A CUDA Implementation of Baum-Welch Algorithm for Hidden Markov Models

Elif Begüm Çığ
*Courant Institute of Mathematical Sciences**
New York University
New York, United States
ebc332@nyu.edu

Mohamed Zahran
Courant Institute of Mathematical Sciences
New York University
New York, United States
mzahran@cs.nyu.edu

***Abstract*—Hidden Markov Models is one of the most popular algorithmic approaches used in traffic monitoring, biological sequence analysis, speech recognition and wireless communications. Live traffic data, protein chains and text data can get extremely large with the complexity of the problem. A fast and robust method of extracting knowledge from the data for inference purposes is needed. Estimating the parameters of a Hidden Markov Model can be achieved by implementing the Baum-Welch algorithm. In this paper we implement a CUDA version of the Baum-Welch algorithm. The forward algorithm implemented for this paper can also be used to calculate the likelihood of a given sequence.**

In the paper, we discuss all the details of our implementation and the challenges faced and how we overcome them. We reached a 19.44x speedup with respect to the sequential version.

***Index Terms*—Forward-Backward Algorithm, Hidden Markov Model, Baum-Welch**

*

I. Introduction

A Markov Chain is a stochastic model describing a sequence of events where the probability of each event depends solely on the state attained in the previous time step of the sequence. For example a Markov chain for estimating today's weather would only use the weather data from the previous day. Hidden Markov Models (HMMs) in this context consists of two sequences X_n and Y_n where X_n is the event sequence of attention, however it is not directly observable (hence "hidden"). Instead, for each element X_i of the sequence, we view an indicator of this state called the "observation". Hidden Markov Models keeps the assumption that a given state is only dependent on the previous state, and adds, that the probability of an output "observation" only depends on the current state. To continue our weather example we can think of the hidden states as the weather forecast, and the observation sequence as the number of people we see on the street carrying an umbrella or not. The Markov Assumption states that for each day in our sequence, whether it will rain or not depends only on the forecast of the previous day and the number of people carrying an umbrella on a specific day depends only on the weather that day [4]. The probability of being in a state given the previous one is known as the transition probability and seeing an observation output given being in a state is called the emission probability as can be seen in Figure 1. Using these variables, we can construct a model to estimate the likelihood of a most probable state sequence or to infer the transition and emission probabilities given an observance sequence.

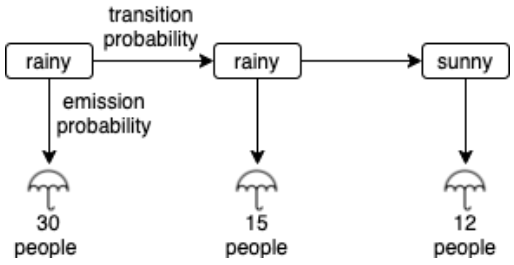

Fig. 1. The daily weather is the hidden state sequence, and the number of people carrying and an umbrella is the observance sequence. The observation on a day solely depends on the hidden state (emission probability) and the hidden state next day solely depends on the hidden state today (transition probability)

Adjusting the transition and emission probabilities for a given observation sequence for HMMs is known as the Baum-Welch Algorithm [1]. There is no known method to analytically solve for transition and emission probabilities. The Baum-Welch algorithm therefore starts with a model λ and adjusts its parameters until the probability of seeing the observation sequence is maximized given the model. The algorithm runs in iterations first to calculate the expected values of the transition and emission states then to generate new parameters using these expected values. These two steps are also known as the expectation step and the maximization step [15].

Hidden Markov Models are especially known for their application in reinforcement learning and temporal pattern recognition such as speech recognition [8] [9], hand writing analysis [13], classifying protein structures [7], face recognition [10], [11] and traffic event detection [14]. Nearly all of application areas of HMMs demand real-time results (e.g. speech, hand writing analysis and traffic event detection) and/or has a large amount of data that needs to be processed

*Graduated as of May 2020.

(protein sequence estimation, nearly all pattern detection applications). This added on top of the complexity of the algorithm itself, demands new hardware and optimization techniques to reduce this complexity.

