DOI: http://dx.doi.org/10.26483/ijarcs.v8i9.4953

ISSN No. 0976-5697

Volume 8, No. 9, November-December 2017

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A SHARED MEMORY BASED IMPLEMENTATION OF NEEDLEMAN-WUNSCH ALGORITHM USING SKEWING TRANSFORMATION

Vibha Patel
Information Technology Department
VGEC Chandkheda,
Ahmedabad, India

Krunal Gandhi and Darshak Bhatti
Department of Computer Science and Engineering,
Nirma University,
Ahmedabad, India

Abstract: Among various algorithms for protein and nucleotide alignment, Needleman-Wunsch algorithm is widely accepted as it can divide the problem into sub-problems. We present two parallel approaches of Needleman-Wunsch algorithm with single kernel and multi-kernel invocation using skewing transformation which is used for traversing and calculation of dynamic programming matrix. We also compare these with traditional CPU sequential approach which resulted in six fold performance improvement. Furthermore, we present same single kernel ideology on shared memory which resulted in two fold performance improvement over non-shared memory approach.

Keywords: Skewing Transformation; CUDA; Single Kernel; Multi-Kernel; Shared Memory; lock-free synchronization

1. INTRODUCTION

Genomics is a course of study in the field of genetics which deals with genomes. Advances in genomics can help us to understand complex biological phenomenon which in turn can help us in prognosis and diagnosis of various diseases. Bioinformatics which is a part of genomics combines various disciplines such as computer science, statistics and mathematics to elucidate and analyze biological data. In bioinformatics, sequence alignment is a method which compares two or more sequences and finds nearly identical areas or identical nucleotide of DNA, RNA or Protein to find the similarities or relationship between two given sequences. Many bio-informatics tasks like, predicting biological function, constructing evolutionary trees, detecting point mutations, classifying genes and proteins, secondary and tertiary protein structure and other prognosis and diagnosis methods depend upon successful alignment. If the sequence length is small then it is possible to align sequences by human effort. However, for longer sequences, it is difficult to align manually. Hence, computational sequence alignment algorithms are developed by researcher to deal with longer sequences.

There are three main categories of computational sequence alignment algorithms: (1) Global sequence alignment [1] (2) Local sequence alignment [2] and (3) Hybrid or Semi-Global sequence alignment [3]. Global alignment method attempts to align every nucleotide and it is usually used when sequence lengths are of approximately same length. Local alignment is used when sequences are unalike but are supposed to contain similar regions within long sequence. On the other hand if end of a sequence overlaps with the beginning of other sequence then hybrid alignment is used because global alignment method attempts to extend the alignment past the overlapping region. Whereas, local alignment might fail to cover the whole overlapping region.

Several computational algorithms are developed for the sequence alignment problem. They generally use the concepts of dynamic programming, heuristic algorithm and probabilistic methods. From all the approaches, dynamic programming based implementations are more time consuming than heuristic based implementations. However, dynamic programming based approach provides a more accurate solution as compared to heuristic based methods. Needleman-Wunsch and Smith-Waterman algorithms are two widely used dynamic programming based approaches. Needleman-Wunsch is used for global sequence alignment and Smith-Waterman is used for local sequence alignment. The detailed discussion of the algorithm used for extension in shared memory implementation is presented in [5]. We summarize the steps of the algorithm which are as follows:

1. ***Initialization:*** This involves construction of Dynamic programming matrix *(D)* with N + 1 rows and M + 1 column. Where N and M are lengths of the sequences to be aligned. We fill the first row and column initially with distance from origin multiplied by GAP value.
2. ***Matrix Fill:*** Fill all other (i, j) cells from the values of (i-1, j), (i, j-1) and (i-1, j-1). Initialize trace-back matrix according to the selected value.
3. ***Trace-back:*** (M, N) cell contains the maximum score and it is the cell from where we begin to trace-back. We follow arrows determined in trace-back matrix and reach the first cell. Hence, we get the path which represents the best alignment. We also put the values of the GAP according to the direction traversed in the matrix into the new sequence that we generate during trace-backing.

At the end reverse both sequences to get final aligned sequences.

With the advent of Compute Unified Device Architecture [4] which is programming interface provided by Nvidia, use of GPUs for general purpose programming has increased. Here we try to utilize computing capabilities of GPU for non-graphics bioinformatics application. Our work focuses on parallelization of Needleman-Wunsch algorithm using skewing transformation on CUDA enabled GPU. We also present implementation of same approach using shared-memory.

