cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification

Chuan Liu

May 6, 2009

1 Overview

Hidden Markov model (HMM) as a sequential classifier has important applications in speech and language processing [Rab89] [JM08] and biological sequence analysis [Kro98].

In this project, we analysis the parallelism in the three algorithms for HMM training and classification, i.e. forward algorithm, Viterbi algorithm, and Baum-Welch algorithm, for graphical processing units (GPU). Based on the analysis, we implement a prototype program for HMM training and classification on the NVIDIA CUDA platform. The evaluation shows for forward algorithm, our CUDA implementation achieves performance of 23.3 GFLOP/s and has up to 800× speedup over the implementation based on single core CPU; and for Baum-Welch algorithm, the performance is 4.3 GFLOP/s and there is also 200× speedup over CPU implementation. We also note our implementation is not fully optimized: several parallelism found during analysis is not implemented due to time and complexity. We expect more sophisticated implementation can further exploit the GPU computing ability.

The remaining sections are organized as follows. In Section 2, we first give a brief introduction to hidden Markov model, followed by description of the three most important algorithms for HMM and our analysis of their parallelism. In Section 3, we describe our implementation in details. In Section 4, we evaluate the implementation by experiments. The experimental results are shown and discussed. Section 5 is conclusion.

2 Parallel Design

In this section, we first give a brief review of HMM and introduce our notations for HMM (Section 2.1). In the next three subsections (Section 2.2, 2.3, and 2.4), we will go through the three algorithms for HMM. Our parallel design for each of the algorithm follows the analysis of the algorithm in each section respectively.

2.1 The Hidden Markov Model

A hidden Markov model is a Markov process with states generated observations. Both the Markov process and observation generation model can be either discrete or continuous-time. In scope of this project, we only focus on discrete time systems.

The discrete time Markov process, often named Markov chain, is a kind of probabilistic finite state automaton, where input can go from one state to another with a probability. The Markov

$S = s_1 s_2 \cdots s_N$	a set of N states.
$V = v_1 v_2 \cdots v_{\|V\|}$	a set of distinct observations symbols. $\|V\|$ denotes the number of distinct observations.
$O = o_1 o_2 \cdots o_T$	a sequence of T observations; each drawn from the observation symbol set V.
$Q = q_1 q_2 \cdots q_T$	a sequence of state; q_t denotes the state at time t.
$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & & & \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix}$	a transition probability matrix A; each a_{ij} representing the probability of moving from state s_i to state s_j, i.e. $a_{ij} = P(q_{t+1} = s_j \| q_t = s_i)$
$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1\|V\|} \\ b_{21} & b_{22} & \cdots & b_{2\|V\|} \\ \vdots & & & \\ b_{N1} & b_{N2} & \cdots & b_{N\|V\|} \end{bmatrix}$	an emission probability matrix B, each b_{ij} representing the probability of the observation v_j being generated from a state s_i, i.e. $b_{ij} = P(o_t = v_j \| q_t = s_i)$
$\pi = [\pi_1, \pi_2, \cdots, \pi_N]$	an initial state distribution: $\pi_i = P(q_1 = S_i)$

Figure 1: The notations for HMM.

definition makes the assumption that the probability of a state transition only depends on the previous state. HMM is Markov chain with outputs. In HMM, an output is produced after a state transition. Given only the outputs, the underlying Markov chain is hidden from the observer. Based on the influential tutorial [Rab89], we formalize our definition of discrete HMM with the notations in Figure 1. With further reference to Jack Ferguson, [Rab89] introduces the following three fundamental problems of interest to HMM.

Problem 1 (Computing Likelihood): Give an HMM $\lambda = (A, B)$ and an observation sequence O, determine the likelihood $P(O|\lambda)$.

Problem 2 (Decoding): Given an observation sequence O and an HMM $\lambda = (A, B)$, discover the best hidden state sequence Q.

Problem 3 (Learning): Given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B.

Problem 1 is solved by forward algorithm; problem 2 is solved by Viterbi algorithm; and problem 3 is solved by Baum-Welch algorithm. The three algorithm are closely related. The Viterbi algorithm is like a variance of forward algorithm. The Baum-Welch algorithm first passes through the output with forward algorithm; then follows a backward pass with a reversed version of forward algorithm; and finally the model parameters are estimated based on the result from the two passes, therefore the first part of the algorithm is sometimes named forward-backward algorithm.

