Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

# CUSMART: effective parallelization of string-matching algorithms using GPGPU accelerators*

Adnan OZSOY‡[1], Mengu NAZLI[1], Onur CANKUR[2], Cagri SAHIN[3]

[1]*Department of Computer Engineering, Hacettepe University, Ankara, Turkiye*

[2]*Department of Computer Science, University of Maryland, College Park, USA*

[3]*Department of Computer Engineering, Gazi University, Ankara, Turkiye*

E-mail: adnan.ozsoy@hacettepe.edu.tr; mengu@hacettepe.edu.tr; ocankur@umd.edu; cagrisahin@gazi.edu.tr

Received Feb. 6, 2024; Revision accepted July 23, 2024; Crosschecked

**Abstract:** This study presents a parallel version of the String Matching Algorithms Research Tool (SMART) library, implemented on Nvidia's Compute Unified Device Architecture (CUDA) platform, and utilizes general-purpose graphics processing unit (GPGPU) programming concepts to enhance performance and gain insight on the parallel versions of these algorithms. We have developed the CUDA-enhanced SMART (CUSMART) library, which incorporates parallelized iterations of 64 string matching algorithms, leveraging the CUDA application programming interface (API). The performance of these algorithms has been assessed across various scenarios to ensure a comprehensive and impartial comparison, allowing for the identification of their strengths and weaknesses in specific application contexts. We have explored and established optimization techniques to gauge their influence on the performance of these algorithms. The results of this study show that the potential highlight of GPGPU computing in string matching applications is highlighted by the scalability of algorithms, suggesting significant performance improvements. Furthermore, we have identified the best and worst performing algorithms in various scenarios.

unedited

**Key words:** String matching; Parallel programming; GPU programming; GPGPU; Nvidia; CUDA; SMART
https://doi.org/10.1631/FITEE.2400091

## 1 Introduction

String matching holds significant importance within the realm of text processing. It serves as an essential component utilized by operating systems' functional software. Even as data are shared and stored in numerous formats, text continues to prevail as the predominant form of information handling. This becomes particularly evident in literature, wherein data consist of extensive corpora. Similarly, in the realm of computer science, large quantities of data are stored in linear data files. Molecular biology presents another instance, wherein biological molecules are approximated as sequences of more fundamental building blocks such as amino acids or nucleotide, treated as strings. The programming methodologies emphasized in string matching serve as foundational paradigms in various other fields of information technology, including computer security (Lu and Fu, 1978), bio-informatics (Maxime and Rytter, 1995; Han et al., 2007), social media content processing (Tian et al., 2012), data mining (Sellis, 1998), data compression (Ziv and Lempel, 1977), coding theory (Lin and Och, 2004; Petrakis, 1993), and numerous other fields.

In computer-centric domains, utilization and accumulation of data are growing continually. Correspondingly, the processing and computational requirements for these data have been escalating. Particularly within the domains of high-performance

‡ Corresponding author
* Project supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with grant number 117E142
ORCID: Adnan OZSOY, https://orcid.org/0000-0002-0302-3721
© Zhejiang University Press 2024

computations and big data, this concern has gained heightened importance in recent years. The extension of calculation time resulting from the expansion of data volume not only leads to notably prolonged response times in string-matching applications but also intensifies the performance demands on string-matching algorithms. As a result, the reduction of calculation times becomes crucial to ensure system availability. This leads to the need for new solutions and hardware support to effectively manage the escalating demands.

Parallel processing serves as the primary solution to reduce calculation times in high-performance computations, with the central processing unit (CPU) having historically underpinning the foundation for parallel calculations (Ceruzzi, 2003). The processing of string-matching algorithms in parallel systems has been previously investigated, with a focus on tackling individual algorithms and aiming to achieve higher performance against CPUs specifically for those particular algorithms (Ashkiani et al., 2016). However, due to limitations such as the finite number of cores in CPUs, the concept of general-purpose CPU (GPCPU) has gained prominence as an alternative that offers significant computational power. Therefore, parallel computing with GPCPU can be utilized for string matching.

In this paper, we present the CUDA-enhanced String-Matching Algorithms Research Tool (CUSMART), applied for a study on parallelization in general-purpose graphics processing units (GPGPUs); parallelization is achieved by implementing classic serial string-matching algorithms on GPUs and applying several optimization techniques. To conduct our study, we leverage the String Matching Algorithms Research Tool (SMART) library, which offers a serial implementations of string-matching algorithms (Faro et al., 2016). This tool serves as an efficient and flexible resource, specifically designed to facilitate the development, testing, comparison, and evaluation of 64 distinct string-matching algorithms. In this respect, these string-matching algorithms have been redesigned and explored, taking into consideration the GPU architecture via NvidiaâĂŹs Compute Unified Device Architecture (CUDA) application programming interface (API). The findings of the study illustrate a remarkable average speedup of up to $51^x$ when compared to the serial CPU versions of these algorithms. This highlights the significant potential of GPGPU computing in the context of string-matching applications. More importantly, these speedup enhancements were achieved through the scalable nature of the algorithms implemented and executed under specific hardware configurations and datasets. The main contributions of this paper can be summarized as follows:

- The parallelization of 64 string-matching algorithms from the literature has been achieved using the CUDA platform.
- The collection of these algorithms under a single open-source library called CUSMART that offers a user-friendly and comprehensive testing environment.
- The enhanced performance in string-matching algorithms through parallelization is highlighted.
- Identification of the most and least effective algorithms under numerous scenarios is carried out and several optimizations are explored.

The remainder of this paper is organized as follows. Section 2 provides background information essential for presenting the features of the CUSMART library. In Section 3, we discuss related works from the literature. Section 4 presents the implementation details of the CUSMART. Section 5 describes the methodology of our study, including testing platform, test scenarios, and the results of the study, including potential threats to its validity. Section 6 provides data analysis and discussion of results. Finally, Section 7 presents our conclusions and future work.

## 2 Background

### 2.1 String matching

Since 1970, over 80 string-matching algorithms have been proposed (Faro et al., 2016). These algorithms, which we will subsequently refer to as 'string matching algorithms', can be categorized into three distinct groups based on the primary strategies that they apply. These categories include comparison-based algorithms, automata-based algorithms, and bit-parallel algorithms.

The earliest examples of proposed string-matching algorithms proposed in the field utilize

unedited

strategies based on character comparison, such as Brute Force (Cormen et al., 2009), Morris Pratt (Weiner, 1973), maximal shift (Kadhim and AbdulRashidx, 2014), and Boyer Moore Horspool with q-grams (Horspool, 1980). These algorithms analyze the pattern prior to the search operation to gain insights into its composition. The acquired information is frequently stored in auxiliary arrays known as *'shift tables'*. During the search operation, if mismatches occur, these tables are utilized to execute improved shifts and avoid unnecessary character comparison operations. When an algorithm is limited to sequentially examining individual characters of the text, its optimal complexity can be denoted as $\mathcal{O}(n)$. Frequently, it is feasible to conclude the search operation without examining every character of the text, leading to sublinear average execution complexity. The optimal average time complexity for matching in a random string is denoted as $\mathcal{O}(n \cdot \frac{\log_\sigma m}{m})$ (Yao, 1979), where $n$ represents the length of the text string, $m$ represents the length of the pattern string, and $\sigma$ represents the size of the alphabet used in the strings. Numerous algorithms achieve this complexity. However, even algorithms with a sublinear average-case operation might necessitate scrutinizing every character of the text in worst-case scenarios. Moreover, many of these algorithms demonstrate even worse worst-case performance with a complexity of $\mathcal{O}(nm)$ (Faro and Lecroq, 2010, 2013). While the character comparison-based strategy may not hold the same level of dominance in recent years as it once did, it still represents the prevailing approach for the majority of string-matching algorithms proposed up to the present day.

Automata-based algorithms play a crucial role in the realm of string matching due to their remarkable efficiency in handling such tasks. One of the pioneering approaches utilizing this technique for string matching was through the application of the deterministic finite automaton (DFA) concept, which achieved linear time complexity in the search process, as the DFA processes each character of the input text exactly once, transitioning from one state to another according to its predefined transitions and maintaining a constant processing time per character due to its fixed number of states and transitions, thus yielding a linear time complexity overall (Cormen et al., 2009). Although it is essential to clarify that DFA itself is not an algorithm but rather a theoretical concept, its implementation as an algorithm can significantly contribute to the advancement of string-matching algorithms. Another notable algorithm is the backward Directed Acyclic Word Graph (DAWG) matching algorithm, which achieves the optimal lower bound of $\mathcal{O}(n \cdot \frac{\log_\sigma m}{m})$ time complexity for the average case. Both of these algorithms exploit the benefits of utilizing finite automata for string-matching tasks. The efficiency of a given algorithm within this category depends on factors such as the automaton used for pattern representation and the simulation technique of the automata, if applicable.

