PARALLEL IMPLEMENTATION OF HIDDEN MARKOV MODELS FOR WIRELESS APPLICATIONS

Shawn Hymel (Wireless@Virginia Tech, Virginia Tech, Blacksburg, VA, USA; hymelsr@vt.edu); Ihsan Akbar (Harris Corporation, Lynchburg, VA, USA); Jeffrey H. Reed (Wireless@Virginia Tech, Virginia Tech, Blacksburg, VA, USA)

ABSTRACT

Hidden Markov Models (HMMs) provide the means to model sequential data that go through a series of states over space or time. HMMs are widely used in speech recognition algorithms and have seen application in wireless communications, including channel modeling, specific emitter identification, and signal detection and classification. Unfortunately, the use of HMMs in cognitive radio is hindered by their computational complexity. This paper proposes an extremely fast accelerator using graphics processing units (GPUs) that allows for model training and pattern recognition on the fly. Specifically, the Baum-Welch, Forward, and the Viterbi algorithms are written to take advantage of the GPU's ability to handle single instruction, multiple data (SIMD) parallelization.

This paper shows that the speed benefits from parallelization are maximized when a large number of HMM states are used. While general purpose computing on graphics processing units (GPGPU) is a fairly recent field, the advent of low-power, small profile graphics accelerators for handheld devices opens new doors for parallel processing in the realm of software defined radios. Additionally, several applications for HMMs in software defined radios are discussed as potential beneficiaries of the proposed accelerator.

1. INTRODUCTION

HMMs have been widely utilized (in research) as a means of pattern recognition in specific emitter identification and signal identification [1][2] but often compete against other well known algorithms, such as neural networks [3].

One of the major issues surrounding the use of HMMs in cognitive radio is its computational complexity. The evaluation problem, which is discussed in detail later, requires a $O(TN^2)$ algorithm, where T denotes the number of observations and N is the number of states in the HMM. For an especially large number of observations and/or large number of states, this computational cost can be quite prohibitive on many systems, including powerful desktops and is especially costly on power-thrifty digital signal processors (DSPs). Fortunately, the advent of handheld devices, such as smart phones and tablets, has ushered in a new era of graphics processing hardware.

Graphics processing units (GPUs) were conceived to handle computer graphics by focusing on massively parallelizing operations using large data sets with a single operation - known as single instruction, multiple-data (SIMD) operations. Ideally, the computational complexity can be reduced to a more manageable limit and allow for near real-time results for methods such as spectrum sensing and identification when utilizing HMMs.

The structure of the paper is as follows: previous work is examined in section 2; a brief overview of HMMs and subsequent algorithms are given in section 3, along with the parallel implementation analysis; section 4 looks at problems encountered with the GPU architecture; section 5 describes how the performance of the algorithms were evaluated; section 6 gives the results of the evaluation; and finally, several conclusions and potential future work are outlined in section 7.

2. RELATED WORK

Due to the complexity of many of the HMM algorithms, there has been ample research on the topic of parallel implementations. GPUs are a natural fit for parallelization work and have been the subject of many studies.

Jun Li et al. examines how the Forward-Backward algorithm can be used to evaluate the fit of several HMMs to a single observation sequence [4]. In their examination, Li et al. create a Compute Unified Device Architecture (CUDA) implementation of the Forward-Backward algorithm as well as analyze several models in parallel using the GPU. As a result, they find that a speed increase of 3.5x is gained from C to CUDA when using 60 HMMs with 8 states, 8 symbols, and 200 observations each.

Chuan Liu, however, assesses the effects of evaluating multiple observation sequences and a single model in parallel using CUDA [5]. With 512 states and 512

sequences of 10 observations each, the author achieved an 880x speedup from C to CUDA for the Forward Only algorithm and a 180x speedup for Baum-Welch Algorithm (BWA).

Additionally, Zhang et al. implemented the Viterbi algorithm in CUDA and observed an average of 3x speedup [6]. Their implementation focused on speech recognition and utilized 2000-3000 words (states).

