

PARALLELIZING SUFFIX ARRAY ON CUDA FOR BIOINFORMATICS APPLICATIONS

A DISSERTATION

*Submitted in partial fulfillment of the
requirements for the award of the degree*

of

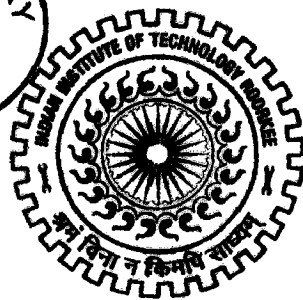
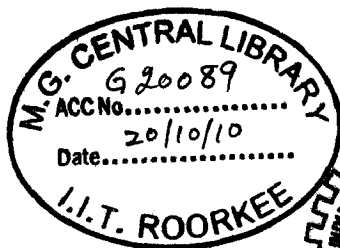
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

LAXMI KANT SAHU



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2010

Candidate Declaration

I hereby declare that the work being presented in the dissertation report titled “**Parallelizing Suffix Array on CUDA for Bioinformatics Applications**” in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my own work carried out under the guidance of Dr Rajdeep Niyogi, in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 02/06/10

Place: IIT Roorkee.



Laxmi Kant Sahu

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dated: 02/06/10

Place: IIT Roorkee


Dr Rajdeep Niyogi
Assistant Professor,
Department of Electronics
And Computer Engineering

Acknowledgements

First of all and foremost, I would like to express my deep sense of gratitude and indebtedness to my guide Dr Rajdeep Niyogi, for his invaluable guidance and constant encouragement throughout the dissertation. His zeal for getting the best out of his students helped me to perform above my par.

I am also grateful to my ex-guide Dr Ankush Mittal, for his guidance and suggestions in my dissertation.

I would want to express thanks to my colleagues Anant Bhushan, Sanketh D and Binay Kumar Pandey, for their “taken for granted” help with trivial matters, without which I am sure my work might have hit a dead end.

I would want to express thanks to Institute Computer Centre Head and their staff for providing me high performance computing facility.

Laxmi Kant Sahu

Abstract

Suffix trees have been applied to fundamental string problems such as finding the longest repeated substring, finding all squares or repetitions in a string, computing substring statistics, approximate string matching and string comparison. They have also been used to address other types of problems such as text compression, compressing assembly code, inverted indices and analyzing genetic sequences.

Suffix array is a simpler and compact alternative to the suffix tree; suffix sorting is the fundamental building block in suffix array construction process. Suffix array widely used in field of bioinformatics and text processing. The repeat structure of genomic DNA is considered an essential mechanism for evolution and other fundamental biological functions. Any kind of repeats finding problems are always deemed as one of the prerequisites for genome sequencing and analysis, and among these problems exact repeat finding is the first step for most other repeats finding problems.

This work depicts the parallel implementation of suffix array and then their applications in bioinformatics such as exact repeats finding, tandem repeats finding, exact string matching etc. the parallel implementation of suffix array algorithm on a GPU using the Compute Unified Device Architecture (CUDA) platform, both from NVIDIA Corporation. CUDA is a parallel computing architecture. It is a middle-ware compute engine which exposes the power of NVIDIA Graphics Processing Units to software developers through industry standard programming language. The thread level parallel code block provides an efficient primitive for building a high performance suffix array construction program and many other applications.

The parallel version runs much faster than any serial implementation on CPU for the large size of input data elements. The parallel algorithm can also be easily adapted for similar type of problems with little modification.

Table of Contents

Candidate Declaration.....	1
Acknowledgements.....	2
Abstract.....	3
List of Figures.....	6
List of Tables.....	7
Chapter 1: Introduction.....	8
1.1 Suffix Array.....	8
1.2 Repeats finding and exact string matching.....	8
1.3 Multicore Architecture.....	9
1.4 Problem Statement.....	9
1.5 Organization of Report.....	9
Chapter 2: CUDA.....	11
2.1 General Programming on GPU (GPGPU).....	11
2.2 General Architecture of GPUs.....	12
2.3 CUDA.....	13
2.4 Programming constructs and Thread Hierarchy.....	16
2.5 Memory Hierarchy.....	18
Chapter 3: Suffix Array.....	20
3.1 Sequential implementation of suffix array.....	21
3.2 Parallel implementation of suffix array.....	22
3.3 Performance comparison of parallel and sequential implementations.....	25
Chapter 4: Application of Suffix Array in Bioinformatics.....	26
4.1 Exact Repeats.....	26
4.1.1 Sequential implementation of exact repeats finding.....	27
4.1.2 Parallel implementation of exact repeats finding.....	28
4.1.3 Performance comparison of sequential and parallel exact repeats finding.....	31
4.2 Tandem Repeats.....	32
4.2.1 Sequential implementation of tandem repeats finding.....	35
4.2.2 Parallel implementation of tandem repeats finding.....	37

4.2.3 Performance comparison of sequential and parallel tandem repeats finding	38
4.3 Exact String Matching.....	39
4.3.1 Sequential implementation of exact string matching	42
4.3.2 Parallel implementation of exact string matching	44
4.3.3 Performance comparison of sequential and parallel exact string matching	45
Chapter 5: Conclusion and future works	46
References.....	47

List of Figures

Figure 2.1: Floating point operations for the CPU and the GPU.....	12
Figure 2.2: General architecture difference between CPU and GPU	13
Figure 2.3: Common CUDA program flow.....	15
Figure 2.4: Thread Hierarchy in CUDA	17
Figure 2.5: How threads access global, shared and local memory	19
Figure 3.1: Flow diagram of sequential implementation of suffix array	22
Figure 3.2: Flow diagram of parallel implementation of suffix array	23
Figure 3.3: Comparison of parallel and sequential implementation of suffix array	25
Figure 4.1: Finding Exact Repeats with length $w=4$	26
Figure 4.2: Flow diagram of sequential implementation of exact repeats finding	27
Figure 4.3: Flow diagram of parallel implementation of exact repeats finding.....	29
Figure 4.4: Comparison of parallel and sequential implementation of exact repeats finding	31
Figure 4.5: Illustration of satellite bands	32
Figure 4.6: A partial human STR profile obtained using Applied Biosystems Identifiler kit	33
Figure 4.7: Schematic of a Variable Number of Tandem Repeats in 4 alleles.....	34
Figure 4.8: Flow diagram of sequential implementation of tandem repeats finding.....	35
Figure 4.9: Flow diagram of parallel implementation of tandem repeats finding	37
Figure 4.10: Comparison of parallel and sequential implementation of tandem repeats finding.	38
Figure 4.11: The time spent in each phase of the suffix tree matching program on the CPU	41
Figure 4.12: The time spent in each phase of the suffix tree matching program on GPU.....	42
Figure 4.13: Flow diagram of sequential implementation of exact string matching	43
Figure 4.14: Flow diagram of parallel implementation of exact string matching	44
Figure 4.15: Comparison of parallel and sequential implementation of exact string matching ...	45

List of Tables

Table 3.1: Example of Suffix Array	20
Table 4.1: Timing comparison for exact repeats finding.....	31
Table 4.2: Timing comparison for tandem repeats finding.....	38
Table 4.3: Timing comparison for exact string matching.....	45

Chapter 1: Introduction

1.1 Suffix Array

Suffix trees and suffix arrays are widely used and largely interchangeable index structures on strings and sequences. Practitioners prefer suffix arrays due to their simplicity and space efficiency while theoreticians use suffix trees due to linear-time construction algorithms and more explicit structure [10]. The suffix array can be used to provide all the facility that can be provided by suffix tree such as all kind of strings operations. The fastest direct suffix array construction algorithms that do not use suffix trees require $O(|N|\log|N|)$ time, where N is a main string [11]. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, “Is W a substring of N ?” can be done in time $O(|W|\log|N|)$. Suffix arrays have been proved to be better in practice than suffix trees for many applications [14].

1.2 Repeats finding and exact string matching

Suffix trees and suffix genomes contain not only genes but also many repetitive DNA sequences, the significance of internal sequence repeats is quite clear in both eukaryote and procaryote. In procaryote, short repeats may act as regulators, virus binding sites, or enzyme binding sites etc. and long repeats seem to have other functions, such as evolution. In eukaryote, it is particularly true that these repetitive elements make up the majority of DNA in most eukaryotes, for example, 50% of the human genome has been identified as repetitive, these repeats drive genome evolution in diverse ways and should be masked off prior to performing homology searches which is not efficient and using the parallel implementation the time can be reduced [1].

In a tandem repeat, four features need to be analyzed: the pattern size, the pattern structure, the number of copies, and the positions of the patterns. According to the pattern size, repeats can be classified into three types: satellites, minisatellites, and microsatellites [4]. Our work makes an attempt to replace an entire computational grid of computers used for repeats findings and exact string matching with a single highly parallel commodity multiprocessing board, in the form of a

implementation by using of suffix array and their performance comparison, and then discusses exact string matching and its sequential and faster implementation by using of suffix array and their performance comparison.

Chapter 5 concludes the dissertation report and gives suggestion for future work.

Chapter 2: CUDA

2.1 General Programming on GPU (GPGPU)

The GPU [5] refers to the commodity off-the-shelf 3D Graphics Processing Units, which are specifically designed to be extremely fast at processing large graphics data sets for rendering tasks. GPU designers traditionally have expressed its image-synthesis process as a hardware pipeline of specialized stages which necessarily involve Vector/Matrix Operations. The need for efficient hardware to perform floating-point vector arithmetic for millions of vertices each second has helped drive the GPU parallel-computing revolution.

GPUs have evolved from a hard-wired implementation of the graphics pipeline to a more programmable one. Fixed-function units for transforming vertices and texturing pixels have been replaced by programmable shaders. These shaders provide units that the programmer can use for performing matrix-vector multiplication, exponentiation, and square root calculations etc. This however necessitates that there should be some means by which general purpose software could be translated into GPU specific primitives.

General purpose computing on the GPU is an active area of research. GPUs are already widespread. The performance of GPUs is improving at a rate faster than that of CPUs. The capabilities of the GPU have increased dramatically in the past few years and the current generation of GPUs has higher floating point performance than the most powerful (multicore) CPUs [5]. The GPU contains hundreds of cores that work great for parallel implementation. The programming is done in SIMD style where same code is worked on different data locations. Until recently a graphics API was needed to code on GPUs which made coding for non graphics oriented calculations tough. Trying to work around this limitation Nvidia released CUDA which allows GPUs to be programmed using a variation of C. This enables a low learning curve and makes programming easier.

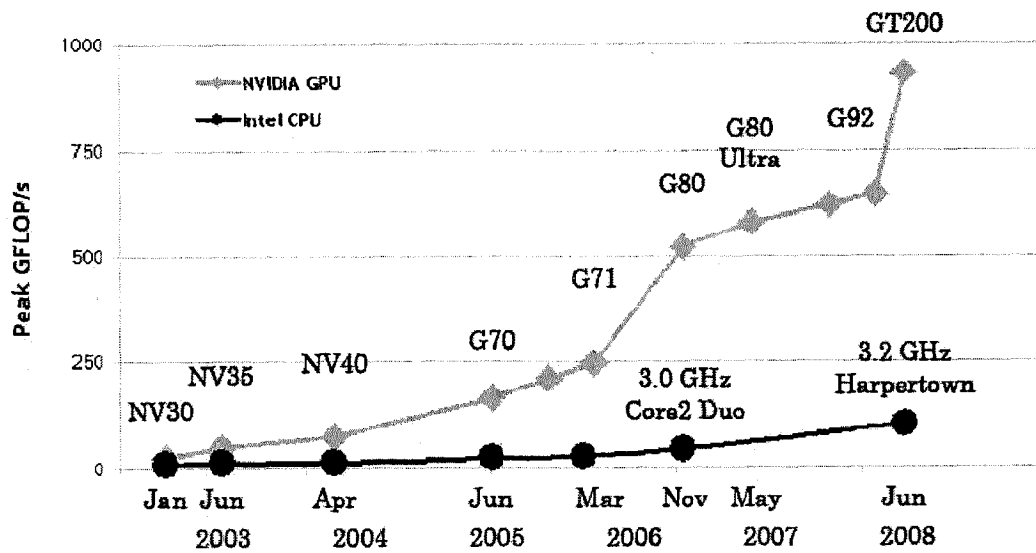


Figure 2.1: Floating point operations for the CPU and the GPU [5]

The figure 2.1 shows the tremendous computational capability of the GPU. GTX 280 a GT 200 family GPU delivers a peak performance of 933 GFLOPS/sec.

