Serial and parallel implementation of Needleman-Wunsch algorithm

Yun Sup Lee [a,1,*], Yu Sin Kim [a,2], Roger Luis Uy [a,3]

[a] College of Computer Studies, De La Salle University, 2401 Taft Avenue, Manila 1004, Philippines
[1] yun_sup_lee@dlsu.edu.ph; [2] yusin_kim@dlsu.edu.ph; [3] roger.uy@dlsu.edu.ph
* corresponding author

ARTICLE INFO

Article history
Received May 24, 2019
Revised January 5, 2020
Accepted February 22, 2020
Available online March 31, 2020

Keywords
Bioinformatics
Global alignment
Needleman-Wunsch
GPU
CUDA

ABSTRACT

Needleman-Wunsch dynamic programming algorithm measures the similarity of the pairwise sequence and finds the optimal pair given the number of sequences. The task becomes nontrivial as the number of sequences to compare or the length of sequences increases. This research aims to parallelize the computation involved in the algorithm to speed up the performance using CUDA. However, there is a data dependency issue due to the property of a dynamic programming algorithm. As a solution, this research introduces the heterogeneous anti-diagonal approach, which benefits from the interaction between the serial implementation on CPU and the parallel implementation on GPU. We then measure and compare the computation time between the proposed approach and a straightforward serial approach that uses CPU only. Measurements of computation times are performed under the same experimental setup and using various pairwise sequences at different lengths. The experiment showed that the proposed approach outperforms the serial method in terms of computation time by approximately three times. Moreover, the computation time of the proposed heterogeneous anti-diagonal approach increases gradually despite the big increments in sequence length, whereas the computation time of the serial approach grows rapidly.

1. Introduction

Bioinformatics is a discipline that applies the principles of computer science, mathematics, and engineering to answer questions related to Biology [1]. Notably, the pairwise sequence alignment is one very common yet the most essential task [2]. It is a technique that computes for the most optimal alignment, given a pair of genetic sequences [3]. The goal is to identify regions of similarity from the genetic sequences that may be a consequence of functional, structural, or evolutionary relationships between the sequences, believing that the similar alignments and functionalities [4].

Various approaches were introduced for the sequence alignment, such as a dynamic programming approach and a heuristic approach at the expense of accuracy [5]. Two widely known dynamic programming based pairwise sequence alignment algorithms are Needleman-Wunsch (NW) algorithm [2] for the global alignment and Smith-Waterman (SW) algorithm [6] for the local alignment [7]. Both algorithms find the most optimal alignment given a pair of sequences, and their computation time is proportional to the length of two sequences to be aligned [8]. Therefore, the computation time may increase significantly when the sequence length reaches more than millions. Tools that employ heuristic approaches such as FASTA [9] and BLAST [10] have shown to perform 40 times faster than the Central Processing Unit (CPU) based serial implementation of the SW algorithm [11]. However, the outputs of these tools are approximations of the optimal solution [5].

The advancements in hardware paved the way for massive computation power [12]. Notably, the improvements in the performance and capabilities of the Graphics Processing Unit (GPU) placed itself to be the leading competitive computing hardware of chip-level parallelism [13]. It reduces the computation time by concurrently executing multiple processes utilizing threads and processing units [14]. Numerous attempts were made to adapt these parallelism techniques to improve the performance of the sequence alignment algorithm [15][16].

Researchers initially debated on the CPU and GPU implementations [17]–[19]; however, it shifted to a paradigm of combining CPU and GPU to achieve further computation gains. This is also known as heterogeneous computing [20]. The heterogeneous computing environment orchestrates the interconnected machines such as CPU and GPU to perform an application whose subtasks have diverse execution requirements [21]. However, software development infrastructure for the parallel programming and libraries supporting the multiple vendors' hardware platforms are needed for parallel architectures and heterogeneous computing [22].

Compute Unified Device Architecture (CUDA) is one recently introduced framework that makes use of parallel compute engines in NVIDIA GPUs to solve complex computational problems efficiently [23]. CUDA is mapped to various applications and enhances the performance significantly. We also aim to benefit CUDA to improve the performance of the NW algorithm [24][25]. However, the parallel execution of sequence alignment algorithms may cause a problem. The problem is related to the data dependency issue. It may lead to the exploration of different parallelization and vectorization techniques [2].

Numerous researches have been conducted to enhance the performance of sequence alignment algorithms by parallelization. In Ling *et al.* [26], the NW algorithm was implemented on a CUDA-compatible GPU employing a divide-and-conquer technique. It divides the entire matrix computation into sub-matrices. A certain number of threads and amounts of memory are allocated in each sub-matrix for parallel execution. The computation time improved 15 times faster than the CPU implementation.

