Journal of Informatics and Mathematical Sciences
Vol. 7, No. 3, pp. 131–147, 2015
ISSN 0975-5748 (online); 0974-875X (print)
Published by RGN Publications

http://www.rgnpublications.com

Performance of DNA Sequences Compression on Multicores and GPUs

Research Article

Ajith Padyana[1,*] and Pallav Kumar Baruah

[1] *Department of Mathematics and Computer Science, Sathya Sai Institute of Higher Learning, Muddenahalli Campus, Chikkaballapura, Bangalore India*
[2] *Department of Mathematics and Computer Science, Sathya Sai Institute of Higher Learning, Prashanthi Nilayam Campus, Puttaparthy, India*
***Corresponding author:** ajithpadyana@sssihl.edu.in

Abstract. The DNA sequences are huge in size and the databases are growing at an exponential rate. For example, the human genome in raw format ranges from 2 to 30 Tera-bytes. The main reason for this is the invention of new species and increasing number of DNA profiles. The growth of the DNA affects the storage as well as bandwidth when these sequences need to be transferred. Applications such as DNA profiling, Real time DNA crime investigation require access to the DNA sequences in real time. The inherent property of DNA is that it contains many repeats which makes it highly compressible. However, the applications mentioned not only require good compression ratio but also needs faster compression. Multicores and GPUs can be used to perform the compression quickly. In this paper, we propose a new algorithm with a focus on the throughput along with the compression ratio. The algorithm scales well on GPUs and achieves a speedup of 11 on multi-cores and upto 23 on GPUs when run on M2070 Tesla card and upto 57 on K20 Kepler GPUs. We also extended this algorithm such that it adapts to the input sequence depending on the number of consecutive repeats and accordingly chooses the right algorithm which leads to a better compression.

Keywords. DNA Sequence; Bandwidth; Throughput; Compression Ratio; Speedup; Code byte

MSC. 92D20

Received: September 1, 2015 **Accepted:** November 28, 2015

1. Introduction

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.

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 [1]. As a result, these algorithms expanded the DNA sequence instead of compressing it [2]. Other algorithms such as Biocompress-2 [3], GenCompress [4], DNACompress [5], DNABIT [6] and GENBIT [7] 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 [8]. 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 throughput along with a better compression ratio. In recent days, the evolution in processor 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. In this paper, we address these issues by obtaining a better compression ratio at a high throughput 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. The adaptive version of the algorithm is presented in Section 4. Section 5 gives the implementation details of our algorithm. Section 6 describes the experimental setup. Section 7 discusses the results. Conclusion and future work are dealt in Section 8.

2. Related Work

Grumbach and Tahi [3] proposed two lossless compression algorithms for DNA sequences, namely BioCompress and BioCompress-2, making use of the Ziv and Lempel data compression method [9]. BioCompress 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 [9], the occurrences of the factor to encode are searched in a window. The second one [10] 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-2 is the use of arithmetic coding.

Gencompress [4] is a one-pass algorithm that searches for the approximate matches. This algorithm uses order-2 arithmetic encoding [2]. 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 be encoded 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 $u = \epsilon$ where ϵ 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 [4] achieves higher compression ratios compared to Biocompress or Biocompress-2.

DNACompress [5] 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* [11] 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's no overlap with the extracted repeat r. If the highest score of repeats in A is still higher than a pre-defined threshold then goto 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 [7] 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 9^{th} bit is set to 1. If the consecutive fragments are different, then the specific 9^{th} 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 number 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 9^{th} bit equals 1, then the corresponding combination is taken two times. Otherwise, it is just considered once. This algorithm just searches and encodes exact repeats instead of searching and encoding approximate repeats. This also works better for upto 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 decent compression ratio but also has a better compress throughput.

