Optimizing Needleman-Wunsch Algorithm through GPU Parallelization Techniques

Jane Huynh
ECE, University of Minnesota
huynh369@umn.edu

Kiana Vang
ECE, University of Minnesota
vang0951@umn.edu

Jeanine Thao
ECE, University of Minnesota
thao0515@umn.edu

Amir Jalili
ECE, University of Minnesota
jalil014@umn.edu

Farris Al-Humayani
ECE, University of Minnesota
alhum006@umn.edu

ABSTRACT

Global sequence alignment is a common technique in bioinformatics to determine an optimal alignment between two biological nucleotide sequences. The Needleman-Wunsch (NW) algorithm is a well known dynamic programming method for computing this optimal alignment. However, computing large input sequences is time-intensive for a CPU-bound implementation of NW. We have developed a GPU-based approach that shows significant speedup for input sequences of ~45,000 nucleotides, but with potential to align much larger sizes on a high end GPU. After testing with different sizes, our observations proved that our implementation can achieve a speedup up to ~48x for an input sequence of 37,000 with a block size of 32x32.

CCS Concepts

• Computing Methodologies Parallel → computing methodologies

• Theory of computation →Design and analysis of algorithms

Keywords

Needleman-Wunsch; Sequence alignment; Parallel Algorithm; DNA Sequence; Global Alignment; CUDA; GPU.

1. INTRODUCTION

There are many methods that compute biological nucleotide sequence alignment of two or more sequences; Global Sequence, Local Sequence, and Multiple Sequence alignments are the main three categories. Among these categories, there have been many computational algorithms developed for sequence alignment such as dynamic programming and heuristic programming. This paper focuses more on the dynamic programming approach to solve the problem of sequence alignment.

Although dynamic programming takes more execution time, it also provides a more accurate solution to the sequence alignment problem. One such example of a dynamic programming algorithm is the Needleman-Wunsch algorithm for global sequence alignment, which is the focus of this paper.

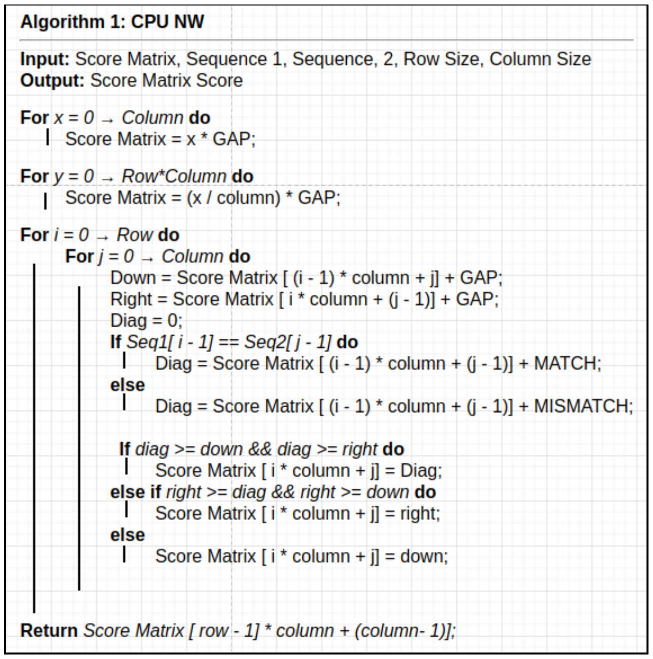
Saul B. Needlman and Christan D. Wunsch created the Needleman-Wunsch algorithm to optimally align two strings of nucleotides or proteins. It calculates a two-dimensional matrix to represent a score based on different alignment options, e.g. choosing to either insert a gap between compared nucleotides or taking its match/mismatch, to completely align 2 sequences. This comparison and score computation is typically done iteratively in row-wise fashion which is an O(n^2) computation and is a dynamic programming algorithm. This is not ideal because nucleotide sequence lengths can vary from hundreds to millions of nucleotides.

In this paper, we build on the basic NW algorithm from CPU algorithm to GPU algorithm. The GPU implementation of the NW algorithm is optimized through the use of different optimization methods such as shared memory and block-level kernel launch computation. Lastly, we present multiple approaches to speed up this computation, besides the methods previously mentioned, through use of parallelism to theoretically decrease computation runtime to O(n).

2. DESIGN OVERVIEW

2.1 CPU Implementation of NW

In this section, we further explore our CPU implementation of the NW to provide a better understanding of the NW algorithm. In this CPU version, we have a single function to compute the score matrices of the two input sequences. The following *Algorithm 1* provides our approach to the NW algorithm in the CPU implementation.