In Chen *et al.* [27], all the elements in the same column of the dynamic programming matrix were computed in parallel independently. It enhanced the NW algorithm performance faster by 37 times. Similarly, Che *et al.* [28] states that the implementation of dynamic programming based sequence alignment in CUDA showed 2.9 times faster in computation time compared to the single-threaded CPU based implementation. In addition, to GPU implementation of the NW algorithm, Jararweh *et al.* [8] represented two-dimensional array as a one-dimensional array for memory optimization, which positively affected the overall performance. The performance became 72.5 times faster than the sequential implementation.

Another research is about accelerating global sequence alignment using Cuda which was conducted by Siriwardena and Ranasinghe [7] stated that the use of different memory types in GPU gives different results. The performance got twice faster when global memory was used, while the use of shared memory achieved 4.2 times faster compared to the CPU based implementation. The entire matrix is divided into blocks, and they are computed in parallel using an anti-diagonal (AD) technique [29]. Research Fakirah *et al.* [30] used a method similar to Siriwardena and Ranasinghe [7], and it was able to achieve the performance gain of 93.7 percent at a sequence length of 20,000.

This paper proposes a heterogeneous AD technique to improve the NW algorithm performance. The main difference is that this research divides the matrix into blocks in a different way, and thus, the parallelization is performed differently. A similar approach to our study was taken by Fakirah *et al.* [30]. It, however, differs in the implementation of vectors and the type of graphics card used. This research only uses three vectors for the entire process, which may contribute to memory optimization as well.

The rest of this paper is organized as follows. The first few sections discuss the conventional NW algorithm, techniques, and approaches. Then the experimental setup is presented. The serial implementation is done in Java programming language, while the proposed heterogeneous AD implementation is done in C/C++ programming language with CUDA. Finally, results and conclusions will follow. Terms such as CPU and host are used interchangeably to avoid possible confusion, while terms such as GPU, device, and kernel are used interchangeably as well.

2. Method

2.1. Needleman-Wunsch (NW) Algorithm

NW algorithm is a well-known dynamic programming algorithm used for global sequence alignment [2]. NW algorithm aims to find the most optimal alignments between the two given genetic sequences among the many possible alignments and the score given to the most optimal alignment is called the maximum score.

NW algorithm makes use of a two-dimensional array, each cell holding two different values: a score and a pointer. The pointer is a directional navigator that points to the left, top, or diagonal (upper-left). It indicates where the current cell's maximum score came from. These pointers are used during the trace-back step to retrieve two aligned genetic sequences. On the other hand, scores are the values assigned to each cell. NW algorithm requires a predefined score schema to compute for the score. The schema includes 1) miss, 2) gap, and 3) match.

Three different scores are computed based on the scoring schema and the maximum score is chosen. The first of the three scores is the horizontal gap score that is the sum of the cell to the left and the gap score. The vertical gap score that is the sum of the cell to the up and the gap score, while a diagonal (upper-left) score that is the sum of the diagonal (upper-left) cell score and either match- or miss-score. Match-score is added if the pairwise sequences at the current index match; otherwise, the miss-score is added. Such computation continues until all the cells are filled up. Once finished, the value at the most bottom-right cell is the maximum score.

The NW algorithm has three steps: initialization, fill, and traceback. However, only initialization and fill steps are discussed in this paper. This is because the traceback step does not involve computations.

2.2. Row-Wise (RW) Technique

The RW technique solves the problem using two vectors, each acting as a reference and a current row. The reference row is used to compute for the current row. Once the current row's computation is done, it becomes a reference row for the next new current row. In other words, the role of row changes on every-iteration from current to reference and vice versa. This process continues until all rows are filled up. The whole process is divided into initialization and fill steps.

The very first row is initialized during the initialization step, as shown in Fig. 1, and it serves as a reference row. The very first column of the row is set to 0, while the rest are set using a gap-score. Then, the fill-step begins having the second row as a current row. The first row now serves as a reference row. Equation (1) is used during the computation. REF refers to the reference row, and CUR refers to the current row.

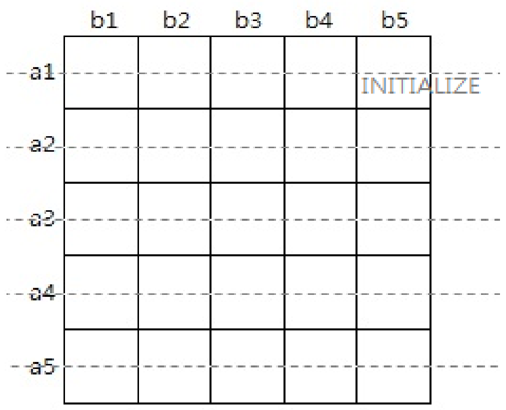

Fig. 1. Row-Wise Technique

In addition to data dependency, using two vectors allows longer input sequences. The maximum size of a matrix is 2^{32} bytes of a 2D matrix as the product of the row and column elements. The size implies that the product of input pairwise sequence lengths is limited to 2^{32} bytes. This is because the lengths of pairwise sequences correspond to the number of row and column elements of a 2D matrix. Therefore, representing the pairwise sequences as two distinct vectors allows each to have a length of at most 2^{32} bytes.

$$CUR\ (i) = MAX\ (REF\ (i-1) + s(x_{row}, y_i), REF\ (i) + gap, CUR\ (i-1) + gap) \tag{1}$$

2.3. Anti-Diagonal (AD) Technique

The AD technique addresses not only the sequence length limitation problem but also the data dependency issue for the parallel implementation [7]. Here, cells that fall under the anti-diagonal line form a row and represent as a vector, as shown in Fig. 2. In AD technique discussion, the terminology both row and vector refer to a group of cells that fall under an anti-diagonal line.

