A Tile-based Parallel Viterbi Algorithm for Biological Sequence Alignment on GPU with CUDA

Zhihui Du[1+], Zhaoming Yin[2], and David A. Bader[3]
[1]Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, 100084, Beijing, China
+Corresponding Author's Email: duzh@tsinghua.edu.cn
[2]School of Software and Microelectronics, Peking University, 100871, China.
Email： zhaoming_leon@pku.edu.cn
[3]College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA.

Abstract **The Viterbi algorithm is the compute-intensive kernel in Hidden Markov Model (HMM) based sequence alignment applications. In this paper, we investigate extending several parallel methods, such as the wave-front and streaming methods for the Smith-Waterman algorithm, to achieve a significant speed-up on a GPU. The wave-front method can take advantage of the computing power of the GPU but it cannot handle long sequences because of the physical GPU memory limit. On the other hand, the streaming method can process long sequences but with increased overhead due to the increased data transmission between CPU and GPU. To further improve the performance on GPU, we propose a new tile-based parallel algorithm. We take advantage of the homological segments to divide long sequences into many short pieces and each piece pair (tile) can be fully held in the GPU's memory. By reorganizing the computational kernel of the Viterbi algorithm, the basic computing unit can be divided into two parts: independent and dependent parts. All of the independent parts are executed with a balanced load in an optimized coalesced memory-accessing manner, which significantly improves the Viterbi algorithm's performance on GPU. The experimental results show that our new tile-based parallel Viterbi algorithm can outperform the wave-front and the streaming methods. Especially for the long sequence alignment problem, the best performance of tile-based algorithm is on average about an order magnitude faster than the serial Viterbi algorithm.**

Keywords hidden Markov model; Viterbi algorithm; GPGPU

I. INTRODUCTION

Biological Sequence alignment is very important in homology modeling, phylogenetic tree reconstruction, sub-family classification, and identification of critical residues. When aligning multiple sequences, the cost of computation and storage of traditional dynamic programming algorithms becomes unacceptable on current computers. Many heuristic algorithms for the multiple sequence alignment problem run in a reasonable time with some loss of accuracy. However, with the growth of the volume of sequence data, the heuristic algorithms are still very costly in performance.

There are generally two approaches to parallelize sequence alignment. One is a coarse-grained method, which tries to run multiple sequence alignment sub-tasks on different processors, for example, see [1, 2, 3], or optimize single pair-wise alignment consisting of multiple parallel sub-tasks, such as [4, 5, 6]. The communication among those sub-tasks is critical for the performance of coarse-grained method. The other is a more fine-grained method, which focuses on parallelizing the operations on smaller data components. Typical implementations of fine-grained methods for sequence alignment include parallelizing the Smith-Waterman algorithm (see [7, 8]) and the Viterbi algorithm [9].

The Hidden Markov Model (HMM) includes three canonical problems: evaluation, decoding, and learning [9]. Typical sequence alignment approaches often involve the last two problems. The learning problem [10, 11] often needs less time compared with the decoding problem — Biological Sequence Alignment Oriented Viterbi Algorithm [12]. Based on our experimental results on Satchmo [13] (a multiple sequence alignment tool), the Viterbi algorithm occupies more than approximately 80% of the total execution time. Therefore, it is critical to design an efficient method to speed up the Viterbi algorithm.

General-Purpose Computation on Graphics Processing Units (GPGPU) [14, 15] provides a powerful platform to implement the parallel fine-grained Biological Sequence Alignment Viterbi algorithm. With the rapidly increasing power and programmability of the GPU, these chips are capable of performing a much broader range of tasks than their graphics niche. The GPGPU technology is employed in several fine-grained parallel sequence alignment applications, such as in [16, 17]. In our study, we employ a new tile-based mechanism into the "wave-front" method [22] to accelerate the Viterbi algorithm on a single GPU. In general, our contributions are as follows:

1) We provide the design, implementation, and experimental study, of a tile-based mechanism based on the wave-front method. This method can enable the alignment of very long sequences. For comparison, we also introduce the wave-front and streaming methods used in parallelizing Smith-Waterman algorithm into the GPU implementation of Viterbi algorithm.
2) We divide the basic computational kernel of the Viterbi algorithm of each tile into two parts, independent and dependent parts. The parallel and balanced execution of independent computing units in a coalesced memory-accessing manner can significantly improve the performance of the Viterbi algorithm on a GPU.

The rest of this paper is organized as follows: in section 2, we introduce the basic idea of the CUDA language used in GPU programming and related work on CUDA accelerated sequence alignment applications. In section 3, we describe the Viterbi Algorithm for Biological Sequence Alignment in detail. In section 4, our design and implementation of the wave-front, streaming and tile-based algorithms on a GPU are presented. The experiments are presented in section 5. In section 6, we discuss future work.

II. Background and Related Work

A. GPU Programming with CUDA

Recent graphics architectures provide tremendous memory bandwidth and computational horsepower at a very inexpensive price compared with the same amount of computing capability of traditional CPUs. Meanwhile, because of its design principle for arithmetic intensity, the speed of graphics hardware is increasing quickly, nearly doubling every six months. The stream programming model provides a compelling solution for applications that require high arithmetic rates and data bandwidths [18].

Compute Unified Device Architecture (CUDA) introduced by NVIDIA [15] is a programming environment for writing and running general-purpose applications on the NVIDIA GPU's, such as GeForce, Quadro, and Tesla hardware architectures. CUDA allows programmers to develop GPGPU applications using the extended C programming language instead of graphics APIs. In CUDA, threads are organized in a hierarchy of grids, blocks, and threads, which are executed in a SIMT (single-instruction, multiple-thread) manner; threads are virtually mapped to an arbitrary number of streaming multiprocessors (SMs). In CUDA, the wider memory is shared, the slower it is accessed. Therefore, how to organize the memory access hierarchy in the application is the golden rule for CUDA programmers.