The high degree of task and data level parallelism makes HMM inference an ideal task to be implemented on a GPU. In this paper we implement the Baum-Welch algorithm using NVIDA's CUDA on a Turing GPU. Our implementation accounts for the data transfer time from and to the GPU, execution time of the kernels and also considers the time required for reading the input data and the calculations done on the CPU. The correctness of the Algorithm is cross-checked with the MATLAB/Octave package mdtoolbox. We also provide a script to generate test data for given parameters of the number of hidden and observation states and the sequence length. The contributions of this paper include:

- Parallelization of discrete time HMM on a GPU
- Using texture memory, asynchronous data transfers and simultaneous kernel executions to demonstrate high utility.
- Utilizing linear algebra and turning arrays used in calculation into matrices to reduce the complexity of expectation and maximization steps.
- Reaching 19.44x speedup with respect to the C implementation on a single core CPU.

The rest of the paper is as follows: In Section II we give an overview on inference in hidden Markov models. In Section III we discuss the previous work done in this area. In Sections IV and V we give the details of our implementation. In section VI We share the details of our experimental setup. In section VII we discuss the experimentation results and we conclude our paper in section VIII.

II. Background

We have defined transition and emission probabilities in the previous section. In this chapter we will give background on the algorithms used in our implementation for readers that are unfamiliar to the subject. We will denote the hidden states as $1, 2, ..., N$, observation states as $V = V_1, V_2, ..., V_K$. If we took observations for T time steps then our observation sequence is $O = O^{(1)}, O^{(2)}, ..., O^{(T)}$, for which, every $O^{(t)}$ is a value between V_1 and V_K. With the same notation we can denote our hidden state sequence as $S = S^{(1)}, S^{(2)}, ..., S^{(T)}$, for which every $S^{(t)}$ is a value between 1 and N. Note that for ease of understanding we denote states with subscripts and times with superscripts i.e. we have N hidden states, K observation states and T time steps. We defined the transition matrix as the probability matrix for transitions from hidden state S_i to S_j. That is: $A = \{a_{ij} | a_{ij} = P(S_t = j | S_{t-1} = i)\}$. Then A is an NxN matrix. Emission matrix is the probability matrix for seeing an observation state V_k at a hidden state S_j. That is $B = \{b_{jk} | b_{jk} = P(O_t = V_k | S_t = j)\}$. Then B is an NxK matrix. Now we can formulate the necessary algorithms.

A. Forward Algorithm

Forward algorithm lets us find the likelihood of the hidden state sequence for a given observation sequence. Following from our weather example from the previous chapter, let's say that we are observing the number of people carrying an umbrella for three days and the results are 20, 160 and 30 for each day. Then the forward algorithm helps us to calculate the likelihood for possible weather forecasts, that is, we can calculate the likelihood of weather being rainy, rainy, sunny or sunny, sunny, sunny etc. It is computed by iteratively calculating the probabilities of the observations for all T. At time step t the probability of seeing O_t given state S_j is given as: $\alpha_t(j) = P(O_t | S_t = j) * P(all\ paths\ to\ state\ j\ at\ t)$. In other words, the forward algorithm at time step t computes the probability of seeing all of the observations until time step t, given that we are at an assumed hidden state S_t. Since it is an iterative process we need an initialization and a final step. Since at time $t = 1$ we do not have $P(all\ paths\ to\ state\ j\ at\ t)$ since we are at the beginning of the time states, we use a probability array π, also know as the prior probabilities. Then we can formulate the forward algorithm as follows:

1. Initialization:

$$\alpha_1(j) = \pi_j * b_j(o_1); 1 \leq j \leq N$$

2. Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) * a_{ij} * b_j(ot)$$

$$1 \leq j \leq N, 1 < t \leq T$$

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

B. Baum-Welch Algorithm

Since there is no way to solve for most optimal A, B matrices and π array, inference in hidden Markov models utilize expectation-maximization method. The idea behind is simple. Continuing from our weather example, let's say that we have a state sequence of 3 days and the weather forecasts for these days were: Sunny, sunny, rainy. We know that the transition array element a_{ij} gives us the probability of transitioning from hidden state S_i to S_j. If we check our hidden state sequence we can see that we have transitioned from sunny to rainy, and sunny to sunny 1 time. So out of all transitions from sunny, half of it was to a sunny day and the other half was to a rainy day. Therefore we can say $a_{sunny,rainy} = \frac{1}{2}$ and $a_{sunny,sunny} = \frac{1}{2}$. Notice that we have not transitioned from a rainy day, ever. This makes $a_{rainy,sunny} = 0$ and $a_{rainy,rainy} = 0$. Of course, the reader can notice that this is far from the truth. But the state sequence we have observed for our toy example only have 2 transitions. In a real life application we can apply this to

a extremely large state sequence. As we increase the length of our sequence T, these values will actually be close to their real life values, as one can also prove with central limit theorem. So we can formulate the optimal transition matrix $\hat{A}$ as:

$$\hat{a}_{ij} = \frac{expected\ number\ of\ transitions\ from\ state\ i\ to\ state\ j}{expected\ number\ of\ transitions\ from\ state\ i}$$

