Implementations of the Needleman-Wunsch Algorithm for GPU Architectures

Furkan Kurt
Department of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
kurtfu@itu.edu.tr

Deniz Turgay Altilar
Department of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
altilar@itu.edu.tr

Ayse Yilmazer Metin
Department of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
yilmazerayse@itu.edu.tr

***Abstract*—Similarity search is a fundamental yet time-consuming algorithm in bioinformatics. Many dynamic programming-based and heuristic algorithms are proposed to solve alignment problems. The Needleman-Wunsch algorithm is a well-known dynamic programming-based algorithm for global sequence alignments. The algorithm has $O(n^2)$ time and space complexity. The quadratic complexity limits the use of the algorithm with relatively smaller sequences. Various parallel and distributed methods were proposed to overcome the quadratic complexity of the algorithm.**

In this paper, we describe a graphics processing unit(GPU) kernel to parallelize and reduce the execution time of the algorithm. We propose a new data partitioning method representation to increase the data transfer throughput between the GPU and the host. We implemented the serial approach of the algorithm and various parallel CUDA methods. We also used CUDA Cooperative Groups for the first time in Needleman-Wunsch algorithm parallelization. The evaluation shows that the new implementation is increased the performance of the algorithm 60 times for similarity score calculations, and 17 times for the alignment calculations.

***Index Terms*—parallel computing, sequence alignment, bioinformatics**

I. Introduction

Similarity search is a fundamental but time-consuming algorithm in bioinformatics. It can be used in sequence databases searches [1], sequence alignment [2], and mutation identification [3]. The similarity between two sequences can be obtained by using optimal alignment approaches. Various methods have been proposed to overcome this problem. They can be grouped under two categories: local alignment and global alignment. Local alignment methods search the most similar parts of sequences while global methods search the total similarity between sequences.

The Smith-Waterman algorithm [4] for the local alignment and the Needleman-Wunsch algorithm [5] for the global alignment are well-known algorithms that both are based on dynamic programming approaches. FASTA [6] and BLAST [7], [8] are two common heuristic approaches. Their performance is 40 times faster than the Central Processing Unit(CPU) based serial implementations. Despite their speeds, the outputs are approximations of the optimal solutions.

General-Purpose Computing on Graphics Processing Units(GPGPU) allows solving complex computational problems by using massively parallel architectures of Graphics Processing Units(GPU). However, the parallel implementations of sequence alignment algorithms face data dependency problems.

In this research, data dependency and synchronization problems are examined for parallel implementation approach of the Needleman-Wunsch algorithm on GPUs. The algorithm is used for two purposes: the final score calculation for the similarity comparison, and the alignment matrix generation for the sequence alignment. The proposed approach calculates the exact same results as the serial approaches. It first calculates the similarity score then if the score is above a threshold value, it generates the alignment matrix for sequence alignment.

II. Related Work

In recent years, various methods have been proposed for improving the Needleman-Wunsch global alignment algorithm. The methods contain massively parallel GPU cards, FPGA architectures, and distributed CPU/GPU implementations. All of the methods divide the alignment matrix into parallelizable elements and distribute the elements to computation units, then process them simultaneously.

Rodinia [9] is one of the first parallel implementations of the Needleman-Wunsch algorithm on GPU. The score matrix is processed from the top-left cell to the bottom-right cell in a diagonal strip manner. Rodinia-NW divides the alignment matrix into square tiles and computes each tile in a different thread block. The main limitation of the Rodinia-NW is that it only supports sequences with the same length which can be divided by 16. Li and Becchi [10] proposed a method to overcome Rodina's restrictions. Siriwardena and Ranasinghe [11] examined the effects of different GPU memory types.

There are also distributed approaches for the Needleman-Wunsch. Li et al. [12] proposed a distributed CPU-GPU implementation that is based on MPI-CUDA framework. Selvitopi et al. [13] proposed a distributed CPU implementation that is based on MPI and OpenMP frameworks.

Prior works focused on Multi-Alignment with a single GPU or global alignment in a distributed environment. This work focused on global sequence alignment for relatively long sequences with a single GPU.