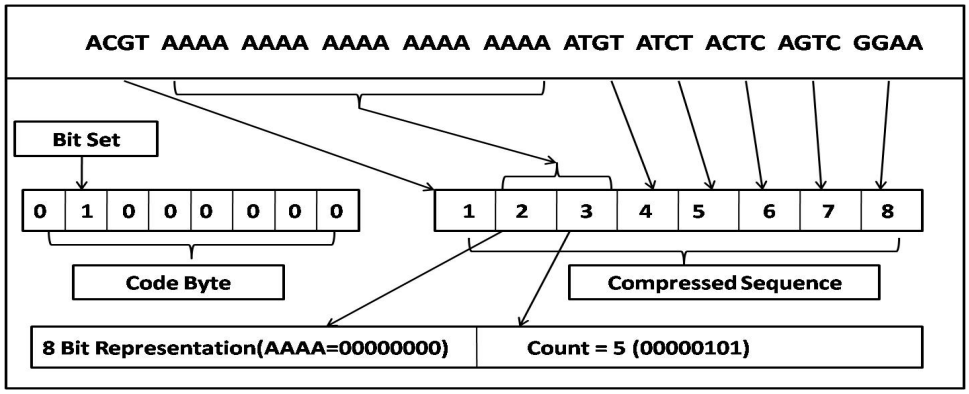

Figure 1. Histogram showing the no. of occurrences of each consecutive repeat

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.

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 are 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.

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.

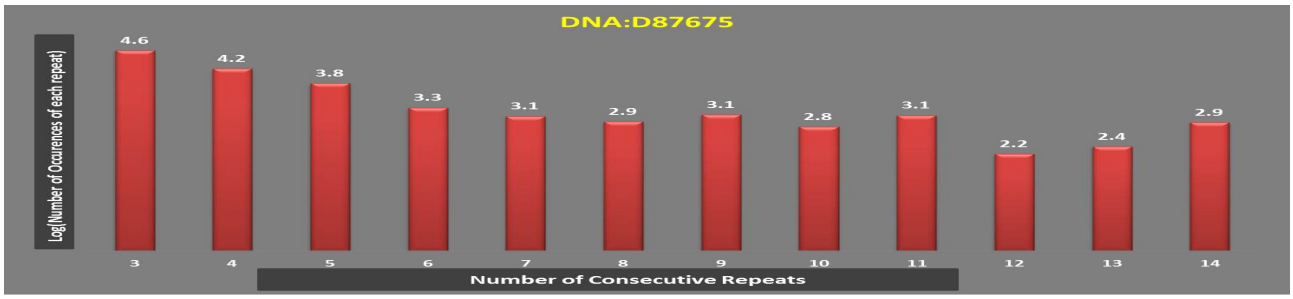

Figure 2. Histogram showing the no. of occurences of each consecutive repeat

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.

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. A detailed analysis on DNA sequences which will be discussed in the next section reveals that a fragment repeating for 3 or 4 times consecutively is more common than a fragment repeating for 255 times. In fact, this analysis done by us shows that maximum of 14 consecutive fragment repeats occur in a DNA sequence. The number of times each consecutive repeat is occurring can be seen in Figure 2.

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.

Table 1. Compression Ratios for Different Algorithms (in Bits/Bytes)

Algorithm	GENBIT	DNABIT	GenCodex
Best Case	1.125	1.04	0.017
Average Case	1.727	1.53	1.420
Worst Case	2.238	1.58	2.250

Though GenCodex supports 256 consecutive fragment repeats which is really huge, a detailed analysis showed that the number of consecutive fragment repeats in various DNA sequences is very less compared to 256. Initially, we set the maximum consecutive repeats to 100 and checked for different data-sets. We checked whether these sequences satisfy this condition. Later, we varied the size of this consecutive repeats. In this process it is revealed that the maximum consecutive fragment (4 characters) repeats occur not more than 14 times. The number of occurences of each consecutive fragment repeats can be seen in Figure 2. Due to this, some extra bytes are used which is not necessary.

Since GenCodex mainly concentrates on the number of consecutive repeats, it depends on the inherent biological property of the DNA sequence. This lead us to develop a new dynamic and adaptive DNA sequence compression algorithm which chooses a suitable algorithm, dynamically according to the biological properties of the input DNA sequence.

4. Adaptive DNA sequence Compression Algorithm

The compression techniques used on DNA sequences differs from the algorithms used on the general data since the position and formation of a DNA sequence varies and it is dependent on the biological properties of that particular genome. In a DNA sequence, there is a property where A is a complementary pair of T and C is a complementary pair of G. This means that A can be replaced with T but not with C or G and vice versa.

