Parallelization of Composition Vector Method for Sequence Similarity Analysis

Manoj Gupta, Aniket Mittal, Rajdeep Niyogi, Manoj Mishra
1234 Department of Computer Science and Engineering, IIT Roorkee,
Roorkee, India - 247667

Abstract—Sequence comparison is an important task in the field of bioinformatics. Due to rapid accumulation of biological sequence data, scalability has become a bottleneck for researchers. In this paper we intend to solve this problem using parallelism on GPUs. GPUs provide the ability to achieve significant performance gains as compared to conventional CPUs due to their massive data parallel ability. Thus, we implement a Composition Vector method for sequence comparison on CUDA and compare its performance with sequential code. Our experiments show that a speedup of the order of 25 times is attainable due to massive data parallel components in the algorithm.

Keywords—Composition Vector Method, GPU, CUDA, Sequence Comparison.

1 Introduction

Biological sequences such as DNA have been studied extensively in the recent times, to extract critical biological information. Primarily, sequence comparison techniques are used to compare these sequences and figure out how similar or different one sequence is to another to formulate relationships between them thus extracting various properties from them, such as common ancestors, similar genes etc. Sequence comparison methods are commonly divided in two categories: alignment-based [16], [17] and alignment-free [4], [20]. Alignment-based methods mostly used dynamic programming to compare the sequences and generate similarity scores. The accuracy of this method is the main cause why this method is widely adopted. The composition vector (CV) is an alignment-free method [12] and is employed due to some advantages that it offers over alignment-based methods. Since every species has its own gene order and content, it is difficult to align two complete genome sequences using alignment-based methods. CV method has been used for phylogenetic analysis of complete genome sequences of bacteria, eukaryote, etc [5], [18], [19]. It is relatively easier to compute the distance matrix because it contains no scoring matrix, thus the computation can be easily parallelized [5].

In this paper, we exploit the fact the most computation in CV method is data parallel and can be parallelized using CUDA programming environment.

The CV method has the following four steps:

1) Construct the frequency vectors: Any string of length k in a sequence of length N is called a k-string. For a DNA sequence the possible values of the k-string can be 4k since at each position there are 4 possible values, {A, C, G, T}. Frequency vector contains the frequencies of each k-string in the sequence. For M sequences, there are M frequency vectors each of length 4k.

2) Construct the composition vectors: The expected frequency for each k-string u, denoted by q(u) is estimated. The composition vector for each sequence is given by [1]:

\[d_{Haa}(a, b) = \frac{1 - \cos \theta}{2}\]

where, 

\[\cos \theta = \frac{\langle u, b \rangle}{|u||b|}\]

For M sequences, there are M composition vectors each of length 4k.

3) Calculate the distance between each pair of sequence: The cosine angle used by Hao et al. [5, 14] is used in this paper for the computation of distance between composition vectors of different sequences.

4) Construct the phylogenetic trees: The distance matrix is then used to generate the phylogenetic tree using the Neighbour Joining method [21], [22] in the software MEGA [15].

As can be inferred from the above method, step 2 is the most important step in this method [18], [19], [20]. There are several estimation formulas for the estimation of the expected frequency q(u). In this paper, an optimized version of the method proposed by Yu et al. [13] has been parallelized. To the best of our knowledge this is the first attempt to parallelize a CV method for sequence comparison method using GPUs and discover the amount of performance improvement obtainable. Although CV method is comparatively less time consuming than other alignment-based methods [16], [17], the computation time increases as the number and size of sequences increases. Thus the goal of our parallelized version is to exploit the computational power of GPUs on CUDA. In recent years, graphics processing units (GPUs) have gained a lot of attention as a cost-effective means in parallel computing and have been widely used in bioinformatics. Some traditional algorithms, such as Smith-Waterman sequence alignment algorithm [7], RAxML [23], DP-based alignment algorithms [8] etc. have been implemented on GPUs and achieved significant speedups.

The paper is organized as follows. In Section 2 we describe the specific algorithm being implemented and the pseudo codes for the parallelized algorithm. Section 3 gives detail about the experimental setup with two datasets and the platform used to implement. An analysis of the results is performed in Section 4 followed by the conclusions made in Section 5 along with the future work.
2 Method

2.1 Frequency Vector

The first step in CV method is computation of frequency of each of the $4^k$ k-strings in each DNA sequence, thus generating frequency vectors. These are computed by the simple formula given in [12],

$$f(u) = \frac{n(u)}{(N - k + 1)} \quad (2)$$

where, $f(u)$ is frequency of the k-string $u$, $n(u)$ is the number of times the k-string $u$ occurs in the DNA sequence, and $N$ is the length of the DNA sequence.