Global memory in GPU is an off-chip device memory which is usually larger in size with life until the application closes or it is freed explicitly. It is visible to all the threads and blocks which have a pointer to the memory region. Shared memory on the other hand is on-chip memory. Due to high capabilities, it is usually smaller in size depending on the device. The visibility of shared memory is restricted to only threads within the same block. Shared memory is magnitudes faster to access than global memory and acts like a local cache shared among the threads of a block. Here we devise a method to effectively utilize the shared memory for our approach.

In section 2, we describe the approach for parallelizing the algorithm using skewing transformation. In section 3, we describe how to use CUDA enabled GPU to improve performance and reduce the time for execution. In section 4, we compare the performance of sequential CPU based implementation with two parallel GPU based implementations and shared memory implementation. We show the effectiveness of our implementations in section 4. In this paper we present a shared memory based approach. The related work is presented in section 5 and then we summarize our work.

2. PARALLEL APPROACH

Each value *(i, j)* in the dynamic programming matrix *(D)* is dependent on three values: *(i-1, j)*, *(i, j-1)* and *(i-1, j-1)*. The dependence relation of the matrix is shown in the Figure 1 and to execute this in parallel we need to calculate the values in anti-diagonal order.

The row major order of calculation in each iteration is shown in Table I. In first iteration only *(1, 1)* will be calculated, in second iteration *(1, 2)* and *(2, 1)* will be calculated in parallel, in third iteration *(1, 3)*, *(2, 2)* and *(3, 1)* will be calculated in parallel and so on. There is no scope for *(1, 1)* and *(8, 8)* to be executed in parallel, as the value of *(8, 8)* depends on *(8, 7)* and *(7, 8)*.

It is evident from Table I that in the eighth iteration maximum parallelism can be achieved. To calculate all the values, we apply skewing transformation on the original iteration space. After applying skewing transformation, the original iteration space as shown in Figure 1(a), gets transformed to the one shown in Figure 1(b). Each iteration with same numbers indicates the elements which can be executed in parallel. However, each parallel section separated by dotted lines indicates requirement of synchronization of the iteration space. The concept of loop skewing and block synchronization is as discussed in [5].

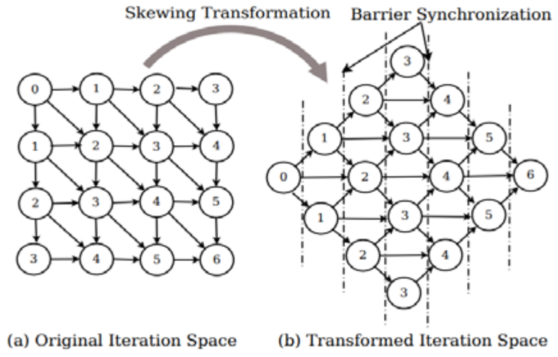

Figure 1. Original and Transformed iteration space

A. *Impementation Approaches*

The parallel approach is implemented using both, lock-based and lock-free mechanism as shown in Figure 2. We briefly discuss the approaches in the following subsection.

*1) **Lock Based Approach:*** In lock-based synchronization approach a global mutex variable is created to count the number of thread blocks that reach synchronization point. Mutex is incremented by 1 each time a block completes its execution. Then the value of mutex is compared with the target value repeatedly. After synchronizing each thread block, execution can move to next phase. Here the value of goal is set to number of blocks in the kernel when the barrier is invoked first. This value is then incremented by *N* each time barrier is invoked. This approach is easier and efficient than resetting mutex each time after completion of a barrier invocation because it reduces the number of instructions and prevents conditional branching.

Table I: Initial Matrix

1	1,1							
2	1,2	2,1						
3	1,3	2,2	3,1					
4	1,4	2,3	3,2	4,1				
5	1,5	2,4	3,3	4,2	5,1			
6	1,6	2,5	3,4	4,3	5,2	6,1		
7	1,7	2,6	3,5	4,4	5,3	6,2	7,1	
8	1,8	2,7	3,6	4,5	5,4	6,3	7,2	8,1
9	2,8	3,7	4,6	5,5	6,4	7,3	8,2	
10	3,8	4,7	5,6	6,5	7,4	8,3		
11	4,8	5,7	6,6	7,5	8,4			
12	5,8	6,7	7,6	8,5				
13	6,8	7,7	8,6					
14	7,8	8,7						
15	8,8							

*2) **Lock Free Approach:*** In lock-free approach, the mutex is incremented by an atomic function. This serializes the incrementation of mutex variable, despite the fact that all the operations are performed in separate blocks. Here we implement lock-free synchronization without having atomic operations. The concept behind this method is to dedicate a sync variable to each individual thread block, so that each block can track its sync status without committing the global mutex variable, thus preventing dead lock.

