Information **S**ciences and **C**omputing
Volume 2015, Article ID ISC520415, 27 pages
Available online at http://www.infoscicomp.com/

Research Article

High Performance Implementation for Aligning Compressed DNA Sequences

D. Satyanvesh[a], B. Kaliuday[a] and P.K. Baruah[a]

[a] Department of Mathematics and Computer Science, Sri Sathya Sai Institute of Higher Learning, Andhra Pradesh, India

Corresponding author: D. Satyanvesh; E-mail: Satyanvesh.d@gmail.com

Received 25 April 2015; Accepted 08 June 2015

Abstract: In molecular biology, sequence alignment is a way of arranging DNA, RNA or protein sequences to identify regions of similarity between the sequences. However, this is a challenging problem since these sequences such as DNA are huge in size and the databases are growing at an exponential rate. It requires tremendous amount of memory and large computational power. For example, the human genome in raw format ranges from 2 to 30Tera-bytes. The growth of the DNA affects the storage as well as bandwidth when these sequences need to be transferred. We need to utilize the repeats present in the DNA sequences to compress them. Applications such as DNA profiling, real time DNA crime investigation require access to the DNA sequences in real time. However, the applications mentioned not only require good compression ratio but also needs faster compression. As part of the first phase, we propose a new algorithm with a focus on the through put along with the compression ratio. The algorithm scales well and achieves a speed up of 11 on multi-cores and upto 23 on GPUs when run on M2070 Tesla card and up to 57on K20 Kepler GPUs. In the second phase, we present two types of alignment namely ungapped and gapped alignment. Multi-cores and GPUs can be used to align the sequences quickly. The focus mainly is on aligning the huge sequences accurately. The ungapped alignment achieves a speedup of up to 56 on K20 Kepler GPUs and the gapped alignment achieves a speedup of up to 15 on 16-core Intel Xeon E5-2680 processor.

Keywords: DNA Sequence; Throughput; Compression Ratio; Code byte; Ungapped alignment; Gapped alignment.

1. Introduction

In bioinformatics, sequence alignment is a way of arranging the sequences of DNA, RNA or protein to identify regions of similarity that may be a consequence of functional, structural or evolutionary relationships between the sequences. Similarities between the sequences lead to discovering the evolutionary relationship

between the species and to annotate the new species. It also helps in comparing the unknown sequences against the already existing sequences in the large databases.

There are two kinds of sequence alignment. Global alignment spans entire sequence and it attempts to align every base in all the sequences. It is suitable if the sequences are of equal size and quite similar. It's also useful for sequences with similar structures and functions such as proteins or genes. Local alignment is a form of alignment which identifies the similar subsequences in the long sequences that are not similar to each other. It is very useful for divergent sequences that may contain parts of similarity. It searches for the conserved regions and so it helps in analyzing long sequences such as chromosomes or genomes.

There are many existing multiple sequence alignment methods such as dynamic programming, iterative methods and progressive methods but most of the existing methods use dynamic programming. This method uses all the sequences pair wise in the given query-set. The alignment is calculated by observing the matches and gaps at intermediate positions. Finally the alignment is constructed using these small sub problems. This technique is computationally expensive as both the time and space complexities are quadratic with respect to the length of the sequences. This method is very useful when the sequences used are very small.

To reduce the time, some of the heuristic methods such as FASTA and BLAST are used but at the cost of sensitivity. These algorithms first search the seeds of consecutive matches instead of searching every single base of the two sequences. The seeds are then extended to by a limited use of dynamic programming to allow gaps and insertions. These methods are up to 40 times faster than the normal CPU based implementation of Smith-Waterman algorithm. Even though the CPU based implementation of FASTA and BLAST works faster, still they are time consuming they take a considerable amount of time when the databases are large.

In molecular biology, the genome consists of all the hereditary information for running and maintaining an organism. This biological information contained in genome is encoded in the form of DNA. DNA chain is made of four bases: Adenine (A), Guanine (G), Thymine (T), and Cytosine (C). When the cells divide to grow, every new cell needs a copy of the DNA to function properly. So, DNA replicates itself before the cell divides. Due to this, the genomic data increases constantly which leads to doubling of the DNA sequences. DNA also has many repeats. This property can be used to compress the data.

As described earlier, the alignment process takes large amounts of memory and space, we need to compress the data in order to save the storage space as well as the time taken when these huge sequences need to be transferred from the centralized servers to the local systems.

General purpose compression algorithms such as gzip, bzip2 do not work well for the DNA sequences since it consists of only 4 bases namely A, T, G and C. As a result, these algorithms expanded the DNA sequence instead of compressing it [1]. Other algorithms such as Biocompress-2 [2], Gen Compress [3], DNA Compress [4], DNABIT [5] and GENBIT [6] have been used in the recent years to compress the DNA sequences.

An important piece of information contained in DNA sequence is tandem repeats. But all the algorithms take quadratic or more amount of time for searching those tandem repeats in a huge DNA sequence [7]. Applications such as DNA profiling, real time DNA crime investigation require access to the DNA sequences in real time. So, the compression must be very quick. The challenging problem is to achieve high through put along with a better compression ratio. In recent days, the evolution in process or architecture starting from dual-core to many cores helps in achieving this challenge. Multi-cores and GPUs hold promise for faster processing. Therefore any algorithm should be adaptable to such architecture in order to achieve good throughput.

After this stage, the compressed sequences are used for alignment which helps in reducing the time required for searching as well as computing. This helps the process of alignment of huge sequences very fast and accurate. In this paper, we address these issues by using graphical processing units (GPUs) and multi-cores.

The rest of the paper is organized as follows: In section 2, we briefly discuss the related work. In section 3, we discuss the details of our proposed algorithm. Section 4 gives the implementation detail so four algorithm. Section 5 describes the experimental setup. Section 6 discusses the results. Conclusion and future work are dealt in section 7.

2. Related Work

Grumbach and Tahi [2] proposed two lossless compression algorithms for DNA sequences, namely Bio Compress and Bio Compress-2, making use of the Ziv and Lempel data compression method [8]. Bio Compress is based on the substitution of factors with shorter references to earlier occurrences of identical factors or complementary factors (palindromes). It encodes a text on the four letter alphabet A, C, G, T into a binary sequence. They proposed two methods. In the first one [8], the occurrences of the factor to encode are searched in a window. The second one [9] is based on a dictionary containing the already encoded factors.

BioCompress-2 finds both the exact and reverse repeats in the target sequence. It encodes them by repeat length and the position of a previous repeat occurrence. If there is no significant repetition then the arithmetic coding of order-2 is used to reduce the number of bits used. The only difference between BioCompress and BioCompress is the use of arithmetic coding.

Gencompress [3] is a one-pass algorithm that searches for the approximate matches. This algorithm uses order-2 arithmetic encoding [1]. Gencompress detects the approximate complemented palindrome (A replaced by T and C replaced by G) in DNA sequences. For input w, assume that a part of it, say v, has already been compressed, and the remaining part is u; i.e., w=vu. GenCompress finds an optimal prefix of u such that it approximately matches some substring in v so that this prefix of u can been coded economically. After outputting the code of this prefix, remove the prefix from u, and append it to the suffix of v. Continue the process until u= where is an empty string.

There are many ways to approximate a string from others. GenCompress adopts a constraint to limit the search. If the number of edit operations located in any substring of length k in the prefix s of u for an edit operation sequence $\lambda(s, t)$ is not larger than a threshold value b, then it is considered that $\lambda(s, t)$ satisfies the condition C = (k, b) for compression. In GenCompress, search is done only for approximate matches that satisfy condition C. In this way, the search space is limited. The average compression ratio is 1.7428 bits/bytes. Gencompress [3] achieves higher compression ratios compared to Biocompress or Biocompress-2.

DNACompress [4] uses Lempel-Ziv compression scheme as BioCompress and BioCompress-2. It finds all the approximate repeats including complemented palindromes and encodes approximate repeat regions and non-repeat regions. It mainly concentrates on approximate repeats. But searching all the approximate repeats that are optimal for compression is time-consuming. So this algorithm uses a software tool named PatternHunter [10] which is used for fast and sensitive homology search. PatternHunter is a search engine which provides all the approximate repeats with highest score including complemented palindromes.

DNACompress mainly consists of 4 phases: Firstly, Run the PatternHunter and output all the approximate repeats into a list A in the order of descending scores. Next, Extract a repeat r with highest score from list A and add r into another repeat list B. Then Process each repeat in A so that there is no overlap with the extracted repeat r. If the highest score of repeats in A is still higher than a pre-defined threshold then go to step 2. Else exit.