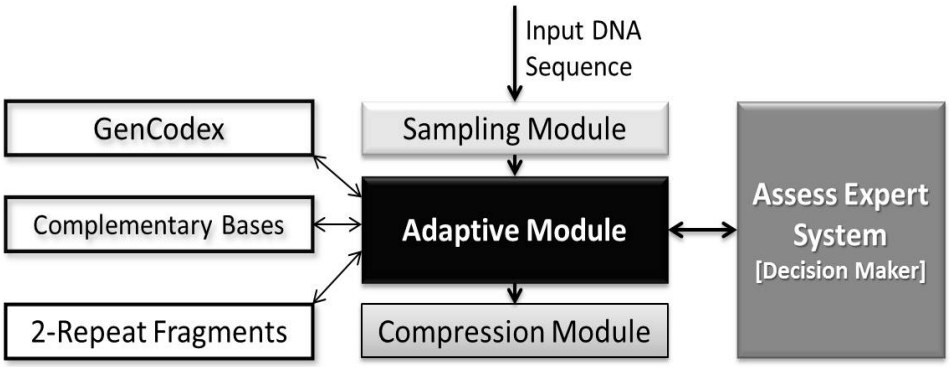

Figure 3. Adaptive compression algorithm for DNA sequences

The proposed GenCodex algorithm is efficient in compressing both repetitive and non-repetitive DNA sequences. One of the main reasons to propose a refined algorithm which is adaptive to the DNA sequences is that the existing algorithms are not adapting themselves to take advantage in using the inherent properties of the DNA. Some of the cases include number of consecutive repeats being very less or ignoring some of the basic properties like Complementary base pairs (Eg: A-T,G-C: ATGC TACG) and palindromes (Ex:ATTA ATTA), which may lead to compression.

The cases explained above are not considered in GenCodex as it concentrates only on the consecutive repeats. So GenCodex performance can be improved by considering the other cases as well. This lead us to develop a new adaptive DNA sequence compression algorithm which augments two more adaptive features to the proposed algorithm namely GenCodex.

It consists of two phases: In the first phase, the small subsequences or samples are taken randomly from the input DNA sequence to find the number of consecutive repeats occurring in the sequence. The sample subsequences are considered instead of the total input sequence because the input sequence may be huge in size. This may take considerable amount of time just finding the number of repeats which will reduce the time gained by compressing the sequences.

In the second phase, the selected samples are given to all the 3 modules namely GenCodex, Complementary and Genbit. The Adaptive compression algorithm can be seen in the Figure 3.

Adaptive DNA sequence compression algorithm can be explained in the following steps:

4.1 Assess Expert System

An expert system is a computer system that emulates the decision-making ability of a human expert [12]. Expert systems are designed to solve complex problems by reasoning about knowledge, like an expert, and not by following the procedure of a developer as is the case in conventional programming [13]. Expert systems is a form of AI software [14], [15], [16].

An expert system has a unique structure. It is divided into two parts, one fixed, independent of the expert system: the inference engine, and one variable: the knowledge base. To run an expert system, the engine reasons about the knowledge base like a human. Later, a third part i.e, a dialog interface was added to communicate with users [17]. This ability to conduct a conversation with users is called conversational.

Assess is a tiny expert system designed for the adaptive DNA sequence compression algorithm which takes a decision on which module to choose for compression based on the best results obtained for the given samples from the input sequence.

- Assess expert system is built using rule-based production language called CLIPS.
- The main working mechanism of this expert system is to take a call on optimal and best compression algorithm for a given data set.
- Once the adaptive module receives the results from compression methods i.e., compression ratios of 3 different algorithms. These results will be transferred to the Assess Expert system.

- These results will form knowledge base for the Assess expert system. The rule base and fact base are loaded with the optimal compression ratios to choose a particular algorithm for final compression.
- The priority level of a rule attribute can be assigned for some rules which focuses on the relevance of inheritance of biological properties of dataset.

