PARALLELIZATION OF WEIGHTED SEQUENCE COMPARISION BY USING EBWT

Binay Kumar Pandey[1], Rajdeep Niyogi[1], and Ankush Mittal[2]

[1]Electronics and Computer Engineering Department,
Indian Institute of Technology Roorkee,
Roorkee, India
(binaydec, rajdpfec)@iitr.ernet.in

[2]Computer Science and Engineering Department,
College of Engineering Roorkee
Roorkee, India
dr.ankush.mittal@gmail.com

ABSTRACT

In this paper, we describe the design of high-performance extended burrow wheeler transform based weighted sequence comparison algorithm for many core GPU$_s$ taking advantages of the full programmability offered by compute unified device architecture (CUDA) and its standard library thrust. Our extended burrow wheeler transform based weighted sequence comparison algorithm with thrust library implementation on CUDA is the fastest implementation of weighted sequence comparison algorithm than the our previous implementation of extended burrow wheeler transform based weighted sequence algorithm without using thrust library, and it is on average 56.3X times faster. Moreover, our present time implementation is also competitive with CPU implementations, being up to 2.9X times faster than comparable routine on 2.99 GHz Intel Pentium (R) 4 CPU with 3 GB RAM.

KEYWORDS

Extended Burrow Wheeler Transform, CUDA, Molecular Weighted Sequence

1. INTRODUCTION

In bioinformatics and molecular biology research, sequence comparison is one of the most important tasks. Biological sequence comparison algorithms usually finds a match between nucleotide sequences which allows to determine the differences that occurs due to substitutions, additions and deletions of nucleotides between them [1]. The foundation process of identifying new nucleotide sequence is generally done by finding the most similar match of an unknown nucleotide sequence to a known biological sequence. The same technique adopted to find out mutations that triggers a disease and is also a substantial part of tracing the evolution of a certain organism [2]. The dynamic programming based smith–waterman algorithm is considered as the only comparison algorithm guaranteed that return an optimal result which is suitable for both of protein and DNA sequences [4]. However, this algorithm took considerable amount of time even to compare two small length sequences and also not suitable for molecular weighted sequences. Thus, we try to alleviate the limitation of this algorithm in terms of its execution time by implementing extended burrow wheeler transform based molecular weighted sequence comparison algorithm.

The implementation of this intuitive idea for large weighted molecular sequence [4] can have take enormous amount of computation time and memory. This can be proved by considering a practical example of comparing molecular weighted sequence of 300 positions with molecular weighted pattern of 10 positions and in each position there is a possibility of four characters. In worst case this weighted sequence gives 4^300 possible sequences and each sequence is 300 characters long, and in similar manner a molecular weighted pattern generates 4^10 possible pattern and each pattern is 10 characters long. Solving sequence comparison problem for such a kind of molecular weighted sequence and pattern requires very large memory space and computation time. This are all the main reasons which motivate us to perform parallel implementation of extended burrow wheeler transform based molecular weighted sequence comparison algorithms on compute unified device architecture($CUDA$). Our whole paper is organized into the following sections. Section II gives brief explanations of burrow wheeler transform. Section III gives a study on extended burrow wheeler. Section IV consists of brief description of parallel architecture that is compute unified device architecture CUDA. Section V gives detail of comparing sequence using extended burrow wheeler transform. Section VI is dedicated to parallel implementation of weighted sequence comparison algorithm using extended burrow wheeler transform. Section VII gives detailed table of our findings. Section VIII dictates final conclusion of our experiment performed

2. BURROW WHEELER TRANSFORM: Preliminaries

In the field of lossless textual data compression, burrows-wheeler transform [5] can be considered as an extremely useful tool. The main idea of burrow wheeler transform is to produce a permutation of an input word ω, defined over an alphabet A, so that it becomes easier to compress the word. In fact this transformation keeps a group characters collectively so that the possibility of finding a character close to another instance of the same character is considerably increased. Burrow wheeler transform convert an input word $\omega = \omega_0\omega_1 \ldots \ldots \omega_n$by constructing all n cyclic rotations of ω, perform lexicographical sorting and then extracting the last character of each rotation. The output of burrow wheeler transform is mainly consists by the sequence of these characters. Furthermore this transformation is also computes the index I, that is the row that denotes the position of original word in the sorted list of rotations. For example, suppose we want to compute burrow wheeler transform of word $\omega = banana$. In (Figure 1) shows the matrix that consists of all cyclic shifts of ω, that are further lexicographically sorted.

$$\begin{array}{cccccccc}
 & & F & & & & & L \\
 & 1 & a & b & a & n & a & n \\
 & 2 & a & n & a & b & a & n \\
 & 3 & a & n & a & n & a & b \\
I & 4 & b & a & n & a & n & a \\
 & 5 & n & a & b & a & n & a \\
 & 6 & n & a & n & a & b & a
\end{array}$$

Figure 1. The matrix contains lexicographic sorted list of all cyclic rotations of the word $\omega =$ banana

The last column L of the matrix represents $BWT(\omega)$ = *nnbaaa* and $I = 4$ because the original word *banana* appears in row 4. The first column F, instead, contains the sequence of the characters of ω lexicographically sorted.

3. AN EXTENSION OF BURROW WHEELER TRANSFORM FOR MULTISET OF WORDS

In this section a description of an extension of the burrows–wheeler transform to a multi-set of words and few of its properties are illustrated. Such an extension is strictly related to a bijection, introduced in [6], between the multisets of words over a finite alphabet A and the words of A^*. In paper [7] it is shown that the burrows wheeler transform is connected to a particular case of the bijection given in [6] . This important statement has stimulated the definition of the extension burrow wheeler transform denoted by $ebwt$ for a multi-set of primitive words. In next subsection, some basics of order of relation between words and some proposition and definitions are given.

3.1. Order Relation between Words

A very deep understanding of an order relation between words that is different from the usual lexicographical one is very essential to define the extended burrow wheeler transformation. In [10] author describe every word $v \in A^*$ as a power of a primitive word, in other word it can also be said that there exists a distinctive primitive word ω and a distinctive integer k such that $v = \omega^k$, where ω by $root(v)$ and k is denoted by $\exp(v)$.