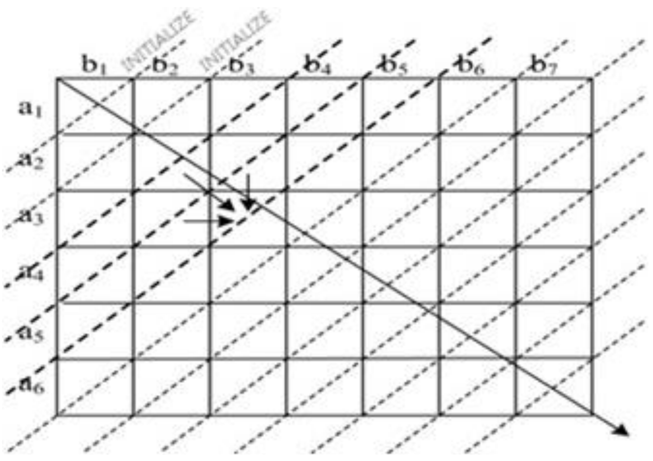

Fig. 2. Anti-Diagonal Technique

There are three vectors used during the computation, two for the references and one for the current row. One of the reference vectors is used for the diagonal score called REF_D, while the other one is used for the horizontal and vertical score called REF_HV. The CUR is the current row that is currently computed. AD technique is divided into initialization and fill steps.

During the initialization step, two reference vectors are filled up, as shown in Fig. 2. The first two initialized anti-diagonal vectors serve as a REF_D and a REF_HV, respectively. The former is set to 0, while the two cells in the latter are set to gap score. After the initialization, the filling step begins with the third anti-diagonal row serving as a current row.

The filling step may further be divided into FORMER, MID, and LATTER phases. These phases are distinguished by the length of the vertical sequence, as shown in Fig. 3. Each anti-diagonal row is given a sequential index from 0 to N, where N refers to the total number of anti-diagonal rows minus one. The anti-diagonal rows with an index less than the length of vertical sequence fall under the FORMER phase. The row with an index equal to the length falls under the MID phase. Lastly, the rest of the anti-diagonal rows fall under the LATTER phase. Their indices are greater than the length of the vertical sequence. Each phase makes use of different equations: (2), (3) and (4) are used for the FORMER, MID, and LATTER phases respectively:

$$\begin{aligned} CUR\ (i) = {} & MAX\ (REF\_D\ (i-1) + s\ (x_{row}, y_{col}), REF\_HV\ (i-1) + \\ & gap, REF\_HV\ (i) + gap) \end{aligned} \tag{2}$$

$$\begin{aligned} CUR\ (i) = {} & MAX\ (REF\_D\ (i-1) + s\ (x_{row}, y_{col}), REF\_HV\ (i) + \\ & gap, REF\_HV\ (i+1) + gap) \end{aligned} \tag{3}$$

$$CUR\ (i) = MAX\ (REF\_D\ (i+1) + s\ (x_{row}, y_{col}), REF\_HV\ (i) + gap, REF\_HV\ (i+1) + gap) \tag{4}$$

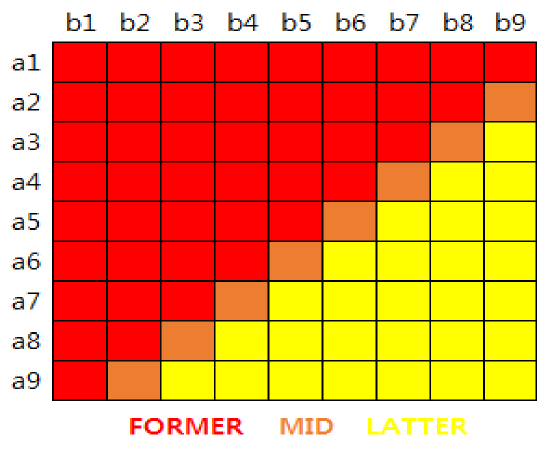

Fig. 3. Anti-Diagonal Technique Phases

2.4. Serial Approach (Java)

The serial approach is implemented serially and uses CPU only. It is implemented in two different ways. The first serial implementation employs an RW technique. The very first row is initialized, as shown in Fig.1, and the computation starts from the second row. Cells in a row are computed from left to right sequentially before moving to the next row.

The second serial implementation employs an AD technique. The first two AD rows are initialized, as shown in Fig. 2, and computation starts from the third AD vector. Cells within the AD vector are computed sequentially before moving to the next AD vector. Computations for both implementations will continue until all the cells are calculated.