In the field of communications, many studies have been published proposing new technologies to increase the speed of algorithms, reduce power consumption, and provide greater throughput. One such study [7] evaluates a new SIMD architecture known as Ardbeg and finds that the parallel operation allows for a 1.5-7x speedup over its predecessor. These findings promise speed improvements in the areas of filtering, modulation, synchronization, and error correction. However, Signal-processing On-Demand Architecture (SODA) and Ardbeg are specifically designed for Software Defined Radio (SDR) applications. GPUs, on the other hand, require more power, but are now ubiquitous in smart phones, tablets, and small computers.

While GPUs may be less power efficient for communication applications than their DSP counterparts, they have the potential to become powerful coprocessors in SDR and cognitive radio. Othman and Aboulnasr examine 2D HMMs and their use for facial recognition [8]. Similar techniques could prove useful for pattern matching in signals analysis. 2D and 3D HMMs are even more computationally complex than their 1D equivalents and would benefit from SIMD-type architectures.

3. ANALYSIS

A Hidden Markov Model is a statistical model that assigns output probabilities based on a series of unobservable states. The modeled system is assumed to be a Markov process, namely one that exhibits the Markov property:

$$\Pr[X_n = x_n \mid X_{n-1} = x_{n-1} \ldots X_0 = x_0] = \Pr[X_n = x_n \mid X_{n-1} = x_{n-1}] \quad (1)$$

This paper uses the HMM notation put forth by Rabiner [10]. As such, the HMM can be described by the triplet:

$$\lambda = (A, B, \Pi) \quad (2)$$

Where A is an $N \times N$ matrix describing the probabilities of transitioning between N states, B is an $M \times N$ matrix that gives the probabilities of a given state producing 1 of M output symbols, and Π is a $1 \times N$ matrix which describes the probabilities that the model will initialize one of N states.

The three canonical problems involving HMMs are as follows:

Problem 1: Given an observation sequence $O = O_1 O_2 \ldots O_T$ and a model λ, find the probability that the model will generate O: $Pr[O|\lambda]$.

Problem 2: Given an observation sequence O and a model λ, find the most likely state sequence $Q = q_1 q_2 \ldots q_T$.

Problem 3: Given an observation sequence O, find a model λ that maximizes $Pr[O|\lambda]$.

Problem 1 is known as the "evaluation problem" and is used in pattern recognition to determine which model best fits an observed sequence. Both the Forward and Backward algorithms are capable of solving the first problem.

Problem 2 attempts to reveal the "hidden" portion of the HMM, in which the exact sequence of states are discovered. Using the Viterbi algorithm [11], we can estimate the most likely sequence of states based on the HMM properties. This algorithm sees use in convolutional decoders as well as areas such as speech recognition.

Problem 3 shows that given a starting model λ and a training sequence O, we can re-estimate the parameters of the HMM to produce λ'. HMM training can be accomplished via the Baum-Welch Algorithm [12], the Viterbi algorithm, or the Segmental K-Means Algorithm [13]. Unfortunately, the re-estimation process is intractable, as only a local maximum for $Pr[O|\lambda]$ is solved and finding the global maximum is nearly impossible.

3.1. Parallelizing the Forward Algorithm

The Forward Algorithm [10] is given as follows:

1) Initialization:

$$\alpha_1(i) = \pi_i b_i(O_1), \; i = 1, 2, \ldots N \quad (3)$$

2) Induction:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1}), \quad 1 \le t \le T-1, \; 1 \le j \le N \quad (4)$$

3) Termination:

$$\Pr[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i) \quad (5)$$

Looking closely at the serial implementation of the Forward Algorithm, we notice that the initialization step is $O(N)$, as each element in the Π matrix is multiplied by the corresponding element in the B matrix, it is indexed by the observation at time $t = 1$. The induction step relies on a $1 \times$

N by $N \times N$ matrix multiplication at each time instant t, resulting in a complexity of $O(TN^2)$. The termination step requires $O(N)$ operations to sum the final α values at time $t = T$. As a result, the Forward Algorithm is $O(TN^2)$.

Without knowledge of the HMM structure, we cannot parallelize the recursive portion of the Forward Algorithm. However, we can parallelize operations across the number of states (N). Operations that utilize different elements of the array can be parallelized into an $O(1)$ operation, and functions that sum, multiply, etc. over an array require $O(\log N)$ operations, as opposed to $O(N)$, using a technique known as "reduction". The initialization step can be reduced to an $O(1)$ operation, as different array elements are used. For the induction step, matrix multiplication ultimately requires a multiply and sum for each element, resulting in $O(\log N)$ operations for each t. Therefore, the induction step can be parallelized to produce an $O(T \log N)$ function. Finally, the termination step is a summation, requiring $O(\log N)$ operations. Given these, the total number of operations in the Forward Algorithm can be reduced to $O(T \log N)$.