As can be observed from equation (2), the calculation for each k-string is independent from the other. This calculation has been done in parallel for each k-string in parallel. Thus, there can be a maximum of $4^k$ threads running in parallel for $4^k$ k-strings.

2.2 Expected Frequency

For any k-string $u$, let us write it as $LwR$, where the characters "L" and "R" represent the first and the last nucleotides of $u$, respectively, and "w" represents the (k-2)string in the middle. The formula for the estimation of the expected frequency proposed by Yu et al. [13] is:

$$q(LwR) = \frac{f(L)f(wR)f(Lw)f(R)}{2} \quad (3)$$

After solving the optimization problem introduced by Chan et al. [3], the new formula for the estimation of the expected frequency is:

$$q(LwR) = \frac{1}{4\sigma} \left[ f(Lw) + f(L)\sum f(wl) \right] \ast \left[ f(wR) + f(R)\sum f(lw) \right] \quad (4)$$

where $I$ is the set of all nucleotides, i.e., $I = \{A, C, G, T\}$ and $\sigma$ is given by,

$$\sigma = \frac{1}{2} \left[ \sum f(lw) + \sum f(wl) \right]. \quad (5)$$

In order to calculate the expected frequency, we need the frequency vector for (k-1)-strings and l-strings.

Equation (4) can be parallelized in different ways, such as one thread for LwR for given L, w, R, or for each LwR for a given w and for all L, R. In the latter case, one thread computes $q(LwR)$ for 16 k-strings because both L and R have 4 possible values $\in I$. In this paper this method has been used to parallelize the algorithm. Thus there can be a maximum of $4^{k+2}$ threads running in parallel for $4^k$ k-strings.

2.3 CUDA Model

The method used in this paper simply divides the tasks in blocks and threads available in CUDA. The sizes of blocks and threads are chosen carefully so that no GPU cycle is wasted idle. As described above, the number of threads required is a power of 4, specifically $4^k$ and $4^{k+2}$. For values of $k \geq 5$ the number of threads is $4^k \geq 1024$ and for $k \geq 6$, it is $4^{k+2} \geq 256$. Typical k-values used for the CV method are $k = 6, 7, 8$.

The configuration of the GPU cards available had a maximum of 512 threads per block. This value dropped to 256 threads per block if the number of registers used per threads exceeded a limit, as specified by the CUDA Occupancy Calculator [10]. For the first step, the code could be written to allow 512 threads per block but for the second step, the maximum went down to 256. Thus for the first step we divided the calculation in multiples of 512, and for the second step in multiples of 256.

2.4 Some notations used

The number of sequences in the dataset are represented by $M$. The length of $i$th sequence is denoted by $N_i$. $vstring$ and $wstring$ are used to denote a (k-1)-string and (k-2)-string respectively. All the arrays used in the code are ordered alphabetically. $vstrings$ and $wstrings$ are used to denote an array of (k-1)-strings and (k-2)-strings respectively. Thus, for $k=7$, the $vstrings$ array will be $\{AAAAAA, AAAAAA ... TTTTTT, TTTTTTT\}$ and the $wstrings$ array will be $\{AAAAA, AAACAA ... TTTTG, TTTTT\}$. The notation, Arr$_{\alpha\beta}$ is used to simplify the pseudo-code. Here $s$ is a string and Arr$s$ means the value for that string $s$ in the array Arr. String $s$ will be some k-string and the array Arr is ordered alphabetically to retrieve the value of string $s$ instantaneously. For e.g., for $k=4$, the ordering is as follows: $AAAA = 0$, $AAAC = 1$, $AAAG = 2 ... TTTT = 255$. This conversion is easily done if we consider the string $s$ as a quinary number with $A=0$, $C=1$, $G=2$, $T=3$. Now a simple function converts this quinary number to decimal. Thus, $TTTT = 3333 = 3*64 + 3*16 + 3*4 + 3 = 255$. Thus Arr$[AAAA]$ means Arr$[0]$. In other words, Arr is the $0^{th}$ 4-string. The subscript $m$ in Arr$_{\alpha\beta}$ represents the $m^{th}$ subscript (starting at 0). For $k=4$, for each sequence there are 256 values, thus Arr$_{[AAAC]}$ converts to Arr$[3*256 + 1]$.