In order to recover an approximate repeat correctly the following information must be encoded. One bit to show which kind of repeat it is, forward repeat or complemented palindrome. A triple (l, i, j). It is used to copy a previous substring of length l starting at i to the current position j; the total number of edit operations contained in this approximate repeat.

It also requires all the triples (e, o, b) where e indicates which kind of edit operation it is, o means its location offset in the repeat and b a base character that will be used by a substitute or insert edit operation. They are used to edit the copied substring. Instead of encoding each edit operation separately, a consecutive region of the same edit operation (or say a block edit operation) can alternatively employ a more efficient encoding method.

DNACompress checks each repeat to see whether it saves bits to encode. If not, it will be discarded. At the end, all the remaining regions other than repeats are concatenated together and then sent as input to a two-order arithmetic coder. The average compression ratio is 1.7254 bits/bytes.

GENBIT Compress algorithm [6] uses a little different method in which each input sequence is divided into fragments of 4 characters each. Hence each fragment can be encoded in 8 bits as each character is represented using 2 bits. If the consecutive fragments are same, then the specific 9th bit is set to 1. If the consecutive fragments are different, then the specific 9th bit is set to 0 for that 8 bit unique representation. This algorithm does not use dynamic programming approach. It takes an input DNA sequence of length n and divides it into n/4 no. of fragments. The remaining

characters or bases (fragment length less than 4) are assigned unique 2 bits (A=00, g=01, c=10, t=11). This algorithm explains about different cases such as DNA sequence with same fragments, DNA sequence with different fragments etc. The decoding process is just opposite to the encoding process where first the given binary code is divided into fragments of 9 characters each. In each fragment, if the 9th bit equals 1, then the corresponding combination is taken two times. Otherwise, it is just considered once. This algorithm just searches and encodes exact re- peats instead of searching and encoding approximate repeats. This also works better for up to sequences of 8 lakh characters. The average compression ratio for this algorithm is 1.727 bits/bytes with the best and worst cases being 1.125 bits/bytes and 2.238 bits/bytes respectively.

The literature shows that the existing algorithms concentrate mainly on compression ratio whereas the proposed algorithm and its parallel implementation not only achieves good compression ratio but also has a better compress throughput.

In [11], a CUDA based solution for Needleman-Wunsch algorithm is implemented. Needleman-Wunsch algorithm uses dynamic programming method and alignment is done using a two-dimensional matrix. Each cell represents a pair of one letter from each sequence. Each cell consists of two values, score and a pointer.

The algorithm consists of three steps:

– **Initialization:** In this step, both the first row and the first column are initialized with zeroes.

– **Fill:** The cells in the matrix are filled by calculating the scores and the pointers. The score of each cell is obtained by finding the maximum value among three scores: a match score, a vertical gap score and a horizontal score (So all three cells which include left cell, top cell and the diagonal cell are required to calculate the present cell).

– **Trace-Back:** This step finds the actual alignment from the calculated matrix. The process starts from the bottom-right corner and follows the path or the pointers until get into the beginning.

Various memory level patterns on GPUs are compared to determine the better parallelization method. The different levels of kernel access and various thread utilization methods have been discussed in [11].

The Needleman-wunsch algorithm can be parallelized only in the fill step. According to the pattern in fill-step, there is no dependency between the scores of all the cells on every minor-diagonal. So the scores of all the cells on every minor-diagonal can be calculated in parallel.

Another method used for parallelizing the algorithm is *minor-diagonal-wise blocking-strategy.* Two types of parallelism can be identified with this strategy.

– The threads in a single block can compute the cells in parallel.

– The cells within the different blocks can be computed in parallel.

In [12], an efficient parallel method was developed to optimize the alignment using smith-waterman algorithm.

– The traditionally used parallelization strategy i.e., the wave front method is described here. The cells which are on the same anti-diagonal in a wave front pattern can be computed in parallel. Here, a key anti-diagonal (KAD) is nothing but two consecutive anti-diagonals whose values are stored in the matrix. The method used here is based on storing x KADs on different cores and recalculating the needed cells in parallel.

In [13], some more novel single-GPU parallelizations are done on Smith- Waterman algorithm which demonstrates an order of magnitude reduction in run time relative to competing GPU algorithms.

– A method named *striped SW algorithm* was introduced here which divides the matrix to be calculated into stripes of width S. Each SM in the GPU works on its assigned strips serially from left to right. Also the same anti- diagonal parallelization strategy is followed in each strip.

– One more method named *chunked alignment* was also proposed where each strip is partitioned into chunks of height h. Each chunk is stored in global memory. The sub paths of the optimal alignment path within each strip are determined using the data stored in the global memory.

The alignment algorithms focused on parallelizing the fill step, but the proposed algorithm ruminates on how fast the sequences can be aligned. So the main focus here is to align the compressed sequences so that the computational resources used are less.

3. Proposed Algorithm

3.1 GenCodex Compression Algorithm

The proposed method is efficient in compressing both repetitive and non-repetitive DNA sequences. The input sequence is divided into fragments of 4 characters each.

In the first phase, each character is represented using two bits namely, A=00, C=01, G=10, T=11. So each fragment is stored using 8 bits i.e., using just one byte. At the end of this phase, we get the compressed sequence where 4 characters of the original sequence are encoded into a single byte.

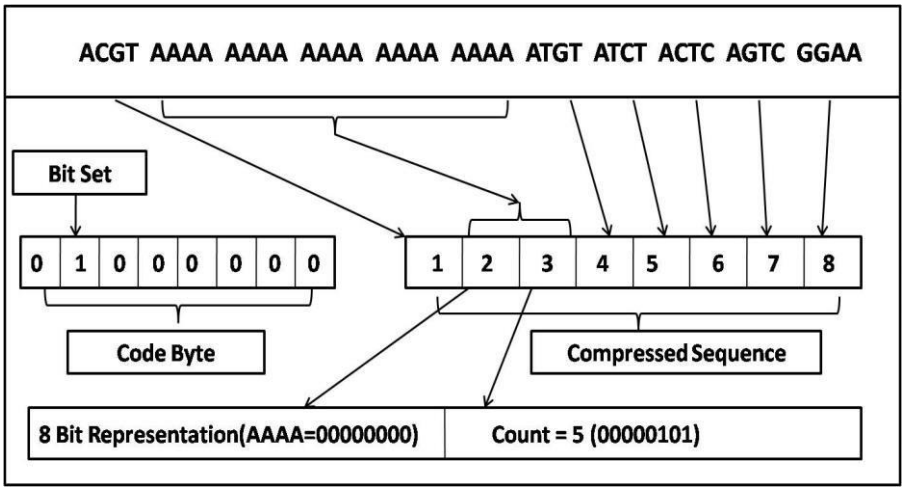

Fig. 1: GenCodex algorithm with an example

In the next phase, the fragments are represented using either one or two bytes. If a fragment is not appearing consecutively, then a single byte is allocated using its 8-bit unique representation. If a fragment is repeating two or more times, then the simple 8-bit representation is put in the first byte and the number of repetitions is represented in the second byte. For every eight bytes of the compressed data, we use an extra byte referred as code byte in which we set the corresponding bit to 1 if there is a repetition. So if a bit is 0 in the code byte, only 1 byte is considered in the compressed sequence and if a bit is set to 1 in the code byte, the next 2 bytes are considered together as part of single coding in the compressed sequence. This is shown in the Fig. 1.

The proposed algorithm is named as GenCodex where x signifies the number of repetitions of a fragment occurring consecutively in a sequence. In this paper, we discuss about 256 repetitions occurring consecutively. The same can be extended if the repetitions occur 128, 64, 32, 16 times etc.

Best Case: Consider an input sequence consisting of 4080 characters where each fragment is repeating 255 times consecutively. Since each fragment in the compressed sequence requires 2 bytes, we need a total of 8 bytes i.e., 64 bits for the compressed data and one byte (8 bits) for the code byte used for this compressed data. So a total of 72 bits are required to represent the compressed data.

t_b = Number of bits.
s_B = Total number of bytes.

Algorithm1: GenCodex Algorithm for Compression

Input: Original DNA sequence
(Base pairs: A, C, G, and T).

Output: Compressed Sequence
along with Code-bytes.

