COMP 429
PARALLEL PROGRAMMING
TERM PROJECT REPORT

Mustafa Mert Ögetürk

Mustafa Acıkaraoğlu

Mustafa Saraç

8 Jan 2017

KOÇ
UNIVERSITY

Introduction

The very first thing we started to do was to search the Needleman-Wunsch algorithm. The Needleman–Wunsch algorithm is an algorithm used in bioinformatics to align protein or nucleotide sequences. Later, we investigated the IBM codes for building the serial implementation. After finding a serial code on the web, we organized the code as desired. We added the time getting methods, and regulated the traversing directions in the matrix. After that, we set the makefile so that it will work smoothly. Needleman-Wunsch algorithm is executed fast enough. The problem was that, if it works with huge nucleotide sequences, it becomes very time consuming. We aimed to decrease the execution time for very big DNA stripes by setting the traverse operations as parallel.

Implementation of Needleman–Wunsch algorithm

Second step was understanding the concepts of Needleman-Wunsch Algorithm because parallel implementation would be limited by the capability of CUDA Libraries. Needleman-Wunsch Algorithm works on 4 main step;

- Determine the scoring system:

In this step, we determined 3 different penalty scores, as 0 penalty score for matching letters, 1 penalty score for mismatching letters and 2 penalty score for INDEL(Insertion and Deletion). For INDEL case one letter aligns to a gap in the other string.

- Fill a 2D array for the use of penalties:

In this step, we created a two dimensional array which is composed with ones and zeros. Diagonal line is full of zeros and other elements are 1 in this array. It will be used in the aligning algorithm to calculate the penalties later on.

- Fill the comparison matrix with initial penalty scores:

Align matrix is represented as two dimensional array in the project. This step aims to build the initial position of the align matrix. For initializing, First row and first column of the matrix are filled with increasing penalty score numbers. Scores are sequentially increasing by 2 penalty score because diagonal of the matrix is for the best fit comparisons. Every step overflowing from the diagonal decreases the similarity.

- Obtaining the minimum score by comparing the stripes:

In this step, DNA chains are placed in first row and first column in order. After that, aligning matrix is filled with penalty scores by comparing all letters in the chain. To do that, the algorithm calculates the similarity score by calculating the values of indexes in order, but this order continues on a diagonal line for every values. Because each matrix cell checks its top cell, left sell and top-left cell to minimize the score in the new calculated cell. Eventually, In the last index of the two dimensional array, similarity score is occurred. (figure 1)

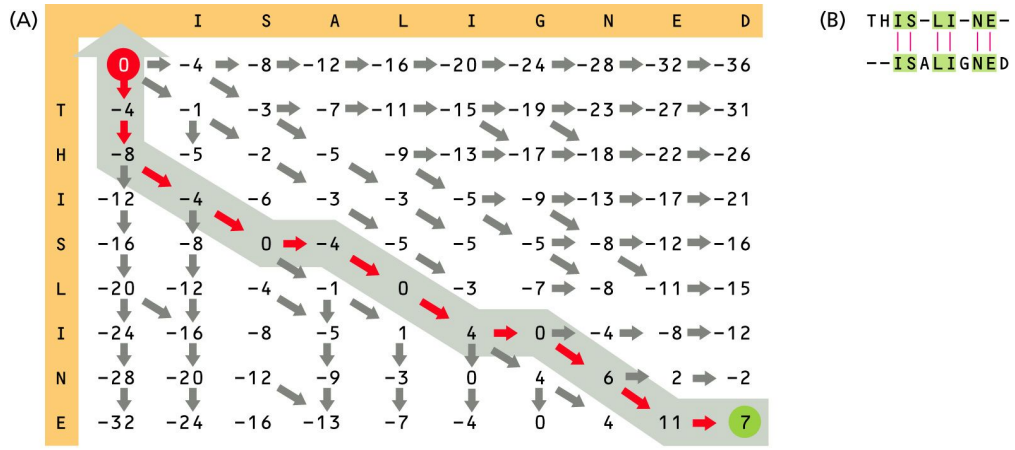


Figure 1 (Needleman-Wunsch)

CUDA Implementation

In this part, we focused on the two array filling and the aligning methods which are the most expensive parts of our project in terms of performance. Firstly, we converted our two dimensional arrays to one dimensional arrays by row major indexing because cudaMalloc operation only allows one dimensional memory allocation.

Kernel 1

The very first kernel initializes the alphabet array in order for the algorithm to figure out whether the given letters match or not. The alphabet array is a 26x26 matrix filled with 1's and 0's. The matching letters (the equal indexes eg. alphabet[0][0] alphabet[1][1]) are initialized as 0 and the non matching ones are initialized as 1. The CUDA implementation simply copies the host array to the device and initializes the array in the fast NVIDIA GPU.

