# GPU-Accelerated Out-of-Core HMM Inference with Concurrent CUDA Streams

MohammadReza HoseinyFarahabady[0000−0002−7851−9377] and Albert Y. Zomaya[0000−0002−3090−1059]

The University of Sydney, School of Computer Science, Center for Distributed and High Performance Computing, Sydney, NSW, Australia
{reza.hoseiny,albert.zomaya}@sydney.edu.au

**Abstract.** Hidden Markov Models (HMMs), along with their extension, Hidden Semi-Markov Models (HSMMs), are powerful tools for modeling complex systems with multivariate states, but scaling them to ultra-large state spaces presents significant computational and memory challenges. This paper proposes a hybrid CUDA-based implementation to overcome these limitations, enabling efficient processing of HMMs with massive state spaces and extended observation sequences. Key optimizations include log-space computations for numerical stability, memory-efficient data partitioning with sparse matrix representations, and asynchronous data transfers using CUDA streams to overlap host-device communication with GPU kernel execution. Our approach achieves significant performance improvements, demonstrating up to 8.5× speedup over multi-threaded CPU implementations for HMM processing.

**Keywords:** Complex Systems with Large State Spaces · Hidden Markov Models · GPU Acceleration · Performance Optimization · Memory Coalescing · Asynchronous CUDA Streams.

## 1 Introduction

Hidden Markov Models (HMMs), introduced by Baum and Petrie [1], and their extension, Hidden Semi-Markov Models (HSMMs), are powerful tools for modeling complex systems with multivariate states and hidden structures [5]. Despite their effectiveness, traditional algorithms like Baum-Welch and Viterbi struggle with computational and memory demands in large-scale scenarios. HMMs, defined as doubly stochastic processes with hidden states inferred from observable sequences, are widely applied in fields such as speech recognition [10], bioinformatics [8], and finance [9].

An HMM is characterized by five key components:

- A set of $N$ hidden states $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$ represents the underlying stochastic process, where each state is not directly observable. The state at time $t$ is denoted by the variable $q_t$, where $q_t \in \mathcal{S}$.
- A set of $M$ observable symbols $\mathcal{V} = \{v_1, v_2, \ldots, v_M\}$.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

- A state transition probability matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ with elements $a_{ij}$, where $a_{ij} = P(q_{t+1} = S_j | q_t = S_i)$ represents the probability of transitioning from state $S_i$ at time $t$ to state $S_j$ at time $t + 1$.
- An emission probability matrix $\boldsymbol{B} \in \mathbb{R}^{N \times M}$ with elements $b_j(m)$, where $b_j(m) = P(O_t = v_m | q_t = S_j)$ represents the probability of observing symbol $v_m$ at time $t$ given that the hidden state is $S_j$.
- An initial state probability vector $\boldsymbol{\pi} \in \mathbb{R}^N$, where $\pi_i = P(q_1 = S_i)$ represents the probability of starting in state $S_i$.

These components, collectively denoted as $\lambda = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\pi})$, enable HMMs to address three fundamental problems [7, 4]:

- *Evaluation*: To estimate the likelihood of an observed sequence $O = (O_1 \dots O_T)$ given the model $\lambda$, denoted as $P(O|\lambda)$.
- *Decoding*: To determine the most probable sequence of hidden states $Q = (q_1, q_2, \dots, q_T)$ that generated the observed sequence $O$, i.e., maximizing $P(Q|O, \lambda)$.
- *Learning*: To estimate the model parameters, $\lambda$, to maximize the likelihood of a training set of observation sequences.

This paper introduces a hybrid CPU-GPU out-of-core algorithm designed to efficiently process HMMs with massive state spaces and longer observation sequences. Our approach integrates several optimization techniques that leverages the parallel processing capabilities of modern GPUs while overcoming memory constraints through innovative optimizations. First, we employ log-space computations to mitigate numerical underflow, ensuring stable and accurate results during algorithm execution in the GPU space. Second, we implement a memory-efficient data partitioning strategy that utilizes sparse matrix representations to optimize memory utilization of device. Large matrices are partitioned into blocks, with dimensions dynamically determined by the available GPU memory which enable efficient processing of large-scale HMMs. These blocks are processed sequentially on the GPU to enhance computational throughput. We also leverage asynchronous data transfers via CUDA streams and a triple-buffered pipeline to overlap data transfers between the host and device with kernel execution on GPU to hide the data-transfer latencies. We further optimize memory access patterns through memory coalescing technique for both dense and sparse matrix computations.

