FLASH Viterbi: Fast and Adaptive Viterbi Decoding for Modern Data Systems

Ziheng Deng[1,†], Xue Liu[1,†], Jiantong Jiang[2], Yankai Li[1], Qingxu Deng[1,*], Xiaochun Yang[1]
[1]*School of CSE, Northeastern University, China* [2]*School of CIS, The University of Melbourne, Australia*
[1]{2310721@stu, liuxue@cse, 20215828@stu, dengqx@mail, yangxc@mail}.neu.edu.cn, [2] jiantong.jiang@unimelb.edu.au

***Abstract*—The Viterbi algorithm is a key operator for structured sequence inference in modern data systems, with applications in trajectory analysis, online recommendation, and speech recognition. As these workloads increasingly migrate to resource-constrained edge platforms, standard Viterbi decoding remains memory-intensive and computationally inflexible. Existing methods typically trade decoding time for space efficiency, but often incur significant runtime overhead and lack adaptability to various system constraints. This paper presents FLASH VITERBI, a Fast, Lightweight, Adaptive, and Hardware-Friendly Viterbi decoding operator that enhances adaptability and resource efficiency. FLASH VITERBI combines a non-recursive divide-and-conquer strategy with pruning and parallelization techniques to enhance both time and memory efficiency, making it well-suited for resource-constrained data systems. To further decouple space complexity from the hidden state space size, we present FLASH-BS VITERBI, a dynamic beam search variant built on a memory-efficient data structure. Both proposed algorithms exhibit strong adaptivity to diverse deployment scenarios by dynamically tuning internal parameters. To ensure practical deployment on edge devices, we also develop FPGA-based hardware accelerators for both algorithms, demonstrating high throughput and low resource usage. Extensive experiments show that our algorithms consistently outperform existing baselines in both decoding time and memory efficiency, while preserving adaptability and hardware-friendly characteristics essential for modern data systems. All codes are publicly available at https://github.com/Dzh-16/FLASH-Viterbi.**

***Index Terms*—Viterbi decoding, resource-adaptive, parallelism, space efficiency, hardware-friendly, edge computing**

I. INTRODUCTION

Viterbi decoding [1] is a critical operator for real-time structured inference in data systems, particularly in edge environments with limited resources. As the core decoding algorithm in probabilistic sequence models and chain-structured probabilistic graphical models such as Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) [2]–[14], it recursively computes the optimal state sequence by maximizing the joint likelihood over all possible paths. This capability enables a wide range of data-driven applications like mobility trajectory inference [3]–[6], online recommendation [15], behavioral labeling [16], and speech recognition [17]–[20].

However, as inference tasks grow increasingly complex, the standard Viterbi algorithm (i.e., Vanilla Viterbi) remains resource-intensive and ill-suited for edge deployment [21]. Specifically, when applied to an HMM, it incurs a space complexity of $\mathcal{O}(KT)$ and a time complexity of $\mathcal{O}(K^2T)$, where K is the hidden state size and T is the observation sequence length, as illustrated in Figure 1. They scale linearly or quadratically with K and T, hindering deployment on resource-limited edge platforms [21]–[23]. Such resource demands have emerged as a challenge in modern serving systems [24], [25].

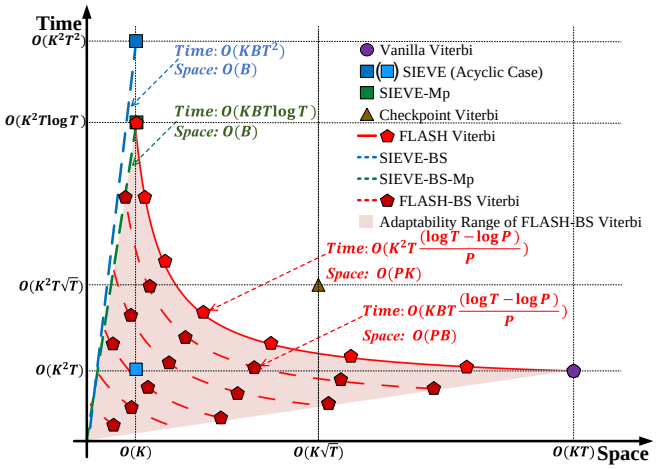

Fig. 1: Theoretical time-space complexity comparison of FLASH VITERBI, FLASH-BS VITERBI, and baseline algorithms. K is the HMM state space size, T is the sequence length, B is the beam width, and P is the parallelism degree.

Recent efforts have attempted to trade off decoding time and memory usage [17], [18], [26]–[28], yet still incur high memory overhead or significant latency. Importantly, most approaches lack adaptivity to system-level constraints and fail to optimize Viterbi under the dynamic resource limitations typical of modern data systems, especially on edge platforms.

To this end, we propose FLASH VITERBI and its beam search variant FLASH-BS VITERBI, a new family of fast, lightweight, adaptive, and hardware-friendly Viterbi algorithms that address the challenges of space and time efficiency jointly. We aim to support Viterbi as a modular operator within real-time processing pipelines on edge systems. Specifically, FLASH VITERBI adopts a non-recursive divide-and-conquer strategy to decompose the decoding task into subtasks, improving memory reuse and reducing scheduling overhead compared to recursive methods. Moreover, this strategy enables large-scale parallelism by reordering subtask execution. At the subtask level, a unified pruning and parallelization strategy eliminates redundant computations and interdependencies, thereby unlocking full parallelism. Building on this, FLASH-

† These authors contributed equally to this work. *Corresponding author.

BS VITERBI integrates dynamic beam search with an efficient data structure tailored for frequent candidate path updates. Both algorithms employ a double-buffered memory scheme to reduce the overhead of intermediate data exchange.

These designs collectively endow FLASH VITERBI and FLASH-BS VITERBI with *fast* execution and *lightweight* memory usage. More importantly, they offer strong *adaptivity* to diverse deployment scenarios by dynamically tuning internal parameters such as parallelism degree P and beam width B. This adaptivity is illustrated by the theoretical time–space complexity comparison shown in Figure 1. The red solid curve captures the performance range of FLASH VITERBI under different degrees of parallelism. By adjusting P, FLASH VITERBI achieves a flexible trade-off between time and space complexity, as indicated by the bright red pentagons, enabling efficient resource utilization. FLASH-BS VITERBI further expands this tunable space by adjusting the beam width B, as shown by the red shaded region. Subsequent experiments show that our method outperforms existing baselines in both time and space dimensions. In addition to the software implementation, our FPGA-based accelerator further demonstrates the *hardware-friendliness* of our design. Our method's non-recursive structure and low memory footprint translate to better hardware utilization, making it suitable for real-time, energy-efficient deployment on edge devices. The key contributions of this paper can be summarized as follows.

- We propose FLASH VITERBI, a Viterbi decoding algorithm that employs non-recursive design, pruning, and parallelization to achieve high time and memory efficiency.
- We introduce FLASH-BS VITERBI, a variant incorporating beam search to enhance adaptability. It dynamically tunes parallelism degree and beam width to balance latency and memory usage under varying resource constraints.
- We demonstrate the hardware-friendliness of our design through an FPGA-based implementation, enabling real-time and energy-efficient deployment on edge devices.

II. RELATED WORK

To enable real-time decoding on resource-constrained edge computing platforms, a variety of improved Viterbi algorithms have been proposed [27].

A. Space-Efficient Viterbi Variants

Tarnas et al. [28] introduced Checkpoint Viterbi, which stores optimal path state information at $\sqrt{T}$ evenly spaced timesteps (checkpoints) and re-executes the algorithm between them. Although this reduces the space complexity to $\mathcal{O}(K\sqrt{T})$, it still scales with sequence length T. Ciaperoni et al. [17] proposed SIEVE, a divide-and-conquer approach based on state-space decomposition. It retains only the midpoint state pairs from the optimal path that lie in the center of state transitions, and uses them to divide decoding. This strategy reduces space complexity to $\mathcal{O}(K)$. Yet, locating such midpoints requires a BFS traversal, rendering SIEVE sensitive to the density of the state-transition graph. In cyclic graphs, the time complexity may increase to $\mathcal{O}(K^2T^2)$. A variant, SIEVE-MiddlePath (SIEVE-Mp) [17], adopts a divide-and-conquer strategy based on sequence length. It maintains a space complexity of $\mathcal{O}(K)$ and consistently achieves $\mathcal{O}(K^2T\log T)$ time complexity regardless of graph density, making it the SOTA approach for space-efficient Viterbi decoding. However, existing space-efficient Viterbi algorithms primarily rely on recursive implementations, which incur considerable overhead from stack memory and scheduling. In addition, the decoding order of subtasks often misaligns with their generation dependencies, hindering parallel execution.

B. Beam Search-based Viterbi Enhancements

Beam search is widely adopted for efficient Viterbi decoding [29]–[34]. Instead of maintaining all possible paths for the full state space (K), it retains only the top B candidate paths (where B is the beam width) at each timestep. This strategy reduces both computational cost and memory footprint with minimal impact on accuracy. Beam search implementations are generally categorized as static or dynamic based on when path pruning occurs. Static beam search either retains the top-B paths after computing all candidates [18] or uses fixed thresholds for pruning [23], [33], [35]. However, such methods must temporarily store scores for all K states at each timestep, leading to limited actual memory savings. In contrast, dynamic beam search [36], [37] performs real-time pruning during the computation of path probabilities, maintaining only B paths at each step. This design significantly improves memory efficiency and makes dynamic beam search more suitable for deployment on resource-constrained platforms. However, its effectiveness depends on efficient data structures that support frequent path updates and replacements during decoding.

C. Hardware-Accelerated Viterbi Decoding

Hardware acceleration has been actively explored to enhance Viterbi decoding [29], [38], [39]. Pani et al. [40] proposed an FPGA-based beam search Viterbi decoder that employs a memory architecture comprising two binary search trees and a max-heap to support path updates. Despite its effectiveness, the design exhibits structural redundancy, leading to high on-chip resource usage and increased power consumption.

III. PRELIMINARIES

In edge-side sequence inference, sensor data is typically represented as a sequence of observed features. Hidden Markov Models (HMMs) offer a probabilistic framework that links these observations to an underlying chain of hidden states. The states evolve according to a first-order Markov process, where each state depends only on its predecessor, while each observation is generated from the emission distribution associated with the current state [41]–[43]. By modeling both the temporal dynamics of the hidden states and their influence on the observations, HMMs effectively capture dependencies in sensor data and allow efficient decoding via algorithms like Viterbi.