3.2. Parallelizing the Viterbi Algorithm

The Viterbi Algorithm finds the most likely state sequence given a model and an observation sequence. This is accomplished by:

1) Initialization:

$$\delta_1(i) = \pi_i b_i(O_1), \ i = 1,2,...N \tag{6}$$

2) Recursion:

$$\delta_{t+1}(i) = \max_j \left[\delta_t(j) a_{ij} \right] b_j(O_{t+1}), \quad 1 \le i \le N, \ 1 \le t \le T-1 \tag{7}$$

$$\psi_t(i) = \arg\max_j \left[\delta_t(j) a_{ij} \right], \quad 1 \le i \le N, \ 1 \le t \le T-1 \tag{8}$$

3) Termination:

$$q_T^* = \arg\max_i \left(\delta_T(i) \right) \tag{9}$$

4) Path backtracking:

$$q_t^* = \psi_{t+1}\left(q_{t+1}^* \right), \ t = T-1, T-2, ...1 \tag{10}$$

Again, we examine the serial and parallel implementations of the algorithm. The initialization step is $O(N)$ operations. The recursion step is $O(TN_2)$ operations, as we must first compute $\delta_{t-1}(i)a_{ij}$, find its maximum value, multiply that answer by $b_j(O_t)$, and then iterate that function over all t. The termination is simply $O(N)$ to find the maximum value, and the path backtracking is $O(T)$.

Because the Viterbi Algorithm is recursive, much like the Forward Algorithm, we cannot parallelize across T. However, we are still able to compute the data sets in parallel across N, which gives us $O(1)$ for the initialization, $O(T \log N)$ for the recursive step, $O(\log N)$ for the termination, and $O(T)$ for the path backtracking. A cumulative complexity of $O(T \log N)$ for entire algorithm written in parallel is achieved.

3.3. Parallelizing the Baum-Welch Algorithm

The BWA can be used to re-estimate the parameters in a HMM given an observation sequence. This is accomplished by the following method [14]:

Step 0: Start with an initial model λ

Step 1: Compute the Forward variables (α)

Initialization:

$$\alpha_1(i) = \pi_i b_i(O_1), \ i = 1,2,...N \tag{11}$$

Induction:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1}), \quad 1 \le t \le T-1, \ 1 \le j \le N \tag{12}$$

Termination:

$$\Pr[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i) \tag{13}$$

Step 2: Compute the Backward variables (β)

Initialization:

$$\beta_T(i) = 1, \ i = 1,2,...N \tag{14}$$

Induction:

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) b_j(O_{t+1}) a_{ij}, \quad 1 \le t \le T-1, \ 1 \le j \le N \tag{15}$$

Step 3: Compute γ

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Pr[O \mid \lambda]}, \ i = 1,2,...N \tag{16}$$

Step 4: Compute ξ

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\Pr[O \mid \lambda]} \tag{17}$$

Step 5: Re-estimate the HMM parameters

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{18}$$

$$\hat{b}_j(e_k) = \frac{\sum_{t=1 | O_t = e_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{19}$$

$$\hat{\pi}_i = \alpha_1(i)\beta_1(i) \tag{20}$$

Step 6: Go back to step 1 using the re-estimated parameters as the initial model and repeat for a set number of iterations or until a desired level of convergence is reached.

Since the Forward variable calculation is the same as Forward Algorithm, the computational complexity is $O(TN^2)$. Similarly, the Backward variables are $O(TN^2)$. The γ variables require $O(TN)$ operations, and the ξ variables require $O(TN^2)$ operations. Re-estimating the A matrix requires $O(N^2)$ complexity; re-estimating B requires $O(TMN)$, where M is the number of possible output symbols; and re-estimating Π requires $O(N)$ operations. The entire algorithm's complexity can be expressed as

$$\begin{cases} O(TMN), & M > N \\ O(TN^2), & else \end{cases} \tag{21}$$

Much like the Forward and Viterbi algorithms, we can parallelize the BWA across states. For the Forward and Backward variables, this gives us $O(T \log N)$. γ requires $O(T)$, and ξ requires $O(T \log N)$. Re-estimating A utilizes $O(\log N)$ operations. Re-estimating B requires $O(T \log N) + O(\log M) = O(T \log N)$ for $T > \log M$. Finally, re-estimating Π uses $O(\log N)$ operations. Except for unusually large values of M, we can assume that the computational complexity of the parallelized BWA will be $O(T \log N)$.