The remainder of this paper is organized as follows: Section 2 provides an overview of key background concepts, including the fundamentals of Hidden Markov Models, the Baum-Welch algorithm, and the challenges associated with scaling HMMs for large state spaces. Section 3 presents our proposed hybrid CPU-GPU out-of-core algorithm, detailing the optimizations implemented for efficient GPU utilization. Section 4 describes the experimental setup and performance evaluation methodology used to assess the effectiveness of our GPU implementation compared to CPU-based approaches. Finally, Section 5 concludes the paper with a discussion of the findings and outlines directions for future research in optimizing HMMs on GPU architectures.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

## 2 Background

Baum-Welch Algorithm is an Expectation-Maximization (EM) technique used for parameter estimation in HMMs. It is an iterative algorithm designed to optimize such parameters when the hidden states are unknown, based only on observed data. The key essential part for this algorithm parameter lies in the *forward-backward procedure* [2].

**The Forward-Backward Procedure.** A central problem in HMMs is efficiently computing the probability of an observed sequence $O = (O_1, O_2, \ldots, O_T)$ given the model parameters $\lambda$. A naive approach of enumerating all possible state sequences of length $T$ has a computational complexity of $\mathcal{O}(TN^T)$, where $N$ is the number of hidden states and $T$ is the number of observations [3, 6]. This exponential complexity renders the direct computation intractable even for moderately sized sequences and models. The forward-backward procedure [1] provides a computationally efficient solution by leveraging dynamic programming. The forward pass computes the forward probabilities $\alpha_t(j) = P(O_1, \ldots, O_t, q_t = S_j|\lambda)$, representing the probability of observing the sequence up to time $t$ and being in state $S_j$ at that time. The recursion step calculates $\alpha_t(j)$ by summing over all previous states $S_{i\leq j-1}$, weighted by the transition probabilities $A_{ij}$ and the emission probability of the current observation $O_t$ in state $S_j$. The algorithm's time complexity is $\mathcal{O}(TN^2)$, significantly more efficient than a direct computation. The backward pass computes the probability of the remaining observation sequence $(O_{t+1} \ldots O_T)$ given state $S_j$ at time $t$, denoted by $\beta_t(j) = P(O_{t+1}, O_{t+2}, \ldots, O_T \mid q_t = S_j, \lambda)$. The recursion step calculates $\beta_t(j)$ by summing over all possible next states $S_k$, weighted by the transition probability $A_{jk}$, emission probability $B_k(O_{t+1})$, and the subsequent backward probability $\beta_{t+1}(k)$. The algorithm has a time complexity of $\mathcal{O}(TN^2)$.

**Parameter Estimation with Baum-Welch.** The Baum-Welch algorithm act as an unsupervised learning technique that iteratively trains HMM parameters using the forward-backward procedure to find the parameters of the HMM, *i.e.,* $\lambda$, that maximize the probability of a given observation sequence. The algorithm comprises two main steps:

1. **E-Step (Expectation Step):** Compute the expected number of state transitions and emissions using forward and backward probabilities for all time steps $t$ and states $S_j$. We define the variable $\xi_t(i, j)$ as the probability of being in state $S_i$ at time $t$ and in state $S_j$ at time $t+1$ given the observation sequence and the model, *i.e.,* $P(O \mid \lambda)$. So, the expected number of transitions from $S_i$ to $S_j$ can be calculated as:

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

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

2. **M-Step (Maximization Step):** Re-estimate the model parameters based on the E-step's expected values:

$$A_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}, \quad B_j(k) = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot \mathbb{I}(O_t = k)}{\sum_{t=1}^{T} \gamma_t(j)}, \tag{2}$$

where $\mathbb{I}(O_t = k)$ is an indicator function (1 if $O_t = k$, 0 otherwise), and $\gamma_t(j)$ is the posterior probabilities that estimate the likelihood of being in state $S_j$ at time $t$ given the entire observation sequence and the HMM model. It can be computed using $\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$. Summing $\gamma_t(j)$ over $t$ gives the expected number of transitions made from state $S_j$. The algorithm iterates these steps until convergence (e.g., no significant change in the log-likelihood or a maximum number of iterations is met).