However, one problem we have is that the states are hidden in a hidden Markov model. Therefore we actually do not know whether the weather was sunny or rainy at any given day. So we introduce a new probability matrix ξ_t, that gives us probability of being at state S_i at time t and S_j at time $t+1$ for all i, j values. Then our formulation for $\hat{A}$ becomes:

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)}$$

In the same manner we can see that the emission probability for seeing an observation state V_k at hidden state S_j can be formulated as:

$$\hat{b}_j(V_k) = \frac{expected\ \#\ of\ times\ in\ state\ j\ and\ observing\ V_k}{expected\ number\ of\ times\ in\ state\ j}$$

But again, we do not know the hidden states so we introduce the probability matrix γ_t, that gives us the probability of being in hidden state S_j at time t. Then we can reformulate our assumption as:

$$\hat{B}_{jk} = \frac{\sum_{t=1 s.t. O_t=k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

The numerator is the sum of the gamma probabilities where we see the observation state V_k. The details of these algorithms will be further discussed in the following sections. However, this should be a sufficient overview for the reader that is unfamiliar to the subject.

III. Previous Work

Jun Li et al. look at the ability to perform the Forward-Backward Algorithm for a single observation sequence [3]. Using a CUDA implementation with 8 states, 8 observation states, and 200 observations, they reached a 3.5x speedup over serial implementation.

Liu, on the other hand, experimented with a single model and multiple observation sequences [2]. Using 512 states, 3 observations, and 512 sequences of 10 observations each, they achieved an 880x speedup of the Forward Algorithm and a 180x speedup of the Baum-Welch Algorithm. However their implementation only works with hidden states that are a multiple of 16, and the speed-up comparison they performed only takes the time spent on the GPU and not the whole running time of the program.

Yu et al. implemented Baum-Welch training for speech recognition and experimented with different optimization techniques [5]. With 4096 hidden states their implementation reached a 20.5x speedup for the forward algorithm and a 43.4x speedup for the complete training.

There has been also other studies to reduce the complexity of the algorithm in different ways. [19] utilizes the fact that in a long protein sequence some parts of the chain repeats itself to speed up the forward algorithm. [20] uses Spark to implement Baum-Welch algorithm for continuous Gaussian HMMs for big data applications.

The summary of previous work can be seen in tableI. The differences between measuring the runtime of the algorithm create great discrepancies between the speedup results. But the implementation we propose in the following sections use state-of-the-art methods like streams, utilization of texture memory and efficiency in the mathematical calculations which the first two papers don't use. It is also a generalized version of the Baum-Welch algorithm and can be used for any HMM application unlike the last paper.

IV. Optimization Techniques

Global memory accesses and data transfers adds a big overhead in GPU applications. Therefore, it is important to try to overlap computation and communication effectively so that the impact of the transfer due to PCIe latency and global memory accesses can be hidden and would not affect the performance too adversely. We give the details of our algorithmic choices in the implementation section. Here we discuss our choices at the application and the device level.

A. Task Level Parallelism

Baum-Welch is a computationally heavy algorithm and some parts of the algorithm needs the results from the previous time step. Therefore we need to come up with a method that ensures the correctness of the algorithm while utilizing our resources to execute maximum number of computations simultaneously. All kernel launches in a CUDA program by default run on stream 0. CUDA C Programming Guide [17] states that all kernel launches and memory set function calls are asynchronous. However, multiple kernel launches and memory copies larger than in a block of 64 KBs happen synchronously. Specifying a stream for a kernel launch and a host-device memory copy operation allows us to run these operations asynchronously as well. For implementing Baum-Welch algorithm one can use the default setting to calculate everything in order, that is; the forward algorithm first, followed by the backward algorithm. Then the expectation step and finally the maximization step. However, computations required for forward and backward algorithm, are completely independent from each other, therefore the necessary memory transfers and kernels can be run at the same time. Figure 2 shows the timeline differences of Forward and Backward algorithms running on a GPU with and without streams. Also γ_i and ξ_i computations needed for expectation step do not depend on each other. For this reason in our implementation we use streams for forward and backward algorithms, and for