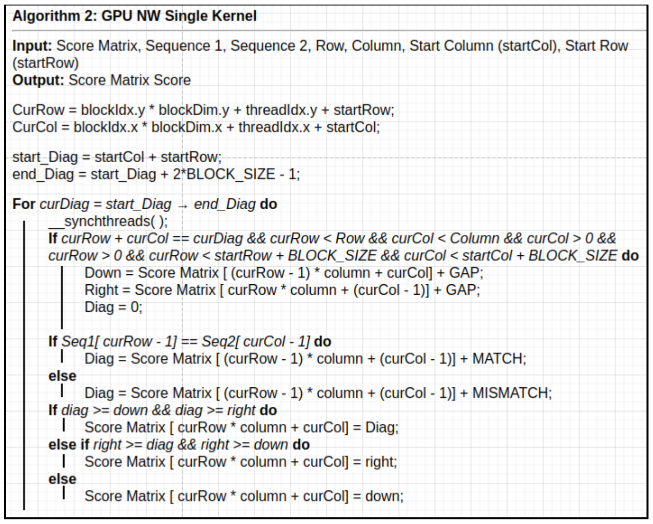


```
Algorithm 1: CPU NW

Input: Score Matrix, Sequence 1, Sequence, 2, Row Size, Column Size
Output: Score Matrix Score

For x = 0 → Column do
    Score Matrix = x * GAP;

For y = 0 → Row*Column do
    Score Matrix = (x / column) * GAP;

For i = 0 → Row do
    For j = 0 → Column do
        Down = Score Matrix [ (i - 1) * column + j] + GAP;
        Right = Score Matrix [ i * column + (j - 1)] + GAP;
        Diag = 0;
        If Seq1[ i - 1] == Seq2[ j - 1] do
            Diag = Score Matrix [ (i - 1) * column + (j - 1)] + MATCH;
        else
            Diag = Score Matrix [ (i - 1) * column + (j - 1)] + MISMATCH;

        If diag >= down && diag >= right do
            Score Matrix [ i * column + j] = Diag;
        else if right >= diag && right >= down do
            Score Matrix [ i * column + j] = right;
        else
            Score Matrix [ i * column + j] = down;

Return Score Matrix [ row - 1] * column + (column- 1)];
```

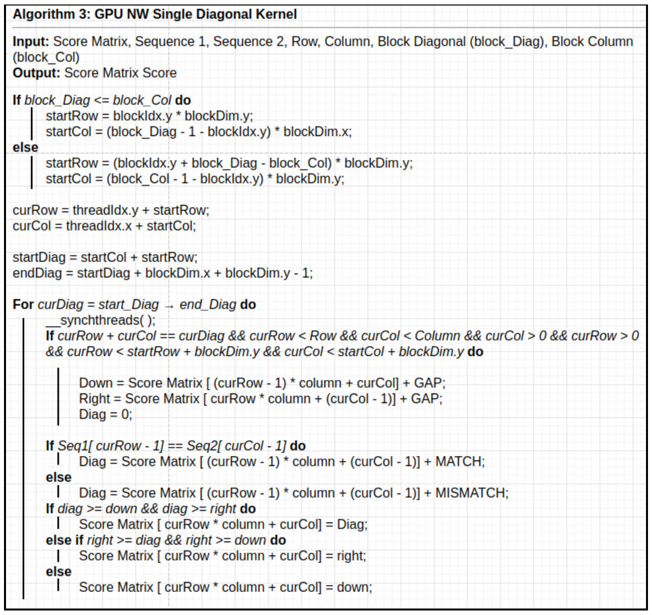
This algorithm computes sequentially and what we used to test our CPU computation. The time execution for the CPU is based on this algorithm using TIME_IT.