## 3 Our Approach

We explored the potential of GPU acceleration to enhance the performance of Hidden Markov Models with large state spaces by implementing the Baum-Welch algorithm using the `CUDA cuBLAS` library. This implementation was specifically designed to leverage the massive parallelism of modern GPUs, enabling highly efficient matrix operations, which are central to HMM computations.

However GPU acceleration is fundamentally limited by the finite capacity of GPU memory. This limitation becomes particularly challenging when processing Hidden Markov Models with ultra-large state spaces or extended observation sequences, where the required data structures, such as the forward and backward probability matrices, exceeds the available GPU memory. To overcome these memory constraints and enable scalable processing of large-scale HMMs on GPUs, we propose a hybrid CUDA implementation incorporating the following key techniques:

- *Log-Space Implementation for Numerical Stability*: Mitigates numerical underflow issues by performing computations in the log domain, ensuring stable results.
- *Memory-Efficient Data Partitioning and Sparse Representations*: Optimizes memory utilization by partitioning large data structures into smaller blocks and leveraging sparse matrix representations and memory coalescing where applicable.
- *Asynchronous Data Transfer for Computation-Communication Overlap*: Maximizes GPU utilization by overlapping data transfers between the host and device memories with kernel execution to minimize the idle time.

These strategies address both the memory capacity limitations of GPUs that enables efficient computation of HMMs with large state spaces and long observation sequences. The following sections detail these techniques.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

### Dynamic Memory Allocation and Data Partitioning

The algorithm begins by analyzing the HMM parameters: the number of states, transition probabilities, and the length of the observation sequence. Based on these parameters, the memory required for the key data structures, *i.e.,* transition matrix, emission matrix, and forward ($\alpha$) and backward ($\beta$) probability matrices, is dynamically calculated. If the total memory required to store these matrices exceeds the available GPU memory, a data partitioning strategy is employed. The matrices are divided into logical blocks, which are then processed sequentially on the GPU. The dimensions of these blocks are determined based on the available GPU memory and the dimensions of the original matrices. Specifically:

- The transition matrix $A$ is partitioned into blocks of size $b_A \times b_A$.
- The emission matrix $B$ is partitioned into blocks of size $b_B \times M$.
- The forward and backward matrices $\alpha$ and $\beta$ are each partitioned into blocks of size $N \times b_T$.

The block sizes $b_A$, $b_B$, and $b_T$ are chosen to ensure that the combined memory size of active sparse matrix blocks residing in GPU memory remains lower than the available GPU memory capacity. These blocks are also transferred asynchronously between the host memory and the device memory using CUDA streams. The asynchronous transfer mechanism overlaps data transfers with computation to ensure minimizing idle GPU time while maximizing the utilization of GPU core cycles.

### Sparse Matrix Representations

In many real-world scenarios, HMMs have sparse transition and emission matrices with numerous near-zero elements. To exploit this sparsity, we use *Compressed Sparse Row (CSR)* or *Compressed Sparse Column (CSC)* formats depending on the computation. These formats minimize data transfer overhead and optimize GPU memory usage by excluding redundant near-zero values, enabling more efficient processing and avoiding unnecessary computations. For a sparse matrix $X \in \mathbb{R}^{N \times M}$, the CSR representation of $X$ is defined by three arrays:

- *Values:* An array containing all non-zero elements of $X$.
- *Column Indices:* An array storing the column index of each corresponding non-zero value.
- *Row Pointers:* An array of size $N+1$, where the $i$-th element specifies the index in the *Values* array where the $i$-th row begins. The last element indicates the end of the data in the *Values* array.

To enable efficient sparse matrix computations, we utilize the *optimized sparse matrix-vector multiplication (SpMV)* routines from the `CUDA cuSPARSE` library. These routines are highly optimized for parallelism, ensuring efficient

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

access to the CSR/C data structure, minimizing warp divergence by aligning threads with non-zero elements, and maximizing memory coalescing for accessing the *Values* and *Column Indices* arrays.

