Using GPUs to Speed-Up Levenshtein Edit Distance Computation

Khaled Balhaf, Mohammed A. Shehab, Wala'a T. Al-Sarayrah,
Mahmoud Al-Ayyoub, Mohammed Al-Saleh and Yaser Jararweh
Jordan University of Science and Technology, Irbid, Jordan
Emails: {khbalhaf14, mashehab12, wtalsarayrah14}@cit.just.edu.jo, {maalshbool, misaleh, yijararweh}@just.edu.jo

***Abstract*—Sequence comparison problems such as sequence alignment and approximate string matching are part of the fundamental problems in many fields such as natural language processing, data mining and bioinformatics. However, the algorithms proposed to address these problems suffer from high computational complexities prohibiting them from being widely used in practical large-scale settings. Many researchers used parallel programming to reduce the execution time of these algorithms. In this paper, we follow this approach and use the parallelism capabilities of the Graphics Processing Unit (GPU) to accelerate one of the most common algorithms to compute the edit distance between two strings, which is known as the Levenshtein distance. To take full advantage of the large number of cores in a GPU, we employ a diagonal-based tracing technique which results in even greater improvements in terms of the running time. In fact, our CUDA implementation of the Levenshtein algorithm is about 11X faster than the sequential implementation. This is achieved without affecting the accuracy.**

I. Introduction

The big revolution in this information age has given rise to many interdisciplinary fields. One major example is the field of bioinformatics which benefits from the computational methods of computer science to address the complex problems in biology, especially the ones related to genomics and proteomics. Coupled with the rapid improvement in the hardware and experimental instruments in the past few decades, this field witnessed jumps in its development and maturity. One of the feats of this field is the success of the Human Genome Project (HGP)[1] which aims at identifying the structure and contents of the human DNA from both a physical and functional standpoint [1].

The HGP as well as many other problems in bioinformatics are basically dealing with very long sequences and strings (of DNA nucleotides, amino acids, etc.). One of the objectives of these problems is to perform different kinds of sequence comparisons such as sequence alignment and approximate string matching. To address these problems, bioinformatics researchers benefited from the rich literature of sequence comparison algorithms in computer science fields such as fast searching for name or strings similarity [2]. For example, one measure of computing string similarity is the Levenshtein distance which has been extensively studied in the field of Natural Language Processing (NLP) for spelling correction [3]. The same measure can be used to get an indication of the functional similarity between different DNA fragments depending on their content similarity [4].

The proposed algorithm for computing the Levenshtein distance is a very neat one. However, it suffers from high computational complexity prohibiting it from being widely used in certain large-scale settings such as that of DNA fragments which are millions of nucleotides (characters) in length. This is where the computational methods and the advanced optimization techniques in computer science can come in handy [5], [6].

Many researchers proposed to use parallel programming to reduce the execution time of similar algorithms. One example is the Smith-Waterman (SW) algorithm for the local alignment problem [7], [8]. Parallel programming is a useful technique in High Performance Computing (HPC) and it helps to reduce the execution time of many algorithms [9]. One of the hardware supporting highly parallel execution of programs is the Graphics Processing Unit (GPU) [10], [11]. Due to its large number of cores, the GPU supports a number of threads far beyond those supported by the Central Processing Unit (CPU). It can run more than 512 threads at same time while the CPU can run at most 8 threads at the same time [12]. Created by NVIDIA, Compute Unified Device Architecture (CUDA) is a programming framework or a toolkit that allows C/C++ developers to have a parallel implementation that utilizes the GPU capabilities for general purpose processing [13].

In this paper, we follow this approach and use the parallelism capabilities of the GPU to accelerate the computation of the Levenshtein distance between two strings. To take full advantage of the large number of cores in a GPU, we employ a diagonal-based tracing technique which results in even greater improvements in terms of the running time. To the best of our knowledge, few previous papers have presented any attempts to accelerate the computation of the Levenshtein distance using GPU. The closest to ours is the work of Siriwardena and Ranasinghe [14] in which the authors implemented the Needleman-Wunsch (NW) algorithm on CUDA. Compared to the fill step on the CPU, they achieved 2 times speed-up by using the GPU global memory with slow speed of accessing. In the next level, they used fast memory access and achieved up to 4.2 times speed-up compared to the CPU implementation.

The structure of paper as the following. On next section, we represent some related work of accelerate the string

[1]https://en.wikipedia.org/wiki/Human_Genome_Project

matching algorithms. After that on third section, we illustrate our methodology. Then, we display the results of this paper. Finally, we deduce all the work in conclusion and suggest the future work.

