. While atomic operations ensure that the data is accurate, they also created a bottleneck in performance for common words; thus, showing that further optimization can be entertained here.**

## ****Occupancy and Thread Count Tuning****

## **Occupancy tuning was performed by changing thread counts to find an appropriate balance between parallelism and good use of GPU resources. After having a look in the CUDA Occupancy Calculator, an ideal block size of 256 threads was found to give a very optimal balance of occupancy and resource utilization. Testing this with the block size set to different values, such as 128 and 512 threads, demonstrated that as block sizes increase, high register use and low occupancy become an issue, while a thread count that is too low is underutilized. There seemed to be a sweet spot where throughput was maximized without hitting resource limits.**

## ****Experimental Block Size Tuning****

## **Block sizes ranging from 32 to 1024 threads were tested in search of the optimal setting. This gives the best performance, as 256 threads/block can balance parallelism and resources utilization perfectly, while larger block sizes have poorer performance due to register pressure and lower occupancy, since fewer blocks can run concurrently on the streaming multiprocessors. This proves that block size selection will really significantly impact the overall performance, especially in handling large texts with lots of changes in word frequency.**

## 4. Hardware Setup, Results, and Analysis

### Hardware Setup

## Number of devices: 1

## Device 0

## Name NVIDIA GeForce RTX 3070

## Revision 8.6

## Memory 8191MB

## Warp Size 32

## Clock 1725000

## Multiprocessors 46

## 5. Conclusion

**However, because each thread is working on a different segment of text and data reuse is limited, the benefits of shared memory did not outweigh the associated overhead in this workload. Profiling indeed indicated that the shared memory version had some extra overhead and thus was slightly slower than the pure global memory version. The message is that shared memory pays off mainly when within a block data re-use is substantial between threads**

By porting the text search algorithm to the GPU, this project achieved substantial performance gains compared to the original CPU-based version. The optimizations applied—including occupancy management, block size tuning, and strategic use of global memory—enabled the GPU implementation to handle large text files efficiently. Shared memory proved unnecessary, as its overhead outweighed the potential benefits for non-reusable data access patterns. Atomic operations introduced a bottleneck for high-frequency words, highlighting an area for further optimization using techniques like histogram-based counting.