Efficient KMP string matching on the GPU

1. Introduction

String Matching is a pervasive approach in contemporary research pursuits spanning several domains, including bioinformatics, cybersecurity, and computer antivirus scanning. While information is stored in various forms, text continues to be a predominant method for information exchange. This is particularly conspicuous in the realm of bioinformatics where string matching algorithms are of great use. To illustrate, the human genome comprises roughly 3 billion base pairs. Through genetic diagnostics, individuals can receive early warnings about genetic disorders, enabling them to take preventative measures or begin treatment before symptoms manifest.

String machine is easily defined as finding a pattern string in a text string and many algorithms, such as Knuth-Morris-Pratt (KMP) algorithm [Knuth77] and the Boyer-Moore (BM) algorithm [Boyer77], have been proposed to efficiently search for patterns in long texts. However, the searching time rises significantly when the data grows enormously large, notably in the task of genomic sequence matching. Extended processing time for identifying patterns in massive data sets using traditional architectures impedes the algorithm's industrial application.

Lately, Heterogeneous High-Performance Computing (HPC) architectures, underpinned by standard multicore microprocessors and Graphical Processing Units (GPUs), have gained increased attention. These architectural progressions have led to massive parallelism and substantial performance enhancements. Additionally, intuitive programming frameworks such as Nvidia's CUDA have been devised for general-purpose computing on GPUs. Hence, leveraging GPUs to parallelize string matching algorithms appears promising. Among various string-matching algorithms, the KMP algorithm is distinctive due to its swift processing speed, especially when managing voluminous input data and benchmark patterns, as well as its ability to parallelize well on the GPU. Therefore, it's essential to diminish processing time by effectively parallelizing the KMP algorithm.

In this study, we focus on accelerating the KMP string matching algorithm to efficiently identify a single gene pattern in a large genomic sequence. Our proposed algorithm employs definite state automata (DFA) to enhance search performance and reduce the branch divergence in the naive KMP algorithm. Moreover, we utilize shared memory and constant memory to further improve its efficiency. We also compress gene data from 8 bits to 2 bits to minimize redundant global memory reads. Finally, we utilize zero-copy to boost speed by eliminating the need to explicitly copy data between the GPU and CPU.

In subsequent sections, we begin by discussing prior work on the optimization of KMP. Next, we revisit the KMP algorithm and DFA. Following this, we introduce the benchmark for GPU-accelerated KMP and our optimizations. In the evaluation section, we carry out experiments and performance analyses to compare our implementations. Finally, we discuss the future work and make a conclusion.

2. Related Work

The Knuth-Morris-Pratt (KMP) algorithm has managed to reduce time complexity by using a pre-processing step that reads every character in a text string only once [Knuth77]. By skipping needless comparisons, the KMP algorithm achieves appreciable performance. Nonetheless, processing pattern matching on millions of text remains a lengthy task under the current CPU. As a result, an increasing number of parallel algorithms for string matching are being redesigned to accommodate GPU configurations.

[Cao11] introduced the parallel KMP algorithm, leveraging MPI. [Rasool12] deployed KMP on SIMD architectures, which cut down the comparisons, thereby amplifying time efficiency. They adopted a two-step search. Firstly, the text is chunked, and each thread searches for the pattern in its own chunk. In the second step, all the threads search for the pattern at every junction point to prevent missing the pattern at the boundary. Their method was complicated and not cache friendly.

[Lin13] applied KMP on GPU using CUDA and OpenMP. However, they only evaluated the method on text strings up to 100,000 characters long, which does not sufficiently represent the performance on extraordinarily large data. In 2014, [Nagaveni14] introduced string-matching algorithms to identify DNA sequences indicative of breast cancer. However, their work only included a basic parallel version of KMP, without any specialized optimizations.

In recent times, [Neungsoo20] fine-tuned the allocation of work-groups and work-items and stored the pattern data and the failure table in the GPU's on-chip local memory. This led to a speedup of up to 15.25 times. However, the work needed to choose computation parameters with consideration to work-groups and work-items to optimize the performance.

3. Algorithms

In this section, we introduce two variants of the KMP algorithm that we have implemented: the naive KMP algorithm and the DFA string matching.

3.1. Naive KMP Algorithm

The naive KMP algorithm is the most common implementation of KMP on the CPU. It has time complexity $O(n + m)$, where n and m are the length of the text and the pattern respectively.