2.2 GPU Implementation of NW

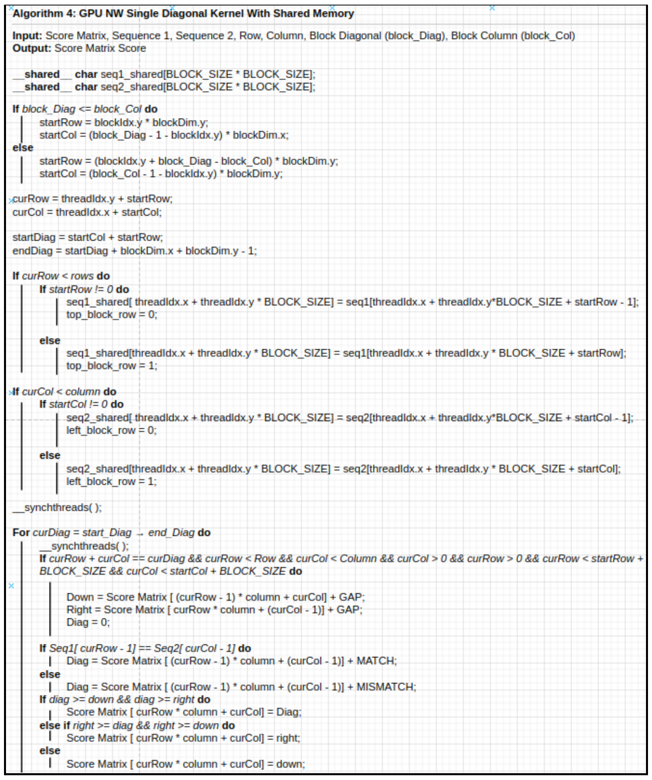
Our GPU algorithm of the NW algorithm builds on the basic NW algorithm to further improve performances through parallelism. In our algorithm, we firstly declared a block size of 32x32 using a maximum of 1024 threads. We also have different kernel functions to compute the NW algorithm including ones with shared memory optimization and diagonal computation for comparison. The following algorithms are the different GPU implementations that we tested including those with optimization.



```
Algorithm 2: GPU NW Single Kernel

Input: Score Matrix, Sequence 1, Sequence 2, Row, Column, Start Column (startCol), Start Row
(startRow)
Output: Score Matrix Score

CurRow = blockIdx.y * blockDim.y + threadIdx.y + startRow;
CurCol = blockIdx.x * blockDim.x + threadIdx.x + startCol;

start_Diag = startCol + startRow;
end_Diag = start_Diag + 2*BLOCK_SIZE - 1;

For curDiag = start_Diag → end_Diag do
    __synchthreads( );
    If curRow + curCol == curDiag && curRow < Row && curCol < Column && curCol > 0 &&
    curRow > 0 && curRow < startRow + BLOCK_SIZE && curCol < startCol + BLOCK_SIZE do
        Down = Score Matrix [ (curRow - 1) * column + curCol] + GAP;
        Right = Score Matrix [ curRow * column + (curCol - 1)] + GAP;
        Diag = 0;

    If Seq1[ curRow - 1] == Seq2[ curCol - 1] do
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MATCH;
    else
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MISMATCH;
    If diag >= down && diag >= right do
        Score Matrix [ curRow * column + curCol] = Diag;
    else if right >= diag && right >= down do
        Score Matrix [ curRow * column + curCol] = right;
    else
        Score Matrix [ curRow * column + curCol] = down;
```



```
Algorithm 3: GPU NW Single Diagonal Kernel

Input: Score Matrix, Sequence 1, Sequence 2, Row, Column, Block Diagonal (block_Diag), Block Column
(block_Col)
Output: Score Matrix Score

If block_Diag <= block_Col do
    startRow = blockIdx.y * blockDim.y;
    startCol = (block_Diag - 1 - blockIdx.y) * blockDim.x;
else
    startRow = (blockIdx.y + block_Diag - block_Col) * blockDim.y;
    startCol = (block_Col - 1 - blockIdx.y) * blockDim.y;

curRow = threadIdx.y + startRow;
curCol = threadIdx.x + startCol;

startDiag = startCol + startRow;
endDiag = startDiag + blockDim.x + blockDim.y - 1;

For curDiag = start_Diag → end_Diag do
    __synchthreads( );
    If curRow + curCol == curDiag && curRow < Row && curCol < Column && curCol > 0 && curRow > 0
    && curRow < startRow + blockDim.y && curCol < startCol + blockDim.y do

        Down = Score Matrix [ (curRow - 1) * column + curCol] + GAP;
        Right = Score Matrix [ curRow * column + (curCol - 1)] + GAP;
        Diag = 0;

    If Seq1[ curRow - 1] == Seq2[ curCol - 1] do
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MATCH;
    else
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MISMATCH;
    If diag >= down && diag >= right do
        Score Matrix [ curRow * column + curCol] = Diag;
    else if right >= diag && right >= down do
        Score Matrix [ curRow * column + curCol] = right;
    else
        Score Matrix [ curRow * column + curCol] = down;
```



```
Algorithm 4: GPU NW Single Diagonal Kernel With Shared Memory

Input: Score Matrix, Sequence 1, Sequence 2, Row, Column, Block Diagonal (block_Diag), Block Column (block_Col)
Output: Score Matrix Score

__shared__ char seq1_shared[BLOCK_SIZE * BLOCK_SIZE];
__shared__ char seq2_shared[BLOCK_SIZE * BLOCK_SIZE];

If block_Diag <= block_Col do
    startRow = blockIdx.y * blockDim.y;
    startCol = (block_Diag - 1 - blockIdx.y) * blockDim.x;
else
    startRow = (blockIdx.y + block_Diag - block_Col) * blockDim.y;
    startCol = (block_Col - 1 - blockIdx.y) * blockDim.y;

curRow = threadIdx.y + startRow;
curCol = threadIdx.x + startCol;

startDiag = startCol + startRow;
endDiag = startDiag + blockDim.x + blockDim.y - 1;

If curRow < rows do
    If startRow != 0 do
        seq1_shared[ threadIdx.x + threadIdx.y * BLOCK_SIZE] = seq1[threadIdx.x + threadIdx.y*BLOCK_SIZE + startRow - 1];
        top_block_row = 0;

    else
        seq1_shared[threadIdx.x + threadIdx.y * BLOCK_SIZE] = seq1[threadIdx.x + threadIdx.y * BLOCK_SIZE + startRow];
        top_block_row = 1;

If curCol < column do
    If startCol != 0 do
        seq2_shared[ threadIdx.x + threadIdx.y * BLOCK_SIZE] = seq2[threadIdx.x + threadIdx.y*BLOCK_SIZE + startCol - 1];
        left_block_row = 0;

    else
        seq2_shared[threadIdx.x + threadIdx.y * BLOCK_SIZE] = seq2[threadIdx.x + threadIdx.y * BLOCK_SIZE + startCol];
        left_block_row = 1;

__synchthreads( );

For curDiag = start_Diag → end_Diag do
    __synchthreads( );
    If curRow + curCol == curDiag && curRow < Row && curCol < Column && curCol > 0 && curRow > 0 && curRow < startRow +
    BLOCK_SIZE && curCol < startCol + BLOCK_SIZE do

        Down = Score Matrix [ (curRow - 1) * column + curCol] + GAP;
        Right = Score Matrix [ curRow * column + (curCol - 1)] + GAP;
        Diag = 0;

    If Seq1[ curRow - 1] == Seq2[ curCol - 1] do
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MATCH;
    else
        Diag = Score Matrix [ (curRow - 1) * column + (curCol - 1)] + MISMATCH;
    If diag >= down && diag >= right do
        Score Matrix [ curRow * column + curCol] = Diag;
    else if right >= diag && right >= down do
        Score Matrix [ curRow * column + curCol] = right;
    else
        Score Matrix [ curRow * column + curCol] = down;
```

