GPU Accelerated Smith-Waterman Local Alignment

Stephen Hwang

University of California, Santa Cruz
`sjhwang@ucsc.edu`

Abstract. Sequence alignment is an important task in bioinformatics used to identify regions of conservation between two sequences. The Smith-Waterman local sequence alignment algorithm is one of the original sequence alignment algorithms, but its *O(mn)* complexity poses practical issues with larger sequences. Altering the Smith-Waterman algorithm, can allow it to be ran parallel utilizing modern GPU architecture resulting in increased performance over the original sequential dynamic programming method.

Keywords: Sequence Alignment · Smith-Waterman · GPU

1 Introduction

Sequence alignment is central bioinformatics task used in identifying conservation and relationships between sequences of nucleotides or proteins in order to identify regions of similarity. The Smith-Waterman algorithm performs a local alignment in which segments, instead of whole sequence, are compared. This makes it unlike its predecessor the Needleman-Wunsch algorithm, which serves a different purpose with a similar approach. As a dynamic programming algorithm, the Smith-Waterman algorithm, developed by Temple F. Smith and Michael S. Waterman in 1981, yields the optimal local alignment between two sequences given a desired scoring system [5].

String searching across biological sequences is computationally intensive with Smith-Waterman at *O(mn)* complexity, where m and n are the sequence lengths. At larger sequence sizes. complexity and runtime are important to consider, especially as researchers are often aligning thousands of pairs of sequences at a time [4] with nanopore reads over 100 kb. Making use of modern computational resources, the Smith-Waterman algorithm can be altered to run parallel across GPU architecture and reap potential performance benefits.

2 Smith-Waterman Algorithm

2.1 CPU Implementation

The Smith-Waterman algorithm aligns pairs of sequences based off of matches, mismatches, insertions, and deletions with scores specified by the user. Insertions and deletions introduce gaps, represented by dashes while aligned sequences are paired, as shown below:

TACGGGCCCGCTA–C
TA—–G–CC—–CTATC

Smith-Waterman Algorithm: Let $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$ be two sequences to be aligned with lengths n and m respectively. Given a substitution matrix and scoring scheme, $s(a_i, b_j)$ is the similarity score and W is the gap penalty. Common scoring schemes involve gap penalties as -2 and match/mismatch scores as 3.

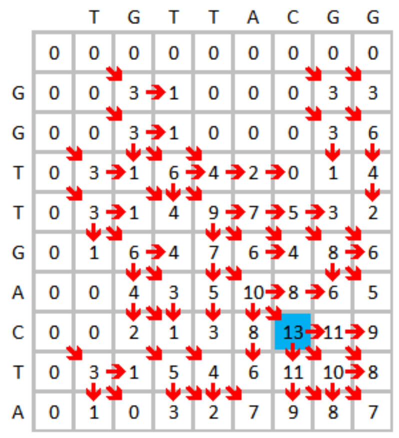

Fig. 1. Filled scoring matrix with highest score in blue. Red arrows are encoded in separate direction matrix D

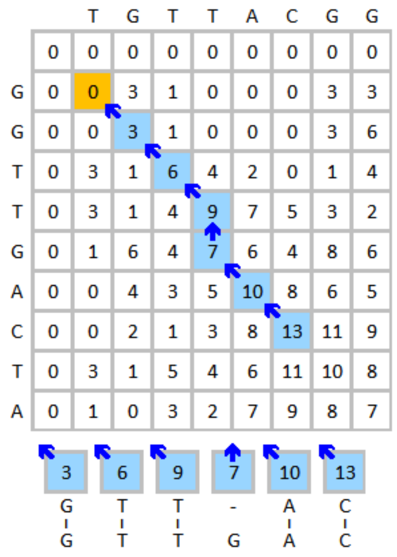

Fig. 2. Traceback of the direction matrix for optimal local alignment

Initialize Matrices: Construct the scoring matrix H and direction matrix D, each of size $(n + 1) * (m + 1)$ and initialize the first row and column as 0.

For $0 \leq i < m$, $0 \leq j < n$,

$$H_{i,0} = H_{0,j} = 0 \tag{1}$$

Fill the Scoring Matrix: Fill the scoring matrix using equation 2 while recording the direction the current cell receives its value from (Fig. 1). The direction recorded is biased towards favoring a match score over gap scores when scores are tied to favor the longest possible local alignment. By recording the direction in a separate direction matrix, it simplifies the traceback step to merely following the direction in which a cell received its data from.