2.2 The Forward Algorithm

To compute the likelihood, the forward algorithm constructs a trellis α of size $T \times N$ for the sequence of T observations and N underlying states. The trellis can be viewed as a matrix, where each cell $\alpha(t, i)$ is the probability of being in state i while seeing the observations until t.

A scratch of forward algorithm is shown in the pseudocode below. The input to the algorithm is a sequence of observations O. The output is the likelihood probability for the observation. The algorithm makes the assumption the first observation in sequence is the start state, and the last observation is the end state. The complexity of this algorithm is $O(N^2T)$.

```
FORWARD(O)
1  initialize all cells of α to 0
2  α(o_1, s) ← 1
3  for t = o_2 to o_T
4        do for i = 1 to N
5              do for j = 1 to N
6                    do p ← a_ji · b_it
7                       α(t, i) ← α(t, i) + α(t − 1, j) · p
8  likelihood ← α(o_T, e)
9  return likelihood
```

Line 3 of the algorithm loops over the entire sequence of the observations. For each observation t, we compute the $\alpha(t, i)$ probabilities for all the states i. The computation is depend on previous column in the trellis ($\alpha(t-1)$ at line 7). The dependence among columns in the trellis make the loop hard to parallilize.

Iterations of the **for** loops at line 4 have no dependence among each other. The code can be parallized follows task decomposition. In Figure 2, we shows a simplified example, where there are only 3 states. The first iteration of the **for** loop will compute the three dashed arcs. Each arc adds a quantity p to the state s_1 the arc points to. The other two tasks will compute arcs point to s_2 and s_3. In the paralleled algorithm, all the tasks can be carried out simultaneously, i.e. all dashed arcs will be computed by one thread.

Further notice each task is merely a summation of array of quantities. Therefore the task can be carried out in a parallel reduction manner [Har08].

By now our discussion assumes there is only one sequence of observation. Suppose we have a number of sequences of observations and all sequences are independent. This is not uncommon in practice. For a total number M of sequences, the time complexity goes to $O(MN^2T)$. Taking advantage of the data independence, we can add another level of parallelism to the algorithm. At each stage of the trellis, we can compute the state probabilities for all sequences in parallel. This design follows data parallel pattern. Suppose all threads in our design can execute in constant time in parallel, the time complexity will be reduced to $O(cNT)$, where c is the execution time of the parallel code.

2.3 The Viterbi Algorithm

Instead of computing the overall observation probability, the Viterbi algorithm finds the most likely path of states that generate the observations. The difference between Viterbi and forward

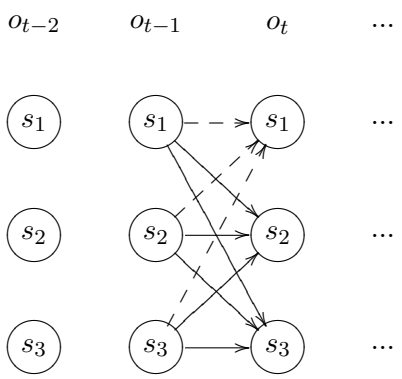

Figure 2: Execution one iteration of line 3 of the forward algorithm in a 3 state HMM.

algorithms is at line 7 of the psedocode. Instead of sum over all α probabilities in previous column, Viterbi algorithm finds the maximum one and keep a pointer to trace the state that leads to the maximum probability. The pseudocode for Viterbi algorithm is given in Figure 3. The input to the algorithm is a sequence of observations and output is a sequence of most likely states that generate the observation.

```
VITERBI(O)
 1  initialize all cells of α to 0
 2  α(o_1, s) ← 1
 3  for t = o_2 to o_T
 4        do for i = 1 to N
 5              do for j = 1 to N
 6                    do p ← a_ji · b_it
 7                       if α(t − 1, j) · p > α(t, i)
 8                          then α(t, i) ← α(t − 1, j) · p
 9                               backpointers(t, i) ← j
10  states = BACKTRACE(backpointers)
11  return states
```

Figure 3: The psedocode of Viterbi algorithm.

