## ****Introduction and Objective****

**The goal of this project is to enhance performance of a CPU-based C++ text search program using the computational resource of a GPU. Originally, the CPU program had counted occurrences of certain target words in large text files sequentially; clearly, a process where performance is bound to the sequential nature of processing on a CPU.**

**The occurrences of certain selected target words are counted sequentially in large text files by the CPU-based program; it does this by scanning each character in the file one at a time and comparing that with the target words selected. Although the approach is indeed straightforward, such a sequential approach depends on the linear processing capability of a CPU, meaning it handles each position in the text individually and in order. This becomes a very critical point when big datasets are being handled since all comparisons have to wait for the one before it to finish. On the other hand, parallel processing-on say a GPU-would be able to manage chunks of text simultaneously and would considerably raise performance by reducing sequential processing. Where the CPU can process in a very limited and sequential manner, on the other hand, the GPU makes use of several thousand cores to process many things in parallel. Thus, GPU can process big datasets extremely fast, thereby creating an unparalleled speed advantage over CPU processing.**

**This allows for a high throughput while ensuring efficient memory access and scalable resource usage across multiple threads. The amount of parallelism is ideal for finding occurrences of certain words in a text file, for example, since each thread can process its sub-part completely independently, and this exceedingly speeds up the performance. The objective was to use a parallel-processing capability on the GPU for converting this algorithm into an ultra efficient algorithm that is able to work through very large amounts of data in least time.**

**Counting word occurrences across large text files is a fundamental task that naturally arises in applications such as document indexing, data analytics, and natural language processing. Although CPUs are quite capable of performing such tasks on relatively small datasets, they suffer from a general limitation in the number of cores and inherently sequential operations. In sharp contrast, modern GPUs offer a way to perform these operations in parallel, thanks to their thousands of cores. This results in considerable time savings during the management of large datasets.**

****Porting of the Algorithm on GPU****

**Following are some important steps that were considered while porting the text search algorithm on GPU to have maximum parallel processing along with correct results:**

**First of all, tasks were divided among threads: the tasks were divided in a manner that for every GPU thread, text data became responsible to check certain segments, thus allowing multiple comparisons to be done simultaneously. The division in such a way made sure that each part of the text would get analysed in parallel while using the huge parallelism of the GPU.**

**A CUDA kernel was then implemented, named calc\_token\_occurrences\_kernel, capable of processing the data in parallel. Given that each thread in this kernel will be iterating through the target words at their respective starting positions, it will use atomic operations with each other to record occurrences.**

**Thirdly, the function gpu\_strncmp was implemented to compare strings directly on the GPU, giving the threads the capability to decide whether there existed in their assigned segment of text a matching target word through a parallel version of standard string comparison.**

**Finally, to handle the out of bounds conditions and make sure that accuracy was maintained, each thread checked that the words matched at word boundaries e.g., "the" vs. "their". This check confirmed the found words were not part of larger words and further refined the results to include targeted-word matches only.**

****Optimizing the Algorithm and Kernel Setup****

**Optimization is key in the text search task to really exploit the full potential of the GPU. The ways in which this was done to enhance efficiency and minimize runtime are listed below:**

****Global Memory Coalescing****

**In this initial GPU port, global memory was directly used, but memory access was optimized to promote coalescence.**

**Coalesced memory access aligns the reads across threads in a warp such that threads with consecutive thread IDs read from consecutive memory locations. This pattern minimized latency for global memory-normally higher than shared memory but allowing a great amount of data to be read out in one go. This was because it made threads capable of accessing memory in a predictable and aligned manner.**

****Usage of Shared Memory****

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

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

**For the most part using atomic operations in the first place was to update shared data in a safe way; namely occurrence counts of a target word might be found by multiple threads of execution.**

**However, the more frequent an atomic operation was used, which for high-frequency words might be the case, the more contention there was due to threads waiting to get hold of the shared data in order to update it.**

**This was partly alleviated by experimenting with block size to balance the load across threads better. While atomic operations ensure that the data is accurate, it also created a bottleneck in performance for common words. Thus, showing further optimization can be entertained here. This would have been possible with more time; however, with the limitation of time, it is not explored further.**

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

**Occupancy and thread count tuning refer to the process of adjusting the number of active threads in each Streaming Multiprocessor(SM) in the GPU to a maximum without overloading its resources. The level of occupancy can be defined as the ratio of active warps to the maximum possible number of warps on an SM. Occupancy allows the GPU to keep its processing units busy by masking memory latency.**

