International Journal of Research in Engineering, Science and Management
Volume-1, Issue-12, December-2018
www.ijresm.com | ISSN (Online): 2581-5792

Speed-up of Sequence Alignment Algorithms on CUDA Compatible GPUs

Pradyot Patil[1], Prasad Pattiwar[2], Sahil Khan[3], Varad Panch[4]
[1,2,3,4]*Student, Department of Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, India*

***Abstract*: GPU has evolved through the years to have applications in new domains such as Bioinformatics. The computational power required by these domains often exceeds that available on traditional CPUs. An emerging alternative is represented by General Purpose scientific computing on Graphics Processing Unit (GPGPU). The aim of this paper is to provide a comprehensive comparison of serial and parallel implementations of well-known Longest Common Subsequence (LCS) algorithms and study their behaviour on different GPU architectures. A major computational approach for solving the LCS problem is dynamic programming. Many dynamic programming methods have been proposed to have reduced time and space complexity. The succeeding topics explores the proposed dynamic programming solutions to Sequence Alignment problem and our approach for parallelization of the discussed solutions.**

***Keywords*: Dynamic Programming, GPU, LCS, Sequence Alignment.**

1. Background

Sequence Alignment [1] is a fundamental technique for biologists to discover the similarity of various species. For computational purposes biological sequences are represented as strings. For instance, DNA sequences (genes) can be represented as sequences of four letters A, T, G and C corresponding to four sub-molecules which collectively form a DNA. When a new DNA sequence is found, biologists are inquisitive to know what other sequences it is most familiar to. One way of detecting similarity between two genes is to find their LCS which is a dynamic programming method.

The LCS [2] problem is to find a substring that is common to two or more input strings and is longest one of such strings. LCS is a special case of global sequence alignment. The dynamic programming [4] approach to solve LCS problem includes filling of scoring matrix through a predefined scoring mechanism. The best core is the length of the LCS and the subsequence can be found by tracing back the table. Consider the length of two input DNA sequence that are to be compared as m and n. The time and space required to find the LCS of input strings is O (m*n). The scoring mechanism involved in finding the LCS s given below.

$$F[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{F[i, j-1], F[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \mathrel{!}= y_j \end{cases}$$

Where F is two-dimensional scoring matrix of size m+1 * n+1 and is filled according to the scoring mechanism given above. The common sequence is found by backtracking this F matrix.

Sequence alignment is of two types: Global Sequence Alignment (GSA) and Local Sequence Alignment (LSA). GSA attempt to align the entire sequence i.e. end to end alignment. If two sequences have approximately the same length and are quite similar, they are suitable for GSA. GSA is usually done for comparing homologous genes like comparing two genes with same function (in human vs. Mouse) or comparing two proteins with similar function. Needleman-Wunsch is a common GSA algorithm. LSA finds local regions with the highest level of similarity between the two sequences. It aligns a substring of the query sequence to a substring of the target sequence. Any two sequences can be locally aligned as local alignment finds stretches of sequences with high level of matches without considering the alignment of rest of the sequence regions. It is used for finding out conserved patterns in DNA sequences for conserved domains or motifs in two proteins. Smith-Waterman algorithm is a common LSA algorithm. These algorithms are implemented on traditional CPUs. But their implementation on GPGPUs has been limited. This is because these problems fall under dynamic programming. In dynamic programming, there are data dependencies i.e. unless previous data value has not been calculated the algorithm cannot proceed to calculate further values. This limits parallelisation. However, attempts have been made to parallelize despite these dependencies by using wave fronts. Consider LCS for example in which unless the cell above, left and diagonally above left has not been calculated the current cell value cannot be found. To break this dependency wave front of anti-diagonals are calculated in parallel. There are still dependencies between wave fronts however each wave front can be parallelized.

2. Introduction

This part of the paper discusses about basic concepts of technology related to parallel algorithms and the proposed dynamic programming solutions to LCS.

A. Smith-Waterman Algorithm

Smith-Waterman algorithm [5] is the most sensitive algorithm for local sequence alignment; that is for determining similar regions between two strings of protein or DNA. Smith Waterman provides a score of similarity between two sequences. This similarity score is sometimes referred to as the Smith Waterman score. Let A and B be the sequences to be aligned, where n and m are the lengths of A and B respectively. S (a, b) is the similarity score of the elements that constituted the two sequences. Wk is the penalty of a gap that has length k. A scoring matrix H is constructed and first row and column are initialized to 0. The size of the scoring matrix is (n+1) *(m+1). Note the 0-based indexing. Scoring matrix is filled with following recursion.

