



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

Adnan OZSOY^{†1}, Mengu NAZLI¹, Onur CANKUR², Cagri SAHIN³

¹Department of Computer Engineering, Hacettepe University, Ankara 06800, Türkiye

²Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA

³Department of Computer Engineering, Gazi University, Ankara 06570, Türkiye

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 Apr. 14, 2025

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 uses general-purpose computing on graphics processing unit (GPGPU) programming concepts to enhance performance and gain insight into 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. 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 highlight the potential of GPGPU computing in string matching applications through the scalability of algorithms, suggesting significant performance improvements. Furthermore, we have identified the best and worst performing algorithms in various scenarios.

Key words: String matching; Parallel programming; Graphics processing unit (GPU) programming; General-purpose computing on GPU (GPGPU); NVIDIA; Compute unified device architecture (CUDA); String matching algorithms research tool (SMART)

<https://doi.org/10.1631/FITEE.2400091>

CLC number: TP391.4

1 Introduction

String matching holds significant importance within the realm of text processing. It serves as an essential component used by operating systems' functional software. Despite data being shared and stored in numerous formats, text remains the predominant form of information handling. This is particularly evident in the literature, where data

consists 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, where biological molecules are approximated as sequences of more fundamental building blocks such as amino acids or nucleotides, 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 (Crochemore and Rytter, 1994; Han et al., 2007), social media content processing (Tian et al., 2012), data mining (Sellis, 1998), data compression (Ziv and Lempel, 1977), coding theory (Petrakis, 1993; Lin CY and Och, 2004), and

[†] Corresponding author

* Project supported by the Scientific and Technological Research Council of Türkiye (No. 117E142). Open access funding provided by the Scientific and Technological Research Council of Türkiye (TÜBİTAK)

ORCID: Adnan OZSOY, <https://orcid.org/0000-0002-0302-3721>

© The Author(s) 2025

numerous other fields.

In computer-centric domains, the 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 computing 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 time in string matching applications but also intensifies the performance demands on string matching algorithms. As a result, the reduction of calculation time 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 time in high-performance computing, with the central processing unit (CPU) 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 GPU (GPGPU) has gained prominence as an alternative that offers significant computational power. Therefore, parallel computing with GPGPU can be used for string matching.

In this paper, we present a compute unified device architecture (CUDA)-enhanced string matching algorithms research tool (CUSMART), applied for a study on parallelization in general-purpose computing on 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 SMART library, which offers serial implementations of string matching algorithms (Faro et al., 2016). This tool is 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 graphics processing unit (GPU) architecture via NVIDIA's CUDA application programming interface (API). The findings of the study illustrate a remarkable average speedup of up to $51\times$ compared to the serial CPU versions of these algorithms, highlighting the significant potential of GPGPU computing in the context of string matching applications. Moreover, these speedup enhancements can be 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:

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

2 Background

2.1 String matching

Since 1970, over 80 string matching algorithms have been proposed (Faro et al., 2016). These algorithms can be categorized into three distinct groups based on the primary strategies that they apply, including comparison-based algorithms, automata-based algorithms, and bit-parallel algorithms.

The earliest examples of the string matching algorithms proposed in the field use strategies based on character comparison, such as brute force (Cormen et al., 2009), Morris–Pratt (Weiner, 1973), maximum shift (Kadhim and AbdulRashid, 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 used 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 $\mathcal{O}(n)$, where n is text length. 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 $\mathcal{O}(n \cdot \log_{\sigma} m/m)$ (Yao, 1979), where n represents the length of the text string, m represents the length of the pattern string, and σ 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 using this technique for string matching is through the application of the deterministic finite automaton (DFA) concept, which achieves linear time complexity in the search process. DFA processes each character of the input text exactly once, transitioning between states according to predefined rules and maintaining a constant processing time per character due to its fixed number of states and transitions, thus yielding an overall linear time complexity (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 $\mathcal{O}(n \cdot \log_{\sigma} m/m)$ time complexity for the average case. Both of these algorithms exploit the benefits of using 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. Such 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 a time 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 the field of parallel programming. When tackling problems using computer programs, it is common to break down the task into discrete calculations, each focusing 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 DL 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 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, 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 there are 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 parallelizing operations on the data being processed. This approach aims to leverage the strengths of 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. GPU cores have been used in a wider range of applications, leading to the term GPGPU. A recent study indicates that 40% of the overall computing power in the TOP 500 supercomputer list is derived from GPU-accelerated systems (NVIDIA, 2019). A similar ranking, known as 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 effi-

ciency achieved by heterogeneous systems through the utilization of GPU devices.

