SWIFT -A Performance Accelerated Optimized String Matching Algorithm for Nvidia GPUs

Sourabh S. Shenoy, Supriya Nayak U. and B. Neelima
Department of Computer Science and Engineering
NMAM Institute of Science and Technology, Nitte
Karkala Taluq, Udupi District, Karnataka, India.
Email: sourabhsshenoy@gmail.com; supriyanayakudma@gmail.com; reddy_neelima@yahoo.com

Abstract—**This paper presents a study of exact string matching algorithms and their performance behavior when executed on dynamic parallelism enabled Kepler Graphics Processing Unit (GPU) by Nvidia. The algorithms considered in this paper are Quick search (QS), Horspool (HP), and Brute force (BF) string matching. Their efficient implementation on Kepler gives a remarkable improvement over their respective multi-core CPU and generic GPU implementations. In addition, the proposed work further optimizes the algorithm by exploiting the fundamental architectural aspects of the GPU like the memory hierarchy, lightweight threads and avoiding strided access. The optimization associated with the memory is called Binning and the one using memory alignment is named Chunking. As a result of the significant boost obtained by extracting the most out of these features, the newer methods are named as SWIFT. SWIFT algorithms give performance benefit of about 1.7X on an average and up to 5X in some cases, which will be discussed in the paper. Also, the paper proposes a hybrid algorithm that employs both regular GPU implementation and SWIFT based on a predefined condition that gives benefit for all pattern sizes. The experimental results for different pattern sizes using benchmark datasets are presented in the paper.**

Keywords—**Exact String Matching; Heterogeneous Architecture; Graphics Processing Unit (GPU); Kepler Architecture; Brute Force String Matching Algorithm; Quick Search String Matching Algorithm; Horspool String Matching Algorithm**

I. Introduction

The world is changing at an unprecedented pace. Technology today produces an enormous amount of data, whose storage and timely retrieval plays a crucial role in any field. String searching algorithms have been in practice for decades now. They play a key role in various applications designed to disentangle real world problems, including plagiarism detection, search engines, intrusion detection systems, spell checkers, spam identifiers, DNA sequencers etc. Due to the increasing demand for speed in various fields of computation, heterogeneous computing devices like multi-core Central Processing Unit (CPU) and Graphics Processing Unit (GPU) have sprouted up. GPUs, although traditionally used for graphics processing, have lately found usage in general purpose computing (GPGPU). GPU, unlike the traditional CPU, has a hierarchy of memory and can manage over thousands of threads. These, if efficiently utilized can parallelize and expedite the task, thus reducing the overall execution time.

The proposed work in this paper identifies Quick search, Horspool, and Brute force exact string matching algorithms to be ported onto Kepler GPU from Nvidia. The performance of these algorithms is enhanced by using the dynamic parallelism feature, whereby the parent thread spawns child threads for those pattern and text substring pairs, which have their respective first and last characters matching. Therefore, only a few threads from the parent kernel are likely to launch the child kernels. This behavior maps exactly to the template of the dynamic parallelism feature and was the key motivation to test the behavior of these algorithms. The enhanced Quick search, Horspool and Brute force exact string matching algorithms are named as SWIFT in this paper.

For very small patterns, SWIFT performs poorly, due to the overhead of launching multiple child kernels. More precisely, in this case, the amount of work done by child kernel is annulled by the overhead involved in calling the child kernel by parent kernel. When tested with large patterns, SWIFT gave significant performance improvement for the algorithms considered. Applications such as plagiarism checking, intrusion detection system and bioinformatics involve practically very large patterns. Hence, the usage of dynamic parallelism for such large patterns to improve the existing algorithms performance is warranted.

The major contribution of this work is identifying the usage of dynamic parallelism feature in improving the performance of exact string matching algorithms for large patterns. Further, the paper also sheds light on optimizations one can achieve by cleverly using the available GPU memory hierarchy and ensuring aligned access to data. The basic SWIFT algorithm gives 1.2X to 1.4X speedup in performance. After employing the optimizations discussed above, the performance goes up to 1.7X in general. Rest of the paper is organized as follows: Section II gives the background information about the string matching algorithms and GPU architecture emphasizing on dynamic parallelism. Next, Section III elaborates the proposed methodology and the optimization techniques used. Section IV summarizes the experimental setup. Section V discusses the results and observation. Conclusions and Future work are given in Section VI. Finally, challenges faced are mentioned in Section VII.

II. BACKGROUND

This section gives the background details of GPU architecture, dynamic parallelism and evolution of exact string matching.