3.4. Complexity Summary

Table 1 shows the computational complexity for the serial and parallel versions of the various HMM algorithms.

4. GPU ARCHITECTURE

The GPU is divided into software and hardware logical sections that map to each other in various ways. On the software side the programmer makes use of kernels, which is the heart of NVIDIA's CUDA extension of C [9]. A kernel is similar to a function call, but it is executed on the GPU. This SIMD operation is accomplished via threads in hardware. The code in the kernel is executed at the same time on each thread. However, each thread may be told to operate on different data.

Table 1. Computational Complexity Comparison

Algorithm	Serial	Parallel
Forward	$O(TN^2)$	$O(T \log N)$
Viterbi	$O(TN^2)$	$O(T \log N)$
Baum-Welch	$O(TN^2)$ or $O(TMN)$	$O(T \log N)$

On the hardware side, blocks are composed of a number of threads. Threads within a block are capable of communicating and sharing data through the use of shared memory. All of the blocks that are executed run on a grid. Generally, a GPU can handle a single grid at a time, but this can be overcome by using multiple GPUs or architectures that support multiple grids. Blocks within a grid have no good means to communicate with each other, offer no guarantee of parallel execution, and can preempt execution of other blocks.

Due to the limited guarantee of block concurrency, the parallel execution of kernels sees a speedup increase only to a certain degree. Once a grid is saturated with executing blocks, the other blocks must wait their turn. As a result, the speedup over serial execution will reach a point of diminishing returns. Diminishing returns have specific ramifications on HHMs.

Additionally, the lack of true inter-block communication poses a problem for HMMs, specifically the Viterbi algorithm. Step 2 of the Viterbi requires the maximum value or argument to be found. Normally, this can be done in a O(N) serial operation. With an SIMD operation, a technique known as a reduction is used, which reduces the computational complexity to O($\log N$). Once the dataset becomes large enough (i.e. more data than threads per block), inter-block communication is required to accomplish a full reduction. The only true inter-block communication is done by performing multiple kernel calls, each of which can add micro- to milli-seconds of overhead. Until a better method of inter-block communication is supported, reductions such as the one in the Viterbi algorithm will suffer additional time delays.

The biggest drawback to using GPGPU is the large overhead incurred during memory transfers to and from the GPU. Generally, the accepted GPGPU practice is to transfer a large chunk of data to the GPU's memory at the beginning of the program, run the necessary calculations, then transfer the results back. This model was followed for the HMM implementation outlined in this paper. However, several of

the algorithms such as the BWA ran out of GPU memory when performing calculations using a large number of HMM states (greater than 10,000). If more states are required, the program will have to be re-written for the current hardware to support freeing of GPU memory during calculations. This action, however, could incur severe penalties on the execution time of the algorithm(s).

5. PERFORMANCE EVALUATION

In order to evaluate the performance of the different algorithms, each algorithm was programmed first in C, utilizing a serial implementation. Then the algorithms were programmed in NVIDIA's CUDA language. Several functions relied on the CUDA-implemented Basic Linear Algebra Subprograms (CUBLAS), which takes advantage of CUDA functions to accomplish various matrix operations.

The algorithms were then given different HMM parameters and timed. First, the number of states were varied, followed by the number of symbols, and finally, the number of observations. For the given hardware, the C and CUDA implementations for all algorithms took the same amount of time with about 60 states. Therefore the other tests (symbol varying and time varying), were conducted with $N = 60$ for a more accurate comparison.

Additionally, the power measurement tool PowerTOP was used to estimate the power consumption of the CPU and GPU under load. Each algorithm was run for 1 minute before a reading from PowerTOP was taken.

The tests were conducted on the hardware given in Table 2.