CUDA is a versatile parallel-computing platform and programming model designed to use 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 works

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

Ligowski and Rudnicki (2009) introduced an enhanced parallel version of the Smith–Waterman algorithm. The performance of this modified algorithm exhibited a $3.5\times$ speedup compared to previous GPU implementations, achieving 70% of the maximum theoretical hardware performance on their experimental setup. Addressing a similar objective, Hains et al. (2011) introduced a high-performance parallel implementation of Smith–Waterman (CUDA-SW++), which was implemented using the CUDA platform. Identifying a bottleneck within the intra-task 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 Nrgrep, a well-known command-line utility used in the UNIX operating system. Their implementation, known as CUGrep, used a GPU-based multistring matching algorithm rooted in the BNDM algorithm, resulting in reported performance gains of $40\times$ compared to Nrgrep on an NVIDIA Tesla C1060 GPU and an Intel Xeon 5110 CPU.

Jaiswal (2014) proposed a parallel implementation of the enhanced Boyer–Moore (EBM) string matching algorithm using GPUs to improve performance in applications like network security. By leveraging OpenCL for GPU programming, the study showed that the GPU-based EBM algorithm achieved a speedup of up to $13\times$ compared to its serial CPU counterpart and outperformed a multithreaded CPU version by nearly a speedup of $9\times$. Yong and Karuppiah (2013) investigated a hash search algorithm executed on a GPU and conducted a performance comparison with parallel versions of the brute force algorithm and the Boyer–Moore–Horspool algorithm. They used two distinct NVIDIA GPU devices: one based on the Fermi architecture, specifically C2075, and the other based on the Kepler architecture, identified as K20c. While their innovative hash search algorithm exhibited lower 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, contributing to the observed performance difference. Additionally, the authors documented a noteworthy speedup of $23\times$ 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 $150\times$ when compared to the global memory-using version of the Boyer–Moore–Horspool algorithm. This outcome underscores the potential of tailored algorithmic implementations and GPU-specific optimizations in significantly elevating 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 with an NVIDIA 9500GT GPU and an Intel Core2Duo 2.2 GHz CPU, their experimental outcomes demonstrated a maximum speedup of $15.72\times$ 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 leveraged intra-task 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 merged 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 $1/22$ of the memory of 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 Gb/s after testing with up to 20 000 patterns. This throughput was 1.45 times higher than that of 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 2.8 GHz quad-core CPU, they observed a noteworthy speedup ranging from $3.1\times$ to $9.5\times$ 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 bit-parallelism variant optimized for multiple pattern matching on a GPU. Using 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 with 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 used the OpenCL framework to facilitate the GPU and multi-core CPU implementations of this algorithm. Notably, their experimental setup featured an 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 $29\times$, 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, using 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 was conducted using an Intel Core i7 3770 CPU and an NVIDIA GeForce GTX680 GPU. The hybrid Aho–Corasick algorithm outperformed the CPU version by $3.4\times$ and the GPU version by $2.7\times$, 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 deoxyribonucleic acid (DNA) sequences, Adey (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 in both serial CPU and GPU versions, such as the multiple shift minimum pattern matching algorithm, inverted Knuth–Morris–Pratt multiple pattern matching algorithm, enhanced prefix matching for multiple pattern searching and preprocessing algorithm, and improved Aho–Corasick for efficient multiple pattern matching algorithm. 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 speedup 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 using 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, it was reported that the efficiency of the GPU implementation was 30 times that of a serial implementation conducted 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 substantial speed gains, with improvements ranging from a factor of 2.5 to 10.9, when using 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 speedup compared to the CPU implementation. These findings underscore the potential of GPU-based optimizations in significantly enhancing 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 method, the divide-and-conquer method, 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 parallelism 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 processing—a feature not feasible for most other string matching algorithms.