2.5. Heterogeneous Anti-Diagonal Approach (CUDA C/C++)

The heterogeneous anti-diagonal approach is implemented in a CPU-GPU heterogeneous [20] manner using CUDA C/C++. At first, five different vectors (one dimension) are initialized on GPU memory: two vectors for pairwise sequences and three vectors (interchangeably used as REF_D, REF_H, and CUR, refer to Section 2.3). The host then determines details (e.g., number of threads, roles of three vectors, etc.) and invokes kernel for parallel computation. Invocation occurs iteratively in a sequential manner from the top-left to bottom-right direction, as shown in Fig. 4 (number of invocations, number of AD vectors, and number of iterations are equal).

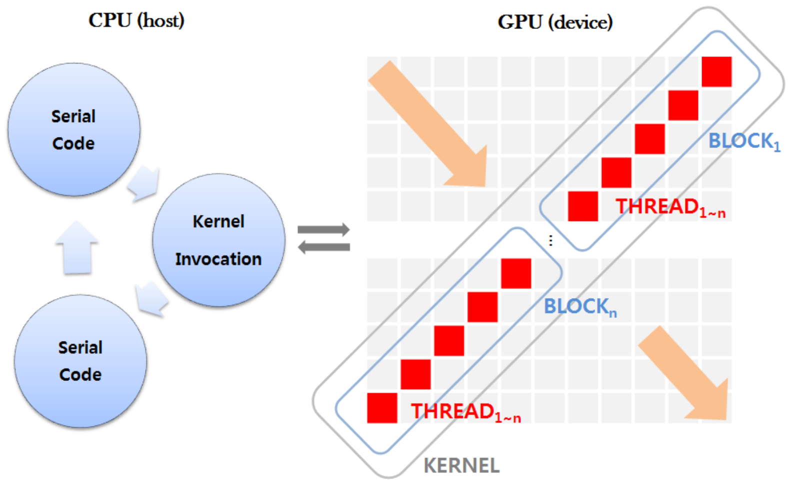

Fig. 4. Heterogeneous Computing Model

CUDA allows a host to access memories residing in a device memory [24]. Each thread is assigned with a local memory called a register, and all threads in all blocks have access to a global memory to enable block-level communication [23]. Other memory types are shared memory, texture memory, and constant memory; however, these are out of scope in this research.

Registers are used temporally to store scores (discussed in Section 2.1), and global memory stores five distinct vectors throughout the computation. The length of two vectors equals to the length of two input sequences, respectively, while the length of three AD vectors equals the sum of two sequence lengths. AD vector is stored as a vector, and cells in the vector are the threads.

Robust computation of CUDA comes from multi-threading, i.e., blocking and threading features. A kernel consists of blocks, and each block contains numerous threads [24]. Ideally, each of the unlimited blocks contains thousands of threads executed simultaneously; however, it varies on the GPU model in reality. GPU used in this research supports a single kernel that is capable of 65,535 blocks as maximum and at most 1,024 threads per block. For the implementation, the number of blocks is computed based on the number of threads. Here, the number of threads equals the number of cells in an AD vector. For example, 2,048 threads form two blocks, and 4,022 threads form four blocks.

The sequence of host invoking device and assigning of three AD vectors' roles are shown in Fig. 5. In Fig. 5, A={a_1, a_2, ..., a_{na}} and B={b_1, b_2,..., b_{nb}} are the input sequences, and colors indicate to which vector the cells belong: red (REF_D), blue (REF_HV), green (CUR), light gray (to be visited), dark gray (done), black rectangle (maximum score), and AD vector index represent the sequential order of device invoked by the host. REF_D, REF_HV, and CUR are interchangeably used. Cells in CUR are computed in parallel, referencing REF_D and REF_HV.

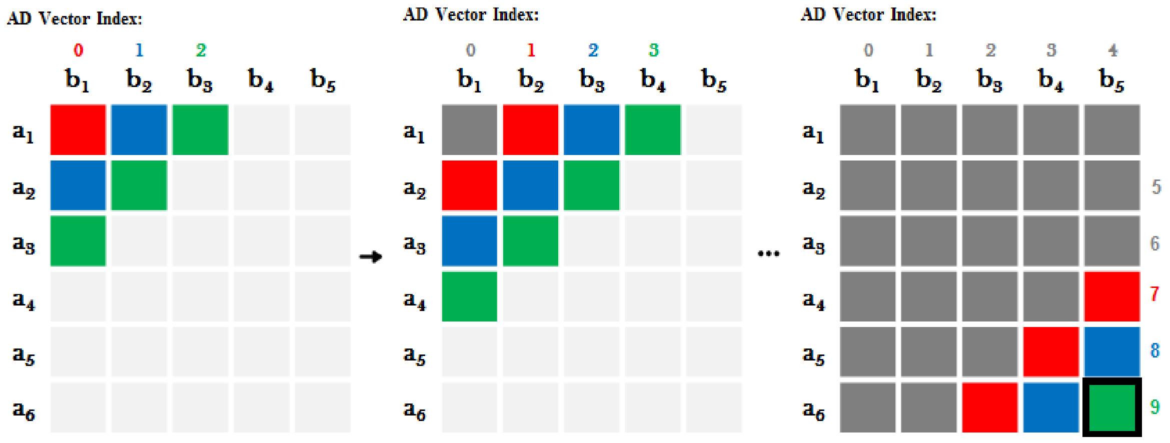

Fig. 5. Heterogeneous Anti-Diagonal Approach.