II. Related Works

The Levenshtein edit distance is one of the most used methods to calculate the similarity/distance between two strings, A and B of lengths n and m, respectively [15]. It is defined as the minimum number of single-character edit operations (such as insertions, deletions or substitutions) that are needed to convert A into B. The basic dynamic programming algorithm to compute the Levenshtein edit distance runs in $O(mn)$ time, which is inefficient considering the large-scale settings in which it is commonly used. It shoud be noted that several extension of the Levenshtein edit distance exists such as the NW algorithms, which deals with the affine gap penalty issue and the Smith-Waterman (SW), which deals with the local alignment problem [16], [17].

There have been several attempts to improve these algorithm. One approach is to employ bit-parallelism to find the fastest approximate string matching. Such as Myers [18] who achieved an optimal algorithm with $O(nm/w)$ complexity, where w is the word size in the machine. Other parallel approaches are discussed in the following paragraph.

In [19] Xu et al. accelerated string matching algorithm by converting Prasad et al. [20] extension of the BPR algorithm [21] for multiple patterns to CUDA implementation. This algorithm is used to detect similar patterns between two strings. This code is also used for DNA string matching in bioinformatics. They tested the sequential version with pure parallel implementation to get the same outputs with faster execution time. For the experiments, they used a machine with a Intel Core i3 CPU, 2GB RAM and a GeForce 310M GPU card with 512MB as GPU memory. The software for this paper is Windows 7 with 32-bit architecture and Microsoft visual studio 2008. The speed up gain in this research is about 28X faster than sequential version.

Parallel programming using CUDA is very efficient to accelerate string matching algorithms as shown in many recent studies [9]. As an example of the works that exploited parallel programming to speed-up string alignment is the work of Ligowski and Rudnicki [7]. Specifically, the authors show to gain significant spped up of the SW algorithm using the parallel capabilities of the GPU. Compared with an earlier implementation of the SW algorithm on GPU, the authors showed that their implementation achieved 3.5 times higher per core. They implemented the SW algorithm on Sony PlayStation 3 (PS3) environment. Both PS3 and CPU use one byte for representing integers. On the other hand, GPU uses a full integer representation. As a result, using PS3 and CPU implementation is better when non similar sequences need to be found in large databases and, for similar sequences, the GPU implementation is better [7].

In [14], the authors implemented NW algorithm using CUDA-GPU to accelerate the computations. Compared to the fill step on the CPU they achieved 2 times speed-up by using the GPU global memory with slow speed of accessing. Then, they used fast memory access and achieved up to 4.2 times compared to the CPU implementation.

For the SW algorithm, the authors of [22] investigated two parallel techniques in order to speed-up the computations on a GPU. These techniques are: wave-front and streaming. The wave-front technique suffers from a particular limitation because it cannot deal with long sequences due to GPU physical memory limit. In streaming technique, long sequences can be processed but with an overhead because of data transmission between CPU and GPU. They implemented a new algorithm (tile-based parallel Viterbi algorithm) that improves the GPU performance. Long sequences are divided into small pieces and every pair of pieces can be handled by the GPU memory. Based on the experiments, their algorithm outperforms both of the wave-front and the streaming techniques.

In [23], the authors implemented ClustalW tool using GPU in order to reduce the run time of computations. They calculated the number of matches between two sequences by using a new recurrence relation which gives the required results ten times faster. Moreover, their design is scalable since it partitions pairs of sequence comparisons to several GPUs located on the same PC or connected via a network.

III. Methodology

In this section, we represent the main approach to accelerate the Levenshtein edit distance computation. This improvement is done by two main steps. The first step is reducing the dependence between elements. Then the second step is done by running this new implantation in parallel using CUDA implementation.

A. Reduce data dependency

This section illustrates how to reduce dependency between matrix cells in the Levenshtein distance algorithm. As shown in Algorithm 1, the sequential implementation of the Levenshtein distance starts by setting the first row and first column of matrix with initial values from the interval $[1 - N]$, where N is the length of strings under consideration. After that, the algorithm continuously fills up all matrix cells to compute the distance between the two strings of DNA sequence.