5. Implementation

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 8^{th} bit in a code byte to 0.
- As described earlier, there is one code byte for every 8 bytes of compressed data. Setting the 8^{th} bit in the code byte to 1 implies that there is a repetition in the 8^{th} and 9^{th} bytes of the sequence and we need to allocate two bytes (occupying 8^{th} and 9^{th} bytes) in the compressed sequence, but this 9^{th} byte corresponds to second code byte which is already allocated.
- In this case, the 8^{th} bit in the code byte is set to zero and the 8^{th} byte in the compressed sequence represents just the 8-bit representation of the fragment.
- The same process repeats from the 9^{th} 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.

5.1 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 does 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.

5.2 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 number of threads launched increases upto 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.

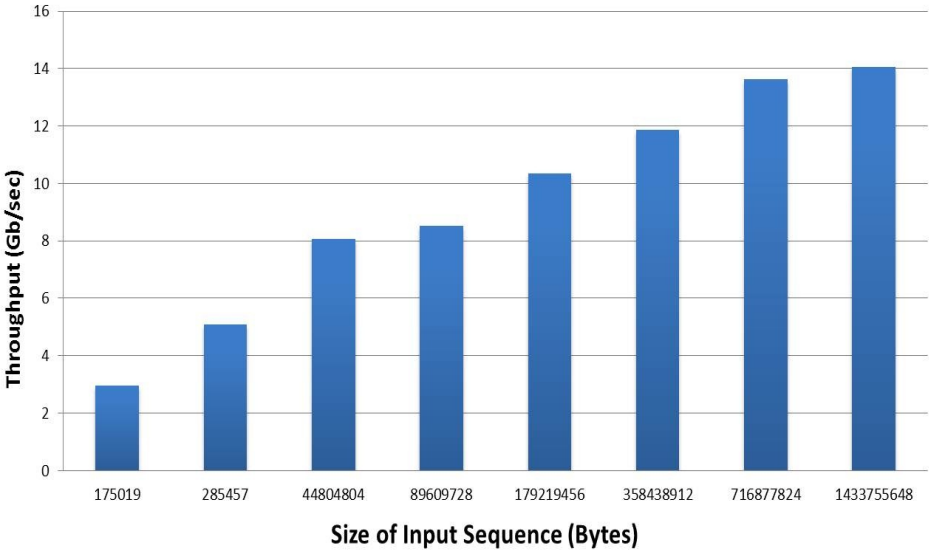

Figure 4. Compression throughput in Gb/Sec for different data sets (On Lonestar M2070 card)

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 4. 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 transfering the data between the host and the device. The compression throughput τ is the rate at which the data is compressed.

τ = size of the input data/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 5. The maximum throughput achieved here too is 14 GBps.

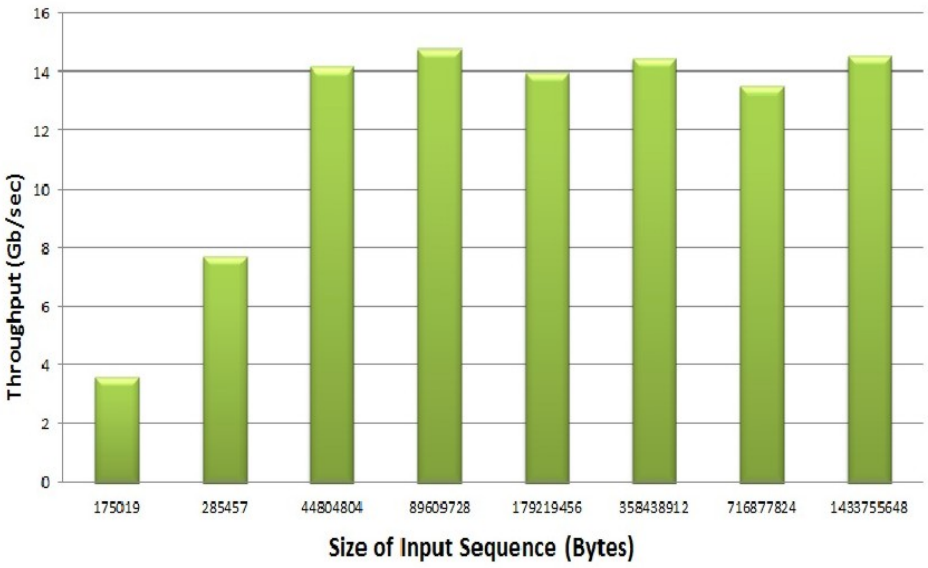

Figure 5. Compression throughput in Gb/Sec for different data sets (On Stampede K20 GPU)

6. 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.