Kernel 2

The second kernel initializes the main calculation matrix by initializing the first row and the first column according to the gap penalty. This kernel simply multiplies each index with the gap penalty and writes its new value, in order for the sequences to be recognized in the matrix.

Kernel 3

The third kernel does the expensive task in parallel, by reading the previous data (A[i-1][j], A[i][j-1], A[i-1][j-1]), calculating the penalty and writing the minimum penalty to its next index (A[i][j]). This part proved to be difficult, because CUDA will automatically divide the big matrix to the small blocks and running them at the same time. This would cause the program to have a race condition. This is why we have developed several solutions to this issue.

Challenges

In Kernel 3, there is a data dependency so that it pushes the limits of CUDA. Each cell is calculated according as its top, left and top-left cell. This dependency indicates a triangular relation between the matrix cells.(Figure 2) Therefore it is impossible to do these calculations in

one kernel and on a global indexing mapped matrix. This became our main challenge in this project.

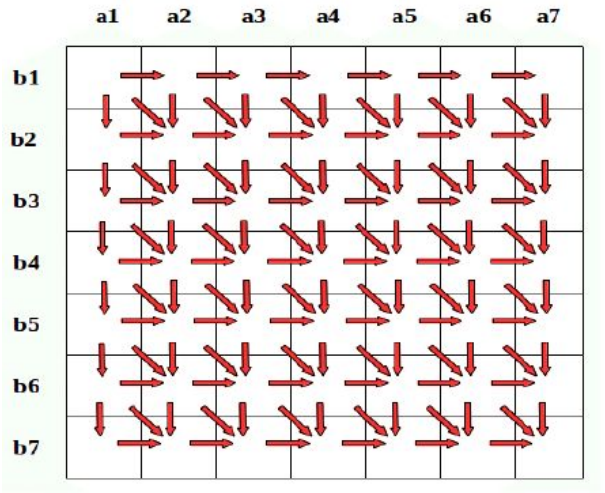
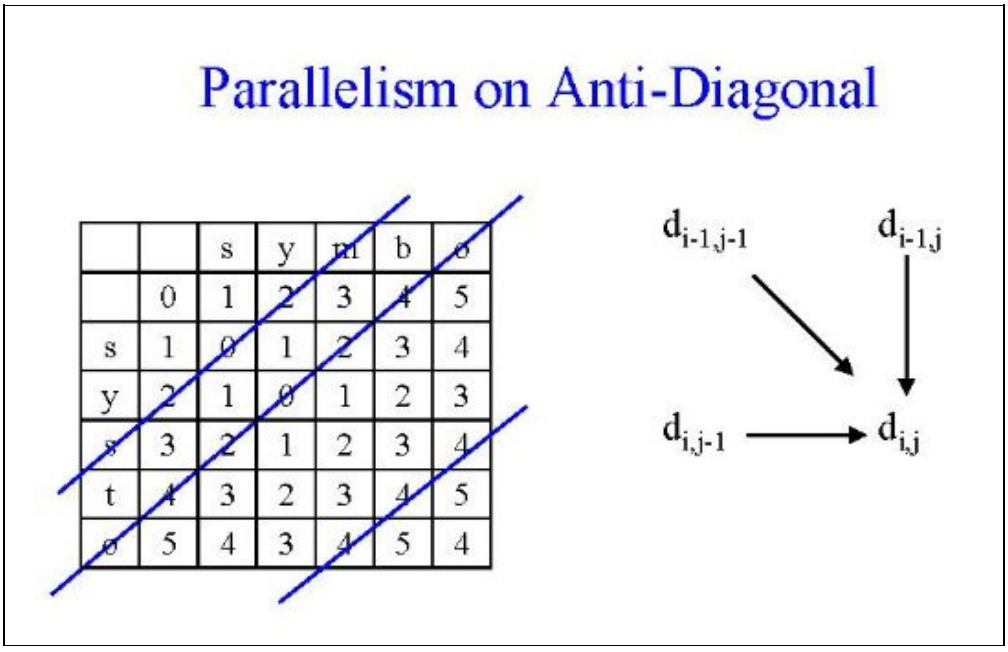


Figure 2

Solutions

To handle the top-left triangular dependency for each cell in Kernel 3, was the most crucial part of our project because the costly calculations are done in this aligning part. When we think about parallelization of this part, only option was running all the anti-diagonals of the matrix serially, but running each anti-diagonal as parallel in itself. To implement this, we apply 3 different ideas.