The naive KMP algorithm contains two steps. The first step is to build a partial match table from the pattern string, which we call the *fail* table. The second step is to search in the text string using the fail table. The purpose of the fail table is to keep some information on how much the pattern should shift forward when there is a mismatch at every index of the pattern.

Here is an example shown in Figure 1 of how the KMP algorithm performs string matching using the fail table. In this example, we are searching for the pattern string ATTG in the text string ATTCATCG. The first mismatch happens at index 3 of both the text and the pattern. Then according to the fail array, the pattern should shift so that pattern index 0 is aligned with the text index 3. After that shifting the pattern is forwarded by a step of 3.

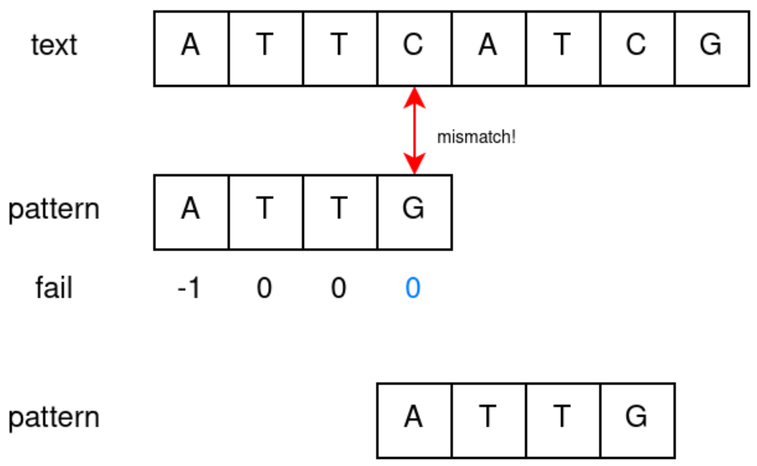

Figure 1: KMP string matching example

The fail table is somewhat similar to a state machine matching, where each number ranging from -1 to m corresponds to a state. If there is a match, we transfer to the next state by increasing the state number by 1. When there is a mismatch, we check the fail table and shift to that state. However, there is no guarantee of how many fail shifts we have to perform. In the example above, after we have shifted, there is still a mismatch and we have to check the fail table and shift to the fail state again until there is a match.

3.2. DFA

DFA is another algorithm to perform string matching. It has close connections with the naive KMP algorithm because it can be viewed as an *optimization* of the naive KMP algorithm. In the naive KMP algorithm, the fail table of KMP has only two cases: matched and not matched. When there is a mismatch, we might check the fail table and perform fail jumps multiple times, thus causing branch divergence between threads. In comparison, in the DFA, we pre-compute the results of the fail jumps for every state when we encounter every possible input character. These results are stored in the jump table. With the jump table, when we read a character from the text, we loop up the jump table and jump only once to the next state, regardless of a match or a mismatch, thereby reducing the number of fail jumps and effectively eliminating branch divergence.

However, the jump table of a DFA is larger than the fail table of KMP. The jump table contains $|\Sigma|(m + 1)$ entries, where $|\Sigma|$ is the size of the alphabet, while the fail table contains only $m + 1$ entries. In the context of genome sequences though, the size of the alphabet is only 4, therefore the memory overhead is not too much compared to the naive KMP algorithm.

4. The Basic KMP Kernel Implementation

In this section, we focus on the implementation and optimization of the naive KMP algorithm on the GPU.

4.1. Basic KMP kernel

In the basic implementation, which serves as the baseline, we split the text into multiple chunks and assign one chunk to every thread. Then each thread searches for the pattern in its chunk using the naive KMP algorithm and stores the result in a global output array. The pseudo-code of our naive KMP algorithm is provided below. text, pattern, and fail are all arrays stored in the global memory.

```
# Assume fail table is already computed on the CPU.
i = chunk_start  # Index of text
j = 0  # Index of pattern
while i < chunk_end:
    if j == -1 or text[i] == pattern[j]:
        i++, j++
        if j == pattern_length:
            write_to_output(i-j)
    else:
        j = fail[j]
```

Following [Neungsoo20], we overlap the chunks by a length of $m - 1$ so that we can find the pattern even if it goes across the boundary. Figure 2 shows an example of three threads finding a length-2 pattern GA. The three chunks are overlapped by a length of 1. Even though GA goes past the boundary of chunk 0, it can still be searched by thread 1.