III. Background Information

A. Needleman-Wunsch Algorithm

Needleman-Wunsch is a well-known, widely used dynamic programming algorithm for global sequence alignment. The algorithm's main purpose is to find the most optimal alignment between two amino-acid sequences. It consists of two steps:

- *Initialize:* In this step, each cell in the alignment matrix is filled by calculating match/mismatch, insert and delete scores.
- *Traceback:* After filling all cells, the recovery process starts from the bottom-right cell and follows the alignment pointer until reaching the origin.

A two-dimensional matrix is used to represent the alignments. Each cell in the alignment matrix holds a pointer that navigates the previous amino-acid in the alignment. The score calculation of each cell depends on three predetermined values. (1) The miss value that represents the score when two amino-acids mismatched, (2) the match value that represents the score when two amino-acids matched, and (3) the gap value that represents an insertion or deletion in the alignment.

Three different scores are calculated to determine the score of a cell. (1) The vertical reference score is the sum of the score of the cell to the up and the gap value. (2) The horizontal reference score that is the sum of the score of the cell to the left and the gap value, and (3) the diagonal reference score that is the sum of the score of the cells to the up-left and the miss or the match value depending on amino-acids correspond to that cell. After the calculation of all three references, the maximum of them is chosen for the score of that cell. Horizontal and vertical references are summed with the gap value, and the diagonal reference is summed with the $\sigma(i,j)$ that indicates match/miss value. Eq. 1 shows the calculation of the score of a cell.

$$M(i,j) = max \begin{cases} M(i-1,j-1) + \sigma(i,j) \\ M(i-1,j) + gap \\ M(i,j-1) + gap \end{cases} \quad (1)$$

With the help of the score matrix M, the alignment matrix can be generated. The alignment matrix is used in the traceback step to calculate the aligned sequences. Each cell of the alignment matrix is filled as in eq. 2.

$$H(i,j) = \begin{cases} diag & if\ M(i,j) = M(i-1,j-1) + \sigma(i,j) \\ up & if\ M(i,j) = M(i-1,j) + gap \\ left & if\ M(i,j) = M(i,j-1) + gap \end{cases} \quad (2)$$

Fig. 1 shows the traceback step of the algorithm. *D* denotes the diagonal movement on the matrix, *U* denotes the move to the upper cell, and *L* represents the move to the left. Arrows represent the best-aligned sequence movement on the matrix.

B. CUDA Execution Model

Compute Unified Device Architecture(CUDA) [14] allows to use GPUs in general-purpose programming. Nvidia GPUs include a set of Streaming Multiprocessors(SMs) and each one of them has a set of cores. These cores execute instructions in a Single Instruction Multiple Data (SIMD) manner. CUDA executes threads in different types of groups. These groups are *blocks* that have a set of threads, and *grids* that have a set of blocks. Fig 2. shows the organization of CUDA threads.

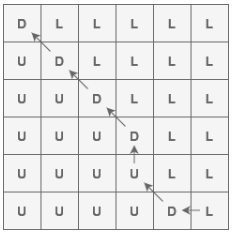

Fig. 1. Traceback step in the alignment matrix.

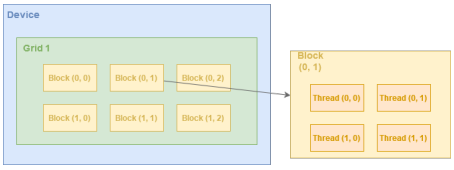

Fig. 2. Hierarchical organization of CUDA threads.

Each CUDA kernel is launched with user-defined dimensions of grid and block. The limits of these dimensions are specific to each GPU device. It could be limited with thread count or required memory per dimension. CUDA provides two types of parallelism: fine-grained parallelism within a thread-block and coarse-grained parallelism across multiple thread blocks.

C. Cooperative Groups

Earlier versions of CUDA only provide a thread synchronization mechanism for a block of thread. Synchronization of a block of the grid could only be done implicitly at the beginning or at the end of the kernel call. Fig 3. shows the prior synchronization model of the CUDA application that has multiple blocks. Each color represents the execution timeline of different blocks and red rectangles represent the synchronization calls.

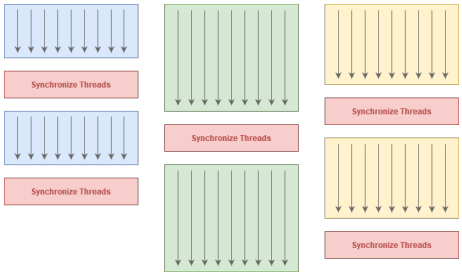

Fig. 3. Synchronization model of the default CUDA blocks.