B. Parallelizing Smith-Waterman algorithm on GPU

The Smith-Waterman algorithm [8] for sequence alignment is a dynamic programming method and one of the first Biological Sequence Alignment algorithms implemented on a GPU. The principles and technologies used in parallelizing Smith-Waterman on a GPU are also applicable to the acceleration of HMM-based algorithms.

Liu *et al.* [19] explore the power of GPU using the OpenGL graphic language. They introduce a way to overlap the communication (data transmitted between CPU and GPU) and the computation. Using this method, they overcome the GPU memory limitation problem. They also provide a solution to the Multiple Sequence Alignment problem on a GPU. Munekawa *et al.* [17] and Cschatz [16] propose the implementation of Smith-Waterman on GPU using CUDA. Munekawa *et al.* improve Liu's work and take advantage of the on-chip memory to improve the performance. Cschatz's parallelization strategy is coarse-grained and the sequence length is limited by memory assigned to each thread.

C. Parallelizing HMM based algorithm on GPU

ClawHMMER [5, 21] is an HMM-based sequence alignment for GPUs implemented using the streaming Viterbi algorithm and the Brook language. ClawHMMER parallelizes a general-purpose Viterbi algorithm, which permits only one loop over the insertion state. It has some efficiency limitations that we will discuss later. In order to improve its throughput, ClawHMMer concurrently issues several sequence alignment tasks on a multi-node graphics cluster.

Our method differs from ClawHMMER in that we use the Biological Sequence Alignment oriented Viterbi algorithm, which is an order of magnitude faster than the general Viterbi. Based on the features of the Viterbi algorithm for biological sequence alignment, we introduce the *wave-front* like parallelizing strategy, which matches the streaming architecture of the GPU very well. At the same time, we propose additional methods to optimize the performance.

III. Viterbi Algorithm

A. Hidden Markov Model (HMM)

An HMM [9] is a quintuple; it can be described as follows:

$$HMM = \{O, S, A, B, \pi\}$$

O stands for the set of observations; S stands for the set of states; A stands for the probability of transferring from one state to another; B stands for the probability of emitting observations given one state; and π stands for the probability of an initial state.

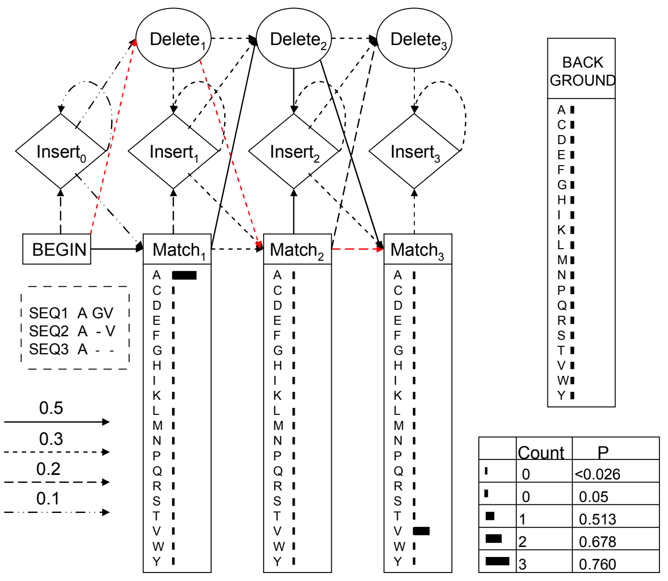


Figure 1. A typical HMM model built from a sequence profile, Match$_i$ stands for the match state of the ith profile column, as do Delete$_i$ and Insert$_i$. There is a probability assigned to each state-transition of every column, which is represented by different type of line. For every match state, there are emit probabilities for each of 20 amino acids in accordance with that state. These probabilities are represented by different lengths of bars. For delete and insert states, emit probabilities are the background probabilities trained from massive amounts of data in the real world.

Fig. 1 shows a typical hidden Markov model built on a sequence profile of length 3 consisting of three aligned sequences. In this figure, 1) the observation stands for the residue(s) in a specific column of the profile, which includes one or a combination out of 20 amino acid residues. 2) There are three possible states for each columns of a profile—Match, Delete and Insert.

States can move to other states via the arrows, and probabilities are assigned to each state transition. In addition, once a match state of a column is decided, there will be an emitting probability of the corresponding residue. And if the delete and insert states are assigned to a column, there will be a "null" emit probability assigned to the residue which is uniform for every column (see the upper right of Fig. 1). For example, in Fig. 1, the match state of column 1 is 50%, and the emit probability of A under the match state is 76%.

B. Viterbi Algorithm

The Viterbi algorithm is used to solve the so-called "decoding" problem, which could be described as: *Given a set of observations, find a route of their matching states, which will maximize the probability of observations*.

For an event that consists of n observations, if one observation is mapped to m states, there will be m^n possible routes. The Viterbi algorithm employs the dynamic programming method. Suppose $\delta(t, i)$ is the probability to be calculated of state i for observation t (here $\delta(t, i)$ is formally called the forward variable), then the Viterbi algorithm can be briefly described as:

```
for: observation $O_t$ in Observations:
    for: state $S_i$ in states of observation $O_t$:
        for: state $S_j$ in states of previous observation $O_{t-1}$:
            $\delta(t, i) = Max [ \delta(t-1, j) \bullet a_{ji}] \bullet b_i(t)$
```

This is a general description of the Viterbi algorithm, and the time complexity is $O(n\, m^2)$, where n stands for the number of observations and m is the number of possible states of an observation.