Idea-1

This idea uses 2D grid of 1D blocks CUDA indexing to obtain the global index of the threads. It basically, runs the threads for each anti-diagonal sequentially and makes the other anti-diagonals busywait. To do that, we calculated the index number of the anti-diagonal sets. When we consider the sum of the coordinates of threads, it returns to all the threads in the same diagonal for every different sum value.For example, [(0,4)(1,3)(2,2)(3,1)(4,0)] coordinates represent the fourth anti-diagonal in the matrix. We were able to traverse the matrix in all blocks but blocks do not wait each other in cuda.(Figure 4) So, this idea did not work with global indexing.

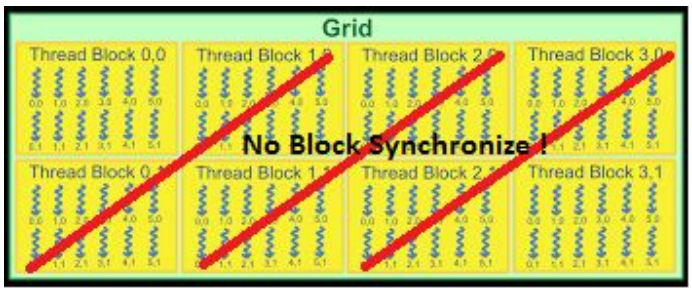


Figure 4

Idea-2

This idea aims to solve the block synchronization problem in the previous trial. Here, we decided to control the anti-diagonals by using the kernel function calls. In this method, aligning kernel iterates itself for each anti-diagonal line in the matrix. So, we can prevent the working of the other threads at the same time in this way. You can find a pseudo-code for iterating the kernel function below;

```
// Invoke kernel.
  for (int d = 0; d < size_DNA1 + size_DNA2; d++) {
   alignment_kernel<<< gridDim, blockDim >>>(d_T, char_arr1,char_arr2, size_DNA1, size_DNA2, d);
  //CHECK_FOR_CUDA_ERROR();
  }
```

Eventually, this means approximately 40.000 kernel calls in the device. Every time, it creates the grid of data and maps the thread coordinates. Therefore, the computation becomes bigger at each kernel call so this affects the performance in a very bad way.

Idea-3

We realized that calling the kernel function over and over, causes a very costly execution. So, need to control the anti-diagonal sequence in the kernel inside. To do that, we decided to create a one dimensional CUDA thread block and work with it, because CUDA does not allow the block synchronization as mentioned in the previous solutions. Then, we tried to map the sequential threads in the block to the anti-diagonal indexes in the matrix. This method could run with no problem until the method comes to the anti-diagonal which has the size of the one dimensional CUDA block. Therefore, we had to shift the function on bigger anti-diagonals. But for every diagonal so for every case in the method requires different boundaries, different indexing and different way of function shifting on the diagonal lines.(Figure 5) So, we could not manage these structure by keeping all the conditions.

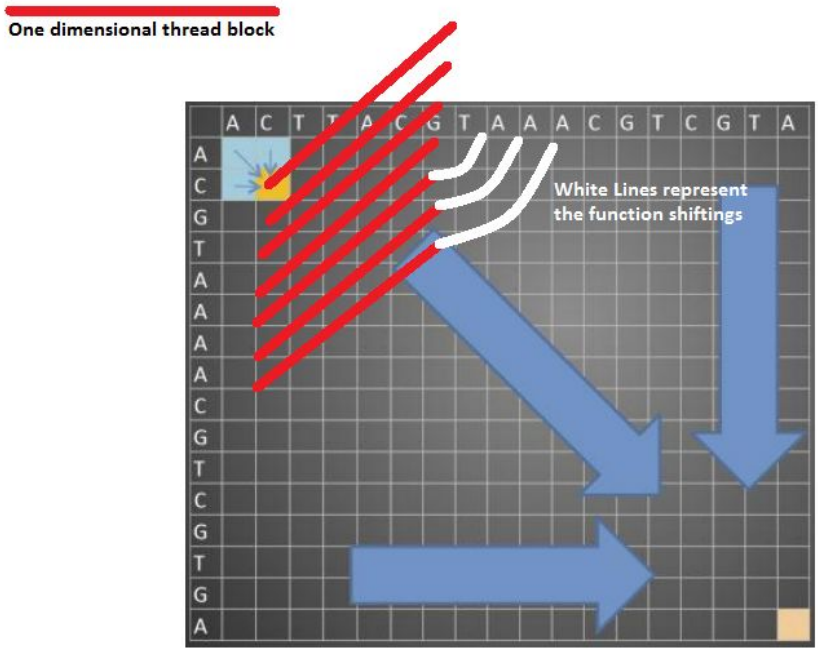
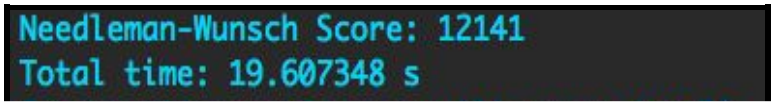


