Parallelizing the Smith-Waterman Local Alignment Algorithm using CUDA

Balaji Venkatachalam

February 28, 2012

1 The Local Alignment problem

1.1 Introduction

Given two strings $S_1 = pqaxabcstrqrtp$ and $S_2 = xyaxbacsl$, the substrings $axabcs$ in S_1 and $axbacs$ in S_2 are very similar. The problem of finding similar substrings is the *local alignment problem*.

Local alignment is extensively used in computational biology to find regions of similarity in different biological sequences. Similar genetic sequences are identified by computing the local alignment of a given sequence against a number of other genetic sequences. Protein molecules fold into unique 3-dimensional shapes. Different regions fold into various shapes – helices, sheets etc. These shapes determine the function of the proteins. Local alignment helps identify the various regions of structural similarity. BLAST and FASTA are two of the programs that compute the local alignment of a sequence against a database of other genetic sequences.

Formally, given a scoring scheme that includes a cost for matching a pair of characters and inserting a character in one sequence (equivalently, introducing a *gap* in the other sequence), a *local alignment* of strings S_1 and S_2 is a pair of substrings s_1 of S_1 and s_2 of S_2 whose score is maximum over all possible substrings of S_1 and S_2 for the scoring scheme.

Unlike the global alignment problem where the entire strings are to be matched, the local alignment problem identifies highly similar substrings. Also, unlike the edit distance problem, where the goal is to minimize the cost of transforming one sequence to another, the local alignment problem identifies highly similar substrings.

1.2 The Smith Waterman Algorithm

For strings of length m and n an $O(mn)$-time dynamic programming (DP) algorithm was designed by Smith and Waterman [1]. The DP has the following recurrence:

$$v(i,j) = \max \begin{cases} 0, \\ v(i-1,j-1) + cost(S_1(i), S_2(j)), \\ v(i-1,j) + cost(S_1(i), -), \\ v(i,j-1) + cost(-, S_2(j)) \end{cases}$$

with $v(0,0) = 0$ and $v(0,j) = 0$ for $j = 1, \ldots, n$ and $v(i,0) = 0$ for $i = 1, \ldots, m$. Here $v(i,j)$ denotes the optimal cost for substrings $S_1[1, \ldots, i]$ and $S_2[1, \ldots, j]$ and $cost(S_1(i), S_2(j))$ is the cost of (mis)-matching characters $S_1(i)$ and $S_2(j)$ and $cost(S_1(i), -)$, and $cost(-, S_2(j))$ are costs for creating a gap against characters $S_1(i)$ and $S_2(j)$ respectively. The largest $v(i,j)$ value in the table represents the local alignment,

The DP table has $(m+1) \times (n+1)$ $v(i,j)$ entries. The table can be filled either row-wise or column-wise and each entry can be filled in constant time. Thus the total time for the DP algorithm is $O(mn)$.

An exact match receives the highest positive score and substitution of similar characters also receives a positive score while highly dissimilar characters are penalized with negative scores. BLOSUM and PAM are two popular substitution matrices with costs for comparing amino acids. In practice, an affine cost function is used for weighting gaps, with a cost G_{init} for starting a gap and G_{ext} for extending the gap. The Smith Waterman algorithm can be easily modified to fit the affine cost function, but we will not consider it in this exposition for simplicity.

See Gusfield [2] for the correctness of the algorithm and a discussion of various cost functions for gaps.

2 Parallelizing in CUDA

2.1 The general idea

Filling a $v(i,j)$ entry depends on three preceding cells – $v(i-1,j-1), v(i-1,j)$, and $v(i,j-1)$. Each of these entries depend on their preceding entries. Due to these dependencies, a naive approach of breaking the table into parts that are computed by different threads independently is not feasible.

The dependencies for filling various cells are shown in figure 1. Notice that $v(1,3)$, $v(2,2)$ and $v(3,1)$ depend on the completion of $v(1,1), v(2,1)$, and $v(1,2)$ but are independent of each other. Therefore these three values can be computed simultaneously in parallel. In general, all the elements of the anti-diagonal depend on the previous anti-diagonal but are independent of each other and can be computed in parallel.

Instead of filling a row (or a column) of the table in every iteration as discussed above, here we will fill an anti-diagonal in every iteration. And the cells of the anti-diagonal are filled in parallel. The scheme for parallel programming is shown in figure 2.1.

There are $n+m-1$ iterations, one for each diagonal. The longest diagonals are length m, and there are $n-m+1$ such diagonals. Therefore using $p = m$

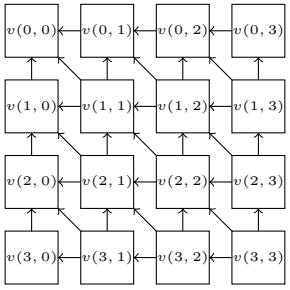

Figure 1: The arrows represent the dependencies for various cells.

