DİZGİ EŞLEME ALGORİTMALARININ GPGPU HIZLANDIRICILARI KULLANILARAK ETKİLİ VE VERİMLİ HIZLANDIRILMASI

EFFECTIVE AND EFFICIENT PARALLELIZATION OF STRING MATCHING ALGORITHMS USING GPGPU ACCELERATORS

Mengü NAZLI

Assist. Prof. Dr. Adnan ÖZSOY
Supervisor

Submitted to
Graduate School of Science and Engineering of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Engineering

2020

ÖZET

DİZGİ EŞLEME ALGORİTMALARININ GPGPU HIZLANDIRICILARI KULLANILARAK ETKİLİ VE VERİMLİ HIZLANDIRILMASI

Mengü NAZLI

Yüksek Lisans, Bilgisayar Mühendisliği
Danışman: Dr. Öğretim Üyesi Adnan ÖZSOY
Haziran 2020, 117 sayfa

Dizgi eşleme, bilgisayar bilimlerinin en eski ve üzerinde en çok çalışılan konularından biridir. Bilgisayar güvenliği, biyoinformatik, sosyal medya işleme, veri madenciliği, veri sıkıştırma, kodlama teorisi ve benzeri pek çok alanda dizgi işleme uygulamalarına rastlanmaktadır. 1970'lerden bu yana dizgi eşleme problemi üzerine pek çok farklı performans karakteristiğine ve çalışma şekline sahip algoritma önerilmiştir. Bu alandaki çalışmaların çeşitliliği, kapsamlı bir karşılaştırmanın hazırlanmasını zor kılsa da, Thierry Lecroq ve Simone Faro tarafından geliştirilen SMART kütüphanesi gibi bu hedefe ulaşmış birkaç çalışmaya rastlanmaktadır. Bu kütüphane, dizgi işleme alanında çalışmak, literatürdeki algoritmaları karşılaştırmak ve yeni algoritmalar geliştirip mevcut algoritmalarla kıyaslamak isteyen araştırmacılar için değerli bir kaynaktır. Ancak kütüphanedeki kodların seri olarak çalışacak şekilde geliştirilmiş olması, içinde bulunduğumuz yüksek performanslı paralel hesaplama çağında bu kütüphanenin uygulamadaki pratikliğini azaltmaktadır. Bu çalışmada, benzer bir kütüphaneyi Nvidia tarafından geliştirilen CUDA platformundan yararlanarak paralel bir şekilde hazırladık ve dizgi eşleme algoritmalarının paralel çalışma performanslarını incelemeyi amaçladık. CUSMART adı verdiğimiz kütüphanede CUDA C++ programlama arayüzünü kullanarak 85'den fazla dizgi eşleme algoritmasının paralel ortama aktardık ve bu algoritmaları farklı senaryolarda test ederek algoritmaların güçlü ve zayıf yanlarını adil bir şekilde karşılaştırabilmeyi hedefledik. Hazırladığımız algoritmaları hem Nvidia tarafından geliştirilen mobil platform Jetson'da, hem de genel kullanım

için piyasaya sürülmüş olan GeForce serisi kartlarda test ettik. Elde ettiğimiz sonuçlar, aynı algoritmaların seri CPU versiyonlarına göre ortalama 40 kat daha hızlı çalıştığını göstermekte ve GPGPU platformunun dizli işleme uygulamalarındaki potansiyeline işaret etmektedir.

Anahtar Kelimeler: dizgi eşleme, paralel programlama, GPU programlama, GPGPU, NVIDIA, CUDA, CUSMART

ABSTRACT

EFFECTIVE AND EFFICIENT PARALLELIZATION OF STRING MATCHING ALGORITHMS USING GPGPU ACCELERATORS

Mengü NAZLI

Master of Science, Department of Computer Engineering
Supervisor: Assist. Prof. Dr. Adnan ÖZSOY
June 2020, 117 pages

String matching is one of the oldest and actively studied problems in computer science. It has applications in computer security, bio-informatics, social media processing, data mining, data compression, coding theory, and many other areas. Since the '70s, there have been dozens of proposed algorithms to this problem with different approaches and performance characteristics. While the sheer amount of proposed algorithms makes it hard to put together a comprehensive performance comparison study, there are a few projects like the String Matching Algorithms Research Tool (SMART) library from Thierry Lecroq and Simone Faro achieving this goal. Their library holds the code implementations for the majority of string matching algorithms in the literature. It is an invaluable tool for studying different string matching algorithms, but it lacks practicality because of its serial implementation in the age of parallel computation. Our aim is to present a parallel version of the library realized on the CUDA platform by Nvidia, employing GPGPU programming concepts for improved performance and gain insight on the parallel versions of these algorithms. We have developed CUSMART library, which contains parallelized versions of around 85 string matching algorithms using CUDA API. The performances of these algorithms are tested with different scenarios to get a fair comparison and determine their strong/weak application scenarios. Also, we have investigated some well-known optimization practices to observe how they impact the performance of these algorithms. Our results show an average of 40x speedup compared to the serial CPU version of algorithms indicating the potential of GPGPU computing on string matching applications.

Keywords: string matching, parallel programming, GPU programming, GPGPU, NVIDIA, CUDA, CUSMART

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Assist. Prof. Dr. Adnan Özsoy for introducing me to the topic of general-purpose GPU computing and providing support. His helpful comments, remarks, and engagement through the learning process of this master thesis were invaluable to me. He continuously allocated a portion of his valuable time for consultation throughout this work despite his busy schedule.

I also had great pleasure working with Onur Cankur and Behzad Naderalvojoud and appreciate their collaboration on the project which this thesis stemmed from. This thesis would not be complete without their contribution and countless hours of work.

I would like to acknowledge the support of The Scientific and Technological Research Council of Turkey (TÜBİTAK) with grant number 117E142. We have obtained the hardware required to conduct the research, thanks to their funding support.

Particularly helpful to me during the final days of my thesis preparation was Koray Tarakçı, who aided me in finishing the required paperwork. I am indebted to him for his invaluable assistance under difficult circumstances of the pandemic crisis in a quarantined city.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

"*People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.*"

Donald Knuth

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

String matching is an important subject in the text processing field. It is an essential component used by the functioning software on many operating systems. The programming methods emphasized by string matching serve as paradigms in other fields of computer science. Although data are shared and stored in many forms, the text remains as the dominant form of information handling. This is especially apparent in literature where data are composed of large corpora. Similarly, computer science is also a case where massive amounts of data are stored in linear data files. Molecular biology is another example where biological molecules are approximated as the sequence of more fundamental building blocks like amino acids or nucleotides handled as strings. String matching solutions are applied in many other areas such as computer security [1], bio-informatics [2], [3], social media content processing [4], data mining [5], data compression [6], coding theory [7], [8] and forms the basis of these areas.

Data usage and collected data are increasing day by day in these computer-centric areas. The processing and calculation requirements of this incremental data are also on the same scale. It requires new solutions and hardware support to meet the increase. This problem, which is of interest to high-performance computations and Big data, has gained more importance in recent years. The increase in the calculation time due to the increase in data corresponds to very prolonged response times in many applications, which results in slow and unacceptable runtimes. Therefore, it is important to reduce the calculation times for the availability of the systems.

Parallel processing to reduce calculation times is the primary solution approach for high-performance calculations. The Central Processing Unit (CPU) was the leading architecture for many years on which parallel calculations are based on [9]. With the increasing number of processors used in the systems, calculations in the applications are distributed to the multi-system supercomputers and the multi-core processors in each system. The physical boundaries of multi-core processors such as heat and power usage limit the increase in the number of cores in today's CPUs [10], [11]. In order to overcome these limitations, researches using hardware other than the CPU for high-performance computations have an important place in the literature [12], [13]. One of these types of equipment is the Graphics Processing Unit (GPU).

In recent years, GPUs dominated the high-performance computation field because of their

core counts reaching to thousands that lead to a very high processing throughput [10], [11]. The use of GPUs for general-purpose other than image processing is called General Purpose Computing on Graphical Processing Unit (GPGPU). The usage of GPGPUs for high performance has resulted in high speed gains in many applications. Computing on the graphics processor can provide significant performance improvements in parallel algorithms. The GPU and CPU together form a heterogeneous computing model. In this model, the programs take advantage of the powerful features of each unit.

Since the string matching is one of the classical computer science problems, it has also been studied on parallel systems for fast processing of string matching algorithms [14]. However, these studies tackle single algorithms and only aim to achieve higher performance against CPU for that specific algorithm. There is no comprehensive tool or code base for parallel string matching algorithms. For the serial implementation, the String Matching Algorithms Research Tool (SMART) provides an efficient and flexible tool designed for developing, testing, comparing, and evaluating 85 different string matching algorithms [15]. However, since the tool only provides a serial implementation of algorithms, not parallel, it is insufficient to meet the requirements of new systems, especially that requires high computation demands.

The direct transfer of serial methods is restrictive in achieving the actual performance of GPUs, and the solutions offered are application-specific. In this respect, string matching algorithms on GPGPU need to be redesigned and explored considering GPU architecture. Although the development of algorithms requires a certain amount of effort, we believe that the algorithms to be developed considering the limitations and possibilities of GPUs are much more efficient and promising than traditional methods and the best algorithms designed in serial architectures. In this thesis, we introduce CUSMART, which is a parallelization study on GPGPUs through the implementation of classical serial algorithms on GPUs with several optimization techniques applied.

The main contributions of this thesis can be listed as below:

- The parallelization of 85 string matching algorithms from the literature using the CUDA platform.
- Gathering these algorithms under one library called CUSMART in order to provide an easy to use testing environment.
- Optimization of these string matching algorithms to achieve good performance on parallel systems.

- Testing these parallel string matching algorithms on different scenarios to acquire a broad understanding of their behavior in several use cases.
- Determining the best and the worst performing algorithms in different scenarios to provide a guideline for different types of applications.
- Comparing desktop and mobile device performances for string matching operation.

This thesis is structured into chapters in the following order:

In chapter 2, background information required to explain the features of the CUSMART library is presented. This chapter includes brief explanations of the string matching algorithms present in our library and introductory knowledge about the CUDA architecture.

In chapter 3, related works from the literature are listed along with comparisons about their similarities and differences to our project.

In chapter 4, the implementation details of the CUSMART algorithm is presented. Features of the library and the considerations behind some design choices are also discussed in this chapter.

In chapter 5, we have discussed our experiment methodology, prepared test scenarios, and the results acquired by conducting these tests. The discussion about the test results is given in this chapter.

In chapter 6, we have presented our conclusion on the results of this work and shared our final thoughts for possible future works.

2. BACKGROUND

2.1. String Matching

String matching is the process of finding one or many occurrences of a string (which is generally called a *pattern*) in another string (generally called *text*)[16]. The pattern is defined as a finite array $x = x\,[0\,..\,(m-1)]$ with $m \geqslant 0$ where m is the length of pattern and the text is defined as a finite array $y = y\,[0\,..\,(n-1)]$ with $n \geqslant 0$ where n is the length of text. Both text and pattern are formed from a set of characters, which is called *alphabet* and represented as Σ with a length of σ.

Some additional definitions have to be introduced in order to continue the discussion on string matching [16]:

- A word u is called *prefix* of a word w if there is a word v (might be empty) that satisfies the condition $w = uv$.
- A word v is called *suffix* of a word w if there is a word u (might be empty) that satisfies the condition $w = uv$.
- A word z is called *substring* or *factor* or *subword* of word w if there are two words u and v that satisfies the condition $w = uzv$

String matching applications are divided into two types depending on the pattern and the text accessibility before the operation. Algorithms based on the access to pattern beforehand are called *online string matching algorithms*[16]. These algorithms aim to use the combinatorial properties of strings and automatons to preprocess the pattern and gain insight before the search. The other type of algorithms that have access to text before the searching operation aims to solve a problem called *offline string matching problem*. These algorithms use various techniques to index the text and try to gain knowledge to improve search performance before the operation. In this thesis, our focus will be on the online string matching algorithms.

There have been more than 120 online string matching algorithms proposed since 1970. These online string matching algorithms (we are going to refer them as string matching algorithms from now on) can be classified further under four different categories based on the primary strategy employed. These categories are comparison based algorithms, automata-based algorithms, bit-parallel algorithms, and packed algorithms. Some of the

algorithms listed under these categories follow a hybrid approach and use two or more of the methods listed above. For the sake of brevity, these algorithms are categorized under their most prominent category.

In the following sections, the algorithms included in our study will be listed in chronological order. Each algorithm entry will be given with a short description of their main features, and an acronym used to refer to this algorithm in the literature.

2.1.1. Comparison Based Algorithms

Algorithms using character comparison based strategies are the earliest examples of proposed string matching algorithms to the field. These algorithms process the pattern before the search operation to gain insight into its composition. This gained knowledge is often stored in auxiliary arrays called *shift tables*. When a mismatch occurs during the search operation, these tables are utilized to do improved shifts and prevent redundant character comparison operations.

When an algorithm is limited to sequential singular character reads of the text, the optimal complexity of this algorithm can be denoted as $O(n)$. This was first demonstrated by the renowned Morris-Pratt algorithm [17]. However, often it is possible to complete the search operation without reading every character of the text, resulting in sublinear execution complexity on average. The optimal average time complexity for matching in a random string is given as $O(^{n \log_\sigma m}/_m)$ [18], and this is achieved by many algorithms. However, even the algorithms with a sub-linear average-case operation may need to read every character of the text in the worst-case scenario. Many of these algorithms have even worse $O(nm)$ performance in the worst-case scenario [19], [20].

Although character comparison based strategy is not as dominant as it used to be in recent years, it still accounts for the majority of string matching algorithms proposed to this day. The character comparison algorithms related to our study are presented in the following section in chronological order.

2.1.1.1. Brute Force (BF) - N/A

Also known as the naive algorithm [21], the brute force algorithm is the most straightforward implementation of string matching. It checks every text character between 0 and $n-m$ whether starts of a pattern is detected at the position or not. After each successful or failed attempt, a shift happens in the comparison window precisely by one step. The brute force algorithm does no preprocessing and requires constant extra space in memory. The

comparison requires no specific order of operation and can be done in any order. The time complexity of the operation is $O(nm)$ on the worst-case and $O(n)$ at best. The algorithm makes $2n$ comparisons on average.

2.1.1.2. Morris-Pratt (MP) - 1970

Morris-Pratt [17] is the first linear algorithm that emerged after an analysis of the brute-force algorithm (2.1.1.1). The authors demonstrated that it is beneficial to remember the last matched part of the string after a mismatch and use this information to improve the length of the shifts. This improvement reduces comparisons needed and results in better performance. Morris-Pratt algorithm accomplishes this task by preprocessing the pattern and calculating the best shift length for every position in the pattern if a mismatch occurred at that position.

The preprocessing strategy is based on the fact that if a mismatch occurs at ith character of the pattern with $0 < i < m$, there is no need to reprocess the text characters $y[j..j+i]$ where j is the search window's position because we already acquired the information about these characters and know this substring is identical to our pattern, aside from the last, mismatched character. The time and space complexity of the preprocessing phase is $O(m)$, and the time complexity of the searching phase is $O(n+m)$. The searching phase time complexity is independent of σ.

2.1.1.3. Knuth-Morris-Pratt (KMP) - 1977

The Knuth-Morris-Pratt [22] algorithm is an improved version of Morris-Pratt algorithm (2.1.1.2). The preprocessing phase of this algorithm is improved to avoid another immediate mismatch after a shift, which resulted from a mismatch.

Aside from this improvement, space and time complexities of the algorithm are similar to MP, with $O(m)$ preprocessing phase and $O(n+m)$ search phase. The Knuth-Morris-Pratt algorithm performs $2n-1$ comparisons on the worst-case scenario.

2.1.1.4. Boyer-Moore (BM) - 1977

The Boyer-Moore [23] algorithm is another derivation of the brute-force algorithm (2.1.1.1), and it is considered as one of the most efficient string matching algorithms for practical applications. The simplified versions of this algorithm are often used as the search function in text processors [24].

The algorithm applies the left-to-right comparison idea along with a sliding search window approach to achieve the sub-linear time complexity of $O(n/m)$ for the best-case searching scenario. This value remains as the best lower bound achieved for string matching to date. Aside from the left-to-right comparison scan order, the Boyer-Moore algorithm also uses multiple rules such as "bad character rule" and "good suffix rule" to generate shift tables and evaluate these tables when a mismatch occurs in order to determine the longest shift possible. The time and space complexity of the preprocessing phase is $O(m + \sigma)$. In the worst-case scenario, the algorithm does $3n$ comparisons when searching for a non-periodic pattern.

2.1.1.5. Horspool (HOR) - 1980

The Horspool algorithm [25] is a simplification of the Boyer-Moore algorithm 2.1.1.4. This algorithm only uses the "bad character rule" table of the Boyer-Moore for the rightmost character of the search window, which improves the average search performance on large alphabets like ASCII. The preprocessing phase is in $O(m + \sigma)$ time and has $O(\sigma)$ space complexity.

2.1.1.6. Galil-Seiferas (GS) - 1981

The Galil-Seiferas algorithm [26] is a linear algorithm using constant extra space for its auxiliary data structure. The preprocessing step employs a decomposition strategy called *perfect factorization*. The algorithm has $O(m)$ preprocessing phase time complexity and makes $5n$ comparisons in the worst-case scenario.

2.1.1.7. Apostolico-Giancarlo (AG) - 1986

The Apostolico-Giancarlo algorithm [27] is another variant of The Boyer-Moore algorithm (2.1.1.4). This algorithm stores some of the information that is extracted from the text by the Boyer-Moore algorithm but then forgotten after the comparison is made. Using this stored information, the Apostolico-Giancarlo algorithm reduces the worst-case comparison amount to half, which is $3/2n$.

2.1.1.8. Karp-Rabin (KR) - 1987

The Karp-Rabin [28] is the first string matching algorithm that used a hashing function in the comparison step. The algorithm compares the hash of search window and pattern hash first to test the pattern on a given window quickly. If a similarity is detected, the algorithm compares the remaining characters regularly. Unloading the first check to hash

control avoids a quadratic number of character comparisons in most practical situations. The algorithms preprocessing phase time complexity is $O(m)$ with constant space, and the expected running complexity is $O(n + m)$.

2.1.1.9. Zhu-Takaoka (ZT) - 1987

The Zhu-Takaoka algorithm [29] is a variant of the Boyer-Moore algorithm (2.1.1.4), which performs the bad character shift based on the last two characters of the search window instead of one. This change caused an improved average-case performance at the cost of $O(m + \sigma^2)$ preprocessing phase time complexity.

2.1.1.10. Quick Search (QS) - 1990

The Quick Search algorithm [30] is a simplified version of the Boyer-Moore algorithm (2.1.1.4), which uses only the bad character shift table like the Horspool algorithm (2.1.1.5). Their version is shorter, easy to implement, and pretty fast in practice for short patterns on large alphabets.

2.1.1.11. Optimal Mismatch (OM) - 1990

The Optimal Mismatch algorithm [30] is a variation of the Quick Search algorithm (2.1.1.10), and uses the character frequencies in a given alphabet. The algorithm compares the characters of the pattern in a custom order from less frequent to more frequent to "optimize mismatch". Like in the Boyer-Moore algorithm, early mismatches provide long shifts that allow the algorithm to scan the text quicker.

2.1.1.12. Maximal Shift (MS) - 1990

The Maximal Shift algorithm [30] is a variation of the Quick Search algorithm (2.1.1.10), that uses a custom scan order prioritizing the comparisons that result in long shifts. The preprocessing phase has $O(m^2 + \sigma)$ time and $O(m + \sigma)$ space complexity.

2.1.1.13. Apostolico-Crochemore (AC) - 1991

The Apostolico-Crochemore algorithm [31] is an improved version of the Knuth-Morris-Pratt algorithm (2.1.1.3). This algorithm reduces the worst-case comparison amount to $^3/_2n$ from $2n - 1$ of the Knuth-Morris-Pratt.

2.1.1.14. Two Way (TW) - 1991

The Two Way algorithm [32] factorizes the pattern into two parts and does the comparison steps from center to borders. The left part is scanned from right to left while the right part is scanned from left to right. The algorithm has a linear running time of $O(2n - m)$ on worst-case and runs in $O(n)$ on average. The algorithms preprocessing phase time complexity is $O(m)$ with constant space.

2.1.1.15. Tuned Boyer-Moore (TUNBM) - 1991

The Tuned Boyer-Moore algorithm [33] is an improved version of the Boyer-Moore algorithm (2.1.1.4). This algorithm adds a fast path that unrolls a certain number of early comparisons and makes a few blind shifts without comparing the pattern characters other than the last one. This approach speeds up the algorithm in practice with the introduction of the fast path.

2.1.1.16. Colussi (COL) - 1991

The Colussi algorithm [34] is a refinement of the Knuth-Morris-Pratt algorithm, which uses a factorized pattern similar to the Two Way algorithm (2.1.1.14) and improves the worst-case performance to $^3/_2n$ maximum comparisons. The time and space complexity of the preprocessing phase is $O(m)$.

2.1.1.17. Smith (SMITH) - 1990

The Smith algorithm [35] combines the approaches of the Horspool algorithm (2.1.1.5) with The Quick Search algorithm (2.1.1.10). The algorithm compares the bad character shift function results of these two algorithms and selects the more significant value for a longer shift.

2.1.1.18. Galil-Giancarlo (GG) - 1992

The Galil-Giancarlo algorithm [36] is an improved version of the Colussi algorithm (2.1.1.16) which reduces the worst-case comparison amount to $^4/_3n$.

2.1.1.19. Raita (RAITA) - 1992

The Raita algorithm [37] is a modified version of the Horspool algorithm (2.1.1.5) that uses a different scan order. The algorithm compares the last character then the first character; if both comparisons are matches, then it proceeds to compare the rest of the pattern.

2.1.1.20. String Matching on Ordered Alphabet (SMOA) - 1992

The String Matching on Ordered Alphabet algorithm [38] uses constant extra space and has no preprocessing phase. The algorithm makes $6n + 5$ comparisons at worst.

2.1.1.21. Turbo Boyer-Moore (TBM) - 1992

The Turbo Boyer-Moore algorithm [39] is an improved version of the Boyer-Moore algorithm (2.1.1.4). The algorithm lowers the maximum required comparison amount to $2n$ by remembering the last matched suffix and altering the decision mechanism slightly. This version requires no extra preprocessing and only constant extra space.

2.1.1.22. Not So Naive (NSN) - 1993

The Not So Naive algorithm [40] is a simple algorithm that is similar to the Brute Force algorithm (2.1.1.1). This algorithm's comparison order starts from the second character and goes until the end; then, it compares the first character as the last step. This approach allows two-character shifts instead of one when the first comparison results in a mismatch and improves the average-case complexity to slightly sub-linear.

2.1.1.23. Reverse Colussi (RCOL) - 1994

The Reverse Colussi algorithm [41] is a combination of the Colussi algorithm (2.1.1.16) and the Boyer-Moore algorithm (2.1.1.4). By employing the Colussi algorithm's factorization scheme on the Boyer-Moore style search, this algorithm lowers the worst-case maximum comparison amount to $2n$. The preprocessing phase has $O(m^2)$ time and $O(m\sigma)$ space complexity.

2.1.1.24. Skip Search (SKIP) - 1998

The Skip Search algorithm [42] processes each character in the pattern and builds a data structure called "buckets of positions". This structure holds the shift amounts for every character in the pattern and has the width of alphabet size σ. After a right-to-left ordered comparison, the algorithm determines a proper shift via these buckets of positions, then proceeds to compare the rest of the string in a left-to-right fashion.

2.1.1.25. Alpha Skip Search (ASKIP) - 1998

The Alpha Skip Search algorithm [42] is an improved version of the Skip Search algorithm (2.1.1.24). This version uses a bucket of positions for each factor of the pattern instead of character.

2.1.1.26. Knuth-Morris-Pratt Skip Search (KMPS) - 1998

The Knuth-Morris-Pratt Skip Search algorithm [42] is another variant of the Skip Search algorithm (2.1.1.24). This variant uses the shift tables of the Knuth-Morris-Pratt algorithm (2.1.1.3).

2.1.1.27. Berry-Ravindran (BR) - 1999

The Berry-Ravindran algorithm [43] is a hybrid between Zhu-Thakaoka (2.1.1.9) and Quick Search (2.1.1.10) algorithms. The algorithm is designed to do two consecutive shifts based on the bad-character shift table. It has $O(m + \sigma^2)$ space and time complexity.

2.1.1.28. Ahmed-Kaykobad-Chowdhury (AKC) - 2003

The Ahmed-Kaykobad-Chowdhury algorithm [44] is a variant of the Apostolico-Giancarlo algorithm (2.1.1.7). This algorithm remembers the pattern suffixes found in the text and alters the shifts accordingly.

2.1.1.29. Fast Search (FS) - 2003

The Fast Search algorithm [45] presents another improvement to the Boyer-Moore algorithm (2.1.1.4) by employing some occurrence heuristics for the first mismatch case.

2.1.1.30. Forward Fast Search (FFS) - 2004

The Forward Fast Search algorithm [45] combines the Fast Search algorithm (2.1.1.29) with the Quick Search algorithm and implements good-suffix heuristics using the following character's information on the current search window of text.

2.1.1.31. Backwards Fast Search (BFS) - 2004

The Backwards Fast Search algorithm [45] is a combination of the Boyer-Moore algorithm (2.1.1.4) and the Horspool algorithm (2.1.1.5). It implements good-suffix heuristics using the mismatching character information.

2.1.1.32. Tailed Substring (TS) - 2004

The Tailed Substring algorithm [46] employs a variation of occurrence heuristics used in the Horspool algorithm (2.1.1.5) to improve the naive approach.

2.1.1.33. Sheik-Sumit-Anindya-Balakrishnan-Sekar (SSABS) - 2004

The Sheik-Sumit-Anindya-Balakrishnan-Sekar algorithm [47] uses the concepts found in the Raita algorithm (2.1.1.19) and the Quick Search algorithm (2.1.1.10).

2.1.1.34. Thathoo-Virmani-Sai-Balakrishan-Sekar (TVSBS) - 2006

The Thathoo-Virmani-Sai-Balakrishan-Sekar algorithm [47] combines the SSABS algorithm (2.1.1.33) and the Berry-Ravindran algorithm (2.1.1.27).

2.1.1.35. Boyer-Moore-Horspool using Probabilities (PBMH) - 2006

The Boyer-Moore-Horspool using Probabilities algorithm [47] applies the statistical character analysis approach of the Optimal Mismatch algorithm (2.1.1.11) to the Horspool algorithm (2.1.1.5).

2.1.1.36. Franek-Jennings-Smyth (FJS) - 2007

The Franek-Jennings-Smyth algorithm [48] applies the concepts found in the Quick Search algorithm (2.1.1.10) to the Knuth-Morris-Pratt algorithm (2.1.1.3).

2.1.1.37. Wu-Manber for Single Pattern Matching (HASHQ) - 2007

The Wu-Manber for Single Pattern Matching algorithm [49] is an improvement over the Horspool algorithm (2.1.1.5). This algorithm demonstrates a use case of super alphabets and computes the q-gram fingerprints of the pattern using a hashing function.

2.1.1.38. Boyer-Moore-Horspool with q-grams (BMHQ) - 2008

The Boyer-Moore-Horspool with q-grams algorithm [50] offers a new approach to the well-known Horspool algorithm (2.1.1.5). The algorithm introduces the use of q-grams to compute the occurrence heuristics.