2.2 General Architecture of GPUs

Whereas CPUs are optimized for low latency, GPUs are optimized for high throughput. Thus applications that do not have requirement for low latency can be ported to GPUs to take advantage of their superior performance. The programmable GPU has evolved into a highly parallel, multi-threaded, many-core processor with tremendous computational horsepower and very high memory bandwidth. There is a widening gap between the raw performance capability of CPUs and GPUs, which is because the GPU is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about, and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The general architectural difference between CPUs and GPUs is schematically illustrated below in figure 2.2

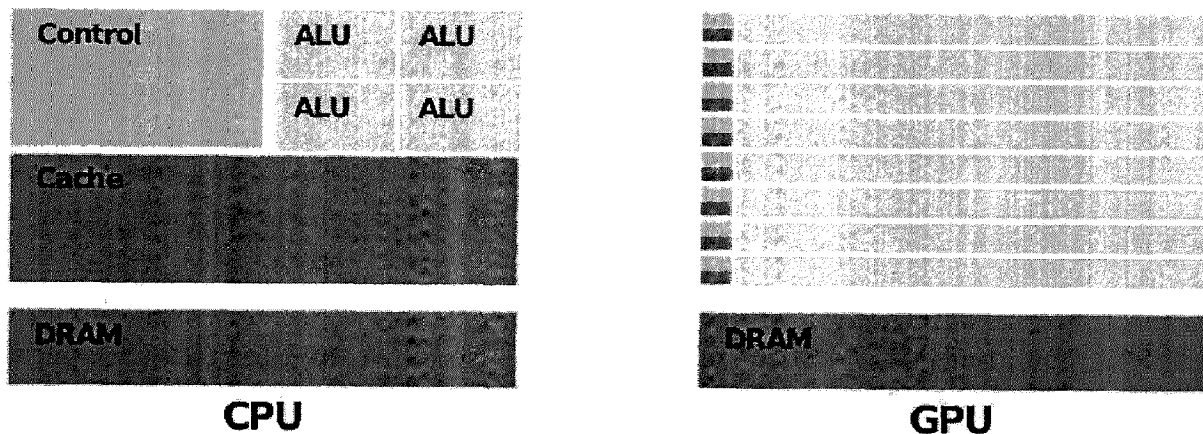


Figure 2.2: General architecture difference between CPU and GPU [5]

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations; the same program is executed on many data elements in parallel, with high ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. The CUDA programming model is very well suited to expose the parallel capabilities of GPUs.

2.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel programming model and software environment developed by Nvidia [5]. It was designed as a middle-ware to allow application software that transparently scales its parallelism on GPU. The core concepts involved with CUDA are a hierarchy of thread groups, shared memories, and barrier synchronization. The thread hierarchy allows user to divide his task in a similar hierarchy, where coarse sub-problems can be solved independently and finer pieces that can be solved cooperatively in parallel using shared memory. CUDA achieves all this using a minimal extension to C thus maintaining a low learning curve for programmers already familiar with the standard programming language.

To manage the numerous threads, the multiprocessor employs a single-instruction, multiple-thread (SIMT) architecture. This allows each thread to execute independent of the other threads on one of eight scalar processors. Instructions are issued to groups of 32 threads called warps,

which execute one common instruction at a time. If the instructions assigned to threads within a warp differ due to conditional branching, the warp executes each path sequentially while disabling threads that are not on the path. When all branch paths are complete, the threads join back to the common execution path. It is for this reason that code within conditional statements such as if/else should be limited.

Given the above information, it is now relevant to note that the GPUs of concern for this work have the following specifications:

- The maximum number of threads per block is 512
- The maximum size of each dimension of a grid of thread blocks is 65535
- The maximum number of active blocks per multiprocessor is 8
- The maximum number of active threads per multiprocessor is 1024
- The maximum number of active warps per multiprocessor is 32

CUDA also provides limited synchronization between threads of the same block via the syncthreads function call. Upon hitting a syncthread, each thread will wait until all remaining threads reach the call. Syncthreads is primarily used to coordinate communication between the threads within a block to prevent read/write data hazards with shared or global memory. The only way to synchronize across thread blocks is by breaking the computation into multiple kernels, as one kernel must complete before another can launch.

Most CUDA applications follow a set program flow. The host first loads data from a source such as a text file and stores it into a data structure in host memory. The host then allocates device memory for the data and copies the data to the allocated space. Kernels are then launched to process the data and produce results. These results are then copied back to the host for display or further processing.

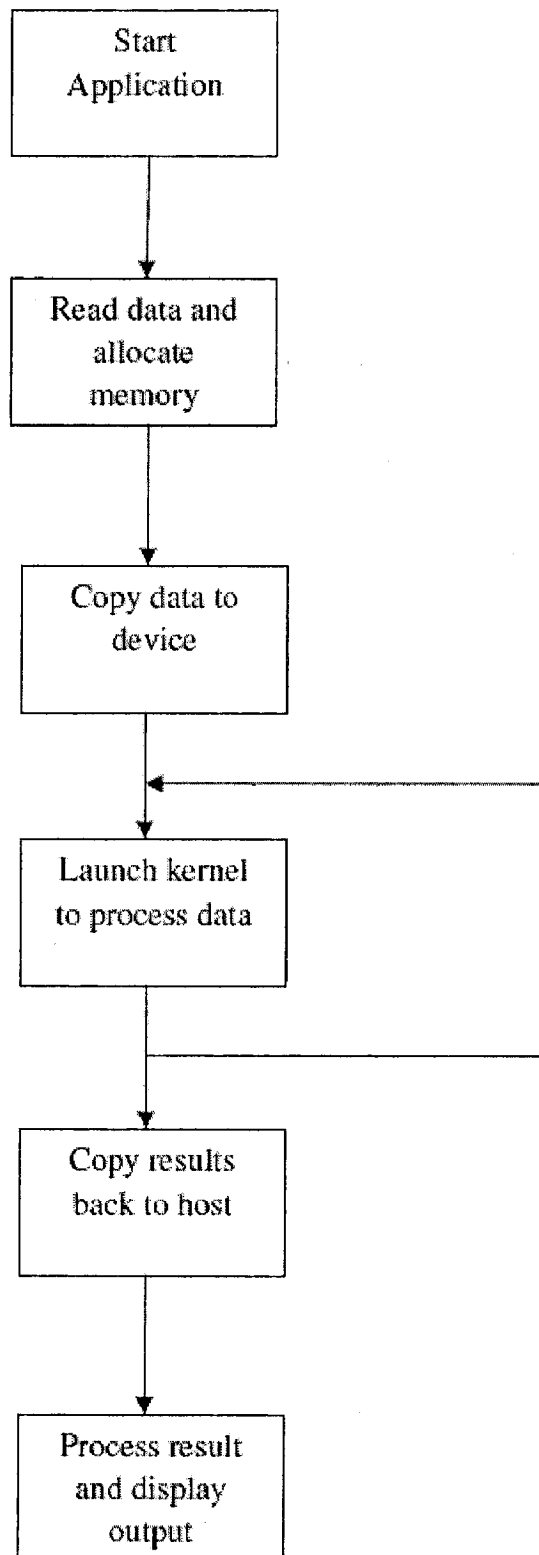


Figure 2.3: Common CUDA program flow

high performance Graphics Processing Unit (GPU) programmed in the Compute Unified Device Architecture (CUDA) framework [8].

1.3 Multicore Architecture

The Graphics Processing Units (GPUs) model themselves as multi-core processing and expect programs to take advantage of them as raw parallel number-crunchers. The multicore processors allow program to leverage their computing power by various means like independent threads per core, or allow user to manipulate efficient data flow between cores, or provide a layer of software which manages the scalability of the cores. With the future micro-processor trends likely to increase number of cores as the only means of their increasing computing power, it becomes necessary to ensure that important algorithms be parallelized to run on next generation of micro-processors. Thus multi-core processors provide the perfect means of increasing the runtimes of our sequential algorithm

1.4 Problem Statement

The performance improvement of Suffix Array on CUDA [5] was majorly due to the possibility of parallel sorting which took bulk of the runtime in the sequential implementation. The overall running time of the can be improved if the suffix sorting part of the algorithm is implemented using techniques of parallel sorting. Then these results can be used to solve lots of problems in the field of bioinformatics such as finding the exact repeats finding, all kind of tandem repeats in genome sequence and exact sequence matching.

1.5 Organization of Report

Organization of this dissertation report is as follows:

Chapter 2 covers a detailed explanation of the architecture of CUDA programming environment, which have been used in this dissertation.

Chapter 3 starts with the explanation of concepts of suffix array. The chapter then discusses the sequential and parallel implementation of suffix array and their performance comparison.

Chapter 4 starts with the role of exact repeats finding in bioinformatics applications. The chapter then discusses the sequential as well as faster implementation of the exact repeating finding and their performance comparison, then discusses the tandem repeats and its sequential and faster

2.4 Programming constructs and Thread Hierarchy

CUDA extends C[5] by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads for each call is specified using a new `<<<...>>>` syntax

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C) {
// Kernel code
}
int main() {
...

// Kernel invocation
vecAdd<<<1, N>>>(A, B, C);
...
}
```

Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx.x` variable. This `threadIdx.x` value gives the index of the current thread within its block. In the above code, if the kernel were to add the two vectors A and B of size N and stores the result into vector C, the kernel code would be