The first two iterations invoke the device for initialization (initialization step), and the filling step begins at the third iteration. The AD vector at nth iteration is assigned as CUR, and two previous AD vectors are used as REF_D and REF_HV. This guarantees that all reference scores such as diagonal, vertical, and horizontal scores are computed in prior and allows parallel computation for new scores without any data dependency issues. On the following iteration, REF_HV becomes REF_D, CUR becomes REF_HV, and REF_D becomes CUR, and values are disposed of. The process continues throughout the entire iterations, and the maximum score is eventually obtained. The host also determines indices used for peeking global memory for sequence characters during the computation.

Based on the implementation, the length of AD vectors exceeds the required number of threads most of the time. We manage this issue by enabling the cells (i.e., threads) that need to be computed and disable excessive ones.

2.6. Experimental Setup

Pair of genetic sequences consisting of four characters, i.e., A, C, G, T, are randomly generated. The length of generated genetic sequences starts at 10,000 and gradually increments to 100,000 by 10,000, followed by the sequences with the lengths of 200,000 and 300,000. The correctness of serial and heterogeneous implementations are measured by comparing their maximum scores. Both applications are expected to output the same value. The computation time for both implementations is obtained from the average of five executions. This is to avoid any misleading results because even the same application may perform at a different speed. Computation time measures the time spent by both CPU and GPU at fill step only for all implementations. All experiments are conducted on a PC with a 1.80GHz Intel Core i5-3337U CPU, 8GB RAM, and running x64-based Windows 7. Nvidia GeForce 710M with 2GB memory is used

3. Results and Discussion

3.1. Results

Table 1 shows the averaged computation time for a given sequence length. Fig. 6 shows the computation time of the NW algorithm's serial and heterogeneous implementations using an AD technique. The computation time includes both initialization and the filling steps. The computation time of heterogeneous implementation increases gradually as the number of input sequences increases. On the other hand, the computation time of CPU implementation starts to increase rapidly at a certain length. From the results, we also derived the following equations:

$$y = 0.019\,x - 4{,}093 \tag{5}$$

$$y = 0.296\,x - 12{,}147 \tag{6}$$

where the (5) is for the heterogeneous implementation while the (6) is for the serial implementation, and x is the sequence length. Both equations are shown in Fig. 6 in dotted lines. They represent the tendency of computation time against the varying sequence length.

Table 1. Averaged Computation Time in Seconds

Sequence Length	Implementations		
	Serial		*Heterogeneous*
	RW Technique [a]	*AD Technique*	*AD Technique*
100,000	119	103	33
200,000	362	411	129
300,000	804	895	299
400,000	1,732	1,555	577
500,000	2,145	2,429	896
600,000	3,103	3,519	1,190
700,000	4,277	4,817	1,544
800,000	5,529	6,394	2,117
900,000	7,057	8,231	2,759
1,000,000	9,419	9,716	3,301
2,000,000	-[b]	40,013	13,238
3,000,000	-[b]	87,247	29,499

[a] An additional experiment performed as discussed in section 3.3
[b] An experiment was not performed due to the limited experimentation time

In addition to computation time, an experiment was conducted to measure the overhead during the data transfer. The data transfer includes copying input pairwise genetic sequences from host to device and retrieving maximum score from device to host. This experiment confirmed that the overhead does not have a significant impact on the overall computation times.

Overhead during the input data transfer increases gradually as the input length increases; however, it is a negligible amount. For example, the overhead at sequence length of 100,000 and 500,000 are only 0.05 seconds and 0.248 seconds, whereas it took 33 seconds and 896 seconds, respectively, for the computation. In percentage, both overheads are less than 0 percent of the computation time. It reached 1.451 seconds at 3,000,000 length sequence, which is 0.005 percent of the computation time 29,499 seconds.