2.1.1.39. Two Sliding Windows (TSW) - 2008

The Two Sliding Windows algorithm [50] is an improvement of the Quick Search algorithm (2.1.1.10) that adds another search window to the process. The first window processes from left to right while the second window processes from right to left.

2.1.2. Automata based algorithms

The automata based algorithms are indispensable tools in the field of string matching as they are able to handle the string matching tasks very efficiently. The first algorithm applying the technique to string matching, Deterministic Finite Automata was one of the first algorithms that achieved linear time complexity on the search task [21]. Then there is the Backward DAWG Matching algorithm, which reached the optimal lower bound $O(n^{\log_\sigma(m)}/m)$ time complexity for the average case. Both these algorithms capitalize on the advantages of using a finite automaton for the job. The efficiency of a given algorithm is varied based on the automaton used for pattern representation and the simulation technique of automata if it is present.

Most of the algorithms in this section stem from the Deterministic Finite Automata (2.1.2.1) and apply the concepts found in comparison based algorithms (2.1.1) to automata-based string matching.

The following list contains the automata-based string matching algorithms that are relevant to our study in chronological order.

2.1.2.1. Deterministic Finite Automata (DFA) - N/A

The Deterministic Finite Automata [21] is a string matching algorithm that works in linear time. In order to perform the search, the algorithm first builds a Deterministic String Automaton $A(x)$ from the pattern x. The preprocessing phase which consist of the construction of automaton has $O(m + \sigma)$ time and $O(m\sigma)$ space complexity. After the Deterministic Finite Automaton is constructed, searching of the pattern for a given text y can be accomplished by parsing the text character by character using the automaton $A(x)$ and advancing the state each time a termination occurs. If the automaton structure is stored in a direct access table, the search step can be completed in $O(n)$ time. Else, it takes $O(n \log \sigma)$ time to complete the search.

2.1.2.2. Reverse Factor (RF) - 1992

The Reverse Factor algorithm [51] applies the Boyer-Moore algorithm (2.1.1.4) strategies to the Deterministic Finite Automata (2.1.2.1). The algorithm builds a suffix automaton from the reversed version of the pattern. The time complexity of the worst-case search operation is $O(mn)$.

2.1.2.3. Simon (SIM) - 1994

The Simon algorithm [52] is a modification of the Deterministic Finite Automata (2.1.2.1). The algorithm builds a minimal automaton by processing the pattern.

2.1.2.4. Turbo Reverse Factor (TRF) - 1994

The Turbo Reverse Factor algorithm [53] improves over the Reverse Factor algorithm (2.1.2.2). The algorithm remembers the last matched prefix of the pattern to avoid redundant comparison operations. By utilizing this strategy, it is possible to achieve $O(2n)$ worst-case time complexity.

2.1.2.5. Forward DAWG Matching (FDM) - 1994

The Forward DAWG Matching algorithm [53] uses a smallest suffix automaton (also called Directed Acyclic Word Graph) to achieve linear time search operation. This was a notable algorithm among string matching algorithms of its era because of its constant search operation time, which is not affected by the length of the pattern. The algorithm has $O(n)$ time complexity and performs exactly n comparisons even on the worst-case.

2.1.2.6. Backward DAWG Matching (BDM) - 1994

The Backward DAWG Matching algorithm [53] is a variant of the Reverse Factor algorithm (2.1.2.2) and uses Directed Acyclic Word Graph of the pattern. The algorithm works similarly to the Forward DAWG Matching algorithm (2.1.2.5), but the scan direction is reversed to right-to-left.

2.1.2.7. Backward Oracle Matching (BOM) - 1999

The Backward Oracle Matching algorithm [54] is a variation of the Reverse Factor algorithm (2.1.2.2) that uses the suffix oracle of the pattern instead of the suffix automaton. The suffix oracle data structure is a very compact automaton with the capability of recognizing at least all suffixes of a word. Although the suffix oracle also recognizes the factors

not present in the pattern, it still can be used for string search because the only factor recognized by the oracle that has a length of m or longer is the pattern itself in its reversed form. The time and space required for the preprocessing phase are $O(m)$. The algorithm achieves optimal time in average-case and has $O(mn)$ worst-case complexity.

2.1.2.8. Double Forward DAWG Matching (DFDM) - 2000

The Double Forward DAWG Matching algorithm [55] is a modification of the Forward DAWG Matching algorithm (2.1.2.5). The algorithm achieves linear worst-case time by using another DAWG that is constructed from the reverse of the pattern.

2.1.2.9. Wide Window (WW) - 2005

The Wide Window algorithm [56] is a combination of the Reverse Factor algorithm (2.1.2.2) and the Forward DAWG Matching algorithm (2.1.2.5). By employing two automata built from the suffixes of pattern and the prefixes of the reverse of the pattern, this algorithm achieves linear time in the worst-case scenario.

2.1.2.10. Linear DAWG Matching (LDM) - 2005

The Linear DAWG Matching algorithm [56] combines the automata from the Deterministic Finite Automata (2.1.2.1) and the Backward DAWG Matching algorithm (2.1.2.6). Deterministic finite automaton of the pattern is utilized along with the suffix automaton of the pattern.

2.1.2.11. Improved Linear DAWG Matching (ILDM) - 2006

The Improved Linear DAWG Matching algorithm [57] applies some tweaks over the Linear DAWG Matching algorithm (2.1.2.10). There are two versions of this algorithm referenced as ILDM1 and ILDM2, which are published in the same study.

2.1.2.12. Extended Backward Oracle Matching (EBOM) - 2008

The Extended Backward Oracle Matching algorithm [58] is an improved version of the Backward Oracle Matching algorithm (2.1.2.7). The algorithm performs transitions based on the checks of two characters instead of one.

2.1.2.13. Forward Backward Oracle Matching (FBOM) - 2008

The Forward Backward Oracle Matching algorithm [58] applies the Quick Search algorithm (2.1.1.10) strategies to the Extended Backward Oracle Matching algorithm (2.1.2.12). The transitions are performed based on the characters trailing the current search window.

2.1.2.14. Simplified Extended Backward Oracle Matching (SEBOM) - 2009

The Simplified Extended Backward Oracle Matching algorithm [59] is a simplified version of the Extended Backward Oracle Matching algorithm (2.1.2.12).

2.1.2.15. Simplified Forward Backward Oracle Matching (SFBOM) - 2009

The Simplified Forward Backward Oracle Matching algorithm [59] is a simplified version of the Forward Backward Oracle Matching algorithm (2.1.2.13).

2.1.2.16. Backward SNR DAWG Matching (BSDM) - 2012

The Backward SNR DAWG Matching algorithm [60] is a modified version of the Backward DAWG Matching algorithm (2.1.2.6). The DAWG structure used on this algorithm is built from the longest substring of the pattern that contains no repeated characters.

2.1.3. Bit-Parallel Based Algorithms

Bit-parallelism is a technique based on the efficient simulation of nondeterministic automata [20]. It takes advantage of the computing unit's inherent properties, in which bit-wise operations can be performed word by word in parallel. This parallelism can be used to cut down the number of instructions required for comparisons in the search step. The number of operations required can be cut down by a factor of w, where w is the number of bits contained in the word that the processor can handle. This property makes bit-parallel algorithms considerably fast when the pattern length $m \leq w$ so that the pattern can fit into a computer word, and only a single operation is sufficient for parallel comparison. The performance of bit-parallel algorithms starts to degrade considerably once the $m > w$, as the m/w grows.

The following list contains the bit-parallelism based string matching algorithms that are relevant to our study in chronological order.

2.1.3.1. Shift-Or (SO) - 1989

The Shift-Or algorithm [61] is the first bit-parallelism technique applied to the string matching problem. The algorithm uses bitwise OR operation followed by shift operations to scan the text for the occurrence of a given pattern. The efficiency of the algorithm is high when the pattern fits into a single computer word. The complexity of this algorithm is $O(n)$ for all scenarios, and running time is not affected by the pattern length or the alphabet size. The preprocessing phase takes $O(m + \sigma)$ time and requires $O(m + \sigma)$ space. This algorithm is suitable for approximate string matching operations with minor tweaks.

2.1.3.2. Shift-And (SA) - 1989

The Shift-And algorithm [61] is similar to the Shift-Or algorithm (2.1.3.1) and simulates a nondeterministic version of the Deterministic Finite Automata algorithm (2.1.2.1). Unlike the Shift-Or algorithm, this algorithm uses bitwise AND operation as the comparison method.

2.1.3.3. Backward Nondeterministic DAWG Matching (BNDM) - 1998

The Backward Nondeterministic DAWG Matching algorithm [62] is a nondeterministic simulation of the Backward DAWG Matching algorithm (2.1.2.6).

2.1.3.4. Backward Nondeterministic DAWG Matching for Long Patterns (LBNDM) - 2000

The Backward Nondeterministic DAWG Matching for Long Patterns algorithm [63] improves the behavior of the BNDM algorithm (2.1.3.3) over long patterns.

2.1.3.5. Simplified Backward Nondeterministic DAWG Matching (SBNDM) - 2003

The Simplified Backward Nondeterministic DAWG Matching algorithm [64] is an improved version of the Backward Nondeterministic DAWG Matching algorithm (2.1.3.3).

2.1.3.6. Two-Way Nondeterministic DAWG Matching (TNDM) - 2003

The Two-Way Nondeterministic DAWG Matching algorithm [64] is a variation of the Backward Nondeterministic DAWG Matching algorithm (2.1.3.3). The algorithm performs a forward scan of the pattern suffix before the backward scan.

2.1.3.7. Shift Vector Matching (SVM) - 2003

The Shift Vector Matching algorithm [64] implements a variation of the Boyer-Moore algorithm (2.1.1.4) using bit-parallelism. The information gathered during the last comparison attempt is stored to improve the performance.

2.1.3.8. BNDM with loop unrolling (BNDM2) - 2005

The BNDM with loop unrolling algorithm [65] improves the performance of the Backward Nondeterministic DAWG Matching algorithm (2.1.3.3) via unrolled loops and blind shifts.

2.1.3.9. Simplified BNDM with loop unrolling (SBNDM2) - 2005

The Simplified BNDM with loop unrolling algorithm [65] improves the performance of the Simplified Backward Nondeterministic DAWG Matching via loop unrolling method.

2.1.3.10. BNDM with Boyer-Moore-Horspool Shift (BNDMBMH) - 2005

The BNDM with Boyer-Moore-Horspool Shift algorithm [65] combines the Backward Nondeterministic DAWG Matching algorithm (2.1.3.3) with the shift tables from the Horspool algorithm (2.1.1.5).

2.1.3.11. Horspool with BNDM test (BMHBNDM) - 2005

The Horspool with BNDM test algorithm [65] is another approach at combining the Horspool algorithm (2.1.1.5) with the BNDM algorithm (2.1.3.3).

2.1.3.12. Forward Nondeterministic DAWG Matching (FNDM) - 2005

The Forward Nondeterministic DAWG Matching algorithm [65] is a simulated nondeterministic version of the Forward DAWG Matching (2.1.2.5).

2.1.3.13. Bit Parallel Wide Window (BWW) - 2005

The Bit Parallel Wide Window algorithm [56] is a simulated nondeterministic version of the Wide Window algorithm (2.1.2.9).

2.1.3.14. Average Optimal Shift-Or (AOSO) - 2005

The Average Optimal Shift-Or algorithm [66] borrows the optimal shift concept from the Optimal Shift algorithm (2.1.1.11) and applies to the Shift-Or algorithm (2.1.3.1). This modification allows longer shifts and improves performance.

2.1.3.15. Fast Average Optimal Shift-Or (FAOSO) - 2005

The Fast Average Optimal Shift-Or algorithm [66] is an improved version of the Average Optimal Shift-Or algorithm (2.1.3.14).

2.1.3.16. Forward BNDM (FBNDM) - 2008

The Forward BNDM algorithm [58] is a simulated nondeterministic version of the Forward Backward Oracle Matching algorithm (2.1.2.13).

2.1.3.17. Forward Simplified BNDM (FSBNDM) - 2008

The Forward Simplified BNDM algorithm [58] is a combination of the Simplified BNDM algorithm (2.1.3.5) and the Forward BNDM algorithm (2.1.3.16).

2.1.3.18. Bit-Parallel Length Invariant Matcher (BLIM) - 2008

The Bit-Parallel Length Invariant Matcher algorithm [67] tries to overcome the pattern length limitation of the bit-parallel algorithms like the Backward Nondeterministic DAWG Matching (2.1.3.3) with a length invariant approach. The algorithm shows improved performance over small alphabet searches like DNA matching.

2.1.3.19. Backwards Nondeterministic DAWG Matching with q-grams (BNDMQ) - 2009

The Backwards Nondeterministic DAWG Matching with q-grams algorithm [68] is another BNDM algorithm (2.1.3.3) variant that uses super alphabets approach implemented via q-grams.

2.1.3.20. Simplified BNDM with q-grams (SBNDMQ) - 2009

The Simplified BNDM with q-grams algorithm [68] is a variant of the SBNDM algorithm (2.1.3.5) employing q-gram based super alphabets.

2.1.3.21. Forward Nondeterministic DAWG Matching with q-grams (FNDMQ) - 2009

The Forward Nondeterministic DAWG Matching with q-grams algorithm [68] is an implementation of the Forward Nondeterministic DAWG Matching algorithm (2.1.3.12) using q-grams.

2.1.3.22. Small Alphabet Bit-Parallel (SABP) - 2009

The Small Alphabet Bit-Parallel algorithm [69] is designed to work on small alphabets like binary and DNA. It uses a match table built based on the character positions.

2.1.3.23. Backwards Nondeterministic DAWG Matching with Extended shifts (BXS) - 2010

The Backwards Nondeterministic DAWG Matching with Extended shifts algorithm [70] is an optimized version of the Backwards Nondeterministic DAWG Matching algorithm (2.1.3.3) for long patterns. The algorithm uses a higher level pattern layered over the pattern to construct the automata of long patterns.

2.1.3.24. Factorized BNDM (KBNDM) - 2010

The Factorized BNDM algorithm [71] is a simulated nondeterministic version of the Backward Nondeterministic DAWG Matching algorithm (2.1.3.3). The algorithm builds a brief version of the automaton based on the pattern factorization.

2.2. General-purpose Computing on Graphics Processing Unit with CUDA

The High-Performance Computing (HPC) environment is continually evolving as new technology and methods become widespread, and HPC concept shifts accordingly. Generally, it involves the use of multiple processors or computers to perform a complex task in parallel in order to achieve high throughput in an efficient manner [72]. High-performance computing has developed dramatically over the last decade, mainly thanks to the emergence of heterogeneous architectures using GPU-CPU pair, which has led to a fundamental shift in parallel programming.

Parallel computing can be defined in several ways from different standpoints. From a computational standpoint, it can be defined as performing many operations in parallel, on the premise that large tasks can be divided into smaller tasks to be solved simultaneously. Another definition is a logical problem from the programmer's perspective about how to map the simultaneous calculations onto computers. Suppose there are several devices for the computing job. Parallel computing can then be described as using multiple computing resources (cores or computers) simultaneously to perform parallel calculations. These two aspects of parallel computing can be examined under the computer architecture and the parallel programming.

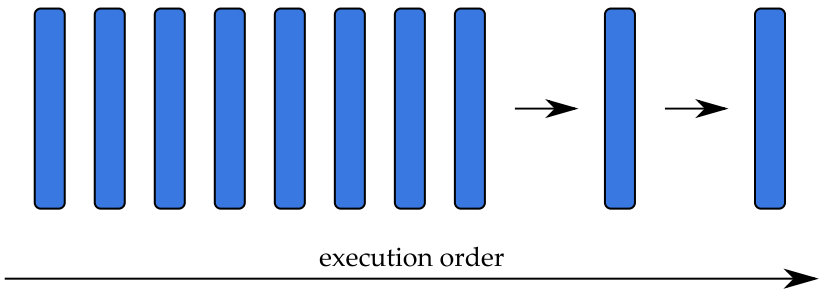

FIGURE 2.1: Serial execution of work partitions in order.

Computer architecture focuses on promoting architectural parallelism, while parallel programming focuses on efficiently solving the problem by utilizing the computing power of the computer architecture. To achieve parallel execution in software, the hardware needs to have a mechanism that allows multiple processes or multiple threads to be executed at the same time.

It is typical to divide a problem into a discrete series of calculations when solving a problem with a computer program; each calculation is performed on a specified part of the problem, as shown in figure 2.1. These parts can have relationships between each other, specifically depending on the result of the prior operation. If there is a constraint of precedence between calculation parts, these calculations have to be performed sequentially

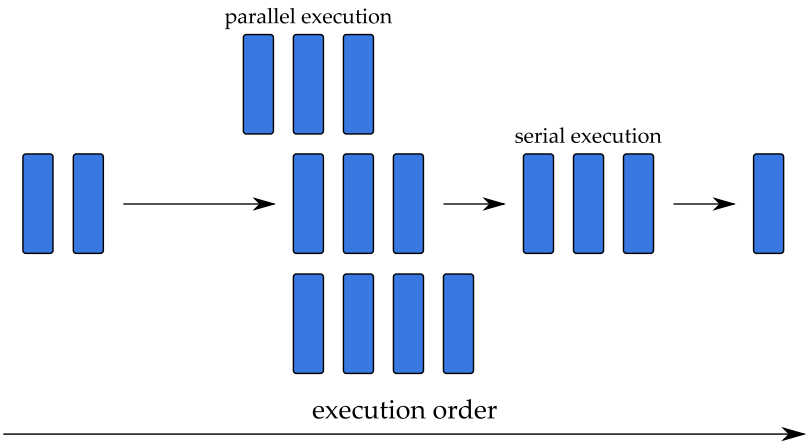

FIGURE 2.2: Execution of work partitions in serial and parallel order.

in order to progress. Otherwise, if there is no constraint of precedence, the calculations can be performed simultaneously on different processing units (figure 2.2).

This property is also called *data dependency*, and it plays an important role in how well a computation can be handled concurrently. Data dependency is usually one of the biggest barriers to parallel programming. It has to be examined thoroughly and eliminated as much as possible in order to achieve some amount of parallelization.

Mainly, the parallelism can be classified under two categories for applications:

- Task level parallelism
- Data level parallelism

In task-level parallelism, the tasks that make up the job are distributed among the processing units and handled concurrently. This approach requires that the individual tasks without dependencies are already present for the selected operation.

In contrast, data-level parallelism divides the main task into sub-tasks, each handling a small part of the whole data. Data level parallelism aims to distribute the data into processing units and apply the same calculation in parallel.

The situations where task-level parallelism can be realized are much rarer compared to the latter. Also, many task-level parallelism scenarios can be translated into data-parallel versions, while the opposite is not true. CUDA programming is particularly suitable for addressing the problems with data-level parallelism. Many applications working on the large datasets employ data-level parallelism to speed up the computation. In order to

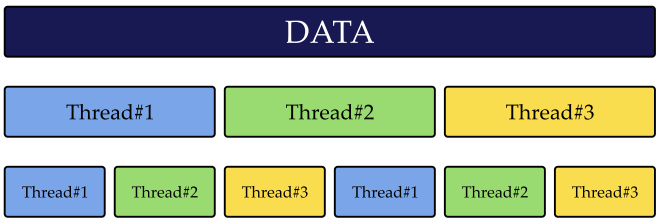

FIGURE 2.3: Different types of partitioning.Top row shows the entire data, second row is block partitioned configuration and the third row is cyclic partitioned configuration.

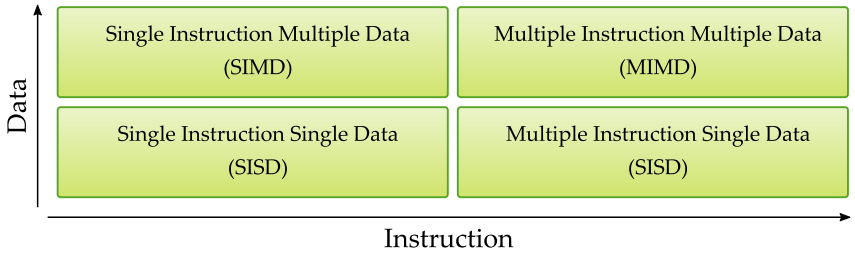

FIGURE 2.4: Different types of instruction and data level parallelism configurations. (Source: [72])

achieve data level parallelism, the data has to be partitioned and distributed across multiple cores. There are two strategies that can be used when partitioning the data for data-parallel applications: The block partitioning and the cyclic partitioning. In block partitioning, data pieces are combined, so every thread processes only one part of it, meaning there are as many data partitions as the thread count. In cyclic partitioning, the threads process more than one part of the data. After processing one partition, the thread gets another one, usually leaping in memory for the number of threads ahead. This way, threads take turns, or "cycle" while requesting data part to process as seen on figure 2.3.

Flynn Taxonomy is a widely accepted classification scheme for computer architecture types [73]. It has four categories for different data and instruction parallelism configurations, as depicted in figure 2.4.

Single Instruction Single Data (SISD) is the most simple design. It includes a single processing unit that performs computation serially. This architecture was very common for CPU designs before the parallel computing gained traction. It is still widely used in many systems.

Single Instruction Multiple Data (SIMD) is a type of parallel computer architecture. In this architecture, there are many cores in the system, and these cores execute the same instructions on different data sources during operation. On most modern computers today, this vectorized computing scheme is implemented for the performance gains. It provides a

good balance between simplicity and performance: Developing applications for the SIMD architecture is relatively simple since the instructions are shared between threads so that the programmer can design the code serially, and the computer handles the organization of multi-threaded operation.

Multiple Instruction Single Data (MISD) is an unusual architecture that is rarely used. In this architecture, many cores operate on the same data source and execute different instructions.

Multiple Instruction Multiple Data (MIMD) is a large scale computer architecture where multiple processors work on multiple data sources. These types of computers usually include sub-systems of SIMD architectures too.

The Central Processing Units with many core units are referred as multi-core processors. The multi-core processors are some of the most prominent parallel computing devices. Aside from these processors, another type of processor gained popularity for parallel computation in recent years. The term *many-core* is used to describe these new kinds of processors. Many-core is used to describe multi-core architectures with a large number of cores, typically containing thousands of cores.

GPUs can be given as an example of many-core systems. There are several different forms of parallelism, such as SIMD, MIMD, or multithreading, that can be realized on a GPU. Because of these traits, NVIDIA coined the term SIMT (Single Instruction Multiple Threads) for their architectures.

GPU architectures are not merely improved versions of the CPU. They do not branch off from a shared predecessor; both are based on different design principles. Both architectures contain processing units referred as cores, but even though they share the name, they have different designs. While the CPU cores are designed for fairly complex logic operations and optimized for serial workflow, GPU cores are more lightweight with simpler logic capabilities and focused on high throughput operations. The difference can be tracked down to the origins of GPU devices. GPU devices originally started their service life as accelerators for graphical computing jobs. Graphical operations usually require simple, repeating transformations on the whole data with low-latency, so the high throughput is important. Later, they are started to be utilized in a broader scope of applications hence the general-purpose computing on graphics processing unit (GPGPU) name.

Fundamental differences between two computing systems allowed them to co-exist as a compound computing unit instead of direct competitors in a high-performance computing

environment. This situation was advantageous for GPU based computing and made it possible to improve along with CPUs without many hindrances.

2.2.1. Heterogeneous Architecture

In the early days of computing, the computers used to contain only CPUs for computational tasks. However, since the last decade, more and more high-performance computers are transitioning into hybrid approaches, utilizing different accelerators and processors. Among these peripherals, the most popular accelerator of choice is the graphics processing units as they gain more focus on general computing tasks.

The impact of the GPU devices on general-purpose computing can be understood when the top lists of supercomputers throughout the years are examined. The latest study shows that 40 percent of the total computing power of the TOP 500 supercomputers list comes from GPU accelerated systems [74]. To put the rapid spread of the technology in perspective, there were no systems with GPU accelerators just over a decade ago on this list. On a similar list named Green500 [75], which ranks the supercomputers based on their floating-point operations per watt performance, 90 percent of the systems on the list include GPU accelerators, indicating the power efficiency of the heterogeneous systems thanks to the GPU devices.

Heterogeneous systems include CPU and GPU devices, which are discrete components and communicate over the PCI-Express bus in a system (figure 2.5). This operation scheme adds a layer of complexity and increases the difficulty of application design for an optimally utilized hardware.

Heterogeneous systems contain at least one central processing unit and some number of graphics processing units. The GPU devices we use today cannot operate standalone and requires a CPU to coordinate the computing task. Because of their ancillary operation on the primary system, they are often named as the *device*, while the CPU is referred as the *host* in GPU computing terms to distinguish where the operation takes place.

Using this naming scheme, the application code can be split into two parts; host code and device code. The host code is the part of the program where the instructions are executed on the CPU. The CPU side of the code is usually responsible for administering the whole operation, sending the data to the GPU device, and retrieving it back, initiating the GPU operation. The device code is the part to be executed on the GPU. This code section includes logic to carry parallel operation on the device, written using a high-level API like CUDA in Nvidia's case.

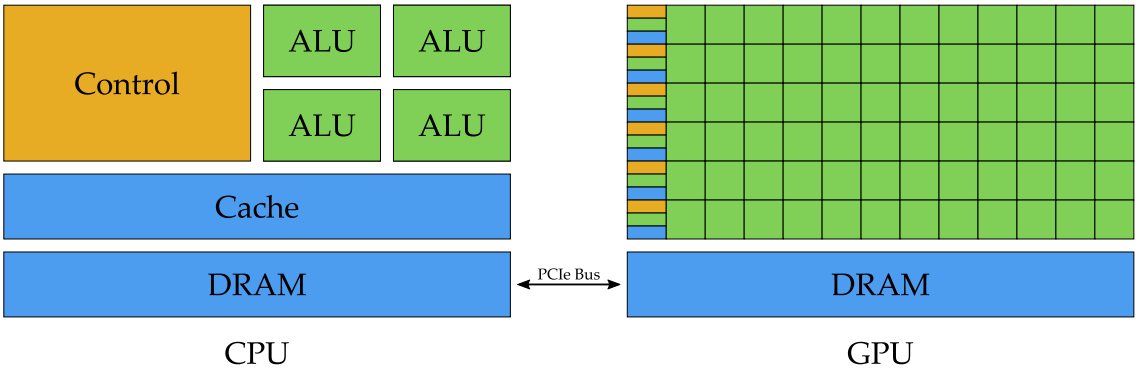

FIGURE 2.5: Communication between CPU and GPU. (Source: [72])

Nvidia has several product families with GPU computing enabled. These families are:

- Tegra
- GeForce
- Quadro
- Tesla

GeForce devices are marketed as consumer graphics units, Tesla for parallel computing in data centers, Quadro is for professional visualization, and the Tegra is targeted for mobile and embedded platforms. Jetson is also a product family that uses Tegra SoC (System-on-Chip) microprocessors and is targeted for autonomous mobile and AI applications.

Underlying these product families, there are different generations of microarchitectures developed by Nvidia (table 2.1). These generations have a code name and compute capability number assigned by Nvidia to indicate the version of each iteration. With every iteration, Nvidia introduces more features and new opportunities to improve parallelism further.