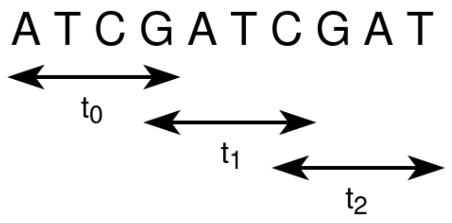

Figure 2: An example splitting of text across 3 threads, with chunk size = 4.

4.2. Basic KMP Kernel with Shared Memory

It is straightforward to optimize the naive kernel using shared memory. Since the pattern and the fail table are frequently used during searching, we load both of them into the shared memory to speed up the searching process. Both the pattern and the fail table are loaded in a coalesced manner.

We could also store the outputs into a buffer in the shared memory first, and then write that buffer collaboratively to the global memory to improve efficiency. Nevertheless, this proves to be less efficient in our setups where the selectivity is low. This is because of the overhead to synchronize the threads and more branch divergence. In the future, we can try to develop a kernel that is able to dynamically choose whether to enable a shared memory buffer or not, but in this project we assume that the selectivity is low such that directly writing to the global memory will not negatively affect the performance.

We also tried to store the pattern and the fail table in the constant memory. Although it may not seem like a good idea because different threads can access different parts of the fail table and these accesses are serialized, thus causing great latency. However, due to the exponentially decreasing selectivity as the pattern length increases, threads in a half warp are unlikely to match at random positions. Instead, most of the time the threads access only the first few elements of the jump table. Therefore the memory latency is lower than expected, which explains why our constant memory implementation is faster than the shared memory one.

4.3. Summary of the Basic KMP Kernels

The naive KMP algorithm is the most widely used algorithm on the GPU for string matching because it is simple and easy to parallelize, but it suffers a lot from branch divergence, which is analyzed in section 7. The basic version of the KMP without shared memory and constant memory is used as our baseline.

5. The DFA Kernel Implementation

In this section, we focus on the implementation and the optimization of the DFA algorithm on the GPU.

5.1. DFA Kernel with Shared Memory

Similar to the case of KMP, the jump table of the DFA can also be stored in the shared memory to speed up the searching process. Although the jump table of a DFA is 4 times larger than the fail table in the naive KMP, shared memory is still large enough. Even for a pattern with a length of 100, the jump table takes only 1.56KB, which is much smaller than the 40KB shared memory size.

5.2. DFA Kernel with Constant Memory

We also explored the usage of constant memory for the jump table, under the assumption that its immutability during kernel execution would yield performance benefits. This involved copying our jump table to the constant memory. However, the experimental results revealed that this approach was less performant than the shared memory implementation.

This outcome was somewhat surprising given that in the basic KMP implementation, the constant memory approach outperformed the shared memory one. We postulate that the key difference lies in the size of the jump table. In our case, the jump table is substantially larger than the one used in the basic KMP. This results in threads accessing more dispersed locations in the constant memory.

To maximize the benefits of constant memory, it is ideal for a half-warp to access the same location; otherwise, the accesses get serialized, which can diminish performance. We suspect that this is precisely what is occurring in our situation — although this time, the threads still concentrate in the first few states, the addresses they are accessing are quite different due to the difference of the input text characters they are reading in. Therefore the accesses by threads to constant memory are getting serialized due to the dispersed access pattern. This is also evident in the reduced memory throughput indicating serialized memory access, as compared to shared memory KMP(Fig 5). This, we believe, is the main contributing factor to the performance degradation observed in this implementation.

5.3. Input Coalescing

In all the kernels previously mentioned, the kernel all access the text from the global memory in an uncoalesced manner. Each thread in a warp will access the beginning of its own chunk, leading to a strided access pattern. Since we still have much space in the shared memory, we can perform the "corner turning" technique by first loading all the chunks needed by a block collaboratively into the shared memory, and then each thread finds the pattern in its own chunk. In this way, the input can be loaded in a coalesced manner, while the strided memory access only happens in the shared memory.

In order to fit all the chunks of the text into the shared memory, we choose the chunk size to be 128. This utilizes a total of 32 KB of shared memory for each SM, which is close to the limit of 40 KB of shared memory per SM.