```
__global__ void vecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
```

The logical organization of the thread hierarchy is thus, with the entire set of threads arranged as a two dimensional grid of blocks, with each block containing a three dimensional set of threads, as shown in figure 2.4

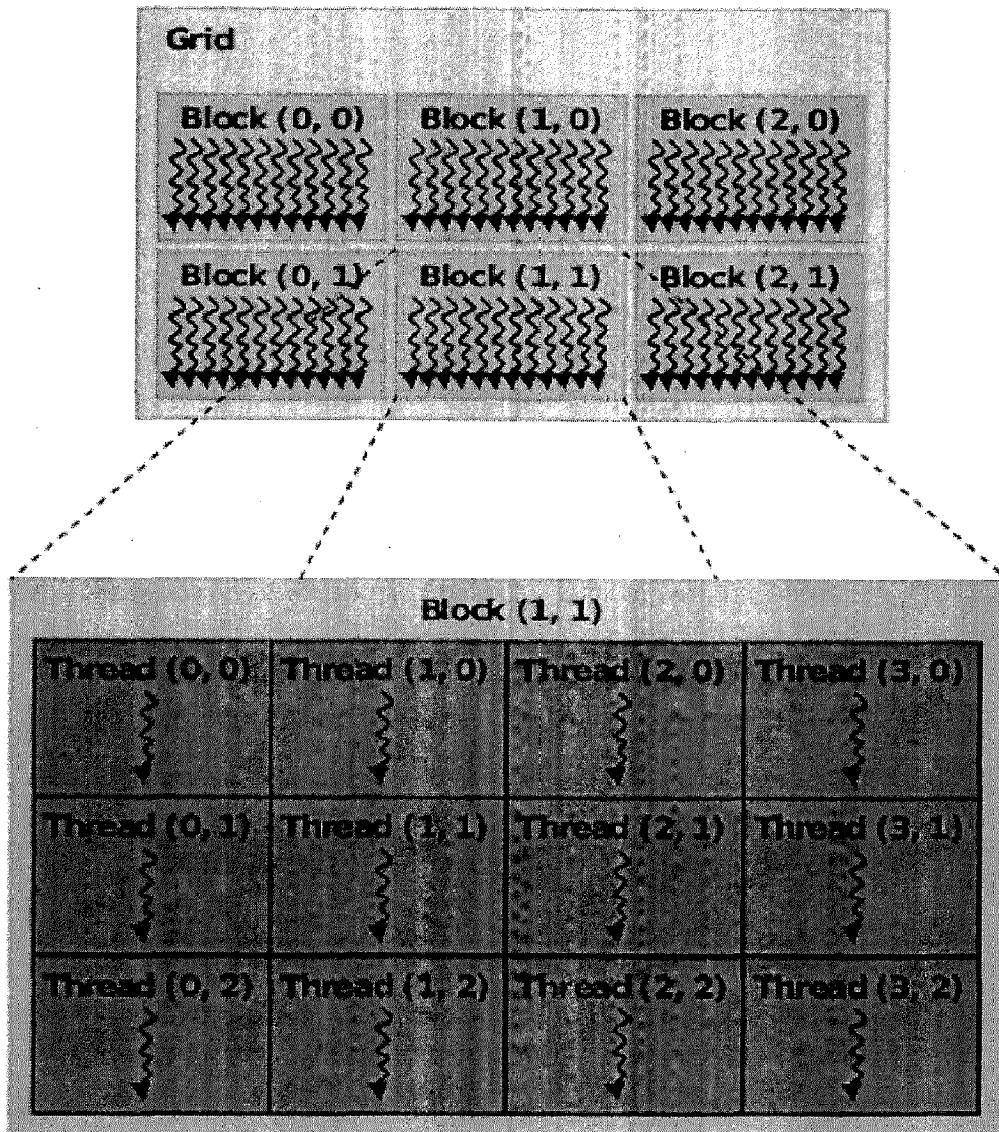


Figure 2.4: Thread Hierarchy in CUDA [5]

Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Such synchronization is possible by means of a programming primitive `__syncthreads()` as exposed by CUDA API. This serves as barrier synchronization. The number of threads per block is restricted

by the limited memory resources of a processor core. On NVIDIA Tesla architecture, a thread block may contain up to 512 threads.

In addition to the variable `threadIdx`, CUDA threads also have a few other built-in variables namely `blockIdx` and `blockDim`. The `blockIdx` variable gives the index of the thread's parent block within the grid, and `blockDim` which gives the number of threads per block, with the `blockDim` being supplied in the call to the kernel as the second parameter to the `<<<>>` syntax. Since grids are two-dimensional, `blockIdx` has a x component and y component and since blocks are three-dimensional, `blockDim` and `threadIdx` have x, y and z components. If the above code was to be a matrix addition instead of vector addition and was to be processed by a hierarchical arrangement of threads as shown in the above figure 2.4, with each thread processing one element of the matrix, the code becomes

```
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
if (i < N && j < N)
C[i][j] = A[i][j] + B[i][j];
}
int main()
{
// Kernel invocation
dim3 dimBlock(16, 16);
matAdd<<<1, dimBlock>>>(A, B, C);
}
```

2.5 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory, which is akin to local variable declaration for any normal CPU code. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory.

CUDA assumes that both the host and the device maintain their own DRAM, referred to as host memory and device memory respectively. The global memory is persistent across kernel launches by the same application and is allocated in the device memory. Memory management at runtime on the GPU RAM is done using CUDA API equivalents. The general procedure is to allocate memory on both host and device RAM, using `cudaMalloc` function call for the device memory. The data contents are copied from host memory to device memory using `cudaMemcpy` function. Writing data directly onto device memory from CPU code is not possible. The kernel calls are then made to do appropriate processing on the data. The processed data contents are copied back from the device to the host using `cudaMemcpy` function.

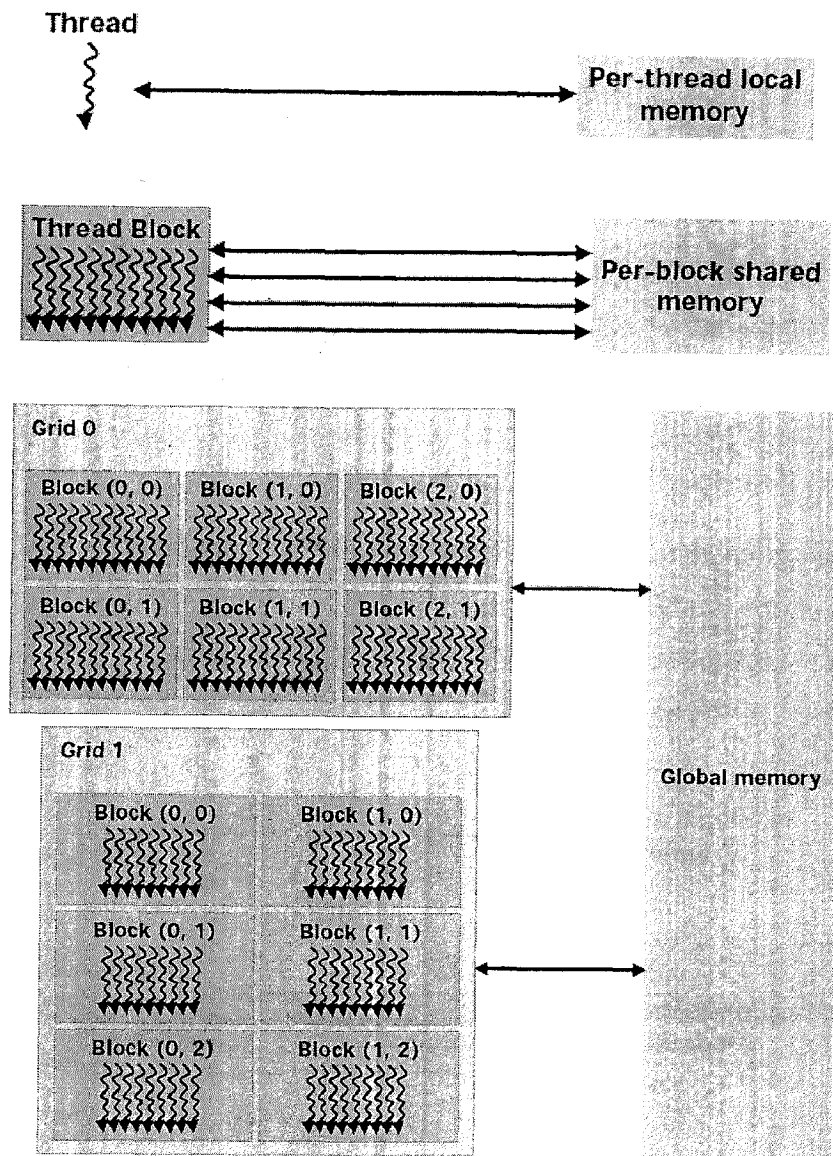
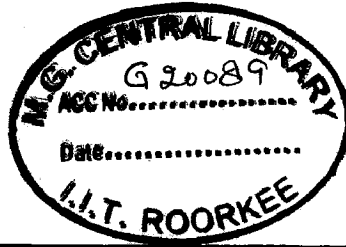


Figure 2.5: How threads access global, shared and local memory [5]



Chapter 3: Suffix Array

A suffix array [12] is an array of integers giving the starting positions of suffixes of a string in lexicographical order.

Following table 3.1 shows the suffix array for string "ATTCGATTCGATTCG"

Table 3.1: Example of Suffix Array

Suffix Array	Suffix
10	ATTCG
5	ATTCGATTCG
0	ATTCGATTCGATTCG
13	CG
8	CGATTCG
3	CGATTCGATTCG
14	G
9	GATTCG
4	GATTCGATTCG
12	TCG
7	TCGATTCG
2	TCGATTCGATTCG
11	TTCG
6	TTCGATTCG
1	TTCGATTCGATTCG

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of the indices of suffixes sorted in lexicographical order. For the string "ATTCGATTCGATTCG", using one-based indexing, the suffix array is {10,5,0,13,8,3,14,9,4,12,7,2,11,6,1}.

The suffix array is a simpler and more compact alternative to the suffix tree. Furthermore, parallel suffix array construction has emerged as an interesting research field to meet high-

performance computing requirement in full text index, data compression [2]. The suffix arrays can also be used to index protein structure [3].

Suffix sorting is one of the fundamental steps in suffix array construction process, which builds the unique index value of each suffix according to its alphabetic order, as shown in table 3.1. The naive method to build these index values is directly sorting all the suffixes, though this is not efficient method but we can make it more efficient by implementation of parallel sorting techniques and CUDA provides the many cores for parallel processing and that will definitely improve the performance with respect to sequential processing.

3.1 Sequential implementation of suffix array

Several algorithms have also been developed which provide faster construction and have space usage of $O(n)$ with low constants. The easiest way to construct a suffix array is to use an efficient comparison sort algorithm which requires no extra space. Though it is more time consuming approach but very much space efficient. Figure 3.1 shows the flow diagram of sequential implementation and the algorithm 3.1 has been used for its implementation. In this algorithm *my_strcmp* is our customized string compare function and *suffixIndex* array is used to store indexes of sorted suffixes.

Algorithm 3.1: Sequential implementation of suffix array

```
for i := 0 to MAXLENGTH - 1
  for j:=0 to MAXLENGTH - (i + 2)
    if (my_strcmp(stringBaseAddr + suffixIndex[j],
      stringBaseAddr + suffixIndex[j + 1]) > 0)
      then
        temp := suffixIndex[j];
        suffixIndex[j] := suffixIndex[j + 1];
        suffixIndex[j + 1] := temp;
      end if
```

CPU
(Sequential Processing)

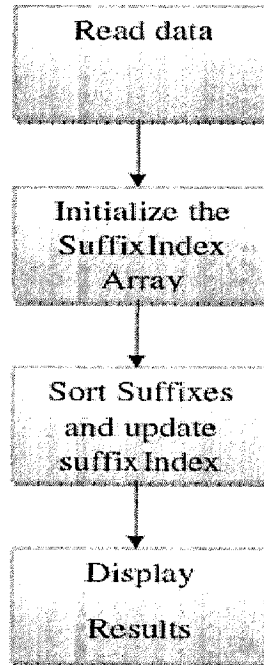


Figure 3.1: Flow diagram of sequential implementation of suffix array

3.2 Parallel implementation of suffix array

The GPU implementation primarily consists of two main phases, namely data construction and data comparisons. The data construction phase consists of allocating memory on the GPU and transferring data onto it from the CPU. The data as required by the algorithms was to be generated and processed on the GPU, with the CPU doing the initial work of reading the gene sequences and have them transferred to the GPU. One of the problem encountered in use of CUDA was the absence of string processing libraries on GPUs (since the device is primarily math-intensive), which required that they be written from scratch as device-level user functions. Once the allocation of memory on device is done, the process of generation of suffix array divided between the thread as show in figure 3.2 on the device using a parallelized form on comparison of suffix at position $2*threadId$ and $2*threadId - 1$ then comparison of suffix at position $2*threadId$ and $2*threadId + 1$. These steps carried out up to half of the input length of gene sequence. Cormen[13] argued that these procedure gives the correct result. Our algorithm 3.2 shows the execution steps in detail.

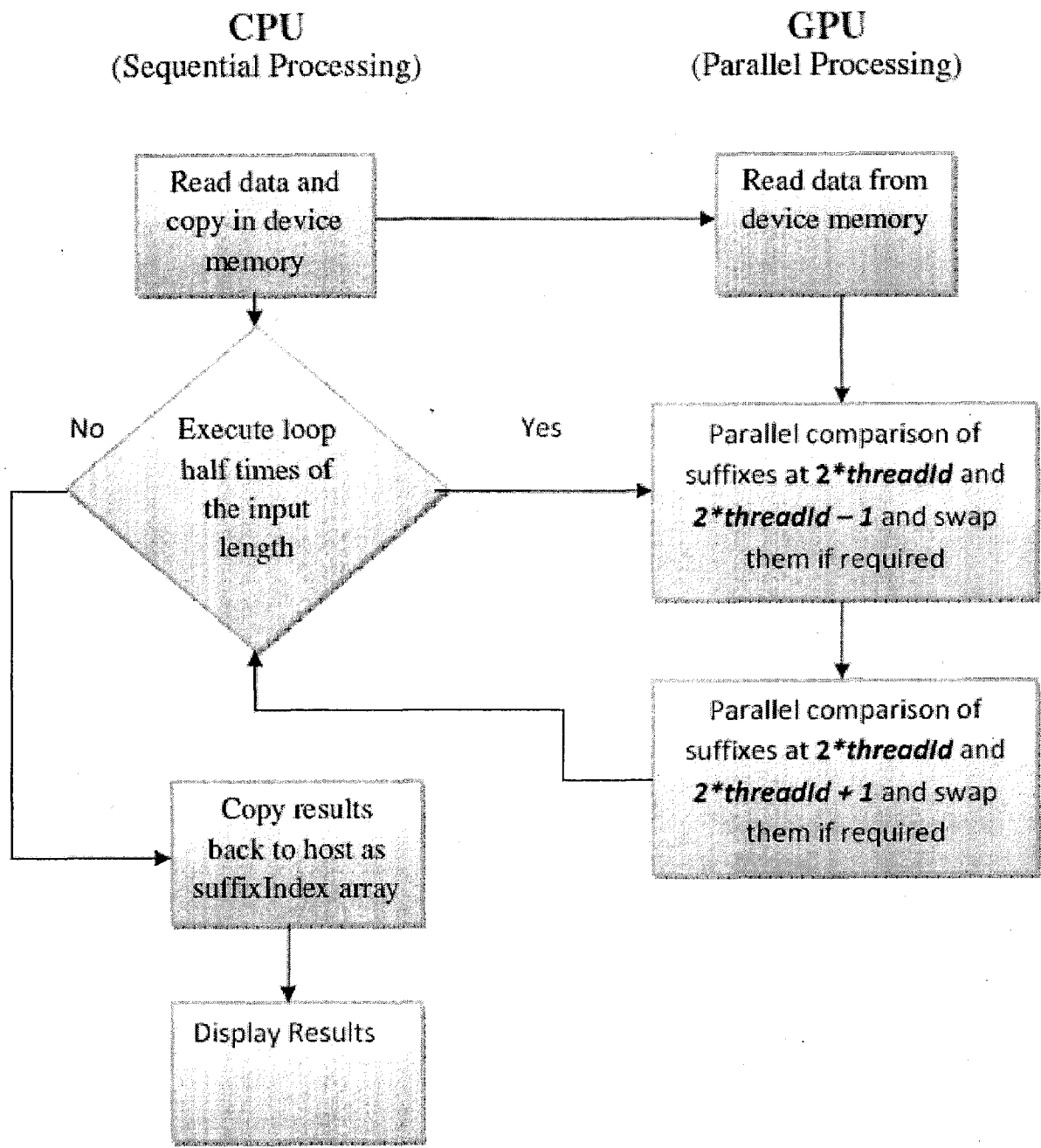


Figure 3.2: Flow diagram of parallel implementation of suffix array

Algorithm 3.2: Parallel implementation of suffix array