CUDA 9 introduces a new mechanism *Cooperative Groups* for inter-block synchronization. With this new mechanism, any

device CUDA 9 compatible with Kepler or newer architecture can explicitly synchronize thread blocks within the kernel. Fig 4. shows the synchronization model of Cooperative Groups. Each color represents the execution timeline of different blocks and the red rectangle represents the synchronization calls.

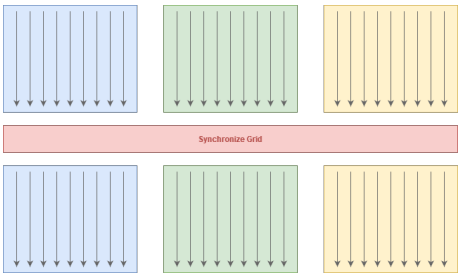

Fig. 4. Execution model of CUDA Cooperative Groups.

The main limitation of this new mechanism is maximum possible grid size. In default launch method grid size is determined by the device's capabilities while the Cooperative Groups only limited with the multiprocessor count of the device and the maximum number of threads per multiprocessor.

IV. Proposed Method

A. Serial Approach

In this approach, each cell is calculated sequentially. To calculate a cell's score, first, its three reference cells must be calculated. Fig. 5 shows the data dependencies in calculating the values of the matrix.

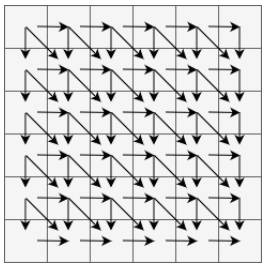

Fig. 5. Data dependency of the matrix.

The process starts from the very first row. The first row is taken as the current row and cells are filled according to the rule described in Section III *Initialize* step. Each value of that row is filled iteratively. Filling the first row is done according to the eq. 3. $CURR$ denotes the very first row of the score matrix and i denotes the column number.

$$CURR\,(i) = i * gap \tag{3}$$

After the calculation of the first row, the first row is taken as the reference for its next row. The process continues until the last cell of the last row as Values from the previous row are taken as references for the current row. Filling the rest of the rows is done according to the eq. 4. $CURR$ denotes the current row and the $PREV$ denotes the previous row.

$$CURR\,(i) = MAX\,(PREV\,(i-1) + \varepsilon, PREV\,(i) + gap, CURR\,(i-1) + gap) \tag{4}$$

The serial approach requires visiting each cell of the matrix individually. For a matrix with a size of n rows and m columns, it takes $n * m$ iterations. That makes the time complexity of the serial approach $O(n^2)$.

B. Parallel Approach

Each value of a cell in an anti-diagonal vector can be calculated simultaneously. The calculation of the value in a cell is dependent on the previous anti-diagonal vector for horizontal and vertical references, and the previous vector of that for the diagonal reference. Fig. 6 shows the data dependencies of the anti-diagonal vectors of the score matrix. Blue denotes the current anti-diagonal vector, red denotes the anti-diagonal vector for horizontal-vertical references, and green denotes the anti-diagonal vector for diagonal references.

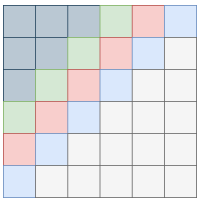

Fig. 6. Data dependencies of the anti-diagonal vectors.

Two additional vectors are required for calculating an anti-diagonal vector. One vector for storing horizontal-vertical references, and one for storing diagonal references. The calculation of the values in an anti-diagonal vector is done as in eq. 5. $CURR$ denotes the current anti-diagonal vector, HV denotes the anti-diagonal vector for horizontal-vertical references, and the $DIAG$ denotes the anti-diagonal vector for diagonal references.

$$CURR\,(i) = MAX\,(DIAG\,(i-1) + \varepsilon, HV\,(i-1) + gap, HV\,(i) + gap) \tag{5}$$

After the calculation of the current vector, the current anti-diagonal vector will be the horizontal-vertical reference of the next anti-diagonal vector, and the horizontal-vertical reference vector will be the diagonal reference vector. Fig. 7 shows the vector transactions in *fill* step of the parallel approach. Blue denotes the current anti-diagonal, red denotes the horizontal-vertical reference, and green denotes the diagonal reference. Dark grey cells are visited cells and light grey cells are to be visited cells.