Nonetheless, we empirically found that the effect of input coalescing is not significant. The main reason is that in the KMP algorithm, there is no data reuse and every text character is read only once. The other reason is that in the shared memory version, most of the data used from the global memory is already stored in the L2 cache.

5.4. Asynchronous Data Transfer

cudaMemcpyAsync is a CUDA runtime API that enables asynchronous data transfer between the host and the device. This means that the data transfer is overlapped with the kernel execution, thus utilizing the unused bandwidth during kernel execution and can potentially improve the overall execution time and the kernel runtime.

In our setup, transferring the text from the host to the device (via 'cudaMemcpy') is very time consuming, which takes more than 10x of the kernel execution time in the input coalesced kernel. By enabling asynchronous data transfer, we find that the time to copy the text to and from the GPU is reduced by 15%.

5.5. Zero copy

Zero-copy is a feature in CUDA that allows the GPU to directly access host (CPU) memory. Normally, when a GPU kernel needs to access data, that data must reside in the GPU's device memory. If the data is initially in host memory, it must be explicitly copied to device memory, which can be a time-consuming operation.

With zero-copy, the GPU can directly access host memory without requiring a separate copy operation, hence the name "zero-copy". However, it's important to note that accessing host memory from the GPU is slower than accessing device memory, because the host memory is not on the same chip as the GPU and therefore has higher latency and lower bandwidth.

In our scenario, we observed a decent enhancement in the overall performance of the execution process. This improvement primarily stemmed from the direct access of read-only data, approximately 250 MB in size, from the host, thereby circumventing the time-consuming data transfer from host to device.

Even though the kernel execution time experienced a 10x increase, we managed to achieve a net reduction in the overall execution time when compared to the State Machine kernel with no optimizations and the Async kernel, by approximately 20% and 10% respectively (refer to Fig. 7).

This improvement can be attributed to our approach of bypassing the explicit copying of input text data to the device. Instead, we utilized the PCIe channels to directly access data from the host memory while operating on the device. To prevent the system from paging out our input data, we strategically 'pinned' the data within the host memory. The pinning process essentially locks the data in place, preventing it from being swapped out to disk and thus, eliminating the time spent on paging.

5.6. Unified Memory

Unified Memory in CUDA simplifies memory management by presenting a single memory space accessible by all GPUs and CPUs in the system. Instead of manually handling data transfers between the host (CPU) and device (GPU), developers can allocate and use data in Unified Memory without worrying about where the data physically resides.

The CUDA system automatically migrates data between host and device to optimize performance. If the GPU accesses data that is not on the device, the system automatically copies the required data from host to device. This process is known as "demand paging". The system also migrates data to the host when the CPU accesses data that has been modified on the device.

In our case, we considered using Unified Memory to simplify our memory management and potentially increase performance. We allocate the managed memory using ‘cudaMallocManaged’ and we directly read the file content into the managed memory. However, we notice that the time to load the file is about 80 ms, which is the longest amongst all our implementations. Since CUDA does not specify where the unified memory is placed, we conjecture that CUDA initially places the pages in the device memory. Once we start loading the input file into those pages on host, we miss all those pages on the host as they reside in the device. This leads to paging in all those pages to host, leading to extra overhead in performance. Therefore we used the CUDA function ‘cudaMemPrefetchAsync’ to serve as a hint to load the uniform memory pages on the host before we read the file. This helped improve loading time by more than 20%, resulting in file loading time in the range of 60 milliseconds. This greatly improved the overall performance as well, resulting in the best performance amongst other kernel implementations so far (refer to Fig 7).

6. Evaluation

6.1. Dataset

The National Center for Biotechnology Information (NCBI) is a branch of the United States National Library of Medicine, a part of the National Institutes of Health (NIH). NCBI provides access to a variety of resources that allow researchers to search, download, and analyze data related to genes, proteins, diseases, and other aspects of biology and medicine. These resources include the PubMed biomedical literature database, the GenBank DNA sequence database, the BLAST sequence similarity search tool, and many others.

The data being downloaded from the NCBI database is the genomic sequence data for human chromosomes. Specifically, from the Nucleotide Core Database, which contains a collection of sequences from all traditional divisions of GenBank, EMBL, and DDBJ excluding EST, STS, GSS, or phase 0, 1 or

2 HTGS sequences.. Each file being downloaded represents a different human chromosome. The human genome is made up of 23 pairs of chromosomes - 22 pairs of numbered chromosomes, from 1 through 22, and one pair of sex chromosomes, X and Y. Each chromosome is a long, continuous thread of DNA - a sequence of nucleotides. Thus, the downloaded files together represent the majority of the human genome.

