

RayFST: Accelerating Finite-State Transducer Composition with Ray Tracing Cores

Tianao Ge*
Shenzhen University of Advanced
Technology
Shenzhen, China
getianao@suat-sz.edu.cn

Xiaowen Chu
The Hong Kong University of Science
and Technology (Guangzhou)
Guangzhou, China
xwchu@hkust-gz.edu.cn

Hongyuan Liu
Stevens Institute of Technology
Hoboken, USA
hliu96@stevens.edu

Abstract

Finite-state transducer (FST) composition is a fundamental but computationally expensive operation in domains such as speech recognition and natural language processing. Despite the increasing computational power of GPUs, existing parallelization approaches for FST composition face challenges due to irregular memory access during traversal and the high computational cost of comparing transitions. In this work, we leverage specialized GPU ray tracing (RT) cores to accelerate this non-graphics computation. We introduce RayFST, which maps the FST structure into a 3D spatial representation and retrieves transitions through hardware-accelerated ray intersection. We also present two ray-based optimizations: Ray Forwarding, which performs label matching directly by emitting rays across the two FSTs, and Dynamic Ray Generation, which eliminates invalid rays and balances workloads across rays. Experiments on real-world FST benchmarks show an $8.9\times$ geometric mean speedup over the state-of-the-art GPU FST library and up to $1.61\times$ end-to-end training speedup on two FST applications.

CCS Concepts

• Computing methodologies → Ray tracing; • Theory of computation → Parallel algorithms; Transducers.

Keywords

Finite-State Transducer, Ray Tracing, OptiX

ACM Reference Format:

Tianao Ge, Xiaowen Chu, and Hongyuan Liu. 2026. RayFST: Accelerating Finite-State Transducer Composition with Ray Tracing Cores. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26)*, September 28–October 01, 2026, Singapore, Singapore. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3832810.3832858>

1 Introduction

Finite-state transducers (FSTs) are a key computational model for data transformation and are widely used in tasks such as natural language processing [24, 25, 39], speech recognition [33, 44], structured data parsing [36, 37], and bioinformatics [23]. FSTs are finite-state machines in which each transition carries an input label

*Part of this work was done while the author was at HKUST (Guangzhou).



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICPP '26, Singapore, Singapore*

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2657-6/26/09
<https://doi.org/10.1145/3832810.3832858>

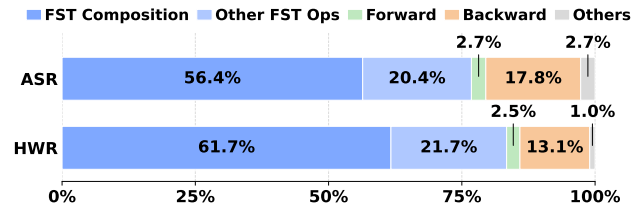


Figure 1: Average execution time breakdown per training iteration for two FST applications on an NVIDIA RTX 3090 GPU. FST composition dominates the runtime in both applications.

and an associated output label from a possibly different alphabet, transforming input sequences into output sequences. A central operation in these applications is FST composition, which constructs a new transducer that combines the behavior of two FSTs. This enables modular construction by incrementally combining transducers, making it possible to express multi-stage processing and map inputs directly to final outputs. However, FST composition can be expensive because its computational cost grows with the cross-product of the transitions in the two input transducers and is further exacerbated by irregular traversal patterns. For instance, large-vocabulary automatic speech recognition systems often construct complex decoding graphs by composing multiple FSTs across acoustic, context, lexical and language-model levels [32]. More recently, differentiable FST frameworks require the FST to be explicitly composed during training [19], making precomputed or on-the-fly composition inapplicable [21]. In these workloads, composition becomes part of the critical execution path, and FSTs are constructed dynamically. As a result, composition latency directly contributes to end-to-end performance.