Let suppose u be a word in A*, then u^ω denotes the infinite number of word that are obtained by iterating u in infinitely number of times, i.e. $u^\omega = uuuuu\ldots u$. The lexicographic ordering denoted by $<_{lex}$ is generally defined on infinite words length. In order to develop much better understanding let us consider two words of infinite length $x = x_1x_2 \ldots$ and $y = y_1y_2 \ldots$, with $x_i, y_i \in A$, then $x <_{lex} y$ if there exists some index $j \in \mathbb{N}$ such that $x_i = y_i$ for $i = 1,2,\ldots,j-1$ and $x_i = y_i$. it is very important to note that if $x = y$, then relation $<_{lex}$ is said to be not defined. It is also observing that $u^\omega = v^\omega$ if and only if $root(u) = root(v)$. In order to support above mentioned fact some important definition, proposition with proof and examples are given below.

Definition 1. Let us consider two words u, v made up of finite alphabets which are elements of A. Then it is said that

$$u \leq_\omega v \Leftrightarrow \begin{cases} \exp(u) \leq \exp(v) & if\ root(u) = root(v) \\ u^\omega <_{lex} v^\omega & otherwise \end{cases} \qquad (1)$$

This order relation shown in "Eq. (1)" is different from the lexicographic one, this can be explain by considering following example $ab <_{lex} aba$ but $aba \leq_\omega ab$. even though when $root(u) \neq root(v)$ the $\preccurlyeq_\omega$ -order of u and v is defined by considering word of infinite length, the proposition given below shows that it is possible to decide mutual $\preccurlyeq_\omega$-ordering between these words by just extending them up to the length $|u| + |v| - \gcd(|u|, |v|)$. Such a bound is a outcome of a well-known consequence of periodicity on words given in [10, 11].

Proposition 2 Let us assume two given words as a u and v, with $root(u) \neq root(v)$, and then order of relation is defined by "Eq. (2)".

then $u \preccurlyeq_{\omega} v \Leftrightarrow pref_k(u^{\omega}) <_{lex} pref_k(v^{\omega})$,

where $k = |u| + |v| - \gcd(|u|, |v|)$ (2)

For any given finite or infinite length word ω, $pref_k(\omega)$ denotes the prefix of word ω of length k.

Example 3. Consider the two words $u = aba$ and $v = abab$. After monitoring each character of each word one can find that $v \preccurlyeq_{\omega} u$ and u^{ω} and v^{ω} differ for the character in position 6 = 4+3 – 1. It is also observe that $u <_{lex} v$

$$\overbrace{aba}^{u}\,\overbrace{aba}^{u}\,\overbrace{ab\ldots.}^{u}$$
$$\underbrace{abab}_{v}\,\underbrace{abab\ \ldots.}_{v}$$

B. The Extended Burrows–Wheeler Transform

This section is basically introducing extended burrow wheeler transform under the assumption that the words considered are primitive. In fact this supposition is not very restraining, because in practice almost all the textual data processed are primitive and if not then it become primitive by adding an end-of-string symbol.

Let us consider $S = \{u_1, \ldots\ldots\ldots. u_k\}$ be a multiset of primitive words of A^* that are not necessarily distinct. Then $||S||$ is denoted by "Eq. (3)".

$$||S|| = \sum_{i=1}^{k} |u_i| \qquad (3)$$

and the value of H is given by "Eq. (4)".

$$\max\{ |u_i| + |u_j| - gcd(|u_i|, |u_j|) \mid i, j = 1, \ldots, k\}. \qquad (4)$$

Consider again the set $C(S)$ that represents all the conjugates of the words in S. Then each $\omega \in C(S)$ can associate to the triplet $(pref_H(\omega^{\omega}), L(\omega), \chi_s(\omega)$.where $pref_H(\omega^{\omega})$ represents the prefix of ω^{ω} of length H, $L(\omega)$ denotes the last character of ω and $\chi_s$represents the characteristic function of S, and represented by "Eq. (5)".

$$\chi_s(\omega) = \begin{cases} 1, & if\ \omega \in S \\ 0, & otherwise \end{cases} \qquad (5)$$

Now consider all these triplets set and sort it by taking first field as sorting key and then use $<_{lex}$ relation. According to proposition 5, this $<_{lex}$ relation applied on the words of $C(S)$ produces the same order words as obtained by applying the $\preccurlyeq_{\omega}$-order relation on the words of $C(S)$. This sorted list is arrange in (Figure 1). The sequences obtained by concatenating the second and the third components triplet $(pref_H(\omega^{\omega}), L(\omega), \chi_s(\omega)$ are denoted by $M_L(S)$ and $M_{\chi}(S)$. It is important note down that in places where there is no threat of vagueness, then the notation M, M_L and M_{χ} may use in place of $M(S)$, $M_L(S)$ and $M_{\chi}(S)$, respectively. If M_C

denotes the sorted list with respect to the $\preccurlyeq_{\omega}$-order, of the conjugate words in $C(S)$, then M_L is the word obtained from the concatenation of the last characters of elements in M_C and M_{χ} is the characteristic vector which show the elements in the list that are coming from S.

Definition 4. The pair $(M_L,\ M_{\chi})$ denoted by $ebwt(S)$ is the outcome of extended burrows wheeler transform for a multi-set of word denoted by S.

Example 5. Let us take a multi-set of word denoted by $S = \{abacb, cbac, bcab, acba\}$. (Figure 2) third column represent $pref_6's$ component that are obtain after performing lexicographic sorting on its rows. For sake of clearness we add also the column (the first one) M_C containing the $\preccurlyeq_{\omega}$-ordered list of all $\omega \in C(S)$. The second column contains the $<_{lex}$-sorted list of prefixes of length $H = 6$ the third column the word M_L and the fourth column the characteristic array of S, M_{χ}.

The complexity of the algorithm for computing $ebwt$ is upper bounded by the time needed to get the sorted list of the conjugates of the words in S, that can be handled by a suitable suffix array generalization. A generalized type of the "skew algorithm [8]" for the construction of the generalized suffix array, make it possible to obtain such a sorted list in linear time on the total size of the set S. Then the algorithm has total complexity $O(||S||)$.