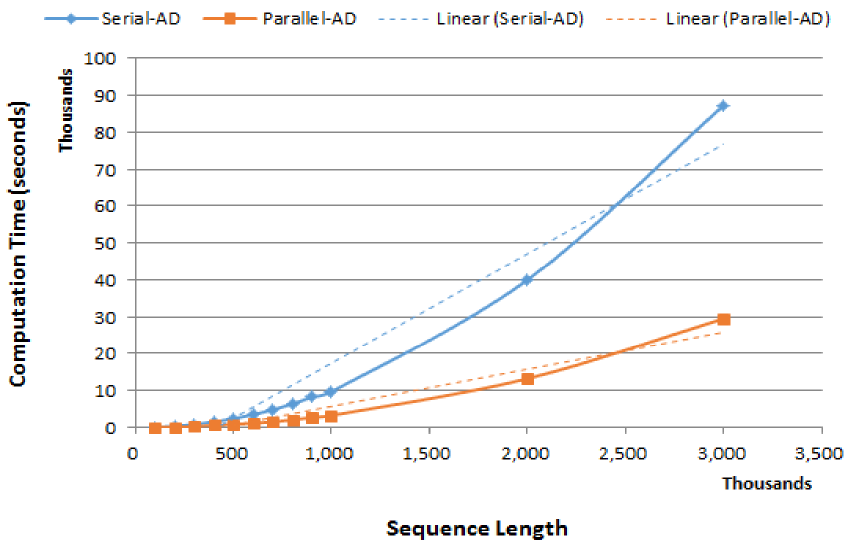

Fig. 6. Computation Time Comparison

The overhead while retrieving the maximum score from device to host is much less than the input data transfer. Besides, data transfer time remains the same regardless of the input data length because it retrieves merely a single value from the device to host. Fig. 7 shows a gradual increase in overhead for input data transfer while there is no overhead difference during maximum score transfer.

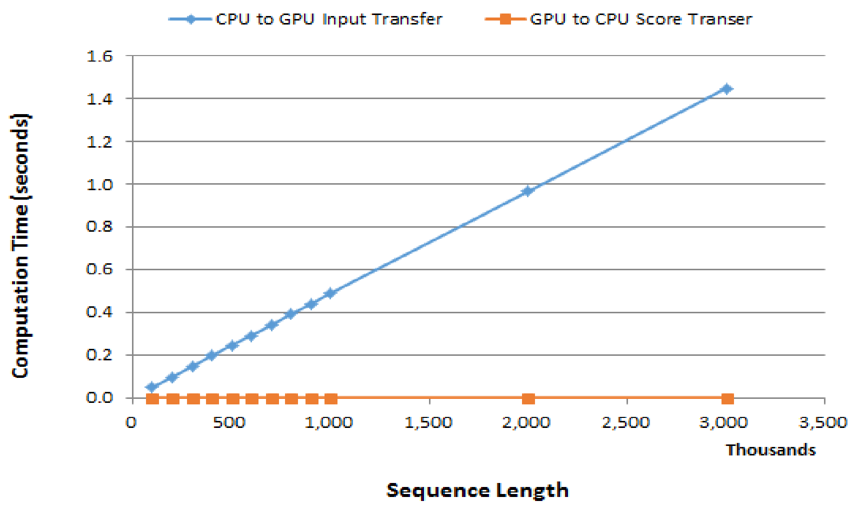

Fig. 7. Data Transfer Overhead Time

Fig. 8 shows how the sum of overheads and computation time affects the original result. The result confirms that heterogeneous implementation still outperforms the serial implementation despite the overheads, proves that the overheads during the data transfers are negligible.

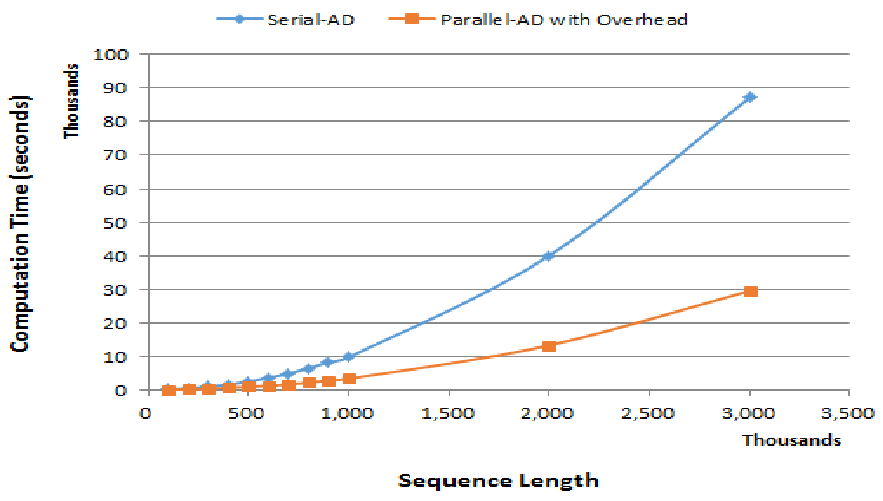

Fig. 8. Computation Time with Overheads Comparison

3.2. Analysis and Discussion

We first briefly analyze the time complexity of both implementations. The serial implementation consists of two nested loops. The outer loop iterates n times, where n is the sum of pairwise sequence lengths. At each iteration, the inner loop iterated at most m times, where m equals the number of shorter sequence length between the two in $O(nm)$. Similarly, the heterogeneous implementation uses a nested loop. The outer loop iterated n times. However, the inner loop performs in a parallel manner. Therefore, the worst case of this implementation would be $O(np)$, assuming that p is the worst case among the parallel executions of the inner loop.