Line 4 - 9 of the algorithm can be parallized the same way as forward algorithm. The BACKTRACE takes linear time and cannot be parallized. Since the main loop (line 3) cannot be parallized neither, BACKTRACE will not effect the running time of the Viterbi algorithm.

2.4 The Baum-Welch Algorithm

The forward-backward algorithm takes sequences of observations as input and estimate the A and B values that maximize the probability for the given observations. The algorithm runs iterations

over the input data and terminate until convergence or certain threshold condition is met, e.g. number of iterations, difference in parameter changes.

The algorithm takes two passes over the data. In the first pass, the algorithm uses forward algorithm to construct α probabilities. In addition to the α probabilities, the algorithm runs a similar backward algorithm to construct β probabilities. The backward probability $\beta(t, i)$ is the probability of seeing observation from o_{t+1} to the end, given that we are in state j at time t.

Based on the α and β probabilities, we can compute the expected number (counts) of transitions from state i to state j at a given observation t as:

$$\xi(i, j) = \frac{\alpha(t-1, i)\beta(t, j) \cdot a_{ij} \cdot b_{jt}}{likelihood}$$

and the expected number of times in state j and observing t can be computed as:

$$\gamma(j, t) = \frac{\alpha(t, j)\beta(t, j)}{likelihood}$$

where *likelihood* is the observation probability as computed by forward algorithm.

We list part of the pseudocode for forward-backward algorithm in Figure 4. The α probabilities are updated after calling the FORWARD function at line 2. The remaining code performs backward pass and accumulates ξ and γ counts.

```
FORWAD-BACKWARD-EXPECTATION(O)
 1  initialize all cells of α, β, γ, ξ to 0
 2  likelihood ← FORWARD(O)
 3  β(o_T, e) ← 1
 4  for t = o_T to o_1
 5       do for i = 1 to N
 6            do γ(t, i) ← γ(t, i) + (α((t, i) · β(t, i)/ likelihood)
 7               ξ(i) ← ξ(i) + (α((t, i) · β(t, i)/ likelihood)
 8               for j = 1 to N
 9                    do p ← α_ji · b_it
10                       β(t − 1, i − 1) ← β(t − 1, i − 1) + β(t, i) · p
11                       ξ(j, i) ← ξ(j, i) + (α(t − 1, j)β(t, i) · p/ likelihood)
12
```

Figure 4: The pseudocode of forward-backward algorithm.

With ξs and γs computed above, we can update A, B with following equation.

$$\hat{a}_{ij} = \frac{\xi(i, j)}{\xi(i)}$$
$$\hat{b}_{jt} = \frac{\gamma(j, t)}{\xi(j)}$$

As in forward algorithm, the **for** loop at line 5 of the backward algorithm can be parallized.

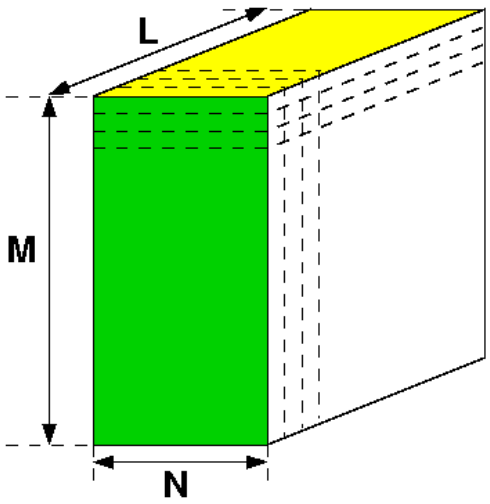

Figure 5: The 3D trellis as a cuboid.

Note accumulation of γ and ξ terms (line 6, 7 and line 10, 11) cannot be parallized in the same manner. If we have all γ and ξ terms, we can calculate their summation using the reduction technique [Har08]. On the other hand, there are $N \times N$ of ξ terms and $N \times M$ of γ terms. Therefore we can also carry out the accumulation in parallel for each term. If the number of ξ or γ terms is much larger than the number of sequences, this alternative approach will have little effect on the overall running time. The ideal way may be to implement the parallel reduction in two dimensional space, i.e. parallel reduction for each term in parallel. However, this will add much complexity to the implementation, and in our prototype program, we only take the term-level parallel approach.

3 Implementation

This section introduces our implementation details, including data structure, kernel functions and current limitations. At the end, we also discuss how to loose some restrictions on input data introduced by the parallel design.