A. GPU Architecture [1]

CPU and GPU work in tandem and are considered a master- slave relationship. GPU is a coprocessor to the Central Processing Unit (CPU) and has been primarily used for an exhilarating gaming experience since its origin. However, with the growing demand for faster applications, GPU was recently extended to general purpose computations. Thus, resulting in a wide range of optimizations in algorithms, which were until then considered to be already optimized to the level of saturation. In order to get the best out of a GPU, the compute intensive sections of any program are offloaded to it. These portions of code are executed by meticulously making use of a large number of threads that a GPU is seamlessly capable of running simultaneously and by using different kinds of memory. CUDA is a programming model developed by the popular GPU manufacturer, Nvidia to design general purpose computations on their GPUs.

A GPU contains execution units called cores or processors. Their number ranges from hundreds to a few thousands, which is significantly large compared to the number of cores in a CPU. The cores are grouped to form streaming multiprocessors (SM). During the execution of a program, each SM gets exactly one instruction and this instruction is carried out by every core in that SM but with different data i.e. Single Instruction Multiple Data (SIMD). The total number of threads allowed in a GPU varies with its architecture. Essentially, a GPU thread is lightweight when compared to that in CPU. Threads in a GPU can be grouped into blocks, which are further grouped into a grid .It is also possible to have threads and blocks in three dimensions. Speaking of the memory, GPU has a sequence of it, with each having varying access rates. The sequence includes global memory, shared memory, texture memory, constant memory, local memory and registers. When offloading the portions of code from CPU to GPU, the required data is initially placed in the global memory. From here, the data that is not liable to change could be placed in constant memory. Moreover, every block has its own shared memory and every thread in the block is associated its own register and local memory. Figure 1 shows the outline of CUDA programming model.

In 2012, Nvidia released the Kepler G110 architecture. It was introduced with many improvements and new features to support parallelism in a broader range of applications. One of the most important features provided by this architecture is Dynamic Parallelism.

B. Dynamic Parallelism [2] [3]

Dynamic parallelism is one of the key features available on Nvidia GPUs with compute capability 3.5 and higher wherein a GPU thread can further launch a set of threads without any involvement of the CPU. It provides support for recursion and dynamic workload balancing. The essence lies in the fact that it adds to the capability for the GPU to generate new tasks for itself, perform synchronization on results and control the scheduling of the task via dedicated, accelerated hardware paths, all without involving the CPU. The general scenario of dynamic parallelism is shown in figure 2 .`CPU_func()` is a function on the CPU, which offloads some compute intensive task to GPU. This task (or kernel) on GPU, `GPU_krnl1()` in turn calls another kernel, `GPU_krnl2()`. As a consequence, `GPU_krnl1()` is called the Parent kernel and `GPU_krnl2()` becomes the Child kernel. Threads in parent kernel spawn set of threads through child kernel. As soon as a child kernel completes its task, control returns back to the parent kernel thread that spawned it.

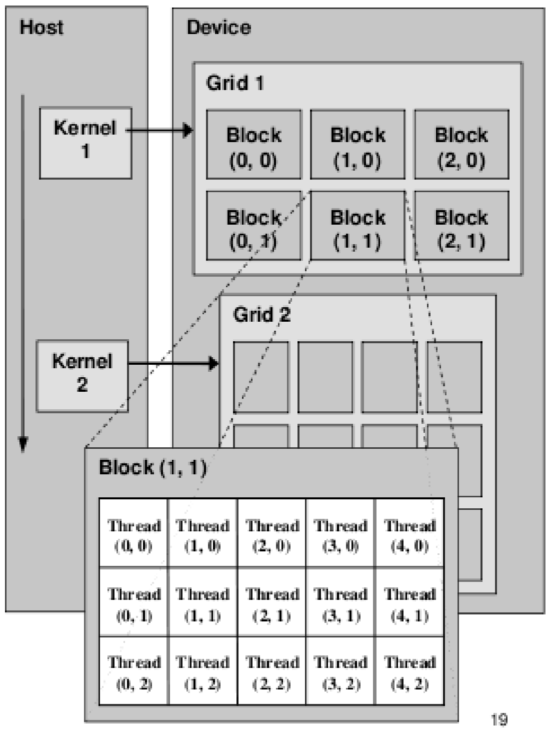


Fig. 1. CUDA Programming model.

The work proposed in this paper makes use of this feature along with an efficient usage of already existing memory hierarchy.