A typical HMM consists of the following five key components: 1) An observation set O containing M observable symbols; 2) A hidden state set S comprising K hidden states;

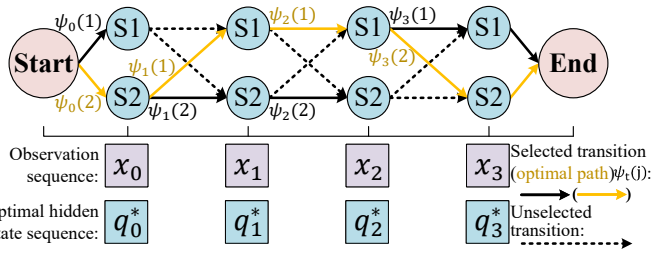

Fig. 2: Trellis diagram of Viterbi decoding, showing the optimal hidden state sequence (q_t^*) for the observation sequence (x_t).

3) An initial state probability vector π, specifying the prior distribution over hidden states at the initial timestep; 4) A state transition probability matrix $\mathcal{A}$, defining the probabilities of transitions between hidden states; 5) An emission probability matrix $\mathcal{B}$, representing the conditional probabilities of observations given hidden states. Hidden states are discrete, while observations may be either discrete (e.g., modeled by categorical distributions) or continuous (e.g., modeled by Gaussian mixture models or deep neural networks [44]). An HMM can be specified by the triplet $(\pi, \mathcal{A}, \mathcal{B})$.

A. Viterbi Algorithm

Given an HMM $\lambda = (\pi, \mathcal{A}, \mathcal{B})$, the decoding problem is defined as follows: For an observation sequence $X = \{x_0, x_1, \ldots, x_{T-1}\}$, drawn from the observation set $O = \{o_1, o_2, \ldots, o_M\}$, determine the optimal hidden state sequence $Q^* = \{q_0^*, q_1^*, \ldots, q_{T-1}^*\}$, denoted as the optimal path, selected from the hidden state set $S = \{s_1, s_2, \ldots, s_K\}$, that satisfies:

$$Q^* = \arg\max_{Q} P(Q \mid X, \lambda),$$

where $Q = \{q_0, q_1, \ldots, q_{T-1}\}$ denotes all possible hidden state sequences of length T. The Viterbi algorithm employs dynamic programming to solve HMM decoding problems. As illustrated in Figure 2, the decoding procedure relies on the recursive computation of two key variables: $\delta_t(j)$,$\psi_t(j)$ [45]:

$$\delta_0(j) = \pi_j \cdot \mathcal{B}_j(x_0), \quad \psi_0(j) = 0 \quad \forall j \in \{1, ..., K\} \quad (t = 0);$$
$$\delta_t(j) = \max_{1 \le i \le K} [\delta_{t-1}(i) \cdot \mathcal{A}_{ij}] \cdot \mathcal{B}_j(x_t),$$
$$\psi_t(j) = \arg\max_{1 \le i \le K} [\delta_{t-1}(i) \cdot \mathcal{A}_{ij}] \quad (t = 1, ..., T-1).$$

Here, $\delta_t(j)$ denotes the highest probability of any state sequence that accounts for the first t observations and ends in state j at timestep t. The variable $\psi_t(j)$ stores the index of the previous state that leads to the maximum $\delta_t(j)$, which corresponds to the selected transition into state j shown by the solid lines in Figure 2, while the dashed lines indicate unselected transitions. After completing the recursion, the optimal final state q_{T-1}^* is selected as the state that maximizes $\delta_{T-1}(j)$. The complete path is then reconstructed by backtracking from q_{T-1}^* through $\psi_t(j)$, as illustrated by the yellow solid path, enabling path-wise attribution [46]–[48].

$$q_{T-1}^* = \arg\max_{1 \le j \le K} \delta_{T-1}(j),$$
$$q_t^* = \psi_{t+1}(q_{t+1}^*) \quad \text{for} \quad (t = T-2, ..., 0).$$

The space complexity of the Viterbi algorithm is $\mathcal{O}(KT)$, as it stores $\delta_t(j)$ and $\psi_t(j)$ at each timestep. Its time complexity is $\mathcal{O}(K^2T)$, due to evaluating all state transitions at each timestep. To prevent numerical overflow, $\delta_t(j)$ and $\psi_t(j)$ are computed using logarithmic summation rather than direct probability multiplications, as shown in lines 5, 8, 14, 15 of Algorithm 2.

IV. Overview of FLASH Viterbi

To efficiently utilize limited resources on edge devices for Viterbi decoding, we propose FLASH Viterbi—a Fast, Lightweight, Adaptable, and Hardware-friendly Viterbi decoding algorithm. It incorporates the following core techniques.

1) Non-recursive Divide-and-conquer for Task Management. FLASH Viterbi pre-generates subtask sets and execution order based on inter-subtask dependencies. It employs a task queue for fine-grained control, reducing space complexity from $\mathcal{O}(KT)$ to $\mathcal{O}(K)$ while avoiding the resource overhead of recursive implementations. Moreover, the resulting execution plan supports parallel decoding of subtasks across threads. We describe the details in Section V-A.

2) Pruning and Parallelization for Decoding Efficiency. During subtask decoding, FLASH Viterbi prunes candidate paths of subsequent subtasks based on the optimal state determined from the current subtask, retaining only transitions from that state. This strategy provides two significant benefits: reducing redundant computations and removing dependencies among parallel subtasks. With parallelism degree P, the overall time complexity is reduced to $\mathcal{O}\left(\frac{K^2T(\log T - \log P)}{P}\right)$. We elaborate our pruning and parallelization strategy in Section V-B.

3) Dynamic Beam Search Integration. By combining dynamic beam search with FLASH Viterbi, we introduce the variant FLASH-BS Viterbi. It retains only the top-B candidate paths during decoding, which reduces space complexity to $\mathcal{O}(B)$ and decouples it from state space size K, FLASH-BS Viterbi offers strong adaptivity to diverse scenarios by configuring beam width. We also design an efficient data structure to support inherent frequent updates. We describe the variant FLASH-BS Viterbi in Section V-C.

4) Hardware-friendly Design. The proposed algorithms eliminate recursion and BFS operations from the execution flow, employ a double-buffered memory scheme in memory structures, and support configurable parallelism, making them well-suited for hardware acceleration. We implement FPGA-based accelerators for both FLASH Viterbi and FLASH-BS Viterbi, achieving high throughput with low resource consumption. The details of our hardware-accelerated implementation are provided in Section VI.

V. FLASH Viterbi Design

In this section, we present the design of FLASH Viterbi and its variant FLASH-BS Viterbi, both of which are optimized for time and memory efficiency. FLASH Viterbi combines a non-recursive divide-and-conquer strategy with pruning and parallelization techniques to enhance decoding performance and parallel adaptability. The variant FLASH-BS Viterbi integrates dynamic beam search and adapts

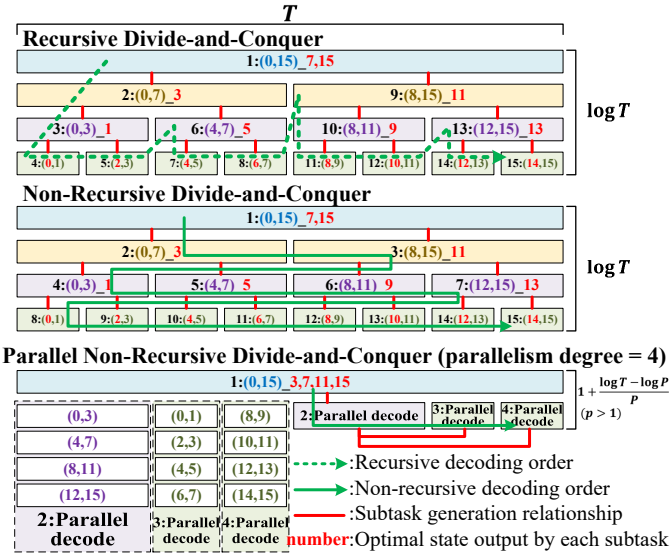

Fig. 3: Comparison of recursive, non-recursive, and parallel non-recursive divide-and-conquer Viterbi decoding. Each rectangle represents a decoding subtask. A subtask labeled $i:(a,b)\_t$ indicates the i-th subtask that decodes the segment (a,b) and outputs the optimal path state at timestep t, i.e., q_t^*.

to available computational resources, making it suitable for deployment on resource-constrained edge devices. In the following discussion, we describe the technical details.

A. Non-recursive Divide-and-conquer Strategy

To address the limitations of prior Viterbi algorithms on resource-constrained devices, FLASH VITERBI adopts a non-recursive divide-and-conquer strategy for task decomposition and decoding. This section discusses the limitations of recursive decoding and presents our alternative strategy that enables fine-grained task management and facilitates efficient parallelization.

1) Limitations of Recursive Viterbi: To reduce the memory footprint of the Viterbi algorithm, a common approach in data systems is to decompose the decoding task into multiple subtasks via a multi-stage divide-and-conquer process, where each subtask retains only limited state information from the optimal path for output. After decoding, subtasks are recursively divided into smaller subtasks to identify the remaining optimal states. Once all subtasks are decoded, the full optimal path is reconstructed by aggregating their outputs. Existing space-efficient Viterbi variants, such as SIEVE-Mp [17], employ recursive strategy to manage the generate and decoding order of subtasks. However, this recursive formulation has two key limitations: First, the recursive structure inherently constrains parallelism across subtasks. As shown in Figure 3, when the sequence length is 16, the red lines indicate the generation relationships between subtasks. Subtasks of the same color within a layer are derived from their parent subtasks in the previous layer and are independent of each other, theoretically enabling parallel decoding. However, the green dashed lines represent the recursive execution order, which conflicts with the generation dependencies. Specifically, subtasks within

Algorithm 1: Non-Recursive Divide-and-Conquer

Input: Sequence length T, parallelism degree P, task queue $\mathcal{Q}_{\text{Task}}$
Output: Decoded sequence

$TotalTask \leftarrow T-P+1$, $TaskCount \leftarrow 0$ ▷ Init counter
Decoding($\mathcal{Q}_{\text{Task}}, 0, T-1, P$) ▷ Decode initial subtask
InitialEnqueue($\mathcal{Q}_{\text{Task}}, 0, T-1, P$) ▷ Enqueue P subtasks
for $i \leftarrow 1$ **to** P ***in parallel*** **do**
  while ***true*** **do**
    while $TaskCount < TotalTask$ ***and*** $\mathcal{Q}_{Task} = \emptyset$ **do**
      WaitForWakeUp() ▷ No task, wait
    if *Dequeue*($\mathcal{Q}_{Task}(m,n)$) $= \emptyset$ **then**
      ProcessEnd() ▷ All tasks done
    $t_{\text{mid}} \leftarrow \lfloor (m+n)/2 \rfloor$
    $TaskCount \leftarrow TaskCount + 1$
    Decoding(m,n) ▷ Subtask decoding
    if $(n-m) > 2$ **then**
      Enqueue($\mathcal{Q}_{\text{Task}}, m, t_{\text{mid}}$)
      Enqueue($\mathcal{Q}_{\text{Task}}, t_{\text{mid}}+1, n$)
    else if $(n-m) = 2$ **then**
      Enqueue($\mathcal{Q}_{\text{Task}}, m, t_{\text{mid}}$)
    WakeUpAllThreads() ▷ Notify other threads

the same layer are executed non-consecutively and far apart, hindering parallel execution at that layer. Secondly, recursive decoding requires auxiliary stack space to preserve context from upper recursion levels. The stack depth scales with the sequence length T, introducing additional memory overhead that undermines the objective of minimizing memory usage.