Bit-parallelism is a technique rooted in the efficient simulation of nondeterministic automata (Faro and Lecroq, 2013). Some of the bit-parallel algorithms include the Shift-Or (Baeza-Yates and Gonnet, 1992), Bit-Parallel Wide Window (He et al., 2005), and factorized backward nondeterministic DAWG matching (BNDM) (Navarro and Raffinot, 1998). This approach capitalizes on the inherent characteristics of computing units, whereby bitwise operations can be executed in parallel, word by word. This parallelism is leveraged to reduce the number of instructions needed for comparisons during the search phase. The processor word size is the amount of data that a CPU's internal data registers can hold and process at one time. By taking advantage of this parallelism, the processor can handle multiple bits at once instead of processing each bit individually, reducing the number of instructions needed for comparisons. The required operations can be significantly reduced by a factor of $w$, where $w$ represents the number of bits within the processor's word size. This property results in noteworthy speed improvements, particularly when the pattern length $m \leq w$, allowing the pattern to fit within a processor's single word size and requiring only one operation for parallel comparison. However, the effectiveness of bit-parallel algorithms diminishes as the pattern length surpasses $w$, leading to a decline in performance as $m/w$ increases.

### 2.2 Parallel computing

Over the past decade, there has been a significant advancement in high-performance computing, largely attributed to the rise of heterogeneous architectures that combine GPUs and CPUs. This emergence has marked a fundamental transformation in

unedited

the field of parallel programming. When tackling problems using computer programs, it is common to break down the task into discrete calculations, each focused on a specific aspect of the problem. Primarily, parallelism in applications can be categorized into two main types: task-level parallelism (Girkar and Polychronopoulos, 1995; Lin and Huang, 2021), which involves parallelizing tasks or processes; and data-level parallelism (Blelloch, 1990), which focuses on parallelizing operations on data. Additionally, hybrid parallelism integrates both task and data level parallelism strategies within parallel computing environments (Subhlok et al., 1993).

In task-level parallelism, the tasks constituting a job are distributed among processing units and executed simultaneously. This method assumes that independent tasks are readily available for execution. Conversely, data-level parallelism breaks down the primary task into subtasks, each handling a portion of the overall data. The objective of data-level parallelism is to distribute data across processing units and perform the same computation in parallel.

Two common strategies are applied for partitioning data in data-parallel applications: block partitioning and cyclic partitioning. In the block-partitioning strategy, the data are grouped into blocks, with each block assigned to a specific thread. This means that each thread handles only one block of data, resulting in as many data partitions as number of threads. The cyclic partitioning approach involves threads processing multiple segments of the data. After completing the processing of one partition, a thread proceeds to another partition, typically by skipping ahead in memory by a number of partitions equivalent to the thread count. This enables each thread to work on multiple noncontiguous parts of the data, ensuring efficient utilization of processing resources.

In hybrid parallelism, tasks or processes are parallelized while also parallelizing operations on the data being processed. This approach aims to leverage the strengths of both task and data level parallelism to achieve optimal performance and efficiency in parallel computing applications.

### 2.3 GPGPU with CUDA

While CPU cores are engineered for intricate logic operations and tailored for sequential workflows, GPU cores are designed to be lightweight, featuring simpler logic capabilities and prioritizing high-throughput operations. The latter have been used in a wider range of applications, leading to the term "general-purpose computing on graphics processing unit". A recent study indicates that 40% of the overall computing power in the TOP 500 supercomputer list is derived from GPU-accelerated systems (Nvidia, 2020). A similar ranking, known as the Green500 (Feng and Cameron, 2007) which evaluates supercomputers based on their floating-point operations per watt performance reveals that 90% of the systems listed incorporate GPU accelerators. This high percentage underscores the power efficiency achieved by heterogeneous systems through the utilization of GPU devices.

CUDA is a versatile parallel-computing platform and programming model designed to utilize the parallel processing capabilities of Nvidia GPU devices. This programming platform is accessible through several widely used programming languages, including C, C++, Fortran, Java, and Python. The CUDA programming model offers mechanisms for structuring threads and accessing memory on the GPU through a well-defined hierarchical framework. This structured approach empowers programmers to design applications for heterogeneous computing using a concise set of decorator commands, facilitating efficient development. The CUDA programming model is primarily asynchronous, featuring numerous nonblocking directives that enable the overlap of CPU-GPU operations to achieve heterogeneous computing.

## 3 Related work

Nvidia's introduction of CUDA in 2007 expanded the accessibility of general-purpose computing on CPUs to a wider range of users. Since then, numerous studies have been published highlighting the enhanced performance achieved through GPU programming across various existing and novel problem domains. In this context, we present several recent studies conducted on GPU-accelerated string-matching algorithms.

Ligowski and Rudnicki (2009) introduced an enhanced parallel version of the Smith Waterman algorithm. Their modified algorithm exhibited a speedup of $3.5x$ compared to previous GPU implementations, achieving 70% of the maximum theoretical hardware

unedited

performance on their experimental setup. Addressing a similar objective, Hains et al. (2011) introduced high-performance parallel implementation of Smith Waterman (CUDASW++), which was implemented using the CUDA platform. Identifying a bottleneck within the intratask aspect of the algorithm, they concentrated on enhancing this particular subprocess. Through their tests on Nvidia Tesla C1060 and Nvidia Tesla C2050 GPUs, they reported a performance enhancement of approximately 25% in comparison to the unoptimized GPU implementation of the algorithm.

Peng and Chen (2010) presented an improved version of the Nrgrep, a well-known command-line utility used in the UNIX operating system. Their implementation, known as CUgrep utilized a GPU-based multistring-matching algorithm rooted in the BNDM algorithm, resulting in reported performance gains of $40x$ compared to Nrgrep on an Nvidia Tesla C1060 GPU and Intel Xeon 5110 CPU.

Parallel implementation of the Boyer Moore Horspool algorithm has also been investigated. For example, Zhou et al. (2011) explored its characteristics and various optimization approaches such as shared memory utilization, access regulation, and parallelism granularity adjustments. Their findings revealed a speedup factor of 40 for their GPU parallel algorithm in comparison to a serial CPU implementation, using an Intel Xeon CPU and an Nvidia GTX275 GPU. Yong and Karuppiah (2013) investigated a hash search algorithm executed on a GPU and conducted a performance comparison with parallel versions of the BruteForce algorithm and the Boyer Moore Horspool algorithm. They used two distinct Nvidia GPU devices: one utilizing the Fermi architecture, specifically the C2075; and the other incorporating the Kepler architecture, identified as K20c. While their innovative hash search algorithm exhibited slower kernel-wise performance in comparison to the Boyer Moore Horspool algorithm, a fascinating contrast emerged when considering the overall performance on the Kepler architecture. In particular, the novel algorithm showcased a remarkable speedup by a factor of 8.34. This substantial enhancement was attributed to the intricate nature of the Boyer Moore Horspool algorithm, involving the generation of supplementary shifting and lookup tables. The utilization of these auxiliary data structures imposed a heightened data transfer load on the GPU, which contributed to the observed performance difference. Additionally, the authors documented a noteworthy speedup factor of $23x$ through the strategic implementation of shared memory on the GPU. This optimization technique yielded considerable performance improvements. In its entirety, the novel algorithm achieved an impressive overall speedup of $150x$ when compared to the global memory-utilizing version of the Boyer Moore Horspool algorithm. This outcome underscores the potential of tailored algorithmic implementations and GPU-specific optimizations to significantly elevate computational efficiency.

The Aho-Corasick algorithm has been frequently investigated for parallelization. Tran et al. (2012) introduced their implementation of the Aho Corasick algorithm using the CUDA architecture on GPUs and investigated its memory-efficient parallelization. Their implementation prioritized memory access optimization and explored various methods using different memory utilization strategies. When evaluated on a system featuring Nvidia 9500GT GPU and Intel Core2Duo 2.2GHz CPU, their experimental outcomes demonstrated a maximum speedup factor of $15.72x$ on the GPU in comparison to the single-core CPU version. Pungila and Negru (2012) proposed a method for implementing compressed Aho Corasick and Commentz Walter automata on GPUs, aimed at intrusion detection and virus scanning. This technique leverages intratask parallelism, concurrently executing regular expression matching using the Aho Corasick algorithm on one thread and searching for regular patterns using the Commentz Walter algorithm on another thread. Their hybrid approach merges the rapid execution benefits of the Aho Corasick algorithm with the memory efficiency of the Commentz Walter algorithm. This integration resulted in a remarkable 38-fold increase in bandwidth compared to serial CPU implementations, while demanding nearly $22x$ less memory than comparable approaches. Tran and Lee (2013) implemented a multistream Aho Corasick algorithm on GPUs, investigating its performance with an extensive pattern set. They aimed to evaluate HyperQ technology's impact on performance in the Kepler architecture. Using an Nvidia Tesla K20 GPU (Kepler architecture) and an Nvidia GeForce GTX 285 for comparison, they observed that the Nvidia Tesla K20 GPU achieved a peak throughput of 585

unedited

Gbps after testing with up to 20,000 patterns. This throughput was $1.45x$ higher than the Nvidia GPU without HyperQ technology. Zha and Sahni (2013) devised GPU adaptations of the Aho Corasick algorithm and the Boyer Moore algorithm. By applying an Nvidia Tesla GT200 GPU alongside an Intel Xeon 2.8GHz quad-core CPU, they observed a noteworthy speed-up ranging between $3.1x$ and $9.5x$ for these algorithms when contrasted with their single-threaded implementations.

Xu et al. (2013) introduced implementations of the multiple approximate string matching (MASM) algorithm and a BPR variant optimized for multiple pattern matching on a GPU. Utilizing an Nvidia GeForce 310M GPU alongside an Intel i3 2.27 GHz CPU, they achieved a remarkable speedup by a factor of 28. This performance enhancement was attained by comparing their GPU implementation against a single-threaded CPU implementation.