C. String Matching Algorithms

A string matching algorithm aims at finding all the occurrences of a pattern in a given large text. It can be broadly classified into exact and inexact string matching algorithms. Given a text T and pattern P, n stands for the length of text and m for the length of pattern. While searching a pattern in the text, a search window is that portion of the

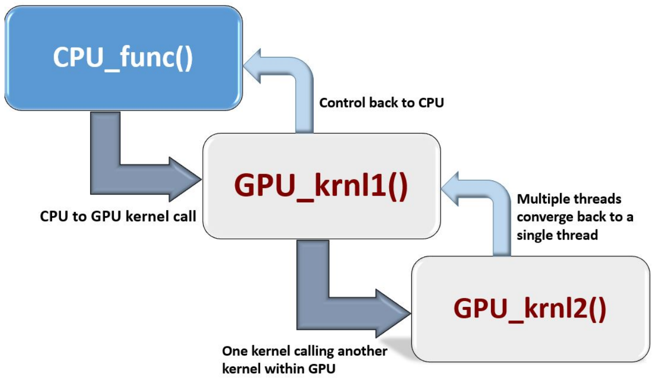

CPU_func()
Control back to CPU
GPU_krnl1()
CPU to GPU kernel call
Multiple threads converge back to a single thread
GPU_krnl2()
One kernel calling another kernel within GPU

Fig. 2. Dynamic Parallelism.

text that is currently compared against the pattern.

Exact string matching algorithm [4] locates all the substrings (each of length m) in the given text that match the pattern exactly. Rabin-Karp [5], Knuth-Morris-Pratt (KMP) [6], Boyer Moore [7], and Horspool [8] are few of the popular algorithms in this category.

Inexact or Approximate or Fuzzy string matching algorithms [9] find the sub-strings (not necessarily of length m) in the text that nearly match the given pattern. Shift-Or, Wu-Manber, Baeza-Yates- Perleberg algorithms [10] etc are a few to name. Since this paper considers exact string matching algorithms, their brief evolution is discussed below.

The Brute force (BF) exact string matching algorithm is a naive way of searching a pattern in the text. The search begins by comparing the first characters of both pattern and the given text. This comparison continues until a mismatch is encountered. In the case of a mismatch, the pattern is shifted one character to the right and compared with the corresponding characters in the text. The limitation of brute force method was the amount of time taken for the character comparisons between pattern and text substrings. To overcome this, Richard M. K. and Karp R. M. [5] developed Rabin-Karp algorithm that applies a hash function to pattern and substrings in the given text. If the hash values of the pattern and substring match, one-to-one comparisons take place between their corresponding characters. Meanwhile, Knuth Morris Pratt algorithm (KMP) [6] uses an approach of tracing and shifting the pattern against the given text. The authors suggest constructing a table called the Partial match or the Failure function table for the pattern, as a part of pattern pre-processing. Within each prefix of the pattern, the longest suffix, which is a prefix as well, is looked for. The length of this suffix is later recorded in the table.

Boyer Moore algorithm (BM) [7] is yet another popular string matching algorithm that pre-processes the pattern and compares the characters right to left. There are two rules governing the amount by which the pattern, in the case of a mismatch, has to be moved past in the given text. They are Bad character table and Good suffix rule. A bad character table contains an entry for every character in the pattern. This entry indicates the first occurrence of the character in the pattern from the right end. According to good suffix rule, on a mismatch in the middle of the search window, if the substring matched so far occurs again in the pattern, they should be aligned accordingly provided the character that resulted in the mismatch does not precede this occurrence in the pattern. If the substring does not repeat in the pattern, it could have a suffix that matches a prefix of the pattern. On a mismatch, one has two options. First, look for the occurrence of the bad letter (one that led to the mismatch) in the text. Second, find the next occurrence of the good suffix (matched part) in the pattern and align it against the text accordingly. Consider the following example

Text- A T G A C C T A C

Pattern- G C T A C

Searching phase

A T **G** A C C T A C

G C **T** A C

The above example A C is the good suffix and T is the bad character in the pattern that led to a mismatch. One can either look for the occurrence of T in the text or find the next instance of A C in the pattern (not preceded by T) to be matched against A C in the text. The option that results in a greater jump has to be opted for. Many optimizations have been proposed since the origin of this algorithm and some of them are as follows.

Further, Boyer Moore Horspool algorithm (HP) [7] is similar to BM but pre-processes and searches the pattern slightly differently. It constructs the bad character table by initially setting all the entries to m. Later, barring the last character, it scans the pattern, making note of rightmost positions of every character. In addition, it does not make use of the good suffix rule. A variant of HP is known as Raita algorithm [11]. Raita varies from HP in the way pattern is searched for but the construction of bad character table remains the same. Turbo Boyer Moore algorithm [12] was proposed to improve the worst-case performance of the Boyer Moore algorithm. It gains better speedup at a cost of constant additional space. This space is required to store a value, say MemVal, which equals the number of character matches between pattern and the text search window in the previous search iteration. In continuation, Zhu-Takaoka algorithm [13] aims to improve average case of the Boyer Moore algorithm.