2) Our Non-recursive Solution: To address the limitations of recursive decoding, FLASH VITERBI pre-generates subtasks based on their generation dependencies. Each subtask is represented as a tuple $(t_{\text{start}}, t_{\text{end}})$, where t_{start} and t_{end} denote the start and end timesteps of a decoding segment, respectively. For a decoding task with sequence length T, the initial tuple is defined as $(0, T-1)$. A task queue is employed to manage execution order through dynamic enqueue and dequeue operations, following two key principles: (i) Inter-layer ordering, where a parent subtask is always processed before its child subtasks to ensure the required state information is available to children (e.g., yellow tasks precede purple tasks in Figure 3); (ii) Intra-layer parallelism, where subtasks within the same layer exhibit no generation dependencies. Such independence allows them to be decoded in any order to support parallel execution. For serial execution, subtasks are processed sequentially according to their start timesteps t_{start}. Figure 3 (middle) illustrates the non-recursive divide-and-conquer strategy employed in FLASH VITERBI. It can be observed that the execution order, denoted by green solid arrows, strictly adheres to the subtask generation hierarchy, without incurring recursive memory overhead.

3) Optimization for Parallel Decoding: To enhance performance, further optimizations are introduced in the parallel implementation of FLASH VITERBI. When the parallelism degree (P) is high, using a binary bisection-based divide-and-conquer strategy—starting from the midpoint—may fail to generate a sufficient number of subtasks during early decomposition, resulting in idle threads and underutilized hardware resources. To address this, our implementation

performs a P-way partition during the initial subtask, dividing the observation sequence into P equal segments generating P subtasks. Subsequent subtasks decoding proceeds via binary bisection. As illustrated in Figure 3 (bottom), for a parallelism degree of $P = 4$, our approach bypasses the yellow-colored bisection phase and directly partitions the observation sequence into four (purple) subtasks after initial subtask, thereby enabling immediate utilization of all four threads. While this initial P-way partitioning introduces a marginal increase in memory overhead compared to pure bisection, the overhead is negligible relative to the memory footprint of later multithreaded processing and does not impact the overall memory usage. This strategy ensures full thread utilization from the outset while maintaining low per-subtask memory overhead.

Algorithm 1 outlines the parallel decoding workflow of FLASH VITERBI. After decoding the initial subtask, P subtasks are enqueued into the task queue and decoded in parallel by P threads. Each thread enqueues new subtasks via binary bisection. Notably, except for the initial task, each decoding subtask outputs the optimal state at its segment's midpoint t_{mid}, as indicated by red numbers in Figure 3. When the segment length reduces to 3, the right subtask shares the same midpoint as its parent, obviating additional enqueuing.

4) Parallelism Analysis under Non-recursive Divide-and-Conquer: In Figure 3, the width of each rectangle denotes the sequence length of a subtask, while the area approximates its decoding time. Using the full decoding time of a length-$T = 16$ sequence (i.e., the blue rectangle) as the decoding cycles, we quantify the temporal consumption across different strategies. The sequential divide-and-conquer Viterbi requires $\log T$ decoding cycles (e.g., 4 cycles for $T = 16$) to complete execution. In contrast, the multithreaded variant with a parallelism degree $P = 4$ completes the decoding process in only $1 + \frac{\log T - \log P}{P}$ cycles (e.g., 1.5 cycles for $T = 16$), illustrating significant parallel acceleration.

Through this optimization, FLASH VITERBI eliminates the inherent limitations of recursive decomposition while establishing the foundation for parallel execution of subtasks.

B. Pruning and Parallelization Strategy

In this section, we detail our pruning and parallelization strategy designed to optimize the computational process of subtask decoding. Our goal is to reduce redundant computations and eliminate dependencies among intra-layer subtasks, thereby enabling parallel execution. We also analyze time and space complexity and provide the proof of correctness of the strategy.

1) Subtask Decoding and Dependency-Induced Challenges: To illustrate the necessity of the pruning and parallelization strategy, we begin by describing the subtask decoding procedure in FLASH VITERBI and analyze the computational dependencies across subtasks. Algorithm 2 presents the subtask decoding process of FLASH VITERBI under binary bisection before applying the pruning and parallelization strategy. Figure 4 illustrates this process for a sequence length of $T = 8$ and a state space size of $K = 4$. As shown in Step 2 of Figure 4, before the backtracking phase, the standard Viterbi algorithm retains all state transitions along each candidate path throughout dynamic programming (depicted by dashed lines). In contrast, FLASH VITERBI retains only the transitions between the division point timestep t_{mid} (i.e., the midpoint for binary bisection) and the terminal timestep (depicted by red solid lines). This selective retention reduces the space complexity from $\mathcal{O}(KT)$ to $\mathcal{O}(K)$. This optimization is realized through iterative updates of three key arrays:

- *OptProb*: Stores the maximum path probability for each state at the current timestep.
- *PreState*: Stores the state index at the previous timestep for each maximum path.
- *MidState*: Stores the state index at the division point timesteps of each maximum path, as shown by the red lines in Figure 4.

As shown in Step 1 of Figure 4 (Dynamic Programming), at timestep $t = 5$, the *MidState* array is updated by combining the *PreState* information at $t = 5$ (brown dashed lines) with the *MidState* values at $t = 4$ (purple solid lines), resulting in the updated *MidState* at $t = 5$ (red solid lines). This iterative update continues throughout the dynamic programming phase, capturing only the transitions at division points and the terminal timestep. Notably, the updates to *PreState* and *MidState* are initiated only after the division point timestep (i.e., $t_{\text{mid}} + 1$ under binary bisection). During backtracking, the optimal path (depicted by the green dashed line) is identified by comparing the *OptProb* values across all K states. Since FLASH VITERBI retains state transitions only at division points and the terminal timestep, it can directly infer optimal states at these specific timesteps (i.e., red-highlighted states in the decoded sequence in Figure 4). The remaining optimal states are recovered through decoding of subsequent subtasks.

During the divide-and-conquer phase (Step 3 in Figure 4), the decoding task $(0, 7)$ is partitioned into two subtasks: $(0, 3)$ and $(4, 7)$. Although these subtasks are structurally independent, the initialization of *OptProb* at timestep 4 in subtask $(4, 7)$ relies on the *OptProb* values at timestep 3 from subtask $(0, 3)$. This cross-subtask dependency—evident in line 8 of Algorithm 2—inhibits their parallel execution.

2) Pruning for Parallel Decoding: To eliminate cross-subtask dependencies and enable parallel decoding, we propose the following pruning strategy: when solving a subtask starting at the division point timestep of the previous divided layer, only the transition paths originating from the optimal state at that timestep are retained, and their corresponding *OptProb* values are re-initialized. This approach is illustrated in Step 3 of Figure 4, highlighted by the two blue dashed boxes.

In the right dashed box, the dashed lines depict all transition paths that need to be traversed at timestep t=3 prior to pruning. Among them, black and green lines indicate maximum-probability transitions for each state. In contrast, the blue dashed lines in the left box depict the pruned transition paths, where only transitions leading to the optimal state S_2 at t=3 are retained. The number of transition computations eliminated through pruning is given by $(K-1)K\sum_{i=1}^{\log T-1}\frac{T}{2^i}$, which matches the overall time complexity of the full decoding

Algorithm 2: Subtask Decoding (Before Pruning)

Input: Initial state $\boldsymbol{\pi}$, transition matrix $\mathcal{A}$, emission matrix $\mathcal{B}$, sequence X, length T, start index m, end index n
Output: Optimal states $q^*_{t_{\text{mid}}}$ and q^*_{T-1}

1 $t_{\text{mid}} \leftarrow \lfloor (m+n)/2 \rfloor$ ▷ Midpoint of segment
2 /* Initialization phase */
3 **if** $m = 0$ **then**
4 **for** $i \leftarrow 1$ **to** K **do**
5 $OptProb[i] \leftarrow \log(\pi_i) + \log(\mathcal{B}_{i,x_0})$ ▷ Base case init
6 **else**
7 **for** $i \leftarrow 1$ **to** K **do**
8 $OptProb[i] \leftarrow \max_k(OptProb[k] + \log(\mathcal{A}_{k,i}) + \log(\mathcal{B}_{i,x_m}))$
9 **for** $i \leftarrow 1$ **to** K **do**
10 $PreState[i] \leftarrow -1$, $MidState[i] \leftarrow -1$ ▷ Reset trace
11 /* Dynamic programming loop */
12 **for** $t \leftarrow m+1$ **to** n **do**
13 **for** $i \leftarrow 1$ **to** K **do**
14 $OptProb[i] \leftarrow \max_k(OptProb[k] + \log(\mathcal{A}_{k,i}) + \log(\mathcal{B}_{i,x_t}))$
15 $PreState[i] \leftarrow \arg\max_k(OptProb[k] + \log(\mathcal{A}_{k,i}) + \log(\mathcal{B}_{i,x_t}))$
16 **if** $t = t_{mid} + 1$ **then**
17 $MidState[i] \leftarrow PreState[i]$
18 **else if** $t > t_{mid} + 1$ **then**
19 $MidState[i] \leftarrow MidState[PreState[i]]$
20 /* Backtracking phase */
21 **if** $t_{mid} = \lfloor (T-1)/2 \rfloor$ **then**
22 $q^*_{T-1} \leftarrow \arg\max_k(OptProb[k])$
23 $q^*_{t_{\text{mid}}} \leftarrow MidState[q^*_{T-1}]$
24 **else**
25 $q^*_{t_{\text{mid}}} \leftarrow MidState[q^*_n]$

process, i.e., $\mathcal{O}(K^2 T \log T)$. This highlights the pruning strategy's effectiveness in eliminating redundant computations. The rationale behind this strategy is that subsequent subtasks primarily serve to reconstruct the optimal path. Thus, it is sufficient to preserve only those transitions along the optimal path from the original decoding task and ensure their corresponding probabilities remain maximal within each subtask. Transitions associated with non-optimal candidate paths do not need to maintain consistency with the original global decoding. Taking Step 3 of Figure 4 as an example, the maximum-probability paths for each state in the right dashed box exhibit probabilities no lower than those of the retained paths in the left box after pruning. Notably, the optimal path—originally the highest-probability path in the right box—remains optimal in the pruned subset shown in the left box. Therefore, this pruning strategy does not alter the final decoding result. To simplify computations, we set the accumulated path probability of the optimal state at the initial timestep to 1 (e.g., state S_2 at timestep $t = 3$ in Step 3 of Figure 4). By applying logarithmic transformation, the dependency on the previous timestep's *OptProb* can be eliminated in the path probability equation, enabling parallel subtask execution. Accordingly, line 8 of Algorithm 2 is modified to:

$$OptProb[i] = \log(\mathcal{A}_{q^*_{m-1},i}) + \log(\mathcal{B}_{i,x_m}).$$

With a parallelism degree of P, this optimization reduces the time complexity from $\mathcal{O}(K^2 T \log T)$ to