Achieving fast FST composition thus demands hardware-level acceleration. Researchers have turned to GPUs, whose massive thread-level parallelism and high memory bandwidth support transition lookup and label matching [10, 40]. To understand the performance implications of FST composition in practice on GPUs, we profile the training of two differentiable FST applications (detailed in §5.4): automatic speech recognition (ASR) [38] and handwriting recognition (HWR) [30]. Both applications require executing multiple FST compositions per training iteration. We use GTN [40], the state-of-the-art GPU implementation of FST composition, to perform these FST computations in the applications. As shown in Figure 1, FST composition takes 56.4% and 61.7% of the training time in ASR and HWR, respectively. FST composition is more expensive than the

forward and backward passes of the neural network. These results show that FST composition remains the main execution bottleneck in end-to-end training. In this paper, our characterization shows that the bottleneck remains on conventional GPU cores for two main reasons: (1) **Irregular memory access from graph traversal**: simultaneous traversal of two graph-structured FSTs scatters memory requests across addresses. This pattern prevents coalesced memory accesses on GPUs and lowers throughput. (2) **Redundant comparisons in label matching**: during composition, transitions from two input states must be compared by their labels to identify matching transitions. This cross-product creates quadratic work, but only a small subset of comparisons produce matches. As a result, GPUs spend much of their computation on comparisons that do not produce composed transitions.

Beyond conventional GPU cores, modern GPUs also integrate specialized ray tracing (RT) cores, which were first introduced in NVIDIA's Turing architecture [2]. RT cores were originally designed to accelerate graphics rendering by efficiently detecting ray-primitive intersections using a hardware-accelerated Bounding Volume Hierarchy (BVH). Recent studies have shown that RT cores can also accelerate irregular, non-graphics workloads [16, 45], which share characteristics with the sparse and irregular traversal patterns of FSTs.

In this work, we present RAYFST, a novel approach that leverages GPU RT cores to accelerate FST composition. To address the challenges on GPUs, RAYFST maps the structure of FSTs into a 3D spatial representation, where transitions are modeled as geometric primitives. This spatial embedding is then indexed using a BVH, enabling RT cores to perform hardware-accelerated traversal. RAYFST further introduces two ray-based optimizations to enhance performance: (1) Ray Forwarding, a ray emission mechanism that integrates traversal and label matching within a single ray-tracing pass. It conditionally generates new rays upon intersections according to the label, eliminating redundant global memory writes. (2) Dynamic Ray Generation, which dynamically controls ray emission based on the available transitions of FSTs. A ray is emitted only if the current state has at least one outgoing transition with the target inner label, eliminating rays that would miss all primitives. Load balance is further improved by issuing rays across multiple partitions. To the best of our knowledge, RAYFST is the first work to leverage RT cores for accelerating FST composition. Our contributions are summarized as follows:

- We identify key performance limitations of FST composition on GPUs and introduce RAYFST, which embeds FSTs into a spatial representation to enable execution on RT cores.
- We further propose two ray-based optimizations: Ray Forwarding, which fuses transition lookup and label matching into the ray-tracing pipeline, and Dynamic Ray Generation, which eliminates invalid rays and balances workloads across rays.
- We conduct extensive experiments showing that RAYFST achieves average speedups of 5.8 $\times$, 11.0 $\times$, and 8.9 $\times$ over OpenFst, k2, and GTN on real-world FST benchmarks without offline tuning. These gains translate into 1.44 $\times$ –1.61 $\times$ end-to-end training latency speedups on two FST applications.

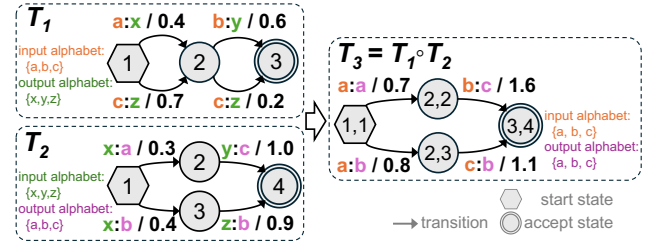


Figure 2: Two weighted finite-state transducers T_1 and T_2 , and the resulting composed transducer $T_3 = T_1 \circ T_2$.