Suppose we have M sequences of observations produced from the same HMM of N states. Further assume each sequence is of the same length L. To compute the forward probabilities in parallel for all sequences, we form a three dimensional trellis as a cuboid shown in Figure 5. Each slice in the cuboid parallel to the top (yellow) face is a trellis for one of the M sequences. Each slice parallel to the front (green) face corresponds to one column of trellises for all M sequences. Each element in the slice corresponds to a state in the trellis for some sequence of observations.

In the serial implementation, the cuboid is computed from top to bottom. Slices parallel to the yellow face is computed in sequence. For each slice, the computation goes from front column to back column in sequence. In our parallel implementation, we compute the cuboid trellis from front to back. At each individual step, we compute an entire slice parallel to the green face. For each element of the slice, the task is the same. In our CUDA implementation, each kernel thread is responsible for one state in the slice.

The computation of one slice of the cuboid resembles matrix multiplication as shown in Figure 6. The $M \times N$ matrix C is the previous slice in the cuboid; the $N \times N$ square matrix A is the state

transition probabilities; the $N \times |V|$ matrix B is observation emission probability matrix; and the matrix D is the result. For an element D_{ij}, i.e. the jth element on the ith row in matrix D, is computed as

$$D_{ij} = B_{O_i} \ .* \ C_i \ \times \ A_j$$

where O_i is the observation for ith sequence in current slice and $.*$ denotes element-by-element multiplication. The $C \times A$ part is the same as matrix multiplication. For a row in the C matrix, the output is fixed. We only need to load the corresponding row in the emission probability matrix and store the element-by-element product of the two rows and use the resulting product row to multiply the corresponding row in A.

In the CUDA programming model, threads are known as kernels running on parallel devices. The kernel threads are organized into grids of blocks. For the partition of kernels into blocks and grids, our prototype program uses the same schema as in matrix multiplication example the NVIDIA CUDA Programming Guide [NVI08].

- Each thread block is responsible for computing one square sub-matrix D_{sub} of D.
- Each thread within the block is responsible for computing one element of D_{sub}.

The block size here is chosen to be 16. The number of threads per block is a multiple of warp size and remains below the maximum number of threads per block [NVI08].

The Viterbi algorithm can use the same design. In each kernel thread, instead of computing the summation of the probabilities, we find the maximum one and the backtrace pointer is stored.

In Baum-Welch algorithm, the computation of one step backward pass follows the same pattern of forward algorithm. As the only change is index, we use the same kernel/block/grid organization as in forward algorithm. However due to the change in matrix index, the computation cannot be represented through matrix multiplication.

During each step of backward pass, we also need to accumulate the estimated γ and ξ counts. As analyzed before, the idea way is to carry out reduction for $N \times N$ states simultaneously. However this involves two different parallel patterns and makes programming difficult. In our current implementation, parallelism is carried out for each state. A thread will accumulate all M counts for the state it is responsible for. This design will lead to different computation load with the thread in forward algorithm. In the forward algorithm, a thread will accumulation N quantities. This may suggests the program will reach its computation peak when $M = N$.

The previous design raises the following two requirements on the input data.

1. The number of states and number of sequences must be a multiple of block size, i.e. 16.
2. The number of output sequences must be of same length.

To allow more general input, we can introduce slack variables into the model and pad the input to meet the two requirements. For both problems, we can add extra states and an output symbol, and pad the input with those states and outputs to make the size of input multiples of block size or make all input of same length with the longest one. After the computation, the results for those added states are simply ignored. This method will eliminate the two requirements but the computation time will be as long as the padded result. As long as the number of states and input sequences is much larger than block size and the lengths of all sequences are close, the extra computation time can be ignored.