The performance of the Knuth Morris Pratt algorithm executed on a GPU was assessed in comparison to its single-core and multicore CPU counterparts (Rasool and Khare, 2012). These researchers utilized the OpenCL framework to facilitate the GPU and multi-core CPU implementations of this algorithm. Notably, their experimental setup featured an Advanced Micro Devices (AMD) Radeon HD 6800 series GPU, diverging from the commonly used CUDA-enabled Nvidia devices. After extensive testing across various text and pattern configurations, they observed that the unoptimized OpenCL GPU implementation outperformed the serial CPU variant by a factor of 9.52. Bellekens et al. (2013) conducted a comparison involving their GPU implementation of the Knuth Morris Pratt algorithm. Their evaluation featured an Nvidia Tesla K20M GPU, pitted against two Intel Xeon E5-2620 CPUs executing the CPU version of the algorithm. The assessment encompassed diverse string sizes and alphabet sizes, and the authors explored various optimization techniques such as shared memory utilization and loop unrolling. Their findings highlighted a substantial speed gain of $29x$, akin to the outcomes observed by Xu et al. (2013). This speed gain was achieved when using the parallel GPU version of the algorithm, underscoring its superiority over the serial CPU version. Lee et al. (2015) introduced a hybrid pattern-matching algorithm for deep packet inspection, utilizing both GPU and CPU resources based on task types. They implemented the Aho Corasick and Knuth Morris Pratt string-matching algorithms, comparing their performance and power efficiency using CPU and GPU versions of the base algorithms. Their experimentation used an Intel Core i7 3770 CPU and an Nvidia GeForce GTX680 GPU. The Aho Corasick hybrid algorithm outperformed the CPU version by $3.4x$ and the GPU version by $2.7x$, demonstrating superior efficiency compared to other algorithms tested. This research showcased the potential of combined GPU CPU processing for enhanced pattern-matching tasks.

In the context of cancer detection within DNA sequences, Adey, SP (2013) conducted tests involving parallel versions of partial string-matching algorithms, executed on GPUs. This endeavor aimed to enhance the efficiency of the search process. The study implemented advanced multipattern algorithms âĂŤ multiple shift minimum pattern matching algorithm (MSMPMA), inverted Knuth Morris Pratt multiple pattern matching (IKPMPM), enhanced prefix matching for multiple pattern searching and preprocessing (EPMSPP), and improved Aho Corasick for efficient multiple pattern matching algorithm (IAEMA) âĂŤ in both serial CPU and GPU versions. The experimental setup featured an Nvidia Tesla C2070 GPU accelerator device paired with an Intel Core i7 CPU. The outcome of their parallel algorithm analysis was notable: a significant speed-up by a factor of 30 was observed when comparing the parallel GPU implementations to the serial CPU versions of the respective algorithms.

Nagaveni and Raju (2014) conducted a study utilizing string-matching techniques to analyze DNA sequences for the purpose of breast cancer detection. Their approach involved implementing the algorithm on an Nvidia Tesla C2070 GPU. The findings of their study were remarkable: the GPU implementation exhibited a substantial efficiency gain. Specifically, the GPU implementation was reported to be $30x$ more efficient compared to the serial implementation carried out on an Intel Core i7 CPU.

Kouzinopoulos et al. (2015) conducted a comprehensive comparison of various algorithms using an Nvidia GTX 280 GPU and an Intel Xeon 2.4 GHz CPU. They implemented Set Backward Oracle Matching, Wu Manber, Set Horspool, Aho Corasick, and Set Occurrence Graph (SOG) multiple pattern-matching algorithms. The study revealed substan-

unedited

tial speed gains, ranging from a factor of 2.5 to 10.9, when utilizing the GPU versions of these algorithms in comparison to their CPU counterparts.

Sharma and Singh (2015) introduced an implementation of the Karp Rabin algorithm tailored for deep packet inspection. Their implementation achieved an impressive 14-fold speed-up in comparison to the CPU implementation. These findings underscore the potential of GPU-based optimizations to significantly enhance the efficiency of various computational tasks. Ashkiani et al. (2016) introduced three GPU-based parallel approaches for the Karp Rabin algorithm. These methodologies include the "cooperative", "divide-and-conquer", and a "hybrid" fusion of both methods. The study found that the cooperative approach, in which scan operations are computed collaboratively by a set of independent threads or processors, was most effective for pattern lengths exceeding 8000 characters. This highlights the benefits of cooperative parallel processing, in which multiple processing units collaborate, compared to task and data parallelism, which involve more independent tasks. In contrast to cooperative parallel processing, the divide-and-conquer method excelled with shorter patterns. The implementation demonstrated a 4.81-fold speed improvement on an Nvidia Tesla K40c GPU. For improved Karp Rabin performance with large patterns, a new parallel two-stage technique was introduced using the divide-and-conquer approach. This involved scanning the text for a smaller subset of the pattern and subsequently validating all potential matches in parallel. The hybrid method divided the text into substrings similar to divide-and-conquer, processing each substring cooperatively within a designated group of processors. The Karp Rabin algorithm was an excellent candidate to test these approaches due to its adaptability to cooperative parallel processingâĂŤa feature not feasible for most string-matching algorithms.

Mitani et al. (2017) introduced a tribrid parallel approach tailored for bit-parallel algorithms like Shift-Or and Wu-Manber algorithms. This innovative method leverages inclusive scan, enhancing the efficiency of bit-parallel algorithms on GPUs. Inclusive scan not only minimizes duplicate searches across threads but also optimizes memory access patterns for enhanced memory efficiency.

A review conducted by Ramos-Frías and Vargas-Lombardo (2017) compiles a collection of GPU-implemented string-matching algorithms, showcasing various techniques. While their selection includes six algorithms to represent the current state of parallel string matching, it is worth noting that these choices might not encompass the entirety of the parallel string-matching landscape.

The majority of discussed studies center on specific algorithms, frequently selecting from a limited range of well-known string-matching techniques. These commonly studied algorithms comprise Boyer Moore, Boyer Moore Horspool, Knuth Morris Pratt, Backwards Nondeterministic DAWG Matching, and Aho Corasick. Code libraries (such as the SMART library) provide researchers with a streamlined platform to readily compare nearly all string-matching algorithms documented in the literature (Faro et al., 2016). These libraries also facilitate the evaluation of their own algorithmic implementations in comparison to others. However, despite ongoing work on parallel GPU implementations for specific and limited string-matching algorithms, there is a lack of an equivalent and comprehensive framework for parallel GPU algorithms that can be considered the parallel counterpart of these tools. A parallel GPU toolkit could greatly benefit researchers studying parallel string-matching algorithms by enabling benchmarking against established methods and facilitating easy testing of all available algorithms to identify the most effective candidates. Our primary aim is CUSMART, a research tool for parallel string-matching algorithms developed on the CUDA platform, aimed at filling the gap in parallel GPU operation codebases. With the help of this tool, we also aim to identify the most and least effective algorithms under numerous scenarios (such as varying pattern length, text length, alphabet size, and so on) and several optimizations; this aspect is also missing in the literature.

## 4 CUSMART library

This paper introduces CUSMART, which is a parallel string-matching library built on Nvidia's CUDA architecture.[1] It is an extension of the SMART library, originally designed for single-core CPUs, featuring serial algorithm versions aimed for single-thread execution and is not optimized to use multicore parallel processing on the CPU. Its setup

[1] https://github.com/adnanozsoy/CUSMART

unedited

| Code | Algorithm name | Code | Algorithm name |
|---|---|---|---|
| ac | Apostolico-Crochemore | ksa | Factorized Shift And |
| ag | Apostolico-Giancarlo | lbndm | Backward Nondeterministic DAWG Matching for long patterns |
| aoso2 | Average Optimal Shift Or | mp | Morris-Pratt |
| bf | Brute Force | nsn | Not So Naive |
| bfs | Backward Fast Search, Fast Boyer-Moore | pbmh | Boyer-Moore-Horspool using Probabilities |
| bm | Boyer-Moore | qs | Quick Search |
| bmh_sbndm | Horspool with BNDM test | raita | Raita |
| bndm | Backward Nondeterministic DAWG Matching | rcol | Reverse Colussi |
| bndmq2 | Backward Nondeterministic DAWG Matching with loop unrolling | rf | Reverse Factor |
| bndmq4 | Backward Nondeterministic DAWG Matching with q-grams | sa | Shift And |
| br | Berry Ravindran | sbndm | Simplified Backward Nondeterministic DAWG Matching |
| bsdm | Backward Suffix Non-deterministic Regular DAWG Matching | sbndm_bmh | Backward Nondeterministic DAWG Matching with Horspool Shift |
| bww | Bitparallel Wide Window | sbndm2 | Simplified BNDM with loop unrolling |
| bxs | BNDM with Extended Shift | sbndmq2 | Simplified Backward Nondeterministic DAWG Matching with q-grams |
| col | Colussi | sebom | Simplified Extended Backward Oracle Matching |
| dfa | Deterministic Finite Automaton | sfbom | Simplified Forward Backward Oracle Matching |
| ebom | Extended Backward Oracle Matching | simon | Simon |
| faoso2 | Fast Average Optimal Shift Or | skip | Skip Search |
| fbom | Forward Backward Oracle Matching | smith | Smith |
| fdm | Forward DAWG Matching | smoa | String Matching on Ordered Alphabet |
| fjs | Franek Jennings Smyth | so | Shift Or |
| fndm | Forward Nondeterministic DAWG Matching | ssabs | Suffix-based String Matching with Adaptive Block Sampling |
| fs | Fast Search | tbm | Turbo Boyer-Moore |
| fsbndm | Forward Simplified Backward Nondeterministic DAWG Matching | tndm | Two-Way Nondeterministic DAWG Matching |
| fsbndmq20 | Forward Simplified BNDM using q-grams and s-forward characters | trf | Turbo Reverse Factor |
| gg | Galil Giancarlo | ts | Tailed Substring |
| hor | Horspool | tsw | Two Sliding Window |
| ildm1 | Improved Linear DAWG Matching | tunedbm | Tuned Boyer Moore |
| ildm2 | Improved Linear DAWG Matching 2 | tvsbs | Trie-based Variable-length Suffix Block Sampling |
| kbndm | Factorized Backward Nondeterministic DAWG Matching | tw | Two Way |
| kmp | Knuth Morris-Pratt | ww | Wide Window |
| kr | Karp Rabin | zt | Zhu Takaoka |