```
//Steps from CPU
```

```
for i := 0 to (MAXLENGTH / 2)
```

```
  Step 1: CUDA steps for first pass comparisons
```

```
  Step 2: CUDA steps for second pass comparisons
```

```
//CUDA steps for first pass comparisons
```

```
Initialize threadIdx := blockIdx.x * 512 + threadIdx.x;  
if (2*threadIdx + 1 >= MAXLENGTH) then return;  
end if  
if (my_strcmp(stringBaseAddr + suffixIndex[2*threadIdx],  
             stringBaseAddr + suffixIndex[2*threadIdx + 1]) > 0)  
then  
  temp := stringBaseAddr + suffixIndex[2*threadIdx];  
  suffixIndex[2*threadIdx] := stringBaseAddr +  
  suffixIndex[2*threadIdx + 1];  
  suffixIndex[2*threadIdx + 1] := temp;  
end if
```

```
//CUDA steps for second pass comparisons
```

```
Initialize threadIdx := blockIdx.x * 512 + threadIdx.x;  
if (2*threadIdx + 2 >= MAXLENGTH) then return;  
end if  
if (my_strcmp(stringBaseAddr + suffixIndex[2*threadIdx + 1],  
             stringBaseAddr + suffixIndex[2*threadIdx + 2]) > 0)  
then  
  temp := stringBaseAddr + suffixIndex[2*threadIdx + 1];  
  suffixIndex[2*threadIdx + 1] := stringBaseAddr +  
  suffixIndex[2*threadIdx + 2];  
  suffixIndex[2*threadIdx + 2] := temp;  
end if
```

3.3 Performance comparison of parallel and sequential implementations

The following table 3.2 shows the comparison between sequential and parallel implementation of suffix array. The processing time depends upon the length of the input data. As we can see in figure 3.3 the GPU implementation giving the better performance over the CPU implementation as long as input length keeps increasing.

Table 3.2: Timing comparison for suffix array

Input Length	Sequential Processing Time (seconds)	Parallel Processing Time (seconds)
64	0.000352	0.000354
128	0.002213	0.000666
192	0.007104	0.000983
256	0.016507	0.001317

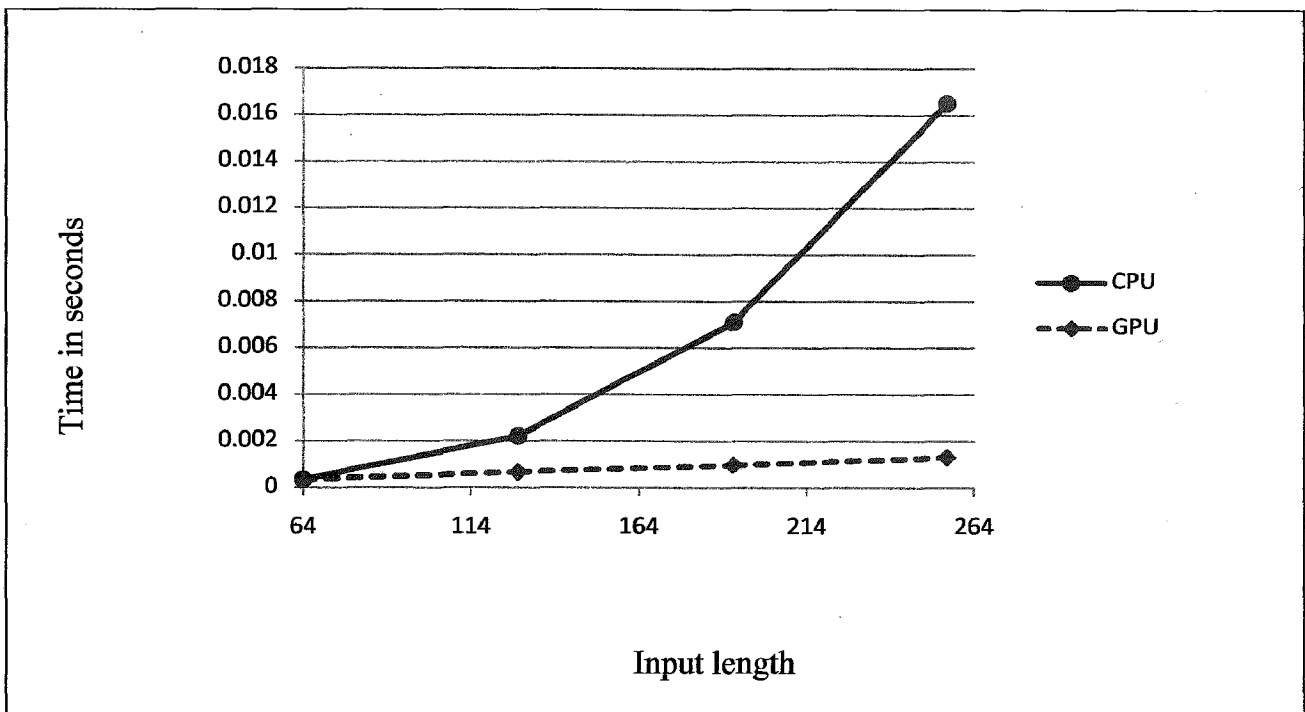


Figure 3.3: Comparison of parallel and sequential implementation of suffix array

Chapter 4: Application of Suffix Array in Bioinformatics

Suffix array has mainly used for indexing purpose and it has many applications in field of bioinformatics. Repeats finding and sequence matching problems are always deemed as one of the prerequisites for genome sequencing and analysis.

4.1 Exact Repeats

The problem can be defined as [1]: given an input DNA sequence S as a string of length n over alphabet $\Sigma = \{A,C,G,T\}$, $S[i]$ is the i^{th} character of S , $i \in [1,n]$, we want to find exact repeats subsequence of length w . For $i \leq j$, let $S[i,j]$ denote the substring of S between the position of i and j . When we refer to a string with fixed length w , S_i can be used as a shorthand for $S[i,j]$, for $j=j+w-1$.

A pair of substrings $R=(S_i,S_j)$ is an exact repeat if and only if $i \neq j$ and each corresponding characters of S_i and S_j are equal.

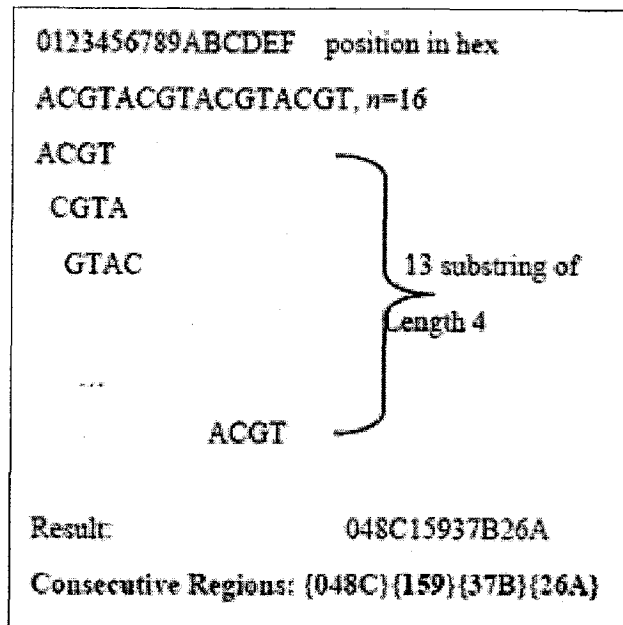


Figure 4.1: Finding Exact Repeats with length $w=4$ [1]

There are many algorithms to compute exact repeats, for example, the most simple and natural idea is exhaustive enumeration, which is easy for implementation and parallelization, but the $O(n^2)$ time complexity is prohibitive when we thinking the problem size at genome level, so we only mention the feasible algorithms in time complexity. Hashing method is the standard approach to do exact repeat which will work at $O(n)$ time complexity, when appropriate hashing function is selected, but the variable length of repeat sequence put a severe restriction on hashing techniques [1].

4.1.1 Sequential implementation of exact repeats finding

We have proposed an algorithm based on suffix array to find out the exact repeats with arbitrary length, which provide a much simple way to solve the DNA sequence repeats problem. The figure 4.2 shows the flow diagram of our sequential implementation. In addition to suffix array an extra space $O(n)$ have been used as *auxiliaryArray* to provide the ease of exact repeats finding and algorithm 4.1 shows the execution steps.

CPU
(Sequential Processing)

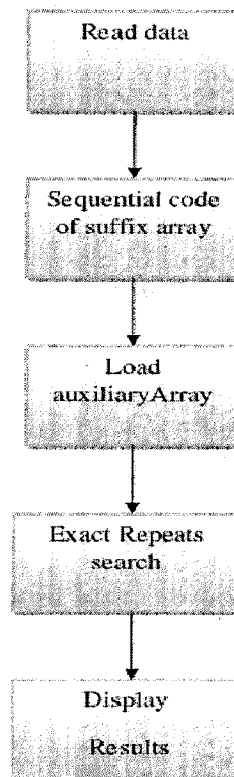


Figure 4.2: Flow diagram of sequential implementation of exact repeats finding

Algorithm 4.1: Sequential exact repeats finding

