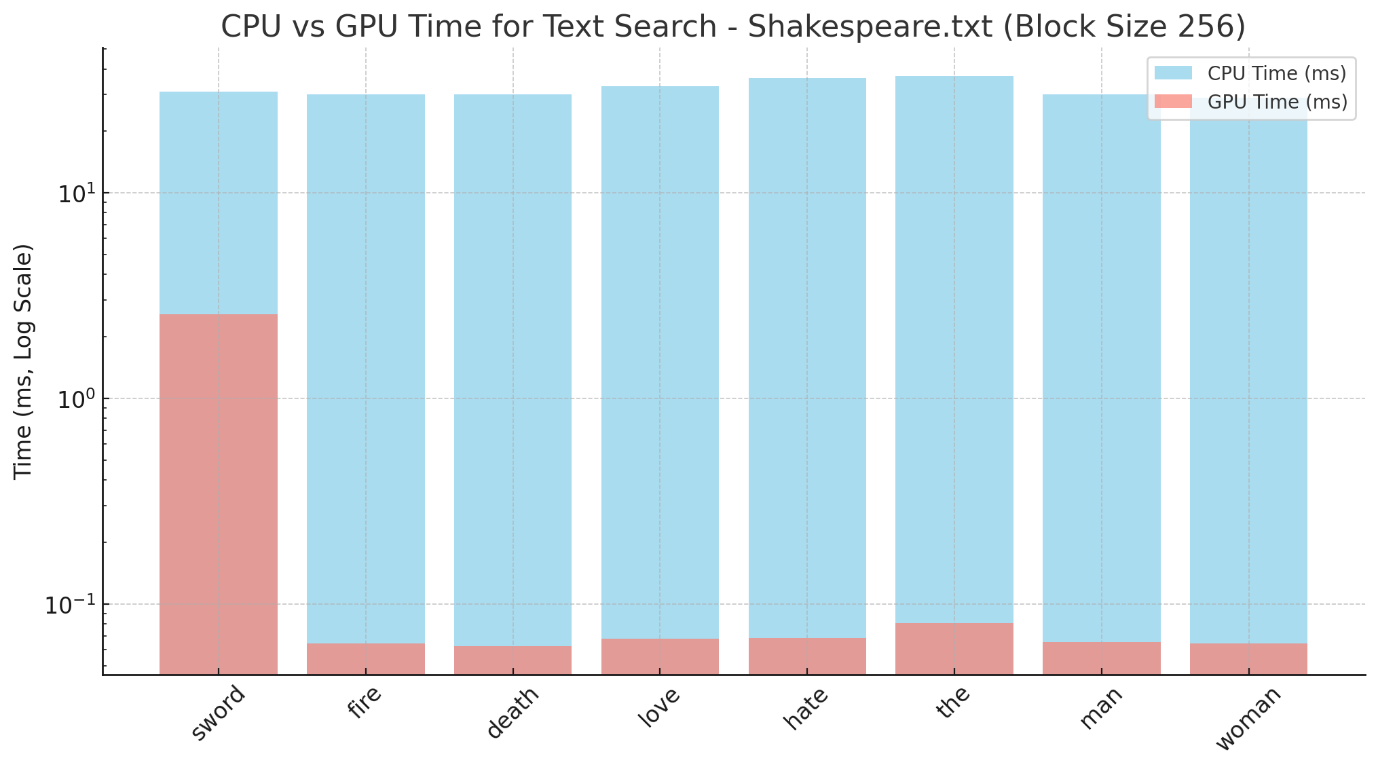