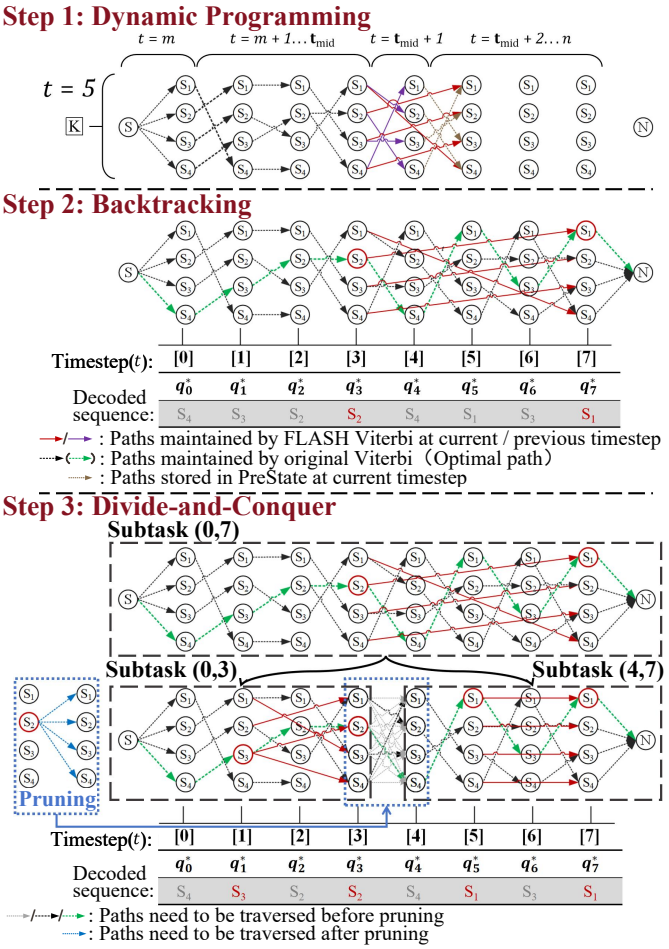

Fig. 4: FLASH VITERBI subtask decoding process.

$\mathcal{O}\left(K^2 T \frac{(\log T - \log P)}{P}\right)$, and the space complexity to $\mathcal{O}(PK)$.

Additionally, the initial probability computation for a subtask starting at a division point is structurally analogous to that of a subtask starting at timestep 0. In both cases, decoding is initiated from a single state (e.g., state S_1 for subtask $(0,3)$ and state S_2 for subtask $(4,7)$ in Figure 4). This structural similarity enables the use of a unified hardware architecture for all subtasks during edge-side deployment, thereby improving hardware efficiency and resource utilization.

3) Correctness of Pruning and Parallelization Strategy: The correctness of the divide-and-conquer formulation for the Viterbi algorithm has been proven in [17]. This section therefore focuses on proving the correctness of the proposed pruning and parallelization strategy. We first formally define the valid path set and the optimal path as follows.

Definition 1 (Valid path set)**.** *Given a divide-and-conquer subtask (m,n) with a state space of size K and $m \neq 0$, the valid path set $\mathcal{Y}$ is defined as $\mathcal{Y} = \{Q_1, Q_2, \ldots, Q_y\}$, representing all feasible paths from timestep $m-1$ to n, with corresponding path probabilities $\mathcal{P} = \{p_1, p_2, \ldots, p_y\}$. The cardinality y satisfies $1 \leq y \leq K^{n-m+2}$, depending on the sparsity of the transition matrix $\mathcal{A}$.*

Definition 2 (Optimal path)**.** *The optimal path Q^* un-*

der the divide-and-conquer strategy is defined as $Q^* = [q^*_{m-1}, q^*_m, \dots, q^*_n]$*, which maximizes the path probability, i.e.,* $p^* = \max(\mathcal{P})$.

For clarity, we define the following two variants of the pruning strategy used in the proofs.

Definition 3 (Retained-pruning variant)**.** *The retained-pruning variant restricts the path set to those sharing the same* $(m-1)$*-th state* q_{m-1} *as the optimal path* Q^**. It defines a subset* $\mathcal{Y}' = \{Q \in \mathcal{Y} \mid q_{m-1} = q^*_{m-1}\}$ *with path probabilities* $\mathcal{P}' = \{p'_1, p'_2, \dots\}$*. The optimal path is* $Q' = [q'_{m-1}, q'_m, \dots, q'_n]$*, which maximizes the path probability, i.e.,* $p' = \max(\mathcal{P}')$.

Definition 4 (Complete-pruning variant)**.** *The complete-pruning variant extends the retained-pruning variant by assuming* $p^*_{m-1} = 1$*. Decoding is performed using* $\mathcal{Y}'' = \mathcal{Y}'$ *with path probabilities* $\mathcal{P}'' = \{p''_1, p''_2, \dots\}$*. The optimal path is* $Q'' = [q''_{m-1}, q''_m, \dots, q''_n]$*, which maximizes the path probability, i.e.,* $p'' = \max(\mathcal{P}'')$.

Theorem 1. *The optimal path* Q^* *belongs to the retained-pruning path set, i.e.,* $Q^* \in \mathcal{Y}'$.

Proof. From Definition 3, we have $\mathcal{Y}' = \{Q \in \mathcal{Y} \mid q_{m-1} = q^*_{m-1}\}$. Since $Q^* = [q^*_{m-1}, q^*_m, \dots, q^*_n]$ satisfies $q_{m-1} = q^*_{m-1}$, it follows that $Q^* \in \mathcal{Y}'$. Moreover, $p^* \in \mathcal{P}'$. □

Theorem 2. *The optimal path from the retained-pruning variant equals the one from divide-and-conquer, i.e.,* $Q' = Q^*$.

Proof. Since $\mathcal{Y}' \subseteq \mathcal{Y}$ and $p' = \max(\mathcal{P}')$, and from Theorem 1, we have $p^* \in \mathcal{P}'$, then $p' = p^*$. Thus, the optimal paths are the same: $Q' = Q^*$. □

Theorem 3. *The optimal path from complete-pruning variant equals the one from divide-and-conquer, i.e.,* $Q'' = Q^*$.

Proof. From Definitions 3 and 4, the update rule in retained-pruning at time m is $\mathit{OptProb}'[i] = \log(\mathcal{A}_{q^*_{m-1},i}) + \log(\mathcal{B}_{i,x_m}) + \log(p^*_{m-1})$, while in complete-pruning it becomes $\mathit{OptProb}''[i] = \log(\mathcal{A}_{q^*_{m-1},i}) + \log(\mathcal{B}_{i,x_m})$. Since $\log(p^*_{m-1})$ is constant w.r.t. i, it does not affect the $\arg\max$ operation. Therefore, $\mathit{MidState}'' = \mathit{MidState}'$. From Algorithm 2, the optimal states are maintained based on *MidState*. Hence $Q'' = Q'$, and by Theorem 2, $Q'' = Q^*$. □

These theorems together confirm that the pruning and parallelization strategy preserves the correctness of the divide-and-conquer approach, yielding the same optimal path. Overall, the pruning and parallelization strategy enhances HMM decoding in three key aspects: 1) Reducing redundant computations without sacrificing memory efficiency; 2) Removing inter-subtask dependencies to enable parallel decoding; 3) Unifying subtask computational structures, enhancing resource utilization.

C. FLASH VITERBI *with Beam Search*

To address memory constraints in standard FLASH VITERBI and further enhance its adaptability, we propose FLASH-BS VITERBI—a dynamic beam search variant. This section

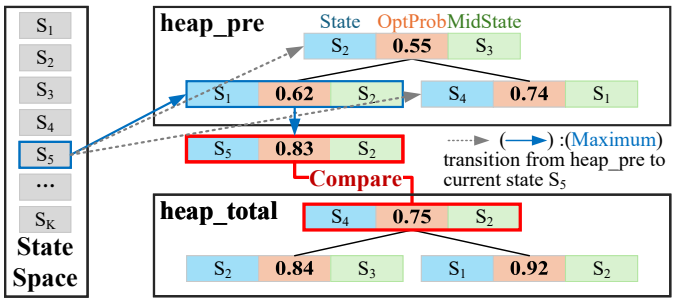

Fig. 5: Efficient data structure for dynamic beam search.

first introduces the key idea, then details an efficient heap-based mechanism to reduce computational overhead, and finally analyzes the resulting complexity.

1) Static vs. Dynamic Beam Search: Prior works have integrated beam search with divide-and-conquer Viterbi algorithm. However, these methods typically adopt static beam selection, wherein the path probabilities for all candidate states are computed and stored before selecting the top-B candidates. This static beam search suffers from limited memory optimization because it still requires storing the state information of the remaining $K - B$ states before discarding them. In contrast, FLASH-BS VITERBI employs dynamic beam search, which incrementally maintains only the top-B states during the process of calculating path probabilities. In this way, FLASH-BS VITERBI eliminates the need to store full intermediate results for the remaining $K - B$ states.

2) Efficient Heap-Based Candidate Maintenance: To mitigate the overhead caused by frequent comparisons and selective replacements in dynamic candidate maintenance, we introduce an efficient data structure based on two min-heaps for optimal state management, as illustrated in Figure 5.