The chromosomes being downloaded are:

- **Chromosome 1**: This is the largest human chromosome. It carries a large number of genes and is estimated to have about 2,000 to 2,100 genes.
- **Chromosome 2**: This is the second largest human chromosome. The fusion of two ancestral chromosomes resulted in Chromosome 2. This chromosome contains 1,491 genes.
- **Chromosome 3**: It is the third largest human chromosome and contains an estimated 1,100 to 1,200 genes.
- **Chromosome 4**: This chromosome is intermediate-sized, containing about 1,000 to 1,100 genes.
- **Chromosome 5**: Contains an estimated 800 to 900 genes.
- **Chromosome 6**: Contains an estimated 1,000 to 1,100 genes. This chromosome also contains the Major Histocompatibility Complex, which plays a critical role in the immune system.

Each chromosome contains many genes that are the coded instructions for creating proteins, which are the building blocks for the body and its functions. By downloading these chromosomes, one is getting the sequences of nucleotides (adenine, guanine, cytosine, and thymine) that make up the DNA for each chromosome.

In our experiment, we generate a dataset containing 1G (10^9) characters out of the 6 chromosomes as the text where we search a randomly generated pattern.

6.2. 2-Bit Compression

The data we have collected is stored in FASTA, which is a text based format that stores the genome sequence with characters A, T, C and G. However, there is a lot of redundant information if we simply store the genome sequence in an array of 8-bit chars. Since we only have four different types of characters, it's reasonable to compress each character from 1 byte into 2 bits. Originally, the text A is 01000001 in binary, C is 01000011, G is 01000111 and T is 01010100. We could replace 8 bits with 2 bits, representing A as 00, C as 01, G as 10, and T as 11. We call this technique *2-bit compression*. For instance, in the text "AGCC", we could 2-bit compress it from 32 bits into just 8 bits.

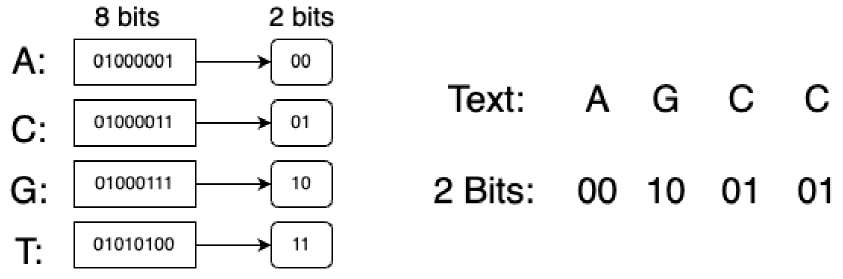

Fig 3: The left figure shows the transformation of elements in the gene dataset from 8 bits character type into 2 bits type; the right figure shows an example of transforming text into 2 Bits type.

By 2-bit compression, we compress our 1G text into 250MB, reducing the global memory usage and the global memory reads by a factor of 4.

6.3. Kernel Runtime Results

To evaluate our kernel, we run them on a Titan V, on RAI servers. We randomly generate patterns of length 13, and launch different kernels to search the random pattern in the 1G text. The pattern length is chosen so that the selectivity is appropriate: on average there are about 16 matches in the 1G text. If we further increase the pattern length to 16, the selectivity will be much lower and there would be almost no occurrence of the pattern in the whole text. 13 is chosen for another reason that the start of each chunk is aligned to byte after overlapping to achieve the best performance.

The runtime of our kernels are shown in Figure 4.