C. Viterbi Algorithm for Biological Sequence Alignment

The task of using an HMM to align sequences can be described as: *Given a template profile (an HMM is built from template profile), and a target profile (which is to be aligned to template profile), find an optimal route of states assigned with each column of the template and align the template to target in accordance with the states in the optimal route*. For example, in Fig. 1, if the target is GV, the red line marks the route of aligning template to target, and the aligned result for target is: - GV

The Insert state is special in building HMM's for the purpose of biological sequence alignment, because Insert states can loop over themselves (as Fig. 1 shows). Since each Insert state can consume one residue of the target, theoretically, the cycle length for an Insert state's loop is as long as the length of the target profile. When looping over Insert states, an HMM cannot jump from one column to another, so these Insert states combined with the match and delete of that column, form the "combinational states" of that column.

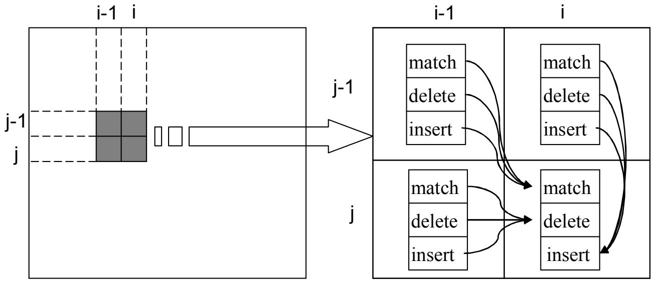


Figure 2. Foundation of the biological sequence alignment oriented Viterbi algorithm. The rectangle on the left represents the whole matrix to be computed by the Viterbi algorithm, and the right side of the figure shows the process of updating a single block of the matrix.

There are two single state: "match" and "delete" and twice the number of target's length "combinational states" for a template column. It will be very expensive if we use the general-purpose Viterbi algorithm to count the alignment. Instead, we use the Viterbi algorithm tailored for Biological Sequence Alignment to solve the redundant computation of the "insert self-loop" problem of the general-purpose Viterbi algorithm. It reduce the time complexity from $O(m^2\, n)$ to $O(len\_t \bullet len\_a)$, where *len_t* stands for the length of template and *len_a* stands for the length of target. For the details of this algorithm please refer to [12].

As Fig. 2 shows, the fundamental task of the Viterbi algorithm for Biological Sequence Alignment is to calculate a matrix consisting of $len\_t \bullet len\_a$ blocks, where each block consists of three states: Match, Delete and Insert, and the value for each state of a block is dependent on the value of previous block. The Match state depends on the upper-left block; the Delete state depends on the left block and the Insert state depends on the upper block. Suppose $\delta(mn, i)$ is the maximal probability of state i for a block in the matrix of column m and row n, then the Viterbi algorithm for Biological Sequence Alignment can be described as follows:

```
for: $Block_{mn}$ in Matrix:
    for: state $S_i$ in three states of $Block_{mn}$:
        for: state $S_j$ in three states of dependent Block:
        //when $S_i$ is Match,the dependent block is $Block_{(m-1)(n-1)}$
        // when $S_i$ is Delete, the dependent block is $Blcok_{(m-1)n}$
        //when $S_i$ is Insert the dependent block is $Blcok_{m(n-1)}$
            calculate $\delta(mn, i)$
```

Because there are $len\_t \bullet len\_a$ blocks, and each block has three states where each state refers to three other states of the dependent block, then the time complexity for this algorithm is $O(len\_t \bullet len\_a)$. This is an order of magnitude faster than general Viterbi algorithm used in biological sequence alignment.

IV. Methods to Parallelize The Biological Sequence Alignment Oriented Viterbi Algorithm

A. Wave-front Pattern to Implement the Viterbi Algorithm

The wave-front algorithm is a very important method used in a variety of scientific applications. The computing procedure is similar to a frontier of a wave to fill a matrix, where each block's value in the matrix is calculated based on the values of the previously-calculated blocks. On the left-hand side of Fig. 3 we show the wave-front structure for parallelizing the Viterbi algorithm. Since the value of each block in the matrix is dependent on the left, upper, and upper-left blocks, in the figure, blocks with the same color are put in the same parallel computing wave-front round. The process for this wave-front algorithm is shown in Pseudo code 1; we call it a simple implementation of the wave-front algorithm.

The right-hand side of Fig. 3 shows the data skewing strategy to implement this wave-front pattern. This memory structure is useful because blocks in the same parallelizing group are adjacent to each other (they are marked with same color in Fig. 4). In this way, because data accessed by neighbor threads are organized adjacent to each other, threads could access memory in a more efficient manner [15].

There are three functions in the simple implementation, *Initmatrix ()*, *InitHMM ()* and *traceback ()*, and the process of these three functions are shown in Fig. 4. *Initmatrix ()* sets up the dynamic programming matrix of the Viterbi algorithm for Biological Sequence Alignment. We use the color red to mark the three initial states as the termination for trace back, which can unify at each parallel round of computation. *InitHMM ()* sets up the probability model for template sequence. We use pseudo-count methods to train the HMM in parallel. *Traceback ()* is used to align the template to target according to the value of the matrix calculated. This process is executed on the GPU because the cost of transmitting the DP matrix back to main memory is high if the trace back is done on the CPU.