## Hardware Setup, Results, and Analysis

### Hardware Setup

## Number of devices: 1

## Device 0

## Name NVIDIA GeForce RTX 3070

## Revision 8.6

## Memory 8191MB

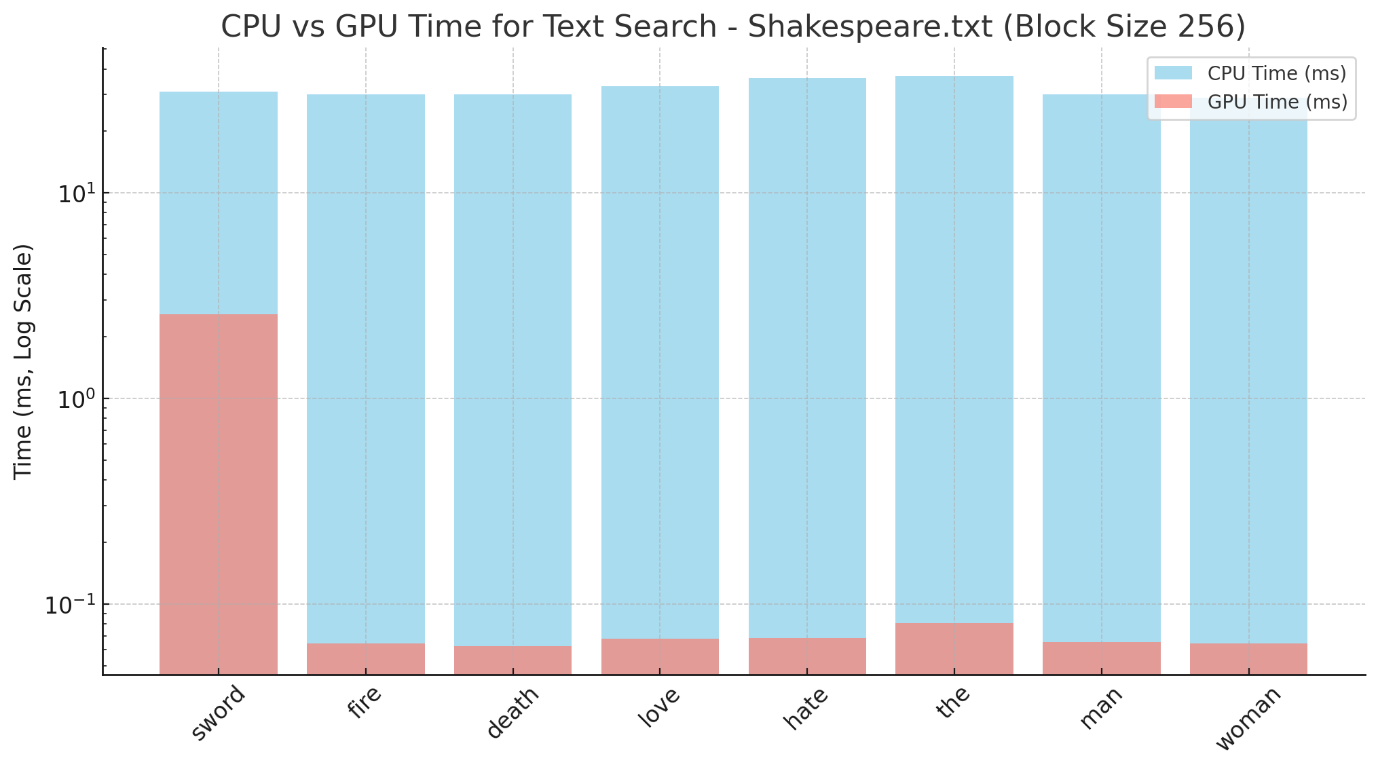
## Warp Size 32

## Clock 1725000

## Multiprocessors 46

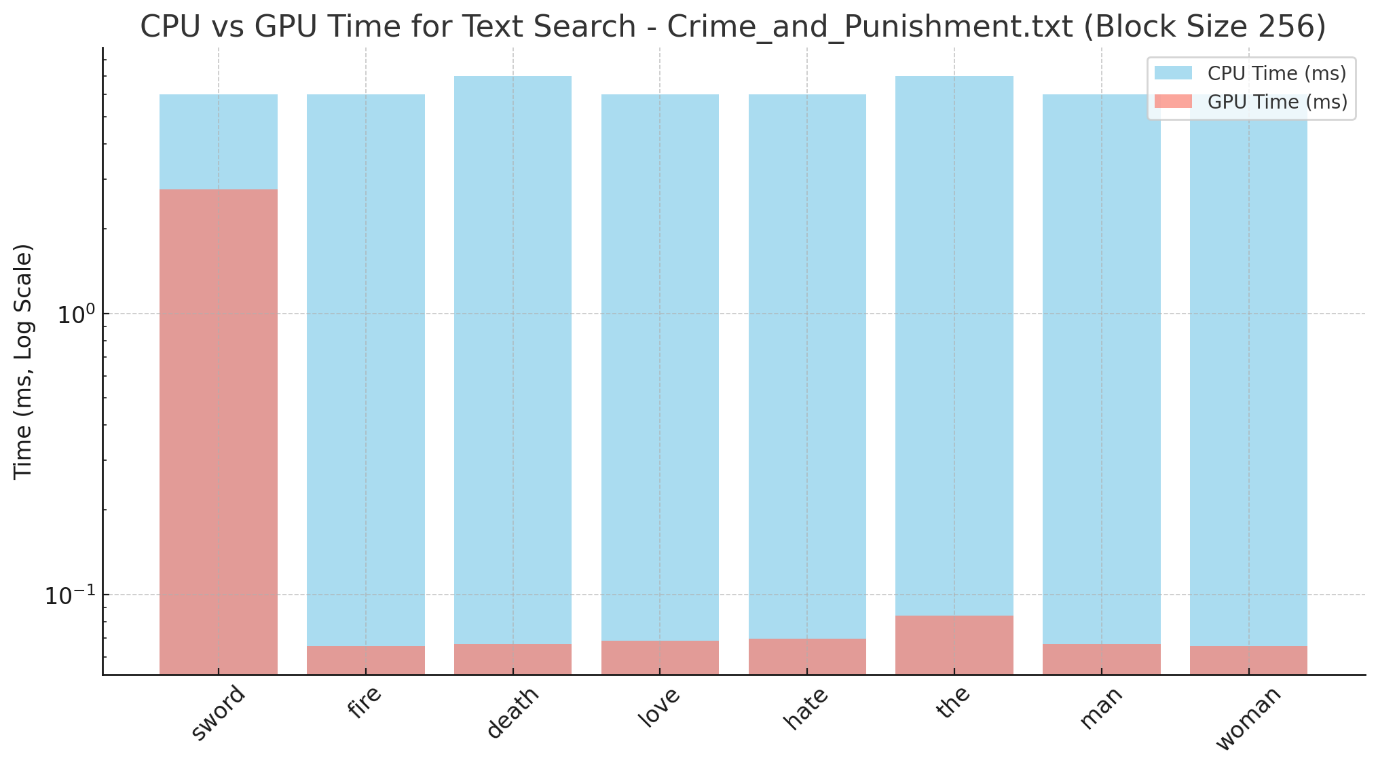
**Plots**

Base code



A graph with text on it

Description automatically generated



Shared memory

A graph of blue and pink bars

Description automatically generated

Occupancy thread

A graph with blue and pink bars

Description automatically generated  
 A graph of blue and pink bars

Description automatically generated

Block sizes ranging from 32,64, 128,, 256, 512, 1024 Shakespeare

A graph with a green line

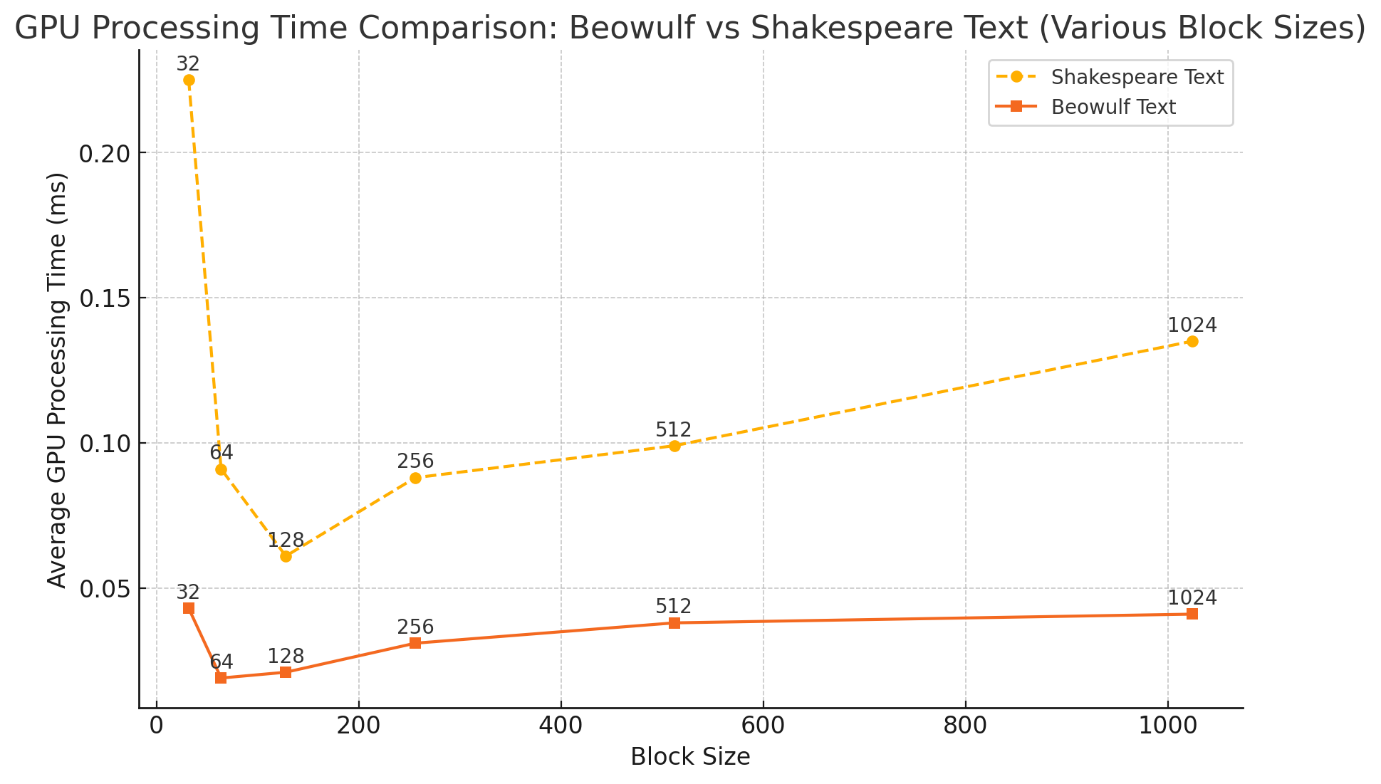
Description automatically generated

Block sizes ranging from 32,64, 128,, 256, 512, 1024 Beowulf

A graph with blue and orange lines

Description automatically generated

Beowulf vs. Shakespeare



**Discussion**

**Average Processing Time for Block Sizes in Beowulf and Shakespeare Texts**

These plots demonstrate that the processing times vary considerably across the block sizes, especially in the case of the Shakespeare text where, for an optimum block size of about 128-256 threads, the processing time was minimized. More than 256 threads, that is block size greater than 256 resulted in reducing returns due to increased register pressure and lower occupancy.

Meanwhile, the block sizes did not affect Beowulf's processing time, indicating that text characteristics are less sensitive in Beowulf when performing block size tuning. The GPU still outperformed the CPU in both texts; hence, it was shown that parallelism benefits both kinds of datasets, just with small differences in the optimal settings.