*Memory coalescing* is another GPU optimization technique that maximizes GPU memory bandwidth utilization by ensuring that threads within a warp access contiguous memory locations. This contiguous access pattern minimizes the number of separate memory transactions required to fetch data for the warp. Instead of multiple scattered accesses, coalescing combines them into a single transaction that can leverage the GPU's memory bus width. This reduction in number of transactions directly translates to improved performance for memory-bound applications. In our implementation, we leverage memory coalescing to optimize access patterns for both dense and sparse matrix representations. For dense matrices, we structure the data layout to ensure that threads within a warp access consecutive elements within a row or column, depending on the matrix's layout. For example, if the computation processes rows, threads in a warp access elements as `matrix[row][thread_id + warp_base_index]`, where `warp_base_index` is aligned to a multiple of the warp size. This alignment guarantees that all the data needed for an entire warp is fetched in a single memory transaction to minimize memory access overhead. For sparse matrix representations, we structure the layout of the *Values* and *Indices* arrays so that threads within a warp process consecutive non-zero elements in a row (or column) and their corresponding column (or row) indices in a contiguous manner.

**Asynchronous Data Transfers and Kernel Execution**

One of the key bottlenecks in GPU-based computation is the data transfer between the host and device memory. To address this issue, we implement asynchronous data transfers using CUDA streams that enable concurrent data movement and kernel execution for increased computational throughput. This technique facilitates computation-communication overlap which effectively reduces GPU idle time. This optimization is particularly important for dense and sparse matrix-vector multiplication operations that form the core of HMM forward, backward, and Baum-Welch training algorithms.

Our asynchronous execution strategy is implemented using a triple-buffered pipelined approach managed by CUDA streams. The pipeline allows us to overlap data transfers with computation to hide the latency of data transfers behind computation. The process is detailed below:

- *Memory Pinning*: Host memory buffers are allocated using `cudaHostAlloc()` with the `cudaHostAllocDefault` flag. This ensures the allocated memory is pinned which allows the GPU to directly access the data via Direct Memory Access during transfers.
- *Stream Creation*: Three distinct CUDA streams are created. These streams operate independently and concurrently which enable the pipelined execution.
- *Triple-Buffered Pipelined Execution*: A three-stage pipeline is implemented using the three streams.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

- *Data Transfer to Device (Stream 0)*: `cudaMemcpyAsync()` is used within stream 0 to transfer a data block from the pinned host memory to the GPU's global memory. This transfer is non-blocking which allows subsequent operations to begin immediately. The transfer size is determined by the size of the data required for the target HMM computation step.
- *Kernel Execution (Stream 1)*: While the data transfer in stream 0 is in progress, stream 1 can start execution of the computational kernel on a previously transferred data block residing in GPU memory.
- *Result Transfer to Host (Stream 2)*: Concurrently with the kernel execution in stream 1 and the data transfer in stream 0, stream 2 transfers the computed results from GPU global memory back to a separate pinned host memory buffer using asynchronous transfer primitives.

## 4 Experimental Results

This section evaluates the performance of our GPU-accelerated HMM implementation against a multi-threaded CPU baseline. Hidden state sizes ($N$) range from $2^{18}$ to $2^{25}$ and observation lengths ($T$) from $2^{10}$ to $2^{13}$. For each ($N$, $T$) pair, 10 randomly initialized HMMs were tested with average matrix sparsity of 1%. We used the 99th-percentile execution time to ensure robust comparisons and analyzed speedup and throughput (in TFLOPS) across CPU and GPU implementations. Experiments ran on a workstation system with an Intel Core i7-14700K (28 logical cores), 128GB DDR5 RAM, and an NVIDIA A10 GPU (24GB, 600GiB/s bandwidth). CPU tests used Intel OneAPI with TBB and C BLAS, while GPU tests employed CUDA 12.6.85 and cuBLAS with `nvcc` optimizations.