```
1) Call algorithm 3.1: Sequential implementation of suffix array (refer page no. 21)
2) Call algorithm 4.2: Loading of auxiliaryArray (refer page no. 28)
3) for i := 0 to (MAXLENGTH - 1)
    if(auxiliaryArray[i + 1] >= w) then
        flag := 1;
        Print: suffixIndex[i], groupIndex
    else
        if(flag == 1)
            Print: suffixIndex[i], groupIndex;
            flag := 0;
            groupIndex++;
        end if
    end if
```

Algorithm 4.2: Loading of auxiliaryArray

```
Initialize i := 0;
for j:=0 to (MAXLENGTH - 2)
    str_1 := stringBaseAddr + suffixIndex[j];
    str_2 := stringBaseAddr + suffixIndex[j + 1];
    while ((str_1[i] == str_2[i]) && str_1[i] && str_2[i])
        i++;
    auxiliaryArray [j + 1] := i;
```

4.1.2 Parallel implementation of exact repeats finding

We proposed an algorithm based on parallel suffix array to find out the exact repeats with arbitrary length, which provide a much simple way to solve the DNA sequence repeats problem with higher efficiency than sequential approach for this type of problems. Figure 4.3 shows the flow diagram of our fast exact repeats finding on CPU and GPU. In addition to suffix array an extra space $O(n)$ have been used as *auxiliaryArray* to provide the ease of exact repeats finding. Algorithm 4.3 shows the execution steps in details.

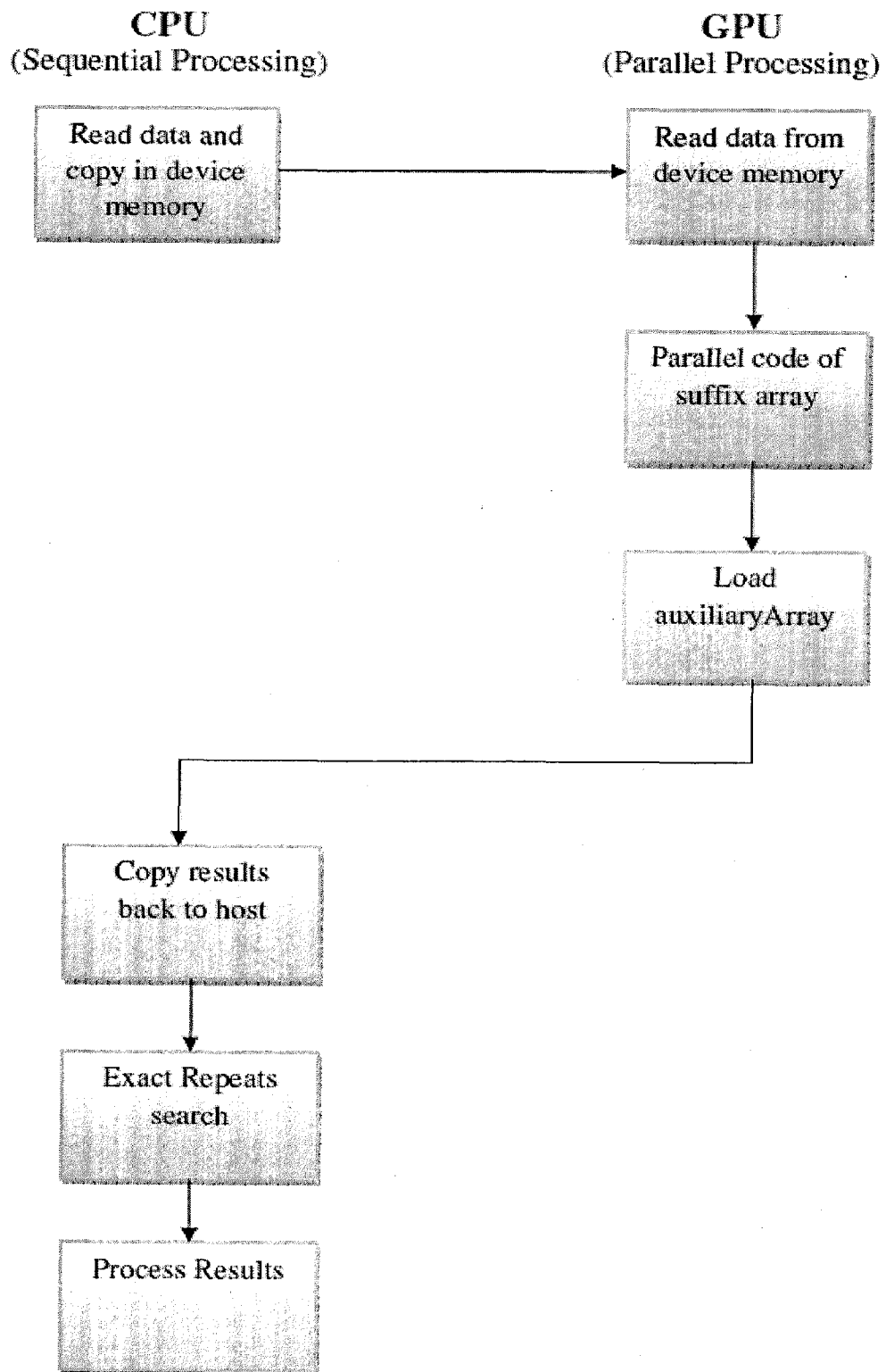


Figure 4.3: Flow diagram of parallel implementation of exact repeats finding

Algorithm 4.3: Parallel implementation for exact repeats finding

```
1) Call algorithm 3.2: Parallel implementation of suffix array (refer page no. 24)
2) Call algorithm 4.4: Loading of auxiliaryArray using cuda (refer page no. 30)
3) for i := 0 to (MAXLENGTH - 1)
    if(auxiliaryArray[i + 1] >= w) then
        flag := 1;
        Print: suffixIndex[i], groupIndex
    else
        if(flag == 1)
            Print: suffixIndex[i], groupIndex;
            flag := 0;
            groupIndex++;
        end if
    end if
```

Algorithm 4.4: Loading of auxiliaryArray using cuda