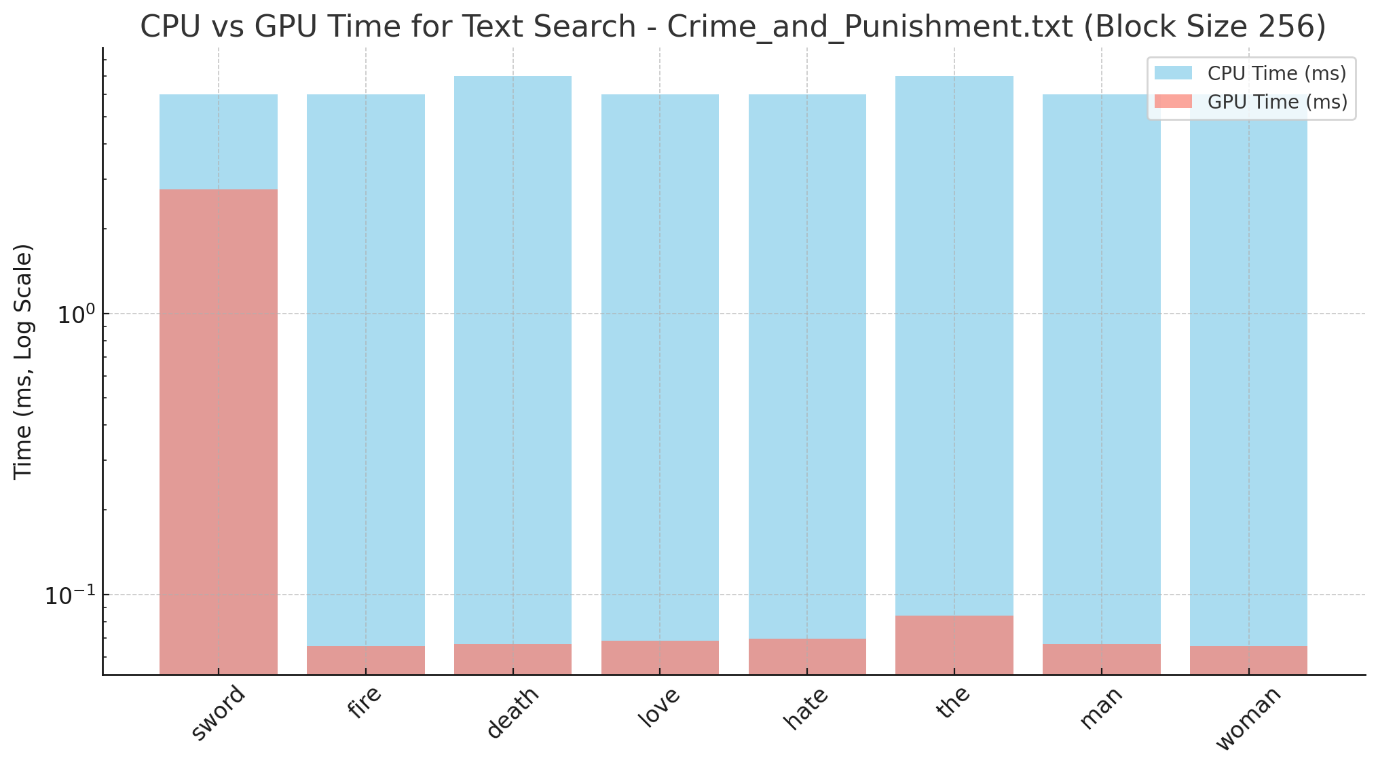
Base code