The proposition given below shows how the two important properties connecting the characters of M_L and M_F.

Proposition 6. Consider S be a multi-set of that hold primitive words and suppose $ebwt(S) = (M_L,\ M_{\chi})$. Let M_F denotes the sequence of the first characters of the sorted list. Then following properties must hold:

(1) For every i where $M_{\chi} = 0$,, $M_L[i]$ follows $M_F[i]$ in one word in S.
(2) For a predetermined character $a \in A$, its occurrences in M_F appear in the same order as in M_L, i.e. its m^{th} instance of a in M_F corresponds to its m^{th} instance of a in M_L.

Remark 7. A very important fact of extended burrow wheeler transforms is that, when the set S has only one element, and then the extended burrow wheeler transformation works exactly as same as the burrows–wheeler transform. In fact, the $\preccurlyeq_{\omega}$-ordering of all the conjugates of a word is equivalent to the lexicographic ordering, because there is only single word and all the possible conjugates of this single words have the same length. Furthermore, according to remark 4, the set of indices holds a single value. Therefore, if $S = \{\omega\}$ then $ebwt(S) = bwt(\omega)$.

S. NO	M_C	*Pref₅*	M_L	$M\chi$	M_F
1	aacb	aacba	b	0	a
2	abacb	abacb	b	1	a
3	abbc	abbc	c	0	a
4	acba	acba	a	1	a
5	acbab	acbab	b	0	a
6	accb	accb	b	0	a
7	bacc	baac	c	0	b
8	babac	babac	c	0	b
9	bacba	bacba	a	0	b
10	bacc	bacc	c	0	b
11	bbca	bbca	a	0	b

12	bcab	bcab	b	1	b
13	cabb	cabb	b	0	c
14	cbaa	cbaa	a	0	c
15	cbaba	cbaba	a	0	c
16	cbac	cbac	c	1	c
17	ccba	ccba	a	0	c

Figure 2. The Table contains output of $ebwt(S)$ that is the pair (M_L, M_χ) where $M_L = ccbbbcacaaabba$ and $M_\chi = 10000000100011$

4. COMPARING SEQUENCES BY USING EBWT

In this section we apply our extended transformation ebwt in order to introduce a new method for comparing sequences. The new distance between words defined here is simple and efficient to compute, and it is particularly advantageous in the case of comparison of a set of sequences. We remark that our distance is not based on sequence alignment. Several alignment-free distance measures have recently been introduced [11, 12, 13, 14] since they better fit with the problem of comparing genomic sequences than the methods based on sequence alignment. In fact, alignment-based methods compare sequences by considering only local edit operations on their fragments. Instead, the recent developments in genome sequences technologies have allowed to handle the complete genome of many different species, and have highlighted that, in order to capture evolutionary and functional mechanisms of different species, we need to consider a new set of sequence modifications that involve recombination or shuffling of segments of genome. The distance we define takes into account such kind of modifications and therefore it can be successfully applied to compare genomic sequences.

The new notion used to measure distance between two sequences is based on the following intuitive idea: when $ebwt$ is applied to the set $S = \{u,\ v\}$, if the same segment s occurs both in u and v, then the conjugates of u and v starting by s are likely to be close in the $\preccurlyeq_\omega$-sorted list of conjugates. This implies that the greater is the number of segments shared by u and v, the greater is the mixing of the conjugates of u and v in the sorted list. The comparison method based on transformation $ebwt$ will measure how similar u and v are, by taking into account how much their conjugates are mixed. In this section, in order to show an application of the extended burrow wheeler transformation, a distance measure is define that computes the number of alternations in the above list between the conjugates of u and those of v. A more detailed study of the class of distance measures based on $ebwt$ can be found in [15], where other different formalizations of distance measures based on ebwt are given. Formally, let

$S = \{u_1, u_2, \dots, u_k\}$ be a multiset of primitive words inA^*, let $m = \sum_{i=1}^{k} |u_i|$ and let $M_{\mathcal{C}} = w_1, w_2, \dots\dots, w_m$ be the sorted list of the conjugates of the elements of S obtained during the computation of $ebwt(S)$.

Consider the new alphabet $\sum = \{U_1, U_2, U_3, \dots\dots, U_K\}$. The coloring of $ebwt(S)$ is the map $:\{1, 2, \dots\dots, m\} \rightarrow \sum$ defined, for as:

$\gamma(i) = U_j$ if w_i is a conjugate of u_j.

We denote by $M_\gamma[i]$ the word over $\sum^*$ such that $M_\gamma[i] = \gamma(i)$. .

The following example describes the coloring of$ebwt$, by adding the sequence M_γ as a column to the table M associated to the computation of $ebwt$, as shown in (Figure 3).

S. No	M_C	M_L	M_γ	M_χ
1	aabb	b	U	1
2	aabb	b	V	0
3	abba	a	U	0
4	abba	a	V	1
5	abab	b	Z	1
6	abab	b	Z	0
7	baab	b	U	0
8	baab	b	V	0
9	baba	a	Z	0
10	baba	a	Z	0
11	bbaa	a	U	0
12	bbaa	a	V	0

Figure 3. Shows extended burrow wheeler transform of mutiset S in which column $M_\gamma$of the coloring is added.

Example 8. Let $S = \{u, v, z\}$, where $u = aabb$, $v = abba$ and $z = abab$. Let U, V, Z are the colors associated, by the map γ, to u, v, z, respectively. As one can see in (Figure 3), $M_\gamma = UVUVZZUVZZUV$.

The definition of coloring allows us to introduce a new notion of distance measure between two sequences that takes into account the alternation of the symbols coming from different sequences in the output of the transformation $ebwt$.

Definition 9. Let $u, v \in A^*$ be two sequences and let us consider $M_\gamma(u, v) = U^{n_1}V^{n_2}U^{n_3} \ldots V^{n_k}$. We define the measure $\delta(u, v)$ as follows:

$$\delta(u,v) = \sum_{i=1,\ n_i \neq 0}^{k} (n_i - 1)$$

A description of the computation of distance δ is given in the Example 11. The following proposition provides some properties of the measure.