Phase1:

I. Divide the sequence into fragments consisting of 4 characters each.

II. Assign unique two bit number to each of the character (A=00, C=01, G=10, T=11).

Phase2:

III. If consecutive fragments are not same, then we represent the fragment with its 8-bit representation (1byte) itself.

IV. If the fragment is repeating two or more times consecutively, and then we represent that fragment using 2 bytes and set the corresponding bit in the Code byte.

V. Repeat the same process until the end of the sequence.

VI. Output the compressed sequence along with the codebytes that are used to represent the compressed sequence.

Table1: Compression Ratios for Different Algorithms (Bits/Bytes)

Algorithm	GENBIT	DNABIT	GenCodex
Best Case	1.125	1.04	0.017
Average Case	1.727	1.53	1.42
WorstCase	2.238	1.58	2.25

C_r=Compression Ratio.

$C_r = t_b / s_B = 72/4080 = 0.017$bits/bytes.

Average Case: The repetitions of each fragment range from 0 to 255. An analysis on DNA sequences done by us showed that a fragment repeating 3 or 4 times consecutively is more common than a fragment repeating 255 times. In fact, this analysis showed that maximum of 14 consecutive fragment repeats occur in a DNA sequence.

So, for the average case analysis, probabilities are assigned suitably for each pattern. We assigned high probability for 2, 3, 4 consecutive repeats and less probability for more number of repeats such as 255. The compression ratio is 1.42 bits/bytes.

Worst Case: Consider an input sequence of length 32 bytes where no fragment is repeating. A total of 72 bits are needed for representing the compressed data.

$C_r = t_b / s_B = 72/32 = 2.25$bits/bytes.

The compression ratios with different algorithms for the best, average and worst cases are shown in the Table 1.

3.2 Alignment of the Compressed Sequences

In this phase, we use the compressed output sequences from the compression phase. So, each fragment consists of 4 characters and occupies a byte since each character is allocated 2 bits (A=00, C=01, G=10, T=11).

The proposed algorithm consists of two types of alignment. Ungapped alignment and gapped alignment.

3.2.1 Ungapped alignment

In this phase, the sequences are aligned without considering insertions or deletions. Firstly, a match of length L is found in both the sequences i.e., all the compressed bytes are compared between both the sequences respectively. First byte in first sequence with first byte in second sequence and second byte in first sequence with second byte in second sequence and so on. After finding the bytes which are matching, we extend it to the next byte both in forward and backward directions of this particular match. This will make sure that we do not miss any particular characters which are matching if not the full byte. This is called partial alignment. This can be seen with an example in the Fig. 2.

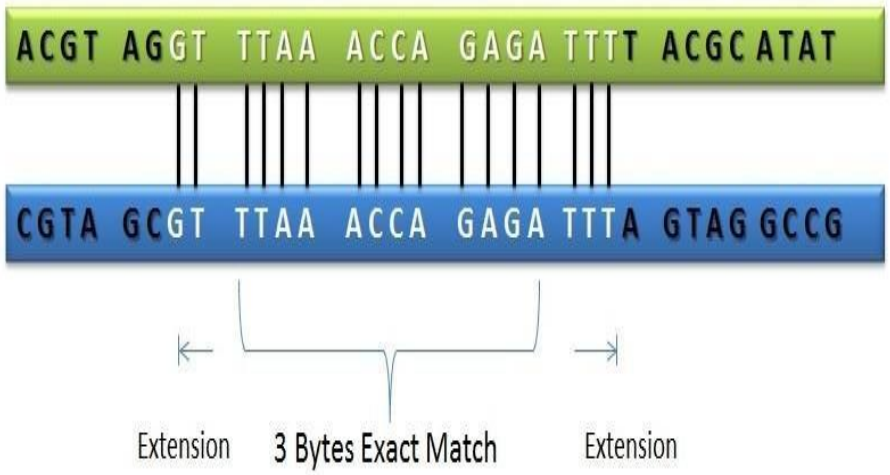

Fig. 2: Example to show the ungapped alignment

The score is given for every particular match as well as mismatch. Whenever a full byte is matching, we assign a match score of 4 since four characters are compressed in full byte. When a single character is matched in the extension process we assign a score of 1 for each match. Mismatch leads to a negative score and we assign -1 for that. This extension process continues until the score falls below a brink or threshold. The highest score is noted at every step and when the extension stops it is given as output. We maintain 3 functions namely Match, Leftmatch, Rightmatch for calculating the score. Match (A, B) records the total score between the pair of bytes A and B. For example, Match (AGCT, AGCG) = 3. Rightmatch (A, B) records the number of consecutive matching characters from first character in the byte to the last character before the first mismatch occurs. For example, Rightmatch(AGTC,AGCT) = 2. Similarly, Leftmatch(A, B) records the number of consecutive matching

characters from last character in the byte to the first character before the first mismatch occurs. For example, *Leftmatch* (T C AG, GC AG) = 3.

```
Ungapped alignment (l, m, x, y, brink)
begin
      score = 0;
      highscore = 0;
      xmax = 0;
     ymax = 0;
do while x ≤ l ∧ y ≤ m
       A = l[x : x + 3], B = m[y : y + 3];
       C = highscore − score;
   if (score > highscore − 2 ∗ match)
       partialscore = score + Rightmatch(A, B);
      if (partialscore > highscore)
        highscore = partialscore;
        xmax = x;
       ymax = y;
  else if (brink ≤ C )
    break;
   score = score + Match(A, B);
  x + +; y + +;
end
```

This algorithm aligns each byte at a time instead of single characters in the given sequence. If the alignment of the single characters achieves a score which is optimal, then the partial alignment in the successive byte is taken into account. We use the variable *partialscore* for this. We also consider l and m as the length of the first and second sequences respectively. The full method can be seen the algorithm given above. The algorithm given works for extending the alignment in forward direction since we are using Rightmatch(A, B). To extend this in backward direction, the algorithm needs to be modified by using *Leftmatch*(A, B).

3.2.2 Gapped alignment

In this phase, the alignment consists of insertions and deletions and hence it is more intricate than computing the ungapped alignment. It is complicated since there are multiple ways of aligning the sequences and gaps can be inserted anywhere. For example, the gap may be inserted or deleted in between the characters which are compressed into a single byte. However, in this work we consider gaps occurring only at either the start or end of the compressed byte. So, gaps can occur only at positions i = 0, 4, 8..... i.e., i = 0 modulo 4. This results in suboptimal alignment but it is a very good approximation to the actual optimal alignment between the two sequences.

The main difference between the actual sequence alignment and this gapped alignment is that each compressed byte takes one row (or column) in the matrix used for alignment instead of one row for a single character. The X-axis of the matrix denotes the series of overlapping compressed bytes of the first sequence and Y-axis represents the second compressed sequence. The score of each cell is

obtained from the maximum of three cells i.e., left, top, diagonal. This means one of the following events may have occurred - a match of full byte from both the sequences or an insertion in the second sequence or 4 insertions consecutively in the first sequence.

The algorithm maintains three values for every cell in the matrix. Best(x, y) maintains the best score for the alignment which ends at (x, y). High(x, y) notes the highest score for the alignment which ends at (x, y) but including an insertion in either of the sequences. Score(x, y) denotes the highest score for the alignment which ends at (x, y) with four matching characters. We also maintain some variables such as g for opening a gap, penalty p for extending the gap, d for the first insertion in the gap i.e., d = g + p.

We calculate the matrix recursively using the following relations.

$$Score(x, y) = Best(x - 4, y - 4) + Match(A, B)$$

Where A = [x − 3 : x], B = [y − 3 : y].

$$High(x, y) = \max \{ Score(x - 1, y) - d,\ High(x - 1, y) - p,\ Score(x, y - 4) - d - 2p,\ High(x, y - 4) - 4p \}$$

$$Best(x,y) = \max \{ Score(x,y),\ High(x,y) \}$$

Initially, the cells of the matrix are initialised with 0 where $x < 4$ and $y < 4$. The matrix is filled with the recurrence relations given.

4 Implementation

4.1 Compression Phase

Serial Implementation:

– In the first phase, each character is read from the file and is allocated two bits. By using the bit-wise shift operations, four characters are encoded into a single byte instead of four bytes.
– In the second phase, the fragments are allocated either a single byte or two bytes according to the number of repetitions in the input sequence.
– But, there is a special case wherein we always set the 8th bit in a code byte to 0.