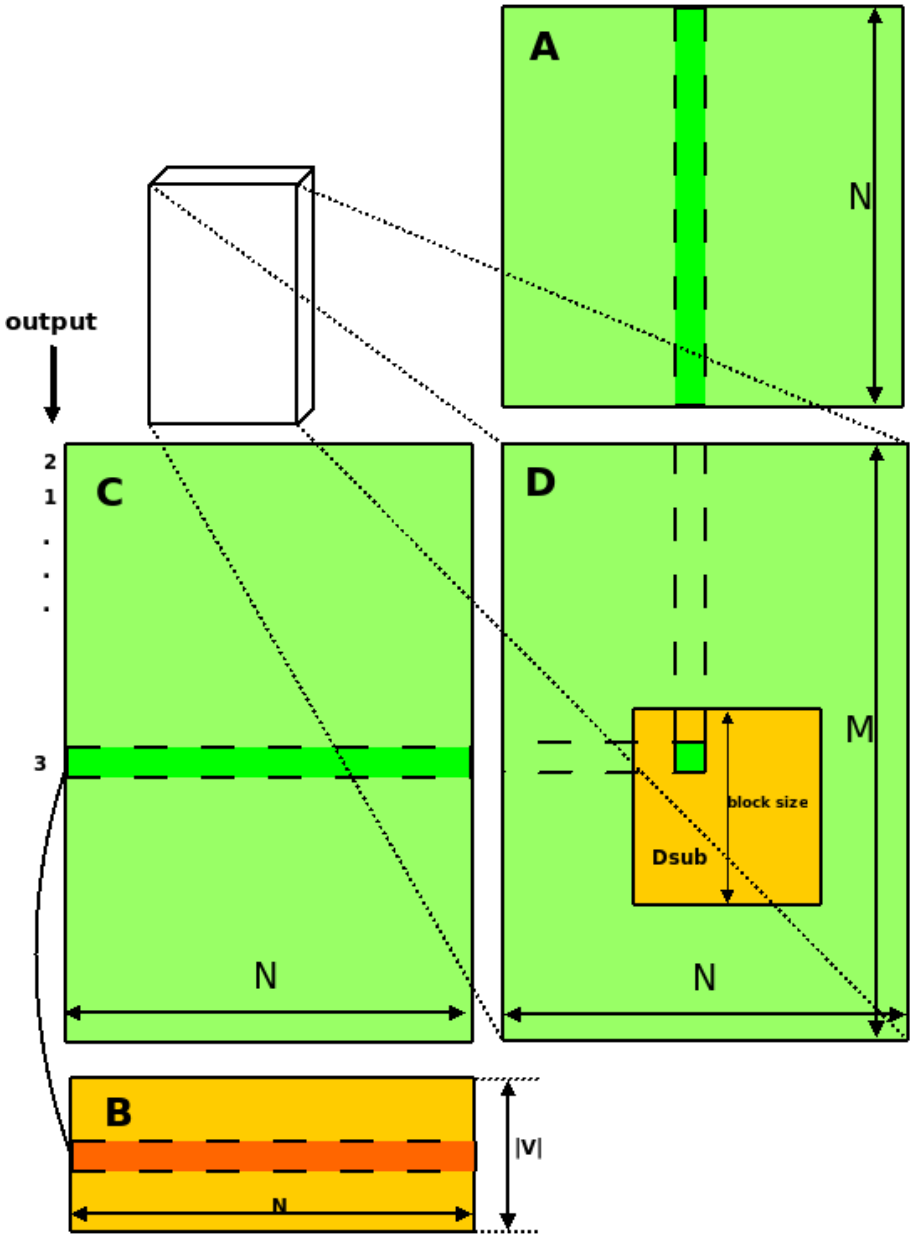

Figure 6: The computation of a slice in the trellis cuboid.

GPU	NVIDIA G92 with 512MB RAM
CPU	Intel Quad Core Q9500 with 8G RAM
OS	GNU/Linux 2.6.18-53 `x86_64`
CUDA	CUDA 2.0
Host Compiler	GCC 4.1.2

Table 1: The test environment.

4 Evaluation

In evaluation our implementation, we first measures the performance in terms of billion floating point operations per second (GFLOP/s). Though GFLOP/s is a good measure for hardware, we feel it may not be as well a metric for comparing our serial and parallel implementations. First, an single floating point operation used in the serial implementation may be decomposed into several separate operations in the parallel implementation. So the speedup in GFLOP/s may not reflect entirely in computation time. Second, besides GFLOP/s, other factors like memory bandwidth may also be the barrier for the program performance. Last the parallel program also uses CPU for some computation. Thus the measure of GFLOP/s in GPU may not reflect the entire program. As such, we also include the running time for comparison.