For $1 \leq i < n$, $1 \leq j < m$,

$$H_{i,j} = max \begin{cases} H_{(i-1),(j-1)} + s(a_i, b_j) \\ H_{i,(j-1)} - \mathrm{W} \\ H_{(i-1),j} - \mathrm{W} \\ 0 \end{cases} \tag{2}$$

Traceback the Direction Matrix: Traceback the sequence from the highest score in the filled matrix H and ending at a score of 0 by using the filled direction matrix D (Fig. 2). Speed of the traceback within the matrix can be be increased by actively searching for the max value during the matrix filling, such that computation is not wasted on iterating over the matrix H during traceback.

2.2 GPU Implementation

Each cell $H_{i,j}$ depends on the three preceding cells left, above, and upper-left diagonal ($H_{(i),(j-1)}$, $H_{(i-1),(j)}$,$H_{(i-1),(j-1)}$). For each anti-diagonal (iterating diagonals from upper left to bottom right of the scoring matrix), the cells are independent. For example, the cells at $H_{1,3}$, $H_{2,2}$, $H_{3,1}$ rely on cells $H_{1,2}$, $H_{1,1}$, $H_{2,1}$ but not on other cells in the same anti-diagonal. Using this fact, and by iterating over the anti-diagonals of the matrix, the Smith-Waterman algorithm can be parallelized within each anti-diagonals. This still requires the previous two diagonals to be complete, but can afford a speed-up across each diagonal.

		G	C	T	G	T	G	C	A
	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0
C	0	0	3	1	0	0	0	0	0
A	0	0	1	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0
G	0	3	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	0	0	0	0

Fig. 3. Computation of the 7th anti-diagonal in which each anti-diagonals only relies upon the previous two diagonals and are independent of other elements in the same diagonal.

One way to construct the code involves a high number of memory transfers between host and device code in which each iteration of an anti-diagonal copies memory back and forth. The sub-matrix to be iterated over is constructed on the host and transferred to the device for parallel computation [1, 3]. Threads are synced after each iteration. This may be faster for large sequences, but this approach poses major drawbacks as this requires $(max(n, m) + 1) * 2$ memory transfers of matrices and $(max(n, m) + 1)$ kernel launches. There is major overhead and operations are bandwidth bound.

My improved method involves the same iterations over anti-diagonals and parallization of each anti-diagonal, but attempts to minimize memory transfer of matrices. This method involves passing the index of the anti-diagonal $(= m+n+1)$ and using this diagonal index number coupled with the thread id to yield the proper anti-diagonal index positions (i, j). Once the index positions are known, calculation and filling of the scoring and direction matrix is identical to the CPU code. This is a implementation of the gather communication scheme; data at each cell is updated using three preceding cells of determined position. The complexity is still $O(mn)$, but by parallel computation within diagonals, working complexity is reduced by diving this into threads. Drawbacks still include launching a kernel every iteration, causing unnecessary overhead, but memory transfer is reduced. Additional optimizations are covered within the discussion.

3 Results

Timing of the CPU implementation used chrono while timing of the GPU implementation used cudaEvents. In previous iterations, the traceback function called a function to iterate over the matrix to locate the maximum and location of the maximum. This was integrated into respective the fill CPU function and GPU kernel and allowed a marginal performance increase, more noticeable with larger sequences. Previously the equivalent GPU code involved a max reduce kernel, but was proven inefficient compared to simply integrating the search while filling the matrices. Both CPU and GPU tests involved using the same randomly generated sequence A and sequence B and used the same scoring scheme of a constant gap penalty of -2 and similarity match/mismatch scores of +3 and -3 respectively.

Table 1. CPU vs GPU runtimes

Sequence Size	CPU Runtime (ms)	GPU Runtime (ms)
16	0.0220	0.261696
32	0.0416	0.5889
64	0.1035	1.257
256	1.289	4.828
1024	18.84	23.31
2048	75.57	45.46
8192	1192	280.1

The above table display runtimes (ms) for the CPU and GPU implementations with sequences sizes $(m = n)$. *Note:* matrix sizes are larger than $m * n$ (see Initialize Matrices).