2.3 IMPLEMENTATION

We uncovered various techniques to achieve performance speedup while keeping the NW algorithm and GPU limitations in mind. In this section, we discuss the robustness of our implementation methods and reasoning in detail.

3.1 Data Dependencies

A major bottleneck of this algorithm is that each score is dependent on the score of the element that is above it, to the left of it, and to its top left diagonal. The score matrix's final result essentially records the best path for choosing an alignment by choosing the best score calculated from moving down, right, or along the diagonal from (0,0) to the bottom right corner of the matrix. Moving right or down introduces a gap to the corresponding sequence, while moving along the diagonal takes the nucleotides to the corresponding sequences as a match/mismatch. In order to maximize parallelism, our implementation focuses on sequential calculation of scores along diagonals through multiple kernel approaches.

3.2 Base Case (Row 0 and Col 0)

To compute the first row and column cells of the score matrix, a 0 is assigned to the first entry in the matrix and then the entire row and column is filled in based on the incremental addition of gap penalties. These cells in the table do not require the data dependencies mentioned above and can be maximally parallelized. To do this task a single kernel is launched with enough threads to map each working thread to a cell and compute its value. This kernel call is consistent across all our implementations and is used as the first step of the algorithm.

3.3 Dynamic Parallelism Multi-Kernel

The core of this approach was to sequentially calculate the diagonal elements of the score matrix and switch between different kernels as the parallelism increases and decreases. The approach switches between the warp level, block level, and multi-block level kernel implementations based on the GPU specifications of the GTX 1080. The warp level is performed for the first and last 32 diagonals since there are 32 threads per warp. The block level is performed for the remaining first 1024 and last 1024 diagonals. The multi-block approach is used for the remaining diagonals.

This theoretically would be computed in O(n) time. Each diagonal is computed sequentially with all elements in the diagonal calculated in parallel. Hence, the number of iterations for this is the number of diagonals in the score matrix, i.e. rows + cols - 1. This is clearly O(n).