– As described earlier, there is one code byte for every 8 bytes of compressed data. Setting the 8th bit in the code byte to 1 implies that there is a repetition in the 8th and 9th bytes of the sequence and we need to allocate two bytes (occupying 8th and 9th bytes) in the compressed sequence, but this 9th byte corresponds to second code byte which is already allocated.
– In this case, the 8th bit in the code byte is set to zero and the 8th byte in the compressed sequence represents just the 8-bit representation of the fragment.
– The same process repeats from the 9th byte onwards.

Since the input sequences are huge in size and the chunks of fragments are independent of each other, there is a scope for parallelization.

Parallel implementation:

The proposed algorithm has been implemented on multi-cores as well as on GPUs.

4.1.1 Multi-Core:

The algorithm is implemented using OpenMP on multi-core. The input sequence is distributed among the cores available and each core finds the repeated fragments and compresses the data allocating the bytes accordingly.

The algorithm is run on Lonestar where each node has 12 core M2070 card. The input sequence is divided among these 12 cores equally. Each core maintains private variables so that the cores do not have any race condition such as two cores updating the same variable simultaneously. All the cores parallely compress their own share of data so that the time taken for total sequence is much less compared to the sequential algorithm. At the end, the compressed sequence is stored in a buffer. This buffer is a private variable where each thread waits for the other to write into it. Finally, this is written into an output file.

4.1.2 GPUs:

The algorithm is implemented on GPUs using CUDA. The steps involved are as follows:

1. The resultant output array from the phase 1 is copied into the global memory of GPU from the CPU host memory.
2. The kernel is launched with the number of threads and the blocks varying according to the size of the given input sequence. For a small input sequence, we use threads starting from 50 and as the size increases the no. of threads launched increases up to 5000.
3. Each thread finds the repetitions and stores the result in a buffer in the global memory.
4. After all the threads finish their job; this buffer is copied from global memory to host memory.
5. From this, the compressed data is finally written to an output file which is done sequentially.

We have run the parallel version with varying number of threads and blocks to achieve the best performance for given input sequence. It is found that the work is more if the number of threads launched is around 50 to 1000 and the threads launched are more for the given input sequence if they are more than 5000. The performance remains same as we increase the number of threads beyond this limit. The optimal result which we achieved was by having 500 blocks with 10 threads each. So we mainly concentrate on these sizes alone and discuss our results. In results section, we briefly discuss about other sizes.

The Parallel implementation of the algorithm is done on NVIDIA Tesla M2070 GPU. To further enhance the speedup of the parallel algorithm, we have also used the recently released NVIDIA Kepler K20 GPU. Kepler K20 comes with enhanced performance improvement in both single and double precision operations. This also showed a good improvement.

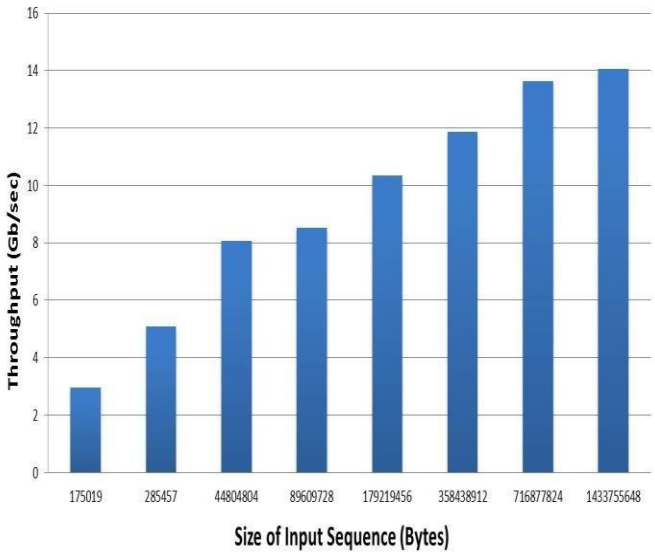

Fig. 3: Compression throughput in Gb/Sec for different datasets (On Lonestar M2070 card)

The throughput is calculated here in terms of the time taken to compress the whole data. The compression throughput achieved when the algorithm is run on Lonestar with M2070 Tesla card for different data sets can be observed in the Figure 3. The maximum throughput achieved is 14 GBPs. The data sets have been chosen with varying sizes. The throughput shown reflects only the computation time. It does not include the time taken for transferring the data between the host and the device. The compression throughput τ is the rate at which the data is compressed.

$$\tau = \text{size of the input data} / \text{time taken (in sec)}$$

The throughput increases as the size of the input data increases. This is due to the fact that the number of threads launched is directly proportional to the data size. As the number of the threads increases, the greater is the utilization of the GPU. GPU power is extracted fully with more than thousands of threads running in parallel.

When the data size is small, a few threads are created to compress the data which results in sub-optimal throughputs.

Similarly, the throughput achieved when the same data sets where run on Stampede with k20 GPUs can be seen in the Figure 4. The maximum throughput achieved here too is 14 GBps.

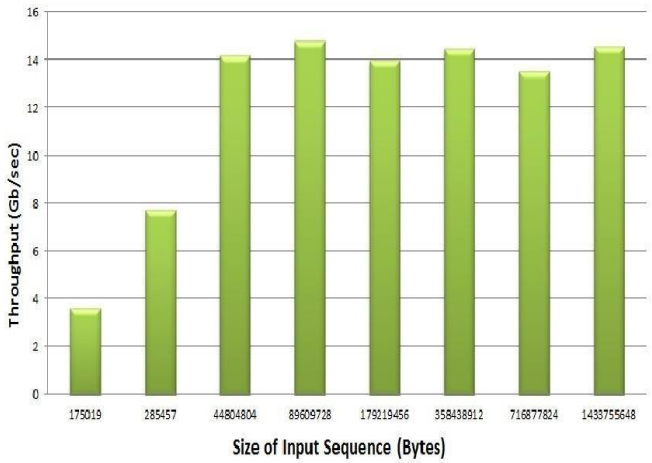

Fig. 4: Compression throughput in Gb/Sec for different data sets (On Stampede K20 GPU)

4.2 Alignment Phase

Serial Implementation:

The ungapped alignment was done in 3 steps. Firstly, the input sequences are read from the files in a loop. This is done recursively till the end of the file since the sequences are huge in size. Each character is read and is allocated two bits. By using the bit-wise shift operations, four characters are encoded into a single byte. In the next step, we find the Match, Leftmatch, Rightmatch for each pair of bytes. This is done in a loop going through all the bits in those particular compressed bytes. Lastly, we call the function Ungapped alignment as shown in the algorithm which computes the alignment score for the given sequences. The loop breaks when the score falls below a brink or threshold.

In the gapped alignment, the dynamic programming matrix is calculated using three arrays each for Best, High, and Match which are allocated using malloc. These recurrence relations are executed in a loop. This in turn calls the Match function which calculates the matching score between a pair of bytes. The cost of opening a gap g is assigned -2 and penalty p for extending a gap is assigned -1. The entire serial implementation is using C.

Parallel implementation:

The proposed algorithm has been implemented on multi-cores as well as on GPUs.

4.2.1 Multi-Core:

The gapped alignment is implemented using OpenMP on multi-core. Each core finds the corresponding cells in the dynamic programming matrix assigned to it. There is a synchronization barrier at the end of the loop since each core must have to wait for the other to get the required values. This also assures that the computed values are not stale.

4.2.2 GPUs:

The ungapped alignment is implemented on GPUs using CUDA. The steps involved are as follows:

1. The resultant output array from the compression phase is copied into the global memory of GPU from the CPU host memory.
2. The kernel is launched with the number of threads being equal to the size of the input compressed sequence.
3. Each thread finds the Match, Leftmatch and rightmatch for its corresponding bytes and stores the result in a buffer in the global memory.
4. After all the threads finish their job, this buffer is copied from global memory to host memory.
5. From this, alignment score is updated accordingly by adding all the scores obtained by different threads. This is done sequentially.

5 Experimental Setup

The serial code was run on Intel(R) Pentium(R) Dual core 2.20 GHz processor with 4GB RAM on Ubuntu 10.04 LTS. The parallel code was run on Lonestar supercomputer (TACC) which has over 22,000 cores with QDR InfiniBand networking (40Mb/s, sub-10us latency). Each core runs at 3.3 GHz (Intel Xeon, 12 MB L3 cache) and has 24 GB, 1333 MHz RAM per 12-core node. NVIDIA Tesla M2070 card with 448 cores and 6 GB global/device memory was used for GPU runs.