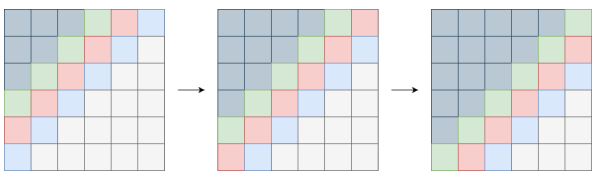

Fig. 7. Parallel approach of final score calculation.

1) Iterative Kernel Launch: This approach requires iterating over each anti-diagonal of the matrix individually. Because of each value of a cell in an anti-diagonal will be calculated simultaneously. The time complexity of this approach will be equal to the iteration count. For a matrix with size of n rows and m columns, the total iteration count $n+m+1$. That makes the time complexity of the parallel approach $O(n)$.

At the beginning of the each iteration, the host side aligns the grid and the block dimensions for the kernel launch. The kernel fills the anti-diagonal vector. After the kernel launch, the host side rearrange the anti-diagonal vectors for the next launch. Fig. 8 shows the iterative process for the kernel launch.

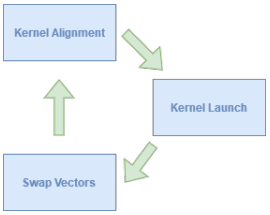

Fig. 8. Iterative kernel launch process.

2) Cooperative Kernel Launch: CUDA execution model does not have a default inter-block synchronization mechanism. Therefor, synchronization must be done explicitly. One way to do that is calling the kernel iteratively. On each kernel invocation, the dedicated grid fills the anti-diagonal vector. Synchronization of threads and blocks is done by the device implicitly and no extra effort is required in this method, yet it costs kernel launch overhead. *Cooperative Groups* allows synchronization of a group of threads in a CUDA program. By launching the kernel with Cooperative Groups, it is possible to synchronize blocks inside the CUDA program. By using Cooperative Groups, it is possible to reduce the kernel launch overheads.

3) Anti-Diagonal Major Ordering: Multi-dimensional arrays are represented in row-major or column-major order. That makes traversing the score matrix row by row or column by column easier for serial approach. However, the classical ordering methods do not suit for parallel score matrix calculation. In parallel calculation methods, every anti-diagonal vector of the score matrix is filled in an iteration. Consecutive elements of an anti-diagonal vector are not stored in contiguous physical memory. Fig. 9 shows an anti-diagonal vector representation in physical memory with row-major ordering.

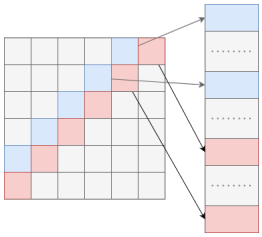

Fig. 9. Memory representation of an anti-diagonal vector with row-major ordering.

As it seen in the figure, two consecutive values for an anti-diagonal vector, M(i, j) and M(i + 1, j - 1) can be physically apart from each other. This might cause memory faults. In addition to that, it is impossible to copy anti-diagonal vector directly to the score matrix that stored on the host. Every value in the anti-diagonal vector must be distributed to corresponding physical memory locations. That brings extra complexity to the implementation.

To overcome this problem the alignment matrix can be represented on the memory in anti-diagonal order. In this representation, consecutive physical addresses in the memory correspond to consecutive values from an anti-diagonal vector. Fig. 10 shows the anti-diagonal major ordering.

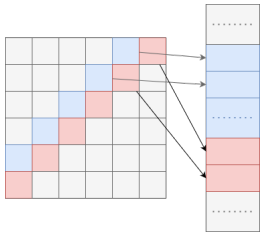

Fig. 10. Memory representation of an anti-diagonal major ordering.

That representation allows to direct copying an anti-diagonal vector that calculated on the device to the host. Because of reduce copying complexity, cache misses, and page faults, it is expected to improve performance and reduce the execution time.

4) Dividing Alignment Matrix: Executing the program with single launch is impossible for the alignmet matrices that their size is bigger than the GPU's physical memory. Iterative kernel launch method might not be wanted because of its kernel launch and data transfer overheads. In that case, the score matrix can be divided into submatrices and each submatrix can be generated then transferred to the host after the kernel execution.