There are two main features for every Nvidia GPU, indicating the capabilities of the device:

- Number of CUDA cores
- Memory Size

Additionally, two main performance metrics are also defined to describe the performance of a given device further:

- Peak computational performance

Year	Microarchitecture	Compute capability versions
2006	Tesla	1.0 - 1.3
2010	Fermi	2.0 - 2.1
2012	Kepler	3.0 - 3.7
2014	Maxwell	5.0 - 5.3
2016	Pascal	6.0 - 6.2
2017	Volta	7.0 - 7.2
2019	Turing	7.5

TABLE 2.1: Microarchitectures introduced by Nvidia over years.

- Memory bandwidth

Peak computational performance metric indicates the maximum number of floating single-point or double-point operations that can be executed per second. The value is usually expressed in billions or trillions of floating-point operations per second, named `gflops` or `tflops`.

2.2.2. Paradigm of Heterogeneous Computing

Using GPU as a computing platform is never considered as a total replacement for CPU computing. Both have their advantages and disadvantages and cannot be ruled as the best overall. CPU computing is suitable for the operations where the workload requires complex calculations, has an unpredictable logic workflow, or the data is too small and unfit for parallel operation. On the other hand, GPU computing is intended to work on the data that is large but requires simple operations like fewer logic decisions and smooth workflow. GPU device can be utilized to carry the operation in a parallel fashion when the data has a profile matching this description.

A combined system of these two units can provide the best of both worlds as the GPU and CPU systems fill the shortcomings of each other. Using such a heterogeneous system, the application can be designed in a way that allows each processor to handle the part of operation most suitable for them. This allows efficient utilization of the computing power, and it is the intended way to perform GPGPU operations on Nvidia CUDA supported devices.

2.2.3. CUDA: A Platform for Heterogeneous Computing

CUDA is a general-purpose parallel computing platform and programming model designed to harness the parallel processing capabilities of the supported Nvidia GPU devices. The CUDA programming platform is available through many popular programming languages

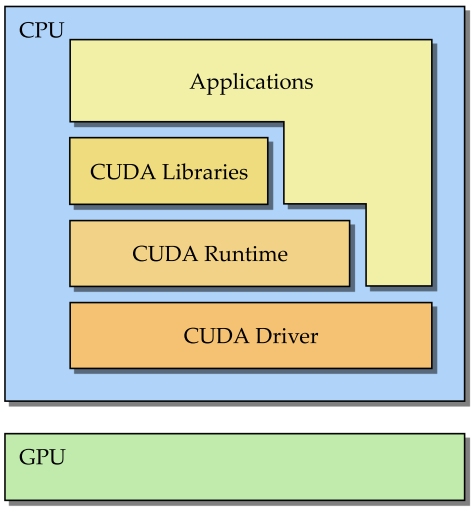

FIGURE 2.6: CUDA application layers (Source: [72])

such as C, C++, Fortran, Java, and Python. In our CUSMART library, we have used the CUDA C toolkit and implemented our library in C language.

CUDA C toolkit is intended to have a low entry barrier and designed as an extension over traditional programming language C. This way, the programmers can harness the power of GPU devices using a language they are familiar with, and achieve high-performance parallel computing. CUDA offers two APIs to access the features of the GPU device. These APIs are the Runtime API and the Driver API (figure 2.6).

The driver API provides low-level access to the device functions and allows fine-grained control of the operation. The runtime API is a high-level API, resulting in an easier to use interface. Runtime API is adequate for most applications since even though it is a higher API level, this interface provides enough granularity of control for frequently used features. The directives included in the runtime API are translated into driver API equivalents at the compilation time.

Nvidia has released their own compiler to develop CUDA applications. This compiler is called Nvidia's CUDA Compiler, usually abbreviated as NVCC, and it is based on a popular, open-source compiler called LLVM. This compiler is included in the CUDA toolkit along with other useful tools for compiling and developing CUDA applications and code examples to get started.

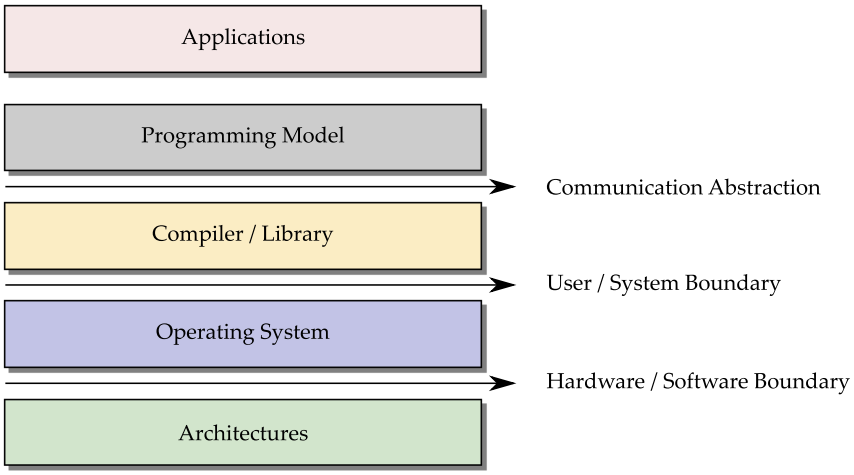

FIGURE 2.7: CUDA programming abstraction layers. (Source: [72])

2.2.4. CUDA Programming Model

Programming models provide layers of abstraction and connect the application implementation with the hardware. The model lies under the application and above compiler, separated by the communication abstraction from below. Some of these important abstraction layers are given on figure 2.7

The CUDA programming model has similarities with many other parallel programming models. Additionally, the CUDA programming model provides means to organize threads and access memory on the GPU through an established hierarchical structure.

The CUDA programming structure allows the programmers to develop their applications for heterogeneous computing by using a small set of decorator commands.

One of the most fundamental blocks of the CUDA programming model is the *kernel*. Kernel is a GPU side function and holds the part of the computation that is carried on the GPU device. It carries the bulk of operation in a heterogeneous application, yet its implementation is relatively straightforward. The kernel code is prepared in serial fashion and does not contain any complex structure to setup parallelism. Instead, there are predefined variables made available in the kernel function scope in order to build parallel operation logic. When the kernel function is called, CUDA manages the scheduling of threads and distributes kernel code to be run on a selection of threads. Each thread communicates its ID programmatically via these predefined variables so the kernel code can define the corresponding thread's workload.

Since the kernel function is executed on the device side, the operation is asynchronous from the host's perspective. When the kernel function invoke command is given, the

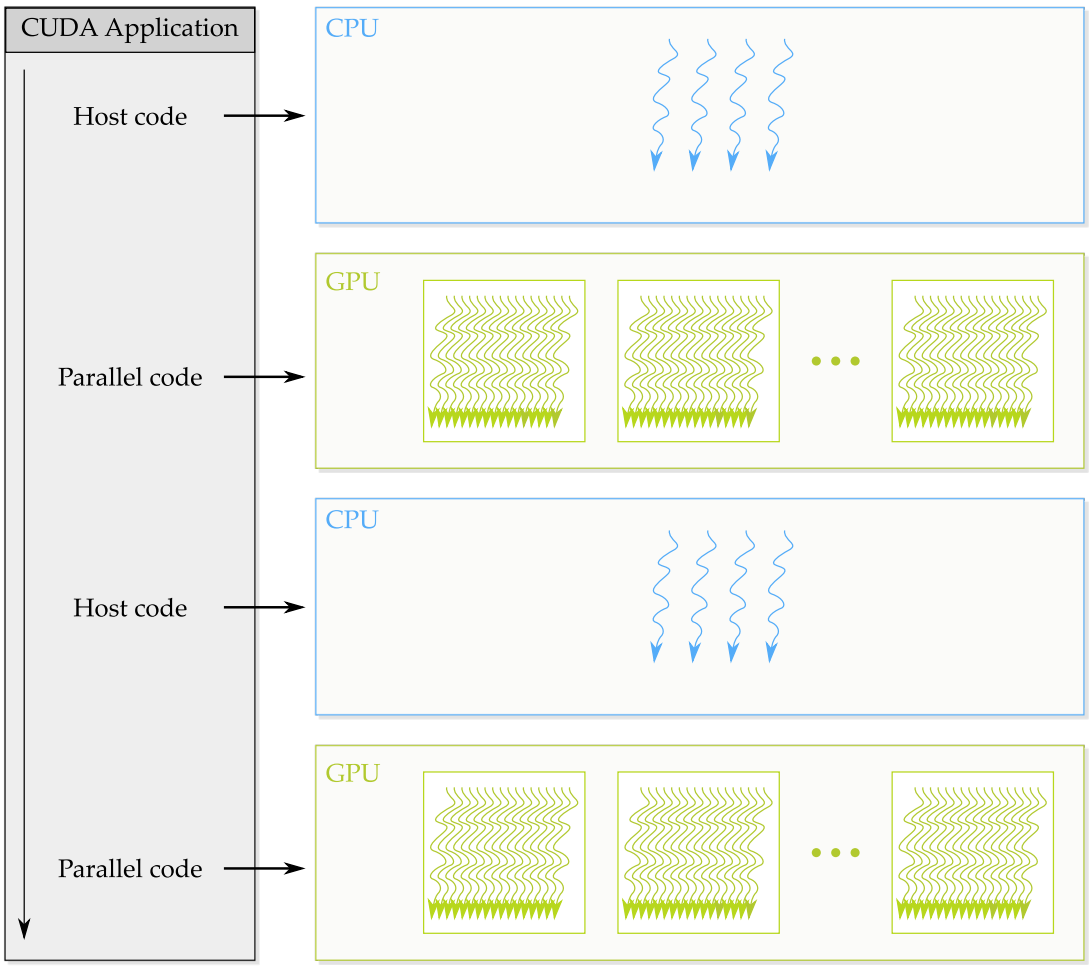

FIGURE 2.8: CUDA application program flow. (Source: [72])

program control is immediately returned to the CPU as the kernel operation is carried in the device. By using this asynchrony, a different operation can be executed on the host in a non-blocking fashion.

CUDA programming model is mainly asynchronous, containing many non-blocking directives that allow overlapping CPU-GPU operations in order to achieve heterogeneous computing.

Typical CUDA program includes device-side code and host-side code together. The NVIDIA C Compiler (nvcc) handles the designation of code sections as well as the coordination between them. The typical program flow of a CUDA application can be seen in figure 2.8.

Standard C function	CUDA function
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

TABLE 2.2: Memory manipulation functions for host and device.

2.2.5. Managing Memory

The CUDA programming model allows the manipulation of device memory as well as host memory. The ability to access device memory is essential since the kernel operations are carried on the device side and only have access to device memory in order to assure low latency memory transactions. The CUDA API exposes a handful of functions to allocate, set, and free memory. These functions are pretty similar to their standard C counterparts and can be seen on table 2.2.

Memory allocation operation can be achieved using `cudaMalloc` function, which acts just like C standard function `calloc` except it allocates memory on the device side. Freeing device memory is possible through `cudaFree` function, just like it's C counterpart `free`. Data transfers between host and device are performed through `cudaMemcpy` function. Again, this function acts similarly to its C counterpart `memcpy`, but it has broader functionality because of the cross-device operations. The `cudaMemcpy` function has four working modes, and these modes can be selected using the following parameters:

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

By giving these parameters to the `cudaMemcpy` function, we can perform transfer operations inside or across devices like host-to-host, host-to-device, device-to-host, and device-to-device.

It should be noted that `cudaMemcpy` function only grants access to the GPU device's global memory. Besides global memory, there are different types of memories present on the GPU device, which will be discussed later.

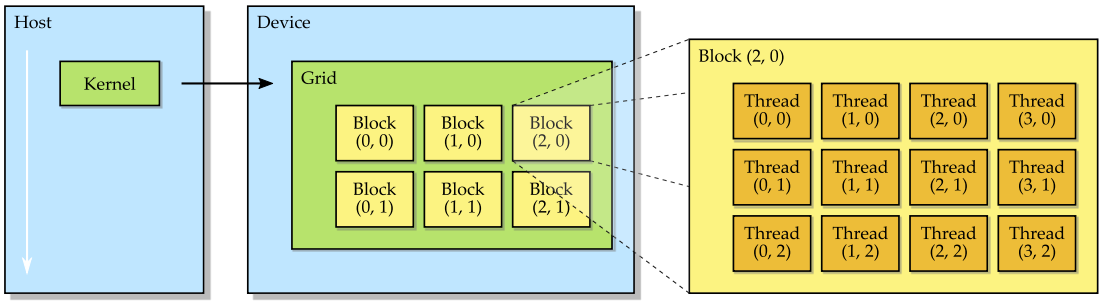

FIGURE 2.9: CUDA Kernel call and thread hierarchy. (Source: [72])

2.2.6. Organizing Threads

When the host invokes kernel functions, CUDA runtime starts the execution on the device and assigns a large number of threads to the kernel operation. The CUDA programming model uses an abstract structure to represent the thread hierarchy, depicted in figure 2.9. This thread hierarchy is made of two levels above threads; these are the blocks containing threads and the grids containing blocks.

At the top of the hierarchy, the threads are represented under a collection named `grid`. When a kernel operation is queued, CUDA runtime assigns the operation to a grid. Threads under the grid share the same global memory space. There are two grid specific values available inside a kernel function. The first one is named `gridDim`, and contains the three-dimensional size of the grid collection, representing the number of blocks in the grid. These coordinates are:

- `gridDim.x`
- `gridDim.y`
- `gridDim.z`

Below one level, the grid contains groups of threads called `blocks`. Each block has a group of threads that cooperate on the given task. The block structure offers additional mechanisms for communication between threads such as shared memory communication and in-block synchronization. Block groups also have predefined size values similar to the grid. The `blockDim` variable holds the three dimensional size value of the block and structured in similar fashion to `gridDim`. Another predefined variable that is provided for the kernel operation is `blockIdx`, which holds the index of block kernel is operating on. The `blockIdx` includes three dimensional index values accessed under following references:

- `blockIdx.x`
- `blockIdx.y`
- `blockIdx.z`

These values indicate the position of the block in the parent grid. At the bottom of the hierarchy, there are threads residing under blocks. The position information of a thread in block is stored in `threadIdx` predefined variable and contains three dimensional position data similar to the `blockIdx`.

To sum up these positional variables, `threadIdx` represents the current threads index in `blockDim` sized block, and `blockIdx` represents the current blocks index in `gridDim` sized grid.

The grid and block groups also represent a barrier between threads. Threads cannot communicate between grids, and only through global memory between blocks. One of the distinctive features of CUDA is that the programming model reveals this two-level thread hierarchy. Dimensions of these units are significant factors of computation performance because of the limited resources physically allocated on-chip like registers and shared memory. The CUDA programming model offers the ability to customize grid and block dimensions to accommodate the application needs better and optimally use system resources.

Another structure contains groups of threads and resides under the block. This structure is called *warp* and will be discussed later under the execution model.

2.2.7. Kernel Execution

Invoking a CUDA kernel similar to calling a function in C for the most part. The CUDA runtime additionally uses an extension to standard C function syntax in order to pass kernel configuration to the device. These configuration values are given in triple-angle-brackets just after kernel function name and before argument list parenthesis:

```
kernel_name <<< kernel_configuration >>> ( argument_list );
```

This syntax has many overloaded versions for different kernel configurations. However, the version including grid and block dimensions is the most familiar and straightforward form. With this version, the syntax for kernel call becomes following:

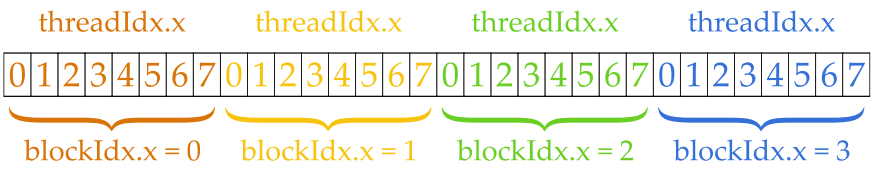

FIGURE 2.10: An example kernel configuration with thread indexes. (Source: [72])

```
kernel_name <<< grid, block >>> ( argument_list );
```

The grid argument in angle brackets sets the grid dimension, which is the number of blocks in a grid, and the block argument sets the block dimension or the number of threads in each block.

For example, if we want to group 32 threads into groups of 8, the kernel call becomes:

```
kernel_name <<< 4, 8 >>> ( argument_list );
```

This configuration launches a grid with 4 blocks with each block containing 8 threads.

In a grid configuration like given in figure 2.10, the data linearly stored in global memory can be accessed collectively using a thread identifier value calculated from threadIdx and blockIdx values:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

This method assigns each thread in block an identification number ordered incrementally. It is a commonly used pattern in CUDA programming to divide the workload among threads with a calculated identifier like this.

2.2.8. CUDA Execution Model

In general, the execution model defines how the instructions are carried under the hood of a computing architecture. It is helpful to gain an understanding of the abstract execution model exposed by CUDA to reason about the thread concurrency.

At the core of GPU architecture, there are units named Streaming Multiprocessors (SM) responsible for the operation. Nvidia GPU devices achieve parallelism via multiple units of these SMs on the system. Each SM unit contains some key components for the operation like:

- CUDA cores
- Warp Scheduler
- Load/Store Units
- Special Function Units (SFU)
- Shared Memory / L1 Cache
- Registers

Streaming Multiprocessors support the parallel execution of lots of threads. Since a GPU device generally contains multiple SM units, thousands of threads can be possibly executed concurrently on a GPU. The SM units work on block-level granularity when it comes to the work distribution. When a kernel function is invoked, thread blocks configured for the kernel are assigned to SM units. One unit can get several blocks assigned for the operation. The operations on SM units are carried through batches of threads called *warps*. Warp represents a set of 32 threads in a block. Threads of a warp execute the same instruction at the same time, but each has its own register state and address counter and applies its instruction on individuals data.

As mentioned earlier, CUDA is built upon their self named Single Instruction Multiple Thread (SIMT) architecture, which is a similar version to Single Instruction Multiple Data (SIMD). Apart from their similarities, the SIMT architecture of CUDA offers more individuality for each thread. In SIMD, each thread of a thread group is executed together in synchrony, while the SIMT architecture makes it possible for threads of the same warp to execute separately. It is possible for threads of the same warp to take different branching paths through control flow even after they start at the same program entry point. The key differences between SIMT and SIMD architectures are the extended individualities of the threads like separate instruction address counters, separate register states, and the ability to have an independent execution path.

While the warps of a block are considered to run in parallel, it is not always the case in reality. According to the warp activity, the SM unit can queue, suspend, and reactivate the threads' operation in a warp, resulting in concurrent operation not guaranteed to be parallel with each other but progressing at a different pace. This is usually the case when a warp is waiting for the peripheral units to complete an instruction, such as memory transactions. While a warp waits for a response from memory, the SM unit starts processing another warp to utilize cores that would idle otherwise.

To lessen the impact of this uncertainty on warp execution order, CUDA offers directives

to create synchronization points between threads on several levels. These directives are implemented at warp level, block-level, and grid level to offer a selection of lock mechanisms with increasing capabilities and performance impacts. The synchronization directives should be avoided as much as possible since they degrade the parallelism. In scenarios where the synchronization is unavoidable, the proper level of synchronization mechanism should be selected to lessen its impact.

Warp groups are the elementary computation units of the CUDA execution model. Warps cannot be divided and placed into different block groups; they are always present as a whole under one thread block. After the threads in a block are divided into warp groups of 32, if there are less than 32 threads left in the last group, these threads are still processed in a warp where any empty threads are left inactive.

2.2.8.1. Warp Divergence

Control logic and decision mechanisms are essential building blocks of the programs. Most programming languages support general control flow concepts such as `if`, `while`, and `for`. Using these directives introduces branching points in machine code to jump different parts of the program according to the state of the system. These branching points pose significant impediments to the parallelization efforts of the computation. Since the next instruction after the branching point cannot be determined before completing this operation, all units of ALU needs to suspend operation until the next instruction is decided.

Modern processors use a myriad of techniques to counteract this problem. Instead of stalling the whole computation hardware, they continue the operation with an assumption at the branching point and flush the carried calculations if the initial guess proves to be wrong later as the branching check completes. There are "branching predictors" designed using different strategies in order to reduce the number of wrong guesses while predicting the branch result before the actual operation resolves. The CPU unit has the luxury of using these sophisticated branching strategies the GPU lacks in order to improve computation performance. Since every thread in a warp has to execute the same instruction, the use of branch predicting is not feasible between threads.

When the threads of a warp have to follow different control flows due to a decision made at the branching point, a conflicting operation order occurs. This problem is called *warp divergence*. If there are divergent threads at a branching point, jumping over a section of conditional code is not possible since some of the threads need to carry those instructions. The GPU hardware handles this situation by disabling some of the threads until the others complete their conditional instructions. For example, if kernel code has an if-else block

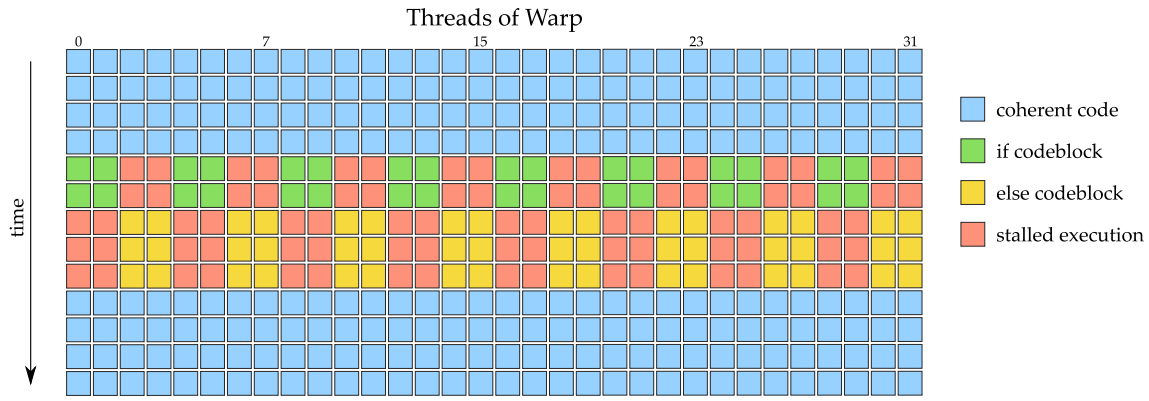

FIGURE 2.11: Warp divergence caused by an if-else clause and resulting thread stall. (Source: [72])

that branches when the thread id is odd or even, this means half of the warp executes one block of code while the second half executes the other. In this situation, the device only allows half of the warp to run while the other half waits, effectively reducing the granularity of parallelism in half. Figure 2.11 demonstrates stalling threads in a scenario like this.

In order to obtain the best performance, care must be taken to avoid warp divergence in code as much as possible. There are strategies to avoid warp divergence in suitable cases, such as designing branching operations in batches of warps.

2.2.8.2. Resource Partitioning

Each warp being executed on an SM unit uses resources such as program counters, registers, and shared memory. These resources are reserved by the SM for the duration of active warps operation. By storing the state context of warps, the SM unit can switch operation between warps with very little overhead if an active warp idles while waiting for a memory transaction.

Due to physical constraints, each SM unit holds a finite amount of these resources. The amount of registers and shared memory available on the SM limits the number of active threads possible since these resources are shared between threads per SM. If a large number of registers are required for the operation in hand, the number of active threads has to be reduced through kernel launch configuration in order to provide enough register memory for each thread. This limitation is also true for shared memory as it is also a limited resource shared between the threads of a block.

For example, we used Nvidia GeForce 1080 Ti in our experiments, and this GPU device has the compute capability 6.1. This compute capability indicates that this device offers 65,536 registers and 48kb of shared memory per block. Since the maximum number of

possible threads per block is 1024, we can safely allocate 64 registers per thread and 48kb of shared memory per block to operate on full potential.

2.2.8.3. Occupancy

While a kernel operation is in progress, SM units constantly queue new warps and swap stalling warps with the active ones to keep the operation going. In an ideal scenario, the kernel should have enough blocks with proper threads numbers to allow SM units to keep the cores occupied. The efficiency of this procedure is expressed by the *occupancy* metric. The occupancy is the ratio of current versus the maximum possible active number of warps. Full occupancy is always desired for optimal operation, but achieving it is not always easy or possible. There are multiple resource constraints to keep in mind while deciding on the block and grid sizes for the kernel. A utility called CUDA Occupancy Calculator is included in the CUDA Toolkit to help on deciding good block and grid configurations for maximum occupancy. This tool lets the user input their device specifications in the form of compute capability version along with thread constrains like required register and shared memory amount and outputs a detailed report of possible different kernel configurations and their estimated occupancy levels.

To reach maximum occupancy, it is often good practice to:

- keep the block size at a multiple of warp size to fill every warp group.
- avoid large blocks, which leads to SM allocating less resources to each thread in the block and possibly resulting in more memory transactions.
- avoid small blocks, which leads to under-utilized SM units since the block limit is reached before the thread limit and less than the maximum number of threads can be executed.

Although the occupancy is a major factor for the performance of kernel, it is not the only contributor. Other factors such as warp divergence and coalescence of memory access have to be studied in order to discover possible problems impacting performance.

2.2.8.4. Synchronization

The parallel computing systems always try to push the boundaries of concurrency. They do so by isolating work units as much as possible so that they can be executed simultaneously, independent of each other. Complex control and scheduling models are designed in favor of computational throughput, processing work units and swapping them with active ones

when they idle to keep the device utilized to its potential. This program flow is different compared to the ordered execution of sequential programs as the finishing order of the work units cannot be determined beforehand. Usually, the parallel programs are developed around this limitation. However, sometimes more control over the thread execution order is desired. Parallel programming languages offer barrier synchronization mechanisms for these situations. In CUDA, there are two levels of synchronization locks available to the developers. These are system-level and block-level synchronization.

The system-level synchronization offers a waiting point between host and device where both systems have to finish their operation before continuing. This is achieved by calling `cudaDeviceSynchronize()` on the host side code. Since most of the CUDA operations are asynchronous, this directive allows the host device to wait for the device operation completion.

The block-level synchronization is a lower scope barrier that allows the synchronization of threads in a block. This barrier is realized by calling `__syncthreads()` function in kernel code. When this function is called, each thread in a block has to wait until all threads of block reach to that point of code. This function is especially useful when the kernel has to transfer data to share memory before an operation. Placing a barrier after shared memory transactions ensure that every thread finishes its memory moving duty, and data is copied entirely before the operation begins.

2.2.9. Memory Model

In the last decades, advances in technology led to better, more powerful processors. As the clock frequencies scaled up and hardware-level parallelism became the norm, the time between executing instructions is drastically reduced. To feed the data to these rapid computations, high-performance, low-latency memory hardware is required. However, providing the ideal high-capacity and low-latency memory unit to the system is not easily attainable or economically feasible.

Instead, the computers in use today employ a memory hierarchy of various speeds and capacities to achieve optimal performance at a fair cost. The memory hierarchies work because of the behavior of data access in applications. The data access is rarely random. The applications usually request the data from a small, localized part of the memory in consecutive fashion. This is called the *principle of locality*, and has two types.