**CPU vs. GPU Time Comparison for Various Words in Beowulf and Shakespeare Texts**

The time comparisons of words across texts, "sword," "love," "hate," appropriately reflect how disparate the GPU/CPU processing times may be. This regularly has each word on the GPU far below its CPU counterpart, with the GPU result being much faster.

Specifically, high-frequency words like "the" and "man" saw slight performance drops in both, possibly because of contention due to atomic operations when several threads were trying to update such words' count all at once. Although this is still way faster than on the CPU, it does suggest that further optimization using techniques such as histogram-based counting might be required to avoid a bottleneck for high-frequency words.

**Shared Memory Optimization and its Impact**

These charts of Shakespeare text processed with shared memory reveal that the anticipated dividends of shared memory were not fully realized, since the processing time did not significantly outperform the global memory version. Shared memory works best when the workloads have a high level of intra-thread data reuse. In this regard, the effectiveness of shared memory is minimized by the limited reuse in text searches.

These findings confirm the fact that for this task, shared memory overhead was not justified by its use and reinforce the fact that global memory was the more efficient choice for this particular task.

**Impact of Atomic Operations and Bottlenecks**

The observed bottlenecks, especially for high-frequency words, are really pointing to the limitation of atomic operations. Though necessary for safe concurrent updates, atomic operations introduce a bottleneck that worsens with the increase in frequency of a word. High contention to update the occurrence count caused delays, especially when many threads attempted to modify the count simultaneously.

For this, it is recommended that alternative counting methods be pursued and tested which could better reduce atomic operation contention and yield further performance increases for common words.

**Comparison Across Texts and Scalability Implications**

Scalability implications are shown by comparing the GPU processing time between Beowulf, Shakespeare, and Crime and Punishment.

This would mean that the GPU optimization is highly scalable and would easily fit into a high-volume text dataset with complication. The gains across different texts affirm the adaptability of the GPU approach; however, further tuning for specific characteristics of each dataset-for example, word frequency distribution-could have resulted in further improved results.

**Conclusion**

The optimizations in the GPU showed a much larger increase in performance compared to the CPU implementation. Tuning of block size and usage of global memory had been crucial for best processing. The parallel processing capability of the RTX 3070 handled all large-sized datasets with ease, where for every dataset the GPU gave out much better performance compared to that of the CPU.

Due to low re-usability of data, shared memory use became dispensable and atomic operations for words used often became the minor bottleneck. Atomic operation contention can be reduced further in upcoming optimizations, and different memory structures should also be explored to gain even higher efficiency.