By representing the score matrix in anti-diagonal major order, it is possible to divide the score matrix in anti-diagonal manner and transfer each submatrix directly from the host to the device. The goal of method is dividing the score matrix to submatrices with as equal amount of element as possible that maximizes the data transfer throughput. With minimum submatrix count, the kernel launch overhead can be minimized by using Cooperative Groups. Fig. 11 shows the submatrices of the score matrix. Each color represents different submatrices that is filled in the device side then transferred back to the host after the kernel execution.

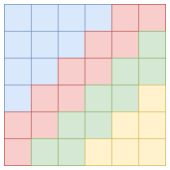

Fig. 11. Dividing the score matrix into sub-matrices.

V. Experimental Setup

We generated random genetic sequences that contain all 20 amino acids with the length started by 10 and gradually incremented to 100,000 by a multiple of 10. Then another set of sequences was generated with the length started by 10,000 and gradually incremented to 100,000 by step of 10,000 for the detailed performance comparison of parallel implementations.

Each computation was repeated 10 times with different sequences for the same length and the average of 10 executions were taken as final execution time to statistically correct results. All tests were run on a computer with a 2.60 GHz Intel Core i7-9750H CPU, 2666 MHz 32 GB RAM, Nvidia GTX 1650 with 4 GB DRAM, and 16 Multiprocessor.

VI. Results

For the performance evaluation, calculating the final score and generating the alignment matrix are executed separately. For the final score calculation, data transfer from the GPU to the host is not necessary. The bottom-right cell is taken as the final score. Because of the absence of data transfer, the submatrix approach is not necessary for the final score calculation.

Implicit thread-block synchronization by launching the kernel iteratively and explicit synchronization with Cooperative Groups was tested for datasets between 10-100,000 and 10,000-100,000 separately. Table I shows the results of the overall comparison and Table II shows the results of the detailed comparison of the final score calculations.

TABLE I
The Final Score Calculation Times.

Sequence Length	Serial (ms)	Iterative (ms)	Cooperative Groups (ms)
10	0	1	1
100	0	1	1
1,000	6	7	5
10,000	636	61	38
100,000	63698	1712	1194

TABLE II
Detailed Comparison of the Final Score Calculation.

Sequence Length	Serial (ms)	Iterative (ms)	Cooperative Groups (ms)
10,000	636	61	38
20,000	2542	127	75
30,000	5724	212	127
40,000	10182	315	194
50,000	15920	434	276
60,000	22913	595	384
70,000	31213	781	519
80,000	40472	1021	702
90,000	51577	1357	925
100,000	63698	1712	1194

The two comparisons show, for sequences with hundreds of amino acids, the serial approach is as effective as parallel approaches. For sequences with thousands of amino acids, the quadratic complexity of the serial approach makes it slower when compared with the parallel approaches. Fig. 12 shows the graph of the serial and parallel execution times.

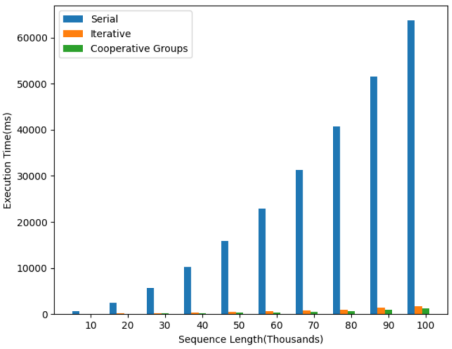

Fig. 12. The final score calculation.

Fig. 13 and Fig 14 show the NVIDIA Visual Profiler results of the iterative approach and Cooperative Groups respectively. In both approaches, data transfer times are nearly equal. The overhead of launching Cooperative Groups is higher than standard kernel launch overhead. However, using Cooperative Groups reduces the total kernel calls. As a result, the total execution time reduces.

```
    Time      Calls       Avg       Min       Max  Name
69.6698s   11000100  6.3330us  1.7600us  33.537us  nw_cuda_score
1.5914ms        700  2.2730us       544ns  9.8240us  [CUDA memcpy HtoD]
80.321us        100       803ns       576ns  1.0240us  [CUDA memcpy DtoH]
80.5634s   11000100  7.3230us  3.7000us  26.230ms  cudaLaunchKernel
```

Fig. 13. Profiler results of the final score calculation with iterative approach.

```
    Time  Calls       Avg       Min       Max  Name
43.6034s    100  436.03ms  31.374ms  1.18728s  nw_cuda_score
1.5869ms    700  2.2660us       544ns  20.768us  [CUDA memcpy HtoD]
68.608us    100       686ns       576ns       960ns  [CUDA memcpy DtoH]
4.4185ms    100  44.185us  28.400us  705.70us  cudaLaunchCooperativeKernel
```

Fig. 14. Profiler results of the final score calculation with Cooperative Groups.