$H_{i,j}$ = max {$H_{i-1,j-1}$ + S (a_i, b_j), (north-west dependencies)
$H_{i-1,j}$ – W, (north dependency)
$H_{i,j-1}$ –W, (west dependency)
0} (1)

In equation (1), "i" is the current position in the sequence A, and "j" is the position in the sequence B. $H_{i,j}$ is the Smith Waterman similarity score for A[i] and B[j]. S (a_i, b_j) is a similarity score for a given residue combination as provided by a substitution matrix. A substitution matrix is a matrix which contains scores for every possible combination of residues. These scores are pre-determined by the life sciences community. If the maximum score for a given cell is based on the west or north cells, a gap is created thus causing a gap penalty (W) to occur. In the equation, a "zero" is added as the fourth parameter to prevent any cells in the matrix from going negative.

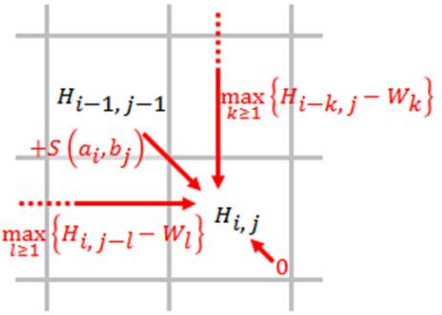

Fig. 1. [5] Shows the dependency and scoring method for single cell in Smith-Waterman Algorithm

B. Needleman-Wunsch Algorithm

Needleman-Wunsch [6] globally aligns two sequences and is often used in bioinformatics to align protein or DNA sequences.

To find the alignment with the highest score, a two-dimensional array (or matrix) F is allocated. The entry in row i and j is denoted by $F_{i,j}$. As the algorithm progresses, the $F_{i,j}$ will be assigned to be the optimal score for alignment of the first i=0……. n characters in A and the first j=1……. m characters in B. Recursion, based on the principle of optimality:

$F_{i,j}$ =max ($F_{i-1,j-1}$ + S (A_i +B_j), $F_{i,,j-1}$ + d, $F_{i-1,j}$ + d).

Scores for aligned characters are specified by a similarity matrix. Here, *S*(*a*, *b*) is the similarity of characters *a* and *b*. It uses a linear gap penalty, here called *d*.

C. CUDA

CUDA [7] is a parallel computing platform and programming model designed for NVIDIA for general computing on graphical processing units (GPUs). In GPU-accelerated applications, the sequential part of the workload runs on the CPU-which is optimized for single-threaded performance-while the compute intensive portion of the application runs on thousands of GPU cores in parallel. CUDA GPUs [14] are organized in multiprocessors, which group multiple streaming processors, the basic execution units (Fig. 3).

```
void increment_cpu(float *a, float
b,
   int N)
{
   for (int idx = 0; idx<N; idx++)
      a[idx] = a[idx] + b;
}
```

```
__device__
void increment_gpu(float *a, float
b,
   int N)
{
   int idx = blockIdx.x *
blockDim.x +
                threadIdx.x;
   if (idx < N)
      a[idx] = a[idx] + b;
}
```

Fig. 2. Plain C vs. CUDA code for implementing a simple code

CUDA executes the same program on all the multiprocessors: the code for the program (kernel) is the same but both the data and the execution flow can be different and diverge. CUDA launches multiple instances of the same kernel, called threads. Threads are grouped in warps for execution on a multiprocessor. Threads are runtime instances of the same kernel, and therefore they execute the same program code; furthermore, all the threads in a warp are executed by one multiprocessor in a fashion that they must execute exactly the same instruction at the same time, although on different data. If threads diverge (taking, for example, different branches of an if statement), they will be split into different warps, leading possibly to under-utilization of the multiprocessors.

Fig. 2. Shows a comparison between serial and parallel version of same algorithm. Fig. 3. Describes about threads and blocks.