7. Results

7.1 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 (upto 255 repetitions). Table 2 shows the compressed size in bytes for all the algorithms using different data-sets.

Table 2. Size of the Compressed Sequence for Different Algorithms (in Bytes)

DNA Sequence	Input size (in Bytes)	GenCompress	DNA Compress	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**

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 upto 11 on a 12-core M2070 card and upto 23 on M2070 GPUs for the data-sets used in our experiment. The results are shown in the Figure 6.

Table 3. Compression Ratios for Different Data-Sets(In Bits/Bytes)

DNA Sequence	Input Size (KB)	GenCompress	DNA Compress	Genbit	**GenCodex**
HSCOMT2	1.700	2.05	1.95	1.84	**1.77**
HUMCYC1A	2.200	2.03	1.95	1.87	**1.79**
HSU37106	2.250	2.03	1.98	1.93	**1.87**
HSGTRH	3.930	2.02	1.96	1.86	**1.80**
HUMGALK1A	7.000	1.92	1.92	1.90	**1.83**
HSU01102	4.200	1.93	1.96	1.84	**1.77**
HSC1INHIB	16.309	1.85	1.94	1.75	**1.69**
HSCST4	3.400	2.00	1.93	1.90	**1.85**
HUMA1ATP	4.786	2.00	1.95	1.85	**1.78**
HSTNT2	8.600	1.90	1.89	1.88	**1.82**
HUMRBPA	8.682	1.97	1.94	1.90	**1.84**
HUMHKPQZ	2.300	2.12	2.02	1.93	**1.86**
HUMRETBLAS	175.019	1.83	1.90	1.78	**1.70**
HUMTBGA	6.275	2.03	1.96	1.89	**1.83**
HSAT3	13.340	1.91	1.94	1.79	**1.73**
D87675	285.457	1.86	1.92	1.81	**1.74**

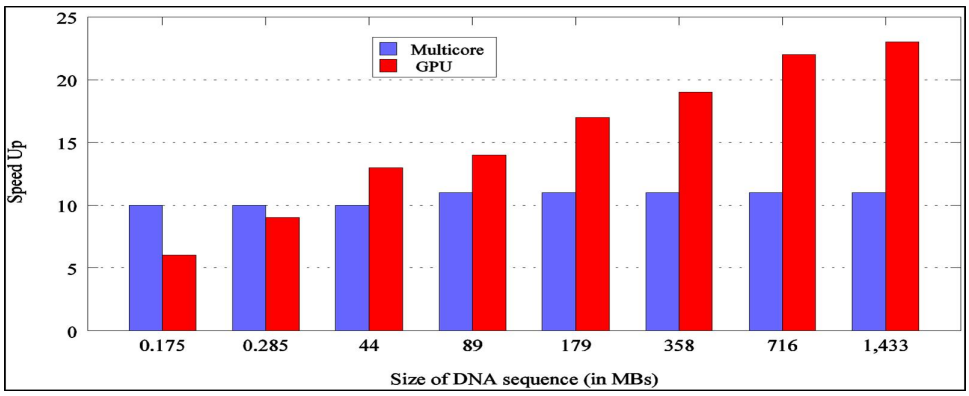

Figure 6. Speedup on multi-cores and GPUs (On Lonestar M2070 card)

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 upto 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 is 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.

Table 4. Timings (in Milliseconds) on Multi-Core And GPUS on LONESTAR

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	1.970	1.940
89609728	141.401	13.700	10.573
179219456	283.237	27.440	17.355
358438912	567.532	57.225	30.268
716877824	1130.988	110.880	52.633
1433755648	2272.355	219.911	102.601

Table 5. Timings (in Milliseconds) 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	173.810	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	98.560	56

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 Figure 7.