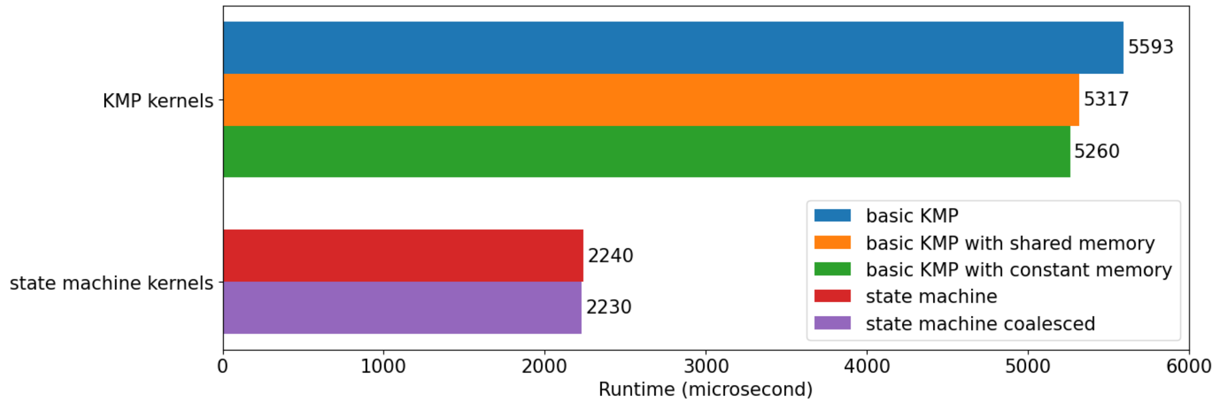

Figure 4: The runtime of kernels searching a random length 13 pattern in 1G text.

It is clearly shown that the state machine algorithm is about 2.5x faster than the baseline KMP algorithm on the GPU. Using shared memory and constant memory also helps improve the performance by about 5%.

We profile all our kernels using Nsight Compute. We use the KMP shared memory kernel to illustrate what the bottlenecks of the naive KMP algorithm are on the GPU. The most prominent one is the branch divergence. The branch efficiency is only about 65%, which impedes the performance. Also the DRAM throughput is only about 7%, which is too low for a memory bound kernel. After switching to the state machine algorithm with shared memory, the branch efficiency rises to above 99% and DRAM throughput to above 17%, and therefore our optimization is successful.

Diving deeper into the analysis, we focus on some memory metrics as computed by the profiler:

- Memory Throughput:
    - Memory throughput is highest in the KMP Shared Memory Kernel and the State Machine Shared Memory Kernel, suggesting efficient use of memory bandwidth.

- The lowest memory throughput is seen in the State Machine Coalesced Zero Copy Kernel, indicating a possible bottleneck with memory access in this case.

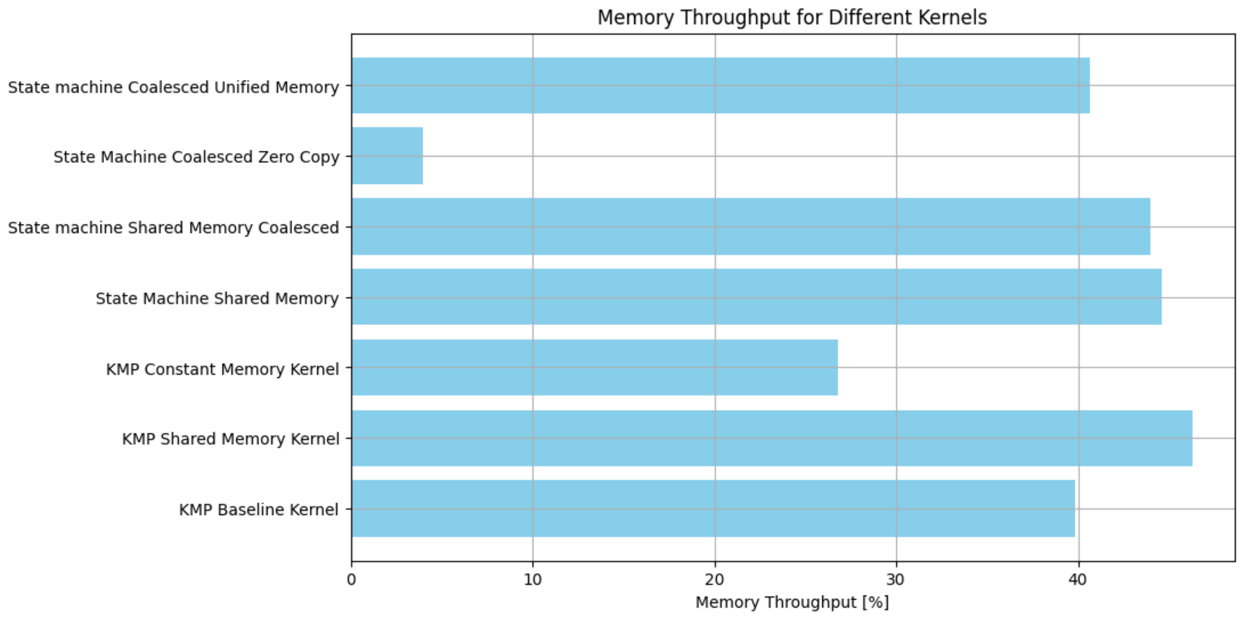

Fig 5: Memory Throughput[%]