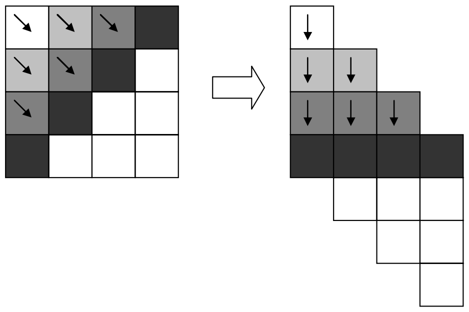


Figure 3. Wave-front structure of Viterbi Algorithm for Biological Sequence Alignment. The left-hand side of the figure shows the wave-front process, and right-hand side of the figure shows the memory data skewing to implement the wave-front algorithm. For the detail of this method, please refer to [6].

Pseudo Code 1: DoWaveAlign (seq_temp, seq_tar)

```
Initmatrix()
InitHMM()
for: round_r in all rounds:
  parallel_for: block_mn in blocks of round_r:
    for: state S_i in three states of Block_mn:
      for: state S_j in three states of dependent Block:
      //when S_i is Match,the dependent block is Block_(m-1)(n-1)
      // when S_i is Delete, the dependent block is Blcok_(m-1)n
      //when S_i is Insert, the dependent block is Blcok_m(n-1)
              do:
          calculate δ(mn, i)
      // δ(mn, i) stands for the forward variable for the i^th
       state of Block_mn
Traceback()
```

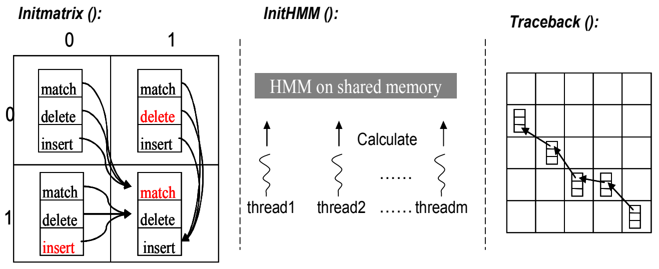


Figure 4. Example of three functions in the simple wave-front implementation.

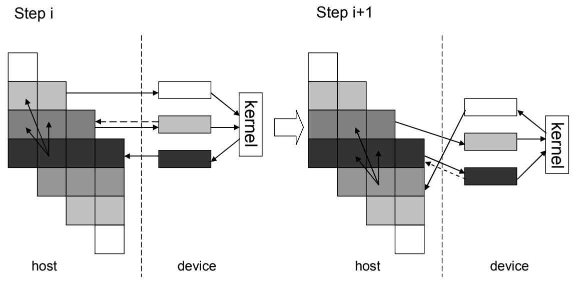


Figure 5. The HMM matrix is updated asynchronously at the host CPU and GPU device. The solid and dashed arrows represent the asynchronous execution between host CPU and GPU device.

B. Streaming Viterbi Algorithm for Biological Seq. Alignment

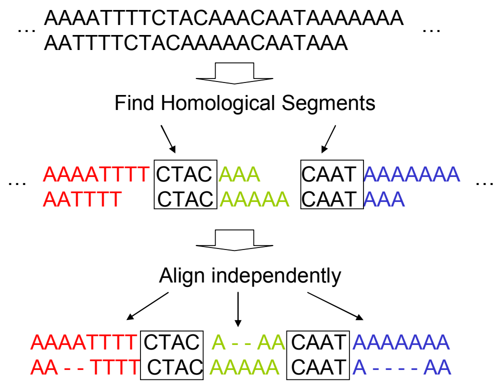
A straightforward implementation of the wave-front pattern has one deficit—when the sequence length is too long. Since the size of the Dynamic Programming (DP) matrix is $O(len\_t \cdot len\_a)$, not all the data can be held in the GPU's memory at one time. Here, we introduce a streaming method, which has already been used successfully in parallel Smith-Waterman algorithms. Because the Viterbi algorithm for biological sequence alignment has similar data dependencies, it is possible to apply this method as well to the parallel Viterbi algorithm to solve the GPU memory limitation problem.

In CUDA, a stream is a sequence of instructions that execute in order. Different streams, on the other hand, may execute their instructions asynchronously [15]. This feature ensures that the execution between the host and GPU device can be overlapped with each other. In the implementation of the Smith-Waterman algorithm, the sequence length supported by this mechanism is longer because only three rounds are needed to place the sequence into the GPU's memory.

Fig. 5 shows the process of computing the DP matrix for the streaming parallel Viterbi algorithm. Only three rounds of communication are needed to load the matrix block into the GPU memory and compute the appropriate kernel. The full DP matrix is stored in the host memory. This mechanism needs the values of the other two rounds to calculate the values of current round. When performing computation, data from the round which had been previously processed, will be transferred back into the host memory concurrently. Pseudo code 2 describes this streaming approach. Note that procedures marked with the same stream[r] will execute instructions independently.

Pseudo Code 2: DoStreamingAlign(seq_temp, seq_tar)

```
Initmatrix()
InitHMM()
InitStream(numOfRounds)
for: round_r in all rounds:
  stream[r]_parallel_for: blocks_mn of round_r:
    for: state S_i in three states of Block_mn:
      for: state S_j in three states of dependent Block_mn:
         do:
          calculate δ(mn, i)
  stream[r]_do: transfer group_(g-1) back to host
Traceback()
```


... AAAATTTTCTACAAACAATAAAAAAA ...
AATTTTCTACAAAAACAATAAA
Find Homological Segments
... AAAATTTT CTAC AAA CAAT AAAAAAA ...
AATTTT CTAC AAAAA CAAT AAA
Align independently
AAAATTTT CTAC A - - AA CAAT AAAAAAA
AA - - TTTT CTAC AAAAA CAAT A - - - - AA
Figure 6. Using homological segments to divide long sequences.