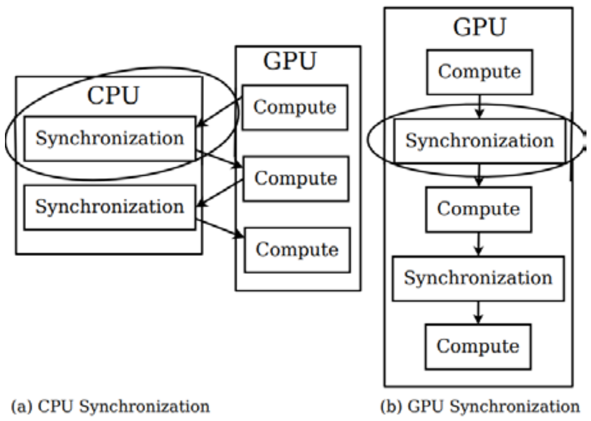

Figure 2. Block Synchronization with CPU and GPU

Figure 3.

*3) **Shared Memory Approach**:* In our shared memory approach first, we divide the dynamic programming matrix

(D) in chunks and then copy these chunks to shared memory. After that skewing transformation is applied so that computation can be done in parallel. The computation results are then copied back to original dynamic programming matrix. Reading and writing overhead is not significant over here as shared memory is much faster than global memory. We elaborate this approach in next section.

3. IMPLEMENTATION

After applying skewing transformation, we can parallelize NW by multiple kernel invocations at the point of block synchronization. The alternative is to use the single kernel call implementation using block synchronization approach. Here, each block in the GPU needs to synchronize the threads using syncthreads(). The block synchronization within single kernel call can use the methods described in subsection 2A. We use Lock-free implementation of the block synchronization as it gives better performance than lock-based approach.

B. Non Shared (Global) Memory Implementation

In this implementation, the GPU kernel is launched with selected number of threads per block. The number of blocks is function of sequence length and number of threads per block *f(sequence_length, number_of_threads)*. To simplify the process, we launch threads and blocks in only *x* direction. The calculation involves comparison of both sequences; hence we pass both the sequences to the kernel at launch time. The dynamic programming matrix *(D)* and trace-back matrix *(T)* is determined using Algorithm 1. It makes dependent looping structure easier to parallelize, and hence gives performance improvement when implemented on GPU.

The calculation of the dynamic programming matrix *(D)* in a single block is shown in the Algorithm 2. The same logic is applicable to all the blocks in the grid, while applying the algorithm for large sequence length. In that case, we require the use block_synchronization() instead of thread_synchronization(). This block synchronization makes use of lock-free approach as discussed in [5].

Algorithm 1: Parallel Needleman-Wunsch Algorithm

```
Input: Dynamic Programming Matrix (d_mat), Sequence 1 (d_seq1), Sequence 2 (d_seq2), No. of Threads/Block, No. of Blocks
Output: Updated Dynamic Programming Matrix (d_mat)
thread_id = calculate_thread_id_in_block( )
block_size = calculate_block_dim( )
d_mat[thread_id] = thread_id * GAP
d_mat[thread_id * block_size] = thread_id * GAP
for i = 0 → block_size do
    row = thread_id
    col  = i – thread_id
    if  thread_id ≤ i  and row ≠ 0 and col ≠ 0 then
       left = d_mat[row * block_size + col  - 1] + GAP
       top= d_mat[(row-1)*block_size + col] + GAP
       if d_seq1[row – 1] == d_seq2[col-1] then
          dia = d_mat[(row-1)*block_size+col–1] + MATCH
       else
          dia=d_mat[(row-1)*block_size+col-1] + MISMATCH
       d_mat[row*block_size + c] = max(t, top, dia)
    thread_synchronization()
for  j = 0 → block_size do
    row = block_size – 1 – thread_id + j
    col = thread_id
    if thread_id  ≤ block_size then
       left = d_mat[row * block_size + col – 1] + GAP
       top = d_mat[(row-1)*block_size + col] + GAP
       if d_seq1[row-1] = d_seq2[col-1] then
          dia = d_mat[( row - 1) * block_size  +  col – 1]  +
                MATCH
       else
          dia = d_mat[(row – 1) * block_size + col – 1] +
                MISMATCH
       d_mat[row * block_size + c] = max(left, top, dia)
    thread_synchronization()
return
```