Proposition 10. The following statements hold:

- $\delta(u, v) = \delta(v, u)$, i.e. the measure δ is symmetric.

If u and v are conjugate, then $\delta(u, v) = 0$.

- If u' is a conjugate of u and v' is a conjugate of v, then $\delta(u, v) = \delta(u', v')$. Therefore, δ is a distance measure for conjugacy classes.

M_C	M_γ
abbc	V
acb	U
bac	U
bbca	V
bcab	V
cabb	V
cba	U

Figure 4. The $\preccurlyeq_\omega$-sorted list of conjugates of *acb* and *bcab* with coloring

Example 11. Let us consider the sequences $u = acb$ and $v = bcab$. In (Figure 4) the sorted list of conjugates of u and v and the coloring of $ebwt(u, v)$ are shown. In this case $M_\gamma(u, v) = VU^2V^3U$, so according to definition 24, the net computed distance between u and v is $\delta(u, v) = 3$.

5. A MULTI-CORE ARCHITECTURE: CUDA

The architecture benefits of both CPU and GPU for general purpose non graphics GPU computing are obtained by developing a unified architecture of GPU and CPU called compute unified device architecture. The programs which are usually created for CUDA hardware [9], executes on the both CPU and GPU. A parallel component of program is called kernel that runs parallel on the GPU. The kernel cannot access the data from CPU's main memory directly, but before launching kernel it's require input data must be copied to the GPU's memory, and output data is also first be written to the GPU's memory and then copied to CPU memory.

The architecture of CUDA and how data flow occurs between GPU and CPU is shown in (Figures 5-6). The necessary amount of memory required by the kernel must be pre-allocated, and the kernel is not able to use recursion or other similar operation that requires a stack, but conditionals and loops are allowed. Additionally, the number of registers present in each multiprocessor is limited and if the number of registers used by kernel is too high then multiprocessor schedules less stream processors [10] to perform computation simultaneously. This causes, a high-performance kernel code to be very careful in order to reduce the number of registers used and put limit on the amount of branching. The better flexibility of CUDA still not able to solve the very basic problems of small cache and associated high memory latency for memory intensive programs present in GPU card G80's.

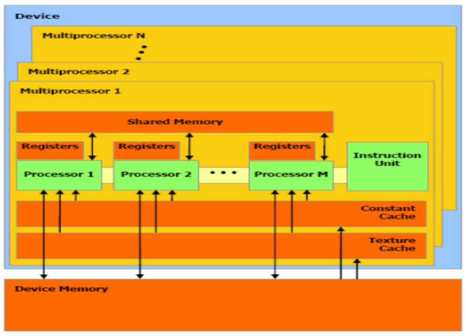

Figure 5. Architecture of CUDA

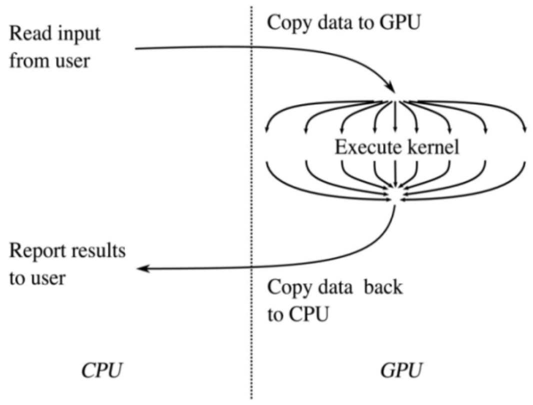

Figure 6. Typical GPGPU application flow

6. PARALLEL IMPLEMENTATION OF WEIGHTED SEQUENCE COMPARISION ALGORITHM USING EBWT

This section basically gives the detail of the implementation steps, that we carried out for parallelization of extended burrow wheeler transform based weighted sequence comparison algorithm on CUDA without Thrust library and with Thrust library.

6.1. Profiling

The most time consuming portion of the sequential code is determine by using Gnu profiling technique. In this technique we obtained a flat profile of sequential code by using gprof utility available in Gnu compiler. The generated flat file contains a set of information. For instance percentage of time required finishing the execution of particular function, number of call made for the function, self millisecond per call, and total milliseconds per call. In addition to this, gprof sort this information according to the functions that uses greater percentage of time. The utility gprof gives fast and simplest way to do function level profiling of the code.

Our main objective behind the use of gprof utility is to determine the expensive functions with respect to time and then parallelize the sequential code accordingly. So that these time consuming functions takes less time and we can gain a significant improvement on sequential code. In order to perform the profiling of sequential code, we compile the code with –pg option and then execute the code as usual.

After the execution of code is completed, an output file named gmon.out is created and this file can studied for parallelization purpose of sequential code. The profiling detail of our sequential implementation of extended burrow wheeler transform based weighted sequence comparison algorithm is shown in "Table 1" and as per our analysis performed on this profiling table, we found that calculate_distances and sort_data are the function that takes maximum percentage of time during their execution. So we parallelized these two functions by exploiting the parallelization features available in CUDA.

Table 1. Profiling results

Each sample counts as 0.01 seconds.

% time	Cumulative seconds	Self seconds	calls	Self s/call	Total s/call	Function name
99.31	40.36	40.36	1	40.36	40.36	calculate_ distances(conjugate*, unsigned int, unsigned int, unsigned int, bool)
0.12	40.41	0.05	1	0.05	0.24	sort_data (conjugate*, unsigned int, conjugate*)
0.12	40.46	0.05				main
0.10	40.50	0.04	1693856	0.00	0.00	std::vector<std::string, std::allocator<std::string >>::operator[](unsigned int)
0.05	40.52	0.02	5297631	0.00	0.00	__gnu_cxx::__normal _iterator<std::string*, std::vector<std::string, std ::allocator<std::string> > ::operator*() const
0.05	0.02	40.54	1703906	0.00	0.00	__gnu_cxx::__normal iterator<std::string *, std::vector<std::string, std::allocator<std::string> >
0.05	40.58	0.02	10046	0.00	0.00	gnu_cxx::__normal _iterator <std::string*, std::vector< std::string, std:: allocator<std::string> > >

6.2. Parallelization Techniques

Our implementation of parallel extended burrow wheeler transform based weighted sequence comparison on CUDA and CUDA with thrust library is divided into four phases that are : finding possible sequence and pattern from weighted sequence and weighted pattern, generating conjugate for each of the possible sequences and patterns, sorting of conjugates, and computation of distance between possible sequences and possible pattern. Each of these phases is handled by either CPU or GPU processor available in CUDA. The selection of CPU and

GPU processor for the execution of above mentioned phases are based on the parallelization feasibility of that phase.