The actual computation time of heterogeneous implementation seemed to increase exponentially at first because it increased by 239 percent from 10,000 to 20,000 length sequences. However, the computation time of 30,000 length sequence has increased only by 128 percent when compared to 20,000 sequence length. On the following inputs, computation time started to rise consistently, eventually increasing at an average of 30 to 40 percent, as presented in Fig. 6, where the slope of the equation is neither steep nor flat, but it gradually increases.

The computation time of serial implementation increased rapidly at first. From the sequence length of 100,000 to 200,000, the computation time increases from 103 seconds to 411 seconds, which is a 299 percent increase. However, this increasing percentage reduced to 118 percent from 200,000 to 300,000 long input sequences, eventually decreasing to 30 to 40 percent. Even though the rising percentage is similar to that of heterogeneous implementation, the actual value is much more significant. This big difference in real value makes the equation's slope steep, which also implies that the computation time increases rapidly as the input length increases.

Overall, the heterogeneous implementation performs approximately three times faster than the serial implementation. Both applications seemed to show equally at first based on the output; however, the difference between the two graphs increases as the input sequence length increases. The computation time difference at 100,000 sequence length was only 70 seconds. However, the heterogeneous implementation outperforms the serial implementation by 282 seconds when the input is 200,000 long. Then, the difference in computation time increases to 57,748 seconds with an input length of 3,000,000. This implies that GPU is maximized on intense computation tasks.

3.3. Additional Experiment

An additional experiment was performed to compare the performance of the serial implementation of the NW algorithm using RW and AD techniques. This required a further application of the RW technique, while previously obtained AD technique results are reused. Other conditions were equally applied, as discussed in the previous sections of this paper.

Fig. 9 shows that although the RW technique seemingly performs faster than the AD technique when the sequence length is approximately 500,000, the computation time gets closer again with 1,000,000 sequence length. The result implies that the heterogeneous implementation outperforms the serial implementation regardless of technique types. The exact computation time is found in Table 1.

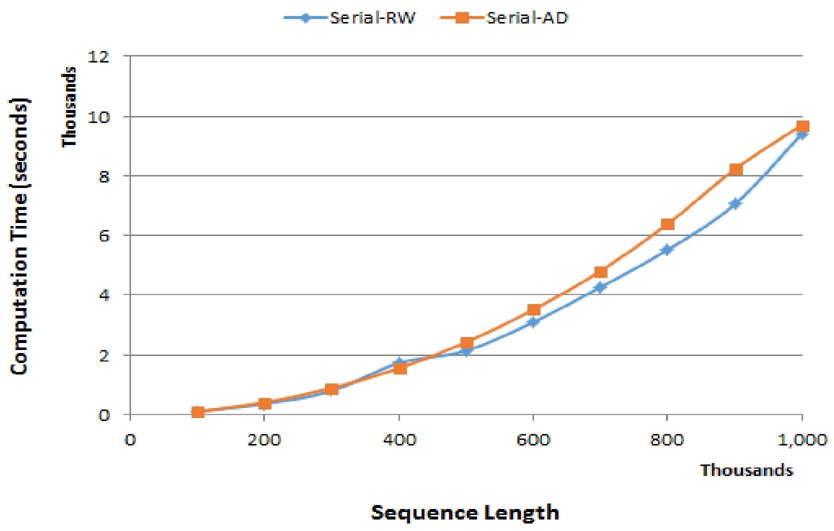

Fig. 9. Computation Time in Row-Wise and Anti-Diagonal Techniques in CPU Implementation

4. Conclusion

This paper aimed to compare serial and heterogeneous implementations of the NW algorithm using an AD technique. The serial application was executed on CPU, while heterogeneous implementation involving both CPU and GPU executions. The experiment showed that the heterogeneous AD implementation of the NW algorithm using three vectors outperforms the serial implementation by approximately three times in terms of computation time. The computation time of a serial implementation increases rapidly as the sequence length increments, while the computation time of a heterogeneous implementation increases gradually. Furthermore, utilizing only three vectors throughout the process allowed accommodating longer sequences. As a recommendation, using an advanced GPU is suggested for better performance. It allows multi-kernel invocation supporting the concurrent execution of multiple kernels. Besides, the proposed technique could be experimented on different GPU memory types as [7] showed that the performance might be faster or slower depending on the GPU memory type used

References

[1] S. El-Metwally, O. M. Ouda, and M. Helmy, *Next Generation Sequencing Technologies and Challenges in Sequence Assembly*, 2014, vol. 7, doi: 10.1007/978-1-4939-0715-1.

[2] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970, doi: 10.1016/0022-2836(70)90057-4.

[3] V. O. Polyanovsky, M. A. Roytberg, and V. G. Tumanyan, “Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences,” *Algorithms Mol. Biol.*, vol. 6, no. 1, p. 25, 2011, doi: 10.1186/1748-7188-6-25.