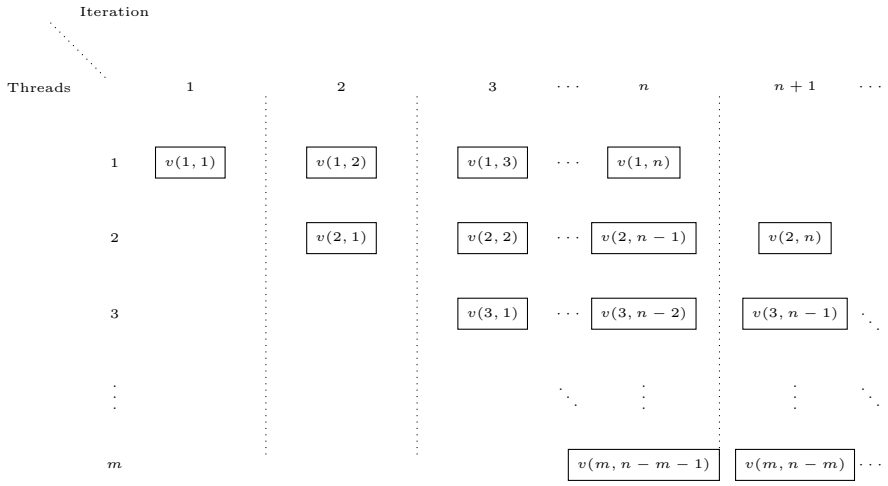

Figure 2: Pictorial representation of the computation done by threads in every iteration. Vertical dotted lines represent synchronization.

threads will achieve the most throughput. However, there are $p(p-1)$ stalls as threads will be idle in the computation.

2.2 Engineering optimizations in CUDA

- Since a query sequence is compared with multiple sequences in the database, the individual sequence comparisons is easily parallelized. The sequences in the database are partitioned to load-balance, i.e., every process has an equal length of sequence to compute.

- In the parallel computing scheme shown above, every thread computes one value and waits for synchronization. Since memory accesses to read and write variables and synchronization between iterations take longer than the computation, assigning multiple rows per thread, minimizes the overhead.

- Instead of storing individual integer values of the database table four values can be read and written at once using short vectors. The advantage is more pronounced when using the affine cost function.

- Query Profile: Accessing the various $cost(\cdot, \cdot)$ values in the substitution matrix on successive iterations will be slower due to the random access pattern. This can be speeded up by creating a query-specific profile for the entire database.

  Every possible character in the domain is matched with successive characters of the query sequence and the result is stored as a profile. That is, in the profile the ith row has the ith character of the query sequence matched against all possible characters. Consecutive rows of this profile are accessed in successive iterations, thus replacing the random access of the substitution matrix with sequential access of the profile.

- The query profile is constant and computed once for the entire computation, This is stored in texture memory to exploit caching in GPU. Similarly, the query and database sequences can be stored in either texture memory or constant memory for fast access.

- Pipelining loads and computations. Loading individual database strings and launching the kernel can be pipelined. That is, the first string is loaded to memory and while the kernel is executing further strings can be loaded into memory. Thus the performance does not drop by idle GPUs.

The discussion above is from various papers [3, 4, 5, 6] on GPU implementations of the Smith Waterman algorithm. The papers used affine cost function for gaps. They report significant speedup (up to 10x in a few cases) compared to various serial implementations.

References

[1] T. F. Smith and M.S. Waterman: Identification of common molecular subsequences. *Journal of Molecular Biology* vol. 147, pp.195-197, 1981.

[2] Dan Gusfield: Algorithms on Strings, Trees and Sequences. *Cambridge university Press*, 1997.

[3] Yang Liu, Wayne Huang, John Johnson and Sheila Vaidya: GPU accelerated Smith-Waterman. in V.N. Alexandrow et al. (Eds.), *ICCS 2006, LNCS 3994*, pp. 188– 195, 2006.

[4] Svetlin A. Manavski and Giorgio Valle: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics,* 9(Suppl 2):S10, 2008.

[5] Yongchao Liu, Douglas L. Maskell and Bertil Schmidt: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2:73, 2009.

[6] Yuma Munekawa, Fumihiko Ino and Kenichi Hagihara: Accelerating Smith-Waterman Algorithm for Biological database Search on CUDA-Compatible GPUs. *ICICE Transactions on Information and Systems*. Vol. E93-D, No. 6, June 2010.