The finding of possible sequences and pattern from weighted sequences and pattern is one of the typical exponential problems and it is not feasible for parallelization because a brute force computation is required to enumerate all possible sequences and patterns. However, it is further possible to improve this brute force computation by applying some heuristics, in which we replace the exponential method for finding possible sequence and pattern into a three subparts that are: coloring of weighted sequence, tree construction of weighted sequence and finally tree traversal.

The coloring of each character positions in a weighted sequence is performed according to some set of rules. A character position is colored with white if one of the possible characters at that position has a probability of appearance equal to 1. A character position is colored gray if only one of the possible characters has probability of appearance greater than the presumed threshold value denoted by θ and if more than one character has probability of appearance greater than the threshold value θ, then that particular position is colored with black.

As soon as coloring is over, a tree is constructed for molecular weighted sequence by scanning it from left to right and if white or gray color appears then corresponding characters is simply added as a one of the node of tree. If a black color is encounter, then branching occurs in the tree.

Once the construction of tree is over we traverse the tree from leaves to root and a weight of each character at each node is multiplied up to root. All the sequence which are generated during tree traversal contains total weighted of appearances that are obtained by multiplying weight of each character is greater than presumed threshold value θ.

The possible sequences and pattern generation operation is followed by their conjugate generation. The conjugate generations of possible sequences or possible patterns were performed in parallel. Every possible sequence or pattern was assigned to a block, wherein, each thread presents in per block was responsible for the generation of one conjugate. For better understanding, let us taken example in which it was shown that how the fifth conjugate of first sequence or first pattern having length 10 characters was formed. The thread which would be responsible for the generation of fifth conjugate would be the fifth thread in the block to which first possible sequence or pattern was assigned.

This fifth thread would simply copy the content of the first sequence or pattern from indices 6 to 10 into a temporary memory location and then copying the content from indices 0 to 5 of first sequence and pattern into the same temporary memory location and concatenate them to form required conjugate. It is found from our deep observation that, the number of conjugate generated for each possible sequence or pattern is equal to the number of characters contained in it. Thus, the maximum number of threads required for conjugate generation would be equal to total number of possible sequence and pattern generated multiplied by maximum width of possible sequence or a pattern.

As the conjugate generation of a sequence is over, the sorting of these generated conjugates is begins. To perform sorting of conjugates in parallel on CUDA is a quite challenging task, because of some strict constraints imposed by underlying CUDA architecture. Our parallel implementation of traditional Bitonic sort [16] on CUDA has a large number of scatter read write operations that are performed on global memory, make it inefficient than sequential sorting algorithms . Therefore we looks towards some more improved sorting scheme which gives remarkable results on its parallel implementation on CUDA and we found that, thrust [17] is a set of library providing standard library functionality in CUDA framework for performing various operations. In thrust, the sorting library makes use of an optimum parallel implementation of radix sort [18].

Once the sorting of conjugate is over, the distance between each weighted sequence and weighted pattern is computed. The sequential distance calculation method between possible sequence and pattern is given in definition 9. During our study on distance calculation algorithms using extended burrow wheeler transform. We found that there is very less number of inter communication required between the distance calculation of each possible sequences and patterns. So, we parallelize this operation by using one thread of each block for distance calculation between each possible sequence and patterns generated from weighted sequence and pattern. Therefore the total numbers of threads running in parallel are able to accurately calculate the distances.

7. RESULT AND DISCUSSION

We evaluate the performance of the implementation of the algorithm for weighted sequence comparison using extended burrow wheeler transform on parallel processing architectures that are CUDA without thrust library and CUDA with thrust library used for sorting the conjugate of sequences. Our experimental setup were consists of a personal computer and CUDA cards having the following configurations: Intel Pentium(R) 4 CPU, 2.99 GHz, with 2.49 GB RAM running and CUDA card of NVIDIA Quadro FX 3700 graphics card (512MB global memory, 16KB shared memory, 1.2 GH_Z clock), Compiling environment used is Microsoft Visual Studio 2005, enabled with NVCC, available through CUDA 3.0 SDK

We designed an experimental scheme to measure the execution time of the algorithm for various lengths of weighted sequences that are generated randomly with threshold value are θ=0.2.

The biological weighted sequences are generated randomly. In the parallel implementation of weighted sequence comparison algorithm using extended burrow wheeler transform. We found that Sorting is the main performance improvement bottle-neck. Since the result shown in “Table 2 “reveals that sorting on uni-processor is significantly better then compute unified device architectures. We use bitonic sort for our implementation on NVIDIA multicore architecture that is compute unified device architecture. The main reasons for such performance degradation of bitonic sorting performs on compute unified device architecture are given below:

- In the device of Compute unified device architecture, each thread is assigned to handle two ptr_d[ ]* that points two element of array that need to sort. In accessing these two elements, the threads create an un-coalesced gather operation from the global memory of the device, thereby, it enormously decrease capable memory bandwidth and increasing the access time by a few hundred clock cycles.
- Once the values of ptr_d[i] and ptr_d[i+1] are obtained by a thread, then the characters pointed to by these two pointers are accessed through a global-memory access. Since this gather operation is more scattered than the previous operation, there exists a further inefficiency in the bandwidth utilization, thereby costing a few hundred more clock cycles for the operation.
- Now as the characters from the sequences are read and compared, then appropriate changes to their positions' is required and this change is reflected onto ptr_d[ ] by making a global-

write. Because these writes too are scattered in nature, no attempt are made to utilize the full-bandwidth.

In addition to this reasons, the number of kernel calls generated by the CPU for bitonic sort too high and it is $O(\text{tot\_length}^*)$. In a typical run, $\text{tot\_length} \approx 10^6$. Hence, the CPU overhead of launching these many kernels is also high.