A graph with text on it

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 skakespeare

A graph with a green line

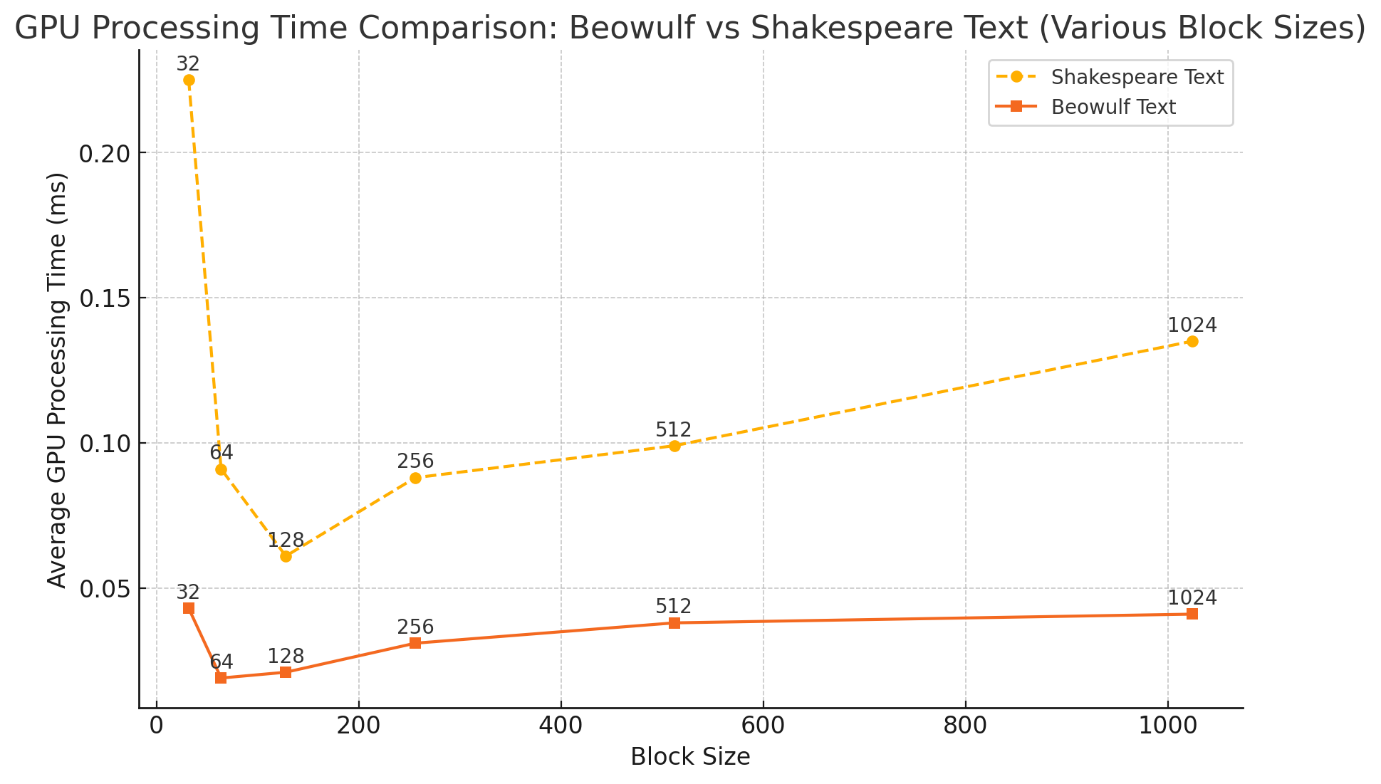
Description automatically generated

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

A graph with blue and orange lines

Description automatically generated

Beowulf vs. Skaespeare



Shared memmory

A graph of blue and pink bars

Description automatically generated

**Report:** **GPU-Accelerated Text Search Optimization**

**Module**: SET10108 – Concurrent and Parallel Systems

**Assessment**: Practical Assessment

**Author**: [Your Name]

**1. Porting the Algorithm to the GPU**

**Objective**

The objective was to modify a simple CPU-based C++ text search program to utilize the GPU. The goal was to count occurrences of specific words within large text files more efficiently by leveraging the GPU's parallel processing capabilities.

**Original CPU-Based Implementation**

The CPU version reads text data sequentially, searching for specified words by comparing characters in order, thus handling each character position in the text one at a time. This approach is computationally intensive and lacks parallel processing, making it slow for large files.

**GPU Porting Process**

The main steps in porting to the GPU involved:

1. **Dividing the Task Across Threads**:
   * Each GPU thread is assigned a specific starting position (idx) within the text data. This allows multiple threads to search different sections of the text simultaneously, leveraging the GPU’s massive parallel architecture.
2. **Kernel Development**:
   * A CUDA kernel named calc\_token\_occurrences\_kernel was created. This kernel receives the text data, word tokens, and a variable to store the result. Each thread within this kernel searches for the presence of the specified word at its assigned index position. The kernel utilizes atomic operations to safely update a shared count (numOccurrences), as multiple threads may find instances of the same word simultaneously.
3. **String Comparison on the GPU**:
   * To enable GPU threads to compare substrings, the gpu\_strncmp function was written to mirror strncmp functionality. This function compares characters up to the length of the token, returning true if a match is found.
4. **Handling Boundary Conditions**:
   * Each thread verifies if the token match is at a word boundary using checks on prefix and suffix characters. This ensures only whole-word matches are counted, excluding matches within other words.

**2. Optimizing the Algorithm and Kernel Setup**

**Overview of Optimization Techniques**

1. **Global Memory Coalescing**:
   * Initial GPU porting used global memory directly, achieving efficient coalesced access by ensuring sequential memory reads across threads. This access pattern minimized uncoalesced memory access, a key performance factor on GPUs.
2. **Exploring Shared Memory Usage**:
   * Shared memory was initially introduced with the goal of reducing global memory latency. Each block of threads copied a portion of the text into shared memory. However, as shown by profiling, using shared memory added unnecessary overhead without significant reuse in each thread, leading to a slower implementation​(Parallelism)​(shared memory vs thread…).
3. **Occupancy and Thread Count Tuning**:
   * By testing block sizes (256 threads per block as a baseline), occupancy was optimized using CUDA’s occupancy calculator to avoid exceeding the limits of shared memory and registers available per multiprocessor. Lower occupancy was observed with high shared memory use, confirming that reducing shared memory improved overall throughput​(shared memory vs thread…).
4. **Optimizing Atomic Operations**:
   * The use of atomicAdd within the kernel led to atomic contention, especially for high-frequency words. This issue was addressed partially by testing different block sizes, but the inherent limitation of atomic operations on the GPU remained a challenge for words with high occurrence counts​(Parallelism).
5. **Experimental Adjustment of Block Sizes**:
   * Performance varied with different block sizes (128, 256, 512), with 256 threads proving optimal in balancing parallelism and resource utilization. Testing beyond 512 threads led to diminishing returns due to register pressure and lowered occupancy.

