On Using a Graphics Processing Unit to Solve The Closest Substring Problem

Jon Calhoun[1,2], **Josh Graham**[1], **and Hai Jiang**[1]
[1]Dept. of Computer Science, Arkansas State University, Jonesboro, AR, US
[2]Dept. of Mathematics and Statistics, Arkansas State University, Jonesboro, AR, US

Abstract— *Finding a string that is close to another is a common dilemma in computational molecular biology and many other fields. The problem comes in two varieties; closest string (CSP), and closest substring (CSSP). The computational complexity increases exponentially as the data-set size increases. We make use of a massively parallel algorithm and the parallel nature of a graphics processing unit (GPU) in order to flatten the data-set size verses time curve and enable more applications to calculate results in reasonable time. In this paper we focus on CSSP and show that GPU devices can be used to reduce the time needed to find the closest substring. We examine an exact algorithm and extract independent parts in order to form a massively parallel interpretation of the sequential algorithm. We contribute a fast, exact, algorithm that can solve the CSSP much faster than sequential versions.*

Keywords: closest substring problem, GPU, CUDA

1. Introduction

Closest substring problem (CSSP) is a common open problem in many applications The closest substring problem was introduced in [3] and is a key theoretical open problem in applications such as antisense drug design, creating diagnostic probes, and creating universal PCR primers [1]. Many applications would benefit from a faster algorithm to find the closest substring. Some applications can accept approximations, but others need an exact result. We focus on using GPU devices to speed up an exact CSSP algorithm.

The CSSP is an NP-hard problem [3], and can be defined formally.

- Let $\sum$ be a fixed finite alphabet.
- Let s and s' be finite strings over $\sum$.
- Let $d(s,\, s')$ denote the Hamming Distance between s and s'.
- Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of strings each of length m. Find a center string c of length L minimizing d such that for each s_i in S there is a length L substring t_i of s_i with $d(c, t_i) \leq d$.

Solving the closest substring problem is moderately difficult, but solving it efficiently has proven very difficult. There is a tremendous amount of computation to be done and the process is compounded by not only being required to find a solution within the tolerance but to find the best solution. Solving the same problem in parallel can be taxing on one's logical skills. Additionally, solving the problem by hand is extremely time consuming for all but trivially sized data-sets. This paper shows that for applications needing an exact solution to the CSSP in a small amount of time can use GPUs to solve the problem, and achieve significant speedups.

This paper makes the following contributions:

- An efficient exact algorithm for computing the closest substring on GPU. We extract parallel sections of the computation in order to have it run efficiently.
- Logical configuration for launching the algorithm as a CUDA kernel.
- Experiment results that demonstrate the efficiency of the algorithm.

In Section 2, $Background$, we will explain the various technologies used. Following, Section 3.1, is a description of how the CSSP can be solved on a CPU using sequential code. After the CPU algorithm is clearly spelled out we turn to detail the inner workings of three parallel algorithms and their benefits and pitfalls in Section 3.2 - 3.3. In Section 4, $Experimental\, results$, we compare our GPU algorithms to the sequential one and discuss the performance increase. Section 5, $Related\, work$, discusses works that apply the GPU to the CSSP and related problems. A brief summary of what was learned from interpreting Section 4 can be found in our $Conclusion$, Section 6. Finally, in Section 7, $Future\, work$, we discuss optimizations that could be implemented that may increase performance.

2. Background

2.1 CSSP

The CSSP is a common problem in many areas. Particularly in computational biology, the CSP and CSSP have found numerous practical applications such as identifying regulatory motifs and approximate gene clusters, and in degenerate primer design [7]. The CSSP problem is much more elusive than the Closest String problem [6]. Many people have studied approximation algorithms for CSSP and there has even been work done on an evolutionary algorithm [8]. Here, we study an exact algorithm.

Many problems in molecular biology involve finding similar regions common to each sequence in a given set of DNA, RNA, or protein sequences. These problems find applications in locating binding sites and finding conserved regions in unaligned sequences, genetic drug target identification, designing genetic probes, universal PCR primer design, and, outside computational biology, in coding theory [6]. Such problems may be considered to be various generalizations of the common substring problem, allowing errors [6].