The *temporal locality* represents the locality in time, meaning that a data region is more likely to be requested if a part of it is recently used. The opposite is also true, meaning

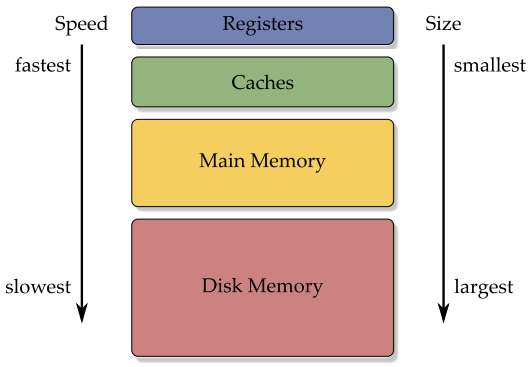

FIGURE 2.12: CUDA memory hierarchy. (Source: [72])

that a region is less likely to be relevant if no part of it is accessed recently.

The *spatial locality* describes a similar relation with space instead of time. If a memory location is recently accessed, the memory around this point is more likely to be requested in the near future. The principle of locality is used to construct caches and mitigate the latency of high-capacity high-latency memories to a degree. Figure 2.12 shows the memory hierarchy of CUDA.

CUDA memory model follows a similar model compared to CPU, but it allows detailed control over exposed memory types to pursue fine-tuning avenues. There are 5 types of programmable memory present in CUDA API:

- Registers
- Global Memory
- Shared Memory
- Constant Memory
- Texture Memory

The topology of device memory can be seen in figure 2.13.

2.2.9.1. Registers

Registers are at the top of the memory hierarchy as the fastest memory type. They are exclusive to their owner thread and can only be accessed by this thread. The lifetime of registers is the same as the kernel owning them. They are used to hold frequently used variables for in thread operations. Also, arguments passed by the kernel invocation can be stored in registers when the data size is appropriate. If the data used in kernel operation

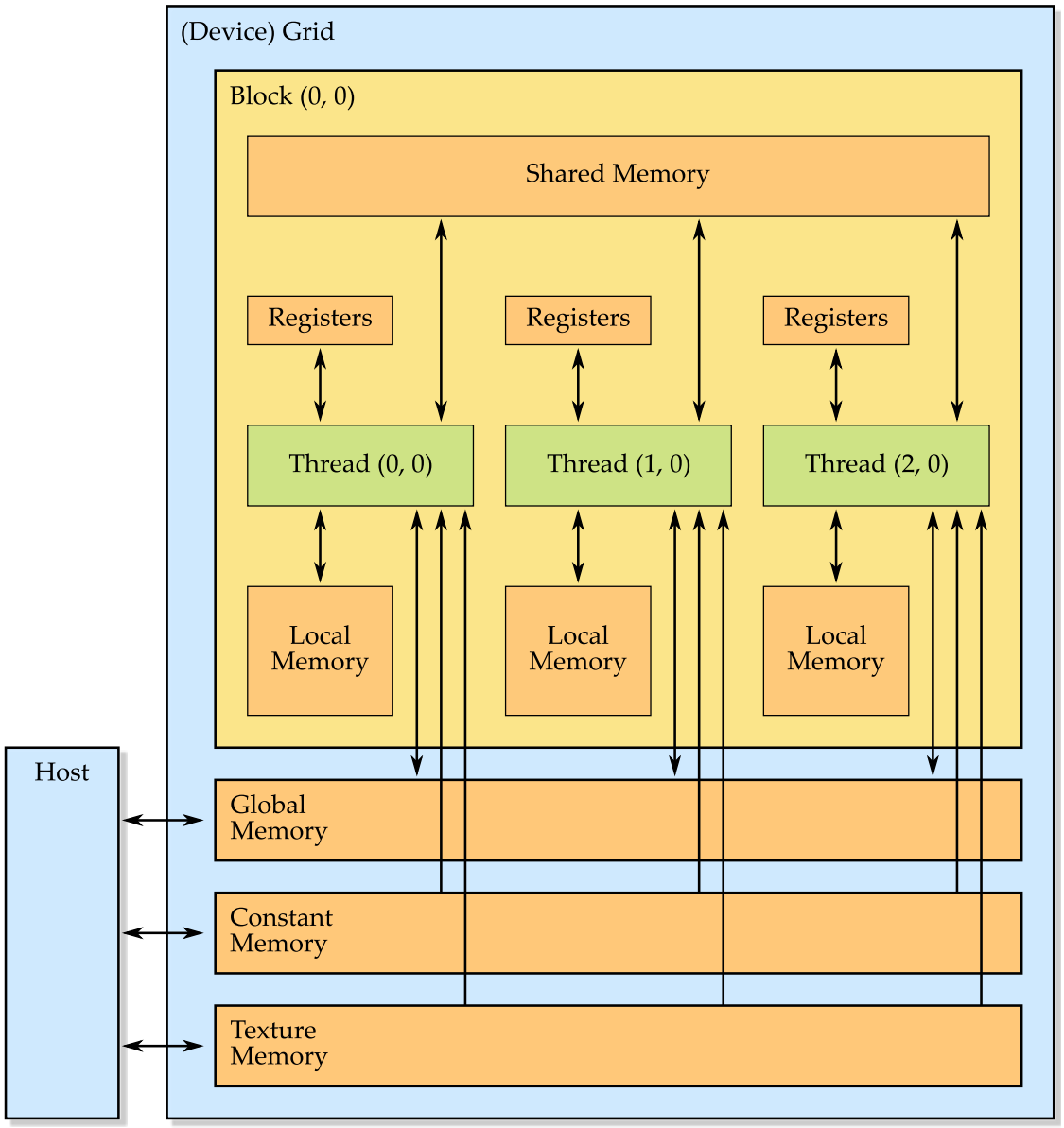

FIGURE 2.13: CUDA device memory diagram. (Source: [72])

becomes too large to fit into register space, some of the variables are spilled into another, non-programmable memory called *local memory*.

Unlike registers, local memory is situated in the lower levels of the memory hierarchy and has higher latency transaction performance. It is important to stay on the register space and avoid local memory as much as possible while doing thread operations. Although they are the fastest memory space on the memory hierarchy, they are also the most sparse. The SM unit contains the register memory and rations it between threads. The amount of available register space per thread varies between different compute capabilities, but it is set as 64 registers per thread for since the compute capability version 5.0.

2.2.9.2. Global Memory

The global memory is the largest memory space on the GPU. It is called global memory because it offers global access and can be reached from both CPU and GPU. The lifetime of global memory spans to the application duration and can persist through multiple kernel operations. Large capacity of the global memory comes at the cost of high-latency operations because it resides on the device memory, away from the SM unit. To mitigate operation delays, access to global memory is provided through batches of aligned memory transactions. These transactions can be 32-bit, 64-bit, or 128-bit in size and have to be aligned on the multiples of transaction size.

2.2.9.3. Shared Memory

Shared memory is below registers but above global memory on memory hierarchy. It offers low latency memory access similar to L1 cache on CPU on top of the customizable operation. Shared memory resides on-chip, close to SM operation and threads, and is managed by the SM unit. It offers limited memory space, which is managed and distributed by the SM between blocks and can be programmatically sized. The initialization of shared memory starts with the kernel, but it shares the lifetime of thread blocks. At the end of each block's operation, shared memory used by that block is released and allocated to a new block by SM.

Shared memory provides a communication mechanism between the threads of a block. Threads of a block can cooperate using this memory, with the aid of synchronization mechanisms. This memory is also placed on the same memory block with L1 cache, and the ratio between L1 and shared memory can be adjusted to fulfill the application's needs. Along with available register partitions, shared memory is one of the limiting factors of parallelism in thread blocks. It is possible to allocate more shared memory per block

Memory	On/Off Chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	Yes	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

TABLE 2.3: Feature summary of CUDA device memory types.

compared to the default value. However, this configuration reduces the number of active blocks that can be processed by SM in parallel.

2.2.9.4. Constant Memory

Constant memory is an on-chip, read-only memory that is accessible from kernels residing in the same compilation unit. It has to be statically declared at the start of the program and stays active for the lifetime of the application. Constant memory has an exclusive cache space reserved and provides low-latency access to its contents.

This memory space is optimized for a specific use case and performs best when the threads of a warp unit need to access the same memory address. Unlike other memory spaces, constant memory has a special mechanism called *broadcasting* that allows the simultaneous delivery of variables to threads when the same value is requested by many of them. It is particularly suited for serving constant values to be used in operations such as formula calculation.

2.2.9.5. Texture Memory

Texture memory is a variation of global memory, accessed through a read-only cache that is present on each SM unit. It offers additional features targeted explicitly for graphics processing. The read-only cache has a built-in hardware interpolation unit and offers fast interpolation results on the subject data. This cache is also optimized for two-dimensional spatial locality and works best when the threads work on 2D data and follow a suitable access pattern. Texture memory is excellent for applications that can leverage its advantages. However, the performance can be worse than global memory on other use cases.

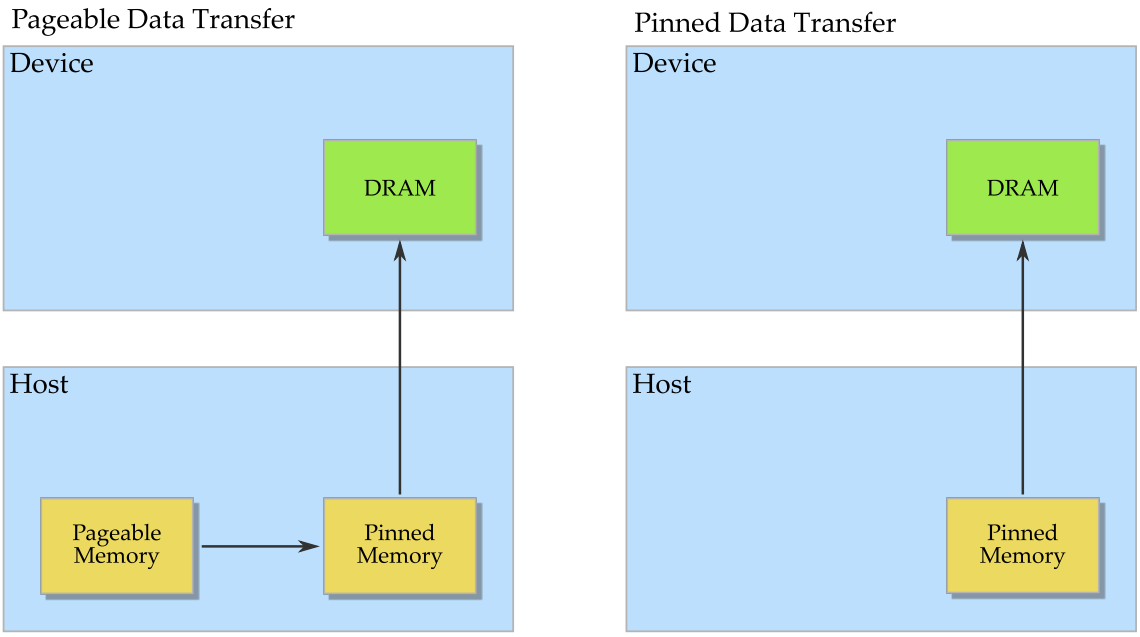

FIGURE 2.14: Allocating pinned memory using CUDA directives. (Source: [72])

2.2.9.6. Pinned Memory

Unlike the other memory types described in this section, pinned memory is not a type of physical memory offered by the GPU. Instead, it is a memory management technique applied to CPU memory by CUDA runtime in order to improve transaction performance. By default, the operating system uses a memory scheme called *paging*. In this scheme, memory contents can be paged in / paged out to secondary storage to open up DRAM memory space that is requested by active applications. Because the GPU has no control over this host-side mechanism, the device cannot safely access the host memory without precautions.

When a memory transfer is initiated to the device, CUDA runtime allocates a page-locked memory on the host memory first, then copies target data from host-side memory to this host-side page-locked memory. Finally, the runtime transfers the data from page-locked memory to the device memory (figure 2.14). Every host to device memory transfer translates into one host-to-host and one host-to-device transaction practically doubling the operations, which is not appealing from a performance perspective.

CUDA offers the ability to allocate page-locked host memory using `cudaMallocHost` function directly. Allocating page-locked memory allows the device to access this host-side memory directly. After allocating pinned memory, an application can fill the data into this host-side memory, and the device can write/read it without extra steps with an improved bandwidth speed.

Care must be taken when allocating large amounts of pinned memory. When this space is reserved in the host memory, the amount of memory available to the system is reduced. The operating system does not control this memory and cannot page it out when the demand rises, potentially resulting in constrained operation.

3. RELATED WORK

The introduction of CUDA in 2007 by Nvidia made the general-purpose computing on graphic processing units accessible to a broader audience. Since then, many studies have been published denoting the improved performance results when GPU programming is applied to many problems old and new. Here, we present some of the recent studies that have been done on string matching using GPU.

In 2009, Ligowski and Rudnicki presented an improved parallel version of the Smith-Waterman algorithm implemented using the CUDA platform on GPU. Their modified version of the algorithm is 3.5 times faster than the previous implementations on GPU, achieving 70% of maximum theoretical hardware performance on their experiment setup. This implementation achieved a respectable 8.67GCUPS operational throughput on an Nvidia 9800 GX2 GPU device.

In 2010, Peng and Chen [76] presented an improved version of the Nrgrep, a well-known command-line utility used in the UNIX system. Their version is called CUgrep, and it uses a GPU-based multi-string matching algorithm based on the BNDM algorithm (2.1.3.3). The underlying algorithm is implemented using the CUDA platform and reported to work 40 times faster than the Nrgrep on an Nvidia Tesla C1060 GPU and Intel Xeon 5110 CPU.

In 2011, Zhou et al. [77] tested their parallel implementation of the Boyer-Moore-Horspool algorithm (2.1.1.5) on GPU and explored the characteristics of it. They have experimented with several optimization avenues like shared memory utilization, access regulation to avoid bank conflict, and the granularity of parallelism at varying degrees. The authors reported speed-up by a factor of 40 for their GPU parallel algorithm compared to serial CPU implementation on an Intel Xeon CPU and an Nvidia GTX275 GPU. They have also reported a relationship between the pattern structure and the algorithm performance.

Hains, Cashero, and Ottenberg [78] proposed an improved parallel version of the Smith-Waterman algorithm implemented using the CUDA platform. Their implementation is called CUDASW++. The authors discovered a bottle-neck at the intra-task part of the algorithm after testing and focused on improving this sub-process. They reported a performance improvement around 25% compared to the unoptimized GPU version of the algorithm, based on their tests conducted on Nvidia Tesla C1060 and Nvidia Tesla C2050 cards.

Tran et al. [79] also presented their implementation of the Aho-Corasick algorithm using CUDA architecture on GPU and studied its memory efficient parallelization. Their implementation focuses on optimizing memory accesses and explores several approaches with different memory utilization strategies. On the experiment system with Nvidia 9500GT GPU and Intel Core2Duo 2.2Ghz CPU, their test results indicate a peak speed-up factor of 15.72 times on GPU compared to the single-core CPU version.

A study from Rasool and Khare published in 2012 [80] compares the performances of the Knuth-Morris-Pratt algorithm (2.1.1.3) implemented on GPU with single-core and multi-core CPU implementations of the same algorithm. The authors of this study used the OpenCL framework to implement both GPU and multi-core CPU versions of the algorithm. Their test system had an AMD Radeon HD 6800 series graphics processing unit instead of a CUDA capable Nvidia brand device and an Intel i3 processor. After testing the serial, the multi-core OpenCL CPU, and the OpenCL GPU versions of the algorithm with different text and pattern configurations, they reported a speed-up by a factor of 9.52 for their unoptimized OpenCL GPU version compared to the serial CPU algorithm.

Pungila and Negru [81] proposed an approach for implementing highly compressed Aho-Corasick and Commentz-Walter automata for intrusion detection and virus scanning on GPU. This approach uses intra-task parallelism to improve concurrency of operation by running regular expressions matching with Aho-Corasick in one thread while searching for regular patterns with Commentz-Walter on another thread. Their hybrid algorithm combined the high-speed advantage of the Aho-Corasick algorithm with low memory demand of the Commentz-Walter algorithm, achieving 38 times higher bandwidth compared to serial CPU implementation while requiring almost 22 times less memory than related implementations.

Yong and Karuppiah [82] implemented their own parallel hash search algorithm on GPU and compared its performance with parallel versions of the Bruteforce (2.1.1.1) and Boyer-Moore-Horspool (2.1.1.5) algorithms. They used two different Nvidia GPU devices; one based on the Fermi architecture on C2075 and the other based on the Kepler architecture K20c. Their novel hash search algorithm runs slower than the Boyer-Moore-Horspool algorithm kernel-wise. However, it is faster by a factor of 8.34 when comparing the overall performance on Kepler architecture. This difference is caused by the additional shifting and lookup tables generated for the Boyer-Moore-Horspool algorithm as these auxiliary data structures increase the burden of data transfers on GPU. The authors also report a significant amount of speed-up, a factor of 23x when the shared memory on GPU is utilized. The overall speed-up is 150x compared to the global memory using version of

the Boyer-Moore-Horspool algorithm.

Tran and Lee [83] implemented a multi-stream version of the Aho-Corasick algorithm (2.1.1.13) on GPU and studied its performance with a large number of patterns. Their aim was to examine the performance impact of HyperQ technology present one the Kepler architecture. To test this feature, the authors used two Nvidia GPU devices; one Nvidia Tesla K20 GPU using Kepler architecture and one Nvidia Geforce GTX 285 for comparison. After testing with up to 20000 patterns, Nvidia Tesla K20 GPU achieved a peak throughput of 585Gbps. This value is 1.45 times higher than the throughput value of the other Nvidia GPU without HyperQ technology.

Adey [84] tested the parallel versions of some partial string matching algorithms running on GPU in order to search in DNA sequences for cancer detection. The advanced multi-pattern algorithms MSMPMA, IKPMPM, EPMSPP, and IAEMA are implemented as both serial CPU and GPU versions. The workstation used in their tests included an Nvidia Tesla C2070 GPU accelerator device along with an Intel core i7 CPU. Their parallel algorithm results indicate a speed-up by a factor of 30 compared to the serial CPU versions of the corresponding algorithms.

Zha and Sahni [85] developed a GPU variation of the Aho-Corasick (2.1.1.13) and Boyer-Moore algorithms (2.1.1.4) in 2013. They used an Nvidia Tesla GT200 GPU and Intel Xeon 2.8GHz quad-core CPU and demonstrated a speed-up between 3.1 and 9.5 times for these algorithms compared to their single-threaded implementations.

In 2013, Xu et al. [86] implemented the MASM algorithm and a BPR variant for multiple pattern matching in GPU. They reported a speed-up by a factor of 28 using an Nvidia GeForce 310M GPU and Intel i3 2.27GHz CPU and comparing their GPU implementation with a single-threaded CPU implementation.

Again in 2013, Bellekens [87] compared their GPU implementation of the Knuth-Morris-Pratt algorithm (2.1.1.3) on an Nvidia Tesla K20M GPU against two Intel Xeon E5-2620 CPUs running the CPU version of the algorithm. The authors conducted a comparison using different string sizes and alphabet sizes with different optimization techniques such as shared memory and loop unrolling. They reported a 29 times speed gain similar to [86] when using a parallel GPU version compared to the serial CPU version of the algorithm.

A 2014 study from Nagaveni [88] used string matching to process DNA sequences in order to detect breast cancer. Their GPU implementation on an Nvidia Tesla C2070 GPU was 30 times more efficient than the serial implementation in an Intel Core i7 CPU.

In 2015, Kouzinopoulos [89] compared several algorithms using an Nvidia GTX 280 GPU and an Intel Xeon 2.4GHz CPU. They implemented Set Backward Oracle Matching (2.1.2.7), Wu-Manber, Set Horspool (2.1.1.5), Aho-Corasick (2.1.1.13), and SOG multiple pattern matching algorithms and reported a speed gain by a factor between 2.5 and 10.9 when using the GPU version of the algorithms compared to CPU versions.

Another work by Sharma [90] presented an implementation of the Karp-Rabin algorithm (2.1.1.8) for Deep Packet Inspection. Their implementation demonstrated a 14 times speed-up compared to the CPU implementation.

Lee, Lin, and Chen [91] proposed a hybrid pattern matching algorithm for deep packet inspection using GPU and CPU, distributing the workload between these two units based on the task type. The authors implemented two string matching algorithms based on the Aho-Corasick algorithm and the Knuth-Morris-Pratt algorithm and compared their performance and power efficiency with CPU and GPU versions of the base algorithms. Their test system included an Intel Core i7 3770 CPU along with an Nvidia GeForce GTX680 GPU. The Aho-Corasick variant of hybrid pattern matching algorithm outperformed the CPU version by 3.4 times and the GPU version by 2.7 times while achieving higher efficiency than other tested algorithms.

For the Karp-Rabin algorithm, three GPU based parallel approaches have been proposed by Ashkiani in 2016 [14]. These three approaches are *cooperative*, *divide-and-conquer*, and a *hybrid* combination of both. This study reported that the cooperative method was most effective for pattern lengths longer than 8000 characters, and the divide-and-conquer approach deemed superior for shorter pattern lengths. The speed improvement by a factor of 4.81 was reported for their implementation on an Nvidia Tesla K40c GPU. They improved the performance of Karp-Rabin with the divide-and-conquer approach for large models by using a new parallel two-stage method that scanned text for a smaller subset of the pattern and then validated all potential matches in parallel. In the hybrid method, they split the text into substrings, just like the divide-and-conquer approach, then assigned each substring to a group of processors and processed in a cooperative manner. Karp-Rabin algorithm is an excellent candidate to test these approaches since the algorithm logic can be adapted into cooperative parallel processing, which is not possible to implement for the majority of string matching algorithms.

In recent research, a tribrid parallel method has been proposed for bit-parallel algorithms such as Shift-Or and Wu-Manber algorithms [92]. The main idea of this method is to use inclusive-scan, which enables bit-parallel algorithms to work efficiently on the GPU. Inclusive-scan not only eliminates duplicate searches between threads but also performs

memory access in a GPU friendly pattern that maximizes memory read/write efficiency.

A review by Ramos-Frias et al. [93] lists several string matching algorithms implemented on GPU. These algorithms are selected to portray the current state of different string matching techniques. However, their selection of six algorithms is not an adequate representation of the whole parallel string matching field.

String matching is not an unexplored problem on the parallel GPU computing field. There are plenty of studies trying to transfer the existing string matching algorithms to GPGPU medium through the implementation of efficient parallel versions as well as proposing novel algorithms. The works mentioned above indicate the capabilities of GPU devices as the main processing unit for string matching problem and demonstrate the feasibility of utilizing these devices on a variety of fields such as network intrusion detection, malicious code scan, deep packet inspection, DNA substring searching for cancer detection, hash table construction and information retrieval. Most of these studies focus on a singular algorithm from a small selection of popular string matching algorithms, such as the Boyer-Moore algorithm (2.1.1.4), Boyer-Moore-Horspool algorithm (2.1.1.5), Knuth-Morris-Pratt algorithm (2.1.1.3), Backwards Nondeterministic DAWG Matching algorithm (2.1.3.3), and Aho-Corasick algorithm (2.1.1.13). However, string matching literature contains a plethora of different solutions to the problem, but other than a selected few, most of the algorithms are challenging to approach due to lack of code samples demonstrating principal working mechanisms.

For serial computing on the CPU-side, there are the works of Hume and Sunday [33], and its spiritual predecessor SMART library from Faro and Lecroq [15]. These code libraries allow the researchers to compare almost every string matching algorithm in the literature easily and test their own algorithm implementations against the others. Authors of the SMART library were also able to conduct the most extensive comparison of string matching algorithms in the literature because of the toolkit they developed. There is no similar work for parallel GPU algorithms that could be considered as the parallel equivalent of these tools even though the parallel GPU implementations of the string matching algorithms are actively studied.

We believe a similar toolkit for the parallel GPU operation would be beneficial for the researchers willing to study the parallel string matching algorithms, benchmark their approaches against the established algorithms of the field, or test already available algorithms on their problem to find out the best performing candidates easily.

To address the absence of such codebase for parallel GPU operation, we present CUSMART, a parallel string matching algorithms research tool developed using the CUDA platform as our main contribution. Also, we are sharing the results of our comparative study conducted using the developed CUSMART library. To best of our knowledge, this is the most extensive comparative study of parallel string matching algorithms on GPU, compiled by performing tests on 85 string matching algorithms. The closest study we have found is a survey from Ramos-Frias et al. [93], only covering 6 algorithms.

4. CUSMART - CUDA ENHANCED STRING MATCHING RESEARCH TOOL

The string matching problem is one of the most studied problems in the computer science field. Although the problem is essentially finding a pattern in a string, in practice, there are many different parameters affecting the operation. Text size, pattern size, alphabet size, distribution of the characters over text or pattern, character frequency of the given alphabet, limited system resources like memory, or processing power are some of the most significant factors that shape the algorithms. The multivariate nature of the problem makes it very difficult, if not impossible, to find a solution that works effectively in every possible case.

The topic amassed a large number of novel methods over the decades. Since the 1970s, over 120 algorithms have been proposed on the exact string matching subcategory of the string matching alone. The published algorithms often emerge either as a new approach resulting from an in-depth analysis of the problem under a selection of conditions or as a result of applied improvements over an existing algorithm after a thorough examination. As a natural outcome, these algorithms have advantages and disadvantages when evaluated over different scenarios as they are usually tailored for a specific use case. For example, some algorithms are designed to perform efficient searches on a small alphabet such as DNA, while others try to minimize memory usage of the algorithm or try to keep it constant. There are also algorithms aimed at searching for multiple patterns at the same time or performing the search operation at a constant time to be as predictable as possible timing-wise. The comparative studies of these algorithms are present in the literature, although not many.

Hume and Sunday presented one of these studies in 1991 [33]. They published a framework developed in C language for string matching algorithms, along with comparative results of various test cases. This framework contains the implementations of 37 string matching algorithms, and helper functions to facilitate the construction of new algorithms. Their framework was one of the first, large-scale string matching tools and used by many researchers in the string matching field. However, over time, some shortcomings of this library became apparent and resulted in new projects to emerge.

The String Matching Research Tool (also abbreviated as SMART) is a similar project released in 2010 and aimed at addressing the limitations of the earlier work. This library

is the most extensive string matching tool available today. With a collection of over 120 algorithms implemented and documented, it is an invaluable tool when studying string matching algorithms old and new. Albeit much more extensive, still like its predecessors, the SMART library is designed to work on single-core CPUs and contains serial versions of the algorithms designed to be executed on a single thread. This arrangement allows the algorithm code to be concise and easy to read; however, it makes multi-threaded performance comparisons not possible since there are no parallel versions of the algorithms in the library. The parallel implementations of these algorithms are not trivial as there is no clear-cut way to realize parallelism without altering the core parts of the procedure.

There are various hardware architectures available for parallel computing, graphic processing units (GPU) being one of them. The usage of graphic processing units for parallel computation operations is called the General Purpose computing on Graphics Processing Units (GPGPU). The GPGPU is a hot topic in the high-performance computing field. Thanks to the effort put into the GPU hardware and software in the last decades, the devices are cost-effective for many applications from a performance standpoint, compared to multi-core CPUs, and the entry barrier for programming is low because of the extensive documentation and simple high-level APIs. After these advantages taken into account, we decided to use the GPU as our platform of choice for parallel processing.

There are string matching algorithms implemented using GPUs in the literature, and some comparative studies with small scopes, examining only a few algorithms. Nevertheless, to the best of our knowledge, there is no codebase similar to the SMART tool for the parallel computing on the GPUs and no comparative study implementing and testing dozens of parallel string matching algorithms.