C. Shared Memory Implementation

As shared memory is very limited we need to use it prudently. First we create a shared memory of 32 x 32 size. Then we transfer dynamic programming matrix *(D)* in chunks of 32 x 32 into shared memory. Here we apply the single kernel lock free approach as discussed in previous section. After computation resulting matrix is transferred back to dynamic programming matrix. This process is repeated until whole dynamic programming matrix *(D)* is computed

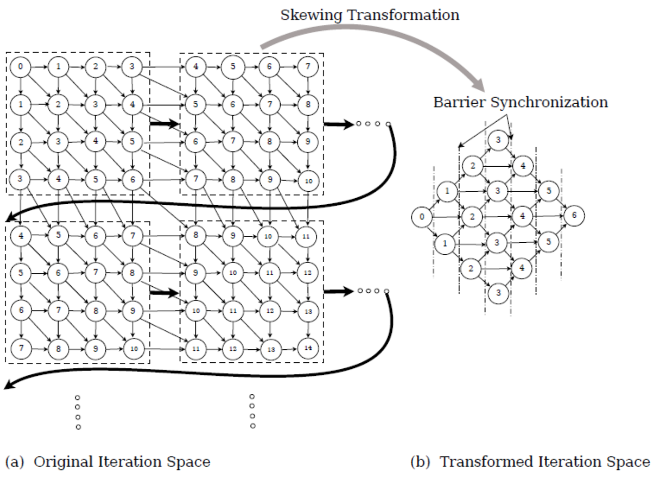

Figure 3: Original and Transformed Iteration space of Shared Memory Approach

Algorithm 2: Dynamic Programming Matrix Calculation (Shared Memory)

```
Input: Dynamic Programming Matrix (d_mat), Sequence 1 (d_seq1), Sequence 2 (d_seq2), No. of Threads/Block, No. of Blocks
Output: Updated Dynamic Programming Matrix (d_mat)
bx = blocks_in_x_dimension
tx = threads_in_x_dimensions
ty = threads_in_y_dimension
beg = length(d_seq1) * 32 * bx
end = beg + length(d_seq1)
step = 32
for a = beg → end  do
    _shared_ int  s[32*32], t[32*32]
    _shared_ char seq1[32], seq2[32]'
    sync_threads()
   /* Copy sequences from global to shared memory*/
```

```
size = blocks_in_x_dim * grids_in_x_dim
grid_size = grid_dim_x
blk_size = block_dim_x
blk_id = block_id_x
thread_id = calculate_thread_id_in_block()
block_size = calculate_block_dim()
s[thread_id] = thread_id * GAP
s[thread_id * block_size] = thread_id * GAP
for i = 1 → block_size do
    row = thread_id
    col = i – thread_id
    if thread_id ≤ i and row ≠ 0 and col ≠ 0 then
        left = s[row * block_size + col – 1] + GAP
        top = s[(row – 1) * block_size + col] + GAP
        if seq1[row – 1] = seq2[col – 1] then
            dia =s[(row–1)*block_size + col–1] + MATCH
        else
            dia=s[(row-1)*block_size+col-1] + MISMATCH
        s[row * block_size + col] = max(left, top, dia)
        sync_threads()
d_mat[a + length(seq1) * ty + tx] = s[ty * 32 + tx]
for w = 0 → 32 do
    d_traceback[((n*32) + tx)*length(seq1)+(n*32) + w] =
                                        t[tx * 32 + w]
return
```

Shared memory is made up of 32 memory banks and it is necessary to perform synchronization. We use a stride to copy values in matrix as it does not lead to bank conflicts. The algorithm of the same is shown in Algorithm 2.

As shown in the Figure 3, matrix is divided into sub matrices of 32 x 32 and copied to shared memory for computation. Arrows indicates the flow of execution.