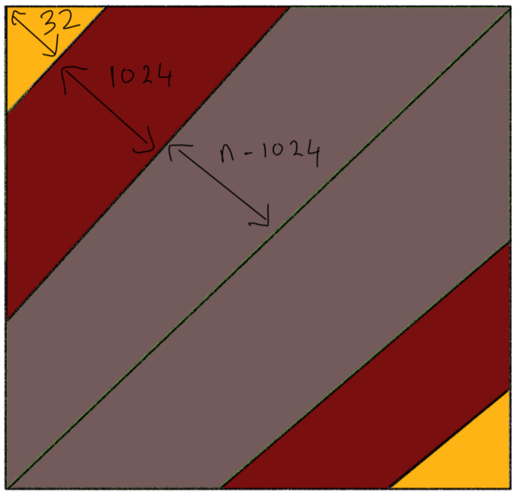


Figure 1. Dynamic Parallelism Multi Kernel

3.3.1 Warp Level

This kernel is launched with a single block of 32 threads and is used when the number of cells to be computed at each diagonal of the score matrix is minimal. Specifically, when the number of elements in the diagonal is less than 32, the yellow regions on figure 1. The kernel iterates through each diagonal computing the cells one at a time. Since all the threads in a single warp execute a single instruction at a time, there is no need to use barrier synchronization at the end of each iteration.

3.3.2 Block Level

This kernel is launched with a single block of 32x32 threads and is used when the number of cells to be computed at each diagonal is less than 1024, the maroon regions on figure 1. The kernel iterates through each diagonal computing the cells one at a time. Barrier synchronization is needed at the end of each iteration to ensure that all threads are done with the computations before moving to the next diagonal.

3.3.3 Multi-Block Level

This kernel is launched when the amount of parallelism to compute each diagonal grows higher than 1024, the brown regions on figure 1 . At this point, since a single block cannot compute all cells of a given diagonal at once, multiple 32x32 blocks are launched mapping each working thread to compute a single cell in the diagonal.

3.4 Diagonal Block Kernel

The core of this approach is to break down the score matrix into diagonal blocks and to sequentially calculate the diagonal blocks of the grid. Each block themselves will calculate all of the elements corresponding to that block in the score matrix. Each diagonal of blocks is launched sequentially through multiple kernel launches.

This theoretically would be computed in O(n) time. Each block in its corresponding diagonal would compute in parallel with blockDimx.x + blockDimx.y - 1 iterations. The number of block diagonals in the grid that are computed sequentially are ⌈ rows / blockDimx.y ⌉ + ⌈columns / blockDimx.x⌉ - 1. Hence, the total number of iterations to be completed would be (blockDimx.x + blockDimx.y - 1) * (⌈ rows / blockDimx.y ⌉ + ⌈columns / blockDimx.x⌉ - 1). Assuming a square score matrix that is n x n with blocks of size 32 x 32, this simplifies to 63 * (⌈ n / 32 ⌉ + ⌈n / 32⌉ - 1) ≅ 4n which is clearly O(n).

3.4.1 Single Block

On the block level, each thread maps to one element of the score matrix. The diagonals of the block would then be looped through starting from the top left and the elements corresponding to that diagonal would be calculated. A syncthreads() would have to be performed between each iteration to verify that necessary data dependencies are calculated. As shown in figure 2, each blue arrow starting from the top left corner would be computed sequentially, with all elements along the diagonal computed in parallel. In this portion of the kernel, only block level synchronization is required.

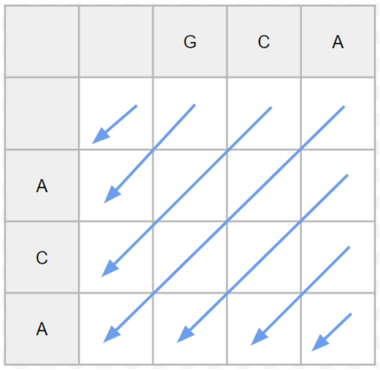


Figure 2. Diagram of Elements Computed in a Matrix

3.4.2 Grid of Diagonal Blocks

In order to get past synchronization limits of the block, the score matrix is broken down into blocks. Then blocks along diagonal blocks of the grid are sequentially calculated. We loop through the grid's diagonals and launch one kernel per diagonal of blocks. The kernel is launched with 1 x *y* x 1 grid dimension with y being the number of blocks in the current grid diagonal. As shown in figure 3 with blocks of size 4x4, there are 3 block diagonals. Looping from 1 to 3: on the first iteration the kernel is launched with 1 block to calculate the orange section, the second iteration is launched with 2 blocks to calculate the blue section, and then the third iteration is launched with 1 block to calculate the green section. This works because if all the elements of block 1 are done, then all the data dependencies required for the second diagonal of blocks are also done, etc.