In this work, we present the CUDA enhanced String Matching Research Tool (CUSMART), which is a parallel string matching library developed using Nvidia's CUDA architecture. Our motivation was to compile a codebase of parallel string matching algorithms to aid studying, testing, and developing string matching algorithms using GPUs. Similar to the SMART library, this project aims to gather string matching algorithms in the literature but contains parallelized versions of these algorithms implemented on the CUDA platform.

As of today, there are 85 string matching algorithms in our library with parallel implementations, and we plan to increase this number as we continue to improve it. The short descriptions of the implemented algorithms, along with the references to their originating published work, can be found in section 2.1.

The following sections cover the implementation details of our project.

4.1. Main Structure of the Library

Although the SMART library is a useful resource for our project in terms of the algorithm codes it provides, it is necessary to make changes in most of the codebase because the original library was designed with different priorities in mind. In the serial library, the source code of each algorithm was placed into a separate file, and the shared logic of the program, which is necessary for the proper start-up and the processing of the initial arguments, was created in a separate file and then included in each algorithm file with preprocessor commands. This shared file also contains the main entry point function for the programs. In this case, after the compilation process, a separate executable file is created for each algorithm, independent of the other code.

As a result of this design decision, each algorithm executable allocates memory at the beginning of the program, reads and processes the text data, and releases this memory space at the end of the program life cycle. This approach causes the text file to be read back repeatedly during batch algorithm tests. When we examine the article that the test results of the SMART library were published, we see the authors were working on files of a few megabytes, which can be considered small compared to our test corpora. In the original project, it may not have been a problem to read the files over and over again when working on files of this scale. However, for example, when a test is run with 85 different algorithms on a 5 GB text file, as we would use in our tests, a repetitive read operation of 400 GB data occurs, which is not desirable. The CUSMART library is intended to compile into a single executable file to avoid repetitive read operations and to ensure compatibility with the rest of the code.

After compiling the algorithms to run under a single executable file, there is a need to change the allocation method of the memory areas used by the algorithm functions. In the serial library, the string subject to search and the auxiliary array structures used by the algorithm are stored in fixed-size static memory blocks. These fixed memory blocks are intentionally chosen larger than necessary in order to avoid insufficient memory problems when a large file is tested. In the original scenario where the executable files are compiled separately, the allocation of static memory larger than necessary might be acceptable. However, when we compile the files together, each source file will allocate a unique memory space and hold this space through the program's lifetime because it is static. For example, with this approach of the serial library, the memory requirement reaches approximately 33,000 times the space normally occupied by the string for a 10-character string search. Since this large memory requirement can create problems for systems with limited resources, the codebase we use in our project is designed to allocate memory space dynamically. This approach gives us the ability to manage memory more efficiently, using as much memory space as we need and releasing this space after the operation is finished.

One side effect of converting static memory space allocation to dynamic was the mandatory change for the arrays of 2 or higher dimensions. These multidimensional arrays can be made dynamic and multidimensional, but it adds unnecessary complexity, requiring to use a large number of memory allocation commands and changing the way these arrays are handled as the function parameters. At this point, in order to avoid complexity and loss of performance, multidimensional arrays are reduced to one dimension, and access logic is appropriately arranged. During these changes, some minor errors were detected in the reference code, and these errors were corrected. Most of these errors are caused by incorrect access to boundary regions of arrays with incorrect boundary conditions. These errors, which are difficult to pinpoint as a result of the static memory allocation of the original library, occurred when switching to the dynamic memory allocation.

Another change we made in the SMART library-based code we added to our project was to modify the outputs of the algorithm functions. The original code keeps track of how many times a string is matched in a single variable, instead of keeping places where matching has occurred within the string.

A requirement that string matching functions used in many applications must provide is the information of the matched positions of pattern in the text string. Therefore, it is necessary to record this positional data for the algorithms to serve a practical purpose. Once this data is available, the match count can be calculated from this data, but not vice versa. The match position data provided by the algorithms are stored in an array, which occupies space in memory equal to the memory occupied by the text string. If a match starting from the n-th character of the text string is detected, the n-th value of the position array is set from 0 to 1. The total match count is then computed by performing the reduction operation over the positional data array. The timing of this operation was also kept in our tests, and care was taken to not affect the algorithm timing measurements.

4.1.1. Applied parallelization techniques

Task parallelism can be classified into two categories [94]. The *inter-task parallelism* refers to the parallelism between tasks where every task is assigned to a single thread, and tasks are performed in parallel. The *intra-task parallelism* is the other type of parallelism where a single task is assigned to a block of threads. Threads of this block cooperate and perform parts of the task in parallel. In general, string matching algorithms contain two main tasks, the preprocessing step, and the searching step. These two tasks are not suitable to be performed in parallel since the search step is dependent on the preprocessing step. While the searching phase of some string matching algorithms can be divided into sub-tasks, most of them cannot. Also, the CUDA uses Single Instruction Multiple Data

(SIMD) architecture, which is more suitable for performing a single task in parallel on a large amount of data, favoring the intra-task method. Our parallel implementation methods follow intra-task parallelism practices.

4.1.2. Granularity of Parallelism

The algorithms implemented in our library are designed to perform the string matching operation in parallel over smaller pieces of text. Each thread is assigned to a part of the text string and tries to match the pattern string using the corresponding string matching algorithm. The length of the substring assigned to each thread is defined by a variable named *stride length*. This variable is also referred as the *granularity factor* in the literature [95]. At runtime, each consecutive thread starts its operation a step away from the previous threads starting point. This step distance is adjusted by the stride length variable.

The stride length is an important factor impacting the concurrency of the operation. Lower stride lengths allow for higher numbers of threads to work in parallel, but thread creation overhead starts to become significant as the individual work gets small. Higher stride lengths distribute bigger workpieces to threads and lower the impact of parallel work overhead but constraints the number of threads that are deployable and degrade the concurrency. There is a balance point to be found for the stride length. We have tested the impact of different stride length values over algorithm performance, and our results can be found in the chapter 5.3.

4.1.3. Streaming Operation

The efficient utilization of all processors is important in heterogeneous computing. To achieve increased throughput, the CUDA architecture queues commands to execute in *streams*. Streams are objects holding sequences of commands that execute in order. The CUDA architecture adds another layer of parallelism via streams. When two operations are put into different streams, CUDA can process these operations in parallel, since the operations are overlappable, meaning they allow asynchrony and do not block the SM unit's operation. The usual use case for the streams is overlapping the kernel execution with memory transfers.

Before the search operation starts, the text and pattern string has to be transferred to the GPU memory. This operation takes a significant amount of the total operation time, longer than the actual search operation in many cases. When the memory transfer operation is in progress, the processors stay idle and wait until the transfer operation is carried. This

unwanted idle time cannot be avoided entirely, but with the clever use of streams, it can be reduced.

Instead of transferring the full text string before search operation, the transfer can be divided into smaller parts, and the search operation can be cascaded between these part transfers. This way, the first part of the text string transfer is queued in a stream, and the kernel can be called to process the transferred part. When the second part of the text string is queued to another stream, the processing of the first part is already begun in the first stream. This type of concurrency is possible because the streaming units handling kernel executions and memory transactions in CUDA are different and independent of each other. Using this property of CUDA architecture, asynchronous operations can be "hidden" behind the others.

In our case, since the string matching kernel executions are usually faster and take less time than memory transactions, they can be hidden behind memory operations to reduce idle time and improve operation speed. There are streaming versions of some algorithms present in our library. Also, it is easy to develop and test streaming versions of other string matching algorithms using our example implementations. We have conducted tests on streaming using different configurations and presented the results in the chapter 5.3.9.

4.1.4. Shared and Constant Memory

CUDA architecture has several memory spaces other than global memory, like constant memory and shared memory.

Constant memory is a small, read-only memory space with caching and broadcasting capabilities. This memory is optimized for fetching a small set of data, like constant variables and static model parameters. Since the text string is usually too large to fit into this memory space, we used constant memory to store the pattern string on preference. The CUSMART library includes an option to use constant memory for pattern storage instead of global memory. The switch operation is handled by the helper functions, and there is no need to modify the algorithm code to enable constant memory use.

Shared memory is another memory space designed to allow fast memory transactions between the threads of the kernel block. It provides fetch-rates hundreds of times better than the global memory but is limited to in-block operation and has low memory capacity. Unlike constant memory, shared memory usage requires additional steps. In order to utilize shared memory, the threads have to fetch the data from the global memory and store it in the shared memory before the operation starts. This fetch phase is placed at

the start of the kernel function. There are many algorithms in our library designed to use shared memory. These algorithms have "shared" word in their title and contain a few tweaks for shared memory operation compared to the original algorithm.

More information about memory types can be found in the chapter 2.2, and our test results regarding different memory types of the CUDA architecture are shared in the chapter 5.3.

4.1.5. Pinned Memory

Pinned memory is another type of memory used by the CUDA programs, and explained in section 2.2.9.6. Unlike constant, shared, or texture memory, this is more of a memory access technique for the CPU side rather than a specialized memory unit. The CUSMART library has the ability to use pinned memory via a command-line option. By enabling the pinned memory option, the library reserves space on the host RAM exclusively used by the CUDA application. The usage of pinned memory is always advantageous performance-wise, and it is recommended to use it if there are no memory constraints. Our test results regarding pinned memory usage can be found in the chapter 5.3.

4.1.6. Occupancy

Occupancy is a metric used to describe how efficiently the CUDA device is utilized for the operation. Achieving the best occupancy is a multivariable problem. It cannot be accomplished without some prior knowledge about the application, meaning an appropriate run configuration has to be determined for every use case. The CUSMART library comes with some helper functions to calculate the proper block and grid dimensions for optimum operation. The algorithms used in our tests include these calculations to achieve the best occupancy possible.

4.1.7. Algorithm Testing

4.1.7.1. Preprocessing Steps

The majority of string matching algorithms implemented in our library requires some preparation before the search operation is carried. The preparations include processing of the pattern to acquire knowledge about the structure of pattern and improve search operation. This operation prior to the search is called the *preprocessing step*.

The preprocessing step generally takes a fraction of the time required for the search operation. Also, most of the algorithms have preprocessing steps with operation-wise precedence restrictions to construct the support variables. During our initial tests, this

part of the operation proved to be not feasible on the GPU side, the transaction takes longer than the operation, and the concurrency is very low. For these reasons, we have decided to run the preprocessing code on the CPU side and transfer the output to the GPU device memory. The preprocessing functions are called in the wrapper function of the selected algorithm and timed separately using the CPU side timing functions.

4.1.7.2. Timing of the Search Operations

Measuring the duration of different phases of a string matching algorithm running on CUDA is not as easy as counting the CPU clock cycles. The CUDA capable graphics processing units run in an asynchronous manner with the CPU of the system, meaning the CPU clock counting does not translate into GPU runtime.

There are event-based recording functions that run on the GPU side for timing, provided by the CUDA API, and these functions are used to time the GPU side operations of our library. Using these capabilities, The library records different steps of the operation individually. These steps are the memory transaction from host (CPU) to device (GPU), the main search kernel operation, and the reduction operation.

Aside from these timers, the preprocessing steps of the algorithms are timed by the appropriate timing functions on the CPU side, because this phase is executed on the CPU. The total duration, along with the step times described above are reported as the final result of the application. The memory allocation operations are not included in the time measurements. We believe these system-specific calls do not represent the examined algorithm's capabilities and should not be included in the final benchmark.

4.1.7.3. Fair Comparison

Since the CUSMART library is designed to compile into a single executable, extra care must be taken so that the algorithm functions do not influence each other's operation. There are a few factors that are hard to regulate, like memory caching and GPU warmup time. In order to reduce the effect of caching on benchmarks, we are running a dummy memory read before operation without timing it. This approach provides an ideal memory transfer state for every algorithm by eliminating the cache irregularity for the first algorithm. The device warmup time is another problematic side-effect that becomes apparent as an increased runtime on the first kernel timer. To avoid this phenomenon from affecting the GPU benchmarks, a simple warmup kernel call is added before every kernel execution. This call is not included in the kernel timing and ensures the following kernels are executed and timed without delays. As another precaution, there is an average time option included

in the library to smooth out variances caused by the system occupancy. By enabling this option and specifying the number of times the search operation has to be repeated, the library can be made to run the selected search algorithm multiple times and report the average runtime value of these runs.

4.1.7.4. Computation of the Total Match Count

Our CUSMART library follows a different approach for counting match cases compared to the SMART library. The SMART library uses preprocessor macros that are tied to a single variable for storing the match count. This counter variable is increased by the running algorithm whenever there is a match case during the search operation. A single storage variable approach does not cause any problems when the memory accesses are performed sequentially, like the single-threaded code of the SMART. The single variable approach even has the advantage of drastically reduced memory requirement at the expense of lost information about match positions, which is not mandatory for every use scenario.

However, single variable storage becomes impractical when the program is executed in a multi-threaded fashion, with the race conditions and read-write conflicts emerging as the result of this conversation. To avoid these problems and the slowdown caused by lock mechanisms while trying to circumvent the single variable access issue, the CUSMART library uses an array allocated on the global memory of the GPU in order to record match positions. This array holds char sized variables to reduce memory requirements but still holds the same amount of space as the text to be searched. This approach significantly impacts the capacity of operation, practically reducing it by half, because to search n bytes of text data, we need to allocate an additional n bytes of match array, increasing the required memory space to twice of the input.

This is under the assumption that the other auxiliary data structures like the pattern data and preprocessing step products can be ignored since their space requirements are negligible. After the searching step, this match array is used the acquire total match count by performing a counting operation over all array cells. This counting operation can be performed efficiently using a technique called *reduction*.

The reduction technique sums the target array values systematically, processing, and halving the operation area in each stage. Luckily, the reduction operation is a well-studied technique and lends itself to parallel operation easily. There are highly efficient reduction functions implemented in parallel environments. The CUSMART library uses such an implementation and runs the reduction operation on the GPU side in order to reduce the amount of required memory transfer back from device to host. Our efficient reduction

operation is around 300 times faster than the CPU implementation, producing match count results much faster. The whole reduction and final result transfer operation is quicker than the array transfer back to host, let alone additional reduction operation on the CPU side. It is recommended to use GPU side reduction, and we have performed our tests this way. Still, we have included the ability to select between GPU and CPU reduction via a runtime parameter if these test conditions are desired.

4.1.8. Introducing New Algorithms to the Library

The CUSMART library is designed to have a streamlined structure, making it easy to implement new algorithms and perform comparisons with the existing ones. The GPU side algorithms are divided into two sections, the kernel function, and the wrapper function.

The wrapper function contains the preparation steps like memory allocation, memory transfer, preprocessing steps, reduction operation, and the timing calls required prior to the searching phase. There are helper functions to provide boilerplate code for repetitive parts of the wrapper function.

The kernel function contains the main logic of the search algorithm. For some special cases like the versions using shared memory, additional memory transaction code is placed in this kernel function before search logic. Lastly, all implemented algorithm wrappers are listed in designated header files as external references to access inside the application. Newly added source files also have to be included in the CMake configuration file in order to ensure they are compiled correctly into the library.

Our preferred method of compilation is using the CMake tool. CMake provides a higher level of abstraction for the code compilation and allows cross-platform compilation using a central configuration file. Using this tool, our library can be compiled on different platforms supporting Nvidia GPU devices and successfully tested on Windows and Linux.

5. EXPERIMENTS AND RESULTS

In this chapter, we are going to list the string matching tests prepared using our CUSMART library. The string matching algorithms are usually optimized around designated use limitations. As a result, these algorithms work efficiently under certain conditions while behaving poorly under others. We aimed to test the algorithms included in the library under different scenarios to get a broader picture of their behavior. The evaluated test scenarios are designed to accentuate the impact of different parameters under typical use cases. These test scenarios include the best-case and the worst-case scenarios of the algorithms, as well as some synthesized patterns and text structures. In addition, the effects of the hardware optimizations used while implementing concurrency were tested on the performance of the algorithms. Different datasets such as novel texts in different languages, text consisting of random characters, Wikipedia dump files, microorganism DNAs, and synthetic datasets were used for the tests. These test configurations are given in more detail in the following sections.

In order to reduce the impact of the system background processes on the timings, the tests were performed at least ten times, and the average times were reported when the deviations between runs were negligible.

5.1. Testing Equipment

5.1.1. Workstation Specifications

For the tests, the workstation which was purchased within the scope of the project was used. This computer is an HP Z8 G4 Workstation using two Intel Xeon Silver 4114 2.20GHz 9.6GTs 13.75MB Cache 2400MHz 10Core as CPU, 128GB (4x32GB) DDR4 2666MHz ECC RAM, 256GB SATA SSD harddisk. Ubuntu 18.04 is chosen as the operating system. The NVIDIA GeForce GTX 1080 Ti is present on the system as the main graphics card for GPGPU operation. This card has 11GB of memory with 11Gbps bandwidth and 3584 CUDA cores running at 1.5 GHz. The CUDA runtime was compatible with version 18.04, and the toolkit version 8.0 was installed.

5.1.2. Mobile System Specifications

NVIDIA has developed a series of hardware mapping devices for embedded systems, which are very small in terms of footprint and consume little energy in terms of energy consumption but have high processing power. These embedded systems, called the Jetson

series, are designed as a System-On-Chip on a common system of credit card-sized CPUs and GPUs. These chips are also sold on a modular form with extra connectors on it, and they are operation ready for application development.

NVIDIA Jetson embedded GPGPU solution architectures were also investigated for string matching, and performance comparisons were performed. The result of these tests can be found in the chapter 5.3.

Nvidia Jetson TX2 computing device was used for these mobile environment tests. The Jetson TX2 device has 256 Cuda cores, HMP Dual Denver 2/2 MB L2 and Quad ARM® A57 / 2 MB L2, 8GB Ram, and 32GB SATA disk space available.

5.2. Test Cases

5.2.1. Structured Text Datasets

The first set of data is generated from the Wikipedia content dump file containing daily text in English. The generated sample text data sizes are selected as 1M, 10MB, 100MB, and 1GB to analyze the impact of text size over the performance of the algorithms. In these tests, four different patterns with the lengths of 3, 10, 50, and 200 characters were chosen, and each pattern was searched in four different text strings. For each file, serial implementations of all algorithms were run on CPU, and parallel implementations were run on GPU, and their average speed-ups were calculated. Different memory sections like shared, constant, and pinned memory are also tested using these text strings, and the performance impact is calculated based on the parallel GPU versions of the algorithms. The results were generated by averaging the output of 10 trials.

5.2.2. Genome Datasets

Since the Genomics study is one of the largest fields that sees the extensive use of string matching, we have included DNA tests to compare the algorithm behavior over the DNA sequencing problem. The DNA sequence of Drosophila melanogaster (commonly known as fruit fly) is used as the source string in some of our test cases [96]. The sequence is duplicated enough to get a 100MB file for the search operation. Different patterns with the lengths of 3, 10, 50, and 200 are used in the tests. These tests examine the performance of algorithms over structured and small alphabets with varying pattern lengths.

5.2.3. Syntetic Repeating Datasets

Our synthetic text data contains two repeated character test files. The first test data contains only a single character repeated for the length of the whole file, which is 100MB in our case. Other test data is constructed in a similar fashion and only contains a repeated string "cusmart" filling a 100MB file. These files are used along with the matching patterns with one mismatch character. The mismatch character is positioned at the beginning in one scenario and at the end of the string in the other scenario to simulate best-case/worst-case scenarios of the algorithms.

5.2.4. Syntetic Randomized Datasets

The performance of the algorithms and the variations in the performance for different pattern lengths and alphabet sizes were examined. The alphabet sizes to be tested were determined as 2, 8, 16, 64, 128 characters, and the search operations were performed on the text files consisting of randomly generated characters from each alphabet. The size of the text files used during these tests was chosen as 100 Mb, which is determined from the earlier tests. The search tests were carried out with four different patterns of different lengths. The lengths of these patterns were chosen as 3,10, 50, and 200 characters and were randomly generated from the characters of the corresponding alphabet. Thus, 20 different test scenarios were prepared using four different length patterns and five different alphabets. In order to reduce variance from the test system, each test scenario was run ten times, and the average of the results was reported. In addition, all the tests were repeated on the re-created random text files, and the averages of the results were taken, reducing the potential effects of inadvertently structured text sequences in the random generation.

5.3. Results

5.3.1. Optimal Search String Length

To determine if there is a preferable file size for performing the search operation, serial and parallel versions of the algorithms are tested using four different patterns on four different files, making 16 tests in total. The patterns were 3, 10, 50, and 200 characters long, and chosen from the words present in the text string to guarantee at least one match. The text strings are taken from the Wikipedia English raw data dump file and constructed from the first 1 MB, 10 MB, 100 MB, and 1 GB of the file. The runtime of each parallel algorithm is divided by their corresponding serial algorithm to obtain the speed-up factor. Then, the averages of these values are calculated for each data size and pattern length. Our results are presented in the table 5.1 and the figure 5.1.

Pattern Length	File Size			
	1 MB	10 MB	100 MB	1 GB
m = 3	10.06	22.08	25.31	28.11
m = 10	7.78	19.79	23.30	20.51
m = 50	6.47	18.04	22.16	19.95
m = 200	2.72	12.99	18.09	16.45

TABLE 5.1: The average speed-ups of parallel algorithms with regards to various pattern lengths over various structured text lengths.

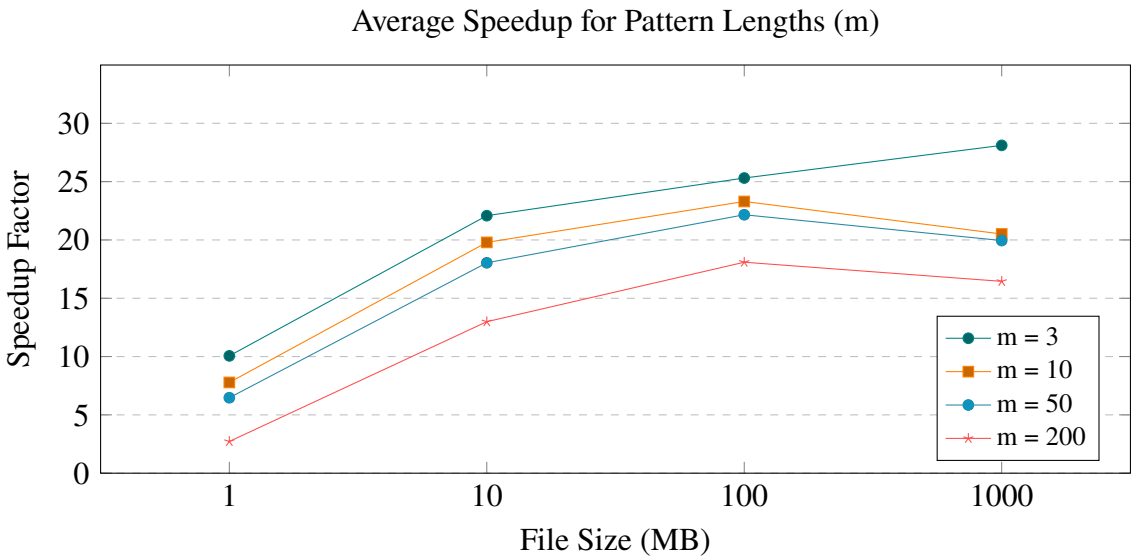

FIGURE 5.1: The average speed-up of parallel algorithms with regards to various pattern lengths over various text lengths.

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
bndmq4	10,9312	sfbom	12,3764	bom	11,1426	fs	11,1482
kr	12,8345	raita	12,5348	rcol	11,3989	lbndm	11,1676
raita	14,4963	sabp	12,8413	bxs	11,4467	sbndm	11,1868
bf	14,7148	fs	12,9553	tndm	11,4561	fsbndm	11,1935
br	14,7840	svm	13,0129	lbndm	11,5302	sbndm_bmh	11,1976

TABLE 5.2: Top 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

Besides the apparent monotonic increase for very short pattern lengths ($m = 3$), overall speed-up factors increase until 100 MB file size mark and peak at this point before starting to decrease. This value is most likely affected by the GPU unit's operation throughput, and the operational saturation is achieved around this point before the scheduler starts to queue up the operations due to high demand.

Even though the speed-up factor of the algorithms never went below 1, meaning there is always a net speed gain when using parallel string matching algorithms over serial ones, some configurations resulted in better performance over others. Determination of the peak speed-up point plays an important role when deciding on the search file size. The applications with the freedom over their preferred search file size can designate a size value known to result in better performance. Also, larger files can be divided into smaller pieces of established units to achieve optimal performance. The optimal search file size study is performed early in the testing to make informed decisions about the following tests and reduce the number of test cases for a concise examination. The rest of the tests are performed on 100 MB search files unless otherwise noted to benchmark the optimal performance results of the algorithms.

5.3.2. The Parallel Algorithm Performance Rankings

Based on the results acquired in section 5.3.1, we have used a 100MB text file split from the Wikipedia raw data dump. Again, four different lengths of patterns are used to perform the search operation over the prepared text file in order to obtain the comparison data. The parallel versions of the string matching algorithms are then sorted into three tables based on their performance. The first table contains the top five performers of the test for every pattern length tested. This table is presented in 5.2.

The table 5.2 shows some points worthy of note. For the shortest pattern test ($m = 3$), we see the parallel implementation of the Bruteforce algorithm at the 4th place. This is

a notable feat for the most basic, unoptimized approach, taking the 4th place among 80+ algorithms, which are designed over decades to improve the Bruteforce algorithm itself.

The simplicity of an algorithm plays an important role (often more decisive than utilized strategy) when it comes to the parallel operation. When the running threads in a warp encounter a decision point in kernel code (like `if` clause), their execution flow might differ based on the decision outcome. This causes a branching on execution path, where some threads halt operation until the other threads finish executing the extra work resulting from the decision, and the parallel operation becomes synchronized again. This partial suspension of the warp threads caused by the branching degrade concurrency and gets worse as the number of the branching points grow. Branch divergence is one of the major slowdown factors along with uncoalesced memory access, causing under-utilization for GPU assisted computing.

The algorithms that use complex decision mechanisms in order to reduce comparisons and redundant memory operations have more branching points. These algorithms are more susceptible to performance degradation caused by branch divergence compared to the algorithms with more straightforward execution flows. This is a trade-off that becomes apparent once we start translating serial string matching algorithms into parallel code.

The automaton based algorithms consistently ranked high across the lists, especially for higher pattern lengths. These algorithms are specifically Backward Nondeterministic DAWG Matching (2.1.3.3) variants like Backward Nondeterministic DAWG Matching with q-grams (2.1.3.19), Backward Nondeterministic DAWG Matching for long patterns (2.1.3.4), Simplified Backward Nondeterministic DAWG Matching (2.1.3.5), Forward Simplified Backward Nondeterministic DAWG Matching (2.1.3.17), Simplified BNDM with Horspool Shift (2.1.3.10), and Two-Way Nondeterministic DAWG Matching (2.1.3.6).

Backward Oracle Matching (2.1.2.7) and its variant Simplified Forward Backward Oracle Matching (2.1.2.15) are top performers of the chart along with comparison based Fast Search (2.1.1.29) on these tests.