For the alignment matrix generation, each anti-diagonal vector data must be transferred from the GPU to the host. In this comparison, the iterative method by calling the kernel for each anti-diagonal vector and transferring that vector at the end of the kernel, dividing the matrix into submatrices and calling the kernel by default, and calling by Cooperative Groups methods were tested. Table III shows the results of the overall comparison and Table IV shows the results of the detailed comparison of the alignment matrix generations.

TABLE III
The Alignment Matrix Generation Times.

			Submatrix	
Sequence Length	Serial (ms)	Iterative (ms)	Standard (ms)	Cooperative Groups (ms)
10	0	2	1	1
100	0	14	1	1
1,000	8	123	8	6
10,000	901	1243	105	79
100,000	89767	17614	5853	5404

TABLE IV
DETAILED COMPARISON OF THE ALIGNMENT MATRIX GENERATION.

Sequence Length	Serial (ms)	Iterative (ms)	Submatrix	
			Standard (ms)	Cooperative Groups (ms)
10,000	901	1243	105	79
20,000	3599	2657	305	249
30,000	8096	3970	605	517
40,000	14415	5468	1013	890
50,000	22570	7044	1514	1347
60,000	32528	8782	2112	1896
70,000	44268	10706	2822	2566
80,000	57778	12935	3698	3401
90,000	72760	14867	4693	4342
100,000	89767	17164	5853	5404

The two comparisons show, for sequences with hundreds of amino acids, the serial approach is as effective as submatrix approaches and more effective than the classical iterative kernel approach. For sequences with thousands of amino acids, submatrix approaches are obviously more effective, and the iterative approach becomes more effective as the sequence length grows. Fig. 15 shows the graph of the serial and parallel execution times.

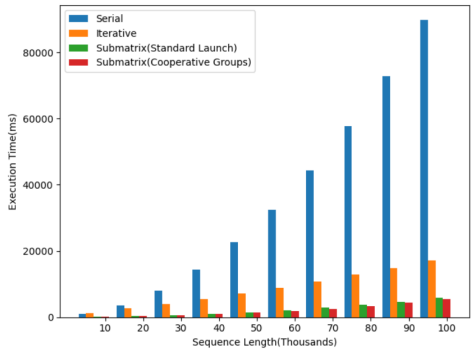

Fig. 15. Detailed comparison of the alignment matrix generation.

Fig. 16, Fig. 17, and Fig. 18 show the NVIDIA Visual Profiler results of the iterative approach, submatrix with standard kernel launch, and Cooperative Groups respectively. Partitioning the alignment matrix reduces the total *cudaMemcpy* calls. That reduces the total execution time as well. By using Cooperative Groups, further performance improvement was obtained by reducing the total kernel calls.

Time	Calls	Avg	Min	Max	Name
90.5282s	11000100	8.2290us	2.4320us	29.536us	nw_cuda_fill
32.8739s	11000200	2.9880us	416ns	23.712us	[CUDA memcpy DtoH]
1.6423ms	700	2.3460us	1.1200us	9.6000us	[CUDA memcpy HtoD]
833.353s	11000400	75.756us	48.500us	7.6498ms	cudaMemcpy
144.456s	11000100	13.132us	11.700us	1.9875ms	cudaLaunchKernel

Fig. 16. Profiler results of alignment matrix generation with iterative approach

VII. CONCLUSION AND FUTURE WORK

In this research, the massively parallel architecture of GPU is used to increase global sequence alignment algorithm called the Needleman-Wunsch. For the final score calculations, it is observed that with the default kernel launching mechanism the performance of the algorithm increased up to 40x, and with Cooperative Groups, it is increased up to 60x Because of reduced kernel launch overhead.