To improvise the algorithm further, Sunday D. M. proposed the use of the character immediately following the search window and named it Quick search [14]. Quick search also constructs a table in the preprocessing stage but it differs t

TABLE I
QSHIFT TABLE

Characters	A	C	G	T
Shift	2	1	8	3

from the one in Boyer Moore algorithm. The table, say qShift, contains all the characters in the given alphabet, each of them associated with a value depending on its existence in the pattern. If a character occurs frequently in the pattern, then its most recent position from the right end becomes shift value i.e. if m is the length of the pattern and i, the index of the rightmost occurrence of character c, then qShift[c] =m-i. For letters not in the pattern, m+1 will be the shift value. Searching phase involves character-to-character comparisons from either the right end or the left. On a mismatch, the pattern is shifted by the value associated with the letter in the text that is immediately after the search window. Therefore, in case a letter does not exist at all in the pattern, it is shifted completely past the search window with an assurance of not losing any of its possible matches in the text. Consider the following example
Text- A C T G A C A G T A C A C T A C C A
Pattern- A C A C T A C
Text length, n= 18
Pattern length, m= 7
The qShift for this is given in Table 1.
Searching phase-
Character comparisons here begin from the right end.

A C T G A C **A** G T A C A C T A C C A
A C A C T A **C**
Shift the pattern by shift value of next character in the text i.e. G.

A C T G A C A G T A C A C T **A** C C A
A C A C T A **C**
Mismatch again. Shift by 1.

A C T G A C A G T **A C A C T A C** C A Match!
A C A C T A C

Furthermore, Boyer Moore Smith algorithm [15] was proposed in order to shift the pattern by maximum possible value. At times, the shift achieved through the rightmost character of the search window is smaller than the shift corresponding to the character to the immediate right of the window. In such cases, this algorithm helps in minimizing the number of character comparisons. To enhance the performance further, Berry T. and Ravindran S. [16] proposed an algorithm that merges the concepts of quick search and Zhu-Takaoka algorithms. Pre-processing of the pattern involves building a matrix that contains values for the character pairs say, ab that occur to the immediate right of the search window in the text.

III. PROPOSED METHOD

This paper presents an implementation of three well-known exact string matching algorithms, Quick Search, Horspool, and Brute Force using dynamic parallelism, with some modification. It is observed that although the performance of SWIFT is bad for small patterns, it improves significantly for large patterns. This anomaly can be attributed to the fact that the overhead involved in launching the child kernel [17] is amortized in the case of a large pattern and thus, a significant performance gain is observed.

Algorithm 1 gives the pseudo code for Quick search and Horspool (SWIFT) string matching algorithms using dynamic parallelism. The pseudo code for both Quick Search and Horspool algorithms is essentially the same, apart from the Pre-process-pattern function. The Pre-process function in the Quick Search algorithm facilitates bigger jumps as compared to the Horspool algorithm and skips unnecessary comparisons, without losing accuracy.

Algorithm 1 Quick Search, HP String Matching Pseudo Code-SWIFT with Dynamic Parallelism.

```
1:  function MAIN
2:      Pre-process-pattern()
3:      Calculate-Shifts-Text()
4:      Initialize all locations to 1. (Found)
5:      Call Search Kernel
6:  end function
7:  function SEARCH
8:      Check if first and last characters match.
9:      Yes: Call Compare Kernel with m threads
10:      ▷ // Dynamically spawn m threads, where m=pattern length
11:     No: Set corresponding location value to 0.    ▷ //Not found
12: end function
13: function COMPARE
14:     Each thread checks for one location. If found
15:     Yes: Do nothing
16:     No: Set corresponding location value to 0    ▷ //Not found
17: end function
```

Apart from the gain obtained through dynamic parallelism, SWIFT algorithms obtain further performance improvement over their respective generic GPU implementations using the following three approaches:

• Binning: Use of shared memory in parent kernel while using global memory in the child kernel. Use of constant memory for the pattern.

• Chunking: Avoiding strided access and accessing aligned memory, thus making better use of spatial locality.

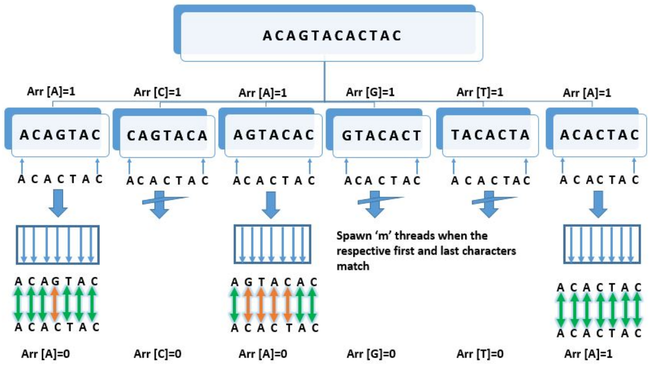
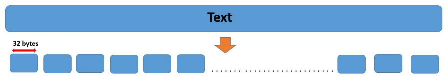
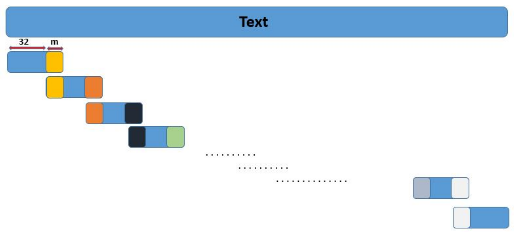


Fig. 3. Working of SWIFT algorithms for text A C A G T A C A C T A C and pattern A C A C T A C.