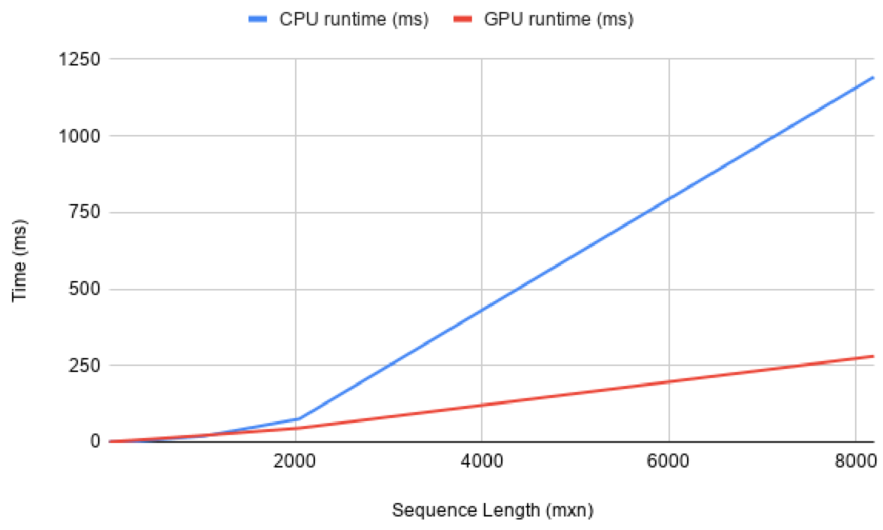

Fig. 4. CPU (blue) and GPU (red) runtimes in mm for equal length sequences $m = n$.

Across square $(m = n)$ matrices, the runtimes range from 0.2 ms to 1192 ms on the CPU and 0.26 ms to 280 ms on the GPU. As pictured in Fig. 4 and Table 1, at small sequence lengths, the CPU implementation is faster than the GPU implementation. This is due to the over head required to launch a kernel and sync threads each iteration, but for larger sequence sizes (starting around $1024 < n = m < 2048$) the GPU wins over the CPU approach. As demonstrated, the GPU Smith-Waterman implementation is more scalable, and, as sequences length increases – on average 13 kb - 20 kb for next-gen nanopore sequencing reads – it is important to have the ability to scale. At small n, the faster CPU is negligible; instead what matters is the practicality of running such an algorithm on real biological-scale data.

4 Discussion

The traceback algorithm is the same between CPU and GPU implementations. This was optimized by integrating the search for the max value and index into filling the matrix. The existing max_score_cpu function is still usable for the CPU and GPU, but is obsolete from integrating the search into the matrix fill function. Additionally, By transferring the max value and index from the GPU kernel, it removes the need to copy the filled scoring matrix from the device to the host, which may prove marginally faster for large sequences.

Other optimizations may include launching one kernel with threads equal to the max size of the anti-diagonal and from that one kernel iterate over the all the anti-diagonals [6]. Drawbacks of this approach may include the 1024 cap on threads and most of the threads will remain inactive for most of the computations – only one iteration will have optimal thread use.

Another approach is an altered form of the anti-diagonal Smith-Waterman where the calculations are performed in parallel one row at a time. This allows for coalesced memory access and maintains threads actively active [2]. This method is a parallel scan Smith-Waterman able to ran across multiple GPUs to achieve massive speed-ups.

To further improve my code I would first minimize the number of kernel launches. This method may involve the launching of one kernel and adopt a similar approach described by Khajeh-Saeeda, et. al. By implementing a parallel scan across rows, kernels can be kept fully in use and have memory coalesced.

5 Conclusion

The Smith-Waterman local sequence alignment is an important algorithm, but one that suffers from complexity and scalability issues. By utilizing modern GPUs and altering the algorithm to run parallel across anti-diagonals, at large sequence size, parallel performance surpasses that of the traditional CPU approach. Although when iterating across the anti-diagonals of a scoring matrix, there is high overhead with consecutive kernel launches and syncs, this method proves reliable for larger sequences. Iterating on the basic Smith-Waterman algorithm yields compounding marginal increased performance including reduced memory transfers and integrated max score searches. Although Smith-Waterman is not the modern golden standard of sequence alignment, its principles and uses still persist across bioinformatics today.

Appendix

Code may be found at https://github.com/StephenHwang/gpu_smith_waterman and includes all necessary functions to run and time the GPU and CPU implementations with a randomly generated sequence.
Note: Comment out printing and output functions for accurate CPU time and block size may be adjusted depending on sequence size.

References

1. Baker M., Welch A., Venkata M.: Parallelizing the Smith-Waterman Algorithm using OpenSHMEM and MPI-3 One-Sided Interfaces. Oak Ridge National Laboratory
2. Khajeh-Saeeda A., Pooleb S., Perota J.,: Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors. Journal of Computational Physics (2010)
3. Liao, H., Yin, M., Cheng Y.: A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. vol 2. The 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, San Francisco, CA (2004)
4. Mount DM.: Bioinformatics: Sequence and Genome Analysis (2nd ed.). Cold Spring Harbor Laboratory Press: Cold Spring Harbor, NY. (2004).
5. Temple S., Waterman, M.:Identification of Common Molecular Subsequences. Journal of Molecular Biology. 147 (1): 195–197.(1981).
6. Venkatachalam, B.: Parallelizing the Smith-Waterman Local Alignment Algorithm using CUDA. (2012)