4. EVALUATION AND DISCUSSION

In this section we present the evaluation results and discuss them in detail. The execution of sequential version of the code is verified with Intel® Core™ i3 CPU with 6 GB of RAM and parallel implementations are verified with Tesla C2070 GPU containing 448 CUDA cores and 5376 MB of global memory for storage. We also verified our approaches with Intel® Xeon™ CPU with 16 GB of RAM and parallel versions are verified with Tesla K40c GPU containing 2880 CUDA cores and 12 GB of global memory storage. The execution time of a program on the GPU consists of 3 different phases:

1) *Time to launch the kernel on the GPU*
2) *Computation done on the GPU*
3) *Inter block GPU communication using Block Synchronization*

A. Results

We compared the execution time of this algorithm using three different approaches: Sequential (CPU), Parallel GPU based implementation with multiple kernel invocation from CPU and Parallel GPU based implementation with single kernel invocation using lock-free block synchronization. Comparison of the time taken for the execution by these three methods is shown in Figure 4. Figure 5 shows the comparison of time taken with different block size i.e. 128 x 128, 256 x 256, 512 x 512 and 1024 x 1024.

B. Discussion

As shown in Figure 4, it is evident that both parallel implementation with single kernel invocation and multiple kernel invocation perform better than the CPU based sequential implementation. These results are compared with different input sequence lengths. It is observed from the graph that with the increase in sequence length the speedup of both the parallel approaches increases.

Figure 5 shows the execution time comparison of GPU implementation with different block sizes. From the figure it is clear that if we increase the block size the performance drops slightly. Non-coalesced memory access and increase in page faults contributes in reduction of performance with the increase in number of threads per block. The device coalesced global memory loads the values into DRAM in row-wise or column-wise manner and as we are accessing the values anti-diagonally it results in increase in page faults and mismatches during memory access. This leads to performance drop and thus we obtain maximum performance with block size of 128 x 128. This also depends on the GPU architecture and memory access mechanism of the GPU, causing different optimum block sizes for different GPUs.

Figure 6 shows the speed-up of parallel GPU based single kernel invocation and multiple kernel invocation with respect to sequential method on CPU with icc and Figure 7 shows the speed-up of parallel GPU based single kernel invocation and multiple kernel invocation over sequential implementation compiled with g++ on CPU. GPU based single kernel implementation with lock-free synchronization gives the same speed-up as multiple kernel invocation with CPU based block synchronization. As the sequence length increases the speed-up also increases. We obtained speed-up of ~5.5 for the sequence
length of 32k with g++ and 2 for sequence length of 32k with icc. One observation which can be made from the performance plot is increase in the input sequence length results in increase in the speed-up gained.

As shown in Figure 6 and 7 single-kernel and multi-kernel invocation give same performance hence we have implemented only multi-kernel implementation using shared memory. From Figure 9 it is apparent that shared memory performs ~9 times better than sequential implementation with g++ and of ~3 times better than sequential implementation with icc for sequence length of 32k. Figure 8 shows that our shared memory implementation on GPU is 2.2 fold faster than non shared memory based implementation.

On GPU device shared memory has bandwidth of *1.5 TB/s* and global memory has bandwidth of *150 GB/s.* Theoretically this means that shared memory should perform 10 times faster but due to our execution flow it does not yield that kind of results in actual scenario.

Suppose memory operation on global memory takes Tg time and on shared memory it takes Ts time hence Tg =10 x Ts. Now suppose there is matrix of 64 x 64. So it will require 64+64 Gap value fills. Now each value is compared for match and mismatch. Hence

Total_Comparisons = 64 x 64 = 4096

As each cell (i, j) is dependent on (i-1, j), (i, j-1) and (i-1, j-1). So,

Total_reads = 4096 x 3

The last total 4096 write operations will be performed to fill whole matrix. Therefore incorporating all these results in O(20736 x Tg). Now for shared memory we apply it on submatrix of 32 x 32. Whole calculation is done as above and we will have O(5184 x Ts).

In addition, we are reading pairs from global to shared memory and writing the whole matrix back to global memory when computed. Therefore our shared memory execution will be O(5184 x Ts + 1088 x Tg). This is for one submatrix, to compute whole matrix we have to compute 4 submatrix, consequently our execution will be O(20736 x Ts + 4352 x Tg).