(a) Dividing text into blocks of 32 bytes each without overlap.



(b) Dividing text into blocks of size 32+m (Binning).

Fig. 4. Binning text into shared data blocks.

- Hybrid Algorithm that uses dynamic parallelism based on a threshold pattern size.

Brute Force, Quick Search and HP string matching algorithms use sequential character comparisons to check if the text substring matches with the pattern. The proposed SWIFT algorithm is compared with the generic GPU versions of the aforementioned algorithms. The generic GPU implementations are from Raymond Tay's implementation [18]. The paper further optimized it to make use of shared memory, in order to have a fair comparison. In SWIFT algorithm, each thread spawns multiple threads to perform the one-to-one character comparisons in parallel, only when the respective first and last characters match. This is illustrated in figure 3. In essence, this is parallelism within parallelism. The new thread id has to be calculated within the child kernel and mapped to the proper location. The index in the text will be the sum of the parent threads id and child threads id.

As shown in figure 3, the text A C A G T A C A C T A C is subdivided into substrings of length equaling the pattern length m = 7. In addition, there is an array Arr[] that has elements corresponding to each of the substrings. These elements set themselves to 1 if their corresponding substrings match with the given pattern A C A C T A C. The other entries retain their values they were initialized to in the beginning. SWIFT algorithms were further optimized using the following approaches:

1) Enhancement involving memory hierarchy (Binning): To maintain memory consistency, local and shared memory of the parent kernel cannot be passed as arguments to the newly spawned child kernels. This problem is circumvented by using shared memory in the parent kernel and global memory in the child kernel. Although the text can be divided into blocks of fixed length, without any overlap, it results in missing out those patterns that fall in between the boundary of the blocks. To avoid missing such patterns, it is necessary to overlap the blocks. Hence, the size of the blocks is chosen to be `WARP_SIZE + PATTERN_LENGTH`. We call this approach binning. Figure 4a and 4b elucidates how the text is binned into blocks of shared memory, such that every occurrence of the pattern is covered. Further, constant memory is used to store the pattern, since it does not change.

In the case of a first and last character match, a pointer that holds the position of the first character in the text is passed to the child kernel. By computing the new thread Ids of the child threads and adding it to the pointer, the threads can match the appropriate characters in the text and pattern. With this approach, an additional performance gain of about 1.1X on an average is obtained over the improvement obtained by the regular SWIFT algorithm and up to 1.55X improvement over Raymond Tays GPU implementation. An alternate approach to achieve the same would be to make a copy of the shared data in a device variable and pass it to the child kernel. However, that entails involves extra overhead. Figure 4 depicts how the memory was divided into blocks.

2) Exploit spatial locality by memory alignment (Chunking): Device Memory in CUDA is allocated in terms of 256-byte memory segments [19]. Access to global memory must be aligned to get the maximum benefits. Moreover, global memory is accessed in multiples of 32, 64 or 128-byte transactions, by the GPU memory bus. Hence, the first byte of a 128-byte transaction must start from a multiple of 128. Since there are 32 threads per warp and transactions are of 128 bytes, each thread reads 4 bytes. Thus, each thread compares a chunk of 4 bytes instead of just one character. In essence, now m/4 threads are launched in the child kernel instead of m, where m represents the pattern length. Each thread in the child kernel will do four comparisons:

$$Id * 4 + 0,\ Id * 4 + 1,\ Id * 4 + 2 \text{ and } Id * 4 + 3$$

When a warp reads 128 bytes from global memory and if the memory is aligned correctly, all reads will be from the

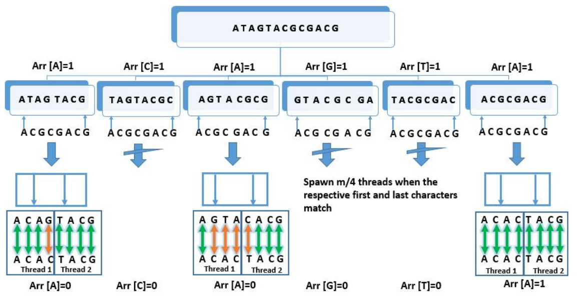


Fig. 5. Working of SWIFT-Chunking for text A T A G T A C G C A C G and pattern A C G C G A C G.

same block as memory transactions are performed by the GPU bus per warp. This is not a mandate, but not following this method would mean that data reads are scattered and the GPU will have to issue multiple transactions to fetch those blocks of data, leading to poor performance.