After finding above, we conceived an idea to optimize lexicographic sorting on CUDA by using thrust library. Thrust is nothing but a library which provides an STL interface to coding in CUDA, along with a user-friendly library data structures. It also provides highly-optimized implementation of standard algorithms, such as sorting and searching. Thrust does not provide any ready-made implementation of lexicographic sorting.

In order to achieve this, an intermediate class had to be written, the objects of which imitated the list of solid words. The objects were then passed to the library defined sorting functions, keeping in regard the syntax and data-structure requirements of these functions. This helped achieve a significant improvement in our performance which is shown in “Table 3”. The library sort functions use the ideas discussed in [18]. The executions timed against the CPU performance along with the user-defined class are tabulated below and it is observed from “Table 4” that CUDA with thrust library on average outperforms the CPU through a 2.9 X times. However, the total time is not only the sum of the times taken to sort the conjugates and calculate the distances between them, but also it includes the card’s overhead of moving data to and from the main memory. In the similar manner “Table 5” shows CUDA with thrust library on average out performs CUDA without thrust library through 56.3X times. Moreover, “Table 6” shows the distance score between varies length of weighted sequence and weighted pattern. This score helps to determine the weighted sequence which is closest to weighted pattern. The pictorial representation of above mentioned results is also shown in (Figures. 7-9).

Table 2. Comparisons of execution time taken by performing sorting on CPU and CUDA for threshold α =0 .2

Number of positions in weighted sequences	Number of position in weighted pattern	Time taken to perform Bitonic Sort (in s)	
		CUDA	CPU
50	6	0.235	0.2500
60	6	0.341	0.3750
70	6	0.473	0.3280
80	6	3.226	2.188
90	6	27.148	25.172
100	6	171.97	120.468
110	6	235.184	155.690
120	6	314.291	199.469
130	6	411.521	269.570
140	6	513.339	298.2797
150	6	639.562	384.219
160	6	782.565	421.141
170	6	904.870	512.000
180	6	1600.89	598.370
190	6	1242.66	678.969

200	6	1433.84	729.078
210	6	1648.09	883.219
220	6	1882.68	974.359
230	6	2134.47	1043.281
240	6	2422.18	1179.797
250	6	2442.58	1310.688
260	6	4086.38	1452.141

Table. 3 A comparison of execution time taken by performing sorting on CPU and CUDA using thrust library for threshold α =0 .2

Number of positions in weighted sequences	Number of position in weighted pattern	Time taken to perform Bitonic Sort (in s)	
		CUDA	CPU
50	6	0.016	0.2500
60	6	0.016	0.3750
70	6	0.031	0.3280
80	6	0.046	2.188
90	6	0.140	25.172
100	6	0.360	120.46
110	6	0.422	155.69
120	6	0.531	199.46
130	6	0.593	269.57
140	6	0.719	298.27
150	6	0.812	384.21
160	6	0.921	421.14
170	6	1.079	512.00
180	6	1.265	598.37
190	6	1.375	678.96
200	6	1.531	729.07
210	6	1.688	883.21
220	6	1.906	974.35
230	6	2.047	1043.2
240	6	2.250	1179.7
250	6	2.422	1310.6
260	6	2.703	1452.1

Table. 4 A comparison of execution time taken by performing sorting, distance calculation and total time including communication cost on CPU and CUDA using thrust library for threshold α =0 .2

Number of positions in weighted sequences	Number of position in weighted pattern	CUDA with thrust library			CPU		
		Sorting time in milliseconds	Distance calculation time in milliseconds	Total Time taken by algorithm in milliseconds	Sorting time in milliseconds	Distance Calculation time in milliseconds	Total Time taken in milliseconds
50	6	0.01	0.01	0.594	.0.25	0.4	0.73
60	6	0.01	0.01	0.625	0.37	0.03	0.4
70	6	0.03	0.01	0.735	0.32	0.4	0.8
80	6	0.04	0.07	1.562	2.18	0.3	2.6
90	6	0.14	0.77	4.016	25.1	2.7	28.1
100	6	0.36	5.50	13.07	120.4	23.8	144.5
110	6	0.42	6.00	14.82	155.6	25.7	181.6
120	6	0.53	6.56	15.92	199.4	28.3	228.1
130	6	0.59	7.09	16.65	269.5	30.5	300.3
140	6	0.71	7.62	18.20	298.2	33.0	331.7
150	6	0.81	8.18	19.36	384.2	35.4	420.0
160	6	0.92	8.79	20.71	421.1	37.8	459.4
170	6	1.07	9.21	22.0	512.0	40.3	552.8
180	6	1.26	9.78	23.48	598.3	42	640.9
190	6	1.37	10.3	25.67	678.9	44.9	724.4
200	6	1.53	12.95	26.50	729.0	47.2	776.9
210	6	1.68	11.45	28.00	883.2	49.6	933.8
220	6	1.90	11.97	29.35	974.3	51.9	1027.5
230	6	2.04	12.57	30.93	1043	53.8	1097.8
240	6	2.25	13.09	32.67	1179	57.1	1237.6
250	6	2.42	13.61	33.62	1310	59.5	1371.9
260	6	2.70	14.14	35.20	1452	61.5	1514.7

Table. 5 A comparison of execution time taken by performing sorting, distance calculation and total time including communication cost on CUDA using Thrust Library and CUDA without using thrust library for threshold α =0 .2

Number of positions in weighted sequences	Number of position in weighted pattern	CUDA with Thrust Library			CUDA without Thrust Library		
		Sorting time in milliseconds	Distance calculation time in milliseconds	Total Time taken im millisecondsl	Sorting time in milliseconds	Distance calculation time in milliseconds	Total Time taken im milliseconds
50	6	0.016	0.016	0.594	0.235	0.016	0.4
60	6	0.016	0.015	0.625	0.341	0.015	0.4
70	6	0.031	0.016	0.735	0.473	0.016	0.6