The parallel code was also run on Stampede supercomputer (TACC) which has over 1,00,000 cores with QDR InfiniBand networking (56Gb/s). Each core runs at 2.7 GHz (Intel Xeon, 20 MB L3 cache) and has 24 GB, 2701 MHz RAM per 16-core node. It has around 6400 compute nodes configured with two Xeon E5-2680 processors and one Intel Xeon Phi SE10P Coprocessor (on a PCIe card). These compute nodes are configured with 32GB of ”host” memory with an additional 8GB of memory on the Xeon Phi coprocessor card. It has also 128 compute nodes for visualization and GPGPU processing each with a single NVIDIA KEPLER K20 GPU with 8GB of on-board GDDR5 memory. NVIDIA K20 GPU with 8 GB global/device memory was used for running the GPU code.

Some of the codes were run on a local machine with Fermi architecture having Tesla M2050 card. It has 4 devices present on it. This has 2 sockets and each socket has 6 cores present on it. Each core in turn has 2 threads. Overall, it can run 12 threads in parallel. Each core runs at 2.67 GHz (Intel(R) Xeon(R) 12 MB L3 cache).

5.1 Kepler architecture

NVIDIA Tesla K-series GPU Accelerators are based on the NVIDIA Kepler compute architecture and powered by CUDA, the world's most pervasive parallel computing model. They include innovative technologies like Dynamic Parallelism and Hyper-Q to boost performance as well as power efficiency and deliver record application speeds for biochemistry simulations, weather and climate modelling, image, video and signal processing etc.

The innovative Kepler compute architecture design includes:

- **SMX:** This design delivers up to 3x more performance per watt compared to the SM in Fermi [14]. It also delivers one petaflop of computing in just ten server tracks.
- **DynamicParallelism:** This capability enables GPU threads to automatically spawn new threads [15]. By adapting to the data without

Table2: Size of the Compressed Sequences for Different Algorithms (In Bytes)

DNA Sequence	Input size (in Bytes)	Gen Com-press	DNA Com-press	Genbit	GenCodex
HSCOMT2	1700	436	416	392	377
HUMCYC1A	2206	560	540	516	496
HSU37106	2256	573	561	546	528
HSGTRH	3938	995	967	918	889
HUMGALK1A	7086	1703	1708	1691	1629
HSU01102	4280	1035	1052	986	950
HSC1INHIB	16309	3789	3960	3575	3465
HSCST4	3489	869	842	832	807
HUMA1ATP	4786	1200	1171	1110	1065
HSTNT2	8657	2052	2049	2038	1973
HUMRBPA	8682	2143	2116	2070	2000
HUMHSKPQZ	2334	619	591	564	544
HUMRETBLAS	175019	40183	41688	39059	37770
HUMTBGA	6275	1594	1541	1486	1441
HSAT3	13347	3189	3250	2987	2892
D87675	285457	66649	68519	64537	62384

going back to the CPU, it greatly simplifies parallel programming. Plus it enables GPU acceleration of a broader set of popular algorithms, like adaptive mesh refinement (AMR), fast multipole method (FMM), and multigrid methods.

– **Hyper-Q:** This feature enables multiple CPU cores to simultaneously utilize the CUDA cores on a single kepler GPU [15]. This dramatically increases GPU utilization, slashes CPU idle times, and advances programmability - ideal for clsuter applications that use MPI.

5.2 Kepler K20 GP Accelerators

These GPUs are designed to be the performance leader in double precision applications and the broader supercomputing market, the Tesla K20 and K20X GPU Accelerators deliver 10x performance of a single CPU. Tesla K20 and K20x both feature a single GK110 Kepler GPU that includes the Dynamic Parallelism and Hyper-Q features. With more than one teraflop peak double precision performance, these GPU accelerators are ideal for the most aggressive high-performance computing workloads that include climate and weather modelling, CFD, CAE, biochemistry simulations and computational finance [14] [15].

Table3: Compression Ratios for Different Data-sets (In Bits/Bytes)

DNA Sequence	Input Size (KB)	Gen Com-press	DNA Com-press	Genbit	GenCodex
HSCOMT2	1.7	2.05	1.95	1.84	1.77
HUMCYC1A	2.2	2.03	1.95	1.87	1.79
HSU37106	2.25	2.03	1.98	1.93	1.87
HSGTRH	3.93	2.02	1.96	1.86	1.8
HUMGALK1A	7	1.92	1.92	1.90	1.83
HSU01102	4.2	1.93	1.96	1.84	1.77
HSC1INHIB	16.309	1.85	1.94	1.75	1.69
HSCST4	3.4	2	1.93	1.9	1.85
HUMA1ATP	4.786	2	1.95	1.85	1.78
HSTNT2	8.6	1.9	1.89	1.88	1.82
HUMRBPA	8.682	1.97	1.94	1.90	1.84
HUMHKPQZ	2.3	2.12	2.02	1.93	1.86
HUMRETBLAS	175.019	1.83	1.90	1.78	1.7
HUMTBGA	6.275	2.03	1.96	1.89	1.83
HSAT3	13.34	1.91	1.94	1.79	1.73
D87675	285.457	1.86	1.92	1.81	1.74

5.3 M2070 vs K20

K20 is much powerful than M2070 in terms of power consumption and processing power. As explained earlier, K20 introduces next generation multi-processors. K20 accelerators deliver highest single and double precision performance 3.52 Tflops and 1.17 Tflops [15] respectively whereas M2070 delivers 1.03 Tflops for single precision operations and 515 Gflops for double precision operations [16]. The Memory bandwidth of Tesla K20 is 208 GB/s whereas the bandwidth for M2070 is 150 GB/s [15] [16]. The M2070 GPU has 448 CUDA cores whereas K20 has 2496 CUDA cores [15] [16].

This is the main reason for gaining further improvement in the speedup on the K20 GPUs. The memory bandwidth plays a significant role in transferring the huge

sequences onto the GPU. Also, as explained earlier, the card GK110 used on K20 along with SMX leads to a greater performance compared to M2070 card used on Fermi.

The data-sets used for the experiments are taken from GenBank (ftp://ftp.ncbi.nlm.nih.gov/genbank). This is a publicly available repository for DNA sequences. These sequences used in the experiments size up to 1.43 GB.

SIMD (Single Instruction Multiple Data) was used while parallelizing the code. The optimization flag O3 was used while compiling the code. The algorithm was implemented on multi-cores using 12 threads on Lonestar supercomputer.

6 Results

6.1 GenCodex

The serial and parallel implementations of the algorithm were evaluated on data-sets of different sizes. We noticed that our algorithm performs better if the consecutive repetitions are more (up to 255 repetitions). Table 2 shows the compressed size in bytes for all the algorithms using different data sets.

The compression ratio remains same for both the serial and parallel versions of our algorithm. We observed that the compression ratio of our algorithm is good when there are more repetitions. Table 3 shows the compression ratios in terms of bits/byte for different algorithms.

The parallel implementation outperformed the serial implementation in terms of the throughput (time taken to compute the data) for all the data-sets. The parallel version on Lonestar achieved a speedup of up to 11 on a 12-core M2070 card and up to 23 on M2070 GPUs for the data-sets used in our experiment. The results are shown in the Fig. 5.

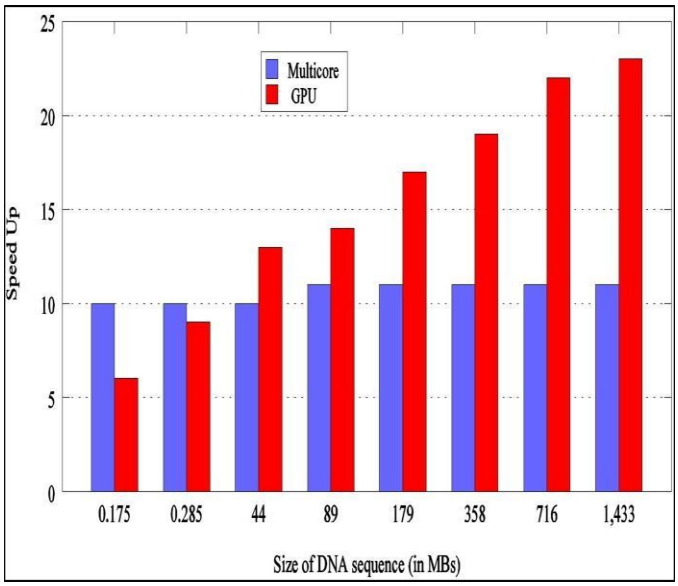

Fig. 5: Speedup on multi-cores and GPUs (On LonestarM2070 card)

Size of the Data	Sequential	Multi-core	GPU (M2070)
175019	0.276	0.028	0.059
285457	0.456	0.047	0.056
44804864	8.682	1.97	1.94
89609728	141.401	13.7	10.573
179219456	283.237	27.44	17.355
358438912	567.532	57.225	30.268
716877824	1130.988	110.88	52.633
1433755648	2272.355	219.911	102.601

Table 4: Timings (In Milli-Seconds) on Multi-Core and GPUs on Lonestar

We observe that as the data size increases, GPUs perform better compared to multi-cores. This can be observed from the Table 4. This scalability is achieved because the work-load on the threads increases as the data-size increases on the multi-core.