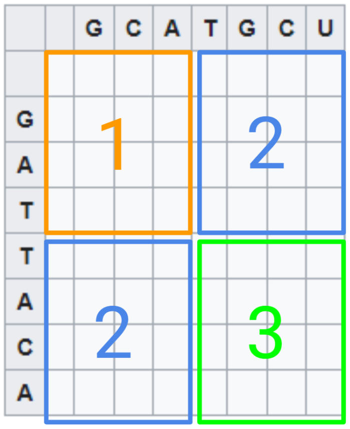


Figure 3. Block Breakdown of Score Matrix

This requires calculating the number of blocks in each diagonal, as well as calculating the x and y offset for the top left element of each block. Because we're launching the kernel multiple times, the blockidx.x and blockidx.y do not directly correspond to the block on the score matrix. For example when calculating block 1, no further offset is necessary. When calculating the second diagonal of blocks we have to offset the top block to the right. When calculating block 3, we have to offset it down and to the right

3.4.3 Shared Memory

This approach is further optimized by utilizing shared memory. The relevant part of the sequences being aligned for a block are stored in shared memory. Each character is accessed once per each element in the row or column that it's aligned to along the score matrix. The way that these characters are accessed are also coalesced. For example when calculating the diagonal elements marked by X on figure 4, the threads will access all the values marked by the blue arrows at the same time and all the values marked by the red arrows at the same time when determining if the score from going diagonal is a match or a mismatch.

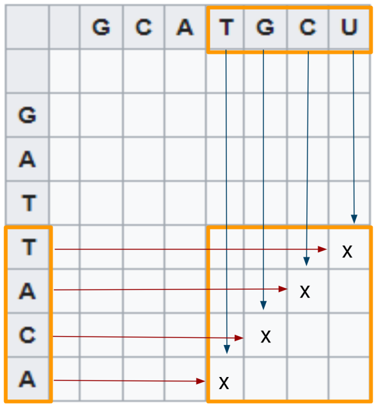


Figure 4. Memory Coalescing Access Pattern Within Block

3.4.4 Block Sizing

This approach examines various block sizes to determine the most optimal block size dimensions for our GPU algorithm. Because each block is limited to 1024 threads, we tested the following block sizes: 8x128, 16x64, and 32x32.

After testing, all the three block sizes produced very similar speedup results. For simplicity of our algorithm, we decided to proceed with the block size of 32x32 when performing further investigation or analysis.

We observed up to ~78x speedup compared to the CPU when using a block size of 16x16 and 8x8, faster than 32x32, but did not fully explore this.

Table 1. Timing of Various Square Block Sizes

Input	Block Size	GPU Time	CPU Time
10000	1	0.2210732	3.72
10000	8	0.0580139	3.72
10000	16	0.0581879	3.72
10000	32	0.0887748	3.72
20000	8	0.1960025	14.875
20000	16	0.1915364	14.875

20000	32	0.3177149	14.875

3.5 Timing Comparison of Kernels

When comparing the kernels mentioned above, i.e. dynamic parallelism multi-kernel and diagonal block kernel with and without shared memory with block size 32 x 32, the timing results are shown below in figure 5 for various input lengths. Note for simplicity, sequences of the same length were aligned.

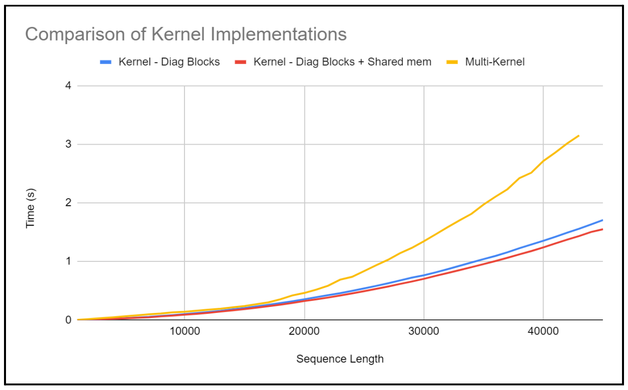


Figure 5. Kernel Implementation Comparison

CPU timing is not pictured here as the timing was much longer than the kernels displayed. Surprisingly, we found the multi-kernel approach to be ~2x longer than the diagonal block approach. Use of shared memory also had about a 10% speedup than without shared memory for the diagonal block approach.