TABLE I
SUMMARY OF PREVIOUS WORK

Paper	Pros	Cons
Jun Li et al.	3.5x speedup	Tested for only 8 hidden states and 200 observations
Liu	180x speedup Tested with large number of hidden states	Only time on GPU is accounted for. Transfers not included. FLOP/s peak far below GPU capacity
Yu et al.	43.4x speedup Tested with 4096 hidden states. Utilised streams and HyperX.	Speedup is calculated for the total of 15 iterations. Specific usage for speech recognition HMMs.

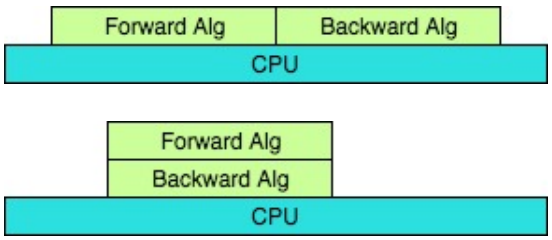

Fig. 2. Kernel launches comparison with the default stream and many streams

both expectation and maximization steps of the Baum-Welch algorithm.

B. Texture Memory

To increase data level parallelism, one must minimize the global memory accesses in their program. When computing the Baum-Welch algorithm, transition and emission matrices are both used in forward and backward algorithms and the expectation-maximization steps. However, updating these matrices happens at the very last step of the algorithm. Texture memory is a read-only device memory, much like the constant memory, and it is cached to be efficient for spatially-localized accesses. For these purposes we load the prior probabilities, the observance sequence and the emission and the transition matrices on texture memory to use the cache benefits.

Shared memory, although much faster than texture memory, cannot be used for loading transition and emission matrices. For example, every thread needs to use the complete row of the transition matrix corresponding to their hidden state which makes it impossible considering the memory limitations of the shared memory. Texture memory will work like the global memory in the worst case scenario of a cache miss.

V. IMPLEMENTATION

This section introduces our implementation details. Implementation of Baum-Welch algorithm can be divided into to parts: First is computing the likelihood of the model given the observation sequence. This is forward algorithm as stated in Chapter 2. The second part is the learning, that is, the prediction of the parameters of the model. This is found through expectation-maximization. Here we give the technicalities on the algorithmic complexities and the kernel details.

A. Forward Algorithm

In the initialization step of the forward algorithm we perform an element by element multiplication for each state s_i with their prior probabilities π_i. The sequential implementation of this step takes $O(N)$ time, however this can be implemented in a GPU at lockstep since all states are independent of each other, thus reducing the complexity to $O(1)$.

The sequential version for the recursion step of the forward algorithm requires a thrice embedded for loop. The outer loop runs for each time step t and two inner loop runs for N to calculate

$$\alpha_t = (A^T * \alpha_{t-1}.B(0_t)) \tag{1}$$

where dot is the notation for element-by-element multiplication. The complexity of sequential forward algorithm is $O(TN^2)$,where T is the sequence length and N is the number of hidden states. Since each for time step t, α_t is dependent on the result coming from α_{t-1}, we cannot parallelize the outermost loop. Therefore we implement the CUDA kernel of the forward algorithm to calculate each time step t, and we simply call this kernel T times in a loop. In the sequential version of the algorithm α is an NxT matrix, however since we are using the i^th column at every time step t_i we use a transposed α to ensure memory coalescing.

At each time step, our CUDA implementation calculates N elements of the forward algorithm for each hidden state s_i, therefore we could have picked a grid size of N, this way each thread would have calculated each element s_i of the time step t. However at each time step t, we still need to use every element of the transition matrix A, which is an NxN matrix. Therefore we picked a grid size of NxN. This way we use the threads in the y coordinate just to load $TILE\_SIZE\ x\ TILE\_SIZE$ parts of the transition matrix to shared memory. This causes N-1 * N threads to do exact same computations as the original N threads we need the answers from, however considering all of these calculations are happening concurrently, this only causes us to lose memory over gaining performance. Since we are calculating each state s_i concurrently, our complexity is reduced to $O(NT)$.

Backward algorithm is implemented in the same manner as the forward algorithm with NxN grids and tiled matrix multiplication. We will not go into details in this paper as there these methods are already discussed for forward algorithm.