Figure 1 shows the p99 execution time (log scale) of the Baum-Welch algorithm (100 iterations, sequence length 1024) as the number of hidden states increases. Due to large matrix sizes, our GPU implementation employed custom memory management techniques described earlier. Single-threaded CPU execution time grew linearly from 334.6s at $N = 2^{18}$ to 47,350s at $N = 2^{25}$. The 28-threaded CPU reduced this to 5625.0s (8.4× speedup). In comparison, our GPU implementation achieved 993.7s at $N = 2^{25}$, a 5.6× speedup over the multi-threaded CPU, demonstrating both scalability and the effectiveness of GPU acceleration for large-scale HMM inference.

## 5 Conclusion

This work presents a GPU-accelerated Baum-Welch algorithm using `CUDA cuBLAS` to efficiently scale large Hidden Markov Models. Achieving up to an 8.5× speedup over multi-threaded CPU approaches, our method leverages log-space computation, memory-efficient sparse partitioning, and asynchronous data transfer. Despite notable gains, challenges in host-device transfer and memory access persist. Future directions include adaptive memory management, hardware-aware optimizations, and support for next-gen GPU architectures to enable real-time processing of large-scale HMMs.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44

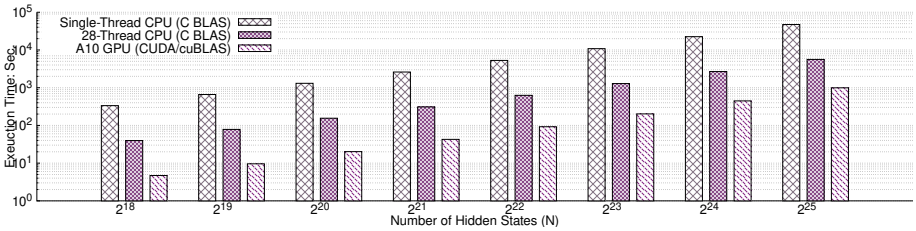


**Fig. 1.** Execution times (log scale) for Baum-Welch algorithm scaling (100 iterations, 1024 sequence length) with increasing hidden states for single-thread CPU, 28-thread CPU, and our proposed GPU implementations.

**Acknowledgments.** Professor Albert Y. Zomaya would like to acknowledge the support of the **ONI Grant NI220100111**. Dr. MohammadReza HoseinyFarahabady acknowledges the continued support of The Center for Distributed and High-Performance Computing at The University of Sydney for giving access to advanced high-performance computing and cloud facilities, digital platforms, and necessary tools.

## References

1. Baum, L.E., Petrie, T.: Statistical inference for probabilistic functions of finite state markov chains. The Annals of Mathematical Statistics **37**(6), 1554–1563 (1966). https://doi.org/10.1214/aoms/1177699147
2. Bouguila, N., Fan, W., Amayri, M.: Hidden Markov Models and Applications. Springer (2023). https://doi.org/10.1007/978-3-030-99142-5
3. Dymarski, P.: Hidden Markov Models, Theory and Applications. IntechOpen (2011). https://doi.org/10.5772/1532
4. Elliott, R.J., Aggoun, L., Moore, J.B.: Hidden Markov Models: Estimation and Control. Springer (2008). https://doi.org/10.1007/978-0-387-84854-9
5. Ibe, O.C.: Markov Processes for Stochastic Modeling. Elsevier, 2 edn. (2009). https://doi.org/10.1016/B978-0-12-374451-0.X0001-7
6. Ibe, O.C.: Markov Processes for Stochastic Modeling. Elsevier, 2 edn. (2013). https://doi.org/10.1016/C2012-0-06106-6
7. Mamon, R.S., Elliott, R.J.: Hidden Markov Models in Finance. Springer (2007). https://doi.org/10.1007/0-387-71163-5
8. Miller, D.R., Leek, T., Schwartz, R.M.: A hidden markov model information retrieval system. In: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 214–221 (1999). https://doi.org/10.1145/312624.312680
9. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989). https://doi.org/10.1109/5.18626
10. Schuller, B., Rigoll, G., Lang, M.: Hidden markov model-based speech emotion recognition. In: 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing. vol. 2, pp. II–1 (2003). https://doi.org/10.1109/ICME.2003.1220939

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_44