The parallel version when run on Stampede supercomputer achieved a speedup of up to 57 on K20 GPUs. The parallel version was run using 500 blocks with 10 threads each. This is the optimal number of blocks for our algorithm as the performance decreases for other sizes as we tested it with varying number of blocks from 50 to 5000. We observed a speedup for other sizes as well but are not significant. The experiments show that the algorithm scales well on GPUs and works better even for the huge sequences.

The time taken for different standard data-sets when run on Stampede Supercomputer (NVIDIA Kepler K20 GPUs) can be observed from the Table. 5.

We can also observe the difference in the speedup level when the parallel code was run on Lonestar (M2070 card) and on Stampede (K20 GPUs). There is a significant amount of speedup on K20 GPU which leads to a very high throughput compression. The results for both can be seen in the Fig. 6.

6.2 Alignment

Table 5: Timings (In Milli-Seconds) and Speedup on GPUs on Stampede

Size of the Data	Sequential	GPU (K20)	Speedup
175019	0.683	0.049	13
285457	1.127	0.037	30
44804864	173.81	3.161	54
89609728	347.236	6.058	57
179219456	694.877	12.816	54
358438912	1388.381	24.799	55
716877824	2778.765	53.006	52

1433755648	5556.102	98.56	56

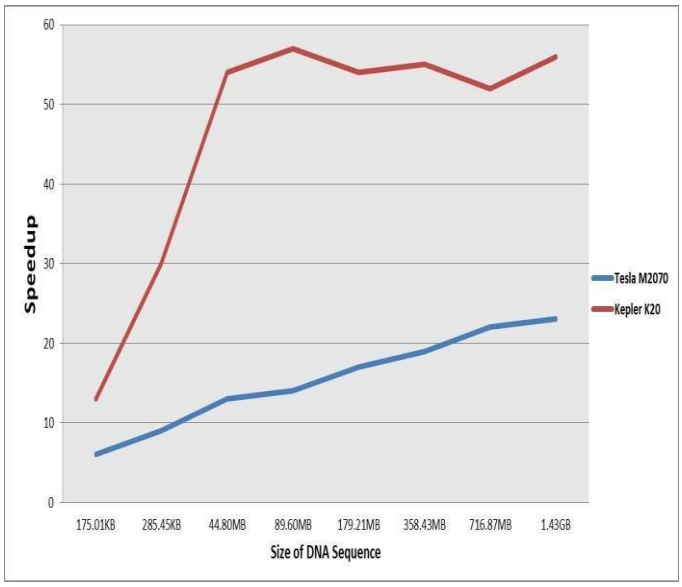

Fig. 6: Speedup on Lonestar M2070 and Stampede K20

The serial and parallel implementations of the alignment methods were evaluated on data-sets of different sizes. We present the results for a few of them. To test the efficiency and the accuracy of our algorithm we tested the scores of different sequences with the traditional Smith-Waterman algorithm. We consider the alignment to be optimal if the scores returned by this algorithm are at least more than three-fourth of the optimal alignment score returned by Smith-Waterman. This is similar to the method used in [17].

Table 6: Timings for different sequences on 16-core Intel Xeon E5-2680 for ungapped alignment (Milli Secs)

Seq1 size (Bytes)	Seq2 size (Bytes)	Serial (Msecs)	Multicore(No. ofthreads)			
			2	4	8	16
8658	8683	0.123	0.213	0.203	0.251	0.367
8683	175019	0.127	0.178	0.216	0.255	0.37
8683	285457	0.124	0.169	0.176	0.225	0.379
175019	285457	2.546	2.575	2.153	1.955	1.957
285457	44804864	4.154	4.161	4.011	3.14	3.036
175019	44804864	0.515	0.985	1.106	1.124	1.237

The parallel implementation for ungapped alignment is run on 16-core Intel Xeon E5-2680 processors on Stampede supercomputer. The code was written in OpenMP. The time taken for aligning different sequences when run using varying number of threads can be observed from the Table 6. We can observe

that the performance decreased when the code was run on multi-cores. When the code was analysed with different options such as openmp-report and par-report, the results showed that some of the loops inside the code cannot be parallelized. This is because of the ANTI (Read after Write), FLOW (Write after Read) and OUTPUT (Write after Write) dependencies involved in the code. A code snippet is shown where the OUTPUT dependency is occurring.

```
for(j=0;j<4;j++)
 {
  switch(j)
   {
    case0:array[j]=var>>6;
                .
                .
                .
    case1: array[j]=extra[j]>>6;
                .
                .
                .
   }
  }
```

In this case, both the case 1 and case 2 are involved in writing to the array sairam[j]. This is OUTPUT dependent. So even though we put *pragma omp parallel* for to parallelize the loop, it's not parallelized because of this OUTPUT dependency involved. One thread may write before the other thread writes into it. This will lead to storing of wrong values. It will take more time because of this overhead.

The time taken for different pair of sequences for ungapped alignment using GPUs can be seen in the Table 7. We can clearly notice the improvement in the performance in terms of time taken when the code was run on Kepler K20 GPUs on

Table7: Timings for different sequences on Stampede (K20) for ungapped alignment (Milli secs)

Size of Sequence 1 (bytes)	Size of Sequence 2 (bytes)	serial	GPU (K20)
8658	8683	0.124	0.007
8683	175019	0.13	0.0072
8683	285457	0.138	0.0074
8683	355872	0.148	0.0076
175019	285457	2.585	0.073
175019	44804864	2.781	0.073
285457	44804864	4.204	0.074

Stampede Supercomputer. The compressed sequences to be aligned are transferred to the GPU and the number of threads launched is equal to the number of bytes in the compressed sequences. Each thread will find its corresponding bytes *Rightmatch* and *Match* and gives the score. Since this is data parallel, the time taken is much low even though the dependencies are there in the code.

The difference between the serial and GPU timings can be observed from the Fig. 7. X-axis represents the average of the sizes of both sequences involved in alignment. Y-axis represents the time taken for alignment. Clearly, there is a good amount of speedup achieved when compared to the serial implementation.

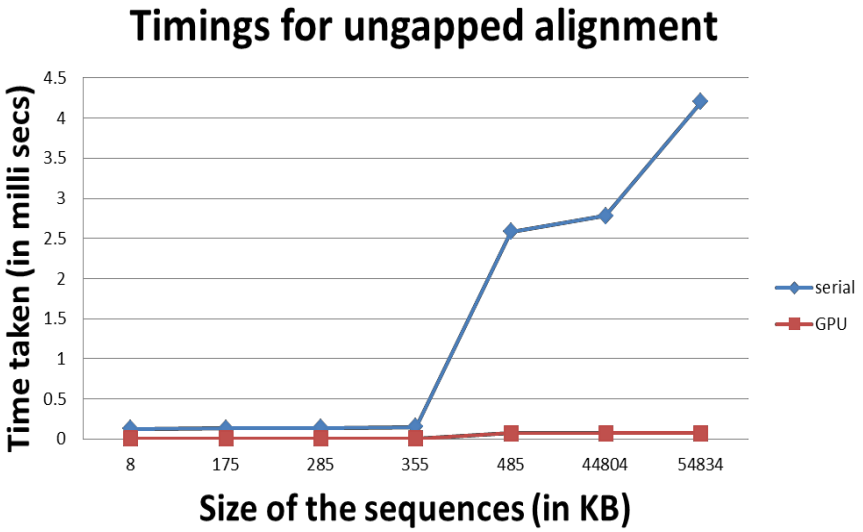

Fig. 7: Comparison between Serial and GPU (K20) timings for ungapped alignment

Table8: Timings for different sequences on 16-core Intel Xeon E5-26 for Gapped Alignment (Secs)

Seq1 size	Seq2 size	Serial	Multicore(No. of threads)			
(Bytes)	(Bytes)	(secs)	2	4	8	16
3490	4787	0.3111	0.1923	0.0812	0.0404	0.0204
8683	2335	0.321	0.1737	0.0834	0.042	0.0216
4787	8658	0.7668	0.4324	0.21	0.0004	0.0501
8658	8683	1.39	0.7218	0.3521	0.1815	0.0904
2335	175019	6.497	3.5321	1.7243	0.9269	0.4921
8683	175019	25.082	15.396	7.21	3.658	1.83

Similarly, The time taken for different sequences for gapped alignment can be seen in the Table 8. The parallel code was run on 16-core Intel Xeon E5-2680 present on Stampede. The code was run with the varying number of threads from 2 to 16 and the timings were taken. There is a significant improvement in the performance as the number of threads increases. We observe that when code was run with 16 threads the performance was bet- ter. The performance did not improve further as we increased the number of threads to 32 and 64. One reason for this is the number of cores being limited to 16. There are many statements in the code such as switch-case which are

highly dependent. So as the threads increases, the overhead of thread scheduling is more and it takes more time which does not result in further improvement.