2.2 GPU

In 1999, NVIDIA created and marketed the worlds first graphics processing unit (GPU). Since then, there have been yearly breakthroughs in GPU technology. With the need to be able to make thousands of intense calculations per second for graphic applications. The architecture of a GPU is composed of thousands of processing units. In order for a algorithm to run efficiently on a GPU, the algorithm must be massively parallel. It is no surprise then, that the hardware and capabilities of GPUs has improved dramatically since their inception.

Programmers have been aware of the performance gain that could be achieved if a parallel portion of a program was executed on the GPU, but it was not until NVIDIA released there Compute Unified Device Architecture (CUDA) language that the job of programming GPUs became more intuitive. Before then to access the computational resources, a programmer had to cast his or her problem into native graphics operations so the computation could be launched through OpenGL or DirectX API calls [5].

2.3 CUDA

CUDA is NVIDIA's parallel computing language. It enables dramatic increases in computing performance by harnessing the power of the GPU. When NVIDIA introduced the GeForce 8800 GTX in November 2006 the CUDA architecture debuted. This architecture included several new components designed strictly for GPU computing and aimed to alleviate many of the limitations that prevented previous graphics processors from being legitimately useful for general-purpose computation [4].

CUDA is the most widely adopted programming platform for GPU development. CUDA applications running on NVIDIA graphics processors enjoy superior performance per dollar and performance per watt than implementations built exclusively on traditional central processing technologies [4]. In the CUDA programming model, GPUs which are called devices, execute highly parallel portions of an application, called kernels which are made up of many threads working cooperatively. CUDA permits the programmer to use different memory spaces explicitly. Examples of these different memory spaces include: global, shared, constant, and texture memory. Each space has its own performance advantages and penalties.

NVIDIA introduced the Fermi architecture recently. Fermi brings in many new capabilities. In this paper we make use of the increased maximum number of threads and blocks to perform more cooperative computations. We also benefit from the faster atomic actions and large memory present in Fermi graphic cards.

3. Algorithms

In order to ease the following discussions we define some terms.

- Let a *window* be any substring of length L from a given string.
- Let a *pitch* be special window from the first string that other notes are compared against.
- Let a *note* be the window from a string that is closest to the pitch.
- Let a *chord* be a set of notes, one per string, closest to a pitch.
- Let the *chord distance* be the sum of all note distances in a chord from the pitch.
- Let the *root* be the average of a chord.

3.1 CPU algorithm (CPU)

The strategy for finding the closest substring on the CPU consist of taking each pitch from the first string, comparing it against all windows, in all other strings and finding the closest window in each string. This window is then deemed a note and is part of the chord based on the pitch taken from the first string.

Given Figure 1, we want to find that something very close to *"gcc"* occurs in every string.

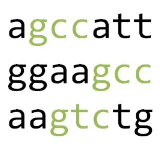

Fig. 1
EXAMPLE INPUT.

We do so by fixing a pitch in string 1 and comparing all others against it. Then we move to pitch 2 and so on. As demonstrated in Figure 2.

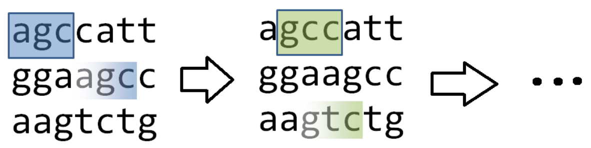

Fig. 2
ILLUSTRATION OF EXECUTION.

From the search we get the best chord, shown in Figure 3.

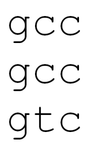

Fig. 3
BEST CHORD FROM DATA SET.

In Figure 4, we average the chord to determine the closest substring. In this example, the second character of each of the notes is c, c, and t respectively. In the averaging process c will be chosen because it occurs more often than any other character.

```
  gcc
  gcc
φ gtc
-----
  gcc
```

Fig. 4
EXAMPLE OF AVERAGING A CHORD TO FIND A ROOT NOTE.