The serial implementation is written in C. Note in practice the HMM program usually can be optimized targeting certain known HMM structures. For example, speech recognition systems often use left-right models [Rab89]. Here for comparison, we make no assumption on the underlying HMM model and make no effort in optimization. Both serial and parallel implementations use single precision arithmetic. The correctness of both programs are verified against the MATLAB HMM Toolbox [1].

The kernels for forward and Viterbi algorithms are almost the same. So we only list the result for forward algorithm here.

Table 1 lists the information of our testing environment.

4.1 Forward Algorithm

We will evaluate how the performance of our CUDA implementation change as the number of sequences (M) and number of states (N) change, since they are the only factors that will effect the performance of the program. The running time of the program will only increase linearly as the sequence length (L) and number of observations ($|V|$) increase. With the same setting of N and M, GFLOP/s will remains the same. In all the evaluations, we use $L = 10$ and $|V| = 3$.

Figure 7 shows the performance change of the program with fixed number of state and increasing number of sequences. The number of states is set to be 16 which is minimalist possible. The peak performance is reached at about 17.6 GFLOP/s. After the peak value is reached, the running time increases almost linearly as the data size increases.

Figure 8 shows the performance change of the program with fixed number of sequences and increasing number of states. Again the number of states here is set to the minimal value 16. The peak performance is reached at about 23.5 GFLOP/s.

We also test the speedup of our CUDA program over serial implementation. The results are shown in Table 2. The effect of speedup is quite impressive. For example, in the last row the serial

[1] Accessible at: `http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html`

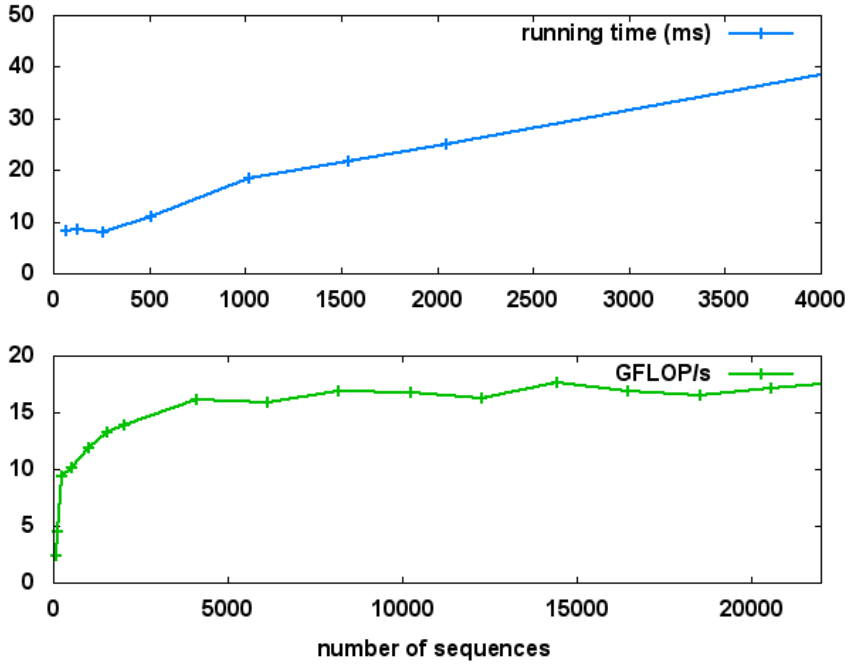

Figure 7: The performance change of forward algorithm with number of sequences increase.