Algorithm 1: Serial FST Composition Algorithm

Input: Two FSTs $T_1 = (Q_1, \Sigma, \Gamma, s_1, F_1, \delta_1)$, $T_2 = (Q_2, \Gamma, \Delta, s_2, F_2, \delta_2)$
Output: Composed FST $T_3 = (Q_3, \Sigma, \Delta, s_1 s_2, F_1 \times F_2, \delta_3)$

```

1  worklist  $\leftarrow \{s_1 s_2\}$ ,  $Q_3 \leftarrow \{s_1 s_2\}$ ,  $\delta_3 \leftarrow \emptyset$ ;
2  while |worklist| > 0 do
3     $q_1 q_2 \leftarrow \text{pop}(\text{worklist})$ ;
4     $E_1 \leftarrow \{(q_1, a_1, b_1, p_1, r_1) \mid q_1 \xrightarrow{a_1: b_1 / p_1} r_1 \in \delta_1\}$ ;
5     $E_2 \leftarrow \{(q_2, a_2, b_2, p_2, r_2) \mid q_2 \xrightarrow{a_2: b_2 / p_2} r_2 \in \delta_2\}$ ;
6    foreach  $(q_1, a_1, b_1, p_1, r_1) \in E_1$  do
7      foreach  $(q_2, a_2, b_2, p_2, r_2) \in E_2$  do
8        process  $\epsilon$ -label transitions
9        if  $b_1 = a_2$  then
10         append  $(q_1 q_2, a_1, b_2, p_1 + p_2, r_1 r_2)$  to  $\delta_3$ ;
11         if  $r_1 r_2 \notin Q_3$  then
12            $Q_3 \leftarrow Q_3 \cup \{r_1 r_2\}$ ;
13           push(worklist,  $r_1 r_2$ );
14           if  $r_1 \in F_1$  and  $r_2 \in F_2$  then
15             mark  $r_1 r_2$  as an accept state;
16 return  $(Q_3, \Sigma, \Delta, s_1 s_2, F_1 \times F_2, \delta_3)$ ;
```

2 Background

2.1 Finite-State Transducers and Composition

Finite-State Transducers. A finite-state transducer (FST) is defined as a 6-tuple $(Q, \Sigma, \Gamma, s, F, \delta)$. Here, Q is a finite set of states, Σ and Γ denote the input and output alphabets, respectively; $s \in Q$ is the start state, and $F \subseteq Q$ is the set of accept states. The transition function $\delta(q, a, b, r)$ specifies when a transition from state q to state r is allowed with *input label* a and *output label* b , where $a \in \Sigma \cup \{\epsilon\}$, $b \in \Gamma \cup \{\epsilon\}$, and ϵ denotes the empty symbol. Finite-state transducers can be weighted, where each transition is associated with a weight p in addition to its labels, written as $q \xrightarrow{a: b / p} r$. The left part of Figure 2 shows an example of two transducers T_1 and T_2 , represented as a labeled directed graph.

FST Composition. Given two transducers $T_1 = (Q_1, \Sigma, \Gamma, s_1, F_1, \delta_1)$ and $T_2 = (Q_2, \Gamma, \Delta, s_2, F_2, \delta_2)$, their composition aligns the output labels of T_1 with the input labels of T_2 and combines transitions that match on this shared label. The resulting transducer is $T_3 = (Q_3, \Sigma, \Delta, s_1 s_2, F_1 \times F_2, \delta_3)$. Each transition $(q_1, q_2) \xrightarrow{a: c / (p_1 + p_2)} (r_1, r_2)$ in δ_3 exists if there are

Iter.	State Pairs (q_1, q_2)	Transitions from q_1	Transitions from q_2	Generated Transitions	Next State Pairs (r_1, r_2)
1	(1,1)	$1 \xrightarrow{a:x/0.4} 2$ $1 \xrightarrow{c:z/0.7} 2$	$1 \xrightarrow{x:a/0.3} 2$ $1 \xrightarrow{x:b/0.4} 3$	$(1,1) \xrightarrow{a:a/0.7} (2,2)$ $(1,1) \xrightarrow{a:b/0.8} (2,3)$	(2,2) (2,3)
2	(2,2)	$2 \xrightarrow{b:y/0.6} 3$ $2 \xrightarrow{c:z/0.2} 3$	$2 \xrightarrow{y:c/1.0} 4$	$(2,2) \xrightarrow{b:c/1.6} (3,4)$	(3,4)
	(2,3)	$2 \xrightarrow{b:y/0.6} 3$ $2 \xrightarrow{c:z/0.2} 3$	$3 \xrightarrow{z:b/0.9} 4$	$(3,2) \xrightarrow{c:b/1.1} (4,3)$	(3,4)
3	(3,4)	ϕ	ϕ		