3.5 Input Options

Our implementation also has different input sequence options to make it well-rounded for real life applications. In our program, we have the option of either passing in two files of different combinations of ACTG sequences for alignment that is pre-defined. Another option is to choose a random sequence generated by a random sequence function provided in our code. This random sequence function generates both the ACTG sequence in random as well as the size of the sequence.

3. ANALYSIS

In this section, we present our observations and results including in-depth discussion of what our results entail. We discuss the different methods of verifications that we included to test the robustness and accuracy of our program compared to other research papers that used different GPU implementation methods. In addition, the performance result of our version of GPU implementation is also examined more closely and discussed below in the subsections.

3.1 Verification

To make sure that our CPU and GPU algorithms are correct, we compared every element of the CPU and GPU score matrix for verification. If they match, then our shell will print out a “Test Passed” statement.

We verified our timing calculation of the CPU and GPU using two methods. We tested our CPU algorithm using a function called TIME_IT; it utilizes the built-in function “gettimeofday” that is available in timer.h and sys/time.h libraries. For our GPU, we timed our algorithm with CUDA Event functions. Both the CPU and GPU were timed over ten iterations. The averages for both were used in our final results.

Furthermore, we compared our results to prior research and found that our implementation produced similar speedup results when using similar input sequence sizes even though our approaches are different. Chaudhary et al. yield a speedup of 75.2x using 1D array for the representation methods instead of 2D array.[1]

4.2 Performance

From our observations, the performance of our first implementation of the block-level computation performed as expected from our initial hypothesis. We achieved a speedup of ~48x for an input sequence length of ~37k by implementing the following optimization approach: 32x32 block size, shared memory, and single kernel per diagonal block. The results are shown in figure 6 and 7. It is observed that the implementations between CPU and GPU greatly differ even though the execution time for both appear to grow exponentially with input size. This difference is expected due to parallelism implementation done on the GPU versus sequential implementation done in the CPU. Our GPU implementation also achieved a greater speedup performance compared to Chaudhary et al. with a speedup of 6x.[1] Jararweh et al. and Fakirah et al. showed a speedup of 75.2x and 99x respectively.[2,3] This confirms our speedup of 48x to be within reasonable range.

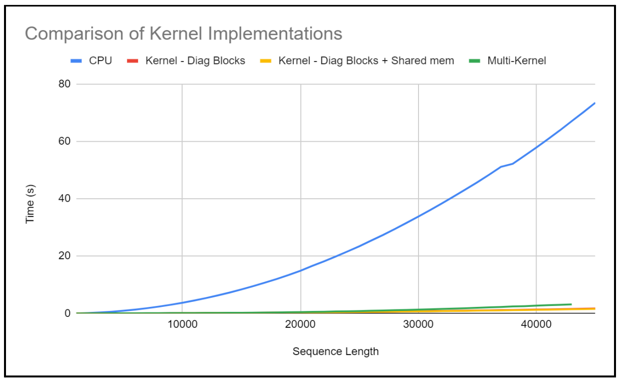


Figure 6. Comparison of Kernel Implementations

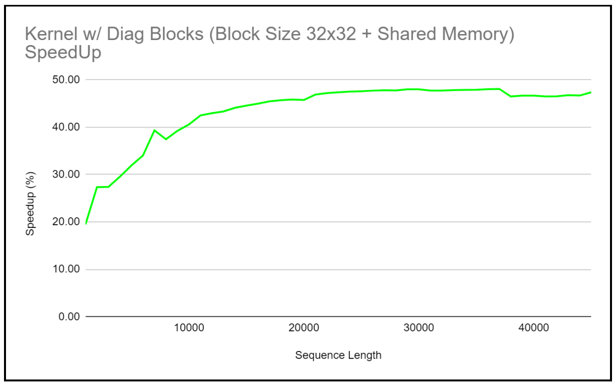


Figure 7. Diagonal Kernel Execution with Shared Memory Optimization Speedup Compared to CPU

In addition to our single kernel implementation (using diagonal blocks with shared memory), we also implemented a multi-kernel approach. After testing, our multi-kernel function yielded a longer execution time compared to our single kernel for the same input sequence sizes. For an input sequence size of 37,000, our single kernel's time execution is ~ 1.1s whereas the multi-kernel implementation yields ~2.2s. Analyzing these approaches further, our multi-kernel implementation ran out of memory for input sequence sizes greater than 43K whereas for the single diagonal with shared memory kernel implementation ran out of memory for input sizes greater than 45K. This concludes that the single diagonal with shared memory kernel implementation results in better performance compared to the multi-kernel implementation.