To solve the problem more formally,

- Let $S = \{s_1, s_2, ..., s_n\}$ be the set of all strings.
- Let $\sum$ be a fixed finite alphabet.
- Let $S_i = \{s_{i_1}, s_{i_2}, ..., s_{i_n}\}$ be the set of all windows in $S_i \,|\, \forall\, s_{i_{j_k}} \in \sum$.
- Let d be the maximum distance.
- Let L be the length of the substrings.
- Let q be the number of windows in a single string.
- Let $P = \{p_1, p_2, ..., p_q\}$ be all the pitches from s_1.
- Let $T = \{t_1, t_2, ..., t_q\}$ be the set of all chords.
- Let $T_i = \{t_{i_1}, t_{i_2}, ..., t_{i_q}\}$ be the set of notes composing a chord.
- Let $B = \{b_1, b_2, ..., b_q\}$ be the set of all chord distances.
- Let $T_{i_j} = \{t_{i_{j_1}}, t_{i_{j_2}}, ..., t_{i_{j_L}}\}$ be the set of characters in a single note.
- Let $C = \{c_1, c_2, ..., c_L\}$ be the characters composing closest substring.
- Let $\varphi(op_1, op_2, ..., op_n)$ denote picking the most common element from a collection.
- Let $+$ denote character concatenation.
- Let $d(s_1, s_2)$ denote the hamming distance between s_1 and s_2.
- Let $k \in \mathbb{N}, [1, q]$

$$if\, \exists\, t_i \in T \,|\, (\forall t_{i_j} \in t_i \,\exists\, s_{i_j} \in s_i \,|\, d(p_k,\, s_{i_j}) \leq d \,\wedge$$
$$d(p_k,\, s_{i_j}) \leq d(p_k,\, S_i)) \,\wedge\, |\, (b_i \leq B)$$

$$then\, c = \varphi(t_{i_{1_1}}, t_{i_{2_1}}, ..., t_{i_{q_1}}) + \varphi(t_{i_{1_2}}, t_{i_{2_2}}, ..., t_{i_{q_2}}) + ... +$$
$$\varphi(t_{i_{1_L}}, t_{i_{2_L}}, ..., t_{i_{q_L}})$$

Applying big-O analysis to the algorithm yields, $O((k-1)n^2)$ where k is the number of lines, and n is the number of windows in a string. A sequential pseudocode algorithm for the CPU can be seen if Figure 5.

```
for each pitch
   for each string
       for each window
           distance = hammingDistance(pitch, window)
           if distance is minimum
               minDistance = distance
       if note found
           numNotesFound++
       chordDistance += minDistance
   if chordDistance is minimum
       bestChord = currentChord

root = calculateRootNote(bestChord)
```

Fig. 5
CPU ALGORITHM PSEUDOCODE.

3.2 Purely parallel GPU algorithm (PP-GPU)

An observation can be made about the sequential algorithm, each window's hamming distance to a certain pitch in the data set is completely independent of all the others. This observation implies that the parallel nature of the CUDA language can be exploited while calculating the hamming distance of all the windows with respect to pitches. An illustration of this is shown in Figure 6. If looking strictly at parallel computation even the process of computing the hamming distance itself can be incorporated into our CUDA algorithm (PP-GPU).

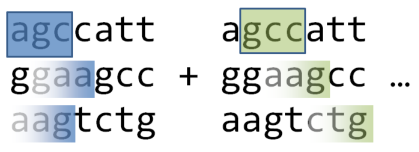

Fig. 6
PP-GPU ALGORITHM DESIGN.

3.2.1 PP-GPU

Initially our strategy was to exploit the parallel nature of CUDA by calculating each window's hamming distance to every pitch concurrently, while at the same time calculating the hamming distance in parallel. The kernel grid was aligned to perform the error calculation in one kernel. The X direction of the grid being the number of windows per row, $x = q$, signifying which window in the row we are, and the Y direction being number of rows, not including the row of pitches, multiplied by the number of windows per row, $y = numRows * q$. The Y direction was aligned in such a way so that the blocks Y coordinate modulo number of rows, not including the row of pitches, provides the row in the data set, the window that this block simulates, $by = blockIdx.y \,\%\, numRows$ and the blocks Y coordinate divided by number of rows, not including the row of pitches, yields the pitch we are to perform the hamming distance on $i = blockIdx.y/numRows$. If the block index was $(0, 4)$ in the 3 x 7 data set listed above, in Figure 6, then the block corresponds to the following where the pitch is on the first row and the window is on the second as shown by Figure 7.

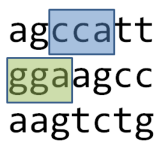

Fig. 7
BLOCK INDEXING SAMPLE.