Those applications that process large amounts of data or objects, and perform the same operations on all of them, will fit well on a GPU: to keep all the streaming processors busy, and therefore to obtain good performances, tens of thousands of threads need to be executed concurrently. Therefore, the applications based on the execution of disparate, short tasks will cause the fragmentation of warps and lead to the under-utilization of multiprocessors. Similarly, the applications that process a small subset of data at each time will fail in feeding the streaming processors with enough data. All the architectural details (threads, warps, multiprocessor, etc.) are hidden to the

end user; CUDA instead exposes the notions of blocks, grids and threads to ease the decomposition of the problem domain. As depicted in Fig. 3, threads are both the 'physical' and 'logical' basic units of execution; the GPU groups and schedules threads in warps, while CUDA offers a higher-level view of grids and blocks. Grids and blocks can be used by the programmer to map the subdivisions inherent in the problem domain (in particular, spatial subdivisions) in a convenient way. Each thread is then provided with variables representing the block and grid coordinates on which it needs to operate; using these coordinates, a thread can access and process a single item or subset of the problem domain. As an example, consider the simple and common scenario of porting computationally intensive loops to the GPU. In order to enable efficient execution, loops have to be transformed, strip-mining or unrolling them. After unrolling each thread executes a single, distinct iteration of the original loop. For instance, Table 2 shows a simple algorithm that takes a vector 'a' of length 'N' and a value 'b' and increments each value of 'a' by 'b'. As expected, the sequential algorithm on the left accesses the elements of 'a' one by one. Instead, the kernel code on the right spawns 'N' parallel threads, each of them incrementing a single value of 'a'. The position in the array 'a' that the thread T has to increment is obtained using a common pattern to compute a linear index: multiply the block index of T (blockIdx.x) by the number of threads per block (blockDim.x) and finally add the current index of T within the block (threadIdx.x).

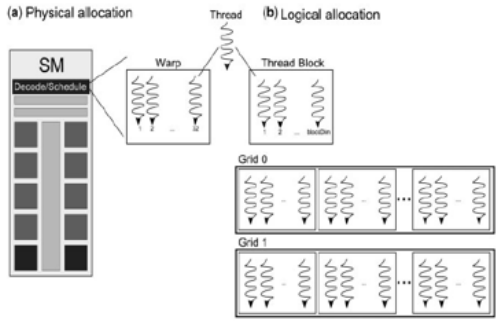

Fig. 3. Physical and logical allocation of a thread

3. Related work

Farrar presents an implementation in [10] which is written for Intel processors supporting SSE2 instructions. SSE2 instructions are a set of single instruction multiple data instruction sets (SIMD). These instructions allow for the usage of 128-bit wide SIMD registers, which Farrar utilized to speed up SW.

Manavski [11] implements SW on a GPU (NVIDIA GeForce 8800 GTX), and compares it against a serial version of SW and Farrar's implementation. Manavski uses a maximum query length of 567 characters in this implementation. It is at this length they achieve an execution time of 11.96(s) (30x speedup over serial version) using dual GPUs, and 23.32(s) using a single GPU. They report 20(s) of execution time for Farrar's SIMD implementation running on an Intel Quad Core processor. For both single and dual GPU configurations, Manavski utilizes the help of an Intel Quad Core processor by distributing the workload among GPU(s) and the Quad Core processor. Manavski's design with a single GPU configuration performs worse than Farrar's implementation. They achieve only two times speedup with the dual GPU configuration against Farrar's implementation which only uses a single processor core. This indicates that their program architecture is not fitting on the GPU architecture well.

There has been attempt to parallelise the algorithm despite being dependencies in stages through the concept of Linear Tropical Dynamic Programming and Rank Convergence [8]. They have achieved speedups of up to 80 times by using up to 128 cores. However, the speedup is not significant on commonly available GPUs.

Data-intensive applications are believed to be less well suited than arithmetic-intensive applications. Nevertheless, a highly data-intensive application MUMmerGPU [9] achieves significant speedup over the serial CPU-based application. A large part of this speedup is due to tuning techniques that may be used in any GPGPU application. The enormous volume of sequencing reads produced by next generation sequencing technologies demands new computational methods. Our software enables individual life science researchers to analyse genetic variations using the supercomputer hidden within their desktop computer.

4. Implementation

This part of the paper describes our approach for parallelizing the dynamic programming solutions to LCS problem. Here, we describe the parallelization of LCS done by Fine-grain Parallelism [12] also sometimes termed as wave front method. Using equation (1), The LCS problem can be solved using a two-dimensional memorization space F. Given by the formulation from equation (1), we can conclude that all elements are dependent on:

1) Element directly above F[i-1, j]
2) Element directly left F[I, j-1]
3) Element directly to northwest

F[i-1, j-1] This dependency creates a diagonal computation chain across the problem space, forming a wave front. The dependency and computation direction are shown in figure.