The algorithm employs two min-heaps: (i) *heap_pre*, which retains the top B candidate states from the previous timestep, and (ii) *heap_total*, which incrementally updates the top B candidates for the current timestep. Each heap maintains three core components: (i) *State*, which records the indices of the top-B candidate states; (ii) *OptProb*, which stores their corresponding path probabilities and serves as the heap's sorting key; (iii) *MidState*, which tracks the state indices at division points timestep along the maximum paths, consistent with the FLASH VITERBI definition. To avoid data copying overhead, two physically separate heap buffers are allocated and managed using a double-buffering scheme. The roles of *heap_pre* and *heap_total* alternate across timesteps, enabling efficient in-place updates and streamlined memory access.

The heap update procedure is depicted in Figure 5. During decoding, FLASH-BS VITERBI evaluates only transitions from the B candidate states in *heap_pre* to the current state, generating a corresponding heap element for each. The update logic for *heap_total* proceeds as follows: 1) If fewer than $B-1$ elements are stored, the new element is inserted directly. 2) If exactly $B-1$ elements are present, the new element is inserted and the heap is heapified. 3) If B elements are already stored, the new element is compared against the current heap root (i.e.,

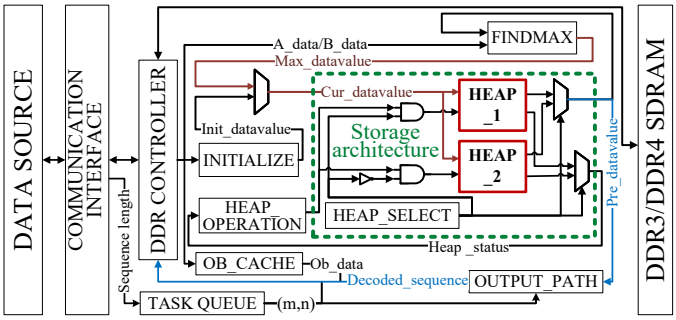

Fig. 6: Architecture of the FLASH-BS VITERBI accelerator.

the candidate with the lowest *OptProb*). It is inserted only if its *OptProb* is higher, thereby maintaining the top-B candidates.

3) Analysis of Efficiency and Adaptability: Leveraging the efficient memory design in dynamic beam search, FLASH-BS VITERBI significantly reduces memory consumption and enhancing adaptability. Given a parallelism degree P and beam width B, the time complexity is reduced to $\mathcal{O}\left(BKT \cdot \frac{(\log T - \log P)}{P}\right)$, as each state only computed transitions from the top B candidate paths at the previous timestep. Correspondingly, the space complexity is reduced to $\mathcal{O}(PB)$, since only B paths per parallel unit need to be maintained. The two tunable parameters—parallelism degree P and beam width B—demonstrate the multidimensional adaptability of FLASH-BS VITERBI, enabling fine-grained trade-offs between decoding latency and memory consumption.

VI. HARDWARE IMPLEMENTATION

In this section, we present the hardware-accelerated implementation of FLASH-BS VITERBI. Compared with existing space-efficient Viterbi algorithms, FLASH VITERBI and FLASH-BS VITERBI exhibit hardware-friendly properties, including non-recursive structures, the elimination of BFS traversal, and double-buffered memory schemes. These features enable efficient parallelization and low-overhead deployment on edge devices. The following describes the accelerator architecture and its FPGA implementation.

A. FPGA-Based Accelerator Architecture

Figure 6 shows the FLASH-BS VITERBI accelerator architecture. After loading HMM data into DDR memory, decoding is initiated on the FPGA. A FIFO-based *TASK QUEUE* generates subtask tuples based on the sequence length and sequentially dispatches them to the DDR CONTROLLER to fetch the corresponding observation segments from DDR memory. Each segment is cached in the *OB_CACHE* in decoding order prior to execution. During subtask initialization, the relevant data is fetched from DDR memory to perform the initial path probability computation (i.e., *Init_datavalue* in Figure 6). For subsequent timesteps, the FINDMAX unit performs dynamic programming using HMM data from DDR memory and intermediate results (*Pre_datavalue*) stored in the storage units. The updated results are written back, as shown by the red paths in Figure 6. In the final backtracking phase, the storage units write the traced state sequence to the *OUTPUT_PATH*. After all subtasks are processed, the complete *Decoded_sequence* is written to DDR memory, finalizing the decoding process (blue paths in Figure 6).

B. Memory Optimization

To eliminate the substantial time and resource overhead caused by data transfer operations, the storage architecture employs a double-buffered memory scheme. In the FLASH-BS VITERBI accelerator, two BRAM-based min-heap structures, *HEAP_1* and *HEAP_2*, dynamically alternate roles between *Heap_total* and *Heap_pre*, as described in Section V-C. These heap structures perform search and update operations based on instructions issued by the HEAP_OPERATION module.

C. Parallelization and Pipelining Optimization

To accelerate decoding, our design applies optimizations to both data access and value computation. Specifically, multiple data groups are fetched from DDR memory in a single clock cycle and concurrently processed by the FINDMAX module. Additionally, the design employs pipelining to overlap memory access with computation in the FINDMAX module, thereby hiding memory latency and improving overall throughput. While beam search introduces irregular memory access patterns that may limit data parallelism, it significantly reduces on-chip memory usage, enabling more efficient deployment on resource-constrained hardware.

VII. EXPERIMENTS

In this section, we present the performance of the FLASH VITERBI and FLASH-BS VITERBI algorithms under various HMM scenarios, compared with several baseline methods.

A. Experimental Setting

Data. We generate various Erdős–Rényi transition graphs to simulate different HMM scenarios. Specifically, we vary the edge probability p (i.e., the likelihood of transition between states), the state space size K, the observation sequence length T, and assess each algorithm's decoding time and memory usage under different conditions. Additionally, to evaluate the impact of beam width on memory usage, decoding time, and recognition accuracy, we construct a forced-alignment dataset with K = 3965 and T = 256. The dataset is generated using the HTK toolkit [49] to force-align speech transcriptions [50] from the TIMIT corpus [51].

Platform. All CPU experiments used 2×16-core Xeon 6226R CPUs (2.90 GHz, 512 GB RAM), and the FPGA accelerator was implemented in Verilog on a Xilinx XCZU7EV at 200 MHz.

Baselines. On the CPU platform, we compare the FLASH variants (FLASH VITERBI and FLASH-BS VITERBI) against five baselines: (i) Vanilla Viterbi [45]: the standard Viterbi algorithm; (ii) Checkpoint Viterbi [28]: stores intermediate decoding states every $\sqrt{T}$ steps to reduce memory usage; (iii) SIEVE-Mp [17]: applies a recursive divide-and-conquer strategy to reduce memory usage; (iv) SIEVE-BS [18]: a SIEVE variant with static beam search; (v) SIEVE-BS-Mp [18]: a SIEVE-Mp variant with static beam search. All baselines were

TABLE I: Overall comparison of time and space for FLASH variants against baseline Viterbi algorithms, for both C and Python implementations. Speedups and memory ratios of our proposed algorithms over other algorithms are also reported.

Class	Algorithm	Impl.	Decoding time (s)			Speedup			Memory usage ($\times 10^3$ B)			Memory ratio			Py/C ratio	
			Seq.	Par.7	Par.16	Seq.	Par.7	Par.16	Seq.	Par.7	Par.16	Seq.	Par.7	Par.16	Time	Mem
Beam search integration (B=128)	SIEVE-BS	C	49.4	–	–	3.5	14.3	18.3	2899.4	–	–	901.5	132.8	58.2	–	–
		Py	208.5	–	–	14.7	60.2	77.4	22928.1	–	–	7129.4	1049.8	460.6	4.2	7.9
	SIEVE-BS-Mp	C	38.0	–	–	2.7	11.0	14.1	2503.3	–	–	778.4	114.6	50.3	–	–
		Py	183.1	–	–	12.9	52.9	68.0	20808.8	–	–	6470.4	952.8	418.0	4.8	8.3
	FLASH-BS	C	14.2	3.5	2.7	1.0	1.0	1.0	3.2	21.8	49.8	1.0	1.0	1.0	–	–
Without beam search	Vanilla Viterbi	C	104.4	–	–	0.3	1.1	1.5	8120.3	–	–	127.8	18.3	8.0	–	–
		Py	151.7	–	–	0.4	1.6	2.1	32549.3	–	–	512.1	73.3	32.1	1.5	4.0
	Checkpoint Viterbi	C	216.0	–	–	0.6	2.3	3.0	824.8	–	–	13.0	1.9	0.8	–	–
		Py	374.5	–	–	1.0	4.0	5.2	3271.8	–	–	51.5	7.4	3.2	1.7	4.0
	SIEVE-Mp	C	672.6	–	–	1.7	7.2	9.4	775.0	–	–	12.2	1.7	0.8	–	–
		Py	1679.8	–	–	4.4	17.9	23.4	127554.1	–	–	2006.8	287.1	125.6	2.5	164.6
	FLASH	C	385.9	93.6	71.7	1.0	1.0	1.0	63.6	444.2	1015.3	1.0	1.0	1.0	–	–

originally implemented in Python. The FLASH variants were implemented in C as it offers higher efficiency and is better suited for resource-constrained edge deployment. To ensure fair comparison, we re-implemented all baselines in C and using these C versions for comparison in all experiments unless specified. On the FPGA platform, we compare our accelerator with Reduced-Memory Viterbi, a SOTA beam search–based implementation [40].

Parameter settings. To evaluate the impact of individual parameters, we use default settings unless otherwise specified: observation space size $|O| = 50$, edge probability $p = 0.253$, state space size $K = 512$, and sequence length $T = 512$. The default settings are chosen with reference to [17] and are commonly adopted in evaluating Viterbi algorithms. In beam width experiments using the forced-alignment dataset, the value of p is inherently determined by the dataset and is unaffected by our default setting. Emission probabilities are randomized. Beam width B is initially set to K, with all candidate paths retained by default. For edge probability analysis, p is varied from 0.05 to 1 in multiplicative steps of 1.5. The state space size K and sequence length T are varied from 32 to 2048. The beam width B is decreased from 1024 to 32.

B. Overall Comparison

In this section, we evaluate the overall performance of FLASH variants against baseline algorithms in both their original Python and optimized C implementations, using the forced-alignment dataset. We report decoding time (in seconds) and memory usage (in bytes). Decoding time is measured from the input of the HMM model to the completion of the optimal path, while memory usage includes all relevant data structures used during decoding. We also report the performance gains of the C-optimized baselines over their original Python versions in the **Py/C ratio** columns. We evaluate algorithms in two categories: with beam search integration and without beam search. For each category, we compare FLASH variants with existing baselines under different levels of parallelism.