Figure 3: Step-by-step example of iterative FST composition for T_1 and T_2 in Figure 2. Each row expands one state pair, lists the outgoing transitions from both FSTs, and shows the composed transitions and newly generated state pairs for the next iteration.

transitions $q_1 \xrightarrow{a:b/p_1} r_1$ in δ_1 and $q_2 \xrightarrow{b:c/p_2} r_2$ in δ_2 . We refer to the label b in Γ as the *inner label* used for matching transitions, while the input label a from Σ and the output label c from Δ serve as the *outer labels* in the composition. The right part of Figure 2 shows the composed transducer $T_3 = T_1 \circ T_2$. Constructing the composed FST is an iterative process, summarized in Algorithm 1. It starts from the start-state pair (s_1, s_2) of T_1 and T_2 . In each iteration, two steps are performed for every state pair (q_1, q_2) : *transition lookup* (line 4–5) and *label matching* (line 6–15). Transition lookup retrieves all outgoing transitions from q_1 in T_1 and from q_2 in T_2 , and label matching compares their inner labels. When an inner label is ϵ , the composition moves forward only in the transducer that has the ϵ -transition and leaves the other transducer at the same state. Matched transitions generate new states and transitions in the composed FST, and any newly created state pairs are added to the worklist for the next iteration until no additional pairs are produced. Figure 3 illustrates how T_1 and T_2 are composed step by step to produce T_3 . Many CPU-based methods have been proposed [8, 11, 12, 21, 22, 34] to accelerate FST composition.

FST Composition on GPUs. Parallelizing FST composition on GPUs has been explored in prior work [10, 40]. These schemes assign GPU threads to process many state pairs and their outgoing transitions in parallel. To support traversal on GPUs, the two FSTs are stored in a CSR (Compressed Sparse Row) layout, as shown in Figure 4. This layout stores the start position and associated data of each state’s outgoing transitions in contiguous arrays, and threads use these offsets to retrieve all transitions of a state. A separate kernel then maps threads to transition pairs and performs label matching. In Figure 4, for the state pair $(1, 1)$, the offsets show that each state has two outgoing transitions, giving four candidate pairs to compare. Four threads check their inner labels, and two of these comparisons result in composed transitions.

2.2 Ray Tracing

RT Cores and BVH Traversal. Ray tracing computes which geometric primitives are intersected by a ray in a 3D scene. A naive search tests every ray against every primitive, so its cost grows quickly with scene size. Modern GPUs therefore include dedicated

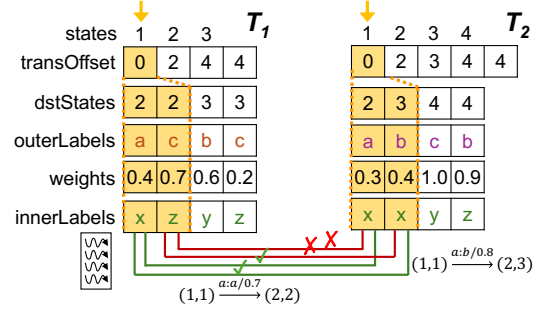


Figure 4: The CSR representation of two FSTs, illustrating transition lookup and label matching for the state pair $(1, 1)$. Each transition from state 1 in T_1 is compared with every transition from state 1 in T_2 , requiring four comparisons.

RT cores to accelerate this process [2]. These units operate on a bounding volume hierarchy (BVH), which organizes primitives into a tree of axis-aligned bounding boxes. During traversal, if a ray misses a box, the entire subtree under that box is skipped. Only the boxes and primitives along hit paths need further processing. This hierarchical pruning reduces both intersection tests and irregular memory accesses. In practice, RT cores accelerate BVH traversal together with ray-box and ray-primitive intersection, while the remaining programmable logic still runs on the regular GPU cores.