3 Pseudo Codes

3.1 Calculation of Frequency Vectors

The first step of CV method is computing frequencies of various k-strings in all the DNA sequences. These frequencies are stored in a vector for each sequence, thus producing M vectors. In other words we get a matrix of size $M \times 4^k$ for a specific $k=q$. We compute 3 such matrices for $q = k, (k-1)$ and 1. The formula used to compute the frequency for a k-string in a specific sequence is given by (2).

Fig. 1 is the kernel code for step 1. Fig. 1a first calculates the number of occurrences of each k-string in each sequence. The kernel code is invoked from the host as shown:

```c
noc_kernel <<<1, M>>> (noc, k, seq)
```

1. $m = threadIdx.x$
2. while ($i+k-1) < N$
3. $k_i = k\_index(seq_{[i]}, k)$
4. noc$_{[k\_i]} += 1$
5. $i++$
6. Endwhile

(a) Kernel for number of occurrences: noc_kernel
Our experiments show that these dimensions of the grid are slightly better than \( \langle \langle M, 1 \rangle \rangle \). noc variable is used to return the result computed. q is the current size of k-string. We use this to generate frequency vectors for \( k, k-1 \) (vfreq) and 1 (ifreq) length kstrings. For \( q = k \), we get kfreq which is used in step 3 of the CV method to calculate composition vector explained in (1). For \( q = (k-1) \) and (k-2) we get vfreq and wfreq respectively that are used in step 2 of the CV method, as explained in the following section. seq is an array of all sequences. k_index function returns the decimal value of the k-string that starts at seq\([i]\) (\( i^{th} \) position of m\textsuperscript{th} sequence) and is of length k. Thus, there are M threads and each thread computes the number of occurrences for all kstrings for a particular sequence.

Fig. 1b uses the values of number of occurrences (noc) computed in Fig. 1a and calculates the final value of frequency vectors. It is called from the host with number of blocks equal to M and number of threads per block equal to 512 as shown:

\[
\text{freq\_kernel } \langle \langle M, 512 \rangle \rangle \text{ (freq, noc, k, seq\_lens)}
\]

freq variable is used to return the value of the result. noc is the result of Fig. 1a. k is the current size of kstring. seq\_lens is an array containing lengths of all the sequences. In Fig. 1b, the maximum values of m is M and maximum value of i is 511. The size of freq array is Mx4\textsuperscript{3}. freq\([ks]\) stores the value of the frequency of k-string \( ks \) for the \( m^{th} \) sequence, i.e. \( m^{th} \) + decimal value of \( ks \). The while loop runs 4\textsuperscript{3}/512 times, for e.g., for \( k=6 \), it is called 8 times.

At the end of the call to kernel function, the freq array is copied to from device to host and we get a 2D array of all the frequency vectors for each sequence, i.e. a matrix of size Mx4\textsuperscript{3}.

3.2 Calculation of Sigma

Fig. 2 is the kernel code for the calculating \( \sigma \) with the formula shown in (5). The call to the kernel function is made from the host as shown:

\[
\text{\( \sigma \_kernel \rangle \rangle \rangle \langle \langle M, 512 \rangle \rangle \rangle \text{ (\( \sigma \), wstrings, vfreq)}
\]

where, ifreq\([A]\) is the frequency of the 1-string A, i.e. frequency of the nucleotide A, in \( m^{th} \) sequence and \( \sigma\_n\[A\] \) is the value of sigma for \( A^{th} \) wstring, in \( m^{th} \) sequence.

Fig. 3 is the kernel code for the calculations of step 2 shown in equation (4). It is called from the host with number of blocks equal to M and number of threads per block equal to 256 as shown:

\[
\text{ \( k3 \rangle \rangle \rangle \langle \langle M, 256 \rangle \rangle \rangle \text{ (expfreq, wstrings, vfreq, ifreq, \( \sigma \))
}\]

The code itself is easy to understand. Since the size of \( \sigma \) is \( 4^{k-2} \), the while loop runs \( 4^{k-2}/512 \) times. At the end of this step, we get a 2D array of all the \( \sigma \) vectors for each sequence, i.e. a matrix of size Mx\( 4^{k-2} \).
The maximum values of \( m \) is \( M \) and maximum value of \( i \) is 255. Thus the size of \( \text{expfreq} \) array is \( M \times 4^k \). The \texttt{while} loop runs \( 4^{k-2}/256 \) times, for e.g., for \( k=8 \), it iterates 16 times. Lines 8 to 10 calculate \( \sum_i f(wl) \) in equation 4 and lines 11 and 12 calculate \( f(L) \sum_i f(wl) \) and \( f(L) \sum_i f(wl) + f(LwR) \) respectively. Similarly, lines 13 to 17 calculate \( f(wR) + f(R) \sum_i f(wl) \). Line 19 calculates the final value of \( q(LwR) \) in equation 4. At the end of this step, we get a 2D array of all the expected frequency vectors for each sequence, i.e., a matrix of size \( M \times 4^k \).