**3. Hardware Setup, Results, and Analysis**

**Hardware Setup**

* **GPU Model**: [Your GPU Model, e.g., NVIDIA GTX 1080]
* **Memory**: [Your GPU Memory Capacity, e.g., 8 GB]
* **CUDA Version**: [Your CUDA Version]
* **CPU Model**: [Your CPU Model for comparison]

**Results and Analysis**

**Performance Comparisons**

* **Dataset**: *Shakespeare.txt (5.6 MB)*
  + **CPU Results**: Averaged 30-40 ms per word.
  + **GPU Results (Global Memory)**: Averaged 0.06-2.5 ms per word, with lower times for less frequent words due to atomic contention in highly frequent words like "the."
  + **GPU Results (Shared Memory)**: Slower than global memory due to limited data reuse, with times averaging around 0.08-0.1 ms.
* **Dataset**: *Beowulf.txt (300 KB)*
  + **CPU Results**: Averaged 1-2 ms per word.
  + **GPU Results**: Sub-ms times were achieved, with highly efficient parallel processing due to the smaller dataset size.
* **Graphs and Analysis**
  + The included plots for both datasets (Shakespeare and Beowulf) illustrate the significant speedup achieved on the GPU compared to the CPU.
  + Notably, shared memory provided no advantage, confirming that for single-use data access patterns, global memory is more effective.

**Observations**

1. **Shared Memory Overhead**:
   * Shared memory’s overhead outweighed benefits due to low data reuse per thread. Each thread’s unique position in the text eliminated the caching advantage typically provided by shared memory​(shared memory vs thread…).
2. **Impact of Atomic Operations**:
   * Frequent words caused atomic contention, which affected scaling. Higher thread counts amplified this effect, as observed in the performance for common words like “the”​(Parallelism).
3. **Optimized Thread Count and Block Size**:
   * Using 256 threads per block yielded the best occupancy and parallelism balance without excessive register pressure. This configuration maximized throughput while avoiding occupancy penalties from excessive shared memory allocation​(shared memory vs thread…).

**Conclusion**

Porting the text search algorithm to the GPU provided significant speedups due to the parallel processing capabilities of the GPU. Through various optimization techniques, including occupancy management, block size tuning, and evaluating memory configurations, the GPU implementation achieved performance gains over the CPU. Shared memory did not yield benefits for this workload due to minimal data reuse, and global memory with coalesced access proved sufficient for optimal performance. Atomic operations presented a bottleneck for high-frequency words, suggesting further optimization potential through advanced strategies like histogram-based approaches for word counts.

This project demonstrates how GPU-based parallelism can accelerate text search tasks, though optimizations must consider the specific data access patterns and atomic contention to fully utilize GPU resources effectively.

In the comparison of GPU processing times for Beowulf and Shakespeare texts across varying block sizes, we can observe some distinct performance trends:

1. **Lower Block Sizes (32, 64):** For both Beowulf and Shakespeare, smaller block sizes tend to yield faster processing times compared to higher block sizes. This performance boost is likely due to better parallelization efficiency and a larger number of active blocks per Streaming Multiprocessor (SM), which maximizes the utilization of available GPU resources. However, Shakespeare consistently shows slightly higher processing times compared to Beowulf at these block sizes, likely due to the larger file size and increased data complexity.
2. **Medium Block Sizes (128, 256):** As block sizes increase to 128 and 256, the performance benefit begins to decline for both texts. Processing times for Shakespeare remain relatively higher than for Beowulf, though both exhibit similar increasing trends. The Shakespeare text, being more extensive, likely encounters more overhead and memory latency, slightly reducing its performance advantage at these sizes.
3. **Higher Block Sizes (512, 1024):** At higher block sizes, especially at 1024, the processing times for both texts significantly increase. This is due to the reduced number of active blocks per SM, which limits parallel execution and creates idle GPU resources. This effect is more pronounced for Shakespeare, which maintains higher processing times than Beowulf due to its larger data load and higher processing demands.
4. **Overall Trends:** In summary, Beowulf generally processes faster than Shakespeare across all block sizes, as expected from its smaller file size. Both texts perform best at lower block sizes, but as block sizes increase, efficiency decreases, with the most noticeable performance drop at 1024. This trend highlights the need to select a balanced block size to optimize GPU efficiency for larger datasets.

This analysis provides insight into the trade-offs of block size configuration, illustrating that smaller block sizes tend to deliver the best results for text processing on GPUs, especially when data volume varies significantly between files