Table 2. Test Hardware

Component	Specification
CPU	Intel Core 2 Duo U7300 @ 1.30GHz
GPU	NVIDIA GeForce GT 335M
GPU Core Speed	450 MHz
GPU Shader Speed	1080 MHz
GPU Memory Speed	1066 MHz
CUDA Cores	72

6. RESULTS

The various algorithms were each timed on the CPU and GPU. Figures 1-3 show the various timing results from the Forward Algorithm. The Viterbi Algorithm is depicted in figures 4-6, and figures 7-9 show the timing results from the Baum-Welch Algorithm.

Examining the graphs given in figures 1-9, it is apparent that the greatest increase in speed comes from utilizing the GPU with a large number of states, typically more than 60. Table 3 gives the speed increases for a select number of states.

Table 3. Speed Increase for Each Algorithm

Number of States	CPU Runtime (s)	GPU Runtime (s)	Speed Increase
Forward Algorithm			
4	0.0010	0.1531	0.007x
40	0.0400	0.1393	0.287x
400	4.2816	0.2379	17.99x
4000	534.2028	2.9495	181.12 x
Viterbi Algorithm			
4	0.0033	0.1605	0.021x
40	0.0436	0.1801	0.242x
400	4.2684	1.6595	2.57x
4000	534.5543	116.2531	4.60 x
Baum-Welch Algorithm			
4	0.0021	0.4142	0.005x
40	0.1946	0.4299	0.453x
400	17.6719	0.7502	23.56x
4000	1834.672	28.1271	65.23 x

It can be seen that for a low number of states, the GPU performs far worse than the CPU. This is due to the overhead incurred when utilizing the GPU, as data must be copied to the GPU's memory and copied back to the host. Additionally, most GPU clock speeds are not as fast as those found in CPUs; GPUs are simply not designed to handle serial data computations. However, applications that require a large number of states (60+ in this case) perform quite well with a GPU.

This speedup works up to a certain point, however. As shown by figures 10-12, the GPU begins to lose its strict parallel processing after about 200-500 states. These diminished returns can be attributed to saturating the processing blocks found within the GPU. Once this point is reached, other blocks must wait their turn to be processed. As a result, the speedup gains do not improve as drastically after a certain point. This can be remedied with larger, more powerful GPUs that can handle more blocks at a time or several GPUs working in parallel.

Additionally, it was found that varying the number of symbols and number of observations had little impact on the execution speed difference between the CPU and GPU. In most cases, the CPU performed slightly better due to the increased overhead found in the GPU.

As shown in table 3, the speed increases can reach 180x for the Forward Algorithm, 65x the BWA, and 4x for the Viterbi Algorithm with 4000 states. These speed

increases at a high number of states are well within the block saturation point of the GPU, but still continue to outperform the CPU. As noted in section 4, the lackluster performance of the parallel implementation of the Viterbi algorithm can be attributed to the reduction technique requiring multiple kernel calls.

The results of the speedup from the Viterbi algorithm compares to the work of Zhang et al. [6]. However, the other algorithms do not quite line up with the results from [4] and [5] as their work included additional parallel steps, such as evaluating multiple models at a time. The implementations found in this paper focuses strictly on the algorithms themselves. Any additional parallel steps can only improve the performance of the application.

In addition to the performance times, the energy consumed for each of the algorithms in the CPU versus the GPU was assessed. This value was estimated by measuring the power utilization during the algorithm and multiplying by the computation time. Table 4 shows the measured power for each algorithm and figures 13-15 depict the total energy consumed during the algorithm runtime. Note that the idle power was measured at 13.8W. Due to the increase in power requirements of the GPU, the break-even point for energy consumption is 100 states for the forward algorithm, 120 states for the Viterbi algorithm, and 70 states for the BWA.

Table 4. Power Utilization

Algorithm	Power (W)	
	C	CUDA
Forward	18.5	26.5
Viterbi	18.5	29.1
BWA	18.3	26.1

7. CONCLUSION AND FUTURE WORK

GPUs currently boast some of the most impressive computing power vs. electrical power efficiency ratings (measured in GFLOPS/Watt). However, this kind of power can only be utilized under special circumstances. If an algorithm cannot be parallelized, then many GPU computing blocks remain dormant, and efficiency is lost. For HMMs, this means that only models with a large number of states can truly take advantage of the GPU's parallel computing capabilities.