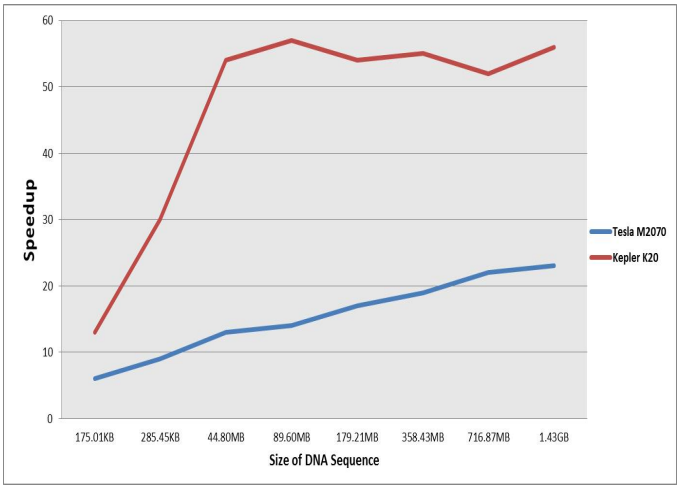

Figure 7. Speedup on Lonestar M2070 and Stampede K20

7.2 Adaptive DNA Compression Algorithm

Adaptive DNA compression algorithm mainly focuses on accurate compression by choosing the compression algorithm based on the biological properties of the given input sequence. Biological properties such as consecutive short repeats (4-mars) and (8-mars), long repeats upto 256 and complementary base pairs, algorithms developed to compress the given input sequence based on these properties. Compression algorithm is selected based on the effect of a particular biological property on given input sequence.

7.2.1 Sampling & Selection of Algorithm

- Table 6 presents the sampling results of the Adapative DNA compression algorithm.
- In this Table 6, GenCodex, 2-Mar Consecutive fragment Compressor, Complementary base pairs compressor are evaluated using input DNA sequence of varying sizes.
- Sampling is done on 10% of the data from the given input sequence, this sampling process includes compressing the 10% data with all the three algorithms.
- A particular algorithm is selected by the expert system using the compressed sequence sizes produced by the three algorithms during the sampling process.
- In Table 6, sampling is done using entire input sequence to show the consistency of the sampling data and overall data.
- Table 6 presents results for 100% input data, the main reason for this is that it might happen that a particular biological property occurs more in the selected sample of either 10% or 20% and It may not occur in remaining 90% or 80% of the data. This results in poor compression.
- DNA sequence compression algorithm is selected based on its sampling results consistency, here consistency means that if a particular algorithm performs for all the samples then that particular algorithm is selected for compressing entire input sequence.
- For example, In Table 6, the first row presents input DNA sequence *HSU1TA* which is sampled by selecting 10% of the data and it got compressed to 265, 289, 219 bytes by *GenCodex*, *2-Mar* and *Complementary* respectively, for 10% data *Complementary* compressor works better.
- In the next step to check the consistency of *Complementary* property in given input sequence, 20% of the data is selected randomly from *HSU1TA*.
- Compression results of 20% data i.e., 434, 472, 396 bytes when compressed using *GenCodex*, *2-Mars* and *Complementary* respectively. Once again *Complementary* property occurs more in the *HSU1TA* sequence.
- The input DNA sequence *HSU1TA* can be compressed using *Complementary* compressor, this results in good and optimal compression.
- When we executed all the three algorithms on 100% data of *HSU1TA* it is found that *Complementary* compressor performs better than other two. In Table 6, the first row presents 2063, 2038, 1825 compressed sequence sizes of *HSU1TA* for *GenCodex*, *2-Mar* and *Complementary* compressors respectively.

- This shows that *Complementary* compressor dominates the performance even on entire data set.
- Similarly data sets in Table 6 *HUMACGT* & *HSHTUM1* performs better when we use *GenCodex* & *2-Mar* respectively. Results can be observe from Table 6.