Ray Tracing Pipeline. NVIDIA OptiX [6] exposes RT hardware through a programmable pipeline. A developer first represents the input data as geometric primitives and builds a BVH over them. A *ray-generation* program issues rays, RT cores traverse the BVH, and user-defined programs handle the corresponding events. For built-in primitives, the intersection test is handled directly by hardware. For custom primitives, the user provides an *intersection* program to determine whether a hit occurs. The *closest-hit*, *any-hit*, and *miss* programs then process the traversal result on the GPU cores. This pipeline is well suited to problems that can be expressed as queries over sparse or irregular data. For non-graphics computing, the key benefit of the ray-tracing pipeline is that hardware-accelerated BVH traversal on RT cores enables efficient search space pruning, while its programmable stages execute user-defined operations on CUDA cores upon hit and miss events. As a result, mapping sparse workloads to this pipeline offloads complex traversal logic to dedicated hardware, thereby mitigating the irregular memory accesses that typically bottleneck conventional SIMT execution.

3 Motivation

As shown in Figure 1, FST composition dominates end-to-end training time. This motivates us to analyze the GPU execution of FST composition itself and identify what limits its efficiency. To this end, we profile GTN [40] with NVIDIA Nsight Compute on an NVIDIA RTX 3090 GPU (methodology is described in §5.1). The FST benchmarks are extracted from four real-world applications, which are detailed in §5.2. We also compare the same metrics against our proposed RayFST, which will be discussed in §4.

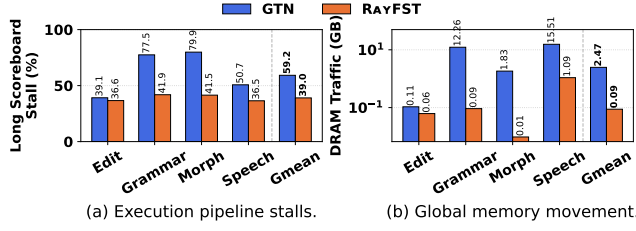


Figure 5: Memory profiling results for FST composition on real-world FST benchmarks.

3.1 Challenge #1: Irregular Memory Access

FST composition traverses two graphs T_1 and T_2 at the same time. At every iteration, each state pair (q_1, q_2) in the worklist issues two independent CSR indirections: one uses q_1 to index T_1 's offsets and fetch its outgoing transitions, the other uses q_2 to do the same in T_2 . The two indirections target disjoint memory regions, so the working set spans both FSTs at once. Worse, the state pairs are produced dynamically by the previous iteration's matching, so neither lookup target is known in advance, and threads in the same warp are typically assigned uncorrelated state pairs. The resulting memory accesses scatter across both FSTs, making them difficult to coalesce and prefetch effectively.

Memory Pressure. We measure memory pressure using two metrics: the long-scoreboard stall ratio, defined as the fraction of warp samples stalled by long-latency memory accesses, and the total DRAM traffic. As shown in Figure 5, GTN is stalled by memory accesses in 39.1%–79.9% of warp samples (59.2% on average), with Grammar and Morph spending nearly 80% of warp samples waiting for data. Meanwhile, DRAM traffic averages 2.4 GB, and large workloads such as Grammar (12.26 GB) and Speech (15.51 GB) generate more than 12 GB of off-chip memory traffic. These results show that the high memory overhead mainly comes from inefficient access patterns rather than computation itself. In comparison, RAYFST (detailed in §4) lowers the average stall ratio to 39.0% and reduces DRAM traffic to 0.09 GB on average.

Summary. Joint CSR traversal turns transition lookup into irregular indirections over two disjoint memory regions. Addressing this bottleneck requires a memory-access mechanism that can tolerate sparse and unpredictable traversal.

3.2 Challenge #2: Redundant Label Comparisons

To compose FSTs at a state pair (q_1, q_2) , the output labels of transitions from q_1 must be matched against the input labels of transitions from q_2 . We refer to each label comparison as a *query*, which checks whether a transition from q_1 and a transition from q_2 share the same inner label. As illustrated in Figure 4, the state pair $(1, 1)$ issues four queries, but only two queries match. If the query matches, the pair can produce a composed transition. Although the number of such queries grows quadratically with the average out-degree, only a small fraction of them match. As a result, exhaustive cross-product matching performs a large amount of redundant work and leads to low hardware efficiency.