Applications such as geolocation and spectrum sensing and classification could potentially use HMMs with hundreds of states. 2D and 3D HMMs could be used in these applications, but suffer from large computational complexity. While the power requirements of a GPU still exceeds that of a CPU for the same functions, massively parallel computations can still prove more power efficient in a GPU. This is especially true of new technology that promises smaller, more efficient GPUs in handheld devices.

Additionally, HMMs are also used in other fields outside of wireless technology, such as speech/handwriting recognition, financial research, economics, and protein folding. Some of these applications use hundreds or thousands of states, and would benefit greatly from GPUs.

It should be noted that the code used to test the algorithm runtimes contained minimal optimization (e.g. only that offered by the compiler and the use of the CUBLAS library). Both the C and CUDA implementations could undoubtedly benefit from programming optimizations to take advantage of other hardware speed increases, including optimizing the reduction techniques used for the Viterbi algorithm. Furthermore, some applications, such as pattern recognition, rely on solving the same algorithm with different sets of data. For example, to find the best model match to an observed pattern, the Forward Algorithm is run on the observation sequence a number of times equal to the number of models in the database. This kind of parallelism could also see potential gains in a GPU.

In summary, the GPU provides promising runtime improvements over the CPU for the various HMM algorithms, given that the model contains a large number of states. As GPUs become smaller and more efficient, they could indeed prove to be a contender or supplement to CPUs, DSPs, and FPGAs for wireless applications.

8. REFERENCES

[1] Z. Chen, Z. Hu, and R. C. Qui, "Quickest Spectrum Detection Using Hidden Markov Models for Cognitive Radio," *Milcom*, 2009.

[2] K. Kim, I. Akbar, K. K. Bae, J. Um, C. M. Spooner, and J. H. Reed, "Specific Emitter Identification for Cognitive Radio with Application to IEEE 802.11," *IEEE Globecom*, 2008.

[3] A. Fehske, J. Gaeddert, and J. H. Reed, "A New Approach to Signal Classification Using Spectral Correlation and Neural Networks," *IEEE International Symposium on New Frontiers in DySpan*, Nov. 2005.

[4] J. Li, S. Chen,and Y. Li, "The Fast Evaluation of Hidden Markov Models on GPU," *IEEE International Conference on Intelligent Computing and Intelligent Systems*, Nov. 2009.

[5] C. Liu. (2006, May). cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. [Online]. Available: http://liuchuan.org/pub/cuHMM.pdf

[6] D. Zhang, R. Zhao, L. Han, T. Wang, and J. Qu, "An Implementation of Viterbi Algorithm on GPU," *International Conference on Information Science and Engineering*, 2009.

[7] M. Woh et al, "From SODA to Scotch: The Evolution of a Wireless Baseband Processor," *International Symposium on Microarchitecture, 2008*. Nov. 2008.

[8] H. Othman and T. Aboulnasr, "A Separable Low Complexity 2D HMM with Application to Face Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, Oct. 2003.

[9] NVidia, *NVidia CUDA Compute Unified Device Architecture Programming Guid*e, 2009.

[10] L. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, vol. 77, no. 2, Feb. 1989.
[11] A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260-269, April 1967.
[12] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics*, vol. 41, pp. 164-171, 1970.
[13] B. Juang and L. Rabiner, "The Segmental K-Means Algorithm for Estimating Parameters of Hidden Markov Models," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, no. 9, Sep. 1990.
[14] W. Tranter, et al., *Principles of Communication Systems Simulation with Wireless Applications*, Upper Saddle River, New Jersey; Prentice Hall, 2004, ch. 15, pp. 605-611.