Using chunking led to a performance improvement of 1.2X 1.4X on an average, over the regular approach to SWIFT and an improvement of 1.6X to 1.7X over the generic GPU implementation.

Working of SWIFT with chunking is shown in figure 5, for text A T A G T A C G C G A C G and pattern A C G C G A C G. Since pattern length is 8, only 2 threads are launched here to do the comparison and set the corresponding result.

3) Hybrid Algorithm: A final enhancement would be to combine the best of both worlds i.e. a hybrid version that uses regular GPU code for pattern sizes that are below a threshold value and then the optimized SWIFT algorithms for pattern sizes above the threshold. From experiments carried out, by trial and error, the paper concludes that this shift occurs for pattern size of around 800. For smaller patterns, the generic GPU implementations using shared memory for text and constant memory for the pattern can be used. As the pattern size crosses the threshold limit, and if the compute capability of the device is greater than 3.5, the hybrid algorithm proposes to switch to SWIFT with chunking or SWIFT with binning. Doing so will also lead to a better maintainable and cleaner code and will yield the best performance for patterns of all sizes. Implementation of this algorithm is straightforward as it just aims to combine the two algorithms based on a threshold limit.

The algorithms using the above optimization techniques will be referred to as SWIFT-Binning, SWIFT-Chunking and Hybrid SWIFT respectively in the rest of this paper.

IV. EXPERIMENTAL SETUP

The experiments were performed using benchmark data such as The Project Gutenberg Edition of THE WORLD FACTBOOK 1992, that has an alphabet size of 256 (The ASCII set). Another benchmark named Bible, which uses common English letters and punctuations, is being used to analyze the algorithms. Yet another benchmark named E.coli genome structure, that has an alphabet size of 4 (A, C, G, T) is also used. A synthetic benchmark (named as Sample-Text) having random letters with no spaces in between is also considered in the experiments. The experiments were performed using CUDA platform on a Kepler machine with 2880 cores, at GPU Centre of Excellence (GCOE), IIT Bombay, which supports dynamic parallelism (Compute Capability 3.5).

V. RESULTS AND OBSERVATIONS

This section gives the results of string matching algorithms for the three algorithms with and without dynamic parallelism for the benchmark dataset described in the previous section with different pattern sizes. Each program was run 10 times and the average execution time was recorded. Our algorithms do not modify the preprocessing of the pattern or the way in which the parent kernel is launched. Hence, the CPU execution time and data transfer time are the same as that of the generic GPU algorithms that are being compared with. The paper reports the GPU execution time for all algorithms, as that is the major contributor for the overall runtime and the focus of this work.

Figure 6 illustrates the execution time in seconds for a pattern of length 500 for brute force, quick search and Horspool string matching algorithms with and without dynamic parallelism. It was observed that the performance of SWIFT algorithms was bad when the pattern sizes were less than 500. This is primarily because the overhead involved in calling the child kernel multiple times negates the performance gained by using dynamic parallelism. For the synthetic dataset Sample-Text, there is neither a gain nor a loss in the performance. However, for the benchmark datasets such as Gutenberg, Bible and E-Coli, the SWIFT algorithms perform poorly. In fact, it causes the performance to be worse than if it were not used in the first place. This drop in performance can be ameliorated by using the optimization techniques as discussed previously, and the performance gap can be bridged. However, then it would perhaps, not be desirable to use dynamic parallelism anymore since there is no evident benefit in exacerbating the execution time and subsequently improving it.

SWIFT algorithms provide better performance for pattern sizes 800 and above. Figure 7 depicts the execution time for patterns of length 1500 for Brute force, Quick search and Horspool algorithms with and without dynamic parallelism. As mentioned earlier, the Brute force, Quick Search and Horspool algorithms with dynamic parallelism are a part of SWIFT algorithm. In contrast to Figure6, it is seen that the SWIFT algorithms perform far better than their non-dynamic counterparts do. For the synthetic benchmark, Sample-Text, it is observed that SWIFT algorithm are up to 5X times better than the optimized shared memory GPU implementation as

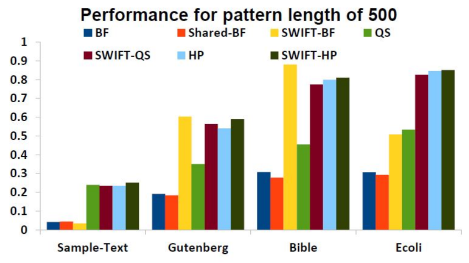


Fig. 6. Comparison of String Matching Algorithm for Pattern Length of 500.