The value of each cell $[i, j]$ in the matrix H is computed based on the values of three adjacent cells: upper cell, left cell and upper left cell as follows.

$$H_{i,j} = \min \begin{cases} H_{i-1,j-1} + Score \\ H_{i,j-1} + 1 \\ H_{i-1,j} + 1 \end{cases} \tag{1}$$

where the score value in our implementation is zero if the character of the first sequence matches the character of the second sequence; otherwise, the score value is one. This way of computation introduces dependencies as the ones shown in Figure 1.

Based on the dependency problem mentioned in the previous paragraph, the algorithm cannot calculate the value of a

Algorithm 1 Sequential Implementation

```
procedure LEVENSHTEIN($Str1, Str2$,N)
    Initialize first row and first column from 1 to N
    for <Row = 0 to N -1> do
        for <Column = 0 to N -1> do
            if $Str1[Row-1] == Str2[Column-1]$ then
                Score = 0
            else
                Score = 1
            end if
            Calculate Distance $H_{Row,Column}$ as Equation 1
        end for
    end for
end procedure
```

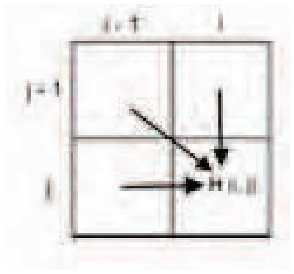

Fig. 1. Data Dependency Problem [14]

certain cell before computing the values of all cells above and to the left of it. This limits the ability to exploit parallelism to speed-up the computation of the edit distance.

As stated by Siriwardena and Ranasinghe [14], one way to circumvent this problem is to compute cells values in a "diagonal way." As shown in Figure 2, if we compute the values of cells in a diagonal way, the dependencies between cells can be decreased. For example, after calculating cell with index [0, 0], the values of the cells with indexes [1, 0] and [0, 1] can be computed in parallel without any problem. This method gives us a chance to compute more than one cell in each iteration in parallel. Figure 2 shows the diagonal technique and the number of cells that will run in parallel in each iteration. Algorithm 2 shows all steps for this diagonal approach.

Algorithm 2 Diagonal Implementation

```
procedure LEVENSHTEIN($Str1, Str2$,N)
    Initialize first row and first column from 1 to N
    for <slice = 0 to N*2-1> do
        if $slice < N$ then
            $Z = 0$
        else
            $Z = sliceN + 1$
        end if
        for <j= Z to slice> do
            $Row = slice$
            $Column = slice - j$
            if $Str1[Row-1] == Str1[Column-1]$ then
                Score = 0
            else
                Score = 1
            end if
            Calculate Distance $H_{Row,Column}$ as Equation 1
        end for
    end for
end procedure
```

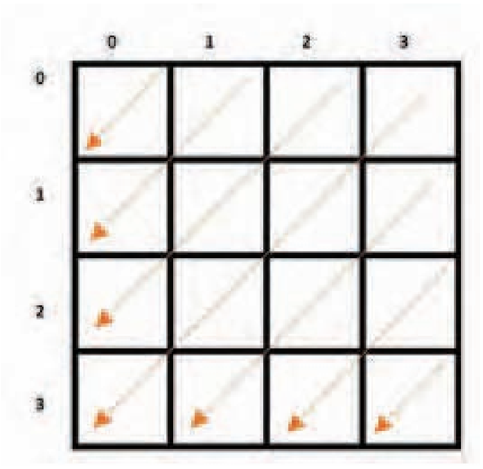

Fig. 2. Diagonal Technique in matrix

B. CUDA implementation

This section represents the parallel implementation of CUDA code. Parallel programming has two main techniques for implementation. The first one is the pure parallel code, where all parts of the code are run in the GPU side. The second one is the hybrid parallel code, where some parts of the code (i.e., some functions) are run on the GPU side and the rest are run on the CPU side.

Our proposed implementation is a pure parallel one because the Levenshtein distance algorithm has only one function which calculates the distance between the two strings. Nevertheless, we still need to perform some preliminary steps on the CPU side before starting the computation on the GPU side. Specifically, the CPU side controls the dimension of array which corresponds to the number of elements that will be run in parallel. Then the GPU side calculates each data element in parallel. This makes the CPU the controller because of its capability to change the direction of instructions.[2] The number of threads we use is 256. As shown in Table I, this number of threads leads to the best utilization of the GPU.