**Table 1 CUSMART algorithms and their corresponding codes**

unedited

prioritizes code conciseness, correctness, and readability, but it hinders the ability to conduct multi-threaded performance comparisons due to the absence of parallel algorithm versions in the library. Developing parallel implementations of these algorithms is complex, as achieving parallelism involves modifying fundamental aspects of the procedures and lacks straightforward solutions.

Creating CUSMART necessitated extensive alterations to the codebase due to the divergent design priorities of the original library. In the SMART library, every algorithm's executable file initiates with memory allocation, followed by the reading and processing of text data. Finally, the allocated memory space is freed when the program's life cycle ends. This approach leads to repetitive reading of the text file during batch algorithm tests, which is undesirable. Compiling the algorithms into a single executable file prompts the need to modify the memory allocation approach of algorithm functions. In the serial library, the search subject string and auxiliary array structures rely on fixed-size static memory blocks, intentionally chosen larger to avert memory scarcity issues during tests with large files. In the context of consolidated compilation, each source file allocates a unique memory space, maintained statically throughout the program's lifespan. Yet, such an approach leads to substantial memory demandsâĂŤaround 33,000$x$ the space required for a 10-character string search. Given the constraints that this situation imposes on resource-limited systems, our project's codebase adopts dynamic memory allocation, offering efficient memory management by utilizing needed space and releasing it after operation.

Table 1 shows the 64 algorithms in our CUSMART library, which are parallel versions of string-matching algorithms from the SMART library, along with their corresponding codes. For example, the "Fast Search" algorithm is represented by the "fs"

code.

### 4.1 Optimization and implementation considerations

In the code that we integrated from the SMART library, we introduced modifications to the algorithm function outputs. The original code used a single variable to track the frequency of string matches, without recording the actual positions of these matches within the string. However, practical string-matching functions, crucial for numerous applications, require information about the positions at which patterns match within the text string. To fulfill this essential requirement, it became imperative to capture positional data to serve practical purposes. Although the match count could subsequently be calculated from these data, the reverse is not feasible. The positional data from the algorithms are stored in an array, occupying memory space equivalent to that of the text string. When a match is detected starting from the n-th character of the text string, the n-th value of the position array transitions from "0" to "1". Ultimately, the total match count is derived by performing a reduction operation across the positional data array. Thus, we provide an additional feature in addition to SMART data, whereby we can give not only the match count but also the match locations. Throughout our testing, we meticulously timed this operation to ensure that it does not interfere with algorithm timing measurements.

Task parallelism falls into two classes (Quinn, 2004). "*Inter-task parallelism*" entails tasks being individually assigned to threads and executed concurrently. Conversely, "*intra-task parallelism*" designates a single task assigned to a block of threads, collaborating to execute portions of the task in parallel. In the context of string-matching algorithms, the primary tasks encompass preprocessing and searching. However, these two tasks often are not amenable to parallel execution since the search phase depends on preprocessing. While some string-matching algorithms allow the search phase to be subdivided into sub-tasks, many do not. Notably, CUDA's Single Instruction Multiple Data (SIMD) architecture is apt for parallel execution of a solitary task across extensive data, favoring intratask methodology when it is designed according to CUDA architecture. As such, our parallel implementation methods align with intratask parallelism principles.

The algorithms within our library are designed for parallel string matching over smaller text segments. Threads are allocated to specific portions of the text, using corresponding string-matching algorithms to identify pattern matches. The length of the assigned substring for each thread is termed as the "*stride length*". Smaller stride lengths permit higher thread parallelism but incur notable thread creation overhead for minor tasks. In contrast, larger stride lengths allocate more substantial tasks to threads, reducing parallel work overhead, but limiting thread deployment and diminishing concurrency. Striking a balance is vital, and our experimentation assessed the impact of various stride lengths on algorithm performance.

Efficiently leveraging all processors is crucial in heterogeneous computing. To enhance throughput, the CUDA architecture schedules commands for execution within "*streams*". These streams are primarily used to overlap kernel execution with memory transfers. Given that string-matching kernel executions are generally quicker than memory transactions, they can be concealed behind memory operations, curtailing idle time and enhancing operation speed. Our library encompasses streaming iterations of select algorithms, and our example implementations facilitate straightforward development and testing of streaming versions for other string-matching algorithms. We conducted diverse tests on

unedited

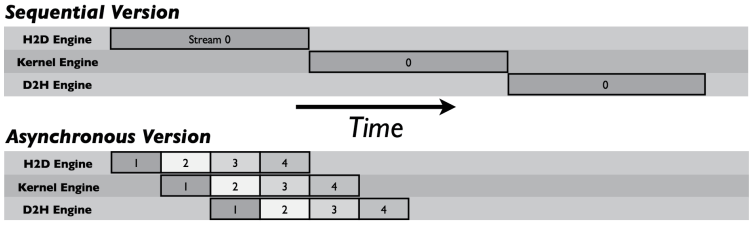


**Fig. 1 Overlapping data transfer and kernel processing (courtesy of Harris, Mark (2012)).**

streaming with varying configurations. Fig. 1 illustrates the resultant behavior.

Given the typically expansive size of the text string, we opted to store the pattern string in constant memory.Within the CUSMART library, an alternative to using global memory for pattern storage is efficiently facilitated by helper functions without requiring any adjustments to the algorithm code. Furthermore, shared memory, another memory space, is devised to facilitate swift memory transactions between threads within the kernel block. However, shared memory cannot be used for all algorithms. Most of the algorithms are designed to make big jumps during the search operation: they do not need to read all characters of the text when processing it. Since we cannot predict which characters need to be read and which are to be skipped before the actual comparison happens, the entire text needs to be transferred to shared memory prior to the search, and many unnecessary memory transactions need to be done. Thus, combining this reasoning with our initial test results, it was realized that moving the search text to shared memory would cause more harm than good, and hence, shared memory was not utilized for this purpose. Instead, we have chosen a few algorithms with the auxiliary data structures resulting from their preprocessing steps to test the feasibility of using shared memory. These data structures are frequently accessed during the operation of kernel. The caching mechanism speeds up the fetching of frequently accessed data, but the cache storage is not guaranteed through the lifetime of kernel. Shared memory offers controlled access to data compared to cache with similar low-latency performance.

Additionally, the CUSMART library offers the capability to utilize pinned memory via a command-line option. Activating the pinned memory option allocates a portion of the host random-access memory (RAM) exclusively for use by the CUDA application.

For parallel string-matching solutions, there is a borderline situation in which half of the pattern is matched by one thread, and another half is matched by another thread. As a solution, we extended the search window for each thread to be (n-1) more character checks at the end of the window, where n is the pattern length. Thus, each thread sees a window that it is supposed to see in a serial implementation process. Fortunately, this still grants us coalescing access to the global memory.

Occupancy serves as a metric to gauge the efficiency of CUDA device utilization during operations. The CUSMART library incorporates helper functions to compute appropriate block and grid dimensions for optimal performance. The algorithms utilized in our tests incorporate these calculations to attain maximum achievable occupancy.

The majority of string-matching algorithms in our library require preparatory steps before the search operation. This involves executing the preprocessing code on the CPU and transferring the results to GPU device memory. Due to longer GPU execution times for the same preprocessing operations as compared to CPU execution, the preprocessing is done on the CPU. These preprocessing functions are called within the wrapper function of the selected algorithm and are timed separately using CPU-side timing functions.

To time the GPU-side operations in our library, event-based recording functions from the CUDA API are used. These functions track the timing for key steps, including memory transactions from the host (CPU) to the device (GPU), the primary search kernel operation, and the reduction operation. Additionally, the preprocessing steps of the algorithms are timed using suitable CPU-side timing functions, as this phase is executed on the CPU.