The comparison between the serial and multi-core (Intel Xeon E5- 2680) timings for the gapped alignment is shown in the Fig. 8. As the alignment is between two sequences with different sizes, we took the average of the sizes and represented on X-axis. Y-axis represents the time taken for aligning the sequences in seconds. We can observe that time taken for serial implementation is more than the time taken for parallel implementation.

Also, the time taken for different sequences for gapped alignment when run on 12-core Intel Xeon X5650 processor can be seen in the Table 9. The parallel code was run on multi-cores present on this machine. The code was run with the varying number of threads from 2 to 16 and the timings were taken. There is a significant improvement in the performance as the number of threads increases. We observe that when code was run with 16 threads the performance was better. The time taken remained same even though we increased the number of threads to 32. As the threads increase, each thread has to wait for the other since there are many dependencies in the code. This is the reason for not gaining the performance after 16 threads. Also, there is not much increase in the performance if we increase the threads from 8 to 16. Since there are only 12 cores it takes some time to load the next 4 threads (when total threads assigned are 16) and finish their work. This lessens the total increase in the performance.

The difference in timings between serial and multi-core with different threads can be observed in the Fig. 9. X-axis represents the average of the two sequences participating in the alignment. Y-axis denotes the time taken for gapped alignment.

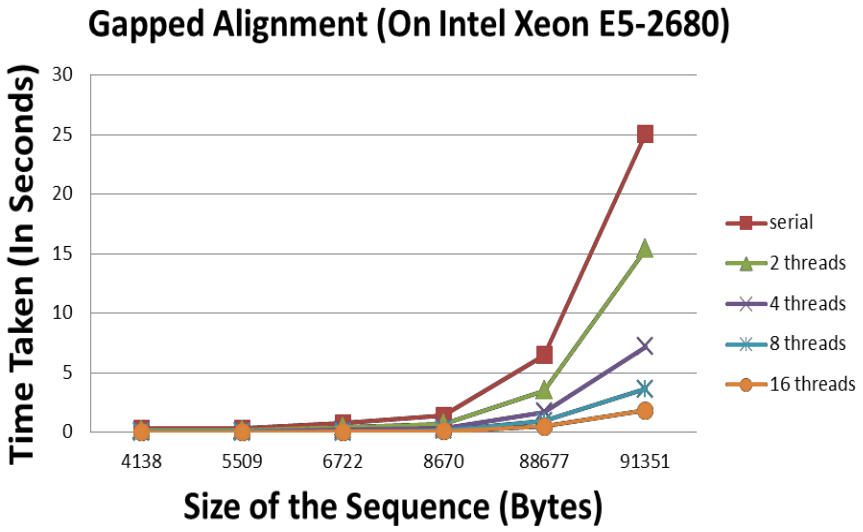

Fig. 8: The serial and Multi-core timings for Gapped Alignment (On 16-core Intel Xeon E5-2680)

Table9: Timings for Different Sequences on 12-core Intel Xeon X5650 for Gapped Alignment (Secs)

Seq1 size	Seq2 size	Serial	Multicore (No.of threads)			
(Bytes)	(Bytes)	(secs)	2	4	8	16
3490	4787	0.3761	0.1948	0.0974	0.0519	0.0470
8683	2335	0.4371	0.2983	0.1036	0.0521	0.0661

4787	8658	0.8054	0.4804	0.2405	0.1265	0.1163
8658	8683	1.62	0.8748	0.4383	0.2297	0.1887
2335	175019	8.2096	4.5081	2.2789	1.2148	1.0022
8683	175019	31.5967	17.4902	9.4118	4.6263	3.7983

There is a very good amount of improvement as the threads increase up to 16.

From the experiments, we can observe a very good speedup for the parallel version when compared to the serial version. For the ungapped alignment we observed a speedup of up to 56 when run on K20 GPUs. This can be seen in the Fig 10. Gapped alignment achieved a speedup of up to 15 when run on 16-core Intel Xeon E5-2680 processors on Stampede. This can be seen in the Fig 11.

The alignment score remains same for both the serial and parallel versions of our algorithm.

There is a significant improvement in the performance when the gapped alignment was implemented on 16-core Intel Xeon E5-2680 and also on Intel Xeon X5650 processors. The highest speedup was achieved when the

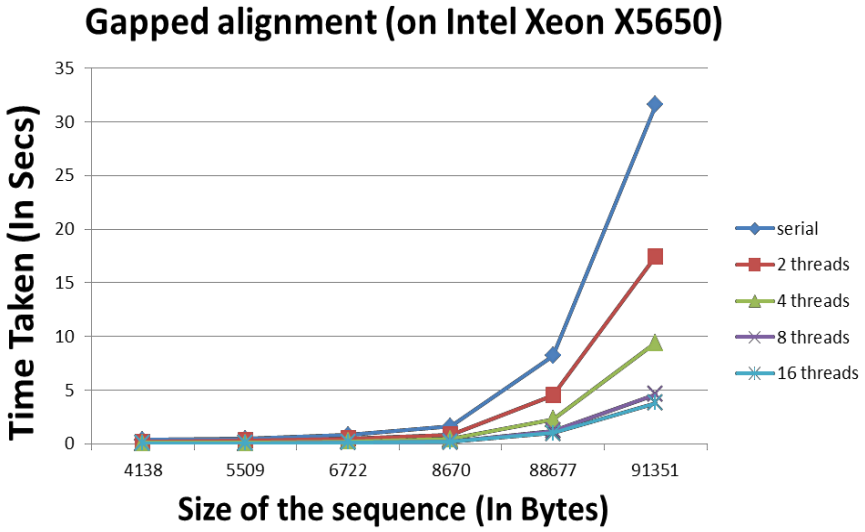

Fig. 9: The serial and multi-core timings for the Ungapped alignment (On 12-core Intel Xeon X5650)