1) With Beam Search Integration: Table I reports the results of the comparison. FLASH-BS VITERBI consistently outperforms both SIEVE-BS and SIEVE-BS-Mp in decoding time and memory usage. In sequential execution, it achieves $\mathbf{3.5}\times$ and $\mathbf{2.7}\times$ speedups over their C-optimized implementations, while reducing memory consumption by $\mathbf{901.5}\times$ and $\mathbf{778.4}\times$, respectively. For the original Python implementations, these advantages become even more significant, with speedups increasing to $\mathbf{14.7}\times$ and $\mathbf{12.9}\times$, and memory reductions reaching $\mathbf{7129.4}\times$ and $\mathbf{6470.4}\times$. The decoding speedup primarily arises from three design optimizations: (i) a non-recursive divide-and-conquer strategy that eliminates recursive calls and BFS traversal overhead; (ii) a double-buffered memory scheme that reduces data transfer latency during dynamic programming; and (iii) an integrated pruning-parallelism mechanism that eliminates redundant path computations across subtasks. In terms of memory usage, the improvement mainly stems from the efficient memory structure of dynamic beam search. Unlike SIEVE-BS and SIEVE-BS-Mp, which store the entire K state space, our method maintains only B candidate states at each timestep, resulting in substantial memory savings.

The advantages of FLASH-BS VITERBI become even more pronounced under parallel execution. At a parallelism level of 16, it achieves speedups of $\mathbf{18.3}\times$ and $\mathbf{14.1}\times$ over the C-optimized implementations of SIEVE-BS and SIEVE-BS-Mp, while reducing memory consumption by $\mathbf{58.2}\times$ and $\mathbf{50.3}\times$, respectively. These results indicate that the performance of our algorithm can be further enhanced by leveraging parallelism.

2) Without Beam Search Integration: In the non–beam search category under sequential execution, FLASH VITERBI shows slower speed than the fastest baseline, Vanilla Viterbi. However, it achieves substantially lower memory consumption, reducing usage by $\mathbf{512.1}\times$ in the original Python implementation and by $\mathbf{127.8}\times$ in the C-optimized version, thereby leading to a markedly improved overall time–space trade-off. By tuning the degree of parallelism to balance decoding time and memory usage, FLASH VITERBI achieves a more favorable trade-off. As shown in Table I, at a parallelism level of 7, it outperforms all baselines in both decoding time and memory usage, demonstrating superior overall resource efficiency.

C. Effect of HMM Conditions

In this section, we evaluate the scalability and efficiency of various decoding algorithms under diverse HMM settings,

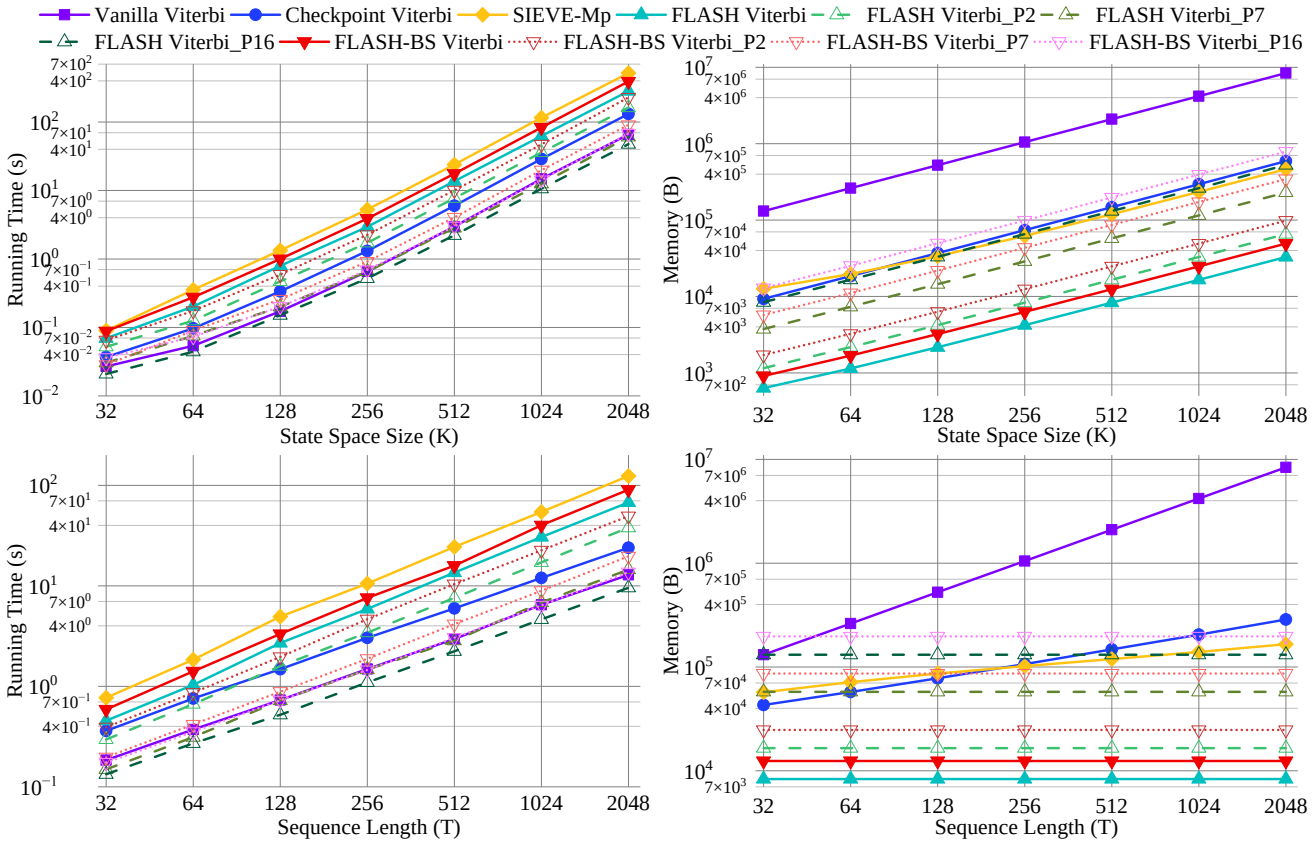

Fig. 7: Decoding time and memory usage vs. state space size and sequence length. FLASH VITERBI_Pn and FLASH-BS VITERBI_Pn represent the algorithms executed under parallelism degree n.

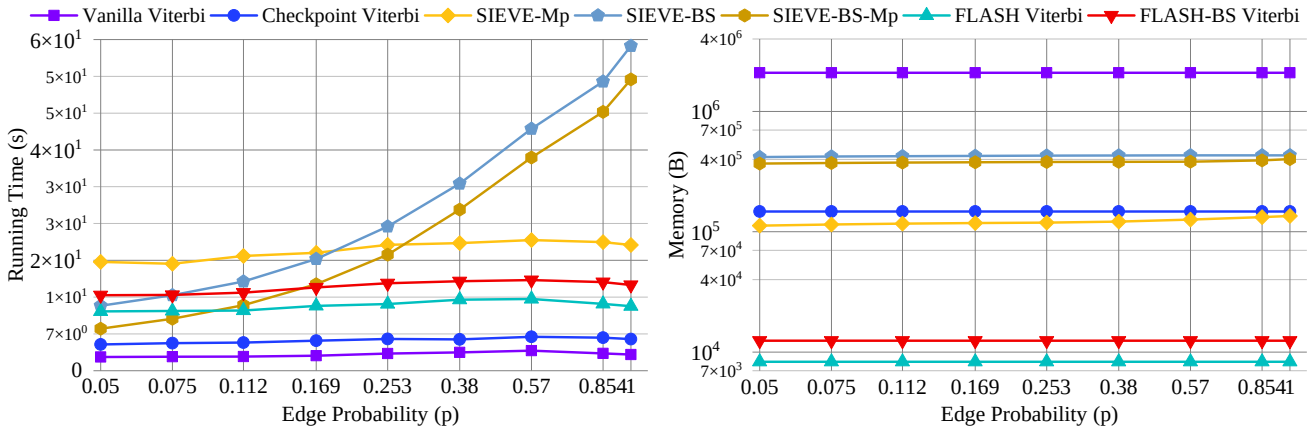

Fig. 8: Decoding time and memory usage vs. edge probability.

focusing on structural scale—determined by sequence length and state space size—and transition density, defined by edge probabilities. We specifically compare the sequential performance of FLASH variants against baseline algorithms.

1) Performance Under Varying Sequence Lengths and State Space Sizes: Figure 7 presents decoding performance under varying sequence lengths T and state space sizes K. Although FLASH VITERBI and SIEVE-Mp share the same theoretical complexity of $\mathcal{O}(K^2 T \log T)$, FLASH VITERBI achieves lower actual runtime due to its non-recursive design and double-buffered memory scheme. These optimizations eliminate recursion and BFS overhead in divide-and-conquer steps, while reducing data transfer latency during dynamic programming. Compared to FLASH VITERBI, FLASH-BS VITERBI shows a slight increase in runtime, primarily due to heap maintenance overhead. Regarding memory usage, FLASH VITERBI and FLASH-BS VITERBI consistently achieve the lowest footprint across varying T and K, by storing only the state transitions between the division point and the terminal timestep. This design decouples memory usage from sequence length T. Additionally, the non-recursive formulation and double-buffered memory scheme further reduce memory usage, enabling our

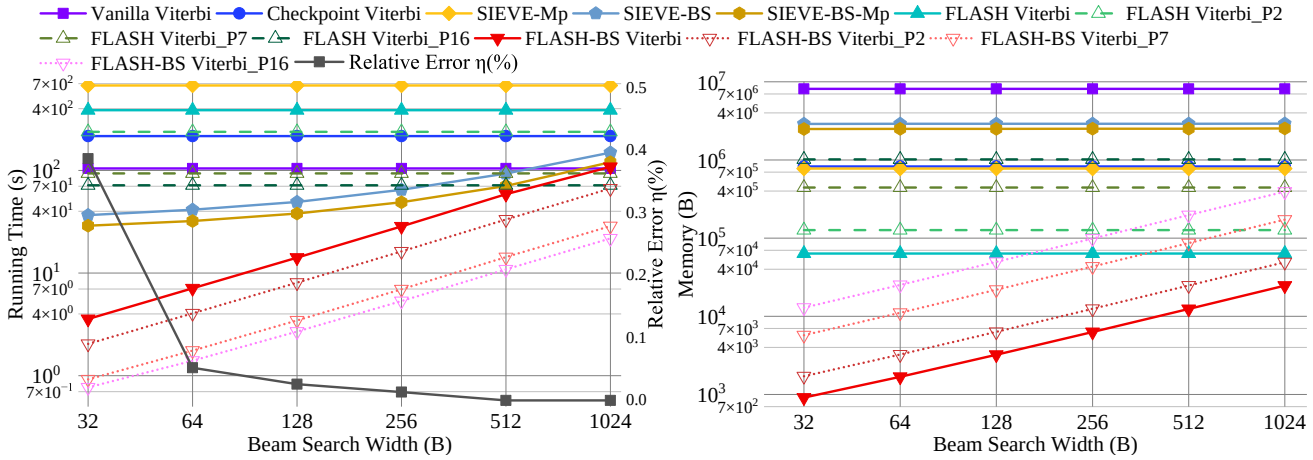

Fig. 9: Decoding time, memory usage, and relative error vs. beam search width. Relative error reflects decoding accuracy loss.