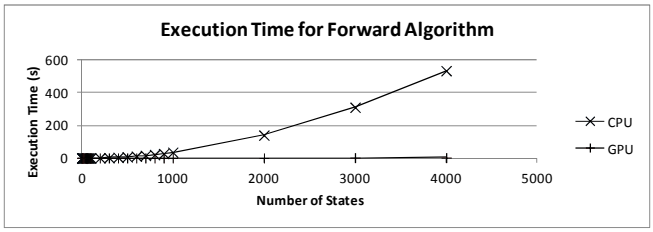

Figure 1. Varying States in the Forward Algorithm

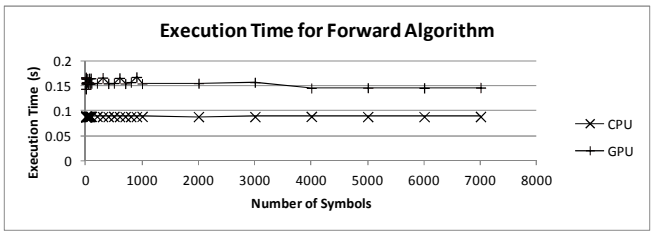

Figure 2. Varying Symbols in the Forward Algorithm

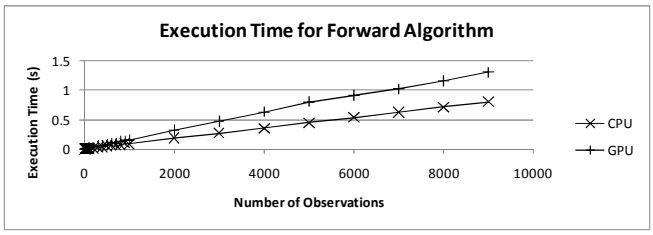

Figure 3. Varying Sequence in the Forward Algorithm

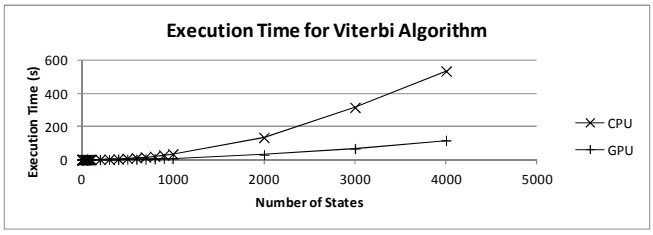

Figure 4. Varying States in the Viterbi Algorithm

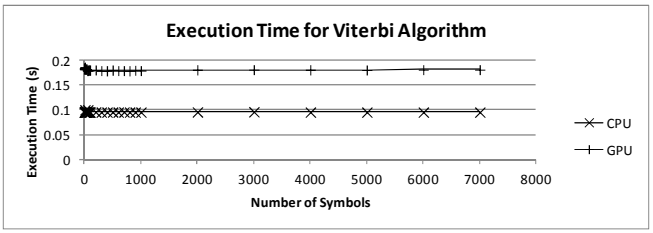

Figure 5. Varying Symbols in the Viterbi Algorithm

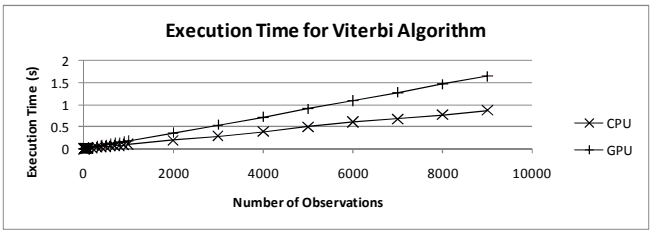

Figure 6. Varying Sequence in the Viterbi Algorithm

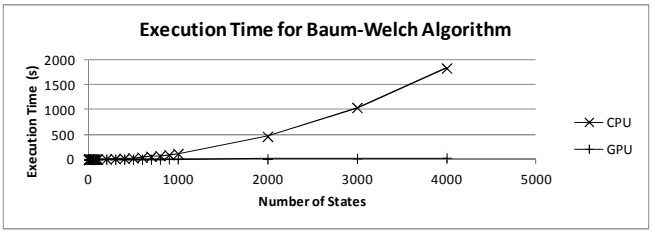

Figure 7. Varying States in the BWA

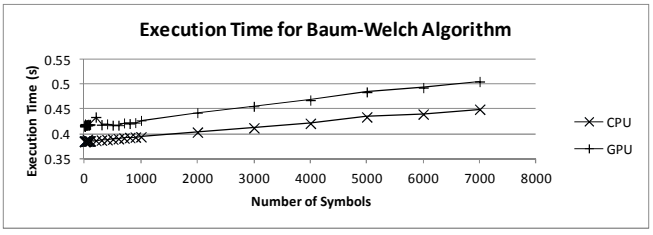