- DRAM Throughput:
  - The State Machine Shared Memory Kernel and State Machine Shared Memory Coalesced Kernel show the highest DRAM throughput.
  - However, the State Machine Coalesced Zero Copy Kernel shows a 0% DRAM throughput. This is expected as the Zero Copy approach accesses host memory directly, bypassing the need for device DRAM access.

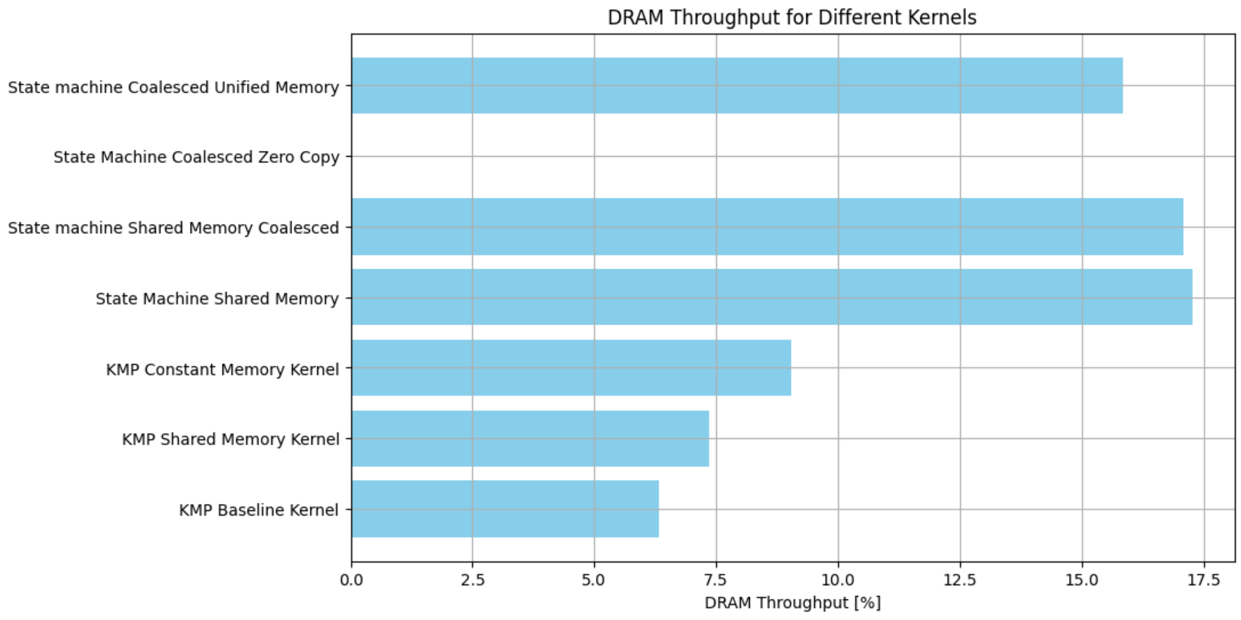

Fig 6: Device DRAM Throughput[%]

6.4. Total Runtime Results

In this section, we analyze the effects of the optimizations that improve total runtime. The total runtime includes three parts: the time to load the dataset file into the memory, the time to transfer data to and from the GPU, and the time to execute the kernel. We compare the performance of three optimizations: asynchronized data transfer, zero-copy and unified memory against no optimization. We implement all these optimizations based on the best kernel we have obtained, i.e. the state machine coalesced kernel.

The total runtime of our optimizations is shown in figure 7. Note that for all our optimizations, data transfer to the GPU is overlapped with the kernel execution.