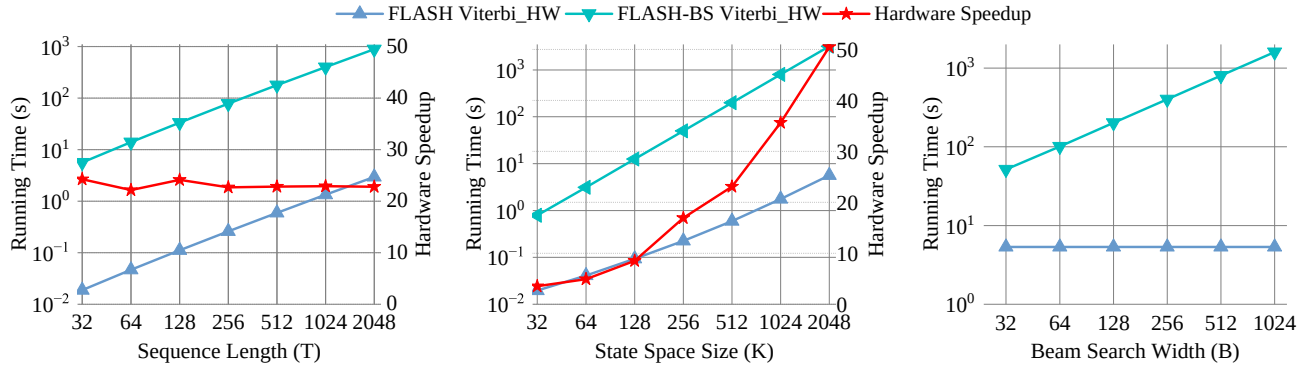

Fig. 10: Decoding time and hardware speedup of FLASH VITERBI and FLASH-BS VITERBI accelerators.

algorithms to outperform SIEVE-Mp, which is the most theoretically memory-efficient baseline to date.

2) Performance Under Varying Edge Probabilities: Figure 8 presents decoding time and memory usage across varying edge probabilities p. Compared to SIEVE-BS and SIEVE-BS-Mp, FLASH variants are less sensitive to changes in transition density. This contrast stems from their underlying HMM representations: FLASH variants use a standard state-matrix-based approach, while SIEVE-BS and SIEVE-BS-Mp adopt a token-passing formulation. Although token-passing avoids traversing unreachable states, it incurs additional overhead due to multiple per-state comparisons required to identify the most probable incoming transitions. As a result, SIEVE-BS-Mp achieves faster decoding at extremely sparse settings ($p \leq 0.075$), but its runtime increases rapidly as p grows. In contrast, FLASH variants maintain stable performance across all values of p, demonstrating robustness to graph sparsity.

D. Impact of Individual Parameters

In this section, we evaluate the decoding performance of FLASH variants by adjusting two internal parameters: parallelism degree n and beam width B.

1) Impact of Parallelism Degree on Runtime and Memory Usage: Figure 7 illustrates the performance of FLASH VITERBI and FLASH-BS VITERBI under different parallelism levels ($n = 2, 7, 16$). By tuning the parallelism degree n, FLASH VITERBI outperforms existing baselines in both decoding time and memory usage. For instance, at $T = 512$ and $K = 256$–2048, FLASH VITERBI_P7 achieves lower decoding latency than the fastest baseline (Vanilla Viterbi) while consuming even less memory than the most memory-efficient baseline (SIEVE-Mp). This improvement stems from its parallel execution strategy: decoding subtasks are evenly distributed across threads, and each subtask stores the state information at the division point during dynamic programming. By eliminating the backtracking table lookup required by Vanilla Viterbi, this design significantly reduces decoding latency. Although memory consumption scales linearly with the parallelism degree n, the inherently low footprint of FLASH VITERBI allows its parallel variants to remain more memory-efficient than existing baselines. More importantly, unlike fixed-cost baselines, FLASH VITERBI can fully utilize memory and compute resources to meet both real-time and deployment requirements. For example, FLASH VITERBI_P16 achieves the lowest latency among all baselines at all times, with only a slight increase in memory usage. These results highlight the efficiency and adaptability of FLASH VITERBI.

2) Trade-offs Introduced by Beam Width in Accuracy and Efficiency: To assess the impact of beam search on decoding accuracy, we follow [18] and report the relative error in log-likelihood, defined as $\eta = \frac{|\ell_{\text{OPT}} - \ell|}{\ell_{\text{OPT}}}$, where ℓ_{OPT} and ℓ denote the log-likelihoods of the optimal path and the path obtained via beam search, respectively. Figure 9 illustrates the performance

TABLE II: Comparison of resource utilization and power consumption between **FLASH-BS VITERBI** (FLASH-BS) and **Reduced-Memory Viterbi** (RM-Viterbi). The "Decoding Unit" refers only to the decoding logic; "Entire Decoder" includes the full accelerator with DDR interface modules.

Width	Algorithm	Decoding Unit			Entire Decoder			Power (W)
		BRAM	DSP	LUT+FF	BRAM	DSP	LUT+FF	
32K	FLASH-BS	173	0	13115	198.5	3	42445	1.835
	RM-Viterbi	757	30	13927	764	30	35136	2.152
512	FLASH-BS	7	0	13127	32.5	3	41954	1.737
	RM-Viterbi	65	30	13927	72	30	34899	1.740

trends and corresponding relative decoding error under different beam widths. In terms of accuracy, when the beam width $B \geq 64$, the relative decoding error remains below 0.05%. However, reducing B to 32 leads to a sharp error increase to 0.39%, due to critical states being excluded from the narrower beam width. When the structural scale of a decoding task is held constant (i.e., fixed K and T), the impact of beam width on accuracy depends on the statistical characteristics of the task data, particularly the distribution of transition probabilities in $\mathcal{A}$, $\mathcal{B}$, and π. In cases where transitions are dense or probabilities are uniformly distributed, optimal path states are more likely to fall outside the local top B at certain timesteps, resulting in accuracy degradation. In contrast, the impact of beam width on the runtime and memory usage of FLASH-BS VITERBI is independent of the data distribution, depending solely on the state space size K and the chosen beam width B. Both theoretical analysis and experimental results confirm that the runtime and memory usage are reduced to B/K of the original values. Overall, the performance–cost trade-off of beam search is influenced by domain-specific data distribution. We recommend analyzing this distribution before using small beam widths, and adjusting B accordingly. For the speech recognition task considered, FLASH-BS VITERBI achieves a 69.5× speedup with only a 0.05% increase in error, demonstrating substantial gains in efficiency.

E. Experiments on Resource-Constrained Devices

To assess the efficiency and deployability of FLASH variants on resource-constrained devices, we evaluate two representative platforms: an FPGA accelerator and a Raspberry Pi 5. The FPGA experiments highlight the hardware friendliness and acceleration potential of our algorithms, while the Raspberry Pi experiments validate their efficiency on an embedded platform.

1) FPGA-Based Acceleration: Figures 10 show the decoding time of FLASH VITERBI and FLASH-BS VITERBI accelerators, along with the hardware speedup of FLASH VITERBI. The results indicate that speedup increases superlinearly with state space size K, reaching 50.5× at $K = 2048$, which demonstrates the efficiency of our hardware design. However, beam search disrupts sequential DDR memory access, leading to longer decoding time for FLASH-BS VITERBI (HW) compared with FLASH VITERBI (HW). Nevertheless, its runtime remains comparable to that of the software implementation.

Table II presents the resource utilization and power consumption of FLASH-BS VITERBI, compared with Reduced-Memory Viterbi (RM-Viterbi), at beam widths of 32K and 512. Power consumption was estimated using AMD's Power Design Manager [52]. As the beam width decreases, FLASH-BS VITERBI achieves significant resource savings. Compared to RM-Viterbi, it incurs a slight LUT+FF overhead due to parallel DDR transfers, which can be mitigated by reducing the degree of transfer parallelism. For all other resource metrics, FLASH-BS VITERBI demonstrates substantial improvements. At a beam width of 32K, the Entire Decoder consumes only 26.0% and the Decoding Unit 22.9% of the BRAMs used by RM-Viterbi. This reduction in resource usage translates into improved power efficiency [53], [54], with FLASH-BS VITERBI consuming approximately 15% less power than RM-Viterbi, demonstrating its hardware efficiency.

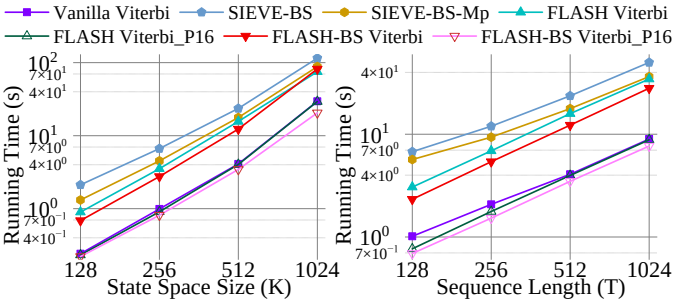

Fig. 11: Decoding time of FLASH variants and baselines on the Raspberry Pi.

2) Deployment on Raspberry Pi: We further evaluate our algorithms on a resource-constrained device, the Raspberry Pi 5 (8GB). The beam width is set to half the state space. As shown in Figure 11, although the limited computational capability of Raspberry Pi slightly reduces the benefits of multi-threading, FLASH VITERBI_P16 still matches or even outperforms the fastest baseline (Vanilla Viterbi). Moreover, beam search in FLASH-BS VITERBI further amplifies this advantage, highlighting the efficiency of FLASH variants on resource-constrained platforms.