Mitani et al. (2017) introduced a tribrid parallel approach tailored for bit-parallel algorithms such as 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 the review 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 centered on specific algorithms, frequently selected from a limited range of well-known string matching techniques.

These commonly studied algorithms include Boyer–Moore, Boyer–Moore–Horspool, Knuth–Morris–Pratt, backward 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 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 to 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 in 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, and alphabet size) and several optimizations—this aspect is also missing in the literature.

4 CUSMART library

This paper introduces CUSMART, a parallel string matching library built on NVIDIA’s CUDA architecture (<https://github.com/adnanozsoy/CUSMART>). It is an extension of the SMART library, originally designed for single-core CPUs and featuring serial algorithm versions aimed at single-thread execution. However, it is not optimized to use multi-core parallel processing on the CPU. Its setup 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 necessitates extensive

alterations to the codebase due to the divergent design priorities of the original library. In the SMART library, each algorithm's executable file is initiated with memory allocation, followed by the reading and processing of text data. Finally, the allocated memory is freed when the program's lifecycle 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 to be 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 demands—around 33 000 times 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 using only the required space and releasing it after operation.

Table 1 shows the 64 algorithms included 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 code "fs."

4.1 Optimization and implementation considerations

In the code integrated from the SMART library, we introduce modifications to the algorithm function outputs. The original code uses 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 becomes 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 the memory 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 obtained 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 time 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 are often 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 single task across extensive data, favoring an intra-task methodology when it is designed according to the CUDA architecture. Therefore, our parallel implementation methods align with intra-task parallelism principles.

The algorithms within our library are designed for parallel string matching across smaller text segments. Threads are assigned 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 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 assesses the impact of various stride lengths on algorithm performance.

Efficiently leveraging all processors is crucial in heterogeneous computing. To enhance throughput, the CUDA schedules commands for execution within

Table 1 CUSMART algorithms and their corresponding codes

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	pbmh	Boyer–Moore–Horspool using probabilities
bm	Boyer–Moore	qs	Quick search
bmh_sbndm	Boyer–Moore–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	Bit-parallel 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	tvsubs	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

DAWG: directed acyclic word graph; BNDM: backward nondeterministic DAWG matching

“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 selected algorithms, and our example implementations facilitate

straightforward development and testing of streaming versions for other string matching algorithms. We conduct diverse tests on streaming with varying configurations. Fig. 1 illustrates the resultant behavior.

Given the typically large size of the text string, we opt 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 is designed to facilitate swift memory transactions among 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 operation, and many unnecessary memory transactions need to be done. Thus, combining this reasoning with our initial test results, it is realized that moving the search text to shared memory would cause more harm than good, and hence, shared memory is not used 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 the kernel. The caching mechanism speeds up the fetching of frequently accessed data, but the cache storage is not guaranteed through the lifetime of the kernel. Shared memory offers controlled access to data compared to cache with similar low-latency performance.

Additionally, the CUSMART library offers the capability to use pinned memory via a command-line option. Activating the pinned memory option allocates a portion of the host's 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 the other half is matched by another thread. As a solution, we extend the search window for each thread to be $m - 1$ more character checks at the end of the window, where m 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 used in our tests incorporate these calculations to attain the 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 memory. Due to longer GPU execution times for the same preprocessing operations 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 transfers from the host