Unlike the SMART library, our CUSMART library uses a distinct method for tallying match cases. In SMART, preprocessor macros are associated with a sole variable for match count storage. However, this approach proves impractical when the program operates in a multithreaded manner, leading to race conditions and read-write conflicts. To avert these challenges and the slowdown stemming from lock mechanisms applied to resolve single variable access conflicts, CUSMART utilizes an array allocated in the GPU's global memory. This array records match positions, sidestepping the issues encountered in the SMART library. Efficient counting can be accomplished through a technique known as "*reduction*". This technique systematically sums the values within the target array, progressively halving the operation area in each stage. Fortunately, reduction is a well-explored approach that seamlessly adapts to parallel processing. In parallel environments, highly effective reduction functions have been developed. The CUSMART library capitalizes on such an implementation, conducting the reduction operation on the

unedited

GPU side to curtail the volume of necessary memory transfers from the device to the host.

The CUSMART library is thoughtfully structured to facilitate the effortless implementation of new algorithms and enable seamless comparisons with existing ones. The GPU-side algorithms are divided into two core sections: the kernel function and the wrapper function. The wrapper function encapsulates preparatory steps such as memory allocation, transfers, preprocessing, reduction operations, and timing calls requisite before the search phase. To simplify repetitive aspects of the wrapper function, helper functions offer boilerplate code. Conversely, the kernel function houses the central logic of the search algorithm. For specific cases, such as versions using shared memory, the kernel function may include supplementary memory transaction code preceding the search logic. All implemented algorithm wrappers are cataloged in dedicated header files, serving as external references accessible within the application. This well-organized structure aids in algorithm development, comparison, and integration. Incorporating newly added source files necessitates their inclusion in the CMake configuration file to ensure accurate compilation into the library. CMake facilitates cross-platform compilation via a centralized configuration file, enhancing the efficiency and consistency of the compilation process.

### 4.2 Threats to validity

The compilation of the CUSMART library into a single executable file requires careful management to prevent algorithm functions from interfering with one another. While every effort has been made to isolate their operations, unexpected interactions might occur. Certain factors, such as memory caching and GPU warmup time, are challenging to regulate. To mitigate the impact of caching on benchmarks, a preliminary memory read is executed before each operation, providing a consistent memory transfer state. The GPU warm-up time can influence initial runtimes, which is countered by introducing a warm-up kernel call prior to every kernel execution, thus ensuring that subsequent kernels are timed without delays.

System-level variables, such as occupancy, can introduce variance in benchmark results. To counter this, an average time option has been incorporated into the library. By enabling this option and specifying the number of repetitions for the search operation, the library can run the selected algorithm multiple times, producing an average runtime value that smooths out system-related variations.

Additionally, to guarantee precise timing measurements and counteract any potential impact from background system processes, we conducted each test a minimum of 10 times.

## 5 Empirical study

This section outlines the string-matching tests conducted using the CUSMART library. The test scenarios are carefully designed to highlight how different parameters affect performance in typical use cases. This involves evaluating the algorithms in terms of speedup, performance ranking, and text structure scenarios. Furthermore, the study evaluates how hardware optimizations implemented for concurrency affect algorithm performance.

Various datasets are utilized for testing purposes, including novel texts from Wikipedia dump files, microorganism DNA sequences, and synthetic datasets. To ensure accurate timing measurements and mitigate the influence of system background processes, each test is executed a minimum of 10 times. Average timings are reported when deviations between runs are negligible, ensuring reliable performance assessment.

In the remainder of the section, we provide a comprehensive overview of our study design, encompassing the test platform utilized and the datasets used for our evaluation.

### 5.1 Test platform

The computer utilized for the tests is an HP Z8 G4 Workstation equipped with two Intel Xeon Silver 4114 CPUs running at 2.20 GHz with a 9.6 GTs bus speed, 13.75 MB cache, and 10 cores each. The system boasts 128 GB of double data rate fourth (DDR4) 2666 MHz error correcting code (ECC) RAM, configured with 4 32 GB modules. Storage is facilitated by a 256 GB serial ATA solid state drive (SATA SSD) hard disk. The chosen operating system is Ubuntu 18.04. The system uses an Nvidia GeForce GTX 1080 Ti graphics card with 11 GB of memory operating at 484 GB/s bandwidth. The graphics card features 3584 CUDA cores running at 1.5 GHz and is utilized for GPGPU operations. The

unedited

CUDA runtime environment applied is compatible with version 18.04 of Ubuntu, and the CUDA toolkit version 8.0 is installed to support GPU-accelerated computations and programming.

### 5.2 Test datasets

To evaluate the performance of CUSMART, we utilized four distinct datasets as follows:

- *Wikipedia:* This dataset is composed of daily text in English, sourced from a Wikipedia content dump file. Sample text data sizes were chosen as 1 MB, 10 MB, 100 MB, and 1 GB in order to assess how text size affects algorithm performance. We selected four distinct patterns with lengths of 3, 10, 50, and 200 characters, and each of these patterns was searched within four different text strings. This execution process was iterated through serial implementations of all algorithms on the CPU and parallel implementations on the GPU. The subsequent step encompassed the calculation of average speedups achieved through the utilization of parallel versions.
- *Genome:* DNA tests were integrated to analyze algorithm behavior concerning a specific domain such as DNA sequencing issues. The DNA sequence of Drosophila melanogaster (known as fruit fly) served as the source string for select test cases (Adams et al., 2000). By duplicating the sequence, a file was generated for the search operation. Across these tests, distinct patterns with lengths of 3, 10, 50, and 200 were used. These evaluations examined algorithm performance over structured and small alphabets with varying pattern lengths.
- *Synthetic randomized dataset:* We analyzed algorithm performance and its variations across different pattern lengths and alphabet sizes. The alphabet sizes under examination included 2, 8, 16, 64, and 128 characters. Search operations were conducted on text files composed of randomly generated characters from each alphabet. The search experiments involved four different patterns of lengths 3, 10, 50, and 200 characters, randomly drawn from the corresponding alphabet characters. This resulted in the creation of 20 distinct test scenarios, combining four pattern lengths and five alphabets. Furthermore, all tests were replicated on newly generated random text files. Averages of these outcomes were used to mitigate the potential influence of unintentionally structured text sequences arising from random generation.

## 6 Data analysis and discussion

We refined our overall question to focus on the effectiveness of parallelization of string-matching algorithms using GPGPU accelerators into the following specific research questions:

*RQ1—Optimal length of search string:* Is there a differential effect of file size on string search operations?

*RQ2—Speedup ranking of parallel algorithms:* What is the observed range of speedup performance of parallel string matching algorithms across varying pattern lengths?

*RQ3—Performance ranking of parallel algorithms:* What are the specific trends in the performance of parallel string-matching algorithms across varying pattern lengths when applied to different datasets?

*RQ4—Impact of alphabet size:* How does varying the alphabet size affect the performance of parallel string-matching algorithms?

*RQ5—Impact of CUDA memory optimizations:* Is there any significant difference in using constant memory instead of global memory?

*RQ6—Impact of overlapping optimization:* What is the impact of the overlapping optimization technique?

The remainder of this section discusses the results of our study in terms of these research questions.

### 6.1 RQ1: optimal length of search string

To ascertain whether a preferred file size exists for conducting search operations, both serial and parallel algorithm versions are assessed using

unedited

the Wikipedia dataset. The runtime of each parallel algorithm is divided by its corresponding serial counterpart to calculate the speedup factor. Subsequently, the average values of these factors are computed for each data size and pattern length.

Fig. 2 illustrates the average speedup of parallel algorithms in relation to different pattern lengths across various text sizes. In the figure, the y-axis corresponds to the speedup factor of parallel algorithms compared to serial ones, while the x-axis represents the file size. Additionally, the parameter "m" signifies the pattern length, which is represented by lines on the graph, encompassing values of 3, 10, 50, and 200 characters. With the exception of the evident monotonic increase for very short pattern lengths (m=3), the overall speedup factors experience an ascending trend up to the 100 MB file size threshold. Therefore, unless explicitly specified, all further tests are conducted on 100 MB search files. This choice ensures a benchmark for evaluating the algorithms' optimal performance outcomes. Another crucial aspect to emphasize is that across all scenarios, the speedup factor of the algorithms consistently remained $\geq$1, signaling a persistent performance improvement when utilizing parallel string-matching algorithms compared to their serial counterparts.

### 6.2 RQ2: speedup ranking of parallel algorithms

After determining the optimal search string length, we can delve into a more detailed analysis to rank and improve the speedup performance of the considered parallel algorithms. This analysis was conducted on the Wikipedia dataset with a search file size of 100 MB for the pattern values of 3, 10, 50, and 200.

Table 2 presents the top five speedup performances of parallel algorithms for different pattern lengths (m). Speedup represents the improvement in processing time achieved by parallelizing the string-matching algorithms compared to a sequential version. For example, the *tbm* algorithm has been optimized to achieve a speedup of $51x$ when applied to pattern matching tasks with a length of three characters. This demonstrates that parallelization can be crucial for string matching. Another noteworthy observation is that the *gg* algorithm, in its parallel version, consistently maintains a position within the top five rankings across various character sizes. This suggests that the *gg* algorithm has a strong parallel performance and is well suited for a range of character sizes in the context of string matching.

### 6.3 RQ3: performance ranking of parallel algorithms

Following the determination of the optimal search string length, we proceeded to establish 100 MB as the size of the search files and subsequently ranked the performance of parallel algorithms. Utilizing the Wikipedia and Genome datasets, we used four distinct pattern lengths to conduct search operations on the prepared text file. This enabled us to acquire comparative data for assessment. We compared the running times of the parallel algorithms for each pattern to assess their respective performance. Based on our experimental runs, we present the top five algorithms based on the best and the worst running times within the parallel algorithm category.