VIII. CONCLUSION

In this work, we propose FLASH VITERBI and FLASH-BS VITERBI, two decoding algorithms designed to address the performance and memory limitations of existing Viterbi implementations. Leveraging a non-recursive divide-and-conquer paradigm and a pruning and parallelization strategy, both methods enable highly parallel execution while reducing scheduling and memory overhead. The beam search variant further employs dynamic strategies and memory-efficient data structures to enhance adaptability under complex scenarios. To support deployment on edge platforms, we develop FPGA-based hardware accelerators for both algorithms, achieving substantial speedups with minimal resource consumption. Extensive experiments demonstrate the superiority of our methods in runtime and memory efficiency, while hardware results confirm their practicality in edge computing environments. Future work includes extending our framework to practical applications and exploring GPU-based acceleration.

REFERENCES

[1] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–1563, 1966. [Online]. Available: http://www.jstor.org/stable/2238772

[2] L. Rabiner and B. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[3] W. Shi, J. Xu, J. Fang, P. Chao, A. Liu, and X. Zhou, "Lhmm: A learning enhanced hmm model for cellular trajectory map matching," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 2429–2442.

[4] L. Zhao, J. Mao, M. Pu, G. Liu, C. Jin, W. Qian, A. Zhou, X. Wen, R. Hu, and H. Chai, "Automatic calibration of road intersection topology using trajectories," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1633–1644.

[5] Z. Liu, Y. Zhou, X. Liu, H. Zhang, Y. Dong, D. Lu, and K. Wu, "Learning road network index structure for efficient map matching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 1, pp. 423–437, 2025.

[6] F. Zhu, M. Yuan, X. Xie, T. Wang, S. Zhao, W. Rao, and J. Zeng, "A data-driven sequential localization framework for big telco data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 8, pp. 3007–3019, 2021.

[7] J. Shang, J. Liu, M. Jiang, X. Ren, C. R. Voss, and J. Han, "Automated phrase mining from massive text corpora," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 10, pp. 1825–1837, 2018.

[8] Y. Liu, Q. Ge, W. Luo, Q. Huang, L. Zou, H. Wang, X. Li, and C. Liu, "Graphmm: Graph-based vehicular map matching by leveraging trajectory and road correlations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 1, pp. 184–198, 2024.

[9] J. Jiang, Z. Wen, P. Yang, A. Mansoor, and A. Mian, "Fast-pgm: Fast probabilistic graphical model learning and inference," *arXiv preprint arXiv:2405.15605*, 2024.

[10] J. Jiang, Z. Wen, A. Mansoor, and A. Mian, "Fast inference for probabilistic graphical models," in *2024 USENIX Annual Technical Conference*, 2024, pp. 95–110.

[11] K. Yao, J. Zhang, C. Qin, X. Song, P. Wang, H. Zhu, and H. Xiong, "Resuformer: Semantic structure understanding for resumes via multi-modal pre-training," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 3154–3167.

[12] X. Shi, Y. Zhang, H. Huang, Z. Hu, H. Jin, H. Shen, Y. Zhou, B. He, R. Li, and K. Zhou, "Maxson: Reduce duplicate parsing overhead on raw data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1621–1632.

[13] X. Lu and T. W. S. Chow, "Duration modeling with semi-markov conditional random fields for keyphrase extraction," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1453–1466, 2021.

[14] J. Jiang, Z. Wen, and A. Mian, "Fast parallel bayesian network structure learning," in *2022 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2022, pp. 617–627.

[15] X. Zhou, D. Qin, X. Lu, L. Chen, and Y. Zhang, "Online social media recommendation over streams," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 938–949.

[16] H. Li, H. Lu, M. A. Cheema, L. Shou, and G. Chen, "Indoor mobility semantics annotation using coupled conditional markov networks," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1441–1452.

[17] M. Ciaperoni, A. Gionis, A. Katsamanis, and P. Karras, "Sieve: A space-efficient algorithm for viterbi decoding," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1136–1145. [Online]. Available: https://doi.org/10.1145/3514221.3526170

[18] M. Ciaperoni, A. Katsamanis, A. Gionis, and P. Karras, "Beam-search sieve for low-memory speech recognition," in *Proc. Interspeech 2024*, 2024, pp. 272–276.

[19] M. Gales and S. Young, "The application of hidden markov models in speech recognition," *Foundations and Trends® in Signal Processing*, vol. 1, no. 3, pp. 195–304, 2008. [Online]. Available: http://dx.doi.org/10.1561/2000000004

[20] H. Braun, J. Luitjens, R. Leary, T. Kaldewey, and D. Povey, "Gpu-accelerated viterbi exact lattice decoder for batched online and offline speech recognition," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 7874–7878.

[21] A.-L. Georgescu, A. Pappalardo, H. Cucu, and M. Blott, "Performance vs. hardware requirements in state-of-the-art automatic speech recognition," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2021, no. 1, p. 28, 2021.

[22] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "Low-power automatic speech recognition through a mobile gpu and a viterbi accelerator," *IEEE micro*, vol. 37, no. 1, pp. 22–29, 2017.

[23] R. Yazdani, J.-M. Arnau, and A. González, "Unfold: a memory-efficient speech recognizer using on-the-fly wfst composition," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 69–81. [Online]. Available: https://doi.org/10.1145/3123939.3124542

[24] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia, "Towards efficient generative large language model serving: A survey from algorithms to systems," *ACM Computing Surveys*, vol. 58, no. 1, pp. 1–37, 2025.

[25] J. Jiang, P. Yang, R. Zhang, and F. Liu, "Towards efficient large language model serving: A survey on system-aware kv cache optimization," *Authorea Preprints*, 2025. [Online]. Available: http://dx.doi.org/10.36227/techrxiv.176046306.66521015/v1

[26] J. Feldman, I. Abou-Faycal, and M. Frigo, "A fast maximum-likelihood decoder for convolutional codes," in *Proceedings IEEE 56th Vehicular Technology Conference*, vol. 1. IEEE, 2002, pp. 371–375.

[27] S. M. Siddiqi and A. W. Moore, "Fast inference and learning in large-state-space hmms," in *Proceedings of the 22nd International Conference on Machine Learning*, ser. ICML '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 800–807. [Online]. Available: https://doi.org/10.1145/1102351.1102452

[28] C. Tarnas and R. Hughey, "Reduced space hidden markov model training." *Bioinformatics (Oxford, England)*, vol. 14, no. 5, pp. 401–406, 1998.

[29] M. Price, J. Glass, and A. P. Chandrakasan, "A low-power speech recognizer and voice activity detector using deep neural networks," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 1, pp. 66–75, 2018.

[30] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The kaldi speech recognition toolkit," Idiap, Rue Marconi 19, Martigny, Idiap-RR Idiap-RR-04-2012, 1 2012.

[31] B. Lowerre and R. Reddy, "The harpy speech recognition system: performance with large vocabularies," *The Journal of the Acoustical Society of America*, vol. 60, no. S1, pp. S10–S11, 08 2005. [Online]. Available: https://doi.org/10.1121/1.2003089

[32] S. Kobashikawa, T. Hori, Y. Yamaguchi, T. Asami, H. Masataki, and S. Takahashi, "Efficient beam width control to suppress excessive speech recognition computation time based on prior score range normalization," in *Interspeech 2012*, 2012, pp. 1011–1014.

[33] T. Hori, S. Watanabe, and A. Nakamura, "Search error risk minimization in viterbi beam search for speech recognition," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2010, pp. 4934–4937.

[34] I. Williams and P. Aleksic, "Rescoring-aware beam search for reduced search errors in contextual automatic speech recognition," in *Interspeech 2017*, 2017, pp. 508–512.

[35] G. He, T. Sugahara, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "A 40-nm 168-mw 2.4×-real-time vlsi processor for 60-kword continuous speech recognition," in *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, 2012, pp. 1–4.

[36] M. Price, J. Glass, and A. P. Chandrakasan, "A 6 mw, 5,000-word real-time speech recognizer using wfst models," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 102–112, 2015.

[37] G. He, T. Sugahara, T. Fujinaga, Y. Miyamoto, H. Noguchi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "A 40-nm 144-mw vlsi processor for real-time 60-kword continuous speech recognition," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 71–72.

[38] F. Vargas, R. Fagundes, and D. Junior, "A fpga-based viterbi algorithm implementation for speech recognition systems," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, vol. 2, 2001, pp. 1217–1220 vol.2.

[39] M. Mozaffari Kermani, V. Singh, and R. Azarderakhsh, "Reliable low-latency viterbi algorithm architectures benchmarked on asic and fpga," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 1, pp. 208–216, 2017.

[40] P. P. Raj, P. A. Reddy, and N. Chandrachoodan, "Reduced memory viterbi decoding for hardware-accelerated speech recognition," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 3, May 2022. [Online]. Available: https://doi.org/10.1145/3510028

[41] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[42] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii, "Distributional smoothing with virtual adversarial training," *arXiv preprint arXiv:1507.00677*, 2015.

[43] P. Yang, N. Akhtar, M. Shah, and A. Mian, "Regulating model reliance on non-robust features by smoothing input marginal density," in *European Conference on Computer Vision*. Springer, 2024, pp. 329–347.

[44] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012.

[45] H.-L. Lou, "Implementing the viterbi algorithm," *IEEE Signal Processing Magazine*, vol. 12, no. 5, pp. 42–52, 1995.

[46] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*. PMLR, 2017, pp. 3319–3328.

[47] G. Erion, J. D. Janizek, P. Sturmfels, S. M. Lundberg, and S.-I. Lee, "Improving performance of deep learning models with axiomatic attribution priors and expected gradients," *Nature machine intelligence*, vol. 3, no. 7, pp. 620–631, 2021.

[48] P. Yang, N. Akhtar, Z. Wen, and A. Mian, "Local path integration for attribution," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 3, 2023, pp. 3173–3180.

[49] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey *et al.*, "The htk book," 1999.

[50] P. J. Moreno and C. Alberti, "A factor automaton approach for the forced alignment of long speech recordings," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2009, pp. 4869–4872.

[51] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren, "Darpa timit acoustic-phonetic continuous speech corpus cd-rom," 1993, nIST Speech Disc 1-1.1, National Institute of Standards and Technology.

[52] AMD. (2025) Power design manager (pdm) for adaptive socs and fpgas. Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/power-design-manager.html

[53] Y. Zhang, S. Sinha, J. Xu, and W. Zhang, "Unit: A highly unified and memory-efficient fpga-based accelerator for torus fhe," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.

[54] P. Kumar Gopalakrishnan, C.-H. Chang, and A. Basu, "Hast: A hardware-efficient spatio-temporal correlation near-sensor noise filter for dynamic vision sensors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 3, pp. 1332–1345, 2025.