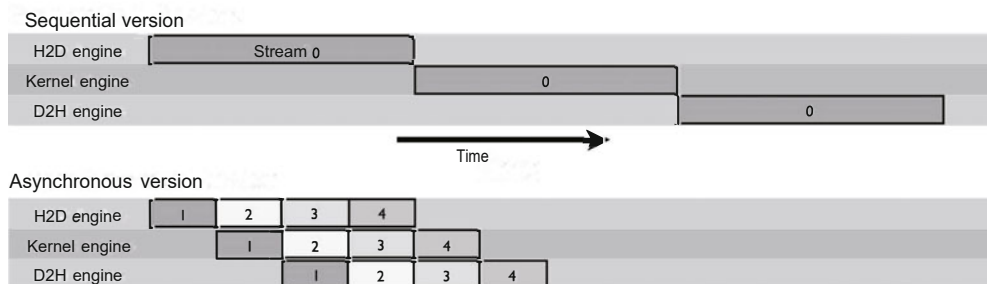


Fig. 1 Overlapping memory transfer and kernel processing (courtesy of Harris (2012)). H2D: host to device; D2H: device to host

(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 single 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 uses an array allocated in the GPU's global memory. This array records match positions, avoiding 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 GPU side to reduce the 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 wrapper function and the kernel function. The wrapper function encapsulates preparatory steps such as memory allocation, memory 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, compar-

ison, 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 the initial runtime, 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 conduct each test at least 10 times.

5 Empirical study

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

Various datasets were used for testing, 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 was executed at least 10 times. Average timings were reported when deviations between runs were negligible, ensuring reliable performance assessment.

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

5.1 Test platform

The computer used for testing was an HP Z8 G4 workstation equipped with two Intel Xeon Silver 4114 CPUs running at 2.20 GHz with a 9.6 GT/s bus speed, 13.75 MB cache, and 10 cores each. The system boasted 128 GB of DDR4 2666 MHz error-correcting code (ECC) RAM, configured with 4×32 GB modules. Storage was facilitated by a 256 GB SATA solid-state drive (SSD). The operating system was Ubuntu 18.04. The system used an NVIDIA GeForce GTX 1080 Ti graphics card with 11 GB of memory operating at 484 GB/s bandwidth. The graphics card featured 3584 CUDA cores running at 1.5 GHz and was used for GPGPU operations. The CUDA runtime environment was compatible with Ubuntu 18.04, and the CUDA toolkit version 8.0 was installed to support GPU-accelerated computations and programming.

5.2 Test datasets

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

1. Wikipedia: This dataset is composed of daily text in English, sourced from a Wikipedia content dump file. Sample text sizes were chosen as 1 MB, 10 MB, 100 MB, and 1 GB 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.

2. Genome: DNA tests were integrated to an-

alyze 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 selected 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.

3. 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 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:

1. Optimal length of the search string: Is there a differential effect of file size on string search operations?

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

3. 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?

4. Impact of alphabet size: How does varying the alphabet size affect the performance of parallel string matching algorithms?

5. Impact of CUDA memory optimizations: Is there any significant difference in using constant

memory instead of global memory?

6. 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 Optimal length of the search string

To ascertain whether a preferred file size exists for conducting search operations, both serial and parallel algorithm versions were assessed using the Wikipedia dataset. The runtime of each parallel algorithm was divided by its corresponding serial counterpart to calculate the speedup factor. Subsequently, the average values of these factors were computed for each data size and pattern length.

Fig. 2 illustrates the average speedup of parallel algorithms about different pattern lengths across various text sizes. The y -axis corresponds to the speedup factor of parallel algorithms compared to serial ones, while the x -axis represents the file size. Additionally, m signifies the pattern length, encompassing values of 3, 10, 50, and 200 characters. With the exception of the evident monotonic increase for $m = 3$, the overall speedup factors experienced an ascending trend up to the 100 MB file size threshold. Therefore, unless explicitly specified, all further tests were conducted on 100 MB search files. This choice ensured a benchmark for evaluating the algorithms' optimal performance outcomes. Another crucial aspect to emphasize is that across all scenarios, the speedup factors of the algorithms consistently remained ≥ 1 , signaling a persistent performance improvement when using parallel string matching algorithms compared to their serial counterparts.

6.2 Speedup ranking of parallel algorithms

Having determined the optimal search string length, we delved into a more detailed analysis to evaluate and improve the speedup performance of the considered parallel algorithms. This analysis was conducted on the Wikipedia dataset with a 100 MB search file size for the pattern lengths of 3, 10, 50, and 200.