4.2.1 Limitations of Performance

Despite attaining a speedup of ~48x, there are still some limitations that hindered our implementation from achieving higher speedup. These limitations consist of both hardware and software. In this section, we will discuss the limitations that impacted our performance.

In terms of hardware, we are limited by the GPU specifications of the GTX 1080. We are limited by the number of threads that we could utilize per warp and the number of threads per block, which are 32 and 1024 respectively. This limited the maximum number of threads that can be calculated in parallel in the diagonal block implementation for every block, requiring greater subdivision of the score matrix. This also limited the number of threads that can be scheduled to do the execution. As shown in figure 7, the speedup eventually flattened out when execution has reached the maximum number of parallel threads available in the GPU. Furthermore, we maxed out all available global memory (~45k) resources due to direct calculation and transfer of score matrix and input sequences to the GPU.

In regards to software, certain aspects of our kernel implementation hindered the possibility of achieving the best performance possible. Due to the nature of the NW algorithm, our implementation's performance is limited by data dependencies, i.e. each element depends on the calculated value in the above, left and diagonal adjacent cells. In the diagonal block kernel, all of the previous diagonal blocks are required to finish first before calculating the current block. In the current block, an element also has to wait for its dependencies to be computed. In the multi-kernel approach, each element must wait for all elements in previous diagonals to be computed. Our implementation of diagonal blocks kernel and dynamic parallelism kernel maximized parallelism to solve our data dependencies issue, but this resulted in large overhead due to multiple kernel launches. This is necessary since the order of block execution is unknown. One kernel launch is performed per diagonal of blocks or elements, respectively.

4.2.2 Future Work

There are numerous ways and room for future improvement. To improve the performance result of our program, we suggest implementing our code on a different GPU such as the GTX1080 Ti that has more memory (28 SMs and 11k MT/s). This would enable us to utilize more memory for larger input sequences and score matrices to analyze larger models of nucleotide sequences such as genomes, which can be ~1M in length. This solution will also allow a higher maximum number of threads that can work in parallel and have faster clock speed.

We can further modify our code to include CUDA streams. This approach will divide the block-level computation to execute in parallel in multiple streams, which will potentially achieve a better time execution and speedup. Another possible improvement is mapping each thread to a row such that each thread has an offset to the start columns and calculates the rest of the rows. This can maximize diagonal computation in parallel and reduce overhead from multiple kernel launches for those rows. Another improvement is to combine the use of the kernel computing 1 thread per row up to diagonal x, where x is the maximum number of threads in a block and the diagonal block computation kernel. This improvement could reduce the total number of kernel launches as well as maximize parallelism during the execution of the small diagonals. Lastly, data marshalling can be implemented for larger input sequences by calculating smaller chunks of the score matrix at a time. This will reduce and prevent overflow of global memory usage.

4. CONCLUSION

After robust testing and verification of our implementation methods, we conclude that we achieved our expected speedup performance. From our analysis, the highest speedup generated from our implementation method is ~48x with an input sequence of size 37k. With the limitations presented, it prevented greater performance and speedup from being achieved. The major limitation is data dependencies within the NW algorithm. This issue is solved using block-level computation. Hardware limitations can be solved by using a different GPU such as the GTX 1080 Ti.

From observation, we tested different block sizes to compare the performance differences between non-square and square sizes. We tested 8x128, 16x64, and 32x32 block sizes as these are the possible combinations that equate to maximum 1024 threads per block. It is observed that non-square block sizes (8x128 and 16x64) appeared to produce similar execution time compared to the size of 32x32. Hence, we chose to use 32x32 as our block size for simplicity.

There is room for improvements and future work that will result in faster speedup and parallelism. We suggest mapping of threads to rows of the score matrix to reduce overhead and experimenting with CUDA streams and data marshalling.

5. ACKNOWLEDGEMENTS

We would like to thank ACM SIGCHI for allowing us to modify templates they developed.

6. REFERENCES

[1] A. Chaudhary, D. Kagathara and V. Patel, "A GPU based implementation of Needleman-Wunsch algorithm using skewing transformation," 2015 Eighth International Conference on Contemporary Computing (IC3), Noida, 2015, pp. 498-502, doi: 10.1109/IC3.2015.7346733.

[2] M. Fakirah, M. A. Shehab, Y. Jararweh, and M. Al-Ayyoub, "Accelerating Needleman-Wunsch global alignment algorithm with GPUs," 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), 2015.

[3] 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," Multimedia Tools and Applications, vol. 78, no. 4, pp. 3961–3977, 2017.