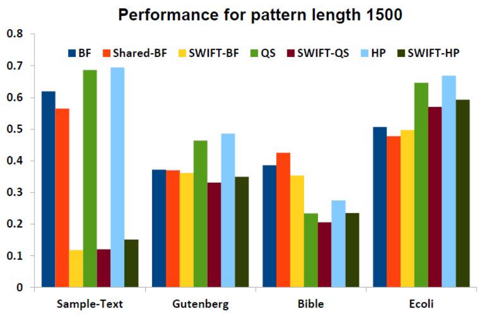


Fig. 7. Comparison of String Matching Algorithm for Pattern Length of 1500

well as the regular GPU implementation.

Interestingly, when the dynamic algorithms are pitted against each other, it is observed that SWIFT-BF performs equal to or sometimes even better than SWIFT-QS and SWIFT-HP algorithms. When the generic GPU and parallel CPU implementations of these algorithms are considered, Quick Search is the best performing among the three. Yet, when these algorithms are seen with respect to data parallel architectures such as GPU, the exploitation of architecture with the features of the algorithm leads to Brute force performing well. This very observation motivated us to study these algorithms for the GPU architecture in detail, the results of which are presented later in this paper.

Apart from making a comparative study of the different string matching algorithms, the paper also studies them individually with respect to the different optimization approaches discussed previously. The results are presented for Brute Force algorithm and its variants, namely Brute Force with Shared Memory, SWIFT-BF (Brute Force with Dynamic Parallelism), and SWIFT-BF-Binning (Brute Force with Dynamic Parallelism and Binning).

Figure 8 shows the results obtained in the case of Brute Force algorithm, for a pattern size of 1500 for various benchmark datasets. It can be observed that SWIFT-BF-Binning is faster than both its dynamic (SWIFT-BF) and optimized non-dynamic counterparts (Bruteforce Shared). The average speedup is around 1.1X more than the speedup

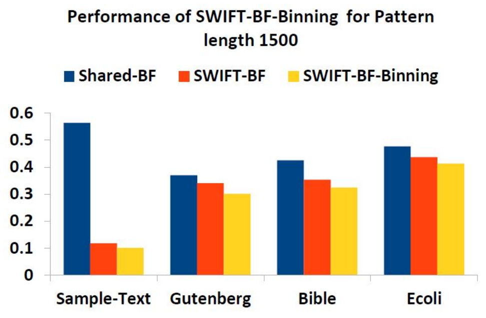


Fig. 8. Brute force algorithm using shared memory, dynamic parallelism and SWIFT-BF-Binning.

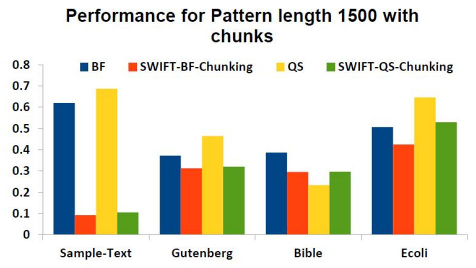


Fig. 9. Comparison of String Matching Algorithms for pattern size of 1500 using chunks.

obtained by basic SWIFT and about 1.5X to 1.6X when compared to the GPU implementation proposed by Raymond Tay. The speedup can be attributed to the usage of shared memory in the parent kernel and global memory in the child kernel. This way, threads in the same warp in the parent kernel share the data, thus reducing the need for multiple data fetches. This results in the speedup of performance. The fact that shared memory cannot be passed to the child kernel is overcome by using global memory in the child kernels and passing to it a pointer indicating the position of the possible match.

Figure 9 depicts the performance comparison between the variants of SWIFT that make use of memory alignment (SWIFT chunking). It is compared with the regular GPU versions of the respective algorithms. The performance gained in this case is 1.2X-1.4X more than the regular SWIFT algorithms and 1.6X 1.7X over the corresponding generic GPU implementations. This improvement is due to the fact that aligned access to data is made and it is ensured that each thread compares 4 bytes of data, thus ensuring that there are minimum transactions by the GPU memory bus and at the same time, maximum use of the each thread is made.

VI. CONCLUSIONS AND FUTURE WORK

String matching algorithms are important in view of applications like plagiarism detection, bioinformatics, intrusion detection systems and more recently, in text processing applications. The periodic, monumental increase in the amount of data available to be processed entails the need for algorithms that can scale quickly. Exact string matching is a fundamental problem in computing and hence is the subject of our study. This paper identifies the usage of Dynamic Parallelism for exact string matching algorithms like Brute force, Quick Search and Horspool, and proposed several optimizations to enhance the performance of the algorithm. The SWIFT algorithm gave a performance benefit of 1.2X to 1.4X without any optimization and about 1.7X with the optimizations. Lastly, the paper also proposed a hybrid algorithm, which gives speedup for patterns of all sizes.