C. Tile Based Method to Harness the Power of GPU

One drawback of the streaming Viterbi algorithm is its high dependency on the computational resources of the host side, and the transition time between host and GPU device cannot be neglected either. The streaming Viterbi algorithm requires additional storage and communication bandwidth. Therefore, we introduce the new *tile-based* method, which simplifies the computational model and also handles very long sequences with less memory transmission.

The tile-based method can be described as follows: *If a matrix can be fully loaded into the GPU memory, do it; otherwise divide the large matrix into tiles to ensure that each tile fits in the GPU's memory as a whole and then calculate the tiles one by one or in parallel.*

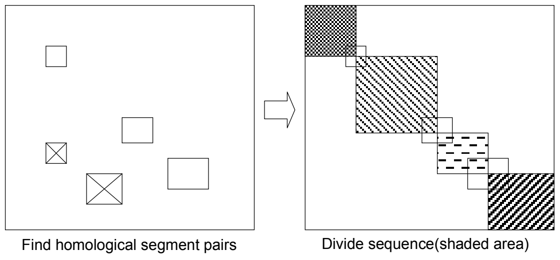


Figure 7. Example of finding homological segment pairs and using them to divide a large matrix into smaller, independent tiles. In the left-hand diagram, homological sequences form small pieces and are aligned using the Dynamic Programming method, and un-aligned homological tiles are marked with an "X". In the right-hand diagram, aligned tiles are used to divide large matrix into small sub-matrices.

Cell-Swat [22] presents a tiling mechanism, but with this method, there are data dependencies among each tile. We introduce the *homological segments* concept into our tiling mechanism to eliminate the data dependency among different tiles. The biological meaning of homological segments makes them serve as separators very well (as shown in Fig. 6). To find homological segments, there are algorithms such as Fast Fourier Transform (FFT) [23] based algorithms and k-mer based algorithms [24]. We choose a k-mer based method for its ease of implementation. When all the homological segments are found, we use a dynamic programming algorithm to align these segments and cut the long sequence from the middle of these homological segments, as shown in Fig. 7. For the details of homological segments, please refer to [23]. The process of finding homological segments and using them to divide independent tiles is as follows:

1) Find all homological segments whose score exceeds the threshold.
2) Using a dynamic programming algorithm to align these homological segments, ignoring those homological segments which are not aligned (such as in the left-hand side of Fig. 7, the homological segments which are not aligned are marked with an "X").
3) Using homological segments to divide the full dynamic programming matrix into small tiles (marked with the dash area).
4) Calculate the tiles with the Viterbi algorithm.

D. Optimization Methods

Another deficiency of the wave-front Viterbi Algorithm for Biological Sequence Alignment is the unbalanced thread load; there are some idle threads at the beginning and the end of the algorithm. We solve this problem by transforming the following formula, which is used to calculate one block.

$$\delta_t(j) = MAX_{1 \leqslant i \leqslant N} [\delta_{t-1}(i)\, a_{ij}(t)\,]\, b_j(O_t)$$
$$= MAX_{1 \leqslant i \leqslant N}\, \delta_{t-1}(i)\, [a_{ij}(t)\, b_j(O_t)\,] \quad 1 \leqslant j \leqslant N$$

In the transformed formula, the calculation of $a_{ij}(t)b_j(O_t)$ does not have any data dependency; therefore, we can calculate $a_{ij}(t)b_j(O_t)$ initially and store the results in a temporary memory.

To express this idea more clearly, we present this method in Fig. 8. In our method, we first perform the formula transformation. Before transforming the formula, the inner computation of one block is tightly dependent. This is because the current round is dependent on its previous rounds and only after the previous rounds have finished can it start. After formula transformation, the calculation is divided into two parts, the independent part and the dependent part. We collect all the independent parts together and let them run in parallel (the first step). At this stage, the threads are load balanced and the coalesced-memory optimization is employed. The dependent part of the computation is moved to step two, and it uses the data previously computed by the independent computation.

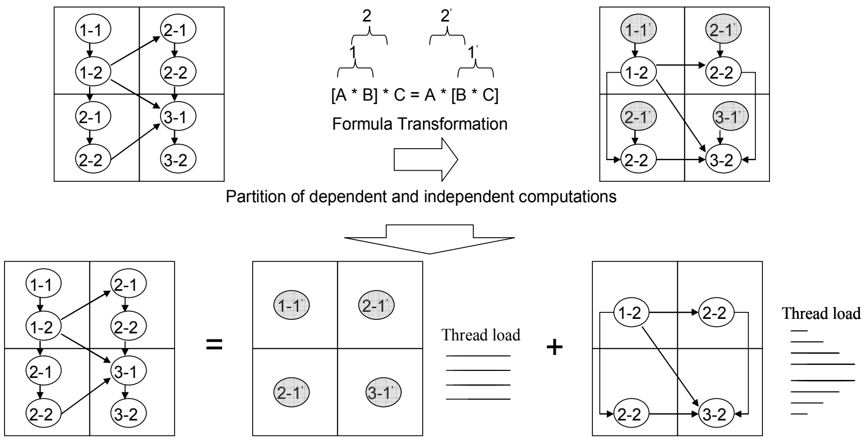


Figure 8. Thread load. The left side of this figure shows the wave-front Viterbi Algorithm for Biological Sequence Alignment and right side shows how the transformed formula can balance the thread load.