Threads in each block calculate the hamming distance in parallel. A pseudo code version of the algorithm shown in Figure 8.

```
concurrently for all windows:
    tx = threadIdx.x
    bx = blockIdx.x
    by = blockIdx.y % numRows
    i =  blockIdx.y / numRows
    result = window[tx] != pitch[tx]
    distance = combine “result” from all threads in the block

chordDistance = Summation of distances of minimal distance to the pitch
locate minimal chordDistance and calculate root
```

Fig. 8
PP-GPU ALGORITHM PSUEDOCODE.

Although this idea provides the greatest amount of parallelism, due to hardware limitations this algorithm proved to be the slowest of the GPU based algorithms. The hardware limitation was with either a large data set and/or a small window size the number of blocks to be placed in the grid out grew the maximum grid limit imposed by CUDA. This required the kernel to be launched several times. With each kernel invocation there is time wasted making the kernel call and with a data set of 512 x 512 characters and window sizes of 15 characters the grid in the Y direction needed $254,976$ blocks. However CUDA only supports $65,535$ blocks in the Y direction on the grid [2], thus resulting in the kernel needing to be called 4 times, and that number increases to 16 with a 1024 x 1024 data set with the same sized windows.

3.2.2 PP-GPU*

Discovering the hardware limitation gain we attempted to remove multiple kernel launches in an attempt to increase performance by shrinking the number of blocks in the Y direction with the use of a for loop that iterates over the windows in a row. This design came to be known as (PP-GPU*). Incorporating this idea into the above algorithm design we can lay out the grid as to align the Y direction with the rows of the data set while the X is aligned on pitches we are to find a hamming distance to. Each block is still calculating the hamming distance in seemingly parallel. A pseudo code algorithm is shown in Figure 9.

```
concurrently for each pitch:
    for each window in my row
        calculate error for my element of the window to the pitch
        distance = summation of local errors
        if distance < minDistance
            store distance and the windows index
    if minDistance > maxError
        throwout this pitch
    chordDistance = summation of minDistances for the pitch

locate minimal chordDistance and calculate root
```

Fig. 9
PP-GPU* ALGORITHM PSEUDOCODE.

PP-GPU attempted to perform as many calculations as possible in parallel after experimentation was performed in order to optimize the kernel. An optimized version PP-GPU* did remove the hardware limitations and increased the speedup shown in $Experimental\ Results$, but more performance was possible. The parallel calculation of the hamming distance was discarded and the layout and functions of the grid, block structure was redesigned.

3.3 Streamlined GPU algorithm (S-GPU)

Upon seeing that hardware limitations will be hit if we take advantage of every bit of parallelism present in the problem, we re-engineered and streamlined the algorithm. The resulting algorithm takes the most efficient parallel ideas and discards those that only caused increased overhead when implemented on current hardware.

The main idea behind the streamlined algorithm is that in order to find the closest substring, all chords must be calculated and compared in order to determine which one has the smallest chord distance. Each chord finding operation is independent of each other. Looking further, within each chord finding operation, each note finding operation is independent. These facts point us toward a parallel algorithm in which all chords can be found simultaneously, and within each chord finding operation, all notes can be found concurrently.

The aforementioned description leads to an implementation of an algorithm that essentially eliminates the outer two *for* loops of the CPU algorithm, by doing them concurrently. We chose to have each CUDA block calculate the best chord associated with a single pitch, each CUDA thread of the block will search one string for the closest note concurrently. This allows us to take advantage of the CUDA programming model that allows threads to cooperate. In CUDA, when a kernel is called the caller must specify the number of blocks that will execute the kernel and the number of threads that each block should contain. Our kernel uses the number of windows possible in a string to be the number of blocks, specifically $numBlocks = q$. The number of threads is directly proportional to the number of strings, specifically $numStrings = numThreads$. The chosen configuration allows for notes of a chord to share their distances so that a chord distance can be calculated, CUDA threads allow this kind of cooperation. This configuration lends itself well to the problem and increases parallelism without adding additional memory requirements. Figure 10 shows a simplified version of the algorithm in pseudocode.

```
each string with each pitch concurrently:
   for all windows in a string
      distance = hammingDistance(pitch, window)
      if distance is minimum
          minDistance = distance
          noteFound = 1
   add(numNotesFound, noteFound)
   sync
   if numNotesFound == numRow
      add(chordDistance, minDistance)
   else
      invalidateChord()
```

Fig. 10
S-GPU ALGORITHM PSEUDOCODE.