Now as we know that Tg =10 x Ts putting these value we get shared memory execution complexity as O(64256 x Ts) and global memory execution complexity as O(207360 x Ts). This means speed up of 3 times in an ideal situation. But due to execution dependencies this algorithm does not produce realistic results. In skewing transformation as the size increases, more parallelization can be exploited which is evident from results. Initially speed up is not much but as we increase the sequence length it increases and we have got speed up of 2.3 for sequence length of 32k.

5. RELATED WORK

Needleman-Wunsch [1] and Smith-Waterman [2] are two well known dynamic programming based algorithms developed in the 70s and early 80s to detect similarity between a pair of DNA/protein sequences. BLAST [6] is the most commonly used sequence alignment program for a pair wise alignment. It is based upon the principle of hashing small matching sequences and then extending the hash matches to create high-scoring segment pairs until the highest possible score is obtained. BLAST is faster than any dynamic programming based approach. However, it does not guarantee the optimal alignment of the query and database as dynamic programming.

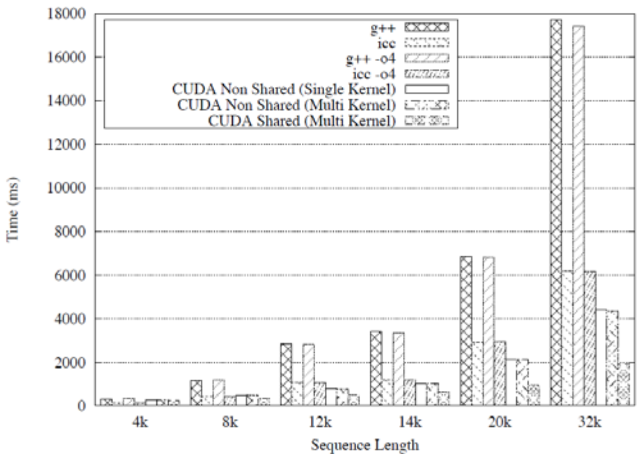

Figure 4: Execution time of CPU and GPU implementation

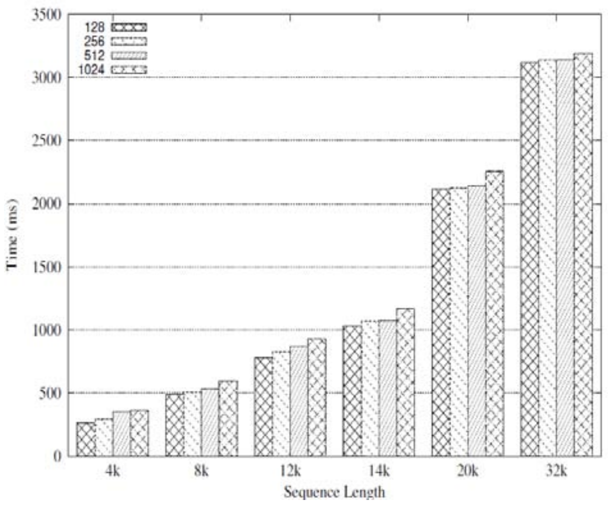

Figure 5: GPU (Shared Memory Execution time for varying block size

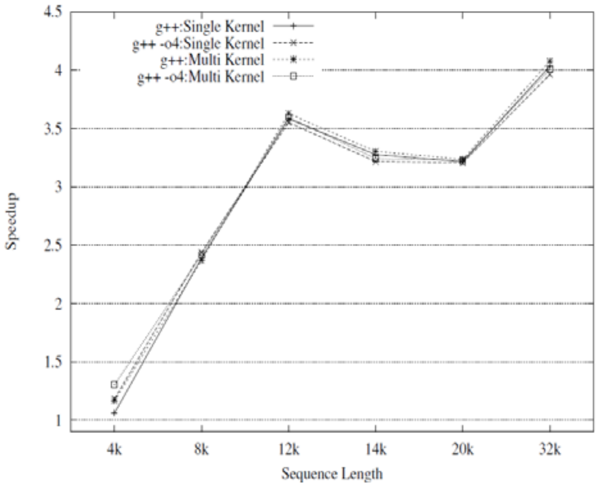

Figure 6: Speedup of Single Kernel/Multi-Kernel Non-Shared
GPU over CPU (icc)