Table 2 presents the top five speedups of parallel algorithms for different pattern lengths m . Speedup represents the improvement in processing

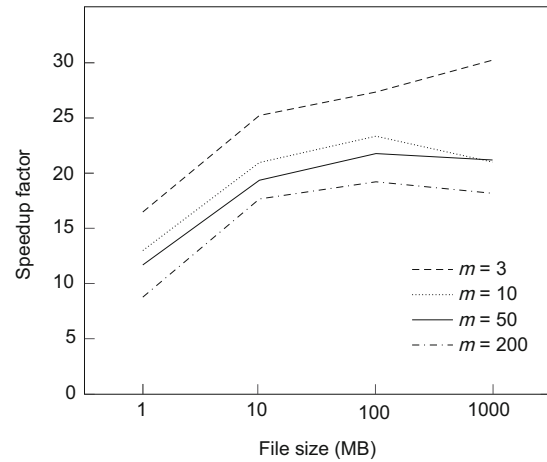


Fig. 2 Average speedup of parallel algorithms

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 $51\times$ 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 ranked among the top five across all tested pattern lengths. 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 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. Using 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 and the bottom five algorithms in terms of running times within the parallel algorithm category.

Table 3 shows the performance of parallel algorithms on 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 exhibited the best performance for the pattern value of 200 in the Wikipedia dataset, completing in 11.1482 ms. In contrast, the ag algorithm demonstrated 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 excelled in specific contexts, highlighting the importance of selecting an algorithm tailored to the characteristics and requirements of the dataset. For instance, when considering parallel algorithms applied to the Genome dataset with a pattern length of 50, the execution time varied

from 11.7540 ms to 129.8907 ms. The variability in execution time underscores how the choice of a parallel algorithm can significantly affect overall performance. In addition to evaluating algorithm performance based on pattern length, we observed interesting consistencies and disparities across the two datasets. One notable finding is the absence of a single algorithm that 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 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

Table 2 Top five speedups of parallel algorithms

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
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 3 Performance ranking of parallel algorithms

Ranking	$m = 3$		$m = 10$		$m = 50$		$m = 200$	
	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
	bsdms	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

W.: Wikipedia; G.: Genome

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.

6.4 Impact of alphabet size

To examine algorithm behaviors 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 encompassed 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 size of 100 MB according to prior experimentation. Our search tests encompassed four distinct patterns, each characterized by different lengths: 3, 10, 50, and 200 characters. These patterns were meticulously crafted through random selection from the corresponding alphabet. Consequently, we created 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 scenario 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. 3 illustrates the average speedup of parallel algorithms in relation to different pattern lengths across various alphabet sizes. 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 2. As the alphabet size increased, the speedup factor decreased, 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 continued to come closer as the alphabet size increased for all the test cases, independent of the pattern length. For alphabet sizes 64 and 128, the

speedup factor almost remained 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.

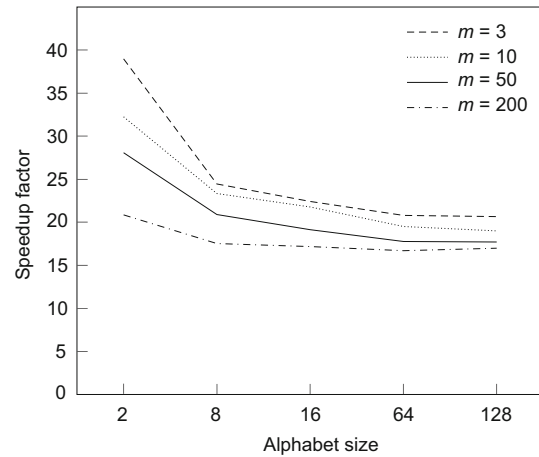


Fig. 3 Impact of alphabet size of parallel algorithms

6.5 Impact of memory setup

We conducted tests using constant memory allocation on a 100 MB text file from the 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 modified kernel transactions to use this memory space.