This, Streamlined GPU algorithm results in less memory use on the device. Simply put, the memory use went from using n^2 amount of memory to store intermediate results with PP-GPU, to using only n memory for intermediate results on both PP-GPU* and S-GPU. Where n is the size of the data-set. This decrease is tremendous when considering inputs for n are typically large. It should also be pointed out that S-GPU cuts the size of the results generated from the kernel form n^2 to n. This is good news considering that the transfer from CPU to GPU memory has historically been a bottleneck. The streamlined algorithm also allows for one kernel to do all necessary calculations, rather than multiple kernels which was required in the PP-GPU algorithm. These improvements result in a faster algorithm.

4. Experimental results

4.1 Test hardware

All experiments were preformed on the following system:

- CPU: 2x Intel Xeon X5660 @ 2.80GHz
  - 6 core 12 threads per CPU
- Memory: 24 GB
- GPU: Tesla C2070 @ 1.15 GHz
  - Driver version: 260.19.26

4.2 Experiments

4.2.1 Experiment 1

Compared to another exact version (running sequentially), we achieved a 18x speedup for an input file of size 1024 x 1024 with a length of 3 as shown in Figure 11.

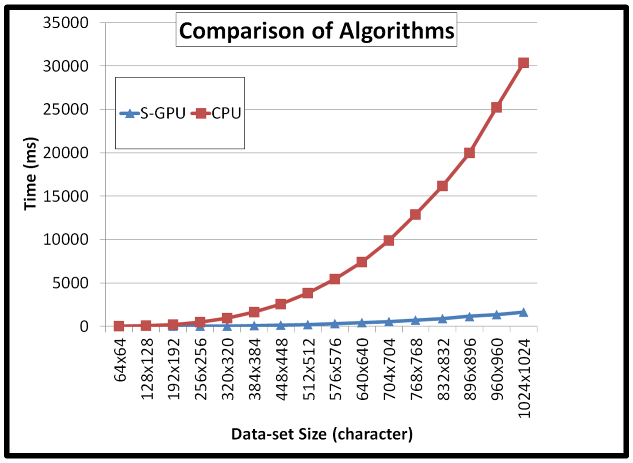

Fig. 11
COMPARISON OF ALGORITHMS CPU AND S-GPU: $L = 3$, $d = 3$

By successfully flattening the time vs. data-set size function it is possible for some applications to get timely results whereas before results would have been prohibitively expensive. The accuracy of the algorithm is unchanged. It still can deduce the closest substring with precise accuracy.