4 Experimental Setup

We compare the sequential version of the CV method to parallel version in our experiments. Both the codes are written in C language and the parallel version uses CUDA [9] programming environment for parallelization on GPUs.

4.1 Datasets

The experiments were performed on two datasets of different sizes. The first dataset (DS-I) is 18S rRNA sequence of human ribosomal DNA complete repeating unit (GenBank: U13369.1) that contains 34 DNA sequences, described in Table 1 [2]. The second dataset (DS-II) is 0.9-kb mtDNA fragments of 12 species of primates, described in Table 2 [6]. The DNA sequences for each species was obtained from NCBI database.

4.2 Platform

We have implemented our sequential version on an Intel Core i5-2430M CPU 2.40 GHz 2.40 GHz 4.00 GB RAM. The method has been parallelized on NVIDIA GeForce GT 525M.

<table>
<thead>
<tr>
<th>Species</th>
<th>ID/accession</th>
<th>Length (bp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haloarcula sp.</td>
<td>U68539</td>
<td>1470</td>
</tr>
<tr>
<td>Haloarcula sp.</td>
<td>U68537</td>
<td>1469</td>
</tr>
<tr>
<td>Halobacterium salinarum</td>
<td>U68538</td>
<td>1471</td>
</tr>
<tr>
<td>Haloferax volcanii</td>
<td>U68540</td>
<td>1471</td>
</tr>
<tr>
<td>Halococcus hispanica</td>
<td>U68541</td>
<td>1469</td>
</tr>
<tr>
<td>Haloarcula japonica</td>
<td>D28872</td>
<td>1424</td>
</tr>
<tr>
<td>Haloarcula marismortui (rrnA gene)</td>
<td>X61688</td>
<td>1472</td>
</tr>
<tr>
<td>Haloarcula marismortui (rrnB gene)</td>
<td>X61689</td>
<td>1472</td>
</tr>
<tr>
<td>“Haloarcula sinaiensis” (major gene)</td>
<td>D14130</td>
<td>1470</td>
</tr>
<tr>
<td>“Haloarcula sinaiensis” (minor gene)</td>
<td>D14129</td>
<td>1471</td>
</tr>
<tr>
<td>Haloarcula vallismorti</td>
<td>U17593</td>
<td>1471</td>
</tr>
<tr>
<td>Halobacterium salinarum (halobium)</td>
<td>M38280</td>
<td>1473</td>
</tr>
<tr>
<td>Halobacterium trapanicum</td>
<td>D14125</td>
<td>1472</td>
</tr>
<tr>
<td>Halorubrum coriense</td>
<td>L00922</td>
<td>1469</td>
</tr>
<tr>
<td>Halobacterium sp.</td>
<td>D14127</td>
<td>1471</td>
</tr>
<tr>
<td>Halobaculum gomorrense</td>
<td>L37444</td>
<td>1474</td>
</tr>
<tr>
<td>Halococcus morrhuae</td>
<td>D11106</td>
<td>1474</td>
</tr>
<tr>
<td>Halococcus morrhuae</td>
<td>X00662</td>
<td>1475</td>
</tr>
<tr>
<td>Haloferax denitrificans</td>
<td>D14128</td>
<td>1469</td>
</tr>
</tbody>
</table>

Table 2. DS-II: 0.9-kb mtDNA sequences [6].