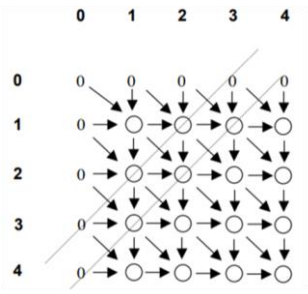

Fig. 4. Wave fronts [13] formed by dependencies in LCS problem

The parallel computation for LCS problem will assign each processing unit with a block of columns. Block size will be smaller than m/p, where m is number of columns and p is the number of processing units. Small blocks sizes will allow the columns to be cyclically assigned to the processors in round robin fashion. For sufficiently small block sizes, the uneven workload of each columns will be compensated by next cyclic distributed block.

5. Results

Exhaustive tests have been performed to compare performances of Smith-Waterman and Needleman-Wunsch algorithms on CUDA using anti-diagonal parallelization with the performances on multi-core CPU.

We have tested our solution on a workstation, with 2.71 GHz Intel Core i5-7200 processor and a single Nvidia GeForce 940MX graphic card which is capable of executing (1024x1024x64) blocks with 1024 threads each in a single launch.

By applying anti-diagonal parallelization, we observed that the speed-up increased for longer sequences as the overheads became negligible compared to the computation time. Moreover, the utilization of GPU resources increased with longer sequences.

A. Smith-Waterman Algorithm

We tested our solution on five DNA sequences whose length ranges from 5000 to 15000. The substitution function mentioned in the above section and a gap-penalty of 2 were used.

The table below shows the timings noted for computing the score matrix for Smith-Waterman algorithm for same length of strings.

Table 1
Timings noted for computing the score matrix for Smith-Waterman algorithm for same length of strings

Sequence Length	Execution Time CUDA(s)	Execution Time CPU(s)
5000	0.15	0.216
7000	0.243	0.501
10000	0.405	0.89
13000	0.536	1.44
15000	0.677	1.89

B. Needleman-Wunsch Algorithm

Similar five sequences with lengths ranging from 5000 to 15000 were tested with a gap penalty 1 and a substitution matrix mentioned in the above section.

Table 2
Timings noted for computing the score matrix for Needleman-Wunsch algorithm for same length of strings.

Sequence Length	Execution Time CUDA(s)	Execution Time CPU(s)
5000	0.188	0.251
7000	0.365	0.545
10000	0.74	1.47
13000	1.213	2.11
15000	1.522	2.54

The table shows the timings noted for computing the score matrix for Needleman-Wunsch algorithm for same length of strings.

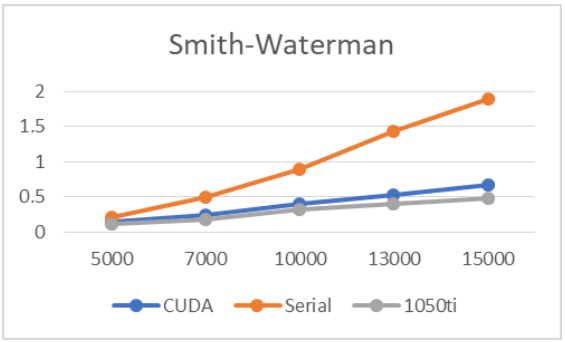

Fig. 5. Comparisons between serial and parallel version of algorithms for Smith-Waterman Algorithm

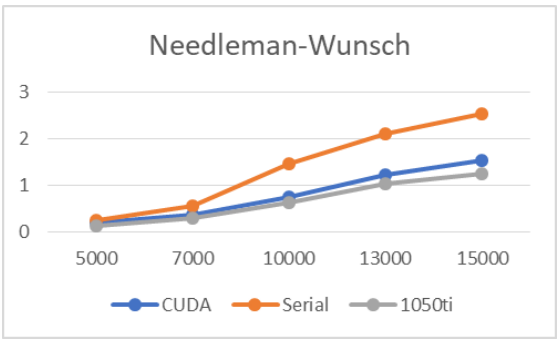

Fig. 6. Comparisons between serial and parallel version of algorithms for Needleman-Wunsch Algorithm

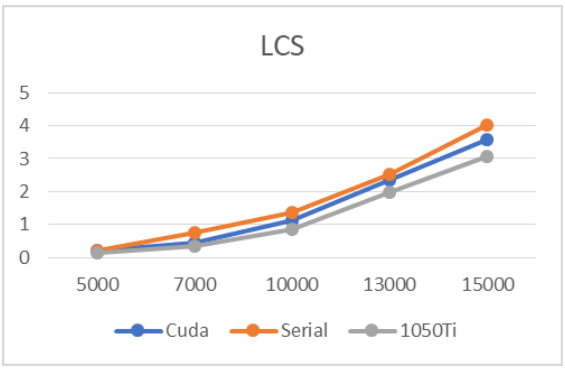

Fig. 7. Comparisons between serial and parallel version of algorithms for LCS Algorithm

The table below shows the timings noted for computing the score matrix for Smith-Waterman algorithm for different length of strings.