Pseudo Code 3: DoTileSchedule(seq_temp, seq_tar)

```
If length(seq_temp) • length(seq_tar) < threshold:
    DoTileAlign(seq_temp, seq_tar)
Else:
    Tiles[] ← splitTiles(seq_temp, seq_tar)
    For tile[i] in Tiles[]:
        DoTileAlign(tile[i] → subTemp, tile[i] → subTar)
```

Pseudo Code 4: DoTileAlign(sub_temp, sub_tar)

```
Initmatrix()
InitHMM()
ComputeTemporaryData()
for: $round_r$ in all rounds:
    parallel_for: $blocks_{mn}$ of $round_r$:
        for: state $S_i$ in three states of $Block_{mn}$:
            for: state $S_j$ in three states of dependent Block:
                do: calculate $\delta(mn, i)$
Traceback()
```

V. Experimental Results

The experiments are performed on the platform which has a dual-processor Intel 2.83GHz CPU with 4 GB memory and an NVIDIA Geforce 9800 GTX GPU with 8 streaming processors and 512MB of global memory. We tested using two operating systems: Windows XP and Linux Ubuntu version 10. To focus on the algorithmic efficiency in our study, we made two simplifications in our experiments, one is that we use a pseudo count method to train the HMM, and another is that we neglected the accuracy comparison with other methods and our parallel method has the same accuracy with the serial method. We employ the automatic sequence-generating program ROSE [25] to generate different test cases.

A. General Test

The general test was executed on four implementations of the Viterbi Algorithm for Biological Sequence Alignment. The first one is the serial implementation; the second is the simple wave-front implementation; the third is the streaming implementation; the last one is our tile-based implementation.

TABLE I. Performance Comparism of Four Different Viterbi Implementations

Seq-Length		Execution Time (Second)/Speedup						
		serial	*Simple Wave-front*		*Streaming*		*Tile-based*	
100	DW	0.73	0.37	1.97	0.38	1.92	0.28	2.61
	RW	0.017	0.007	2.42	0.02	0.85	0.006	2.83
	DL	0.063	0.008	7.87	0.023	2.74	0.007	9
	RL	0.027	0. 007	3.86	0. 023	1.17	0. 007	3.86
200	DW	2.34	0. 39	6	0. 44	5.32	0. 39	6
	RW	0.05	0. 03	1.67	0. 061	0.82	0. 028	1.79
	DL	0.324	0. 035	9.26	0. 065	4.98	0. 029	11.17
	RL	0.142	0. 035	4.06	0. 065	2.18	0. 029	4.9
300	DW	5.89	0. 42	14.02	0. 46	12.8	0. 43	13.7
	RW	0.12	0. 068	1.76	0. 1	1.2	0. 055	2.18
	DL	0.647	0. 07	9.26	0. 112	5.78	0. 054	11.98
	RL	0.283	0. 068	4.16	0. 116	2.44	0. 054	5.24
400	DW	9.93	0. 50	19.86	0. 52	19.1	0. 45	22.07
	RW	0.21	0. 13	1.61	0. 159	1.32	0. 098	2.14
	DL	1.112	0. 12	9.27	0. 2	5.56	0. 099	11.23
	RL	0.485	0. 122	3.98	0. 174	2.79	0. 097	5
500	DW	15.9	0. 54	29.44	0. 54	29.44	0. 52	30.58
	RW	0.34	0. 19	1.78	0. 239	1.42	0. 174	1.95
	DL	1.783	0. 198	9	0. 262	6.8	0. 155	11.5
	RL	0.783	0. 191	4.10	0. 251	3.12	0. 153	5.12
1000	DW	62.1	0. 99	62.73	1. 10	56.45	0. 86	72.21
	RW	1.34	0. 64	2.09	0. 686	1.95	0. 554	2.42
	DL	6.98	0. 64	10.91	0. 725	9.63	0. 53	13.17
	RL	3.07	0. 635	4.83	0. 62	4.952	0. 512	6.0

In the table, the first line of a group is the results for Debug-Windows mode (DW), second line, Release Windows (RW), third line, Debug Linux (DL), Fourth line, Release Linux (RL).

We have compiled and run our test programs under four different versions. The first is Windows-Debug; the second is Windows-Release; the third is Linux-Debug and the last is Linux-release. For long sequences, the simple wave-front version cannot load the entire dynamic programming matrix into the GPU memory. Therefore, we select groups of sequences which have lengths under 1000 to test all of the versions. The experimental results are shown in Table 1.

The results show that the best acceleration rate is achieved under the Windows-Debug mode, and we can see that the speedup under debug mode is better than release mode. This is because in the serial version, the compiler's optimization methods have a great effect on the manner of accessing memory; however, in CUDA, this is not true. Because CUDA has a special hierarchy of memory structure and the access time for different levels of the hierarchy greatly varies. For example, global memory has an access time of about 500 cycles and the on-chip memory such as a register of only 1 cycle. Due to the reason that CUDA does not provide a good mechanism to optimize memory accesses at a compiler level, the speedup in Debug mode is better than in Release mode. Also we can see that among three versions of the algorithm, the tile-based method is the best and the streaming algorithm is the worst. In fact, since the length of the test sequences is not long, the only difference between simple- and tile-based algorithms is that tile-based algorithm introduces some optimization methods. The results show that our optimization methods are effective.

B. Test of Streaming Viterbi Algorithm

Since there is data communicated between host and device memory, we must consider the communication cost. Here we test the time composed of computing and data transfer of the streaming implementation. The results are shown in Fig. 9.

In Fig. 9, we see that the longer the sequence is, the less percentage of time consumed in data transfer. This means that the computation and communication can be overlapped with each other better when the sequence length is longer. However, these time of communication still can not be neglected.