<table>
<thead>
<tr>
<th>Species</th>
<th>ID/accession</th>
<th>Length (bp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macaca fascicular</td>
<td>M22653</td>
<td>896</td>
</tr>
<tr>
<td>Macaca fuscata</td>
<td>M22651</td>
<td>896</td>
</tr>
<tr>
<td>Macaca mulatta</td>
<td>M22650</td>
<td>896</td>
</tr>
<tr>
<td>Macaca sylvanus</td>
<td>M22654</td>
<td>896</td>
</tr>
<tr>
<td>Saimiri sciureus</td>
<td>M22655</td>
<td>893</td>
</tr>
<tr>
<td>chimpanzee</td>
<td>V00672</td>
<td>896</td>
</tr>
<tr>
<td>Lemur catta</td>
<td>M22657</td>
<td>895</td>
</tr>
<tr>
<td>gorilla</td>
<td>V00658</td>
<td>896</td>
</tr>
<tr>
<td>hylobates</td>
<td>V00659</td>
<td>896</td>
</tr>
<tr>
<td>Orangutan</td>
<td>V00675</td>
<td>895</td>
</tr>
<tr>
<td>Tarsius syrichta</td>
<td>M22656</td>
<td>895</td>
</tr>
<tr>
<td>human</td>
<td>L00016</td>
<td>896</td>
</tr>
</tbody>
</table>

5 Results and Discussion

5.1 Time taken by each step of CV method

Table 3 shows the time taken by the sequential code for various functions for DS-I and DS-II when the value of \( k \) is 8.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ms) DS-I</th>
<th>Time (ms) DS-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. k frequency vectors</td>
<td>2143</td>
<td>413</td>
</tr>
<tr>
<td>2. Sigma vectors</td>
<td>220</td>
<td>93</td>
</tr>
<tr>
<td>Expected frequency vectors</td>
<td>1923</td>
<td>483</td>
</tr>
<tr>
<td>3. Composition vectors</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>4. Distance vectors</td>
<td>150</td>
<td>23</td>
</tr>
</tbody>
</table>

Row 1 in Table 3 is the time taken to calculate the frequency vectors for all sequences mentioned in Table 1 and Table 2. This is the same as the time taken to compute step 1 in CV method. Row 2, 3 and 4 describe the time taken to
compute step 2 of CV method. It has been observed from the Table 3 that maximum time is consumed in the calculation of the functions in row 1, 2 and 3. Thus these three functions have been parallelized to achieve maximum performance improvement.

5.2 Comparison of Sequential and Parallel algorithms

From Fig. 4 and Fig. 5 we observed that there is significant reduction in computation time while parallelizing step 1 of CV method for DS-I (Table 1) and DS-II (Table 2) respectively. The pseudo-code for step 1 is given in Fig. 1. Similarly, the computation time is greatly reduced in parallelizing the step 2 of CV method as shown in Fig. 6 and Fig. 7 for the same datasets shown by Table 1 and Table 2 respectively. The times shown in Fig. 6 and Fig. 7 are the summation of the time taken by both the pseudo-codes, Fig. 2 and Fig. 3. Fig. 8 shows the total speedup obtained after parallelization. Speedup is calculated by summing the times taken in step 1 and 2 from Fig. 4 with Fig. 6, and Fig. 5 with Fig. 7. Then the ratio of sequential time versus parallel time is calculated. As, it is evident from these figures, the speedup in step 1 is much more than that in step 2 of CV method. The parallel code for step 1 is about 20-27 times faster, whereas it is about 4-6 times faster for step 2. This is mainly due to the fact that the kernel functions in Fig. 2 and Fig. 3 are much larger than those in Fig. 1. Thus, each thread on the GPU is doing larger number of computations and taking longer to finish them, hence reducing the performance.

Another useful observation can be made from Fig. 8. It shows how the speedup improves as the value of k increases. As k increases the size of all the vectors (kstring, vstring, wstring and all frequency vectors) increases exponentially, thus increasing the scope of parallelization. This in turn increases the performance speedup manifolds.
6 Conclusion and Future Work

Composition vector method is widely used alignment-free method used for similarity analysis among the sequences. The first two steps of the method is highly computational intensive and hence parallelization will significantly reduce the computation time. Therefore, a parallelized implementation of Composition Vector method for sequence comparison has been performed. The result shows significant performance improvement as compared to its sequential counterpart. The computationally intensive phases of the algorithm were identified and their data parallel components exploited to parallelize them.

The approach used in this paper can be applied to different types of estimation methods [1] and performance improvement by exploiting the data parallel components can be obtained.

Although there have been significant improvements obtained in parallelizing the CV method but in future we can further improve by appropriately selecting CUDA parameters such as the optimal kernel code and number of blocks and threads.

7 Conflict of Interests

The Authors declare that there is no conflict of interest.

8 References


Fig. 8 Speedup comparison between parallel and sequential with total time of step 1 and 2.