Table 4 shows the average speedup factors with constant memory compared to with global memory for our parallel algorithms for each pattern. For $m = 3$, the speedup factor was 1.007, which suggested that using constant memory was slightly faster (approximately 0.7% faster) than using global memory. Conversely, for $m = 200$, the speedup factor was 0.997, indicating that using constant memory was slightly slower (approximately 0.3% slower) than using global memory. Overall, the results indicated that there is no significant performance difference

when using constant memory.

Table 4 Comparison of average speedup factors with constant memory versus with global memory

Pattern length m	Speedup factor*
3	1.007
10	1.012
50	1.004
200	0.997

*: speedup factors with constant memory compared to with global memory for our parallel algorithms for each pattern

6.6 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 20 test cases, each designed to explore different aspects of the overlapping optimization technique, 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 of our overlapping tests are presented in Fig. 4. Each bar chart represents the speedup factor achieved for a specific pattern length across various alphabet sizes. For example, an average speedup factor of 1.46 has been achieved without pinned memory, indicating that algorithms using overlapping were generally 46% faster. Notably, the speedup was significantly more pronounced for binary alphabets ($\sigma = 2$), 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 shown in Fig. 4b. 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 stood out as an effective strategy for enhancing algorithm performance.

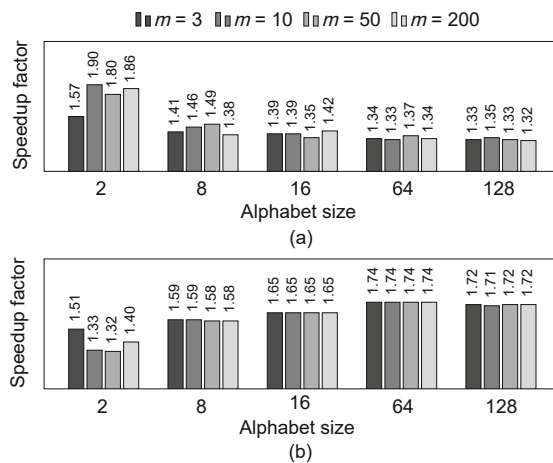


Fig. 4 Overlapping optimization without (a) or with (b) pinned memory

7 Conclusions

String matching is a long-standing and continually researched challenge in the field of information technology, with its utility extending across a

wide array of domains. In this paper, we introduced the CUSMART library, 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 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 the 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 were created using varying text sizes, pattern lengths, and alphabets. We also investigated the impact of memory setup and overlapping optimization on the performance of parallel string matching algorithms.

The results of our experiments demonstrated the following:

1. The enhanced performance achieved in string matching algorithms through parallelization has significantly improved processing efficiency.
2. The average speedup of parallel algorithms varied in relation to different pattern lengths across various text sizes. The optimal text size for our experiments has been determined as 100 MB.
3. In the experiments performed herein, we have observed speedups of up to $51\times$ for the *tbm* algorithm on the Wikipedia dataset under pattern matching tasks with a length of 3 characters.
4. While the performance of parallel algorithms for string matching highly varied based on the text size and pattern length, the *fdm* algorithm consistently appeared among the five lowest performers.
5. For alphabet sizes 64 and 128, the speedup factor remained nearly constant regardless of pattern length. 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.
6. When constant memory was used instead of global memory, there was no significant difference in performance.
7. Using overlapping optimization with pinned memory proved to be an effective strategy for enhancing the performance of parallel string matching algorithms.

Our findings suggested 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 of our study is that not all of the algorithms in the SMART library can be parallelized. This is due to structural limitations or sequential dependencies, which make some algorithms not well suited for parallel processing. In addition, optimizations for GPU parallelization, such as shared memory, cannot be applied to all algorithms.

Based on 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 the workstation and mobile platforms in terms of energy efficiency and cost-effectiveness.

Contributors

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

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are openly available at <https://github.com/smart-tool/smart/tree/master/data> and <https://github.com/adnanozsoy/CUSMART/tree/main/samples>.