[4] X. Xia, “Sequence Alignment,” 2018, pp. 33–75, doi: 10.1007/978-3-319-90684-3_2.

[5] H. Khaled, R. El Gohary, N. L. Badr, and H. M. Faheem, “Accelerating pairwise DNA Sequence Alignment using the CUDA compatible GPU,” *Int. J. Comput. Appl.*, vol. 84, no. 1, pp. 25–31, Dec. 2013, doi: 10.5120/14542-2619.

[6] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981, doi: 10.1016/0022-2836(81)90087-5.

[7] T. R. P. Siriwardena and D. N. Ranasinghe, "Accelerating global sequence alignment using CUDA compatible multi-core GPU," in *2010 Fifth International Conference on Information and Automation for Sustainability*, 2010, pp. 201–206, doi: 10.1109/ICIAFS.2010.5715660.

[8] Y. Jararweh, M. Al-Ayyoub, M. Fakirah, L. Alawneh, and B. B. Gupta, "Improving the performance of the needleman-wunsch algorithm using parallelization and vectorization techniques," *Multimed. Tools Appl.*, vol. 78, no. 4, pp. 3961–3977, Feb. 2019, doi: 10.1007/s11042-017-5092-0.

[9] W. R. Pearson, "Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms," *Genomics*, vol. 11, no. 3, pp. 635–650, Nov. 1991, doi: 10.1016/0888-7543(91)90071-L.

[10] S. D. Eric, T. K. D. D. Nicholas, and K. A. Theophilus, "Bioinformatics with basic local alignment search tool (BLAST) and fast alignment (FASTA)," *J. Bioinforma. Seq. Anal.*, 2014, doi: 10.5897/ijbc2013.0086.

[11] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990, doi: 10.1016/S0022-2836(05)80360-2.

[12] M. A. Rahman and R. C. Muniyandi, "Review of GPU implementation to process of RNA sequence on cancer," *Informatics Med. Unlocked*, vol. 10, pp. 17–26, 2018, doi: 10.1016/j.imu.2017.10.008.

[13] M. Garland *et al.*, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008, doi: 10.1109/MM.2008.57.

[14] S. Warris *et al.*, "pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment," *PLoS One*, vol. 13, no. 1, p. e0190279, Jan. 2018, doi: 10.1371/journal.pone.0190279.

[15] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Correction to: GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data," *BMC Bioinformatics*, vol. 20, no. 1, p. 597, Dec. 2019, doi: 10.1186/s12859-019-3185-7.

[16] C.-L. Hung, Y.-S. Lin, C.-Y. Lin, Y.-C. Chung, and Y.-F. Chung, "CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on Multi-GPUs," *Comput. Biol. Chem.*, vol. 58, pp. 62–68, Oct. 2015, doi: 10.1016/j.compbiolchem.2015.05.004.

[17] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144, doi: 10.1109/ISPASS.2011.5762730.

[18] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU myth," in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, 2010, p. 451, doi: 10.1145/1815961.1816021.

[19] J. S. Vetter and S. Mittal, "Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing," *Comput. Sci. Eng.*, vol. 17, no. 2, pp. 73–82, Mar. 2015, doi: 10.1109/MCSE.2015.4.

[20] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 1–35, Jul. 2015, doi: 10.1145/2788396.

[21] S. Wang, A. Prakash, and T. Mitra, "Software Support for Heterogeneous Computing," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 756–762, doi: 10.1109/ISVLSI.2018.00142.

[22] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, May 2010, doi: 10.1109/MCSE.2010.69.

[23] Khoirudin and J. Shun-Liang, "GPU Application in Cuda Memory," *Adv. Comput. An Int. J.*, vol. 6, no. 2, pp. 01–10, Mar. 2015, doi: 10.5121/acij.2015.6201.

[24] *NVIDIA CUDA C Programming Guide (Version 4.2)*. 2012, available at: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[25] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014, available at: Google Scholar.

[26] C. Ling, K. Benkrid, and T. Hamada, "A parameterisable and scalable Smith-Waterman algorithm

implementation on CUDA-compatible GPUs," in *2009 IEEE 7th Symposium on Application Specific Processors*, 2009, pp. 94–100, doi: 10.1109/SASP.2009.5226343.

[27] B. Chen, Y. Xu, J. Yang, and H. Jiang, "A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform," 2010, pp. 79–90, doi: 10.1007/978-3-642-13119-6_7.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008, doi: 10.1016/j.jpdc.2008.05.014.

[29] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. S2, p. S10, Mar. 2008, doi: 10.1186/1471-2105-9-S2-S10.

[30] M. Fakirah, M. A. Shehab, Y. Jararweh, and M. Al-Ayyoub, "Accelerating Needleman-Wunsch global alignment algorithm with GPUs," in *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, 2015, pp. 1–5, doi: 10.1109/AICCSA.2015.7507113.