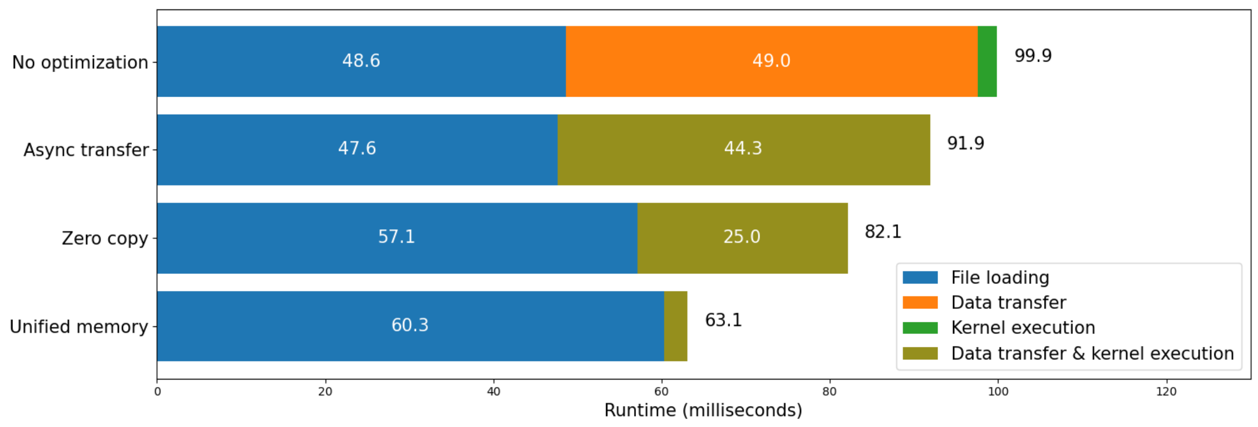

Figure 7: The total runtime of four different optimizations, based on the state machine coalesced kernel.

The state machine coalesced kernel, which we use as the baseline in this section, takes about 100 ms to complete the whole searching process. The time is mostly spent on loading the file and copying the data to the GPU. Through asynchronous data transfer, the total execution time goes down by 10%. Zero copy and unified memory introduce extra overhead during the file loading process, but they reduce the time of data transfer and kernel execution, thus achieving speedup of *1.2x* and *1.58x* respectively.

7. Future work

In the future, our objective is to formulate a kernel with the ability to dynamically decide when to engage a shared memory buffer. Additionally, we only utilize a single pattern for string matching. As a subsequent goal, we aim to adopt the AC state machine to facilitate multiple pattern matching. Moreover, the coalescence of KMP is challenging due to the stridden access pattern and limited shared memory. We will continue to think of other ways of coalescing memory access to boost performance.

8. Conclusion

In this project we have tried to optimize the KMP algorithm on the GPU using CUDA. We have implemented the optimizations of two algorithms: the naive KMP algorithm and the DFA algorithm. We have also explored three optimizations that focus on improving the total runtime. The best kernel we have got is **2.5x** faster than the baseline kernel, utilizing memory coalescing. We have also cut down the total execution time by a factor of **1.58** using unified memory.

9. References

[Knuth77] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Com-
put. 6, 323–350 (1977)

[Boyer77] Boyer, Robert S., and J. Strother Moore. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

[Cao11] Cao, Panwei, and Suping Wu. "Parallel research on KMP algorithm." 2011 International Conference on Consumer Electronics, Communications and Networks (CECNet). IEEE, 2011.

[Lin13] Lin, Kuan-Ju, Yi-Hsuan Huang, and Chun-Yuan Lin. "Efficient parallel knuth-morris-pratt algorithm for multi-GPUs with CUDA." Advances in Intelligent Systems and Applications-Volume 2: Proceedings of the International Computer Symposium ICS 2012 Held at Hualien, Taiwan, December 12–14, 2012. Springer Berlin Heidelberg, 2013.

[Rasool12] Rasool, Akhtar, and Nilay Khare. "Parallelization of KMP string matching algorithm on different SIMD architectures: Multi-core and GPGPU's." International Journal of Computer Applications 49.11 (2012).

[Nagaveni14]Nagaveni, V., Raju, G.: Various string matching algorithms for DNA sequences to detect breast cancer using CUDA processors. Int. J. Eng. Technol. (2014)

[Neungsoo20] Park, Neungsoo, Soeun Park, and Myungho Lee. "High performance parallel KMP algorithm on a heterogeneous architecture." Cluster Computing 23 (2020): 2205-2217.