Open access

This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images

or other third-party materials in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Adams MD, Celniker SE, Holt RA, et al., 2000. The genome sequence of *Drosophila melanogaster*. *Science*, 287(5461):2185-2195. <https://doi.org/10.1126/science.287.5461.2185>
- Adey SP, 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 Symp on Parallelism in Algorithms and Architectures, p.275-285. <https://doi.org/10.1145/2935764.2935800>
- Baeza-Yates R, Gonnet GH, 1992. A new approach to text searching. *Commun ACM*, 35(10):74-82. <https://doi.org/10.1145/135239.135243>
- Bellekens X, Atkinson RC, Renfrew C, et al., 2013. Investigation of GPU-based pattern matching. Proc 14th Post Graduate Symp on the Convergence of Telecommunications, Networking and Broadcasting, Article 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.
- Crochemore M, Rytter W, 1994. Text Algorithms. Oxford University Press, Inc., New York, 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):13. <https://doi.org/10.1145/2431211.2431212>
- Faro S, Lecroq T, Borzi 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. <https://doi.org/10.1109/MC.2007.445>
- Girkar M, Polychronopoulos CD, 1995. Extracting task-level parallelism. *ACM Trans Program Lang Syst*, 17(4):600-634. <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 Symp 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 on Machine Learning and Data Mining in Pattern Recognition, p.585-600. https://doi.org/10.1007/978-3-540-73499-4_44
- Harris M, 2012. How to Overlap Data Transfers in CUDA C/C++. <https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/> [Accessed on Feb. 3, 2024].
- He LT, Fang BX, Sui J, 2005. The wide window string matching algorithm. *Theor Comput Sci*, 332(1-3):391-404. <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/10.1002/spe.4380100608>
- Jaiswal M, 2014. Accelerating enhanced Boyer–Moore string matching algorithm on multicore GPU for network security. *Int J Comput Appl*, 97(1):30-35. <https://doi.org/10.5120/16973-6934>
- Kadhim HA, AbdulRashid 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):e0139301. <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 Symp 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 of the 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.
- Lu SY, Fu KS, 1978. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Trans Syst Man Cybern*, 8(5):381-389. <https://doi.org/10.1109/TSMC.1978.4309979>
- Mitani Y, Ino F, Hagihara K, 2017. Parallelizing exact and approximate string matching via inclusive scan on a GPU. *IEEE Trans 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 Eng Technol*, 14(3):42-47.
- Navarro G, Raffinot M, 1998. A bit-parallel approach to suffix automata: fast extended string matching. 9th Annual Symp on Combinatorial Pattern Matching, p.14-33.
- NVIDIA, 2019. Record 136 NVIDIA GPU-Accelerated Supercomputers Feature in TOP500 Ranking. <https://blogs.nvidia.com/blog/record-gpu-accelerated-supercomputers-top500/> [Accessed on Feb. 3, 2024].

- Peng JF, 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 Thesis, University of Crete, 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/ijasia.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, 1988. 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 Netw Inform 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 Symp on Principles and Practice of Parallel Programming, p.13-22. <https://doi.org/10.1145/155332.155334>
- Tian XX, Song YL, Wang XL, 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 Symp on Switching and Automata Theory, p.1-11. <https://doi.org/10.1109/SWAT.1973.13>
- Xu KF, Cui WK, Hu Y, et al., 2013. Bit-parallel multiple approximate string matching based on GPU. *Proc 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, Karuppiyah EK, 2013. Hash match on GPU. IEEE Conf on Open Systems, p.150-155. <https://doi.org/10.1109/ICOS.2013.6735065>
- Zha XY, Sahni S, 2013. GPU-to-GPU and host-to-host multipattern string matching on a GPU. *IEEE Trans Comput*, 62(6):1156-1169. <https://doi.org/10.1109/TC.2012.61>
- Ziv J, Lempel A, 1977. A universal algorithm for sequential data compression. *IEEE Trans Inform Theory*, 23(3):337-343. <https://doi.org/10.1109/TIT.1977.1055714>