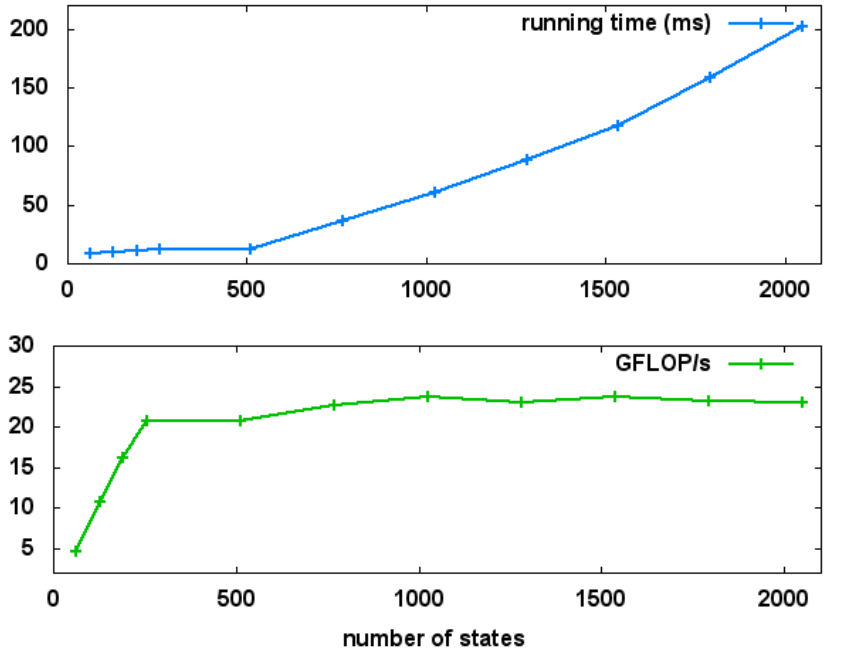

Figure 8: The performance change of forward algorithm as number of states increase.

Data size ($N \times M$)	Running time (ms)		Speedup
	CPU	GPU	
64×64	698.7	10.02	70×
128×128	5621.9	19.74	280×
192×192	18990.5	40.93	460×
256×256	45031.6	71.77	620×
320×320	88090.8	128.44	680×
448×448	152374.8	208.08	730×
512×512	360899.3	410.17	880×

Table 2: Comparison of running time between parallel and serial versions of forward algorithm.

Data size ($N \times M$)	Running time (ms)		Speedup
	CPU	GPU	
64×64	2138.4	35.9	60×
128×128	7891.1	142.6	50×
192×192	57681.3	339.1	170×
256×256	136694.2	903.3	150×
320×320	267038.8	1328.6	200×
448×448	461297.6	2479.6	180×
512×512	1094036.4	6054.8	180×

Table 3: Comparison of running time between parallel and serial versions of Baum-Welch algorithm.

implementation uses over 6 minutes while the running time of CUDA program is within 0.5 second.

4.2 Baum-Welch Algorithm

We evaluate the Baum-Welch algorithm with the same setting for forward algorithm. Figure 9 shows the change in performance of our implementation as the number of sequences increase. The peak performance is about 1.1 GFLOP/s. Figure 10 shows the performance change as the number of states increase. The peak performance is about 4.3 GFLOP/s. The speedup over serial implementation is shown in Table 3. The best speedup is 200×. Though the speedup are not as much as in forward algorithm, the results still make a different. For example, in the last row, the serial implementation use 18.2 minutes and the CUDA program uses 6 seconds.

5 Conclusion and Future Work

In this project, we have analyzed how to parallel the three fundamental algorithms of hidden Markov model for GPU computing environment. Based on the analysis we implement the three algorithms on NVIDIA CUDA platform and evaluate their performance. The evaluation shows the GPU implementation has up to 800× speedup for forward algorithm and 200× speedup for Baum-Welch algorithm.

Our implementation is not fully optimized and current peak GFLOP/s of our program does not reach GPU's throughput limit. We expect further speedup can be achieved in following ways.