80	6	0.046	0.078	1.562	3.226	0.078	3.4
90	6	0.140	0.772	4.016	27.148	0.772	28.2
100	6	0.360	5.50	13.07	171.97	5.50	177.8
110	6	0.422	6.00	14.82	235.184	6.00	241.5
120	6	0.531	6.563	15.92	314.291	6.563	321.3
130	6	0.593	7.093	16.65	411.521	7.093	419
140	6	0.71	7.625	18.20	513.339	7.625	521.5
150	6	0.81	8.188	19.36	639.562	8.188	648.2
160	6	0.92	8.797	20.71	782.565	8.797	791.9
170	6	1.07	9.219	22.0	909.870	9.219	919.6
180	6	1.26	9.783	23.48	1600.89	9.783	1611.3
190	6	1.37	10.34	25.67	1242.66	10.34	1253.6
200	6	1.53	2.953	26.50	1433.84	2.953	1437.5
210	6	1.68	11.45	28.00	1648.09	11.45	1660.5
220	6	1.90	11.97	29.35	1882.68	11.97	1896.0
230	6	2.04	12.57	30.93	2134.47	12.57	2148.0
240	6	2.25	13.09	32.67	2422.18	13.09	2436.8
250	6	2.42	13.61	33.62	2442.58	13.61	2458.6
260	6	2.70	14.14	35.20	4086.38	14.14	4101.7

Table 6. shows the distance between each sequence generated from weighted sequence with net probability of occurrence greater then threshold value = 1/5 to the pattern generated from weighted pattern with net probability of occurrence is greater than threshold value = 1/5

weighted sequence	weighted pattern	Average distance between weighted sequences with weighted pattern
weighted sequence 1	weighted pattern	45
weighted sequence 2	weighted pattern	55.4
weighted sequence 3	weighted pattern	65.4
weighted sequence 4	weighted pattern	75.5
Weighted sequence 5	weighted pattern	85.9
Weighted sequence 6	weighted pattern	94.5
Weighted sequence 7	weighted pattern	104.5
Weighted sequence 8	weighted pattern	114.4
Weighted sequence 9	weighted pattern	124.8
Weighted sequence 10	weighted pattern	134.4

Weighted sequence 11	weighted pattern	144.4
Weighted Sequence 12	weighted pattern	154.4
Weighted sequence 13	weighted pattern	164.4
Weighted sequence 14	weighted pattern	174.4
Weighted sequence 15	weighted pattern	184.4
Weighted sequence 16	weighted pattern	194.4
Weighted sequence 17	weighted pattern	204.4
Weighted sequence 18	weighted pattern	214.4
Weighted sequence 19	weighted pattern	224.4
Weighted sequence 20	weighted pattern	234.4
Weighted sequence 21	weighted pattern	244.4
Weighted sequence 22	weighted pattern	254.4

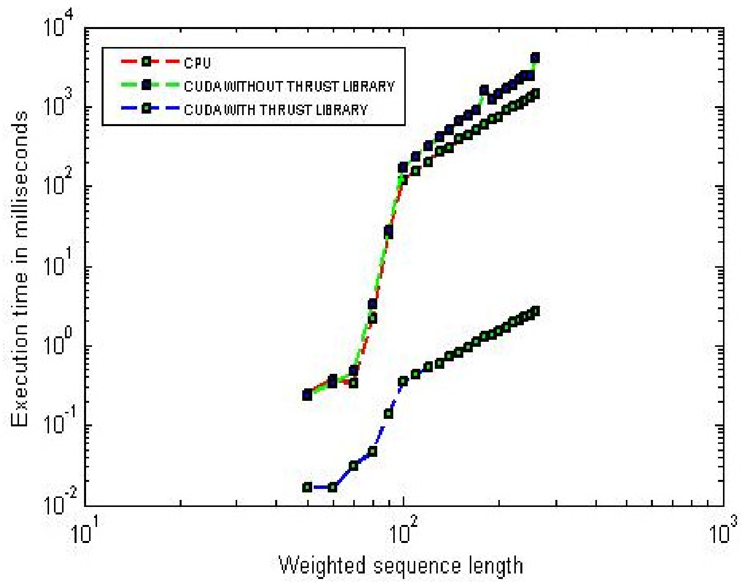

Figure 7. A comparision of execution time of sorting performed on CPU, CUDA without thrust library, and CUDA with thrust library

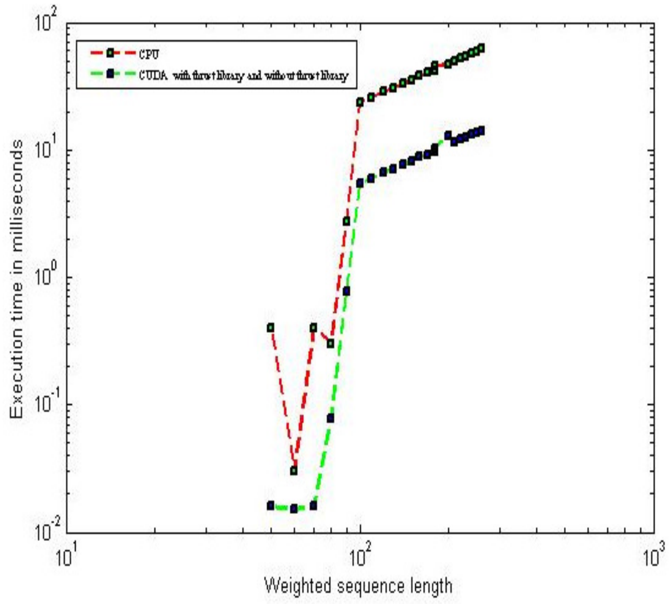

Figure 8. A comparison of execution time of distance calculation performed on on CPU, CUDA without thrust library, and CUDA with thrust library