[2]https://en.wikipedia.org/wiki/Central_processing_unit

TABLE I
GPU UTILIZATION %

Compute Capability Threads per block	1.0	1.1	1.2	1.3	2.0	2.1	3.0
64	67	67	50	50	33	33	50
96	100	100	75	75	50	50	75
128	100	100	100	100	67	67	100
192	100	100	94	94	100	100	94
256	100	100	100	100	100	100	100
384	100	100	75	75	100	100	94
512	67	67	100	100	100	100	100
768	N/A	N/A	N/A	N/A	100	100	75
1024	N/A	N/A	N/A	N/A	67	67	100

Furthermore, we use one direction for data transfer as an optimization technique to reduce the effect of bus delay. The CPU just calculates the number of elements that will run in parallel for current iteration and sets the size of memory blocks to be appropriate for the GPU session. The block size (bs) is computed based on the slide ID (sID) by using the following equation.

$$bs = sID - 2 \times Z + 1$$

where: Z is calculated as follows.

$$Z = \begin{cases} 0, & ifsID < N \\ sID - N + 1, & otherwise \end{cases} \quad (2)$$

One of the limitations we faced with our experimental setting is the lack of support for large-scale 2D arrays. So, as a preprocessing step of our parallel diagonal implementation (shown in Algorithm 3), we convert the 2D array into 1D array and fill the 1D array in a column by column fashion. We use the following equation to do this conversion.

$$index = row \times width + column$$

In the GPU side we use the following two equations to convert the index back from 1D to 2D in order to get the correct location in matrix. Then, access the correct index same as sequential version as shown in pseudo code of CUDA kernel Algorithm 4.

$$row = \frac{index}{width} \quad (3)$$

$$column = index\%width \quad (4)$$

IV. RESULTS AND DISCUSSION

This section displays the results for our new implementation. The main objective is to compute the amount of speed-up for scalable data sizes starting from strings of length 250 characters and growing up to strings of length 8000 characters. For each data size, 30 different tests are conducted and the the averages are reported. Finally, the improvement is calculated by dividing the CPU time over the GPU time as the following equation shows.

$$improvement = \frac{time_{CPU}}{time_{GPU}}$$

The experimental setup we use is as follows.

Algorithm 3 Parallel Diagonal Implementation

```
1: procedure LEVENSHTEIN($Str1, Str2$,N)
2:     Initialize first row and first column from 1 to N
3:     for <Slice = 0 to N*2-1> do
4:         if $slice < N$ then
5:             $Z = 0$
6:         else
7:             $Z = sliceN + 1$
8:         end if
9:         Size = $\lceil slice - 2 * z + 1 \rceil$
10:        CUDA_KERNEL<<<SIZE, 265>>>
11:    end for
12: end procedure
```

Algorithm 4 CUDA_KERNEL for Diagonal Implementation

```
1: procedure CUDA_KERNEL($Str1, Str2$,N,z,slice,Increment)
2:     Calculate thread ID
3:     if $Z <= 0$ then
4:         Start index = slice
5:     else
6:         Start index= Increment * z + slice
7:     end if
8:     j = start Index+(ID*Increment)
9:     Calculate Row using Equation 3
10:    Calculate Column using Equation 4
11:    $Index = Row * Width + column$
12:    H[index]= Calculate Distance as Equation 1
13: end procedure
```

- Hardware: GPU (GT 740M NVIDIA) 2GB memory, Intel CPU I7 with 6GB RAM and memory bandwidth 14.40 GB/s.
- Software: Windows 10 as operating system, Microsoft Visual Studio 2013, CUDA toolkit V7.5 and NVIDIA drivers.

A shown in Table II, the big size of the array and the length of the two sequences does not affect the GPU performance. On the other hand the CPU performance is degraded when we have bigger arrays and longer sequences. Due to the parallelism used in GPU the performance is increased 11X compared to the CPU performance.

TABLE II
THE PERFORMANCE OF THE CPU AND GPU IMPLEMENTATIONS FOR DIFFERENT INPUT SIZES

Data Size	CPU(sec)	GPU(sec)	Improvement
250	0.000733333	0.000566667	1.294117647
500	0.004366667	0.001766667	2.471698113
1000	0.023933333	0.004933333	4.851351351
2000	0.0878	0.0155	5.664516129
3000	0.242166667	0.023933333	10.1183844
4000	0.4593	0.0459	10.00653595
5000	0.722333333	0.0658	10.97771023
6000	1.052433333	0.091833333	11.46025408
7000	1.477566667	0.129366667	11.42154084
8000	2.018066667	0.181333333	11.12904412

V. Conclusion

Levenshtein Distance algorithm is an algorithm proposed to calculate the distance between two strings. This algorithm is widely used in NLP for spell checking and other tasks. Moreover, this algorithm is very useful in bioinformatics, as it used to calculate pattern matching between two DNA or protein sequences. However, the high computation cost of this algorithms prohibits it from being very useful in practical large-scale scenarios. In this work, we decreases the execution time for long two sequences with two main techniques. First one is done by reading the matrix diagonally, which mitigates the dependence problem between matrix cells. Secondly, we improved the performance of Levenshtein algorithm by 11X faster more than sequential implementation. This improvement is done by using many techniques with parallel implementation, such as converting the array from 2D to 1D and using one direction data transfer. As for memory management, we avoided using shared and global memory of the GPU, because they are slower than registers.