C. Test of Tile-based Viterbi Algorithm

The selected homological segments and the size of the sub-sequence will significantly affect the algorithm's efficiency. We implement the serial algorithm of finding homological segments in Java. In our implementation, the length of the sub-sequences partitioned by homological segments is decided by the following three factors:

1) K-mer window length: When this is larger, the homological segment found will be better; however this also means more computation and storage cost.
2) Homological-segment window length: With the growth of the homological-segment window length, there will be less homological segments detected, which means the sub-sequence length will be longer.
3) Homological-segment threshold: This threshold is used to score a segment of a sequence. The larger the threshold is, the fewer the number of homological segments will be detected.

In our test, we focus on how the average length of the sub-sequence will affect the final performance. Our tests are divided into two parts, the time of computing homological segments and the time of sequence alignment using our tile-based algorithm. The test results are shown in Fig. 10. The time for computing homological segments is linearly increased. We do not include the time for computing homological segments in Fig. 10 because we want to show the time changes for computing sub sequences separately.

From Fig. 10, we see that with the growth of the average sequence length, the number of the segments decreases and the time for sequence alignment increases (there are some waves in the figure; this is because of the variation of sequence length for the given average length). We select the parameters, which are used to decide how to partition the sequence length of average length, to keep the residues length at 200. This is because when the average sequence length exceeds 200, the time of sequence alignment will increase much faster.

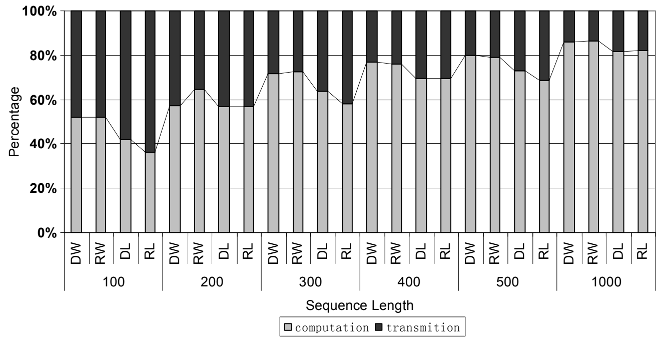


Figure 9. Results of the test on streaming Viterbi algorithm implementation for Biological Sequence Alignment.

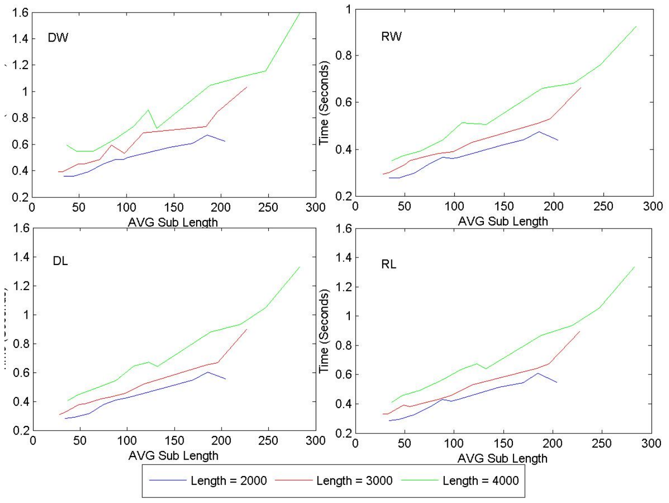
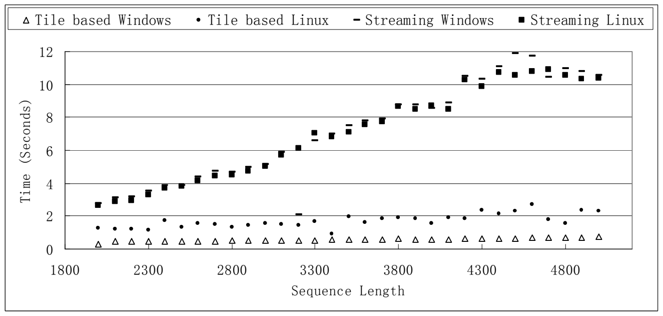


Figure 10. Results of testing the tile-based Viterbi Algorithm for Biological Sequence Alignment. Here we include only the time for computing sub sequences



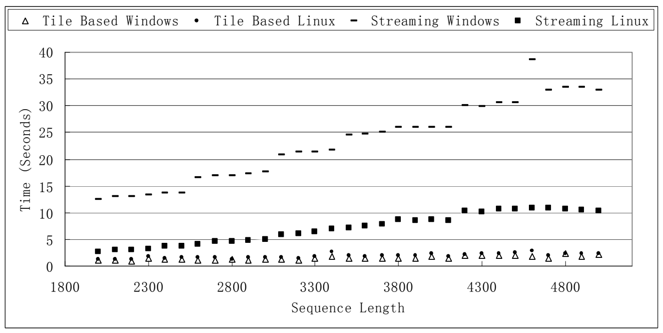


Figure 11. Results of testing on longer sequences, upper graph shows the results in Release mode and lower graph shows the results in Debug mode.

D. Testing of Long Sequence

The last test shows the comparison of streaming and tile-based implementations on longer sequences. For the running time of tile-based algorithm, we included both the time for computing tiles and the time for the parallelized Viterbi algorithm. Fig. 11 shows that the tile-based implementation is an order of magnitude faster than the streaming implementation, because it only needs to calculate a portion of the dynamic programming matrix. In addition, the time growth of the tile-based implementation is lower. From Fig. 11, we see that as the residue length increases from 2000 to 5000, the time for tile-based method increases only about 2 seconds under both the Linux and Windows system, and the streaming-based algorithm grows at least 8 seconds. As the sequence length increases, the growth of the calculation time needed by our tile-based method is very low.