B. Expectation-Maximization

The expectation step of Baum-Welch algorithm consists of calculating two variables γ and ξ. To update the values of transition and matrix first we need to find the expected values of each transition from state i to state j, then we can divide this number to total number of transitions from state i to get our new value for a_{ij}. Since the states are hidden, we do not know the number of times each state has appeared in the sequence. Instead we calculate state transition probability matrix ξ. Same way, updating the emission matrix requires the number of times we observed an observation state i at hidden state j. We also cannot possibly calculate this directly, so we calculate the transition probability matrix γ.

$$\xi_t(i,j) = \frac{\alpha_t(i)A_{ij}\beta_{t+1}(j)B_j(o_{t+1})}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)} \tag{2}$$

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)} \tag{3}$$

The sequential algorithm calculating the ξ values runs in a thrice embedded loop and its complexity is $O(N^2T)$. Here we propose a matrix multiplication approach to reduce the complexity. To implement this approach first we need to turn the observation sequence into an observation matrix. The observation sequence O is of length T and and each O_i takes values $1...K$ where K is the number of observation states V. We reconstruct the observation sequence as a KxT matrix where for $O_t = V_k$, $Obs\_mat(V_k, t) = 1$ and for other observation state values V'_k, $Obs\_mat(V'_k, t) = 0$. ξ calculation requires the element $b_j(o_{t+1})$, which is the value of the emission array at hidden state j for the observation state k at time step $t+1$. Instead of computing this value each time, we can multiply the NxK emission matrix with our newly constructed KxT observation matrix. Let's call this newly created NxT matrix B'. $B'(j,t)$ will have value of the emission array at hidden state j for the observation state k at time step t. The complexity for calculating the observation matrix is $O(1)$ and the complexity for calculating B' is $O(K)$.

Now that we have B', the numerator computation becomes

$$\xi_t(i,j) = \alpha_t(i)A_{ij}\beta_{t+1}(j)B'_{jt} \tag{4}$$

We can also formulate a variation on this as a matrix multiplication as follows:

$$\xi' = (\alpha * (\beta^T.B)).A \tag{5}$$

For memory coalescing purposes our α and β matrices are in transpose form so the computation becomes:

$$\xi' = (\alpha^T * (\beta.B)).A \tag{6}$$

Where dot is the element-by-element multiplication. Note that ξ the original paper propose is an $NxNxT$ matrix. The ξ' we implement is an NxN variant of this where each element is a summation over all time step values. We implement the kernel for calculating ξ' as a tiled matrix multiplication. Since matrix multiplication will loop over every row of α and every column of β our complexity is $O(T)$. Therefore our total complexity for the numerator of ξ becomes $O(T) + O(K) + O(1)$. Considering T will most likely be larger than K, this makes our complexity $O(T)$.

The sequential algorithm calculating the γ values runs in a double loop and is of complexity $O(NT)$. This can be implemented in a GPU in a $O(1)$ algorithm since none of the gamma values depend on each other. However for the maximization step we need to sum the $\gamma(j)$ values from time step 1 to T where $O_t = k$; that is we need the gamma values with respect to the observation states. Therefore instead of just computing the γ values we used the observation matrix we created for ξ calculation and computed a gamma-observance matrix as following:

$$\gamma_k = (\alpha.\beta) * Obs\_mat \tag{7}$$

Note that the γ_k matrix is not NxT like the original γ matrix but instead is NxK. This is because the original γ matrix is simply the every gamma value for time step t. The γ_k variant we computed however, is a sum of all γ values for every observance state V_k. The complexity of computing γ_k on the GPU is $O(T)$.

The maximization step is where we calculate the updates for our A and B matrices. The prior probabilities also gets updated during this phase; however, it is a trivial computation and doesn't contribute to the overall complexity of the implementation so it will not be discussed. The new A and B matrices are calculated as following:

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N}\xi_t(i,k)} \tag{8}$$

$$\hat{B}_{jk} = \frac{\sum_{t=1 s.t. O_t=k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)} \tag{9}$$

Notice that the numerator of both of these computations are actually the γ_k and xi' matrices we calculated in the expectation step. The denominators are just row-sum operations which can be implemented in a recursive fashion in CUDA effectively. ξ and γ sum calculations are independent from each other, and for each hidden state row-sums of ξ and γ in themselves are independent, as well. Therefore for each hidden state we launch kernels in different streams to compute these sums concurrently. The reduce sum has a complexity of $O(logN)$. This makes the total complexity of EM steps $O(T + logN)$.