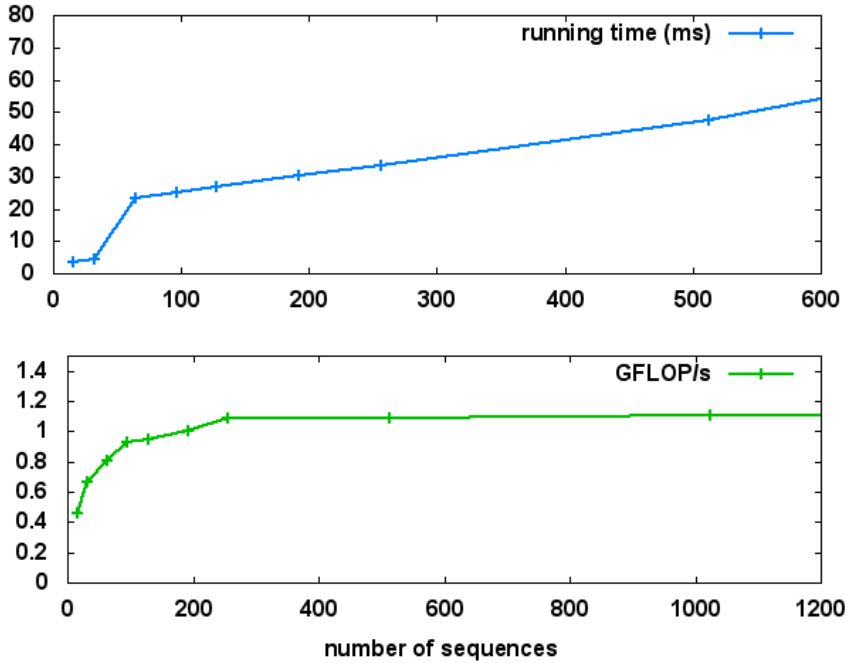

Figure 9: The performance change of Baum-Welch algorithm with number of sequences increase.

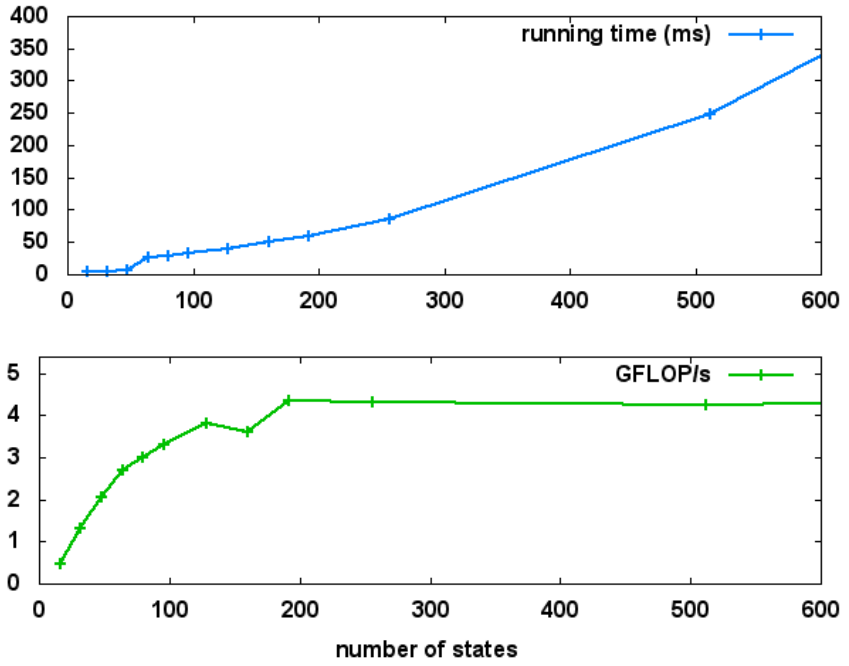

Figure 10: The performance change of Baum-Welch algorithm as number of states increase.

- Using texture memory. The transition matrix and observation emission matrix are accessed frequently throughout program and are not altered for forward and Viterbi algorithms or only altered at the end of each iteration for Baul-Welch algorithm. Currently we only store them using on-device global memory. We expect increasing in memory bandwidth throughput if texture memory is used.

- 2-Dimensional reduction. As explained in previous section, for Baulm-Welch algorithm, we expect further increasing in GPU throughput if the parallel reduction is implemented in 2D space.

- More efficient implementation of matrix multiplication. Our partition schema for matrix multiplication is not optimized. More performance may be achieved with better thread partition schema or by using CUBLAS library.

We also note both our serial and parallel implementation can only use single core. We expect speedup can also be made by taking advantage of multi-core CPUs or multiple GPUs.

We have made the code public available at http://code.google.com/p/chmm/.

References

[Har08] Mark Harris. Optimizing parallel reduction in cuda. NVIDIA CUDA SDK, 2008. Accessible at: `http://www.nvidia.com/object/cuda_sdks.html`.

[JM08] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2nd edition, May 2008.

[Kro98] Anders Krogh. An introduction to hidden markov models for biological sequences. In S. L. Salzberg, D. B. Searls, and S. Kasif, editors, *Computational Methods in Molecular Biology*, pages 45–63. Elsevier, 1998.

[NVI08] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2008. Version 2.1.

[Rab89] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of The IEEE*, 77(2):257–286, 1989.