There are two special effects in the figure to explain. One is that for the tile-based method, the performance under Windows is faster than the Linux system, and it is different for streaming algorithm. The other is that the time for the streaming algorithm under the Windows system grows like a ladder. Both of these effects are because of the fact that, in Windows, the kernel initiated by CUDA cannot run too long consistently. Thus, we added some code to allow the kernel to sleep for some time for every 1000 kernel initiations.

VI. Conclusion and Future Work

The experimental results show that the best performance of our parallel Viterbi algorithm can achieve one order of magnitude of speedup comprised with the serial implementation of the Viterbi algorithm; in addition, both streaming-based and tile-based implementations are better than the simple wave-front implementation. In future work, we will use the CUDA-enabled Viterbi Algorithm for Biological Sequence Alignment in the Multiple Sequence Alignment and Sequence database search problem. For these two problems, more computing resources will be needed. When adding multiple GPU's into our work, there will be difference between the streaming-based and the tile-based algorithm. Theoretically, the tile-based algorithm should be easier to achieve scalable performance due to its few needs of communication between host and device.

ACKNOWLEGEMENT

This paper is partly supported by National Natural Science Foundation of China (No. 60773148), Beijing Natural Science Foundation (No. 4082016), NSF Grants CNS-0614915 and OCI-0904461, and an IBM Shared University Research (SUR) award.

REFERENCES

[1] K. Jiang, O. Thorston, A. Peters, B. Smith, C.P. Sosa. "An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence-Search on a Massively Parallel System" IEEE Transactions on Parallel and Distributed Systems, Vol.19(No.1) (2008)

[2] M. Ishikawa, T. Toya, M. Hoshida, K. Nitta, A. Ogiwara, M. Kanehisa. "Multiple sequence alignment by parallel simulated annealing" Comput. Appl. Biosci. 1993 Jun; 9(3):267-73.

[3] S.-I. Tate, I. Yoshihara, K.Yamamori, M. Yasunaga. A parallel hybrid genetic algorithm for multiple protein sequence alignment. Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress 12-17 May 2002 Volume: 1, On page(s): 309-314.

[4] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, GPU Accelerate Smith-Waterman, Proc. Int'l Conf. Computational Science (ICC 06) pp.188-195,2006

[5] R. Horn, M. Houston, P. Hanrahan. ClawHMMer: A streaming HMMer –search implementation. Proc. Supercomputing (2005).

[6] W. Liu, B. Schmidt, G. Voss, W. Muller Wittig. "Streaming Algorithms for Biological Sequence Alignment on GPUs" IEEE TPDS, Vol. 18, No. 9. (2007), pp. 1270-1281.

[7] D. Feng and R. F. Doolittle. "Progressive sequence alignment as a prerequisite to correct phylogenetic trees". J. Mol. Evol., 60:351-360, 1987.

[8] T.F. Smith, M.S. Waterman. "Identification of Common Molecular Subsequences". Journal of Molecular Biology 147: 195–197 (1981)

[9] L.R Rabiner "A tutorial on hidden Markov models and selected applications in speech recognition".In Proceedings of the IEEE, Vol. 77, No. 2. (06 August 2002), pp. 257-286.

[10] K. Sjolander, K. Karplus, M. Brown, R. Hughey, A. Krogh, I.S. Mian, and D. Haussler. "Dirichlet Mixtures: A Method for Improving Detection of Weak but Significant Protein Sequence Homology".

[11] M. P. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjolander, and D. Haussler. In L. Hunter, D. Searls, and J. Shavlik, editors, Using Dirichlet mixture priors to derive hidden Markov models for protein families. Proc. of First Int. Conf. on Intelligent Systems for Molecular Biology , pages 47--55, Menlo Park, CA, July 1993. AAAI/MIT Press.

[12] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, Biological sequence analysis: probabilistic models of proteins and nucleic acids, Cambridge University Press, 1998.

[13] R.C. Edgar and K. Sjolander. "SATCHMO: Sequence alignment and tree construction using hidden Markov models", In Bioinformatics 19(11), 1404-1411 (2003).

[14] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, A System for Programming Graphics Hardware in C-Like Language, ACM Trans. Graphics, vol.22, pp. 896-907 ,2003.

[15] CUDA Compute Unitfed Device Architecture, Programming Guide,Version 2.0. NVIDIA Corp. 2008.

[16] S.A. Manavski, G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics. 2008 Mar 26;9 Suppl 2:S10

[17] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. 8th IEEE International Conference on BioInformatics and BioEngineering, pages 1 C6, Oct .200

[18] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, Vol. 26, No. 1. (March 2007), pp. 80-113.

[19] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller Wittig, BioSequence Database Scanning on a GPU, Proc. 20th IEEE Int¯ Parallel and Distributed Processing Symp .(High Performance Computational Biology (HiCOMB) Workshop), 2006.

[20] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, GPU ClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment, Proc. 13th Ann. IEEE Int'l Conf. High Performance Computing (HiPC06) pp.363-374,2006.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman , K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware (2004) ACM Trans. On Graphics.

[22] S. Aji, F. Blagojevic, W. Feng, D.S. Nikolopoulos. Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine Proceedings of the 2008 ACM International Conference on Computing Frontiers 13-22

[23] M. Katoh and M. Kuma. "MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform". In Nucleic Acids Res. 30:3059-3066 2002.

[24] F. Jiang. "Multiple Sequence Alignment based on k-mer and FFT" graduation thesis, Xidian University 2006 (in Chinese)

[25] J. Stoye, D. Evers and F. Meyer. "Rose: generating sequence families". In Bioinformatics. 1998;14(2):157-163