TABLE II
NODE SPECIFICATIONS

Component	Specification
GPU	GeForce RTX 2080 Ti
CPU	Intel Xeon CPU E5-2660 v3 @ 2.60GHz
OS	CentOS Linux 7
Linux Kernel Version	3.10.0
CUDA Version	10.1.243
gcc Version	4.8.5
nvcc Version	10.1

VI. SETUP

The specifications of the computer on which the tests are run is given on Table II. Table III provides the specifications for the GeForce RTX 2080 Ti [12].

TABLE III
GEFORCE RTX 2080 TI SPECIFICATIONS

Component	Specification
CUDA Capability	7.5
Clock Rate	1.63 GHz
Memory Bus Width	352 bit
L2 Cache Size	5.7 GB
CUDA Cores	4352
Global Memory	11 GB
Streaming Multiprocessors (SMs)	68

VII. EXPERIMENTAL RESULTS AND ANALYSIS

We have tested the running time of the sequential implementation and the GPU implementation for the complete program by using the Linux command $time$ The number of observation states doesn't change the complexity of the algorithm since the complexity grows only with the length of the observance sequence and hidden states so for all tests executed we used 3 observation states.

First we kept the observation sequence length at 320 and used 64, 128, 256 , 512, 1024 and 2048 hidden state variables. Execution time and speedup can be seen in 3 and 4. Our GPU algorithm has reached 19,44x speedup with respect to the sequential CPU implementation when there are 1024 hidden states. For 64 states GPU implementation memory allocation on the host takes around 900 ms whereas all kernel operations account for around 10-11 ms. That means memory allocation takes 97% of all execution time. The speedup gained from the GPU cannot makeup for the time lost for the memory allocation operations. Therefore for 64 states, the CPU implementation is faster than the GPU implementation. Also notice that after 1024 state our implementation reaches a saturation point and loses speedup. When we have 2048 hidden states, the data we have does not fit into GPU completely therefore not all of the steps execute concurrently. Then keeping the number of hidden states at 64, we used sequence length of 160, 320, 640 and 1280. The results can

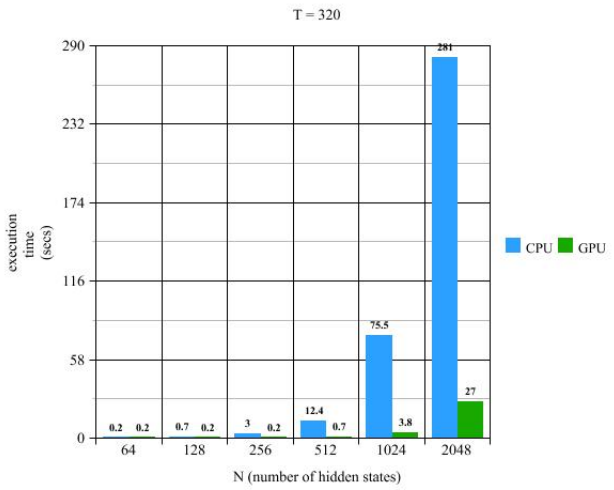

Fig. 3. Running times with changing number of hidden states

be seen in 5. Our GPU implementation reached a 4x speedup with respect to the sequential CPU implementation. Note that varying the sequence length increases both the CPU and GPU running time almost linearly, since both the GPU and CPU implementations grow linearly with the sequence length. However the CPU implementation is $O(N^2)$ for N hidden states while the GPU implementation is N, therefore the speedup in Baum-Welch algorithm comes from increasing the number of hidden states.

We profiled our GPU implementation using the *nvprof* [16] profiling tool. You can see the results for $N = 1024$ in 6 and 7. You can see that when the number of hidden states are large and since we kept the data transfer to a minimum by keeping the results from forward and backward algorithms on the GPU, the memory transfers only takes up $\%0.01$ of the total GPU execution time. The overall time is dominated by the forward and backward algorithms since, at each time step their computation depends on the results coming from the previous time step. Since we used a novel tiled matrix multiplication for computing the ξ and γ values and used reduce operations to sum each row, Expectation-Maximization takes up less than $\%0.35$ of total execution time.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we have shared our GPU-accelerated implementation of Baum-Welch algorithm for HMMs. The takeaways from this project can be summed to:

- The GPU implementation of the Baum-Welch algorithm executes the necessary computations for the hidden states at lockstep, therefore it manages to surpass the memory allocation cost on the host and really shine when the number of hidden states are large.
- The bottleneck of the program is the forward and backward algorithms since each time step depends on the results from the previous time step.