Figure 5

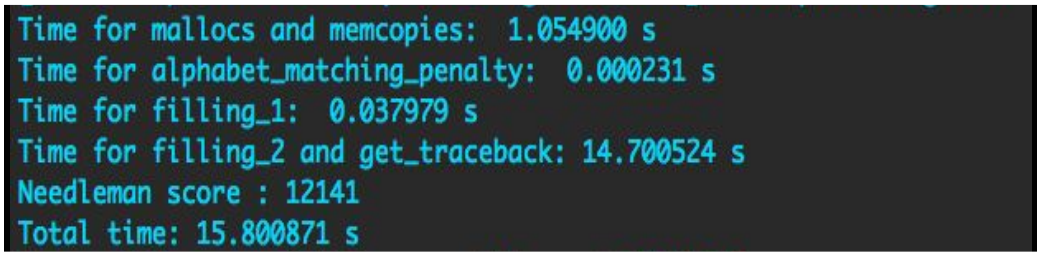
Results

As we mentioned before, because of the reason that our Kernel 3 has a top-left triangular dependency for each cell, we could not get the result of this kernel in a successful way. Then, we reimplemented the source code by changing the Kernel 3 to the serial implementation and leaving the first two kernels same. As a reminder, the source code that we uploaded, also contains the failure implementation of the Kernel 3 but we commented out it to get a successful result. Therefore, the only results that we got from the cluster are for the serial implementation and the parallelized version, which contains the Kernel 1 and Kernel 2.

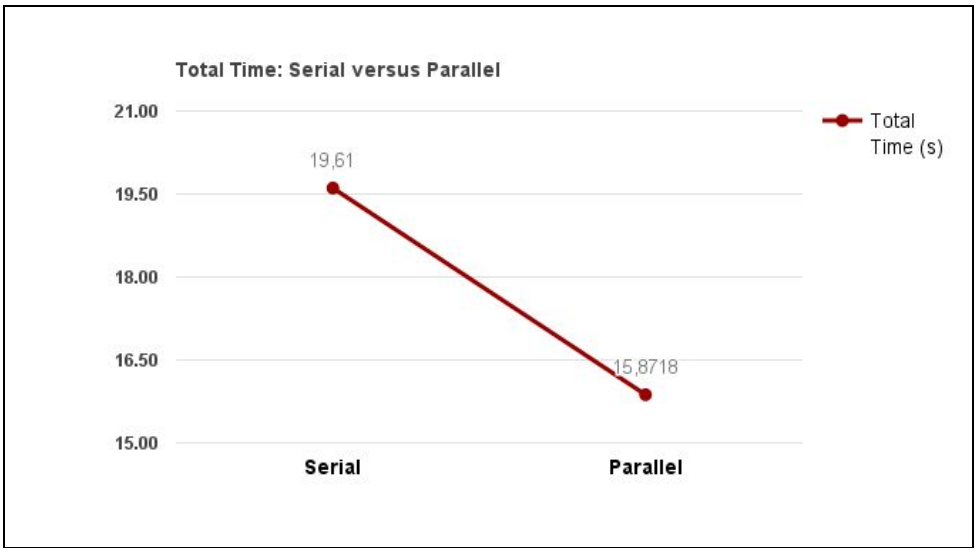
Here is the result of the serial implementation of Needleman-Wunsch algorithm. We executed the code with two DNA sequences which are separately consisted of 20000 nucleotides. The Needleman-Wunsch score, which represents the similarity between the DNA sequences, is 12141 out of 20000 nucleotides. Therefore, the similarity percentage is approximately %60. In addition to this, the total execution time of this implementation is 19.607348 seconds.



Here is the result of the parallel implementation of Needleman-Wunsch algorithm. We again executed the code with two DNA sequences which are separately consisted of 20000 nucleotides. The Needleman-Wunsch score is again 12141 out of 20000 nucleotides. Therefore, the similarity percentage is approximately %60. This shows that the parallel code executed properly. The total execution time of this implementation is 15.800871 seconds.



When we consider the the most meaningful results that we achieve successfully, we got approximately 1.24x speed-up. It was lower than our expectations before starting the project. It can be seen in the table below.



Conclusion

The CUDA approach to this problem was problematic, because CUDA cannot handle these kinds of data dependencies better than the other parallel libraries. Since the problem consists of reading the previous data and writing the next value according to them, CUDA did not fully synergize with this operation. Other libraries such as OpenMP or MPI would prove to be much more useful when it comes to parallelizing this problem, because we could better talk between threads and figure out whether the previous thread has finished processing its own data. CUDA lacks the capability of letting other thread blocks to talk with each other, that is why there was a problem reading the complete array into small blocks and processing their data. If the blocks could synchronise with each other, then we could have easily resolved the issue of data dependency. We would have simply iterated over the all blocks respectively.