```
//cuda steps for loading of auxiliaryArray

Initialize threadId := blockIdx.x * 512 + threadIdx.x, i := 0;
if (threadId + 1 > MAXLENGTH) then
    return;
end if
str_1 := stringBaseAddr + suffixIndex[threadId];
str_2 := stringBaseAddr + suffixIndex[threadId + 1];
while ((str_1[i] == str_2[i]) && str_1[i] && str_2[i])
    i++;
auxiliaryArray [threadId + 1] := i;
```

4.1.3 Performance comparison of sequential and parallel exact repeats finding

The linear and parallel programs were executed in a CUDA based machine having Intel Xeon CPU (2.5 GHz) and NVIDIA Quadro FX 3700. The figure 4.4 shows the performance comparison of parallel and sequential implementation and it's very clear that GPU implementation giving the better performance than CPU implementation.

Table 4.1: Timing comparison for exact repeats finding

Input Length	Sequential Processing Time (seconds)	Parallel Processing Time (seconds)	Speed Up
512	0.130170	0.219764	0.592317
1024	1.021849	1.309891	0.780102
2048	8.098172	5.971737	1.356083
4096	64.530180	31.382746	2.056231
8192	483.522656	202.490094	2.387883

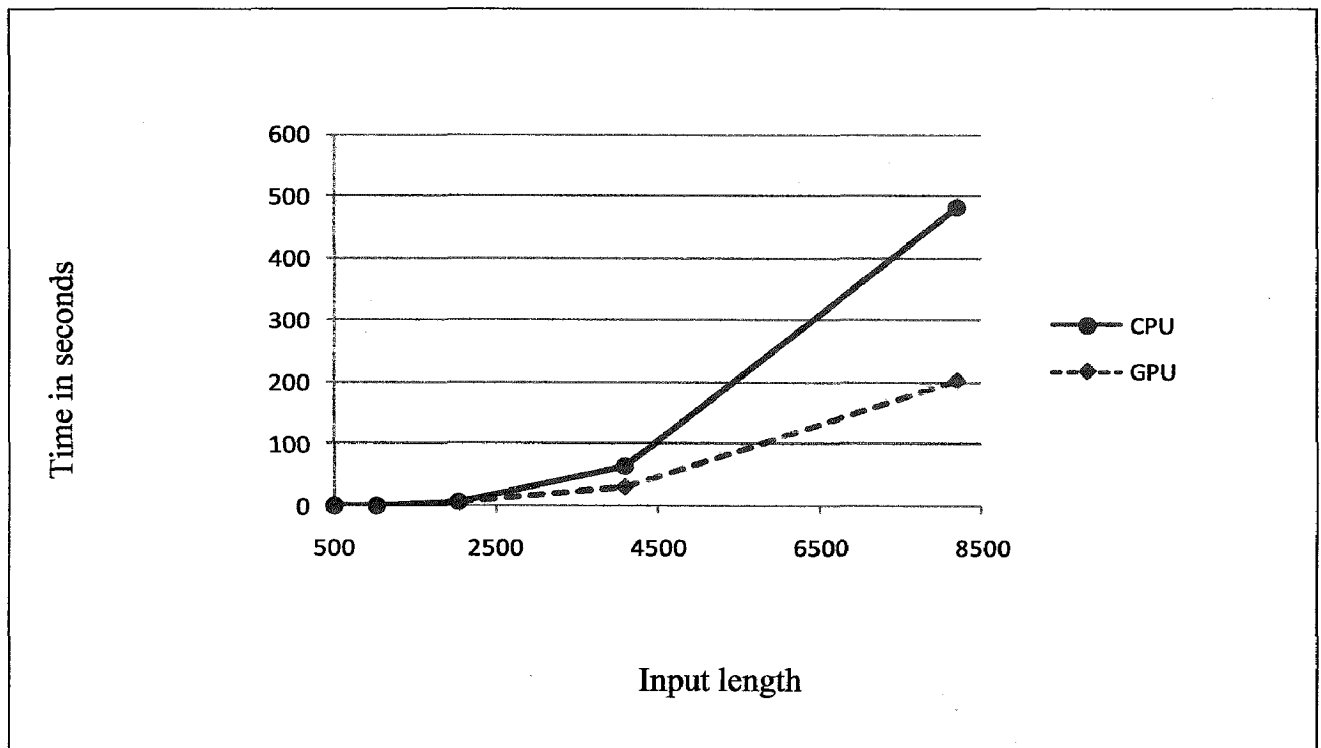


Figure 4.4: Comparison of parallel and sequential implementation of exact repeats finding

4.2 Tandem Repeats

A tandem repeat consists of two or more contiguous copies of a nucleotide pattern [4]. An example would be: A-T-T-C-G-A-T-T-C-G-A-T-T-C-G in which the sequence A-T-T-C-G is repeated three times [6]. In a tandem repeat, four features need to be analyzed: the pattern size, the pattern structure, the number of copies, and the positions of the patterns. According to the pattern size, repeats can be classified into three types: satellites, minisatellites, and microsatellites [4].

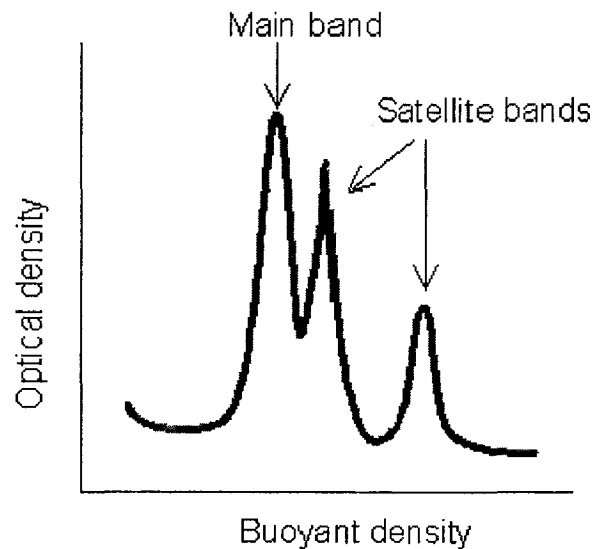


Figure 4.5: Illustration of satellite bands [7]

By using buoyant density gradient centrifugation, DNA fragments with significantly different base compositions may be separated, and then monitored by the absorption spectra of ultraviolet light. The main band represents the bulk DNA, and the "satellite" bands originate from tandem repeats.

Satellites

The size of a satellite DNA ranges from 100 kb to over 1 Mb. In humans, a well known example is the alphoid DNA located at the centromere of all chromosomes. Its repeat unit is 171 bp and the repetitive region accounts for 3-5% of the DNA in each chromosome. Other satellites have a shorter repeat unit. Most satellites in humans or in other organisms are located at the centromere [7]. Figure 4.6 shows the a partial human SFR profile obtained using Applied Biosystems Identifiler kit.

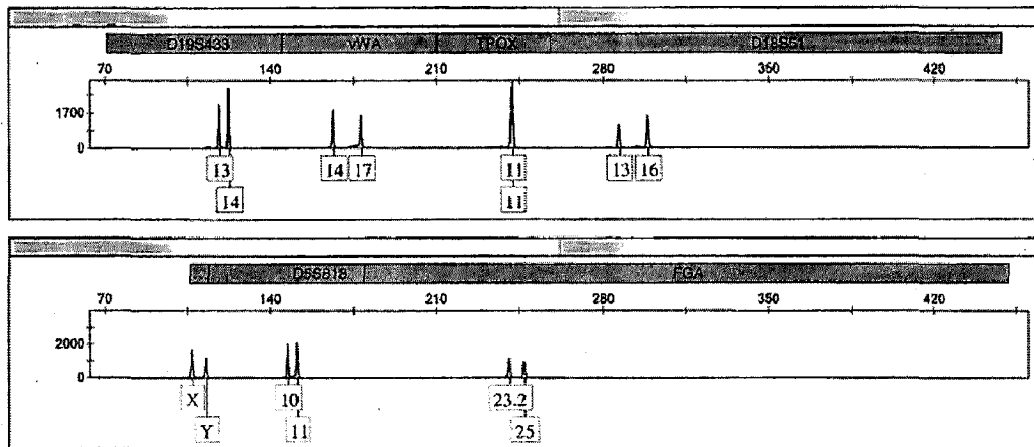


Figure 4.6: A partial human STR profile obtained using Applied Biosystems Identifiler kit [6]

Minisatellites

The size of a minisatellite ranges from 1 kb to 20 kb. One type of minisatellites is called variable number of tandem repeats (VNTR). Its repeat unit ranges from 9 bp to 80 bp. They are located in non-coding regions. The number of repeats for a given minisatellite may differ between individuals. This feature is the basis of DNA fingerprinting. Another type of minisatellites is the telomere. In a human germ cell, the size of a telomere is about 15 kb. In an aging somatic cell, the telomere is shorter. The telomere contains tandemly repeated sequence GGGTTA [7].

Microsatellites

Microsatellites are also known as short tandem repeats (STR), because a repeat unit consists of only 1 to 6 bp and the whole repetitive region spans less than 150 bp. Similar to minisatellites, the number of repeats for a given microsatellite may differ between individuals. Therefore, microsatellites can also be used for DNA fingerprinting [7].

One of the most interesting features of prokaryotic and eukaryotic genomes (both coding and non-coding regions) is the presence of relatively short perfect tandemly repeated DNA sequences. These repeated DNA sequences are distributed almost at random throughout the genome. Repeats containing DNA sequences have attracted much attention from researchers since (i) they play important roles in the formation of hairpin structures that may provide some structural or replication mechanism, (ii) they are often associated with neurological disorders and

(iii) they are used as DNA markers, such as microsatellites or Simple Sequence Repeats (SSR), Inter Simple Sequence Repeats (ISSR) and Directed Amplification of Minisatellite DNA (DAMD-PCR) in Marker Assisted Selection (MAS), positional cloning, identification of quantitative and qualitative loci and mapping for breeding and evolutionary studies. Recent evidence also suggests that some Variable Number of Tandem Repeats (VNTRs) and SSR sequences play significant roles in the regulation of transcription, and that some may also influence the translational efficiency or stability of mRNA, or modifies the activity of proteins by altering their structure [9].

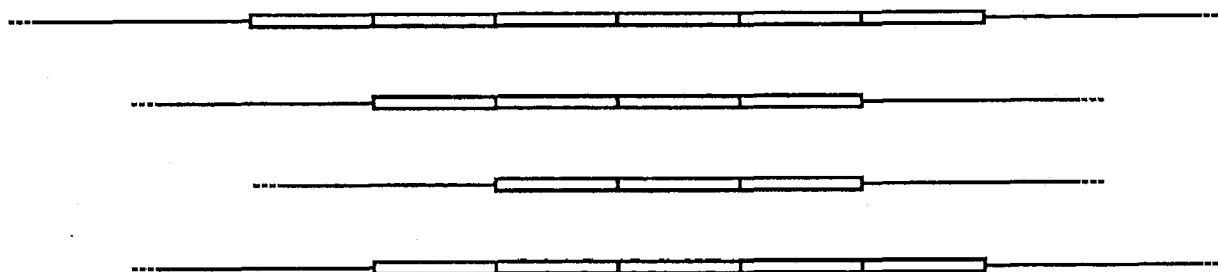


Figure 4.7: Schematic of a Variable Number of Tandem Repeats in 4 alleles [6]

Expressed Sequence Tags (ESTs) are single-pass DNA sequences, usually about 300–500 nucleotides in length, obtained from mRNA (cDNA) representing genes expressed in a given tissue and/or at a given development stage. A typical EST usually contains only a portion of the coding region (either translated or untranslated, or both) of the original gene transcript. One of the useful applications of ESTs is in the study of the gene expression pattern in a given organ, tissue or development stage in response to a particular treatment. The composition of a tissue specific EST population, therefore, offers an overall overview of the expressed genes and, consequently, is a novel tool in gene discovery and in understanding the biochemical pathways involved in physiological responses. ESTs have also been mined for Single Nucleotide Polymorphisms (SNP) and SSR. Microsatellites or SSRs are stretches of DNA consisting of exact simple tandemly repeated short motifs of 1–6 base pairs in length. SSRs are one of the best DNA markers because they are highly polymorphic, inherited in a co-dominant fashion, and highly abundant, being dispersed evenly throughout the genome. They can serve as sequence-tagged sites for anchoring in genetic and physical maps. The standard procedure for developing

SSRs involves the construction of a small-insert genomic library, its subsequent hybridization with tandemly repeated oligonucleotides, and the sequencing of candidate clones. Unfortunately, this process is time consuming and labourintensive [9].

4.2.1 Sequential implementation of tandem repeats finding

We have proposed an algorithm based on suffix array to find out the exact repeats with arbitrary length, which provide a much simple way to solve the DNA sequence repeats problem. The figure 4.8 shows the flow diagram of our sequential implementation. The algorithm 4.5 shows the actual implementation in uniprocessor.

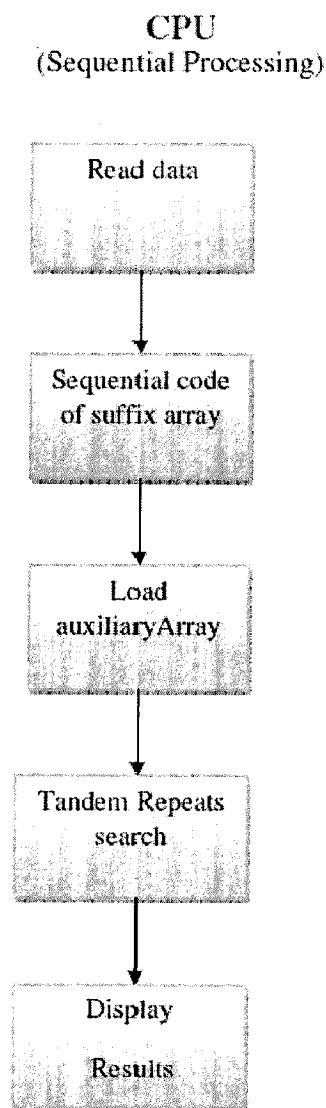


Figure 4.8: Flow diagram of sequential implementation of tandem repeats finding

Algorithm 4.5: Sequential implementation for tandem repeats finding

- 1) Call algorithm 3.1: Sequential implementation of suffix array (*refer page no. 21*)
- 2) Call algorithm 4.2: Loading of auxiliaryArray (*refer page no. 28*)
- 3) Call algorithm 4.6: Function for tandem repeats search (*refer page no. 36*)

Algorithm 4.6: Function for tandem repeats search

Step 1: Top down pass

```
for i := 0 to (MAXLENGTH - 2)
    if(flag == 0) then
        temp := auxiliaryArray[i + 1];
    end if
    if(suffixIndex[i + 1] == (suffixIndex[i] - temp))then
        Print: suffixIndex[i];
        flag := 1;
        Goto step 1;
    end if
    if(flag == 1)then
        Print: suffixIndex[i];
        flag := 0;
    end if
```

Step 2: Bottom up pass