Figure 8. Varying Symbols in the BWA

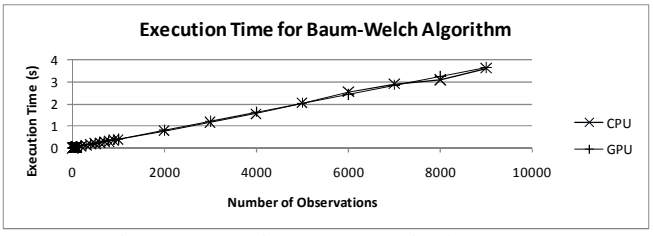

Figure 9. Varying Sequence in the BWA

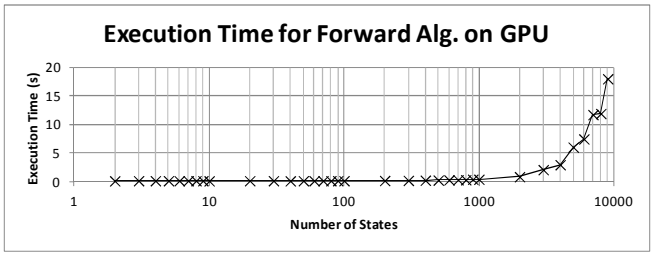

Figure 10. Varying States for Forward Algorithm on GPU

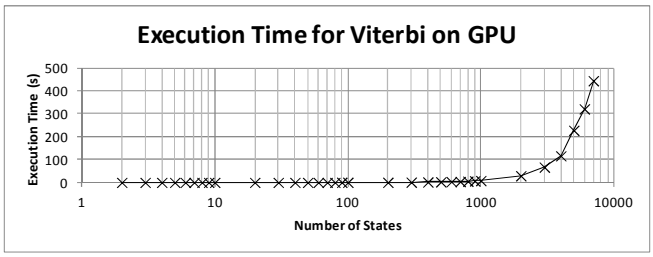

Figure 11. Varying States for Viterbi Algorithm on GPU

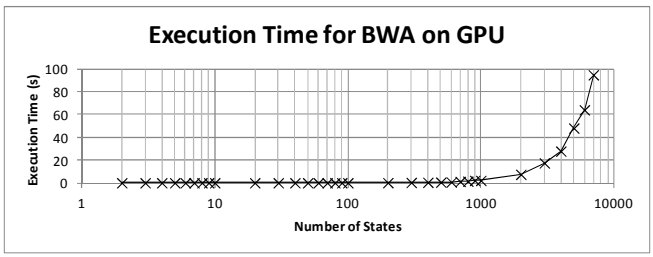

Figure 12. Varying States for BWA on GPU

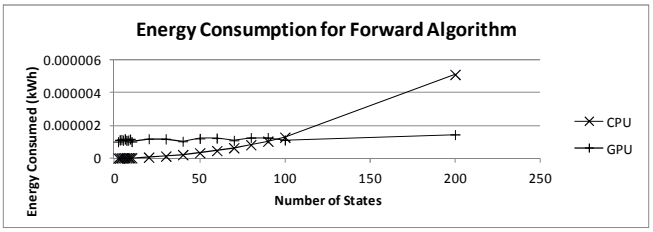

Figure 13. Energy Consumption for Forward Algorithm

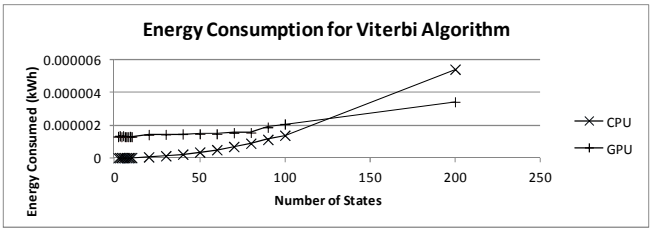

Figure 14. Energy Consumption for Viterbi Algorithm

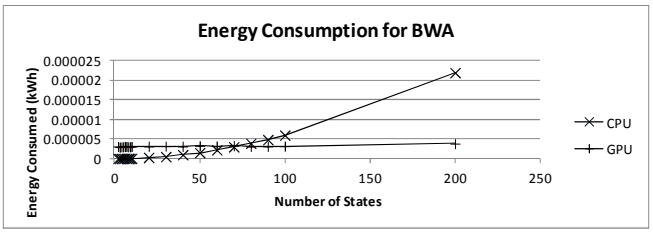

Figure 15. Energy Consumption for BWA