Table 3
Timings noted for computing the score matrix for Smith-Waterman algorithm for different length of strings

Sequence Length	Execution Time CUDA(s)	Execution Time CPU(s)
3000x6000	0.12	0.19
5000x10000	0.25	0.45
7000x14000	0.43	0.9
9000x18000	0.52	1.28

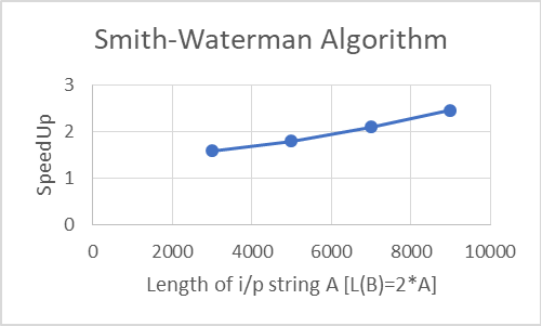

Fig. 8. Comparison between serial and parallel version of algorithms for Smith-Waterman Algorithm for different length of strings

The table below shows the timings noted for computing the score matrix for Needleman-Wunsch algorithm for different length of strings.

Table 4
Timings noted for computing the score matrix for Needleman-Wunsch algorithm for different length of strings

Sequence Length	Execution Time CUDA(s)	Execution Time CPU(s)
3000x6000	0.14	0.23
5000x10000	0.33	0.61
7000x14000	0.64	1.14
9000x18000	1.14	1.65

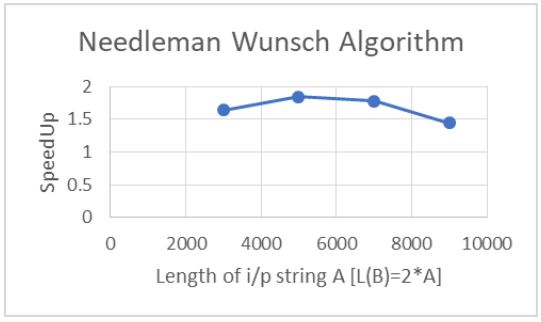

Fig. 9. Comparison between serial and parallel version of algorithms for Needleman-Wunsch Algorithm for different length of strings

6. Conclusion

Our implementation was able to achieve 2.5-3x speed-up in these algorithms. Further improvements can be achieved by better utilization of the GPGPU's processing power.

Even though dynamic programming algorithms are difficult to parallelize new methods such as wave front method can be used to parallelize it to an extent. Further developments in this area will open gateways to a wide range of applications in bioinformatics. One such development is Linear Tropical Dynamic Programming.

References

[1] J. Xiong, Essential Bioinformatics, Sequence Alignment. Cambridge, UK: Cambridge University Press, pp. 31-49, 2006.
[2] N. C. Jones, P. A. Pevzner, "An Introduction to Bioinformatics Algorithm, Dynamic Programming Algorithms, Longest Common Subsequence. Pp. 180-184, 2004.
[3] Computer Science Department of University of Maryland. [Online]. https://www.cs.umd.edu/class/fall2011/cmsc858s/Alignment.pd
[4] https://en.wikipedia.org/wiki/Dynamic_programming
[5] https://en.wikipedia.org/wiki/SmithWaterman_algorithm
[6] https://en.wikipedia.org/wiki/NeedlemanWunsch_algorithm
[7] https://developer.nvidia.com/about-cuda
[8] S. Maleki, M. Musuvathi, T. Mytkowicz, Parallelizing Dynamic Programming Through Rank Convergence
[9] Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23, pp. 156, 161, (2007).
[10] C. Trapneli, M.C. Schatz, Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.
[11] G.M. Striemer, A. Akoglu, Sequence Alignment with GPU: Performance and Design Challenges.
[12] Manavski, S.S., Valle, G.: Cuda compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 2008, 9(Suppl 2): S10 (2008).
[13] Engineering at Illinois. http://web.engr.illinois.edu/~snir/patterns/wavefront.pdf
[14] https://en.wikipedia.org/wiki/Graphics_processing_unit