For the alignment matrix generations, it is observed that the performance increased up to 5x for the iterative method, up to 15x with default kernel launch, and up to 17x with Cooperative Groups. There is a slight performance difference between default method and Cooperative Groups, but because of the data transfer overhead is much more than the kernel launch overhead, the speed up can be negligible for the matrix generation.

From the results of this work, it is observed that the main actor of the execution time of the Needleman-Wunsch algorithm is data transfer overhead. The possible limiter for aligning relatively long sequences can be the data size of the alignment matrix. Scaling the growing data size with distributed architecture can be considered as a part of our future work.

Time	Calls	Avg	Min	Max	Name
64.6387s	11000100	5.8760us	1.6640us	30.561us	nw_cuda_fill
35.0528s	195210	179.56us	576ns	299.52us	[CUDA memcpy DtoH]
1.6071ms	700	2.2950us	1.1520us	9.6960us	[CUDA memcpy HtoD]
112.286s	195410	574.62us	38.400us	3.5159ms	cudaMemcpy
56.9199s	11000100	5.1740us	3.8000us	2.5034ms	cudaLaunchKernel

Fig. 17. Profiler results of alignment matrix generation with partitioning and standard kernel call.

Time	Calls	Avg	Min	Max	Name
39.0308s	195110	200.04us	136.07us	3.8859ms	nw_cuda_fill
35.6430s	195210	182.59us	576ns	251.78us	[CUDA memcpy DtoH]
1.6033ms	700	2.2900us	1.1520us	11.520us	[CUDA memcpy HtoD]
118.612s	195810	605.75us	600ns	4.5501ms	cudaMemcpy
3.11797s	195110	15.980us	14.600us	515.90us	cudaLaunchCooperativeKernel

Fig. 18. Profiler results of alignment matrix generation with partitioning and Cooperative Groups.

REFERENCES

[1] W. R. Pearson, D. J. Lipman, “Improved tools for biological sequence comparison,” Proceedings of the National Academy of Sciences, 85(8): 2444–2448, 1988.

[2] J. D. Thompson, D. G. Higgins, T. J. Gibson, “Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,” Nucleic acids research, 22(22): 4673–4680, 1994

[3] M. I. Irawan, I. Mukhlash, A. Rizky, A. R. Dewi, “Application of needleman-wunch algorithm to identify mutation in dna sequences of corona virus,” Journal of Physics: Conference Series, 1218(1), 2019

[4] Temple F. Smith, Michael S. Waterman, ”Identification of Common Molecular Subsequences”, Journal of Molecular Biology 147(3):195–197, 1981.

[5] Saul B. Needleman, Christian D. Wunsch, ”A general method applicable to the search for similarities in the amino acid sequence of two proteins”, Journal of Molecular Biology 48(3):443–53, 1970.

[6] David J. Lipman, William R. Pearson, ”Rapid and sensitive protein similarity searches”, Science 227(4693):1435–41, 1985.

[7] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, David J. Lipman, ”Basic local alignment search tool”, Journal of Molecular Biology 215(3):403-410, 1990.

[8] S. F. Altschul, T. L. Madden, A. A. Schaffer et al., “Gapped Blast and Psi-Blast : A new-generation of protein database search programs”, Nucleic Acids Res, 25: 3389-3402, 1997.

[9] S. Che, M. Boyer, J. Meng et al., “Rodinia: A benchmark suite for heterogeneous computing,” in IEEE International Symposium on Workload Characterization, 2009, 44-54.

[10] D. Li, M. Becchi, "Multiple Pairwise Sequences Alignments with NeedlemanWunsch Algorithm on GPU" in SC Companion: High Performance Computing, Networking Storage and Analysis, 2012
[11] T. R. P. Siriwardena, D. N. Ranasinghe, "Global Sequence Alignment using CUDA compatible multi-core GPU" in HIPC, 2010.
[12] D. Li, K. Sajjapongse, H. Truong, G. Conant and M. Becchi, "A distributed CPU-GPU framework for pairwise alignments on large-scale sequence datasets," International Conference on Application-Specific Systems, Architectures and Processors, 2013
[13] O. Selvitopi, S. Ekanayake, G. Guidi, G. Pavlopoulos, A. Azad, A. Buluç, "Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices", International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.
[14] NVIDIA CUDA C Programming Guide (Version 11.5), 2021