```
Initialize flag := 0;
for i := MAXLENGTH to 1
    if(flag == 0)then
        temp := auxiliaryArray[i];
    end if
    if(suffixIndex[i - 1] == (suffixIndex[i]-temp)) then
        Print: suffixIndex[i];
        flag := 1;
        Goto step 2;
    end if
    if(flag==1)
        Printf: suffixIndex[i];
        flag := 0;
    end if
```

4.2.2 Parallel implementation of tandem repeats finding

We proposed an algorithm based on parallel suffix array to find the exact repeats with arbitrary length, which provide a much simple way to solve the DNA sequence tandem repeats problem with higher efficiency than sequential approach. In this algorithm tandem repeats group occurs either in ascending order or descending order so we have used two passes for that (i) Top down pass, if tandem repeated sequence occurs in ascending order (ii) Bottom up pass, if tandem repeated sequence occurs in descending order and before execution of above passes, executed the parallel version code of suffix array and parallel loading of auxiliary array. Figure 4.9 shows the flow diagram of fast tandem repeats finding and algorithm 4.7 shows the execution steps in details.

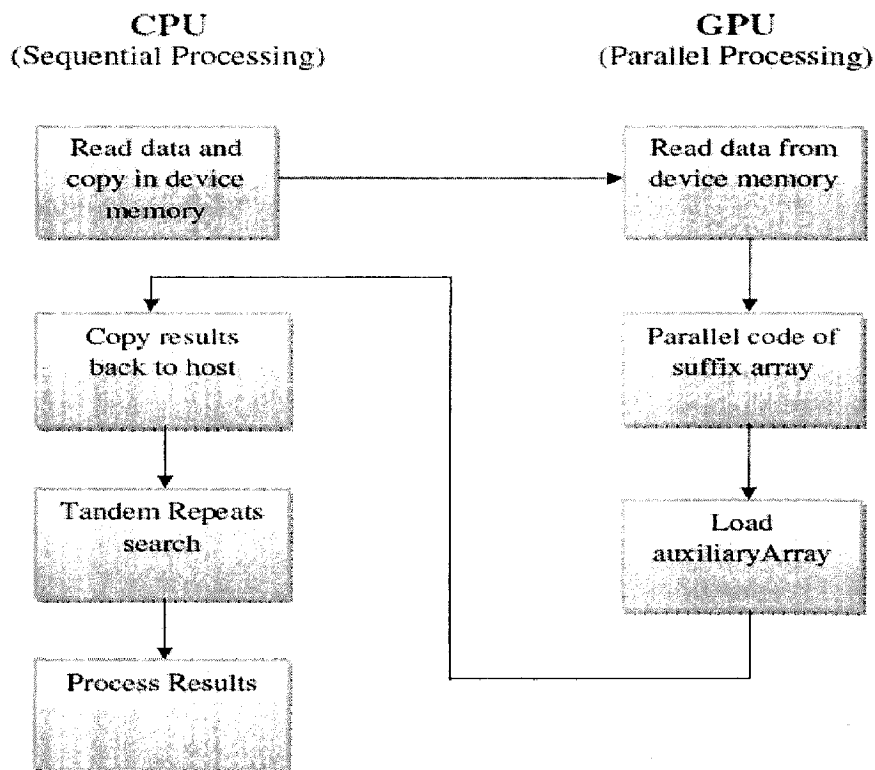


Figure 4.9: Flow diagram of parallel implementation of tandem repeats finding

Algorithm 4.7: Parallel implementation for tandem repeats finding

- 1). Call algorithm 3.2: Parallel implementation of suffix array (*refer page no. 24*)
- 2). Call algorithm 4.4: Loading of auxiliaryArray using cuda (*refer page no. 30*)
- 3). Call algorithm 4.6: Function for tandem repeats search (*refer page no. 36*)

4.2.3 Performance comparison of sequential and parallel tandem repeats finding

The linear and parallel programs were executed in a CUDA based machine having Intel Xeon CPU (2.5 GHz) and NVIDIA Quadro FX 3700. The figure 4.10 shows the performance comparison of parallel and sequential implementation and it's very clear that GPU implementation giving the better performance than CPU implementation.

Table 4.2: Timing comparison for tandem repeats finding

Input Length	Sequential Processing Time (seconds)	Parallel Processing Time (seconds)	Speed Up
512	0.129631	0.220139	0.588859
1024	1.020677	1.310667	0.778746
2048	8.097908	5.969227	1.356609
4096	64.629652	31.351945	2.061424
8192	483.478469	202.494625	2.387611

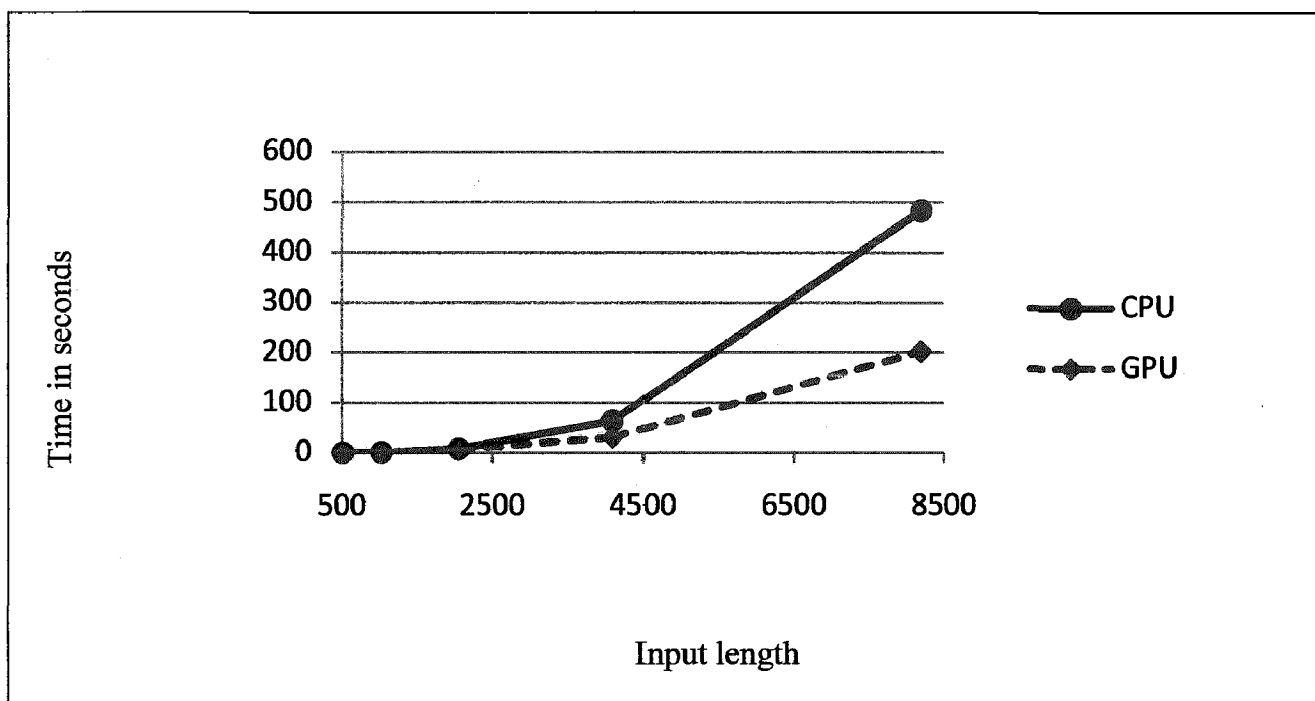


Figure 4.10: Comparison of parallel and sequential implementation of tandem repeats finding

4.3 Exact String Matching

String matching has a long history in computational biology with roots in finding similar proteins and gene sequences in a database of known sequences.

Genetic information is passed from parent to offspring in the biological macromolecule DNA, and is encoded in the individual's sequence of 4 different nucleotides. The full sequence of nucleotides for an organism, its genome, varies in length from hundreds of thousands of nucleotides for genetically simple organisms, such as bacteria, to billions of nucleotides for genetically complex organisms, such as humans. Genes are the regions of the genome that encode for proteins, which are the functionally active molecules in an organism. Gene sequences specify a protein's sequence of component molecules called amino acids. Each triple of nucleotides in the DNA of a gene specifies one of the 20 possible standard amino acids. The chain of hundreds or thousands of amino acids specified by the gene folds into a very complex configuration, and the physical shape of a folded protein determines its biological function.

Pairs of similar sequences are biologically related under the commonly held assumption that two homologous (highly similar) protein sequences will fold into similar configurations, and thus have a similar biological function. Given a database of sequences with known function and a novel sequence with unknown function, one can use a String matching algorithm to discover homologous sequences, and thereby infer the function of the novel sequence. A String matching algorithm reports the biological similarity between a pair of sequences, by modelling potential mutations between the sequences.

Early work on detecting homologous sequences focused on creating optimal algorithms that would exactly compute and report the similarity score between pairs of sequences. However, improvements in high throughput sequencing technology led to an explosion in number of known sequences, and motivated research for more efficient methods for detecting similar sequences [8]. This inspired the development of heuristics that find highly similar alignments very quickly at the cost of potentially not reporting lower similarity sequences.

Computational biologists entered into the era of comparative genomics, in which the entire genomes of organisms are compared. In comparative genomics, the relatively simple techniques

used for scanning a single short query sequence against a database of known sequences proved too inefficient yet again, and required the use of even faster algorithms, using sophisticated data structures such as suffix array.

The earliest algorithms, such as the Needleman-Wunsch global alignment and Smith-Waterman local alignment algorithms from the 70s and early 80s, were developed to detect similarity between a pair of DNA or protein sequences. These early algorithms use dynamic programming to compute optimal alignments between a pair of sequences in time proportional to the product of their lengths.

In the 80s and 90s, the number of DNA and protein sequences grew exponentially and it quickly became infeasible to compute an optimal alignment between a query sequence and every sequence in the database of known sequences. As a result, researchers developed heuristic techniques such as FASTA and BLAST to more quickly discover highly similar alignments, while not reporting more distant alignments. Researchers found this to be an acceptable tradeoff between sensitivity and speed, since in most applications only the highly similar alignments are relevant. Consequently, BLAST became the de facto bioinformatics tool of choice for finding homologous gene or protein sequences.

These techniques use the key insight that two highly similar sequences must share common substrings, although possibly separated by relatively short sequences of mutated residues. They use this insight by pre-processing the database to construct a catalog of the short fixed-length substrings found in the database sequences. After this pre-processing, each query string is processed by first finding exact matches between substrings of the query string and the catalog of database substrings. Using a hash table or similar technique, this executes in time proportional to the number of occurrence of each short substring in the database. These short exact matches then act to seed longer, possibly inexact alignments, by connecting together overlapping exact matches, and using Smith-Waterman local alignments to align regions with mismatching characters. The time to pre-process the database may be considerable, but is amortized by constructing it once for many query searches, after which the seed-and-extend style algorithms run in time proportional to the length of the query string and relatively independent of the number of strings in the database.

Unlike protein searches, comparative genomics commonly performs comparisons between the much longer whole genome sequences of related organisms. As such, the two genomes will typically share many very long substrings. Consequently, BLAST and related tools which seed their alignments using short fixed length substrings will process many seeds from these exact matches, and thus run relatively inefficiently. This motivated research in the late 90s for new alignment algorithms which could be used for aligning sequences of any length and potentially containing very long exact matches.

A suffix tree [15] can be naively constructed in $O(n^2)$ time and $O(n)$ space by iteratively inserting all n suffixes, but can be constructed in $O(n)$ time and $O(n)$ space for a string over a fixed alphabet, such as for DNA or amino acids, by more carefully exploiting the relationships between suffixes. The more sophisticated algorithms make extensive use of additional pointers called suffix links which connect nodes along the path to the i^{th} suffix to same relative position on the $i+1^{\text{th}}$ suffix.

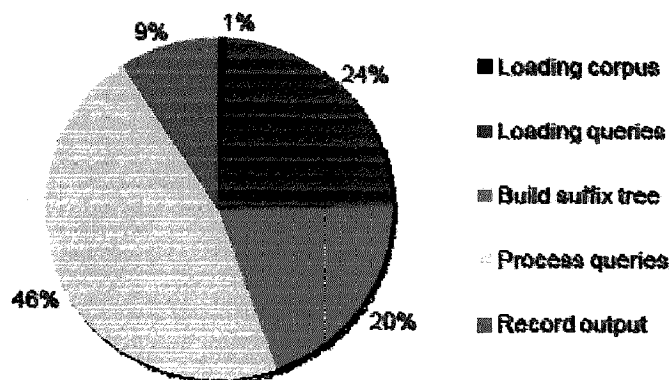


Figure 4.11: The time spent in each phase of the suffix tree matching program on the CPU [8]

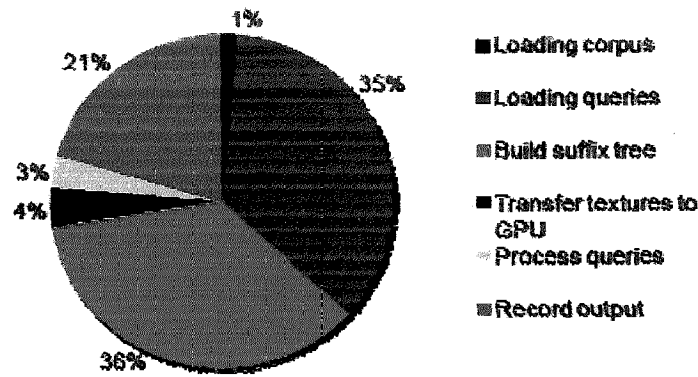


Figure 4.12: The time spent in each phase of the suffix tree matching program on GPU [8]

As shown in figure 4.11 and figure 4.12 that GPU implementation of suffix tree gives the better query processing time over CPU and as we already discuss the suffix array can be alternatively used in place suffix tree so its GPU implementation will give the better performance over its CPU implementation. Even with the highly efficient algorithms and data structures invented, modern computational biologist still rely on computational grids consisting of many computers executing algorithms in parallel to increase the throughput of their searches. Our work acts to replace an entire computational grid of computers used for string matching with a single highly parallel commodity multiprocessing board, in the form of a high performance Graphics Processing Unit (GPU) programmed in the Compute Unified Device Architecture (CUDA) framework [5].

4.3.1 Sequential implementation of exact string matching

We have proposed an algorithm based on suffix array to find out the exact repeats with arbitrary length, which provide a much simple way to solve the sequence matching problem. The figure 4.13 shows the flow diagram of our sequential implementation and algorithm 4.8 has been used for implementation.

Algorithm 4.8: Sequential implementation for exact string matching

- 1) Call algorithm 3.1: Sequential implementation of suffix array (*refer page no. 21*)
- 2) Call algorithm 4.9: Function for exact string matching (*refer page no. 43*)

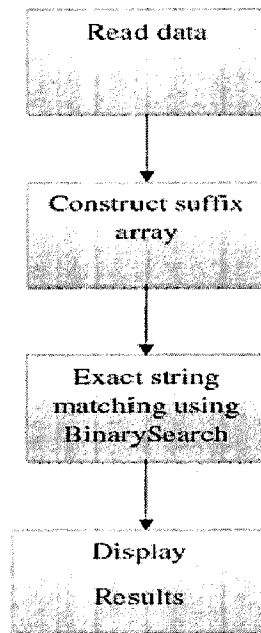


Figure 4.13: Flow diagram of sequential implementation of exact string matching

Algorithm 4.9: Function for exact string matching