Bottom 5 of the performance chart is given on the table 5.3 and shows the worst-performing string matching algorithms during these tests. These algorithms are poorly translated or simply not compatible with the parallel operation. They might require special care, partial or total redesign of the underlying logic when transforming into parallel operation. Rigorous examination of these algorithms for the possibility of finding a feasible parallel remodeling is out of the scope of this work.

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
bfs	83,1402	ksa	47,7346	ts	55,8337	ag	95,2882
ffs	77,1774	fdm	46,8584	ag	50,6226	kr	67,4551
bsdm	47,7171	bfs	45,7860	ksa	50,3636	fdm	54,1321
fdm	45,7309	ts	41,5247	fdm	49,3713	tw	52,8663
ksa	41,8872	ldm	32,2837	so	33,9019	ksa	51,2419

TABLE 5.3: Bottom 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

Name	Time (ms)			
	m=3	m=10	m=50	m=200
dfah	21.8992	21.8619	21.9484	21.9329
bf	14.7148	14.3534	15.0921	15.3365
faoso2	20.7582	21.8128	22.4165	22.4480
raita	14.4963	12.5348	11.5432	11.3684
br	14.7840	13.0556	11.7371	11.4242

TABLE 5.4: Most stable 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

The bottom 5 list contains algorithms from multiple categories, meaning no string matching strategy can be ruled as superior to others. This list also contains our Bruteforce modification Bruteforce Shared (bfs), which is designed to study the effect of shared memory usage on the Bruteforce algorithm performance. As apparent, the bfs algorithm performed poorly and will be discussed in the following sections.

Lastly, we have compiled the list of most stable algorithms and presented the top 5 in the table 5.4. In our tests, these algorithms are the least affected by the pattern length and run at almost constant time. This is a vital parameter for some applications requiring predictability over speed and uncertainty. The list contains our version of Deterministic Finite Automata (2.1.2.1), the Bruteforce (2.1.1.1), the Fast Average Optimal Shift-Or (2.1.3.15), the RAITA (2.1.1.19), and the Berry-Ravindran (2.1.1.27) algorithms.

5.3.3. Genome Data Test Results

Since the Genomics study is one of the largest fields that sees the extensive use of string matching, we have included DNA tests to compare the algorithm behavior over the DNA sequencing problem. The DNA sequence of Drosophila melanogaster (commonly known as fruit fly) is used as the source string in some of our test cases [96]. The sequence is duplicated enough to get a 100MB file for the search operation. Different patterns with the

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
sabp	13.0156	sabp	12.2282	sbndm2	11.7540	bmh_sbndm	11.3339
ildm2	13.3445	ildm2	12.4135	bmh_sbndm	11.7779	sbndm2	11.6962
ildm1	13.3451	ildm1	12.4138	bndm	11.8721	ildm1	11.8898
bww	13.4991	bww	12.4280	ildm1	11.8810	ildm2	11.8968
bndm	13.8913	lbndm	12.5027	ildm2	11.8815	trf	11.9659

TABLE 5.5: Top 5 parallel string matching algorithms for each pattern length(m) over 100MB DNA sequence.

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
fdm	50.4712	ldm	75.7919	ag	129.8907	ag	244.8445
ag	47.2614	ag	63.3685	fdm	54.0337	ldm	65.0828
ts	40.5084	fdm	62.4360	ts	40.7435	kr	64.4945
ldm	32.3250	ts	35.5995	kr	30.9725	fdm	51.5552
smoa	31.1867	smoa	31.8406	ksa	30.9442	tw	49.1941

TABLE 5.6: Bottom 5 parallel string matching algorithms for each patternlength (m) over 100MB DNA sequence.

lengths of 3, 10, 50, and 200 are used in the tests. These tests examine the performance of algorithms over structured and small alphabets with varying pattern lengths.

These tests are conducted in a similar fashion to section 5.3.2, using the DNA string as the source file. The pattern strings are randomly extracted from the DNA sequence in four different lengths; 3, 10, 50, and 200 characters.

These test results display similar rankings over different pattern lengths. Both Improved Linear DAWG Matching algorithm variations (2.1.2.11) are placed in the top 5 of the list for all cases. For shorter patterns, the ranking includes the Small Alphabet Bit-Parallel (2.1.3.22) as the top performer and the Bit Parallel Wide Window (2.1.3.13) in top 5. For longer patterns, the top two slots are shared between the Simplified BNDM with loop-unrolling (2.1.3.9) and the Horspool with the BNDM test (2.1.3.11) algorithms. Genome-based workloads generally use long sequences as the pattern of the search operation. Based on our results, these string matching algorithms using automata are good candidates for the operations involving genome processing.

Among the worst performing algorithms, it is interesting to see Linear DAWG Matching Algorithm (2.1.2.10) since the Improved Linear DAWG Matching algorithm variants (2.1.2.11) are some of the best performing algorithms on our lists. Some repeating names

Name	Time (ms)			
	m=3	m=10	m=50	m=200
bww	13.4991	12.4280	12.1474	13.3290
sabp	13.0156	12.2282	12.1871	13.2165
tvsbs	16.6004	15.2832	14.5014	16.3633
bndm	13.8913	12.5958	11.8721	13.6282
smith	15.2085	15.9766	14.6251	15.4721

TABLE 5.7: Most stable 5 parallel string matching algorithms for each pattern length (m) over 100MB DNA sequence.

can be seen on the table 5.6 like the Forward DAWG Matching (2.1.2.5), the Karp-Rabin (2.1.1.8), the Apostolico-Giancarlo (2.1.1.7), and the Tailed-Substring (2.1.1.32) algorithms. These algorithms performed 3 to 23 times slower than the best performing algorithms compared to the same test case.

The most stable 5 algorithms are given on the table 5.7. Among the most stable algorithms, the Bit Parallel Wide Window (2.1.3.13), the Small Alphabet Bit-Parallel (2.1.3.22), and the Backward-Nondeterministic-DAWG-Matching (2.1.3.3) are also some of the fastest algorithms and ranked top 5 in our tests. These algorithms are not affected by the pattern length variations easily and display good performance overall.

5.3.4. Different Alphabet Test Results

The algorithm behavior for different alphabet sizes is examined, and the results are presented in this section. The performance of the algorithms and the variations in the performance for different pattern lengths and alphabet sizes were examined. The alphabet sizes to be tested were determined as 2, 8, 16, 64, 128 characters, and the search operations were performed on the text files consisting of randomly generated characters from each alphabet. The size of the text files used during these tests was chosen as 100 Mb, which is determined following the earlier tests. The search tests were carried out with four different patterns of different lengths. The lengths of these patterns were chosen as 3,10, 50, and 200 characters and were randomly generated from the characters of the corresponding alphabet. Thus, 20 different test scenarios were prepared using four different length patterns and five different alphabets. In order to reduce variance from the test system, each test scenario was run ten times, and the average of the results was reported. In addition, all the tests were repeated on the re-created random text files, and the averages of the results were taken, reducing the potential effects of inadvertently structured text sequences.

The following tables (5.8, 5.9) list the fastest five and slowest five algorithms for each

TOP 5 OF ALPHABET SIZE TEST

$\sigma = 2$		$\sigma = 8$		$\sigma = 16$		$\sigma = 64$		$\sigma = 128$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
bndmq4	16.31	sbndm2	14.53	lbndm	14.43	ssabs	14.16	ssabs	14.01
sbndm2	17.07	fsbndm	14.56	sbndm2	14.64	ms	14.40	bxs	14.34
bmhsbndm	17.80	bndmq4	14.64	fsbndm	14.72	bxs	14.51	lbndm	14.67
sbndm	18.50	sbndmq2	14.81	sbndm	14.96	lbndm	14.62	fjs	14.79
tndm	18.58	fsbndmq2	14.87	sbndmbmh	14.96	tndm	14.80	tndm	14.80

TABLE 5.8: The Runtime of top 5 algorithms for different alphabet sizes (σ), times are the average values of 4 different pattern tests

BOTTOM 5 OF ALPHABET SIZE TEST

$\sigma = 2$		$\sigma = 8$		$\sigma = 16$		$\sigma = 64$		$\sigma = 128$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
smoa	171.18	ag	118.16	smoa	89.57	fdm	68.33	ffs	111.81
ag	165.66	smoa	102.62	ldm	68.65	ldm	67.22	bsdm	74.61
sabp	101.10	fdm	82.44	ag	65.74	smoa	58.91	fdm	70.45
ldm	72.81	ldm	68.81	fdm	65.51	ksa	49.87	ldm	67.97
fdm	70.49	ts	50.57	ts	45.62	bsdm	49.63	ksa	50.80

TABLE 5.9: The Runtime of bottom 5 algorithms for different alphabet sizes (σ), times are the average values of 4 different pattern tests

alphabet size sorted by the runtime. These times were obtained by averaging the search times for four different patterns of each algorithm on the selected alphabet. In this way, our aim was to compare the overall performance of the algorithms on the selected alphabet, including various pattern length scenarios.

When we look at the average time test results, the Backwards Nondeterministic DAWG Matching algorithm (2.1.3.3) derivatives are in the top five for the alphabets with 2, 8, or 16 characters. In particular, the Simplified Backwards Non-Deterministic DAWG Matching with Loop Unrolling algorithm (2.1.3.9), represented by the `sbndm2` tag, is consistently one of the two fastest algorithms. When it comes to larger alphabet sizes, we observe that different algorithms rise in the ranking. The SSABS algorithm is at the top of the performance chart for 64 and 128 character alphabets.

At the other end of the ranking, when we look at the five slowest algorithms, recurrent names can be seen on the table. Although these algorithms provide some speed gain compared to CPU versions after parallelization, their performances are below average. The reason is that these algorithms are not suitable for the hardware and technique applied; some processor features available on the CPU side cannot be transferred to the GPU side.

In the following graphs (5.2, 5.3, 5.4, 5.5, 5.6), we see how much acceleration algorithms have gained compared to CPU versions as a result of tests on selected alphabets. In these graphs, vertical bars corresponding to each algorithm represent the range of acceleration

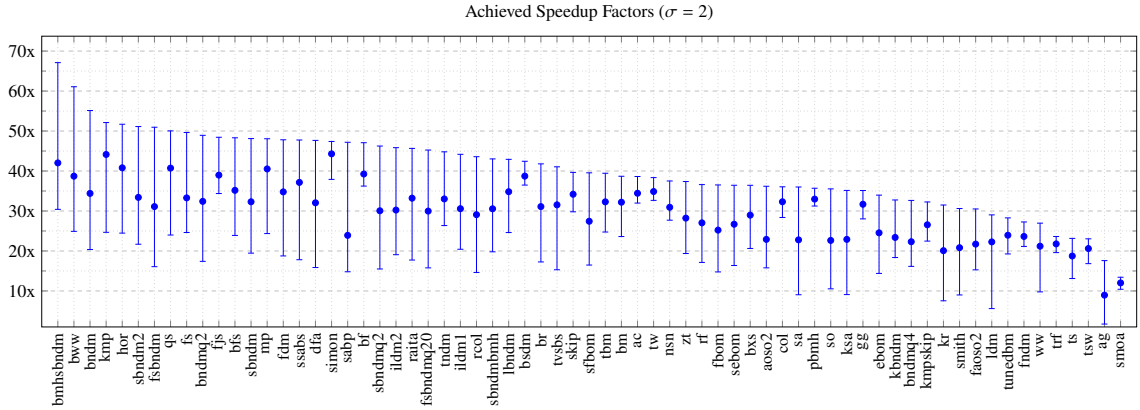

FIGURE 5.2: The obtained speedup factors for 2 character alphabet, ordered from highest to lowest. Vertical lines represent the range of speedup factors the algorithm achieves with the tested set of patterns.

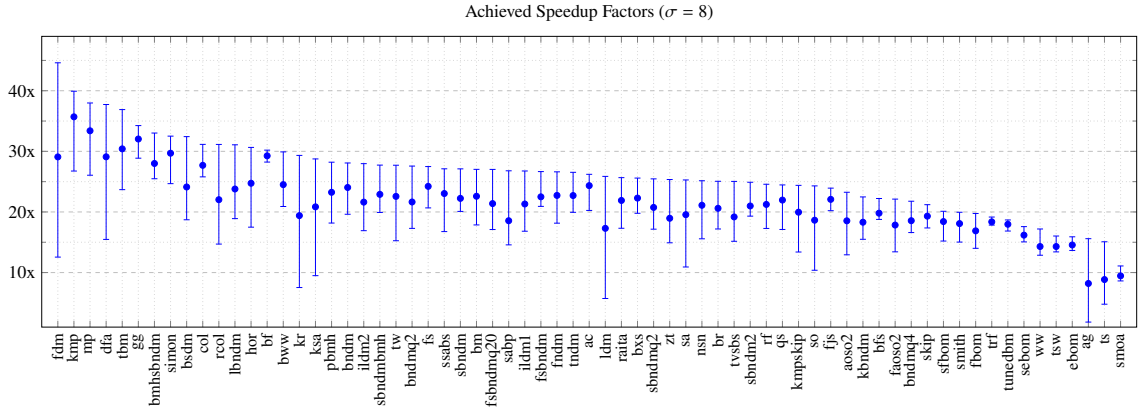

FIGURE 5.3: The obtained speedup factors for 8 character alphabet, ordered from highest to lowest. Vertical lines represent the range of speedup factors the algorithm achieves with the tested set of patterns.

rates obtained as a result of searches with patterns of different lengths. The upper end of the bar represents the highest speedup factor achieved, and the lower end represents the lowest speedup factor. The point in the middle of the bar shows the average value of the speedup factor obtained from all pattern tests.

When the graphs 5.2, 5.3, 5.4, 5.5, and 5.6 are examined, some inferences can be made about the behavior of the algorithms. First of all, as the size of the alphabet grows, an overall decrease in the values of the figure is noticeable.

For example, while approximately half of the algorithms in the figure 5.2, in which the alphabet size is 2 characters, can take values over the 40x limit, only the first two algorithms can reach this number in the 8 character alphabet figure (5.3). Although the downtrend continues to persist in other figures, there is no difference as big as the first two figures. In the light of this information, we can say that it will be advantageous to use parallel

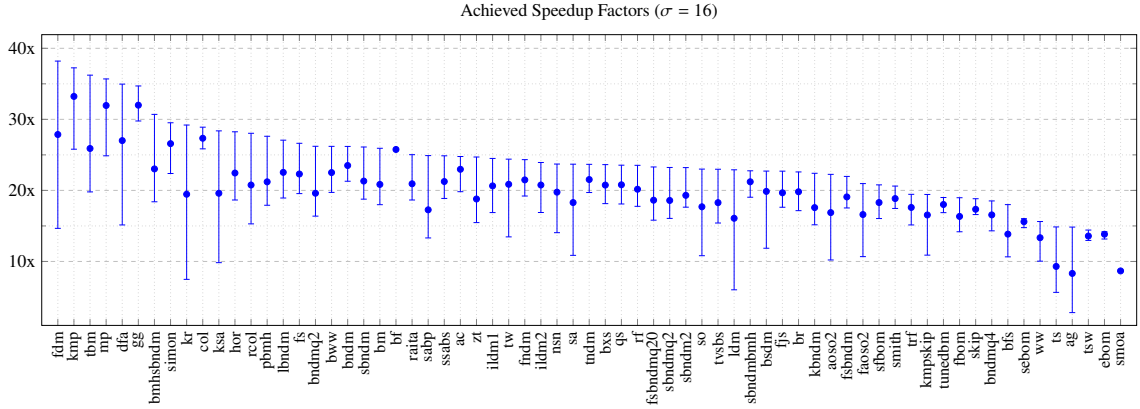

FIGURE 5.4: The obtained speedup factors for 16 character alphabet, ordered from highest to lowest. Vertical lines represent the range of speedup factors the algorithm achieves with the tested set of patterns.

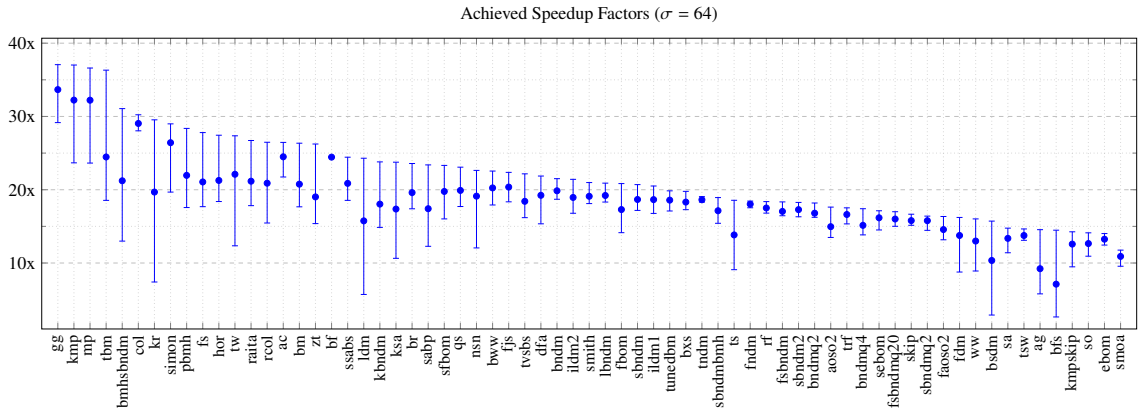

FIGURE 5.5: The obtained speedup factors for 64 character alphabet, ordered from highest to lowest. Vertical lines represent the range of speedup factors the algorithm achieves with the tested set of patterns.

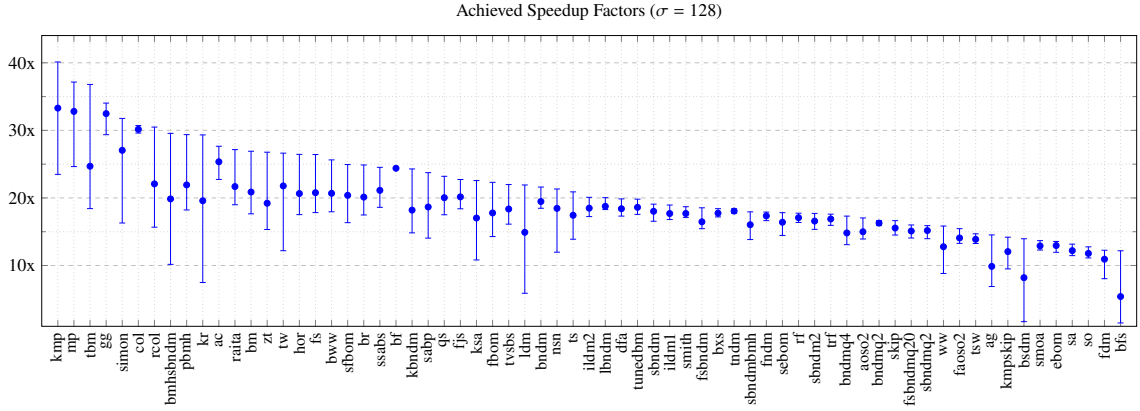

FIGURE 5.6: The obtained speedup factors for 128 character alphabet, ordered from highest to lowest. Vertical lines represent the range of speedup factors the algorithm achieves with the tested set of patterns.

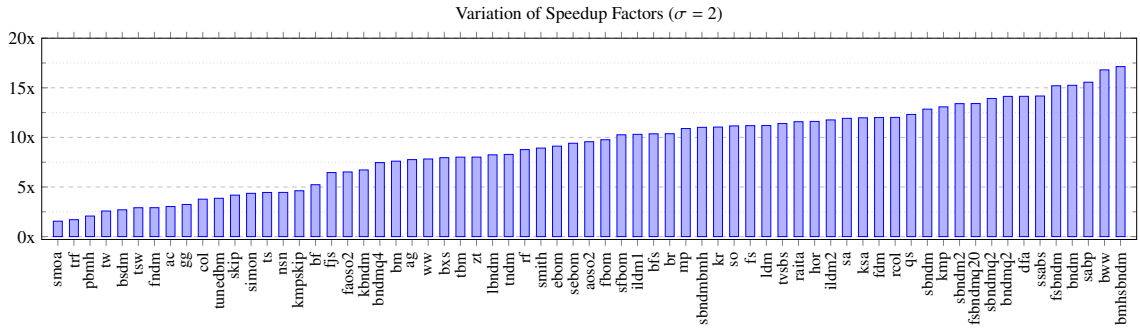

FIGURE 5.7: The variation values of algorithm speedups for alphabet size $\sigma = 2$

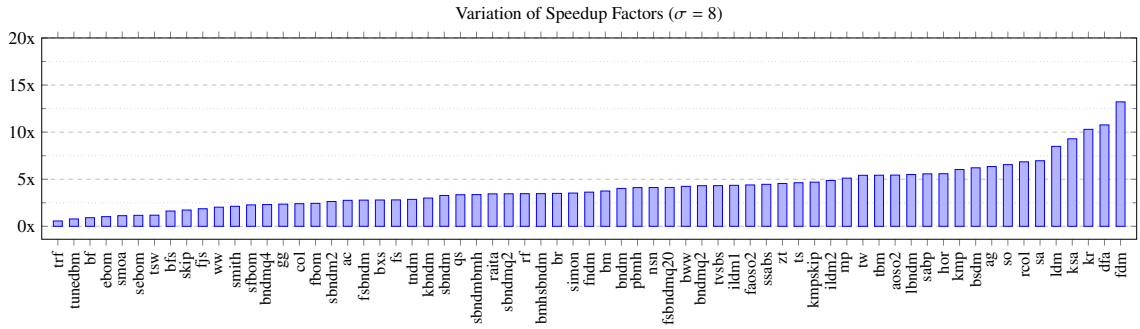

FIGURE 5.8: The variation values of algorithm speedups for alphabet size $\sigma = 8$

algorithms with GPU support, especially in cases where small alphabet sizes such as binary system or DNA are dealt with.

If we go back to the figure 5.2 again, we see that the bar lengths are longer compared to other figures and they get shorter as the alphabet size increases. This is due to the variance of speedup depending on the pattern length. In small alphabets, the relationship between speed gain and pattern length is more substantial, while this effect decreases as the alphabet size increases.

It can be seen from the graphs that the acceleration amounts of the algorithms change according to the length of the pattern searched. The relationship between algorithm performance and pattern length is also a case to be evaluated. For some applications, it may be preferable that the algorithm is less affected by the pattern length, so that the runtime is predictable rather than having variable performance gain. In different alphabets, the way each algorithm works may differ depending on the pattern length. While some algorithms slow down as the pattern length increases, some algorithms may accelerate, or the speed of some algorithms may not vary much, although the pattern length changes. In order to examine the relationship between the speed gains of the algorithms and the pattern lengths, the graphs 5.7, 5.8, 5.9, 5.5, and 5.11 were obtained by calculating the variation of the speed gains of the tests on each alphabet.

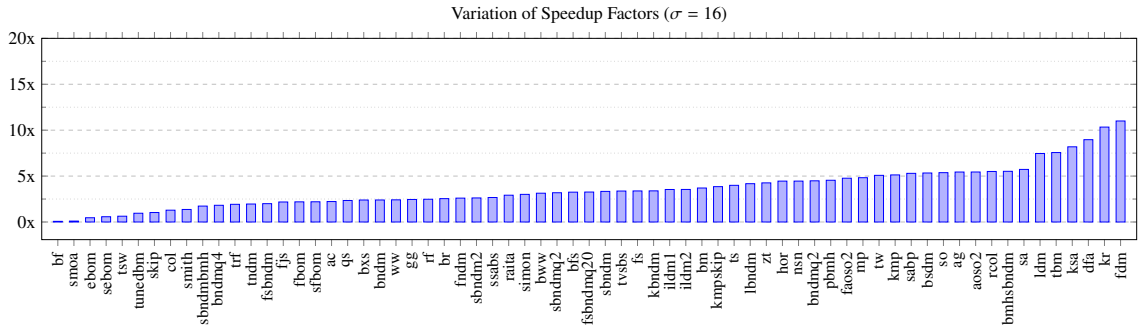

FIGURE 5.9: The variation values of algorithm speedups for alphabet size $\sigma = 12$

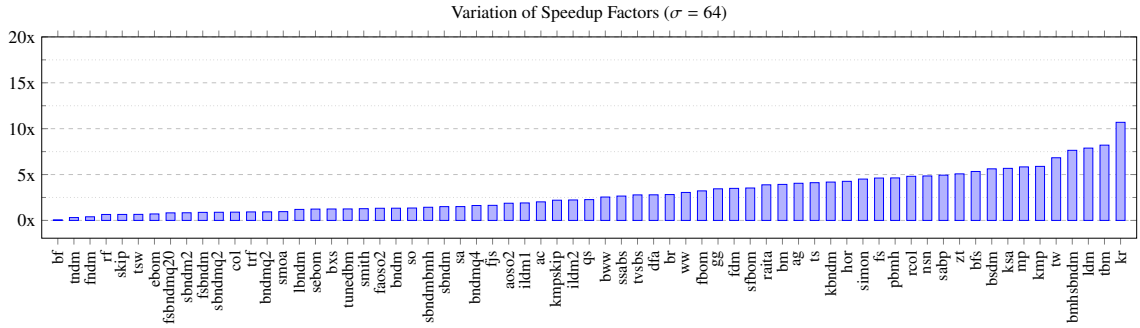

FIGURE 5.10: The variation values of algorithm speedups for alphabet size $\sigma = 64$

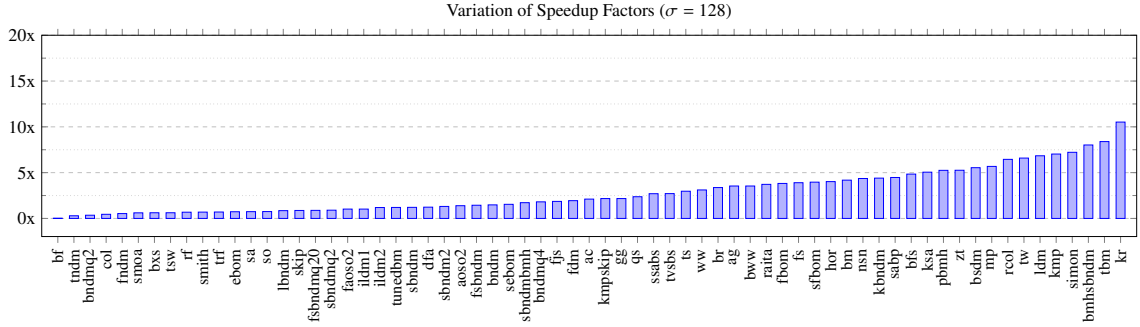

FIGURE 5.11: The variation values of algorithm speedups for alphabet size $\sigma = 128$

For each alphabet, the variation of the run times of each algorithm in different patterns is presented. For example, although the pattern length changes, the Bruteforce algorithm's standard variation value is very low, as can be seen from the graphics. The Bruteforce algorithm's inability to perform long shifts causes searches by sliding one by one, regardless of the pattern length. Thus, as the pattern length increases, the variation value stays low since the working time does not change much.

Like the stability tests performed in sections 5.3.2 and 5.3.3, these tests indicate the steadiness of the parallel algorithms on GPU for varying alphabet lengths and can be valued as a decisive factor for numerous applications.

5.3.5. Mobile unit tests performed on Nvidia Jetson TX2

Our tests were repeated on Nvidia Jetson TX2, a mobile computing platform, and this platform was compared to the desktop computer card. The specifications of the two cards we have used in our tests can be seen in table 5.10.