Fig. 3 shows the performance of parallel algorithms with the Wikipedia and Genome datasets, presenting both the top and bottom five results for the pattern values of 3, 10, 50, and 200. For instance, the *fs* algorithm exhibits the best performance for the pattern value of 200 in the Wikipedia dataset, completing in 11.1482 ms. In contrast, the *ag* algorithm demonstrates the poorest performance, needing 95.2882 ms to complete the same process.

Our thorough analysis of parallel string-matching algorithms on both Wikipedia and Genome datasets revealed distinct trends in performance across various pattern lengths. It is evident that certain algorithms excel in specific contexts, highlighting the importance of selecting an algorithm tailored to the dataset characteristics and requirements. For instance, when considering parallel algorithms applied to the Genome dataset with a pattern size of 50, the execution times vary between 11.7540 ms and 129.8907 ms. The variability in execution times underscores how the choice of a parallel algorithm can significantly affect overall performance. In addition to evaluating algorithm performance based on pattern length, we also observed interesting consistencies and disparities across the datasets. One notable finding is the absence of a single algorithm that was consistently ranked in the top five for all different pattern lengths in the Wikipedia dataset. Conversely, the *ildm1* and *ildm2* algorithms consistently maintained their positions among the top

unedited

five performers for all pattern lengths in the Genome dataset. Shifting our focus to the bottom-performing algorithms, we uncovered distinct patterns in their underperformance across datasets. Across both the Wikipedia and Genome datasets, the *fdm* algorithm consistently appeared among the bottom performers, accompanied by *ksa* for the Wikipedia dataset and *ag* for the Genome dataset.

```
\begin{tikzpicture}
\begin{axis}[
        xlabel={File Size (MB)},
        ylabel={Speedup Factor},
        xmin=0.5, xmax=4.5,
        ymin=0, ymax=35,
        xtick={1,2,3,4},
        xticklabels={1,10,100,1000},
        ytick={0,5,10,15,20,25,30},
        ymajorgrids=true,
        grid style=dashed,
        cycle list name=exotic,
        legend style={
        legend pos=outer north east,
        font=\footnotesize,
        legend cell align={left},}
]

\addplot[color=black,mark=none,densely dashed]
        coordinates {
        (1,16.46)(2,25.18)(3,27.30)(4,30.18)
        };

\addplot[color=black,mark=none,densely dotted]
        coordinates {
        (1,12.98)(2,20.93)(3,23.31)(4,20.97)
        };

\addplot[color=black,mark=none,solid]
        coordinates {
        (1,11.68)(2,19.33)(3,21.74)(4,21.16)
        };

\addplot[color=black,mark=none,dash dot]
        coordinates {
        (1,8.77)(2,17.64)(3,19.19)(4,18.15)
        };

\legend{
        m = 3,
        m = 10,
        m = 50,
        m = 200,
}
```

**Fig. 2 The average speedup of parallel algorithms**

unedited

### 6.4 RQ4: impact of alphabet size

To examine algorithm behavior across various alphabet sizes and present the corresponding results, we used the synthetic randomized dataset, discerning variations across different pattern lengths and alphabet sizes. The alphabet sizes subjected to testing encompass 2, 8, 16, 64, and 128 characters, each acting as a distinct parameter in our experiments. We conducted search operations on text files, each comprising randomly generated characters drawn from its designated alphabet. To ensure a consistent scale, we maintained a fixed text file size of 100 MB, a choice made based on prior experimentation. Our search tests encompassed four distinct patterns, each characterized by different lengthsâĂŤ3, 10, 50, and 200 characters. These patterns were meticulously crafted through random selection from the corresponding alphabet. Consequently, we created a total of 20 unique test scenarios, stemming from the combination of four varying-length patterns and five distinct alphabets. To minimize variance originating from the testing environment, we executed each test sce-

| $m = 3$ | | $m = 10$ | | $m = 50$ | | $m = 200$ | |
|---|---|---|---|---|---|---|---|
| Name | Speedup | Name | Speedup | Name | Speedup | Name | Speedup |
| tbm | 51.08 | gg | 45.09 | gg | 43.43 | bf | 41.84 |
| kr | 45.37 | mp | 43.70 | col | 40.40 | gg | 31.56 |
| gg | 44.35 | col | 42.43 | bf | 40.36 | fdm | 29.57 |
| mp | 42.42 | kmp | 42.01 | mp | 37.49 | col | 29.24 |
| simon | 42.29 | simon | 40.33 | kmp | 37.47 | kmp | 26.41 |

**Table 2 Top five speedup ranking of parallel algorithms.**

unedited

| | $m = 3$ | | $m = 10$ | | $m = 50$ | | $m = 200$ | |
|---|---|---|---|---|---|---|---|---|
| | Name | Time (ms) | Name | Time (ms) | Name | Time (ms) | Name | Time (ms) |
| W. Top 5 | bndmq4 | 10.9312 | sfbom | 12.3764 | rcol | 11.3989 | fs | 11.1482 |
| | kr | 12.8345 | raita | 12.5348 | bxs | 11.4467 | lbndm | 11.1676 |
| | raita | 14.4963 | fs | 12.9553 | tndm | 11.4561 | sbndm | 11.1868 |
| | bf | 14.7148 | bww | 13.0278 | lbndm | 11.5302 | fsbndm | 11.1935 |
| | br | 14.7840 | ildm2 | 13.0482 | raita | 11.5432 | sbndm_bmh | 11.1976 |
| G. Top 5 | ildm2 | 13.3445 | ildm2 | 12.4135 | sbndm2 | 11.7540 | bmh_sbndm | 11.3339 |
| | ildm1 | 13.3451 | ildm1 | 12.4138 | bmh_sbndm | 11.7779 | sbndm2 | 11.6962 |
| | bww | 13.4991 | bww | 12.4280 | bndm | 11.8721 | ildm1 | 11.8898 |
| | bndm | 13.8913 | lbndm | 12.5027 | ildm1 | 11.8810 | ildm2 | 11.8968 |
| | rcol | 14.0671 | sbndm2 | 12.5142 | ildm2 | 11.8815 | trf | 11.9659 |
| W. Bottom 5 | bfs | 83.1401 | ksa | 47.7346 | ts | 55.8337 | ag | 95.2882 |
| | bsdm | 47.7171 | fdm | 46.8584 | ag | 50.6226 | kr | 67.4551 |
| | fdm | 45.7309 | bfs | 45.7860 | ksa | 50.3636 | fdm | 54.1321 |
| | ksa | 41.8872 | ts | 41.5247 | fdm | 49.3713 | tw | 52.8663 |
| | ebom | 30.7356 | dfa | 31.3969 | so | 33.9019 | ksa | 51.2419 |
| G. Bottom 5 | fdm | 50.4712 | ag | 63.3685 | ag | 129.8907 | ag | 244.8445 |
| | ag | 47.2614 | fdm | 62.4360 | fdm | 54.0337 | kr | 64.4945 |
| | ts | 40.5084 | ts | 35.5995 | ts | 40.7435 | fdm | 51.5552 |
| | smoa | 31.1867 | smoa | 31.8406 | kr | 30.9725 | tw | 49.1941 |
| | ksa | 27.9368 | ksa | 30.1738 | ksa | 30.9442 | smoa | 47.6212 |

**Fig. 3 The performance rankings of parallel algorithms.**

nario 10 times, subsequently aggregating the results to report the average performance. Furthermore, to mitigate any inadvertent structure within the text sequences, we replicated all tests on randomly generated text files, allowing us to derive averages of the results. This comprehensive approach serves to enhance the robustness and reliability of our findings.

Fig. 4 illustrates the average speedup of paral-

```
\begin{tikzpicture}
\begin{axis}[
        xlabel={Alphabet Size},
        ylabel={Speedup Factor},
        xmin=0.5, xmax=5.5,
        ymin=0, ymax=45,
        xtick={1,2,3,4,5},
        xticklabels={2,8,16,64,128},
        ytick={0,5,10,15,20,25,30,35,40},
        ymajorgrids=true,
        grid style=dashed,
        cycle list name=exotic,
        legend style={
        legend pos=outer north east,
        font=\footnotesize,
        legend cell align={left},}
]

\addplot[color=black,mark=none,densely dashed]
        coordinates {
        (1,38.98)(2,24.45)(3,22.42)(4,20.80)(5,20.66)
        };

\addplot[color=black,mark=none,densely dotted]
        coordinates {
        (1,32.25)(2,23.34)(3,21.78)(4,19.51)(5,19.00)
        };
\addplot[color=black,mark=none,solid]
        coordinates {
        (1,28.08)(2,20.90)(3,19.14)(4,17.77)(5,17.71)
        };
 \addplot[color=black,mark=none,dash dot]
        coordinates {
        (1,20.86)(2,17.53)(3,17.18)(4,16.70)(5,16.99)
        };

\legend{
        m = 3,
        m = 10,
        m = 50,
        m = 200,
}
```

unedited

**Fig. 4 The impact of alphabet size on parallel algorithms.**

lel algorithms in relation to different pattern lengths across various alphabet sizes. In the figure, the y-axis corresponds to the speedup factor of parallel algorithms compared to serial ones, while the x-axis represents the alphabet size. The parameter "m" is represented by lines on the graph, with each line corresponding to a specific "m" value that is listed in the legend. Overall, the results suggest that, for the particular parallel algorithms, smaller alphabet sizes tend to yield better speedup factors for a range of pattern lengths, especially for the alphabet size of "2". As the alphabet size increases, the speedup factor decreases, indicating that the algorithm's parallelism might be more effective when dealing with smaller sets of characters. Additionally, it is important to highlight that the speedup factor continues to come closer as the alphabet size increases for all the test cases, independent of the pattern size. For alphabet sizes of 64 and 128, the speedup factor almost remains constant, suggesting that the algorithm's performance improvement plateaus beyond a certain alphabet size, and further increases in alphabet size may not significantly affect its speedup.