In our implementation we proposed a matrix-multiplication method for calculating the ξ and γ variables. More generally,

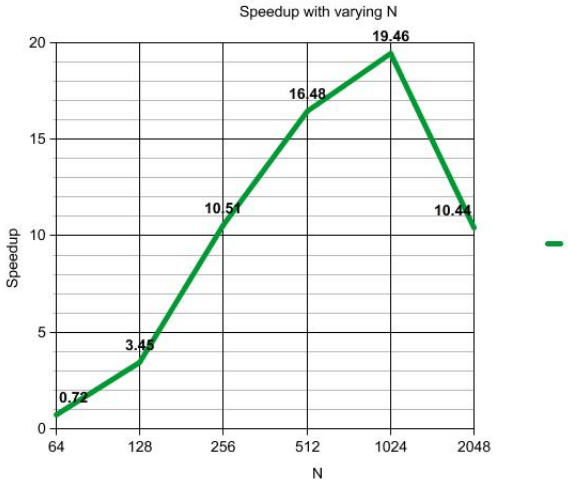

Fig. 4. Speedup with changing number of hidden states

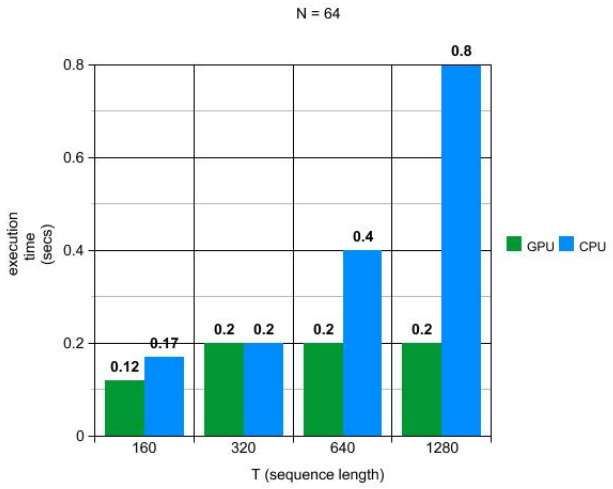

Fig. 5. Running times with changing number sequence length

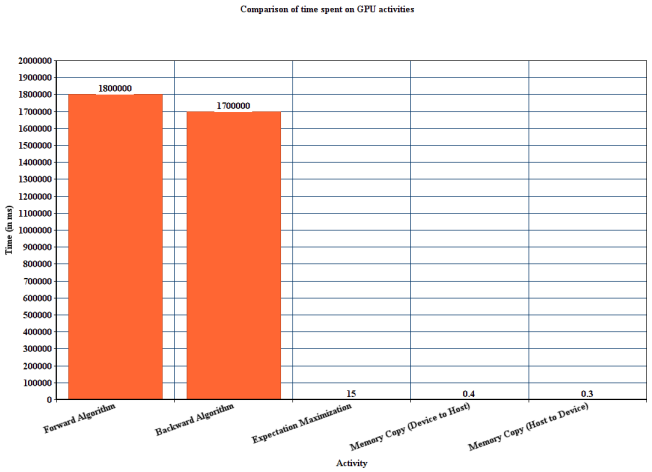

Fig. 6. Running times of the algorithms. Forward and backward algorithms take around 98% of the total GPU time.

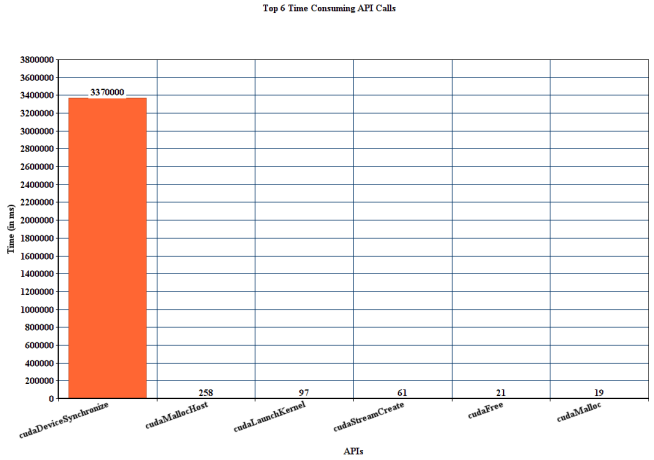

Fig. 7. Time taken for top 6 API calls. CudaDeviceSynchronize takes up 88% of total API calls