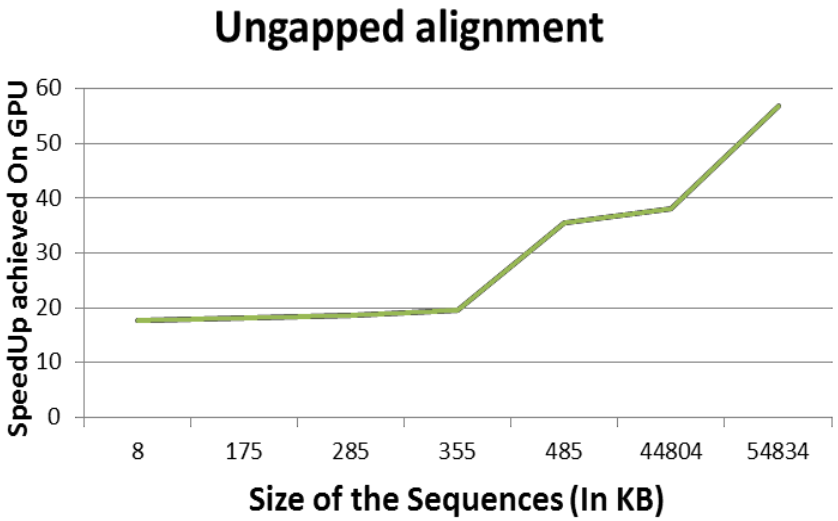

Fig. 10: Speedup achieved on Kepler K20 GPU for the Ungapped alignment

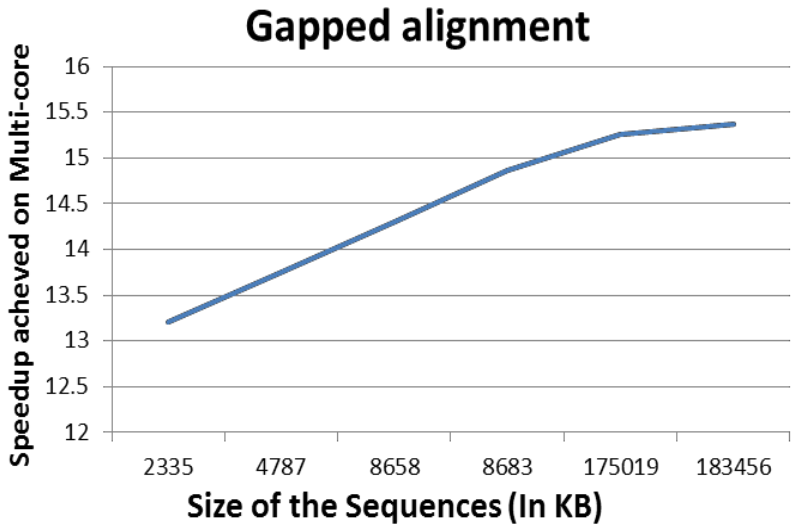

Fig. 11: Speedup achieved on16-core Intel Xeon E5-2680 for the Gapped alignment

number of threads are 16 on both of them. The speedup achieved on E5- 2680 processors is more than what is achieved on Xeon X5650 for all the sequences. This is because the formers design itself delivers 3x more performance compared to the latter. The difference in speedup can be observed from the Fig. 12.

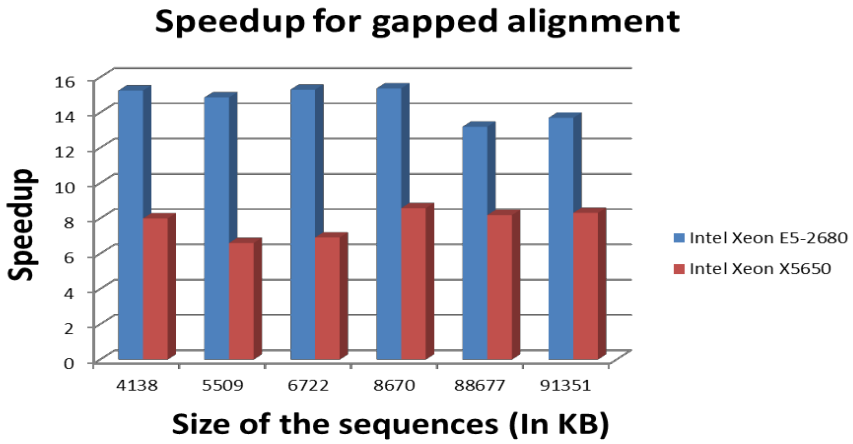

Fig.12: Comparison between the speedups achieved for Gapped alignment

7 Conclusions and Future work

A new compression algorithm is proposed to compress the DNA sequences. The main focus was on the throughput along with the compression ratio. As the number of consecutive repeats increases, the algorithm achieves the best compression. If the

fragments are repeating only twice or there are no repetitions then the algorithm may not perform better.

The compression ratio remains same for both the serial and parallel versions. We noticed a very good improvement in the throughput when the algorithm was implemented on multi-cores and GPUs. We observed a speedup of 11 on multi-cores and 23 on GPUs when run on Lonestar which has NVIDIA M2070 card and there is a significant improvement when the parallel version is run on Stampede which has Kepler K20 GPUs with a speedup of up to 57 being achieved. Experiments showed us a good scalability on GPUs for the standard data-sets. The results show that our method achieves a good compression ratio along with better throughput compared to other existing methods.

An innovative algorithm is proposed to align the sequences on the compressed sequences. The alignment was done in 2 phases. Ungapped alignment and gapped alignment. There are advantages for this approach. Sequences can be aligned without decompressing the sequences. Also, this alignment is faster compared to the traditional methods since a full com- pressed byte consisting of 4 characters is aligned at a time instead of a single character. The performance was improved further by porting this algorithm onto multi-cores as well as GPUs. The gapped alignment achieved a speedup of up to 15 on multi-cores and ungapped alignment achieved a speedup of up to 56 on GPUs.

We are modifying the proposed algorithm in such a way that it can utilize the property of dynamic parallelism of Kepler K20 GPUs. This results in achieving high throughput as GPU threads themselves spawn new threads without communicating with the CPU. Also, we are working further such that different CPU cores simultaneously utilize the CUDA cores on a single Kepler GPU which increases GPU utilization and cuts down the CPU idle times. This really makes our algorithm robust achieving a very high throughput.

We are also extending the proposed algorithm to RNA sequences for compression. It also helps to calculate phylogeny. Also, the GPU implementation is used to solve the multiple-sequence alignment problem and the work is in progress in this direction. This algorithm also helps in reducing the time in searching databases especially when the sequences are really long. A half-byte can be used instead of a full code-byte in order to save the space consumed by the extra code-byte when there are no repetitions.

The proposed gapped and ungapped alignment algorithms can be extended to RNA sequences. This alignment method on compressed sequences can be used to find the structure of unknown sequences. This makes the process much faster. In future, the gapped alignment method can be modified such that it is possible to introduce the gaps in between the compressed byte.

References

1. Bell, T.C., Cleary, G. G. and Witten, I.H., “Text Compression”, Prentice Hall, Englewood Cliffs NJ, 1990.

2. Grumbach, S. and Tahi F. (1993) Compression of DNA sequences. In Data Compression Conference, IEEE Computer Society Press, Snowbird, Utah, USA, p 340-350.
3. Ming Li Xin Chen, Sam Kwong. A Compression algorithm for DNA sequences. In Proceedings of the Fourth Annual International Conference on Computational Molecular Biology, Tokyo, Japan, April 8-11, 2000.
4. Bin Ma Xin Chen 1, Ming Li and John Tromp. DNACompress: Fast and effective DNA sequence compression. (18), 2002, 1696–1698.
5. P Raja Rajeswari & Dr Allam AppaRao. DNABit compress genome compression algorithm. Bio information, (5), 2011, 350–360.
6. P Raja Rajeswari & Dr Allam AppaRao. Genbit compress - algorithm for repetitive and non-repetitive DNA sequences. International Journal of Computer Science and Information Technology, (2), (2010), 25–29.
7. Sadakane K. Matsumoto, T. and H Imai. Biological sequence compression algorithms. In Genome Informatics Workshop, Universal Academy Press, 2002, 43-52.
8. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory, (23), 1977, 337–343.
9. Ziv and Lempel. Compression of individual sequences via variable-rate coding, (24), IEEE Transactions on Information Theory, 1978, 530–536.
10. Tromp J. Ma, B. and M. Li. Pattern hunter: Faster and more sensitive homology search. Bioinformatics, (18), 2002, 440–445.
11. Siriwardena, T. and Ranasinghe, D. Accelerating global sequence alignment using CUDA compatible multi-core GPU. In Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference, 2010 , 201 –206.
12. Ashraf S Hussein Mohamed F Tolba, Ahmad M Hosny, Howida A Shedeed. An efficient solution for aligning huge DNA sequences. International Journal of Computer Applications, 32(6) (2011), 0975-8887.
13. Junjie Li, Sanjay Ranka, and Sartaj Sahni. Pairwise sequence alignment for very long sequences on gpus. Computational Advances in Bio and Medical Sciences, IEEE International Conference on, 0:1– 6, 2012.
14. NVIDIA, Tesla K-20-K20X GPU Accelerators – Benchmarks, November 2012, Available from: http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf
15. NVIDIA, TESLA KEPLER GPU ACCELERATORS, October 2012, Available from : http://www.nvidia.in/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf
16. NVIDIA, TESLA M-CLASS GPU COMPUTING MODULES, August 2011, Available from : http://www.nvidia.com/docs/IO/105880/DS_Tesla-M2090_LR.pdf
17. Chuck Staben Weixi Li, Cathryn J. Rehmeyer and Mark L Farmen. Terminus - telomeric end-read mining in unassembled sequences. Bioinformatics, pp 1, 2004.