	Nvidia GeForce GTX 1080 Ti	Nvidia Jetson TX2
CUDA cores	1500MHz 3584 cores	1300MHz 256 cores
RAM	11 GB 352-bit GDDR5X	8 GB 128-bit LPDDR4
Power Consumption	250 W	7.5 - 15 W
Price	$ 1,239	$ 299

TABLE 5.10: The specifications of the hardware tested.

As can be seen from the table 5.10, although Nvidia Jetson TX2 has a lower computing power compared to Nvidia GeForce GTX 1080 Ti, it is much more efficient in terms of power consumption. When we consider the average operation power consumption as around 11.5 W [97], the Jetson TX2 has 5% power consumption of the GTX 1080 Ti. Considering that Jetson TX2 is an embedded system-on-module (SoM) and GTX 1080 Ti requires additional peripherals to function, it is safe to say that the difference in power consumption is expected to be more than 20 fold. When reviewing the performance of the Jetson TX2, these points need to be considered for a fair comparison.

We benchmarked the Jetson TX2 with the tests that were performed on GTX 1080 Ti in section 5.3.2, and compared the performance of two devices. The search tests are carried using different patterns with 3, 10, 50, and 200 characters, extracted from the 100 MB structured English text. The text file used in the search operation is constructed from the Wikipedia data dump.

Results are presented on the tables 5.11, 5.12, and 5.13.

JETSON TX2 TOP 5 LIST

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
bndmq4	76.6029	sbndmq2	88.1343	sbndm2	73.4692	fsbndm	70.5400
sbndmq2	138.7765	bndmq2	89.7639	sbndmq2	74.3597	lbndm	70.6075
bndmq2	139.9111	fsbndmq2	89.7701	tndm	74.4925	sbndmbmh	70.6345
fsbndmq2	142.6781	sbndm2	90.2837	bndmq2	74.5494	sbndm	70.6546
sbndm2	145.6219	fsbndm	93.9733	fsbndmq2	74.6377	ildm1	71.8680

TABLE 5.11: Jetson TX2 top 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

JETSON TX2 BOTTOM 5 LIST

$m = 3$		$m = 10$		$m = 50$		$m = 200$	
Name	Time (ms)	Name	Time (ms)	Name	Time (ms)	Name	Time (ms)
smoa	1156.6370	smoa	871.5062	smoa	1072.0115	ag	1453.0983
ldm	1080.9525	ts	595.2252	ag	913.9572	smoa	1205.4846
ww	956.1476	ag	541.0140	ts	779.2892	fdm	766.2142
bfs	753.8951	fdm	533.6792	bfbs	622.5566	kr	526.5508
ag	730.0417	ldm	484.9097	fdm	594.7779	ksa	445.1425

TABLE 5.12: Jetson TX2 bottom 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

JETSON TX2 CONSTANT 5 LIST

Name	Time (ms)			
	m=3	m=10	m=50	m=200
bf	236.3945	238.0761	237.5805	244.1697
dfah	276.8970	280.6578	274.3632	286.8196
bndmq4	76.6029	97.4885	81.7373	99.5955
kmp	255.5782	226.6539	239.6609	243.4364
mp	254.9024	225.2835	239.2712	244.0607

TABLE 5.13: Jetson TX2 most stable 5 parallel string matching algorithms for each pattern length (m) over 100MB wikipedia text.

Best performing algorithms table 5.11 lists different tags compared to the corresponding test performed on the GPU. Still, both rankings are consistent category-wise, most of the algorithms listed here are automata-based, Backward Nondeterministic DAWG Matching (2.1.3.3) variants. Especially the variants employing q-grams approach like Backwards Nondeterministic DAWG Matching with q-grams (2.1.3.19), Simplified BNDM with q-grams (2.1.3.20), and Forward Simplified BNDM with q-grams (2.1.3.17) display improved performance on Jetson TX2 platform.

The average runtime of the top 5 algorithms table 5.11 is approximately 91 ms, when compared to 12ms, which is the average of GPU top 5 given on the table 5.2, Jetson TX2 comes out 7.5 times slower than the GPU. However, when the power consumption factor is added to the equation, 20 times less power consumption of Jetson TX2 makes it 2.5 times more efficient than the GPU device tested.

On the downside, the initial cost of the Jetson TX2 system for the same amount of computing power is worse, the GPU device used is approximately 4.2 times more expensive, but performs 7.5 times better for a single unit. The Jetson TX2 unit is 80% more expensive for the initial cost. Although, when the costs of peripheral units are added, a typical computer system running with the mentioned GPU costs around $2,000. This value puts the TX2 cost much closer to the GPU unit, only 10% more expensive.

Overall results indicate a significant work-per-watt ratio for Jetson TX2 and put the device in a favorable position for string matching operations.

5.3.6. The Parallelism Granularity Test Results

To acquire a good balance between efficient thread utilization and parallelism, a good granularity factor must be determined. In order to do that, we have performed 12 different stride factor tests for 4 different patterns on the 100 MB structured text file. In our tests, base stride length is a multiple of pattern length, and it is further modified by the selected factor. The reasoning behind selecting a constant, pattern sized base stride length is simple. Statistically, the first comparison of the search operation has a high probability of mismatch, and this phenomenon is observed during our tests too. Many string matching algorithms are improved to the point that they do full pattern length shifts after the first mismatch, meaning they process the given text string in pattern sized jumps. Portioning out the text into pattern sized string pieces allows these algorithms to perform efficient shifts without wasting read operations on small substrings. In practice, this strategy improves the performance of the majority of algorithms. The stride factor can be vaguely understood as the number of work packets assigned to each thread. Table 5.14 indicates

that the lower factors of stride length results in poor performance due to under-utilization, while the higher factors also degrade performance because of reduced parallelism. There is a sweet spot between 8 to 15 times for the most of the algorithms presented on table 5.14. We have used the factor of 10 in our other tests to get peak performance granularity-wise.

5.3.7. Constant Memory Test Results

We carried out constant memory tests on the 100MB genome dataset with 4 different patterns lengths: These patterns contain 10, 20, 50, and 800 characters.

The small size of the constant memory limits its usage possibilities for the string matching problem, despite the high number of read-only access to data sources. For example, the text string used in our tests takes a large amount of memory space and does not fit into space reserved for constant memory. As a result, we have used constant memory space to store our pattern strings. For these tests, the patterns are transferred into constant memory instead of global memory, and kernel transactions are modified to use this memory space. After performing tests and looking at the average runtimes given on table 5.15, the results suggest that there is no significant performance difference when using constant memory.

Still, there are some outliers performing better with constant memory usage. These algorithms are presented in table 5.16.

A few algorithms display noticeable performance gains, especially on short patterns, but even these gains gradually decrease and almost disappear on the longest pattern test. Still, if these algorithms are planned to be used, constant memory storage can be exercised for the potential performance gains.

5.3.8. Shared Memory Test Results

Shared memory is a memory segment superior to global memory in terms of access speed. Its impact on algorithm performance is examined during our tests. Since it can speed up the repetitive accesses to the same memory region, each algorithm kernel block is arranged to transfer the data it will process to the shared memory section reserved for it.

When this scenario was tested, instead of a decrease in working time, there was an increase, meaning the performance is degraded. This is an expected result, given the algorithm characteristics:

		Kernel times (ms) for different stride factors										
Tag	Name	1	2	3	6	9	12	15	30	60	120	150
bf	Bruteforce	5.1671	5.1754	5.1760	5.1780	5.1776	5.1785	5.1766	5.1770	5.1773	5.1829	5.1773
bfb	Bruteforce block	10.4847	7.9903	7.2194	6.5729	6.6224	6.7905	7.0072	8.3807	8.6354	11.1742	12.8924
bfbs	Bruteforce block shared	9.8287	7.4124	6.5947	5.7822	5.5313	5.4052	5.3320	5.1908	5.1846	6.7309	6.7071
mp	Morris Pratt	14.9288	12.4246	11.6599	10.9195	10.9171	11.0967	11.2455	11.8337	12.2151	13.8776	15.6797
kmp	Knuth Morris Pratt	14.9380	12.4236	11.6619	10.9165	10.9775	11.0955	11.2451	11.8306	12.2107	13.8819	15.6847
simon	Simon	28.7933	19.0938	15.9713	12.6539	11.8243	11.4939	11.3289	11.7533	11.7731	12.6386	13.6887
dfa	Det. Finite Automaton	40.5894	28.2631	22.9177	17.8172	16.3458	15.7572	15.6029	16.0958	15.7992	15.7941	16.6280
bm	Boyer Moore	10.3113	6.5763	5.2542	5.3725	5.4618	5.5730	5.7177	6.5910	6.7786	7.8939	8.9111
ag	Apostolico Giancarlo	22.4853	13.4570	12.1341	9.3335	8.5474	8.2141	8.0096	7.8015	7.8438	9.8757	11.2265
hor	Horspool	11.8220	8.1584	7.1300	6.3749	6.2151	6.2355	6.3417	7.1733	7.2710	8.4991	9.5766
zt	Zhu Takaoka	22.8672	12.0132	9.4441	9.8399	10.1299	10.3686	10.6319	11.9218	12.0531	12.1374	12.1739
kr	Karp Rabin	3.9346	2.9713	2.8867	3.4064	4.0014	4.6062	5.2086	8.2562	8.5771	9.8551	12.9047
qs	Quicksearch	14.4442	7.9683	6.0535	5.4868	5.3416	5.0580	4.9625	5.4944	5.4380	6.7290	7.6448
om	Optimal Mismatch	20.8013	11.2083	8.4390	6.6878	6.2576	5.8600	5.7109	5.9460	5.7880	7.3051	8.0840
ms	Maximal Shift	27.1669	14.5611	10.8935	8.4887	7.8348	7.2078	6.9697	6.7361	6.5045	8.0695	8.3204
smith	Smith	13.6797	7.9621	6.1095	5.8308	5.8050	5.8291	5.8346	6.7833	6.8283	7.3298	7.7793
ac	Apostolico Crochemore	15.0714	11.4889	10.2307	9.0822	8.9052	8.9818	8.9508	9.7027	9.8745	12.7404	12.5387
raita	Raita	7.5343	5.0291	4.2104	4.0857	4.1273	4.2463	4.4037	5.3303	5.4400	7.8698	9.0920
tunedbm	Tuned Boyer Moore	24.5196	20.0914	14.2823	10.7845	9.8242	9.4754	9.3567	9.5316	9.9133	10.1013	10.1285
so	Shift Or	41.0770	28.4429	23.0100	17.9005	16.4239	15.8369	15.6189	16.1091	15.8678	15.8619	16.7875
sa	Shift And	37.6204	28.4408	23.0719	17.9010	16.4228	15.8449	15.6176	16.1102	15.8681	15.8673	16.7954
nsn	Not So Naive	3.8861	4.5927	4.6086	4.8336	5.1351	5.4326	5.7281	7.1887	7.4938	9.6598	11.3452
col	Colussi	16.6689	14.8716	13.2462	11.7453	11.4373	11.4607	11.4051	12.0190	12.2058	13.7950	15.0331
tw	Two Way	13.8391	11.0688	9.2609	7.6319	7.3151	7.2648	7.3275	8.1203	8.1855	10.3662	11.9102
smoa	SM Ordered Alphabet	14.5005	11.6431	9.9122	8.1935	7.9259	7.8705	7.9704	8.9484	8.9350	10.5102	11.3912
gg	Galil Giancarlo	16.7075	14.8876	13.2625	11.7483	11.4543	11.4836	11.4274	12.0171	12.2188	13.7155	14.7866
rf	Reverse Factor	17.1769	16.7863	11.9414	9.2485	8.4296	8.0751	7.9111	7.8608	7.6084	7.5282	7.5938
skip	Skip Search	17.9930	10.9616	14.9690	11.6247	10.6142	10.1254	9.8943	9.7334	9.5330	9.3516	9.3029
tbm	Turbo Boyer Moore	9.5256	6.5770	5.2533	5.3736	5.4592	5.5753	5.7201	6.5922	6.6924	7.8561	8.8781
kmpskip	KMP Skip Search	12.4100	7.9943	9.5415	8.4171	8.3899	8.5822	8.8269	9.7521	10.1882	10.2276	10.2167
br	Berry Ravindran	9.6850	6.2074	4.8934	4.3102	4.1608	3.8242	4.1523	4.7394	4.6415	5.8782	6.6997
fs	Fast Search	9.2713	6.4184	5.1404	5.6488	6.0752	6.4373	6.7758	8.0138	9.0099	9.6559	9.7624
fdm	Forward DAWG Matching	71.7002	51.0376	42.5043	34.5391	31.3384	29.8581	29.1308	28.6461	27.8645	27.4548	27.3197
bndm	Backward nondet. DAWG Matching	16.2991	11.0557	10.6433	9.8891	9.1191	8.7938	8.7293	8.6751	8.5039	8.4701	8.5626
ssabs	SSABS Algorithm	10.1333	6.9415	4.9988	5.2790	4.7428	4.5937	4.7905	5.3645	5.3591	6.6786	7.6235
ts	Tailed Substring	14.0173	14.0826	13.9369	15.4564	16.3200	16.5434	16.5631	15.3883	12.9272	11.9587	12.2994
bom	Backward Oracle Matching	31.3083	21.2970	16.5209	12.7377	11.5078	10.8839	10.4902	9.8080	9.5315	9.8251	9.8884
ww	Wide Window	13.3386	9.0926	7.7378	6.5707	6.3798	6.2683	6.0846	5.7764	6.1964	9.1988	10.7037
rcol	Reverse Colussi	15.0996	8.3503	10.3971	8.3628	7.8830	7.8294	7.7962	8.1090	8.3198	8.7583	8.7365
svm	Shift Vector Matching	15.6223	9.4875	7.3987	7.1531	7.0727	7.0532	7.1370	7.4303	7.4348	7.4605	7.4915
ldm	Linear DAWG Matching	27.9260	19.2598	16.6748	14.4635	14.3668	14.2811	13.8804	13.2418	14.5003	22.1287	26.3385
tndm	Two-way nondet. DAWG Matching	21.3108	20.5819	13.8967	10.7233	9.7458	9.3183	9.0957	8.9919	8.7479	8.6609	8.6422
tvsbs	TVSBS	22.2747	14.1416	9.8301	8.5422	7.1233	7.8523	7.0661	7.7981	7.8241	7.8297	7.8811
sbndm	Simplified BNDM	22.7919	12.8767	16.0392	13.2313	14.3090	13.6849	14.4930	15.7030	15.8530	16.0138	16.4118
lbndm	Long BNDM	19.5410	11.7745	12.5415	9.7518	8.9319	8.6316	8.4165	8.4561	8.2707	8.2250	8.2094
fjs	Franek Jennings Smyth	11.4811	10.2626	7.6212	6.3307	6.5574	6.5835	6.9313	7.9871	8.4101	8.6022	8.6154
sbndm2	Simplified BNDM /w loop unrolling	22.8690	13.2135	16.3044	13.5667	14.7000	14.1261	14.8837	15.9089	16.1504	16.1910	16.1871
fndm	Forward nondet. DAWG Matching	17.7703	9.1956	12.0349	9.1729	8.3034	7.9144	7.7075	7.6519	7.4060	7.3694	7.3632
tsw	Two Sliding Window	24.0496	15.0686	10.4902	9.1385	7.5487	8.3169	7.4736	8.2757	8.2519	8.2367	8.2867
bfs	Backward Fast Search	178.8744	109.3281	90.5356	56.7785	46.4477	39.4158	36.8212	26.2257	20.7737	17.8026	16.6573
bndmq2	BNDM with loop unrolling	19.2271	10.7458	13.5764	11.1347	12.1467	13.0446	13.8619	15.7053	15.9211	16.0203	16.0238
sbndmbmh	Simplified BNDM with Horspool Shift	19.1681	10.8352	13.4948	11.8317	12.9422	13.7468	14.1130	15.6075	15.7916	16.0162	16.3498
pbmh	Boyer Moore Horspool using Probabilities	9.4807	7.1094	5.6316	5.6669	5.4912	5.5078	5.6278	6.5128	6.6121	7.9857	8.9009
trf	Turbo Reverse Factor	11.9395	9.7684	6.9741	6.5042	5.9916	5.7831	5.7053	5.8396	5.6793	5.8482	5.9751
bww	Bitparallel Wide Window	0.5238	0.2670	5.1211	6.2940	6.4606	6.6059	6.7959	7.3196	7.5267	7.5650	7.5815
ebom	Extended Backward Oracle Matching	37.9217	20.8187	21.1009	18.5191	19.2979	18.1226	19.1311	20.3659	21.9094	23.9442	24.5605
fbom	Forward Backward Oracle Matching	19.1060	10.4080	8.5611	9.7644	10.5980	9.9647	10.4208	11.7088	12.6345	13.2371	13.3752
sebom	Simplified Extended Backward Oracle Matching	10.1885	5.8486	5.7385	5.1295	5.4986	5.3291	5.8130	6.8400	6.8751	7.1734	7.3480
sfbom	Simplified Forward Backward Oracle Matching	7.0314	4.3687	3.4347	3.1124	3.1855	2.9920	3.1851	3.7583	3.8458	4.0148	4.0974
bmhsbndm	Horspool with BNDM test	15.0872	8.5104	7.5539	8.4900	8.3025	8.2176	8.2072	8.4875	8.4424	8.4720	8.5310
faoso2	Fast Average Optimal Shift Or	19.2967	19.4648	13.2954	12.4700	8.5531	8.8152	9.1195	9.1589	8.3318	8.1569	8.2152
aoso2	Average Optimal Shift Or	22.2899	14.8065	10.1128	10.3901	8.6462	8.8371	8.1676	8.6601	8.3754	8.5144	8.7649
ildm1	Improved Linear DAWG Matching 1	11.2710	5.0117	6.6623	6.2146	5.7265	5.5244	5.4444	5.5618	5.4344	5.5014	5.5899
ildm2	Improved Linear DAWG Matching 2	11.3155	5.3873	7.6616	6.2151	5.7277	5.5243	5.4447	5.5586	5.4320	5.4951	5.5911

TABLE 5.14: The algorithm runtimes (ms) for different stride length factors.

Constant Memory Average Speedups	
Pattern Length (m)	Speed Factor
10	1.0210
20	1.0158
50	1.0065
800	0.9974

Table 5.15: Average speed factors with constant memory usage compared to global memory. Tested on 100 MB DNA dataset.

Constant Memory Speedup Top 5							
$m = 10$		$m = 20$		$m = 50$		$m = 800$	
Name	Speedup (%)	Name	Speedup (%)	Name	Speedup (%)	Name	Speedup (%)
mp	28.13	mp	24.46	pbmh	16.57	rcol	0.72
kmp	28.00	kmp	24.41	mp	11.63	tbm	0.60
skip	21.78	pbmh	19.00	kmp	11.59	kmpskip	0.41
pbmh	21.75	skip	18.89	ac	10.75	smoa	0.40
ac	13.88	fdm	16.95	simon	10.21	mp	0.23

Table 5.16: Top 5 algorithms with improved performance when constant memory is used.

Although the algorithms we examined perform searches using a variety of strategies, it can be seen that almost all of them follow some basic principles of string matching. One of these principles is to avoid excessive reading and to make maximum use of the search information obtained by reading each character. Many algorithms avoid repetitive comparisons of the same character by encoding the information gained about the search state into various variables and arrays. The absence of recurring access to the same memory region prevents us from taking advantage of the shared memory.

Also, since each global memory reading process will turn into one global memory reading plus one shared memory reading operation, the second method is slower than the first for single accesses. Even though the shared memory reading process is fast, it is not instant. In addition, since most 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 are to be read and which are to be skipped before the actual comparison happens, all the 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 understood that moving the search text to shared memory would cause more harm than good, and shared memory was not utilized for this purpose.

Algorithm Name	Pattern Size	Global Memory (ms)	Shared Memory (ms)	Speedup
SSABS	3	16.1650	12.1057	25.11%
	6	14.2783	11.7297	17.85%
	10	13.1251	11.5359	12.11%
	20	12.5579	11.4647	8.71%
Knuth Morris Pratt	3	21.8544	16.1452	26.12%
	6	21.8586	16.6721	23.73%
	10	19.8805	16.4145	17.43%
	20	20.9353	17.5718	16.07%
Simplified BNDM /w loop unrolling	3	24.5025	12.1523	50.40%
	6	17.6477	11.7191	33.59%
	10	15.0924	11.6362	22.90%
	20	13.4188	11.4281	14.84%

TABLE 5.17: Comparison of total search operation times based on the memory type. Preprocessed data is stored on global and shared memory for these test cases

Instead, we have chosen a few algorithms with the auxiliary data structures resulted from their preprocessing steps to test the feasibility of 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.

To take advantage of shared memory, first we have transferred the auxiliary data structures by grouping them under a monolithic data block ın memory. This concatenation operation is necessary to lower the number of uncoalesced accesses at the data termination points of each variable. After the kernel initialization, the threads are instructed to transfer these data structures to shared memory before the searching operation begins. A block-wide synchronization lock is added following the shared memory transaction in order to make sure the entire data is transferred successfully.

We have selected the SSABS algorithm(2.1.1.33), Knuth Morris Pratt Algorithm(2.1.1.3), and Simplified BNDM with Loop Unrolling algorithm(2.1.3.9), and evaluated their performance using multiple patterns on structured English text. Our test results are presented on table 5.17. On this table, the comparison of the kernel using global memory for auxiliary data and the kernel modified to use shared memory for the same data is given for different patterns. Also, the speed-up acquired in each test scenario is reported. The results show that there are varying amounts of speed-ups for different algorithms when using shared memory. The performance gain is always positive and varies between 8.71% and 50.40%, and averages at 22% for all tests. The improvement starts bigger for shorter pattern lengths and diminishes as the pattern length gets longer.

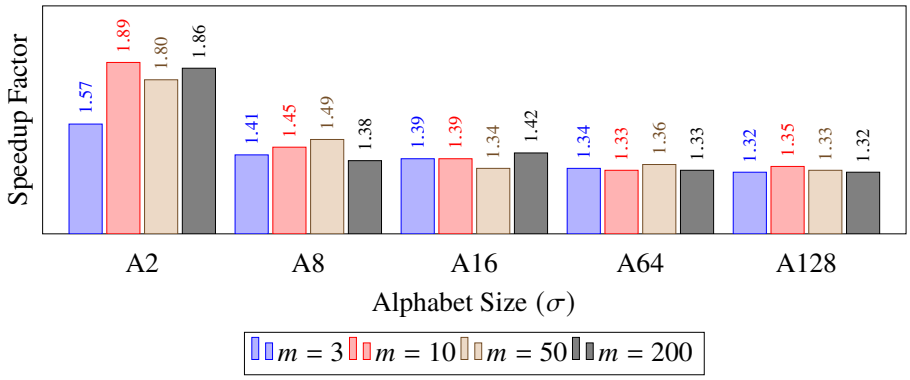

FIGURE 5.12: Speed-up factors of overlapping for different alphabet and pattern sizes, without pinned memory usage.

Overall, the positive impact of the shared memory usage for auxiliary data is apparent in the test results.

5.3.9. Overlapping Test Results

Another one of the optimization techniques tested is the technique called overlapping. The data transfer units on the graphics processor and the kernel units can work without getting blocked by each other. In this technique, the fact that the memory transaction unit being independent of the kernel processor unit is used to reduce the downtime of cores. In order to use this feature, the data to be transferred to the graphics card is transferred in small batches, and after the transfer of each piece, the kernel that will process the related part is called. When the data is transferred and processed sequentially, the kernel processors idle during the data transfer and wait until the memory is ready. In the case that the data is partitioned into pieces and kernel operations are thrown in between, the hardware can be used more efficiently since the kernel process can start from the moment the transfer of the first piece is completed. This way, most of the kernel process can be overlapping with data transfer and hidden under the transfer in a sense.

For the overlapping tests, stream queues of the CUDA are used. We have evaluated 20 test cases to benchmark overlapping strategy. The patterns with the length of 3, 10, 50, and 200 characters are searched on text strings constructed from several alphabet sizes. These alphabets consist of 2, 8, 16, 64, and 128 characters.

The overlapping test results on the figure 5.12 indicate an average speed-up factor of 1.46, which means the algorithms running with overlapping are 46% faster in general. The speed-up is noticeably higher for the binary alphabet results ($\sigma = 2$), peaking at 89%, and averaging at 78%.

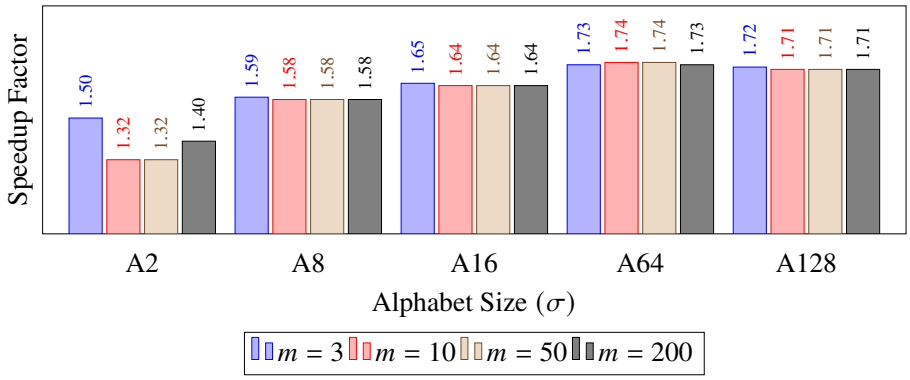

FIGURE 5.13: Speed-up factors of overlapping for different alphabet and pattern sizes, with pinned memory usage.

Another interesting result is observed while testing overlapping strategy along with the pinned memory usage and presented in figure 5.13. Compared to pinned memory use without overlapping, using both strategies yielded extra performance gain. The average speed-up of overlapping with pinned memory tests goes up to 62% from 46% of no pinned memory usage case. Also, an upward trend on speed-ups is observed to a point for combined test cases, while the overlapping alone shows decreasing performance gains. These results can be seen on the figure 5.13.

Overall, the overlapping strategy is a widely implemented strategy to improve performance and introduced a positive impact on our use case too. The overlapping strategy is also important for online operation since the memory transaction and search phase can be layered efficiently. This approach has the potential to hide a large chunk of operation. For the best-case scenario, when the memory transaction takes as long as the search operation, the theoretical speed-up is 50%, because half of the operation can be hidden behind the other half. Our overlapping implementation achieved 46% speed-up on some tests compared to normal, which is very close to the theoretical maximum.

6. CONCLUSION

In this work we presented our main contribution; the CUSMART library. The CUDA powered String Matching Research Tool (CUSMART) is a parallel string matching library developed using NVIDIA's CUDA toolkit and C++ language. Our motivation was to compile a codebase of parallel string matching algorithms to aid studying, testing, and developing string matching algorithms using GPUs. We have demonstrated the implementation details of the library along with reasoning behind design decisions and explained the steps necessary to add new algorithms to the library in order to compare with existing ones. Our library contains over 85 parallel string matching algorithms implemented using CUDA along with their CPU versions. To best of our knowledge, a parallel string matching library of this caliber was never done before.