References

[1] J. Serra, E. Gómez, P. Herrera, and X. Serra, “Chroma binary similarity and local alignment applied to cover song identification,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 16, no. 6, pp. 1138–1151, 2008.

[2] V. Pais and D. Ciobanu, “Fast name searching on gpu,” p. 217, 2013.

[3] K. Kukich, “Techniques for automatically correcting words in text,” *ACM Computing Surveys*, vol. 24, p. 64, 1992.

[4] S. Soroushnia, M. Daneshtalab, T. Pahikkala, and J. Plosila, “Parallel implementation of fuzzified pattern matching algorithm on gpu,” pp. 341–344, 2015.

[5] V. Saikrishna, A. Rasool, and N. Khare, “String matching and its applications in diversified fields,” *International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 219–226, 2012.

[6] D. Cantone, S. Cristofaro, and S. Faro, “Efficient string-matching allowing for non-overlapping inversions,” *Theoretical Computer Science*, vol. 483, pp. 85–95, 2013.

[7] Ł. Ligowski and W. Rudnicki, “An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–8.

[8] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, “Gpu accelerated smith-waterman,” pp. 188–195, 2006.

[9] S. Cook, *CUDA programming: a developer’s guide to parallel computing with GPUs.* Newnes, 2012.

[10] Y. Jararweh, S. Hariri, and T. Moukabary, “Simulating of cardiac electrical activity with autonomic run time adjustments,” *AHSC frontiers in biomedical research*, 2009.

[11] M. Al-Ayyoub, A. M. Abu-Dalo, Y. Jararweh, M. Jarrah, and M. Al Sa’d, “A gpu-based implementations of the fuzzy c-means algorithms for medical image segmentation,” *The Journal of Supercomputing*, vol. 71, no. 8, pp. 3149–3162, 2015.

[12] M. A. Shehab, M. Al-Ayyoub, and Y. Jararweh, “Improving fcm and t2fcm algorithms performance using gpus for medical images segmentation,” in *Information and Communication Systems (ICICS), 2015 6th International Conference on.* IEEE, 2015, pp. 130–135.

[13] C. El Amrani, “A learning approach to introducing gpu computing in undergraduate engineering program,” *International Journal of Computer Applications*, vol. 107, no. 20, 2014.

[14] T. Siriwardena and D. Ranasinghe, “Accelerating global sequence alignment using cuda compatible multi-core gpu,” in *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on.* IEEE, 2010, pp. 201–206.

[15] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[16] L. Chao and W. Fan, “Pair-wise sequence alignment algorithm in bioinformatics,” in *Electrical & Electronics Engineering (EEESYM), 2012 IEEE Symposium on.* IEEE, 2012, pp. 36–38.

[17] M. Refat, M. Shehab, Y. Jararweh, and M. Al-Ayyoub, “Accelerating needleman-wunsch global alignment algorithm with gpus,” in *The 12th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2015).* IEEE, 2015.

[18] G. Myers, “A fast bit-vector algorithm for approximate string matching based on dynamic programming,” *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[19] K. Xu, W. Cui, Y. Hu, and L. Guo, “Bit-parallel multiple approximate string matching based on gpu,” *Procedia Computer Science*, vol. 17, pp. 523–529, 2013.

[20] R. Prasad, A. K. Sharma, A. Singh, S. Agarwal, and S. Misra, “Efficient bit-parallel multi-patterns approximate string matching algorithms,” *Scientific Research and Essays*, vol. 6, no. 4, pp. 876–881, 2011.

[21] S. Wu and U. Manber, “Fast text searching: allowing errors,” *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[22] Z. Du, Z. Yin, and D. A. Bader, “A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda,” in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–8.

[23] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, “Gpu-clustalw: using graphics hardware to accelerate multiple sequence alignment,” in *High Performance Computing-HiPC 2006.* Springer, 2006, pp. 363–374.