4.2.2 Experiment 2

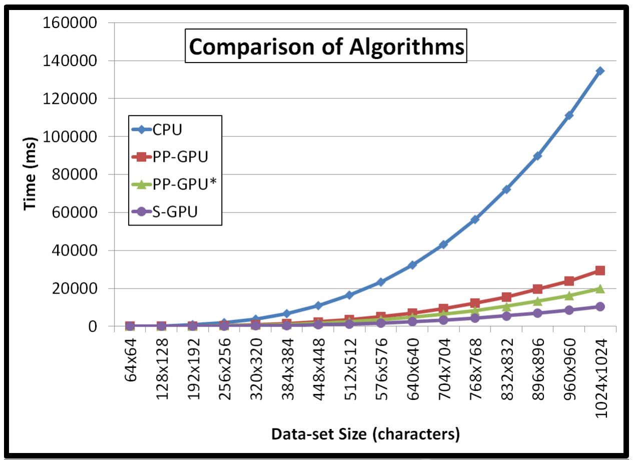

Fig. 12
COMPARISON OF ALGORITHMS CPU, PP-GPU(*), AND S-GPU: $L = 16$, $d = 4$

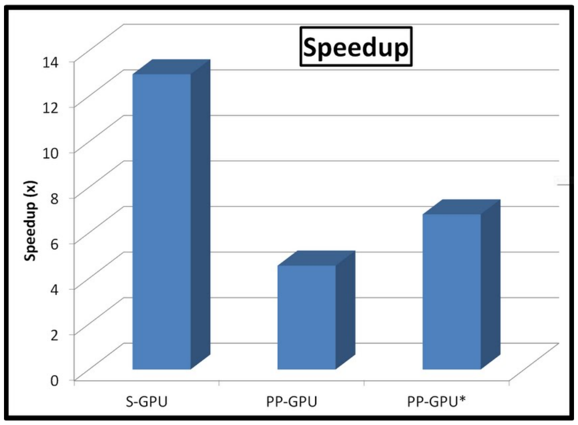

Fig. 13
OVERALL ALL SPEEDUP OF GPU ALGORITHMS:
$L = 16$, $d = 4$

Although PP-GPU attempted to preform the calculations of finding the hamming distances and chord errors in parallel, this action did not equate in much of a performance gain when compared to S-GPU. Its design was more fine grain than S-GPU, and thus needed more synchronization and collaboration to preform the calculation. The original design also facilitated the need for multiple kernel invocations which also contributed to the degradation in performance. The hardware fixed version PP-GPU* does provide a speed up of about 1.5x over its predecessor (PP-GPU) which had an overall speed up of 4.5x, and an speedup of 6.8x when compared to the CPU. S-GPU improved on this by utilizing a less fine grained approach to the problem allowing for the removal of key bottlenecks. In experiments, S-GPU ran 1.9x faster than PP-GPU* and achieved an overall speed up of 13x. It can be inferred from the results that more threads preforming small jobs is not always conducive to better performance, rather a smaller number of threads preforming sightly more work produced the best results in our experiments.

5. Related work

There has been much work done in areas closely related to CSSP. Problems such as multiple sequence alignment have experienced heavy research as of late.

Some have attempted to apply GPU algorithms to speedup multiple sequence alignment [9]. Although sequence alignment and CSSP are related they are not the same. CSSP is a more general problem. Perhaps ideas formed here can be adapted and applied to multiple sequence alignment tools such as Clustal. Perhaps ideas here can be applied to other areas in which forms of CSSP are represented.

Related work specific to CSSP has also been done. There have been many novel optimizations to approximation algorithms [1]. These algorithms generally perform much faster than an exact algorithm. The downside of course is that the results obtained are not verified. Our research focused on optimizing an exact algorithm, one who's result can be verified.

An approximate evolutionary algorithm for CSSP has been studied [8]. The algorithm attempts to change itself to make its results more accurate over time.

6. Conclusion

GPU devices can be used to efficiently solve the CSSP using parallel algorithms and GPU technology. For applications that require solving the CSSP or any of its relatives, efficient GPU algorithms can be developed that will permit computations that were previously too expensive.

Using parallel GPU algorithms and smart parallelization strategies we were able to greatly speedup the process of calculating the closest substring. The many existing applications that use a form of the CSSP can make use of GPUs to make their calculations faster. New applications could be built that previously could not due to the time required to calculate the closest string.

7. Future work

The algorithm can be expanded to work on larger data sets. The process of locating the lowest chord distance can be parallelized. The algorithm can be further optimized by using parallel minimization of the chord distances. This could result in an additional speedup for large data sets. The process of finding the chord distances can also be optimized using a parallel addition rather than the, sequential at worst, atomic add function included in the CUDA toolkit.

References

[1] Bin Ma. “A Polynomial Time Approximation Scheme for the Closest Substring Problem (2000)” In Proceedings of the 11th Annual Symposium on Combinatorial Pattern matching, 2000, pp. 97-107
[2] ”NVIDIA CUDA C Programming Guide”. Internet: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf November 9,2010 [March 21, 2011]
[3] K. Lanctot, M. Li, B. Ma, S. Wang, L. Zhang. Distinguish string search problems, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 633-642, San Francisco, 1999
[4] J. Sanders, E. Kandrot, CUDA By Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 2010
[5] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
[6] Ming Li, Bin Ma, and Lusheng Wang. “On The Closest String and Substring Problems” eprint ARXIV 2000.
[7] Markus Chimani, Matthias Woste, Sebastian Bocker “A Closer Look at the Closest String and Closest Substring Problem” 2011 Workshop on Algorithm Engineering and Experiments (ALENEX) 2011.
[8] Holger Mauch. “Closest Substring Problem - Results from an Evolutionary Algorithm,” in Neural Information Processing, 1st ed., vol 3316. Ed. Nikhil Pal, Ed. Nik Kasabov, Ed. Rajani Mudi, Ed. Srimanta Pal, Ed. Swapan Parui, 2004, pp. 205-211.
[9] Andrew Bellenir, Christian Trefftz, Greg Wolffe, "Graphics Processor Based Implementation of Bioinformatics Codes", 2008.