Using our library, we have also evaluated the parallel string matching algorithms in several test scenarios to see how they behave under different use cases. These test scenarios are constructed using different text sizes, different pattern sizes, and different alphabets. Using a diverse set of configurations, we have tried to simulate many different application fields. Aside from task specific parameters, many system and architecture specific features like different memory types, overlapping, and different stride lengths are also evaluated. On average, a speedup factor of 40 is achieved on GPU versions of string matching algorithms compared to CPU versions. Our results indicate that the GPU devices are great candidates to process string matching heavy workloads. They can be used as the main computation units for the job as well as the supporting devices to sideload some of the operation. On a composite operation with different types of tasks, string matching job can be handed out to the GPU device, making the CPU available for other tasks.

One of our main objectives was to evaluate the feasibility of string matching operation on GPU unit as an extra computation resource instead of main operational unit. When we exclude the specialized high performance computing hardware, there are hundreds of millions of desktop computers with graphics processing units already present in use today. The GPU devices used to be seen as extra hardware not mandatory for the operation of computers. Nowadays, they are essential parts of the computer system. Besides the discrete GPU devices with continuously increasing power, marketed to mainstream audience, most of the processor chips sold by the CPU manifacturers come with powerful integrated GPUs. It is not uncommon to encounter multiple GPUs on a desktop computer today. Using this untapped computing potential of GPU devices on other applications

besides graphics computing is beneficial for system performance. Consumer applications and small-scale workstations can leverage the power of off-the-shelf GPU devices to gain a boost of performance easily thanks to their affordability and availability.

The GPU accelerators also offer superior performance per watt on the high-end of the computing spectrum. There is an actively maintained list of supercomputers worldwide called Green500 [75], and this list rates supercomputers by their energy efficiency. It is no surprise that 90% of the top supercomputers on the Green500 list employs GPU devices for their computing needs.

On the other end of the computing spectrum, GPU accelerators also have significant presence on mobile platforms. Our Jetson test results demonstrated increased power efficiency compared to the GPU device on workstation. The Jetson time benchmarks are 8 to 10 times slower than GeForce device. But when the impressively low power consumption of the Jetson device is considered (which is 20 times less than the tested GeForce device on average), operation efficiency becomes superior. After comparing the running costs of both GPUs, the Jestson device proved to be around 2.5 times more power efficient during our tests. These mobile devices offer top-notch performance for operations carried with limited resources.

We have tried to showcase the capabilities of modern GPUs for the exact string matching operations in this study. During our research, the lack of a large codebase on the parallel string matching subject became apparent, and the CUSMART library was born as a result of our pursuit. The CUSMART library constantly gained new features and improvements for the duration of our study. However, there is still work to do to further the project. Due to our time constraints and the magnitude of the project, we were only able to apply the generic optimization techniques for GPU programming. These optimizations yielded a substantial amount of performance gain but there is still room to improve. It is possible to perform a tighter analysis on each implemented algorithm for a chance to discover more performance improvements. CUDA programming model exposes the detailed customization options for many of its functioning parts, which means there are lots of potential optimization avenues to explore. These possibilites can be examined in future studies.

Still, at its current state, the CUSMART library test results demonstrate the feasibility of the string matching operation on GPU devices.

BIBLIOGRAPHY

[1] S. Y. Lu and K. S. Fu, "A Sentence-to-Sentence Clustering Procedure for Pattern Analysis", *IEEE Transactions on Systems, Man and Cybernetics*, vol. 8, no. 5, pp. 381–389, 1978, ISSN: 21682909. DOI: 10.1109/TSMC.1978.4309979.

[2] C. M. and W. Rytter, "Text algorithms", *Choice Reviews Online*, vol. 32, no. 10, pp. 32–5696–32–5696, 1995, ISSN: 0009-4978. DOI: 10.5860/choice.32-5696.

[3] T. S. Han, S. K. Ko, and J. Kang, "Efficient subsequence matching using the longest common subsequence with a dual match index", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4571 LNAI, pp. 585–600, 2007, ISSN: 03029743. DOI: 10.1007/978-3-540-73499-4_44.

[4] X. Tian, Y. Song, X. Wang, and X. Gong, "Shortest path based potential common friend recommendation in social networks", *Proceedings - 2nd International Conference on Cloud and Green Computing and 2nd International Conference on Social Computing and Its Applications, CGC/SCA 2012*, pp. 541–548, 2012. DOI: 10.1109/CGC.2012.106.

[5] T. K. Sellis, "Multiple-query optimization", *ACM Trans. Database Syst., vol. 13, no. 1*, pp. 23–52,

[6] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977, ISSN: 15579654. DOI: 10.1109/TIT.1977.1055714.

[7] C.-Y. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics", *in Proceedings of ACL '04. Stroudsburg, PA, USA: Association for Computational Linguistics*, 605–es, 2004. DOI: 10.3115/1218955.1219032.

[8] E. G. M. Petrakis, "Image Representation , Indexing and Retrieval Based on Spatial Relationships and Properties of Objects Image Representation , Indexing and Retrieval Based on Spatial Relationships and Properties of Objects", *Medicine*, no. March, 1993.

[9] P. E. Ceruzzi, "A history of modern computing", in *A history of modern computing*, London, Eng.: MIT Press, 2003, p. 161.

[10] T. P. Morgan, "Top 500 Supers – The Dawning of the GPUs", vol. 201, no. 5/11/2013, http://www.theregister.co.uk/2010/05/31/top_500_su, 2010.

[11] H. Hacker, C. Trinitis, J. Weidendorfer, and M. Brehm, "Considering GPGPU for HPC Centers: Is It Worth the Effort?", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6310 LNCS, 2010, pp. 118–130, ISBN: 3642162320. DOI: 10.1007/978-3-642-16233-6_13. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16233-6%7B%5C_%7D13.

[12] J. Darlington, M. Ghanem, Y. Guo, and H. W. To, "Guided resource organisation in heterogeneous parallel computing", ... *of High Performance Computing*, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.4309%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf%7B%5C%%7D5Cnpapers2://publication/uuid/A8639BFB-B08A-46EA-A115-EF761321B949.

[13] D. M. Kunzman and L. V. Kalé, "Programming heterogeneous systems", *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 2061–2064, 2011. DOI: 10.1109/IPDPS.2011.377.

[14] S. Ashkiani, N. Amenta, and J. D. Owens, "Parallel approaches to the string matching problem on the GPU", *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, vol. 11-13-July, pp. 275–285, 2016. DOI: 10.1145/2935764.2935800.

[15] S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio, "The String Matching Algorithms Research Tool", *Proceedings of the Prague Stringology Conference 2016*, pp. 99–113, 2016. [Online]. Available: http://www.stringology.org.

[16] S. Faro, "Exact Online String Matching Bibliography", 2016. arXiv: 1605.05067. [Online]. Available: http://arxiv.org/abs/1605.05067.

[17] J. H. Morris and V. R. Pratt, "A linear pattern-matching algorithm", 1970.

[18] A. C.-C. Yao, "The Complexity of Pattern Matching for a Random String", *SIAM Journal on Computing*, 1979, ISSN: 0097-5397. DOI: 10.1137/0208029.

[19] S. Faro and T. Lecroq, *The Exact String Matching Problem: a Comprehensive Experimental Evaluation*, 2010. arXiv: 1012.2547 [cs.DS].

[20] ——, "The exact online string matching problem", *ACM Computing Surveys*, vol. 45, no. 2, pp. 1–42, Feb. 2013, ISSN: 0360-0300. DOI: 10.1145/2431211.2431212. [Online]. Available: https://dl.acm.org/doi/10.1145/2431211.2431212.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms - Selected Solutions*. 2009, ISBN: 0262033844. DOI: 10.2307/2583667. arXiv: arXiv:1011.1669v3.

[22] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977, ISSN: 0097-5397. DOI: 10.1137/0206024.

[23] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977, ISSN: 15577317. DOI: 10.1145/359842.359859.

[24] C. and Charras and T. Lecroq, "Handbook of Exact String Matching Algorithms", p. 238, 2004.

[25] R. N. Horspool, "Practical fast searching in strings", *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980, ISSN: 1097024X. DOI: 10.1002/spe.4380100608.

[26] Z. Galil and J. Seiferas, "Time-space-optimal string matching", in *Proceedings of the Annual ACM Symposium on Theory of Computing*, ser. STOC '81, New York, NY, USA: Association for Computing Machinery, 1981, pp. 106–113, ISBN: 0897910419. DOI: 10.1145/800076.802463. [Online]. Available: https://doi.org/10.1145/800076.802463.

[27] A. Apostolico and R. Giancarlo, "Boyer-Moore-Galil String Searching Strategies Revisited.", *SIAM Journal on Computing*, vol. 15, no. 1, pp. 98–105, 1986, ISSN: 00975397. DOI: 10.1137/0215007.

[28] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms", *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, Mar. 1987, ISSN: 0018-8646. DOI: 10.1147/rd.312.0249. [Online]. Available: http://ieeexplore.ieee.org/document/5390135/.

[29] Zhu Rui Feng and T. Takaoka, "on Improving the Average Case of the Boyer-Moore String Matching Algorithm.", *Journal of information processing*, vol. 10, no. 3, pp. 173–177, 1987, ISSN: 03876101.

[30] D. M. Sunday, "A very fast substring search algorithm", *Communications of the ACM*, vol. 33, no. 8, pp. 132–142, Aug. 1990, ISSN: 15577317. DOI: 10.1145/79173.79184. [Online]. Available: http://portal.acm.org/citation.cfm?doid=79173.79184.

[31] A. Apostolico and M. Crochemore, "Optimal canonization of all substrings of a string", *Information and Computation*, vol. 95, no. 1, pp. 76–95, Nov. 1991, ISSN: 08905401. DOI: 10.1016/0890-5401(91)90016-U. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/089054019190016U.

[32] M. Crochemore and D. Perrin, "Two-way string-matching", *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 650–674, Jul. 1991, ISSN: 0004-5411. DOI: 10.1145/116825.116845. [Online]. Available: http://dl.acm.org/doi/10.1145/116825.116845.

[33] A. Hume and D. Sunday, "Fast string searching", *Software: Practice and Experience*, vol. 21, no. 11, pp. 1221–1248, Nov. 1991, ISSN: 00380644. DOI: 10.1002/spe.4380211105. [Online]. Available: http://doi.wiley.com/10.1002/spe.4380211105.

[34] L. Colussi, "Correctness and efficiency of pattern matching algorithms", *Information and Computation*, vol. 95, no. 2, pp. 225–251, Dec. 1991, ISSN: 08905401. DOI: 10.1016/0890-5401(91)90046-5. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0890540191900465.

[35] P. D. Smith, "Experiments with a very fast substring search algorithm", *Software: Practice and Experience*, vol. 21, no. 10, pp. 1065–1074, Oct. 1991, ISSN: 00380644. DOI: 10.1002/spe.4380211006. [Online]. Available: http://doi.wiley.com/10.1002/spe.4380211006.

[36] Z. Galil and R. Giancarlo, "On the Exact Complexity of String Matching: Upper Bounds", *SIAM Journal on Computing*, vol. 21, no. 3, pp. 407–437, Jun. 1992, ISSN: 0097-5397. DOI: 10.1137/0221028. [Online]. Available: http://epubs.siam.org/doi/10.1137/0221028.

[37] T. Raita, "Tuning the boyer-moore-horspool string searching algorithm", *Software: Practice and Experience*, vol. 22, no. 10, pp. 879–884, Oct. 1992, ISSN: 00380644. DOI: 10.1002/spe.4380221006. [Online]. Available: http://doi.wiley.com/10.1002/spe.4380221006.

[38] M. Crochemore, "String-matching on ordered alphabets", *Theoretical Computer Science*, vol. 92, no. 1, pp. 33–47, Jan. 1992, ISSN: 03043975. DOI: 10.1016/0304-3975(92)90134-2. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0304397592901342.

[39] A. Finkel and M. Jantzen, "STACS 92: 9th Annual Symposium on Theoretical Aspects of Computer Science Cachan, France, February 13-15, 1992 Proceedings", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1992, ISBN: 9783540552109.

[40] C. Hancart, "Une analyse en moyenne de l'algorithme de Morris et Pratt et de ses raffinements", *Théorie des Automates et Applications, Actes des 2e Journées Franco–Belges, D. Krob ed., Rouen, France*, no. s 99, p. 110, 1992.

[41] L. Colussi, “Fastest Pattern Matching in Strings”, *Journal of Algorithms*, vol. 16, no. 2, pp. 163–189, Mar. 1994, ISSN: 01966774. DOI: 10.1006/jagm.1994.1008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S019667748471008X.

[42] C. Charras, T. Lecro, and J. D. Pehoushek, “A very fast string matching algorithm for small alphabets and long patterns”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1448 LNCS, 1998, pp. 55–64, ISBN: 3540647392. DOI: 10.1007/bfb0030780. [Online]. Available: http://link.springer.com/10.1007/BFb0030780.

[43] T. Berry and S. Ravindran, “A Fast String Matching Algorithm and Experimental Results.”, in *Stringology*, 1999, pp. 16–28.

[44] M. Ahmed, M. Kaykobad, and R. A. Chowdhury, “A New String Matching Algorithm”, *International Journal of Computer Mathematics*, vol. 80, no. 7, pp. 825–834, Jul. 2003, ISSN: 0020-7160. DOI: 10.1080/0020716031000087113. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/0020716031000087113.

[45] D. Cantone and S. Faro, “Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm”, *Journal of Automata, Languages and Combinatorics*, vol. 10, no. 5/6, pp. 589–608, 2005.

[46] D. Cantone and S. Faro, “Searching for a substring with constant extra-space complexity”, in *Proc. of Third International Conference on Fun with algorithms*, 2004, pp. 118–131.

[47] S. S. Sheik, S. K. Aggarwal, A. Poddar, N. Balakrishnan, and K. Sekar, “A FAST Pattern Matching Algorithm”, *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 4, pp. 1251–1256, Jul. 2004, ISSN: 0095-2338. DOI: 10.1021/ci030463z. [Online]. Available: https://pubs.acs.org/doi/10.1021/ci030463z.

[48] F. Franek, C. G. Jennings, and W. F. W. W. F. Smyth, “A simple fast hybrid pattern-matching algorithm”, in *Journal of Discrete Algorithms*, 4, vol. 5, Dec. 2007, pp. 682–695. DOI: 10.1007/11496656_25. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1570866706001067%20http://link.springer.com/10.1007/11496656%7B%5C_%7D25.

[49] T. Lecroq, “Fast exact string matching algorithms”, *Information Processing Letters*, vol. 102, no. 6, pp. 229–235, Jun. 2007, ISSN: 00200190. DOI: 10.1016/j.ipl.2007.01.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0020019007000117.

[50] P. Kalsi, H. Peltola, and J. Tarhio, "Comparison of exact string matching algorithms for biological sequences", in *Communications in Computer and Information Science*, vol. 13, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 417–426, ISBN: 9783540705987. DOI: 10.1007/978-3-540-70600-7_31. [Online]. Available: http://link.springer.com/10.1007/978-3-540-70600-7%7B%5C_%7D31.

[51] T. Lecroq, "A variation on the Boyer-Moore algorithm", *Theoretical Computer Science*, vol. 92, no. 1, pp. 119–144, Jan. 1992, ISSN: 03043975. DOI: 10.1016/0304-3975(92)90139-7. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0304397592901397.

[52] I. Simon, "String matching algorithms and automata", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 812 LNCS, 1994, pp. 386–395, ISBN: 9783540581314. DOI: 10.1007/3-540-58131-6_61. [Online]. Available: http://link.springer.com/10.1007/3-540-58131-6%7B%5C_%7D61.

[53] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Speeding up two string-matching algorithms", *Algorithmica*, vol. 12, no. 4-5, pp. 247–267, Nov. 1994, ISSN: 0178-4617. DOI: 10.1007/BF01185427. [Online]. Available: http://link.springer.com/10.1007/BF01185427.

[54] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1725, 1999, pp. 295–310, ISBN: 354066694X. DOI: 10.1007/3-540-47849-3_18. [Online]. Available: http://link.springer.com/10.1007/3-540-47849-3%7B%5C_%7D18.

[55] C. Allauzen and M. Raffinot, "Simple optimal string matching algorithm (Extended Abstract)", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1848, 2000, pp. 364–374. DOI: 10.1007/3-540-45123-4_30. [Online]. Available: http://link.springer.com/10.1007/3-540-45123-4%7B%5C_%7D30.

[56] L. He, B. Fang, and J. Sui, "The wide window string matching algorithm", *Theoretical Computer Science*, vol. 332, no. 1-3, pp. 391–404, Feb. 2005, ISSN: 03043975. DOI: 10.1016/j.tcs.2004.12.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0304397504007881.

[57] C. Liu, Y. Wang, D. Liu, and D. Li, "Two Improved Single Pattern Matching Algorithms", in *16th International Conference on Artificial Reality and Telexistence–Workshops (ICAT'06)*, IEEE, 2006, pp. 419–422, ISBN: 0-7695-2754-X. DOI: 10.

1109/ICAT.2006.134. [Online]. Available: http://ieeexplore.ieee.org/document/4089285/.

[58] S. Faro and T. Lecroq, "Efficient variants of the backward-oracle-matching algorithm", *International Journal of Foundations of Computer Science*, vol. 20, no. 6, pp. 967–984, Dec. 2009, ISSN: 01290541. DOI: 10.1142/S0129054109006991. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0129054109006991.

[59] H. Fan, N. Yao, and H. Ma, "Fast Variants of the Backward-Oracle-Marching Algorithm", in *2009 Fourth International Conference on Internet Computing for Science and Engineering*, IEEE, Dec. 2009, pp. 56–59, ISBN: 978-1-4244-6754-9. DOI: 10.1109/ICICSE.2009.53. [Online]. Available: http://ieeexplore.ieee.org/document/5521633/.

[60] S. Faro and T. Lecroq, "A fast suffix automata based algorithm for exact online string matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7381 LNCS, 2012, pp. 149–158, ISBN: 9783642316050. DOI: 10.1007/978-3-642-31606-7_13. [Online]. Available: http://link.springer.com/10.1007/978-3-642-31606-7%7B%5C_%7D13.

[61] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching", in *Proceedings of the 12th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '89*, New York, New York, USA: ACM Press, 1989, pp. 168–175, ISBN: 0897913213. DOI: 10.1145/75334.75352. [Online]. Available: http://portal.acm.org/citation.cfm?doid=75334.75352.

[62] G. Navarro and M. Raffinot, "A bit-parallel approach to suffix automata: Fast extended string matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1448 LNCS, 1998, pp. 14–33, ISBN: 3540647392. DOI: 10.1007/bfb0030778. [Online]. Available: http://link.springer.com/10.1007/BFb0030778.

[63] ——, "Fast and flexible string matching by combining bit-parallelism and suffix automata", *Journal of Experimental Algorithmics*, vol. 5, 4–es, Dec. 2000, ISSN: 10846654. DOI: 10.1145/351827.384246. [Online]. Available: http://portal.acm.org/citation.cfm?doid=351827.384246.

[64] H. Peltola and J. Tarhio, "Alternative algorithms for bit-parallel string matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2857, 2003, pp. 80–94.

DOI: 10.1007/978-3-540-39984-1_7. [Online]. Available: http://link.springer.com/10.1007/978-3-540-39984-1%7B%5C_%7D7.

[65] J. Holub and B. Durian, "Fast variants of bit parallel approach to suffix automata", *Unpublished Lecture, University of Haifa*, 2005.

[66] K. Fredriksson and S. Grabowski, "Practical and optimal string matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3772 LNCS, 2005, pp. 376–387, ISBN: 3540297405. DOI: 10.1007/11575832_42. [Online]. Available: http://link.springer.com/10.1007/11575832%7B%5C_%7D42.

[67] M. O. Külekci, "A method to overcome computer word size limitation in bit-parallel pattern matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5369 LNCS, 2008, pp. 496–506, ISBN: 3540921818. DOI: 10.1007/978-3-540-92182-0_45. [Online]. Available: http://link.springer.com/10.1007/978-3-540-92182-0%7B%5C_%7D45.

[68] B. Ďurian, J. Holub, H. Peltola, and J. Tarhio, "Tuning BNDM with q -Grams", in *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, Philadelphia, PA: Society for Industrial and Applied Mathematics, Jan. 2009, pp. 29–37. DOI: 10.1137/1.9781611972894.3. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972894.3.

[69] G. Zhang, E. Zhu, L. Mao, and M. Yin, "A bit-parallel exact string matching algorithm for small alphabet", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5598 LNCS, 2009, pp. 336–345, ISBN: 3642022693. DOI: 10.1007/978-3-642-02270-8_34. [Online]. Available: http://link.springer.com/10.1007/978-3-642-02270-8%7B%5C_%7D34.

[70] B. Ďurian, H. Peltola, L. Salmela, and J. Tarhio, "Bit-parallel search algorithms for long patterns", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6049 LNCS, 2010, pp. 129–140, ISBN: 3642131921. DOI: 10.1007/978-3-642-13193-6_12. [Online]. Available: http://link.springer.com/10.1007/978-3-642-13193-6%7B%5C_%7D12.

[71] D. Cantone, S. Faro, and E. Giaquinta, "A compact representation of nondeterministic (suffix) automata for the bit-parallel approach", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6129 LNCS, 2010, pp. 288–298, ISBN: 3642135080.

DOI: 10.1007/978-3-642-13509-5_26. [Online]. Available: http://link.springer.com/10.1007/978-3-642-13509-5%7B%5C_%7D26.

[72] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014.

[73] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972, ISSN: 00189340. DOI: 10.1109/TC.1972.5009071.

[74] *136 GPU-Accelerated Supercomputers Feature in TOP500 | NVIDIA Blog*. [Online]. Available: https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/ (visited on 05/21/2020).

[75] W.-c. Feng and K. Cameron, "The green500 list: Encouraging sustainable supercomputing", *Computer*, vol. 40, no. 12, pp. 50–55, 2007.

[76] J. Peng and H. Chen, "CUgrep: A GPU-based high performance multi-string matching system", in *Proceedings of the 2010 2nd International Conference on Future Computer and Communication, ICFCC 2010*, vol. 1, 2010, pp. V1–77–V1–81, ISBN: 9781424458226. DOI: 10.1109/ICFCC.2010.5497832.

[77] J. Zhou, H. An, X. Li, M. Xu, and W. Zhou, "Implementation of String Match Algorithm BMH on GPU Using CUDA", *Energy Procedia*, vol. 13, pp. 1853–1861, Jan. 2011, ISSN: 18766102. DOI: 10.1016/j.egypro.2011.11.261. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1876610211030773.

[78] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye, "Improving CUDASW, a parallelization of smith-waterman for CUDA enabled devices", *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 490–501, 2011. DOI: 10.1109/IPDPS.2011.182.

[79] N. P. Tran, M. Lee, S. Hong, and M. Shin, "Memory efficient parallelization for Aho-Corasick algorithm on a GPU", in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications, HPCC-2012 - 9th IEEE International Conference on Embedded Software and Systems, ICESS-2012*, 2012, ISBN: 9780769547497. DOI: 10.1109/HPCC.2012.65.

[80] A. Rasool and N. Khare, "Parallelization of KMP String Matching Algorithm on Different SIMD Architectures: Multi-Core and GPGPUs", *International Journal of Computer Applications*, vol. 49, no. 11, pp. 26–28, Jul. 2012, ISSN: 09758887. DOI: 10.5120/7672-0963. [Online]. Available: http://research.ijcaonline.org/volume49/number11/pxc3880963.pdf.

[81] C. Pungila and V. Negru, "A Highly-Efficient Memory-Compression Approach for GPU-Accelerated Virus Signature Matching", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, vol. 7483 LNCS, 2012, pp. 354–369, ISBN: 9783642333828. DOI: 10.1007/978-3-642-33383-5_22. [Online]. Available: http://link.springer.com/10.1007/978-3-642-33383-5%7B%5C_%7D22.

[82] K. K. Yong and E. K. Karuppiah, "Hash match on GPU", in *2013 IEEE Conference on Open Systems, ICOS 2013*, 2013, ISBN: 9781479902859. DOI: 10.1109/ICOS.2013.6735065.

[83] N. P. Tran and M. Lee, "High performance string matching for security applications", in *Proceedings - International Conference on ICT for Smart Society 2013: "Think Ecosystem Act Convergence", ICISS 2013*, IEEE, 2013, pp. 1–5, ISBN: 9781479901456. DOI: 10.1109/ICTSS.2013.6588052.

[84] S. P. Adey, "GPU Accelerated Pattern Matching Algorithm for DNA Sequences to Detect Cancer using CUDA Dissertation", *Coll. Eng. Pune*, 2013.

[85] X. Zha and S. Sahni, "GPU-to-GPU and host-to-host multipattern string matching on a GPU", *IEEE Transactions on Computers*, 2013, ISSN: 00189340. DOI: 10.1109/TC.2012.61.

[86] K. Xu, W. Cui, Y. Hu, and L. Guo, "Bit-parallel multiple approximate string matching based on GPU", *Procedia Computer Science*, vol. 17, pp. 523–529, 2013, ISSN: 18770509. DOI: 10.1016/j.procs.2013.05.067. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050913002007.

[87] X. Bellekens, R. C. Atkinson, C. Renfrew, T. Kirkham, I. Andonovic, R. C. Atkinson, C. Renfrew, and T. Kirkham, "Investigation of GPU-based Pattern Matching", in *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013) (PGNet2013)*, 2013, p. 5, ISBN: 9781902560274. [Online]. Available: http://www.cms.livjm.ac.uk/pgnet2013/Proceedings/papers/1569777259.pdf.

[88] V. Nagaveni and G. T. Raju, "Various String Matching Algorithms for DNA Sequences to Detect Breast Cancer using CUDA Processors", *Int J Eng Technol*, vol. 14, no. 3, pp. 42–47, 2014.

[89] C. S. Kouzinopoulos, P. D. Michailidis, and K. G. Margaritis, "Multiple string matching on a GPU using CUDA", *Scalable Computing*, 2015, ISSN: 18951767. DOI: 10.12694/scpe.v16i2.1085.

[90] J. Sharma and M. Singh, "CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU", *International Journal of Computer Network and Information Security*, 2015, ISSN: 20749090. DOI: 10.5815/ijcnis.2015.10.08.

[91] C. L. Lee, Y. S. Lin, and Y. C. Chen, "A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection", *PLoS ONE*, vol. 10, no. 10, 2015, ISSN: 19326203. DOI: 10.1371/journal.pone.0139301.

[92] Y. Mitani, F. Ino, and K. Hagihara, "Parallelizing exact and approximate string matching via inclusive scan on a GPU", *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1989–2002, 2017, ISSN: 10459219. DOI: 10.1109/TPDS.2016.2645222.

[93] R. Ramos-Frías and M. Vargas-Lombardo, "A Review of String Matching Algorithms and Recent Implementations using GPU", *International Journal of Security and Its Applications*, 2017, ISSN: 17389976. DOI: 10.14257/ijsia.2017.11.6.06.

[94] M. J. Quinn, "Parallel Programming in C with MPI and OpenMP", *Star*, 2003.

[95] K. Hwang and N. Jotwani, *Advanced computer architecture, 3e*. McGraw-Hill Education, 2016.

[96] M. D. Adams, S. E. Celniker, and R. A. Holt, *The genome sequence of Drosophila melanogaster*, 2000. DOI: 10.1126/science.287.5461.2185.

[97] D. Franklin, "Nvidia jetson tx2 delivers twice the intelligence to the edge", *NVIDIA Accelerated Computing| Parallel Forall*, 2017.