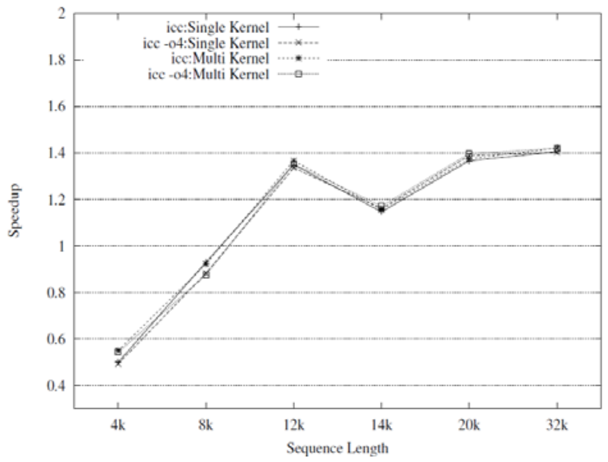

Figure 7: Speedup of Single Kernel/Multi-Kernel Non-Shared
GPU over CPU (g++)

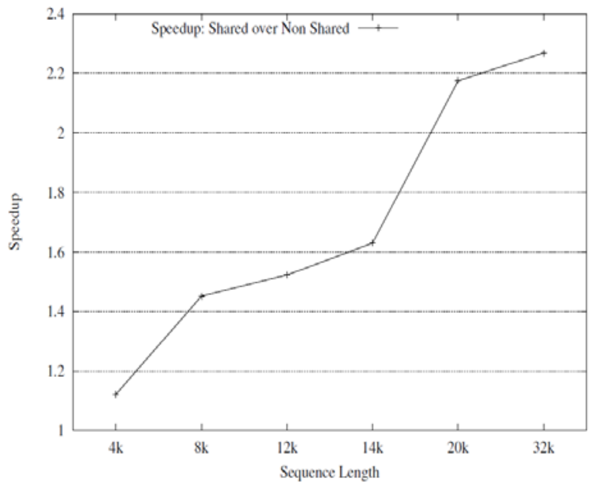

Figure 8: Speedup of Shared Memory over Non-Shared Memory

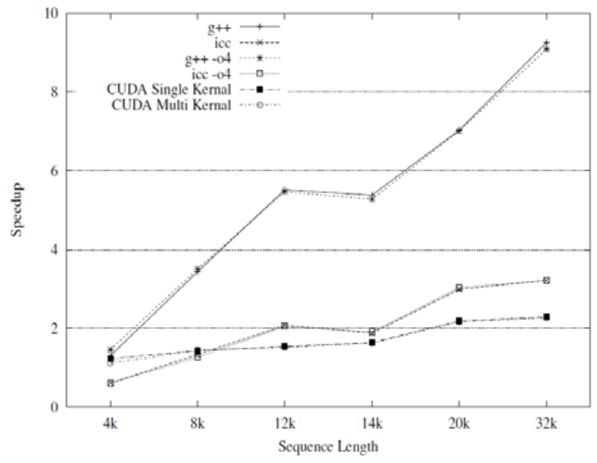

Figure 9: Speedup of Shared Memory Multi-Kernel

CS-BLAST [7] a protein sequence search tool is an extension of BLAST, which is based on context-specific mutation probabilities. Several researchers have developed parallel versions of the Smith-Waterman algorithm that are suitable for Graphics Processing Units (GPUs) [8], [9], [10], [11], [12]. Zheng et. Al. [13] introduced a metric based approach to estimate the performance of compute-bound GPU kernels with control flow divergence. The thread re-grouping algorithms further make use of the metric based value function.

An efficient GPU based implementation of Multiple Sequence Alignment is given by Liu et. al. [14]. They reformulated the compute intensive stage of CLUSTAL-W, so that it suits the GPU architecture. It involves parallelizing the Needleman-Wunsch algorithm. An efficient implementation of Needleman Wunsch algorithm on graphics processing unit is also presented in [15]. Our approach differs from the one presented in [15] by the use of lock free and lock based approaches for block synchronization on GPU. Our approach for parallelizing the Needleman-Wunsch algorithm differs by using skewing transformation on the original data access pattern to exhibit the inherent parallelism existing in the code.