this paper shows there is potential for inference applications to be implemented on GPUs. It's important to note that the GPU implementation of the Baum-Welch algorithm reduce the complexity of the CPU algorithm with respect to hidden states, and we got the best speedup when the hidden states are large. Therefore while rewriting any algorithm to run on a GPU it's crucial to determine which part of the implementation is reducing the theoretical complexity of the overall algorithm, and plan the hardware and hardware supported software specifications (like memory allocation, streaming) for this part to maximize the speedup.

The emission matrix in our implementation can be padded for memory coalescing purposes in the future. Also, we only used a transposed version of α and β matrices. However, texture memory accesses are also optimized when the accesses by threads are spatially close. Therefore this implementation can further be improved by using a transposed version of A and B matrices.

REFERENCES

[1] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," Proceedings of the IEEE, vol. 77, no. 2, 1989.

[2] C. Liu, "cuHMM: A CUDA implementation of hidden Markov model training and classification," Available online, http://liuchuan.org/pub/cuHMM.pdf, 2006.

[3] J. Li, S. Chen, and Y. Li, "The fast evaluation of hidden Markov models on GPU," International Conference on Intelligent Computing and Intelligent Systems, IEEE, 2009.

[4] A. Leon-Garcia, "Probability, statistics, and random processes for electrical engineering," Upper Saddle River, NJ: Prentice Hall, 2008.

[5] L. Yu, Y. Ukidave and D. Kaeli, "GPU-accelerated HMM for speech recognition," 43rd International Conference on Parallel Processing Workshops of IEEE, pages 395–402, 2014.

[6] D. Jurafsky and J. H. Martin, "Speech and Language Processing," Prentice Hall, 2nd edition, May 2008.

[7] G. Mirceva and D. Davcev, "HMM based approach for classifying protein structures," International Journal of Bio- Science and Bio-Technology, vol. 1, no. 1, pp. 37-46, Dec. 2009.

[8] G. A. Fink, "Markov models for pattern recognition," p127, 2008.

[9] M. Nilsson and M. Ejnarsson, “Speech recognition using hidden Markov model,” Department of Telecommunications and Speech Processing, Blekinge Institute of Technology, 2002.
[10] A. V. Nefian and M. H. Hayes, “Face detection and recognition using hidden Markov models,” Proceedings 1998 International Conference on Image Processing, Chicago, IL, USA, pp. 141-145 vol.1, 1998.
[11] H. Le and H. Li, “Face identification system using single hidden Markov model and single sample image per person,” 2004 IEEE International Joint Conference on Neural Networks, Budapest, pp. 455-459, 2004.
[12] GeForce RTX 2080 TI specifications. Retrieved from: https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/
[13] J. Hu, M. K. Brown, and W. Turin, “HMM based online handwriting recognition,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 18, no. 10, pp. 1039-1045, Oct. 1996.
[14] X. Li and F. M. Porikli, “A hidden Markov model framework for traffic event detection using video features,” 2004 International Conference on Image Processing, pp. 2901-2904 Vol. 5, 2004.
[15] A. P. Dempster, N. M. Laird, and D.B. Rubin, “Maximum likelihood from incomplete data via the EM algorithm,” J. Roy. Stat. Soc., vol. 39, no. 1, pp. 1-38, 1977.
[16] NVIDIA, “Profiler, Compute Visual,” May 2011.
[17] CUDA C Programming Guide, Retrieved from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[18] S. R. Hymel, “Massively parallel hidden Markov models for wireless applications,” Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2011.
[19] A. Sand, M. Kristiansen, C.N. Pedersen, T. Mailund, “zipHMMlib: a highly optimised HMM library exploiting repetitions in the input to speed up the forward algorithm,” BMC Bioinformatics 14, 339, 2013.
[20] I. Sassi, S. Anter and A. Bekkhoucha, “A new improved Baum-Welch algorithm for unsupervised learning for continuous-time HMM using Spark,” International Journal of Intelligent Engineering and Systems, pp. 214-226, 2020.