Since Dynamic Parallelism involves launching multiple child kernels, care has to be taken to avoid race condition, which may lead to incorrect results. Code written using Dynamic Parallelism technique is cleaner and more maintainable in the long run.

As a future work, SWIFT algorithm and the optimization techniques for various architectural features can be tested for approximate string match and multi-pattern search and their variants. Multi-GPU variants of the algorithms are also expected to scale up well, due to the limited data transfer overhead. Lastly, it can also be tried in combination with the new shuffle instruction, which enables threads in the same warp to read each other's registers. However, it should be kept in mind that shuffle does not work in tandem with Dynamic Parallelism. CUDA Streams can also be considered to enable concurrent kernel launches to execute SWIFT algorithms.

VII. CHALLENGES

Not all algorithms are well suited for Dynamic Parallelism. Moreover, shared memory in the parent kernel cannot be passed to the child kernel, in favor of maintaining memory consistency. It is due to this that global memory had to be used in the child kernel and a pointer to indicate the global position of the substring in the text had to be passed to it. Lastly, Hybrid SWIFT will work well only on devices with Kepler GPU having compute capability 3.5. If not, it will act like the generic GPU implementation of the corresponding algorithm.

VIII. ACKNOWLEDGEMENT

The Authors would like to thank GPU Centre of Excellence (GCOE) at Indian Institute of Technology (IIT) - Bombay for giving GPU-based server access to carry out our work. We also thank Nvidia, India for the support extended through GPU Education Centre.

REFERENCES

[1] http://www.nvidia.in/object/gpu-computing-in.html

[2] www.devblogs.nvidia.com/parallelforall/cuda- dynamic-parallelism-api-principles/

[3] http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf

[4] Charras C., Lecroq T., Exact String Matching Algorithms, www-igm.univ-mlv.fr

[5] Karp R. M. and Rabin M. O., Efficient Randomized Pattern matching Algorithms, IBM J. Res. Dev., vol.31, n.2, pp.249-260,(1987).

[6] Knuth D. E. Morris J. H. and Pratt V. R., Fast Pattern Matching in Strings, SIAM Journal on Computing, vol. 6, n. 2, pp. 323-350, (1977).

[7] Boyer R. S. and Moore J. S., A Fast String Searching Algorithm, Commun. ACM, vol. 20, n. 10, pp. 762- 772, (1977).

[8] Horspool R. N., Practical Fast Searching in Strings, Softw. Pract. Exper, vol. 10, n. 6, pp. 501-506, (1980).

[9] Al-Khamaiseh K., and ALShagarin S., A Survey of String Matching Algorithms, Int. Journal of Engineering Research and Applications, vol. 4, Issue 7(Version 2), pp. 144-156, (2014).

[10] Michailidis P., Margaritis K., On-line approximate string searching algorithms: survey and experimental results, International Journal of Computer Mathematics, Intern. J. Computer Math., vol. 79(8), pp. 867888, (2002).

[11] Raita T., Tuning the Boyer-Moore-Horspool String Searching Algorithm, Softw. Pract. Exper., vol. 22, n. 10, pp. 879-884, (1992).

[12] Crochemore M., Czumaj A. Gasieniec L., Jarominek S., Lecroq T., Plandowski W. and Rytter W., Speeding Up Two String-Matching Algorithms, Algorithmica, vol. 12, n. 4-5, pp. 247-267, (1994).

[13] Zhu R. F. and T. Takaoka, On Improving the Average Case of the Boyer-Moore String Matching Algorithm, Journal of Information Processing, vol. 10, n. 3, pp. 173-177,(1987).

[14] Sunday D. M., A Very Fast Substring Search Algorithm, Commun. ACM, vol. 33, n. 8, pp. 132-142, (1990).

[15] Smith P. D., Experiments with a Very Fast Substring Search Algorithm, Softw., Pract. Exper., vol. 21, n. 10, pp. 1065-1074, (1991).

[16] Berry T. and Ravindran S., A Fast String Matching Algorithm and Experimental Results, Proceedings of the Prague Stringology Club Workshop 1999, Prague, Czech Republic, pp. 16-28, (1999).

[17] Wang J. and Sudhakar Y., Characterization and analysis of dynamic parallelism in unstructured GPU applications, International Symposium on Workload Characterization, pp. 51-60, (2014).

[18] Raymond T., A demonstration of Exact String Matching Algorithms in CUDA, code.google.com

[19] www.nvidia.com/content/GTC/documents/1029_GTC09.pdf