```

Initialize end := MAXLENGTH - 1, mid := (beg + end)/2;
while(beg <= end && my_strcmp(stringBaseAddr+suffixIndex[mid],pattern,pattern_len) != 1)
  if(strcmp(pattern,str + suffixIndex[mid]) < 0) then
    end := mid - 1;
  else
    beg := mid + 1 , mid := (beg + end)/2;
  end if
if(my_strcmp(pattern,str+suffixIndex[mid], pattern_len) == 1)
then
  up := down := mid;
  while((auxiliaryArray[up] >= pattern_len) && (up>0))
    up--;
    Print: suffixIndex[up]
  while((auxiliaryArray[down + 1] >= pattern_len)
  && ((down + 1) <= MAXLENGTH))
    down++;
    Print: suffixIndex[down];
end if
  
```

4.3.2 Parallel implementation of exact string matching

We proposed an algorithm based on parallel suffix array to find out the exact string matching with arbitrary length, which provide a much simple way to solve the sequence search problem with higher efficiency than sequential approach. Figure 4.14 shows the flow diagram of our fast exact string matching on CPU and GPU.

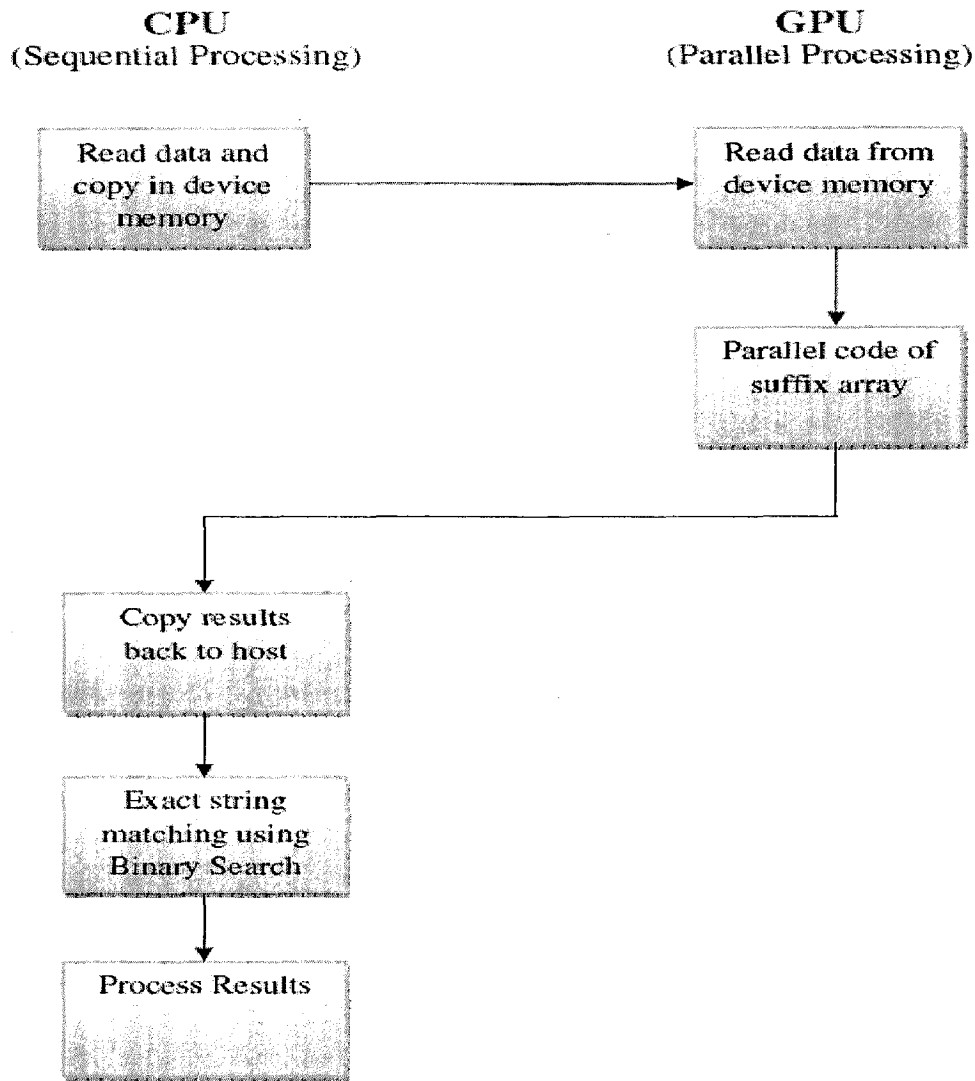


Figure 4.14: Flow diagram of parallel implementation of exact string matching

Algorithm 4.10: Parallel implementation for exact string matching

- 1) Call algorithm 3.2: Parallel implementation of suffix array (*refer page no. 24*)
- 2) Call algorithm 4.9: Function for exact string matching (*refer page no. 43*)

4.3.3 Performance comparison of sequential and parallel exact string matching

The sequential and parallel programs were executed in a CUDA based machine having Intel Xeon CPU (2.5 GHz) and NVIDIA Quadro FX 3700. The figure 4.15 shows the performance comparison of parallel and sequential implementation and it's very clear that GPU implementation giving the better performance than CPU implementation.

Table 4.3: Timing comparison for exact string matching

Input Length	Sequential Processing Time (seconds)	Parallel Processing Time (seconds)	Speed Up
512	0.029631	0.044035	0.672889
1024	1.029656	0.983634	1.046787
2048	4.098279	2.363038	1.734326
4096	52.654532	24.529133	2.146612
8192	287.783169	89.065052	3.231157

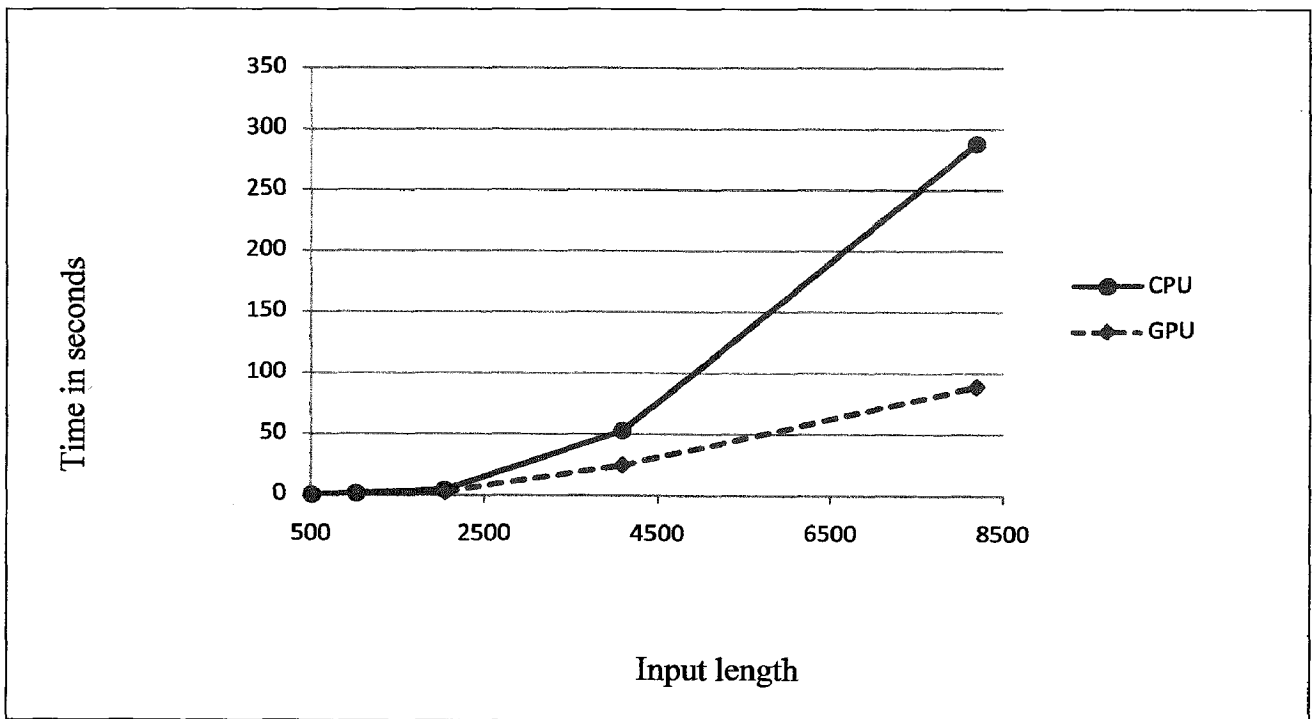


Figure 4.15: Comparison of parallel and sequential implementation of exact string matching

Chapter 5: Conclusion and future works

As we have seen that our parallel algorithm giving better performance than CPU version code for large size of input data elements. For small size input data the GPU implementation is slower than CPU implementation.

The major bottle neck for performance in the Suffix Array algorithm was the process of sorting the intermediate data. The performance improvement of Suffix Array on CUDA was majorly due to the possibility of parallel sorting which took bulk of the runtime in the sequential implementation. The running time of the algorithm can be improved if the sorting part of the algorithm is implemented using $O(\log n)$ techniques of parallel sorting. However this would require that such a technique is implemented without recursion.

Future programming model may be the hybrid of current serial CPU and data parallel GPU execution paradigm for high performance computing so we will try to employ these codes as the fundamental functions to solve more complex bioinformatics and text processing applications to meet the high computing power requirements.

References

- [1] W. Sun and Z. Ma. A Fast Exact Repeats Search Algorithm for Genome Analysis. In Proc. 9th International Conference on Hybrid Intelligent Systems (HIS'09), Shenyang, China, pp.427-430, 2009.
- [2] W. Sun and Z. Ma. Parallel Lexicographic Names Construction with CUDA. In Proc. 15th International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, pp.913-918, 2009.
- [3] T. Gharib. A hybrid approach for indexing and searching protein structures. WSEAS Transactions on Computers, 8(6), pp.966-975, 2009.
- [4] H. Zhou, L. Du and H. Yan. Detection of Tandem Repeats in DNA Sequences Based on Parametric Spectral Estimation. IEEE Transactions on Information Technology in Biomedicine, 13(5), pp.747-755, 2009.
- [5] CUDA Programming Guide, Version 2.3, NVIDIA, January 2009.
- [6] http://en.wikipedia.org/wiki/Tandem_repeats (Last accessed on 20th May 2010).
- [7] <http://www.web-books.com/MoBio/Free/Ch3G1.htm> (Last accessed on 20th May 2010).
- [8] C. Schatz and C. Trapnell. Fast exact string matching on the gpu, Technical Report. Available from: <http://www.cbcb.umd.edu/software/cmhatch> (Last accessed on 20th May 2010).
- [9] M. Karaca, M. Bilgen, A. Onus, A. Ince and S. Elmasulu. Exact Tandem Repeats Analyzer (E-TRA): a new program for DNA sequence mining, *Journal of Genetics*, 8(4). pp.49–54, 2005.
- [10] J. Kärkkäinen, P. Sanders and S. Burkhardt. Linear work suffix array Construction. *J. ACM*, 53(6), pp. 918-936, 2006.

- [11] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03), LNCS 2719, Springer, pp. 943-955, 2003.
- [12] http://en.wikipedia.org/wiki/Suffix_array (Last accessed on 20th May 2010).
- [13] T. Cormen, C. Leiserson, R. Rivest and C. Stein. Introduction to Algorithms. 2nd edition. MIT Press and McGraw-Hill, 2001.
- [14] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. SIAM Journal of Computing, 22(5), pp.935–948, 1993.
- [15] D. Gusfield Algorithms on strings, trees and sequences: computer science and computational biology. New York: Cambridge University Press, 1997.