A shared memory implementation of Needleman-Wunsch is presented in [16] by Shivaram Venkataraman, Reza Farivar, Harshit Kharbanda, Roy Campbell for pairwise alignment. They have modified the original NW algorithm to make it two pass process. In first pass original dynamic programming matrix is divided into quadrants by computing only boundary values of quadrants using original NW algorithm and in second pass all these quadrants are processed in shared memory simultaneously. The results are very impressive.

Another shared memory approach presented by Siriwardena and Ranasinghe in [17] is improvement over the sequential approach up to 4.2 times. It uses blocking strategy in minor diagonals which copies minor diagonal blocks in shared memory and computes the results and copies back to global memory. Here they have used barrier synchronization in shared memory for threads. Our approach differs from this with skewing transformation which changes iteration space to improve performance in parallel.

6. CONCLUSION

In this research, we used CUDA enabled GPU to improve the performance of the Needleman-Wunsch algorithm. Although, the data level parallelism in Needleman-Wunsch algorithm is low, the data dependencies are such that skewing transformation technique is used to solve anti-diagonal dependencies. Using this approach, we achieved a speed-up of ~6 using multiple kernel GPU implementation as compared to CPU based implementation. The single kernel, lock-free block synchronization technique gave a speed-up of 6 over CPU based implementation. The speed-up increases with the increase in the sequence length. Shared memory implementation gave speed up almost double of our GPU implementation for sequence length more than 12k. Our CPU results of Intel C Compiler (icc) gave 3 times speed up compared to CPU sequential code. This result clearly acknowledges the effective use of GPU hardware for computation.

REFERENCES

[1] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” Journal of molecular biology, vol. 48, no. 3, pp. 443–453, 1970.

[2] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” Journal of molecular biology, vol. 147, no. 1, pp. 195–197, 1981.

[3] A. Layeb, S. Meshoul, and M. Batouche, “A hybrid method for effective multiple sequence alignment,” in . IEEE Symposium on Computers and Communications, 2009. ISCC 2009, pp. 970–975.

[4] “NVIDIA CUDA Programming Guide, Version 4.2.” [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA C Programming Guide.pdf

[5] A. Chaudhary, D. Kagathara, and V. Patel, “A gpu based implementation of needleman-wunsch algorithm using skewing transformation,” in Eighth IEEE International Conference on Contemporary Computing (IC3), 2015, pp. 498–502.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” Journal of molecular biology, vol. 215, no. 3, pp. 403–410, 1990.

[7] C. Angerm uller, A. Biegert, and J. Soding, “Discriminative modelling of context-specific amino acid substitution

probabilities," Bioinformatics, vol. 28, no. 24, pp. 3240–3247, 2012.

[8] S. A. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," BMC bioinformatics, vol. 9, no. 2, p. 1, 2008.

[9] L. Ligowski and W. Rudnicki, "An Efficient Implementation of Smith Waterman Algorithm on GPU Using CUDA, for Massively Parallel Scanning of Sequence Databases," in Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, ser. IPDPS '09, 2009, pp. 1–8.

[10] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the smith–waterman algorithm using single and multiple graphics processors," Journal of Computational Physics, vol. 229, no. 11, pp. 4247–4258, 2010.

[11] A. Khalafallah, H. F. Elbabb, O. Mahmoud, and A. Elshamy, "Optimizing smith-waterman algorithm on graphics processing unit," in 2nd IEEE International Conference on Computer Technology and Development (ICCTD), 2010, pp. 650–654.

[12] J. Li, S. Ranka, and S. Sahni, "Pairwise sequence alignment for very long sequences on gpus," International Journal of Bioinformatics Research and Applications, vol. 10, no. 4-5, pp. 345–368, 2014.

[13] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate gpu performance model for effective control flow divergence optimization," in IEEE 26th International Symposium on Parallel & Distributed Processing Symposium (IPDPS), 2012. IEEE, 2012, pp. 83–94.

[14] W. Liu, B. Schmidt, G. Voss, and W. M uller-Wittig, "Gpu-clustalw: using graphics hardware to accelerate multiple sequence alignment," in High Performance Computing - HiPC 2006, Springer, 2006, pp. 363–374.

[15] C. S. Khaladkar, "An efficient implementation of needleman wunsch algorithm on graphical processing units," Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia , 2009.

[16] R. Farivar, H. Kharbanda, S. Venkataraman, and R. H. Campbell, "An algorithm for fast edit distance computation on gpus," in Innovative Parallel Computing (InPar), 2012, IEEE, 2012, pp. 1–9.

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