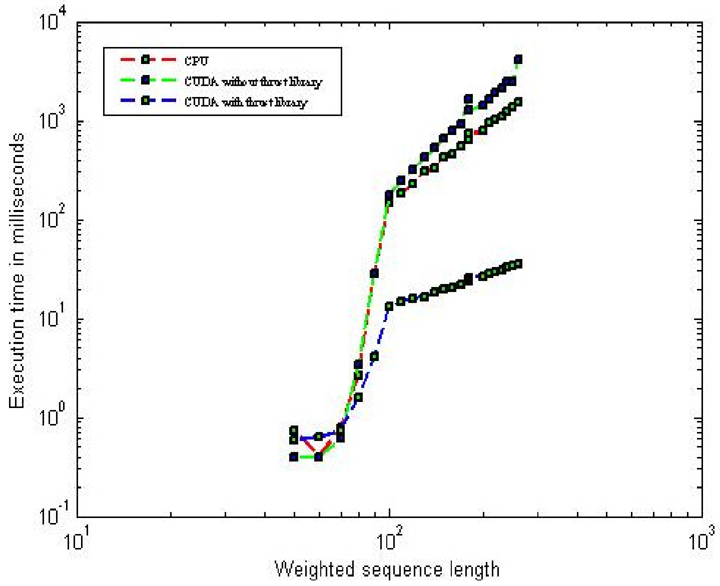

Figure 9. A comparison of total execution time of algorithm on CPU, CUDA without thrust library, and CUDA with thrust library

8. CONCLUSION

We have presented the implementation of extended burrow wheeler transform based weighted *s*equence comparison algorithm on many core NVIDIA compute unified device architecture (CUDA). Our experimental results shows that our parallel extended burrow wheeler transform based weighted sequence algorithm with thrust library is the fastest published weighted sequence comparison algorithm for modern many core CUDA processors and is up to 2.9X times more efficient than the techniques that maps weighted sequence comparison onto the CPU. In addition to being the only and fastest CUDA weighted sequence comparison techniques, it is also faster than our previous extended burrow wheeler transform based weighted sequence comparison algorithm implemented on CUDA without using thrust library. We are able to achieve this algorithmic efficiency by deploying as much as our working the fast on chip memory provided by the NVIDIA CUDA architecture and by exposing as much as possible fine grained parallelism, in order to take the advantage of the thousand of parallel threads supported by this architecture. In general we also believe that once we make transition towards fine grained parallelism of manycore chips, the structure of algorithm is also moves towards data –parallel structure. The implicit cached or explicit managed memory spaces also play important for improving efficiency on modern processors, therefore we believe that techniques that we apply on CUDA will also applicable for the implementation of this algorithm on other manycore processors and distributed computing like cell broadband engine, cluster computing, cloud computing . There are obviously a number of possible directions for future work in weighted sequence comparison problem. For instance, in case of weighted sequence greater 300 positions. An efficient algorithm might formulate somewhat different efficiency trade-offs than ours.

References

[1] D. Gusfield, “Algorithm on strings, tree, and sequences: computer science and computational biology”, Cambridge University Press, New York, NY, 1997.

[2] P. Baldi, and S. Brunak, “The machine learning approach“, Bioinformatics:, 2nd ed. MIT Press, Cambridge, MA, 2001.

[3] T. F. Smith,, and M. S. Waterman, “Identification of Common Molecular Subsequences”, J. .Mol .Biol., vol. 147, pp.195-197, 1981.

[4] C. Iliopoulos, K. Perdikuri, E. Theodoridis, A. Tsakalidis and K. Tsichlas, “Algorithms for extracting motifs from biological weighted sequences”, Journal of Discrete Algorithms, vol.5, pp. 229-242, 2007

[5] M. Burrows and D.J. Wheeler, “A block sorting data compression algorithm”, Technical Report, DIGITAL System Research Center, 1994

[6] M. Gessel and C. Reutenauer, “Counting permutations with given cycle structure and descent set”, J. Combin. Theory, vol. 64(2), pp. 189–215, 1993.

[7] M. Crochemore, J. Dȇsarmȇnien, and D. Perrin, “A note on the burrows-wheeler transformation”, Theoret. Comput. Sci, vol. 332(1-3), pp.567-572, February 2005.

[8] J. Kӓrkkӓinen, and P. Sanders, “Simple linear work suffix array construction”, In Proceedings of the 30th International Conference on Automata, Languages and Programming, pp.943-955 , 2003,

[9] M. Gokhale, J. Cohen, A. Yoo, W. M. Miller, A. Jacob, C. Ulmer, R. Pearce, Hardware Technologies for High Performance Data Intensive Computing, In Computer Magazine of IEEE Computer Society, vol. 41(4), pp.60–68, 2008.

[10] L. Ligowski, W. Rudnicki, An Efficient Implementation of Smith Waterman Algorithm on GPU, In IEEE International Symposium on Parallel and Distributed Processing, Rome, pp.1-8, 2009.

[11] F. Ergun, S. Muthukrishnan, and C. Sahinalp. “Comparing sequences with segment rearrangements”. Lecture Notes in Comput. Sci, In the proceedings of Foundations of Software Technology and Theoretical Computer Science, pp.183–194, 2003, Bombay, India.

[12] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, “The similarity metric”. IEEE transaction on Information Theory, vol. 12(5), pp.3250–3264, 2004.

[13] H.H. Out, and K. Sayood, " A new sequence distance measure for phylogenetic tree construction", Bioinformatics, vol.19(16), pp.2122–2130, 2003.

[14] S. Vinga, and J. Almeida, "Alignment-free sequence comparison – a review", Bioinformatics, vol.19 (4), pp.513–523, 2003.

[15] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino, "A new combinatorial approach to sequence comparison", Theoretical Computer Science, Lecture Notes in Computer Science, vol.3701, pp.348-359, 2005.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms", MIT Press, Second edition, Sept. 2001.

[17] J. Hoberock, and N. Bell, "Thrust: A Parallel Template Library", version 1.2, http://www.meganewtons.com/, 2009.

[18] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs", In Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, Italy, *23*-29 May 2009.

Binay Kumar Pandey received the B. Tech. (Information Technology) and M. Tech in (Bio-Informatics) degrees from the Institute of Engineering and Technology Lucknow, Maulana Azad National Institute of Technology Bhopal in 2005 and 2008 respectively. He was third topper during graduate studies and was awarded Prime Minister scholarship for meritorious ward of defence personnel for his excellent performance. After completing his M. Tech programme,
He proceeds towards to complete his PhD degree from Electronics and Computer Engg, Indian Institute of Technology Roorkee.