Table 6. Adaptive DNA Compression Algorithm Sampling Results (in Bytes)

DNA	Size of DNA	**GenCodex**			**2-Mar**			**Complementary**		
		10%	**20%**	**100%**	**10%**	**20%**	**100%**	**10%**	**20%**	**100%**
HSU 1TA	13347	265	434	2063	289	472	2038	**219**	**396**	**1825**
HUM ACGT	285457	**5455**	**12176**	**64537**	6231	128 50	64982	5861	13151	65996
HSHT UM1	89712156	2081 201	4162408	20161982	**1963411**	**4011709**	**20003229**	2192314	4034156	25181893

Table 7. Adaptive DNA Compression Algorithm Sampling Results (Compression Ratios (Bits/Byte))

DNA	Size of DNA	**GenCodex**			**2-Mar**			**Complementary**		
		3%	**5%**	**100%**	**3%**	**5%**	**100%**	**3%**	**5%**	**100%**
HSU 1TA	13347	1.58	1.32	1.20	1.73	1.41	1.22	**1.3**	**1.18**	**1.09**
HUM ACGT	28545 7	**1.52**	**1.7**	**1.8**	1.74	1.80	1.82	1.64	1.84	1.85
HSHT UM1	8971 2156	1.85	1.89	1.79	**1.75**	**1.78**	**1.78**	1.95	1.79	2.24

8. 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. This lead us to develop an adaptive DNA compression algorithm which takes samples from the given input sequence and chooses the correct algorithm depending on the number of consecutive repeats.

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 upto 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.

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 simultaneoulsy 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.

Acknowledgment

We would like to thank kali uday and satyanvesh for their initial work and implementation. This work was partially supported by a NVIDIA grant under professor partnership program and the Extreme Science and Engineering Discovery Environment(XSEDE), which is supported by National Science foundation grant number OCI-1053575.

Competing Interests

The authors declare that they have no competing interests.

Authors' Contributions

All the authors contributed significantly in writing this article. The authors read and approved the final manuscript.

References

[1] Amar Mukherjee, Matt Powell, Tim Bell, Don Adjeroh and Yong Zhang., *DNA Sequence Compression using the Burrows-Wheeler Transform*, IEEE (2002).

[2] T.C. Bell et al., Text Compression, Prentice Hall (1990).

[3] A. Grumbach and F. Tahi, Compression of DNA sequences, in *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, USA, March 30-April 2, 1993.

[4] X. Chen, S. Kwong and M. Li, A compression algorithm for DNA sequences, in *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, Tokyo, Japan, April 8-11, 2000.

[5] X. Chen, M. Li, B. Ma and J. Tromp, DNA compress: Fast and effective DNA sequence compression, *Bioinformatics Applications Note*, **18** (2002), 1696–1698, `https://www.researchgate.net/profile/Xin_Chen78/publication/10984449_DNACompress_fast_and_effective_DNA_sequence_compression/links/5445db610cf22b3c14ddf08b.pdf`

[6] P.R. Rajeswari and A.A. Rao, DNABIT compress genome compression algorithm, *Book on Bioinformation* **5** (2011), 350–360.

[7] P.R. Rajeswari and A.A. Rao, GENBIT compress – algorithm for repetitive and non repetitive DNA sequences, *International Journal of Computer Science and Information Technology* **2** (2010), 25–29.

[8] T. Matsumoto, K. Sadakane and H. Imai, Biological sequence compression algorithms, *Genome Informatics Workshop*, Universal Academy Press, 2002, 43–52.

[9] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory* **23** (1977), 337–343.

[10] J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding, *IEEE Transactions on Information Theory* **24** (September 1978), 530–536.

[11] B. Ma, J. Tromp and M. Li, PatternHunter: faster and more sensitive homology search, *Bioinformatics* (2002), 440–445.

[12] P. Jackson, *Introduction to Expert Systems*, Addison Wesley, 1998, 3rd edition, ISBN 978-0-201-87686-4.

[13] R. Barzilay, D. Mccullough, O. Rambow, J. Decristofaro, T. Korelsky, B. Lavoie and Cogentex Inc, *A New Approach to Expert System Explanations*, in *9th International Workshop on Natural Language Generation*, 1998, 78–87.

[14] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice Hall (2003).

[15] N.J. Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers, San Francisco (1998).

[16] P. McCorduck, *Machines Who Think*, A.K. Peters, Ltd. (2004).

[17] C.G. Koch, B.A. Isle and A.W. Butler, Intelligent user interface for expert systems applied to power plant maintenance and troubleshooting, *IEEE Transactions*, March 1988, 3rd volume.