### 6.5 RQ5: impact of memory setup

We conducted tests using constant memory on a 100MB Wikipedia dataset with four different pattern lengths: 3, 10, 50, and 200 characters. Despite the high frequency of read-only data source access, the limited size of constant memory poses challenges for the string-matching problem. For instance, the text string used in our tests consumes a significant amount of memory, exceeding the capacity reserved for constant memory. Consequently, we used constant memory to store our pattern strings. In these tests, we moved the patterns to constant memory from global memory, and we modified kernel transactions to utilize this memory space.

| Pattern Length (m) | Speedup Factor |
|---|---|
| 3 | 1.007 |
| 10 | 1.012 |
| 50 | 1.004 |
| 200 | 0.997 |

**Table 3 Comparison of average speedup factors with constant memory versus global memory**

Table 3 shows the average speedup factors that with constant memory compared to global memory for our parallel algorithms for each pattern. For a pattern length of "3", the speedup factor is 1.007, which suggests that using constant memory is slightly faster (approximately 0.7% faster) than using global memory. Conversely, for a pattern length of 200, the speedup factor is 0.997, indicating that using constant memory is slightly slower (approximately 0.3% slower) than using global memory. Overall, the results indicate that there is no significant performance difference when using constant memory.

### 6.6 RQ6: impact of overlapping optimization

One of the optimization techniques we explored is known as "overlapping". It enables the data transfer units within the graphics processor to operate independently of the kernel units, preventing them from blocking each other. This approach aims to minimize downtime for the processing cores. To implement this technique, data destined for the graphics card are transferred in small batches. After each piece of data is transferred, the corresponding kernel responsible for processing that portion is invoked. Unlike the traditional sequential approach, where kernel processors remain idle during data transfers, this technique allows for a more efficient use of hardware resources by breaking the data into smaller pieces and interleaving kernel operations between transfers. The kernel processing can commence as soon as the first piece of data transfer is completed, thereby enabling a significant overlap between the kernel processing and data transfer operations, effectively concealing much of the kernel processing time under the data transfer process.

In the context of the overlapping tests, we used CUDA stream queues as a critical component of our evaluation process both with and without pinned memory. While memory is locked in place, preventing it from being moved or swapped out by the operating system's memory management system in pinned memory, data transfers between the CPU and the GPU might involve copying data between different memory locations without pinned memory. Our objective was to comprehensively benchmark the effectiveness of the overlapping strategy in a variety of scenarios. To achieve this, we examined a total of 20 test cases, each designed to explore different aspects of the overlapping optimization technique,

unedited

involving patterns with lengths of 3, 10, 50, and 200 characters searched within text strings constructed from varying alphabet sizes of 2, 8, 16, 64, and 128 characters.

The results from our overlapping tests are represented in Fig. 5. Each bar chart represents the speedup factor achieved for specific pattern lengths across various alphabet sizes denoted by A2, A8,

```
\begin{tikzpicture}
	\begin{axis}[
	ybar,
	bar width=0.9em,
	% enlargelimits=0.15,
	enlarge y limits={value=0.50,upper},
	enlarge x limits=0.15,
	ymin=1,
	legend style={at={(0.5,1.35)},
	anchor=north,legend columns=-1,draw=none},
	ylabel={Speedup Factor},
	xlabel={Without Pinned Memory},
	symbolic x coords={A2,A8,A16,A64,A128},
	ytick=\empty,
	xtick=data,
	xtick style={draw=none},
	nodes near coords,
	nodes near coords align={vertical},
	nodes near coords=\rotatebox{90}{\scriptsize\pgfmathprintnumber\pgfplots-
pointmeta},
	every node near coord/.append style={
	/pgf/number format/fixed zerofill,
	/pgf/number format/precision=2
	},
	legend cell align={left},
	width=35em,
	height=12em,
	]
	\addplot[fill=black!80] coordinates { (A2, 1.57) (A8, 1.41) (A16, 1.39) (A64, 1.34)
(A128, 1.33) };
	\addplot[fill=black!60] coordinates { (A2, 1.90) (A8, 1.46) (A16, 1.39) (A64, 1.33)
(A128, 1.35) };
	\addplot[fill=black!40] coordinates { (A2, 1.80) (A8, 1.49) (A16, 1.35) (A64, 1.37)
(A128, 1.33) };
	\addplot[fill=black!20] coordinates { (A2, 1.86) (A8, 1.38) (A16, 1.42) (A64, 1.34)
(A128, 1.32) };
	\legend{$m=3$,$m=10$,$m=50$,$m=200$}
	\end{axis}
\end{tikzpicture}

\begin{tikzpicture}
	\begin{axis}[
	ybar,
	bar width=0.9em,
	% enlargelimits=0.15,
```

**Fig. 5 Without and with pinned memory usage in overlapping optimization.**

unedited

A16, A64, and A128. The y-axis represents the speedup factor, which indicates how much faster the algorithms run compared to a baseline. For example, an average speedup factor of 1.46 has been achieved without pinned memory based on the results. This indicates that algorithms utilizing overlapping are generally 46% faster. Notably, the speedup is significantly more pronounced for binary alphabet results ($\sigma = 2$), reaching a peak of 90% and averaging at 78%. We further carried out tests that combined the overlapping strategy with the utilization of pinned memory, as depicted in Fig. 5. In comparison to using pinned memory alone without overlapping, combining both strategies resulted in additional performance gains. The average speedup for tests with overlapping and pinned memory reached 62%, a notable increase from the 46% observed in scenarios without pinned memory. Additionally, there was an upward trend in speedups for combined test cases, while the performance gains from overlapping alone showed a decrease. Overall, overlapping optimization with pinned memory stands out as an effective strategy for enhancing algorithm performance.

## 7 Conclusions

String matching is a long-standing and continually researched challenge within the realm of information technology, with its utility extending across a wide array of domains. In this paper, we have presented the CUSMART library, which is a parallel string-matching library developed using Nvidia's CUDA toolkit. To generate a codebase of parallel string-matching algorithms, we leveraged the SMART library, which provides a serial implementations of string-matching algorithms. Overall, we have developed 64 parallel string-matching algorithms implemented using CUDA, each accompanied by its corresponding CPU version.

With the aid of our CUSMART library, we conducted evaluations of the parallel string-matching algorithms in various test scenarios to observe their performance across different usage scenarios. These test scenarios have been created using varying text sizes, pattern sizes, and alphabets. We have also investigated the impact of memory setup and overlapping optimization on the performance of parallel string-matching algorithms.

The results of our experiments demonstrate the following:

- The enhanced performance achieved in string-matching algorithms through parallelization has significantly improves processing efficiency.
- The average speedup of parallel algorithms varies in relation to different pattern lengths across various text sizes. The optimal text size for our experiments has been determined as 100 MB.
- In the experiments performed herein, we have observed speed-ups of up to $51x$ for the *tbm* algorithm on the Wikipedia dataset under pattern matching tasks with a length of "3" characters.
- While the performance of parallel algorithms for string matching highly varies based on the text file and pattern size, the *fdm* algorithm consistently appeared among the five lowest performers.
- For alphabet sizes of 64 and 128, the speedup factor remains nearly constant regardless of pattern size. This suggests that the algorithm's performance improvement plateaus beyond a certain alphabet size, and further increases in alphabet size may not significantly affect its speedup.
- When constant memory is utilized instead of global memory, there is no significant difference in performance.
- Utilizing overlapping optimization with pinned memory proves to be an effective strategy for enhancing the performance of parallel string-matching algorithms.

Our findings suggest that GPU devices are excellent candidates for handling string-matching intensive workloads. They can serve as the primary computation units for this task or act as supportive devices to offload certain operations. In composite operations involving various types of tasks, the string-matching workload can be efficiently delegated to GPU devices, thus freeing up the CPU for other concurrent tasks.

One limitation in our study is that not all of the algorithms in the SMART library can be parallelized. This is due to structural limitations or sequential

unedited

dependencies, which make some algorithms not well suited for parallel processing. In addition, optimizations for GPU parallelization, such as shared memory, cannot be applied for all algorithms.

On the basis of these conclusions, there are several potential areas for future work. First, we plan to enhance the CUSMART library by incorporating new features, introducing additional parallel string matching algorithms, and exploring performance improvement options. Second, we intend to compare GPU accelerators between workstation and mobile platforms in terms of energy efficiency and cost-effectiveness.

## Contributors

Mengu NAZLI and Onur CANKUR drafted the paper and collected the test results. Adnan OZSOY and Cagri SAHIN helped organize the paper, made revisions and finalized it.

## Conflict of interest

All authors declare that they have no conflict of interest.

## References

Adams MD, Celniker SE, Holt R, et al., 2000. The genome sequence of drosophila melanogaster. *Science*, 287(5461):2185-2195. https://doi.org/10.1126/science.287.5461.2185

Adey, Snehal P., 2013. GPU Accelerated Pattern Matching Algorithm for DNA Sequences to Detect Cancer using CUDA. MS Thesis, College of Engineering, Pune, India.

Ashkiani S, Amenta N, Owens JD, 2016. Parallel approaches to the string matching problem on the gpu. Proc 28th ACM Symposium on Parallelism in Algorithms and Architectures, p.275âĂŞ285. https://doi.org/10.1145/2935764.2935800

Baeza-Yates R, Gonnet GH, 1992. A new approach to text searching. *Commun ACM*, 35(10):74âĂŞ82. https://doi.org/10.1145/135239.135243

Bellekens X, Atkinson RC, Renfrew C, et al., 2013. Investigation of gpu-based pattern matching. Proc 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting, p. 5.

Blelloch GE, 1990. Vector models for data-parallel computing. MIT Press, Cambridge, USA.

Ceruzzi PE, 2003. A History of Modern Computing. MIT Press, Cambridge, USA.

Cormen TH, Leiserson CE, Rivest RL, et al., 2009. Introduction to Algorithms. 3rd ed. MIT Press, Cambridge, USA.

Faro S, Lecroq T, 2010. The exact string matching problem: a comprehensive experimental evaluation. -, -. http://arxiv.org/abs/1012.2547

Faro S, Lecroq T, 2013. The exact online string matching problem. *ACM Comput Surv*, 45(2):1-42. https://doi.org/10.1145/2431211.2431212

Faro S, Lecroq T, Borzì S, et al., 2016. The string matching algorithms research tool. Proc Prague Stringology Conf, p.99-111.

Feng Wc, Cameron K, 2007. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50-55.

Girkar M, Polychronopoulos CD, 1995. Extracting task-level parallelism. *ACM Trans Program Lang and Syst*, 17(4):600-634. https://doi.org/https://doi.org/10.1145/210184.210189

Hains D, Cashero Z, Ottenberg M, et al., 2011. Improving CUDASW++, a parallelization of smith-waterman for cuda enabled devices. IEEE Int Symposium on Parallel and Distributed Processing Workshops and Phd Forum, p.490-501. https://doi.org/10.1109/IPDPS.2011.182

Han TS, Ko SK, Kang J, 2007. Efficient subsequence matching using the longest common subsequence with a dual match index. 5th Int Conf Machine Learning and Data Mining in Pattern Recognition, p.585-600. https://doi.org/10.1007/978-3-540-73499-444

Harris, Mark, 2012. How to overlap data transfers in CUDA C/C++ - https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.

He L, Fang B, Sui J, 2005. The wide window string matching algorithm. *Theor Comput Sci*, 332(1-3):391-404. https://doi.org/https://doi.org/10.1016/j.tcs.2004.12.002

Horspool RN, 1980. Practical fast searching in strings. *Softw Pract Exp*, 10(6):501-506. https://doi.org/https://doi.org/10.1002/spe.4380100608

Kadhim HA, AbdulRashidx N, 2014. Maximum-shift string matching algorithms. Int Conf on Computer and Information Sciences, p.1-6. https://doi.org/10.1109/ICCOINS.2014.6868423

Kouzinopoulos CS, Michailidis PD, Margaritis KG, 2015. Multiple string matching on a gpu using CUDAs. *Scalable Comput*, 16(2):121-137. https://doi.org/10.12694/scpe.v16i2.1085

Lee CL, Lin YS, Chen YC, 2015. A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection. *PLoS ONE*, 10(10). https://doi.org/10.1371/journal.pone.0139301

Ligowski L, Rudnicki W, 2009. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. IEEE Int Symposium on Parallel & Distributed Processing, p.1-8. https://doi.org/10.1109/IPDPS.2009.5160931

Lin CY, Och FJ, 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. Proc 42nd Annual Meeting on Association for Computational Linguistics, p.605-612. https://doi.org/10.3115/1218955.1219032

Lin DL, Huang TW, 2021. Efficient GPU computation using task graph parallelism. 27th Int Conf on Parallel and Distributed Computing on Euro-Par: Parallel Processing, p.435-450.

unedited

Lu SY, Fu KS, 1978. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Trans on Syst Man Cybern*, 8(5):381-389.
https://doi.org/10.1109/TSMC.1978.4309979

Maxime C, Rytter W, 1995. Text algorithms. *Choice Rev Online*, 32(10):32-5696-32-5696.
https://doi.org/10.5860/choice.32-5696

Mitani Y, Ino F, Hagihara K, 2017. Parallelizing exact and approximate string matching via inclusive scan on a GPU. *IEEE Trans on Parallel Distrib Syst*, 28(7):1989-2002.
https://doi.org/10.1109/TPDS.2016.2645222

Nagaveni V, Raju GT, 2014. Various string matching algorithms for dna sequences to detect breast cancer using CUDA processors. *Int J of Eng Technol*, 14(3):42-47.

Navarro G, Raffinot M, 1998. A bit-parallel approach to suffix automata: Fast extended string matching. 9th Annual Symposium on Combinatorial Pattern Matching, p.14-33.

Nvidia, 2020. 136 GPU-Accelerated Supercomputers Feature in TOP500 | NVIDIA Blog — https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/.

Peng J, Chen H, 2010. Cugrep: A GPU-based high performance multi-string matching system. Proc 2nd Int Conf on Future Computer and Communication, p.77-81.
https://doi.org/10.1109/ICFCC.2010.5497832

Petrakis EGM, 1993. Image representation , indexing and retrieval based on spatial relationships and properties of objects. PhD Dissemination, University of Crete, Greece.

Pungila C, Negru V, 2012. A highly-efficient memory-compression approach for GPU-accelerated virus signature matching. 15th Int Conf on Information Security, p.354-369.
https://doi.org/10.1007/978-3-642-33383-5-22

Quinn MJ, 2004. Parallel Programming in C with MPI and OpenMP. McGraw-Hill Higher Education, Boston, USA.

Ramos-Frías R, Vargas-Lombardo M, 2017. A review of string matching algorithms and recent implementations using GPU. *Int J Secur Appl*, 11(6):69-76.
https://doi.org/10.14257/ijsia.2017.11.6.06

Rasool A, Khare N, 2012. Parallelization of kmp string matching algorithm on different SIMD architectures: multi-core and GPGPUs. *Int J Comput Appl*, 49(11):26-28.
https://doi.org/10.5120/7672-0963

Sellis TK, 1998. Multiple-query optimization. *ACM Trans Database Syst*, 13(1):23-52.
https://doi.org/10.1145/42201.42203

Sharma J, Singh M, 2015. Cuda based Rabin-Karp pattern matching for deep packet inspection on a multicore GPU. *Int J Comput Network Inf Secur*, 7(10):70-77.
https://doi.org/10.5815/ijcnis.2015.10.08

Subhlok J, Stichnoth JM, O'hallaron DR, et al., 1993. Exploiting task and data parallelism on a multicomputer. Proc 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, p.13-22.
https://doi.org/https://doi.org/10.1145/155332.155334

Tian X, Song Y, Wang X, et al., 2012. Shortest path based potential common friend recommendation in social networks. 2nd Int Conf on Cloud and Green Computing, p.541-548.

Tran NP, Lee M, 2013. High performance string matching for security applications. Int Conf on ICT for Smart Society, p.1-5.
https://doi.org/10.1109/ICTSS.2013.6588052

Tran NP, Lee M, Hong S, et al., 2012. Memory efficient parallelization for Aho-Corasick algorithm on a GPU. IEEE 14th Int Conf on High Performance Computing and Communication & IEEE 9th Int Conf on Embedded Software and Systems, p.432-438.
https://doi.org/10.1109/HPCC.2012.65

Weiner P, 1973. Linear pattern matching algorithms. 14th Annual Symposium on Switching and Automata Theory, p.1-11.
https://doi.org/10.1109/SWAT.1973.13

Xu K, Cui W, Hu Y, et al., 2013. Bit-parallel multiple approximate string matching based on GPU. *Procedia Comput Sci*, 17:523-529.
https://doi.org/10.1016/j.procs.2013.05.067

Yao ACC, 1979. The complexity of pattern matching for a random string. *SIAM J Comput*, 8(3):368-387.
https://doi.org/10.1137/0208029

Yong KK, Karuppiah EK, 2013. Hash match on GPU. IEEE Conf on Open Systems, p.150-155.
https://doi.org/10.1109/ICOS.2013.6735065

Zha X, Sahni S, 2013. GPU-to-GPU and host-to-host multi-pattern string matching on a GPU. *IEEE Trans Comput*, 62(6):1156-1169.
https://doi.org/10.1109/TC.2012.61

Zhou J, An H, Li X, et al., 2011. Implementation of string match algorithm BMH on GPU using CUDA. *Energy Procedia*, 13:1853-1861.
https://doi.org/10.1016/j.egypro.2011.11.261

Ziv J, Lempel A, 1977. A universal algorithm for sequential data compression. *IEEE Trans Inf Theory*, 23(3):337-343.
https://doi.org/10.1109/TIT.1977.1055714

unedited