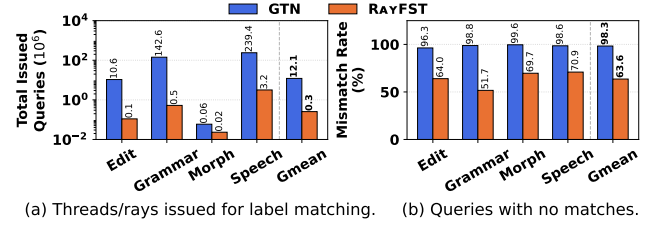


Figure 6: Profiling results for label-matching queries in FST composition on real-world FST benchmarks.

Matching Inefficiency. To quantify this inefficiency, we instrument the kernels and report two metrics: *Total Issued Queries* and *Mismatch Rate*. *Total Issued Queries* counts the number of work items dispatched for label matching, where a work item is a thread in GTN and a ray in RAYFST. *Mismatch Rate* is the fraction of issued queries that do not produce a matched label. As shown in Figure 6 (a), GTN launches up to 239.4M threads on Speech and 12.1M threads on average across the four benchmarks. Meanwhile, Figure 6 (b) shows that 96.3%–99.6% of these queries miss, with an average miss rate of 98.3%. This indicates that GTN dispatches tens of millions of work items per benchmark, only to discard 98% of them as no-ops. This sparsity comes from the composition semantics: most (e_1, e_2) pairs do not share an inner label. However, explicitly launching one thread to verify each mismatch is unnecessary. RAYFST issues only 0.3M rays on average, which is 1/40 as many as GTN. It also lowers the miss rate to 63.6%. More importantly, a RAYFST miss is pruned during BVH traversal, rather than being handled by a dispatched CUDA thread that explicitly executes a failed comparison.

Summary. Cross-product label matching couples quadratic dispatch growth with near-zero hit rates. Eliminating this overhead requires a matching primitive that surfaces only label-aligned transitions, instead of enumerating the full cross-product up front.

3.3 Opportunity: Ray-Based FST Composition

The two challenges point to two complementary requirements: a memory-access mechanism that tolerates irregular dual indirections, and a matching primitive that avoids exhaustive cross-product enumeration. Software indexing techniques, such as hashing and sorting, can reduce transition lookup overhead. However, irregular search still runs on conventional GPU cores. Modern GPU ray-tracing pipelines provide both capabilities: (1) The bounding-volume hierarchy (BVH) organizes sparse and irregular spatial data into a hardware-traversable index, which can prune empty regions during traversal and reduce unnecessary memory accesses. (2) Meanwhile, the ray-primitive intersection engine reports only the primitives intersected by a ray. This provides a selective matching mechanism that avoids explicitly comparing all edge pairs. By embedding FST transitions as 3D primitives and representing label queries as rays, FST composition can be reformulated as an intersection problem. This mapping allows us to use existing GPU ray-tracing hardware for the two main bottlenecks of FST composition, motivating the RAYFST design in the next section.

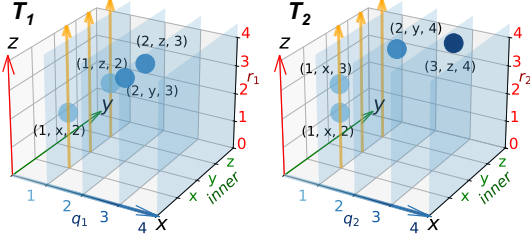


Figure 7: Example of mapping FSTs to 3D primitives for ray tracing, where the x -, y -, and z -axes correspond to the source state, inner label, and destination state, respectively. The yellow-highlighted rays illustrate the process of querying outgoing transitions from a given state and inner label.

4 RAYFST: Ray Tracing for FST Composition

We present RAYFST, a ray-tracing-based FST composition approach that maps transitions to geometric primitives and leverages RT cores for BVH traversal. This section first introduces how RAYFST uses ray tracing for transition lookup (§4.1), and then presents two optimizations (§4.2 and §4.3) that address the remaining challenges.

4.1 Mapping FSTs to Ray Tracing

Spatial Mapping of FSTs. To leverage RT cores, we first embed FST transitions into a three-dimensional space that is indexed by a BVH. Each transition of an FST is represented as a sphere primitive together with its associated attributes. We use *spheres* instead of *triangles* because a sphere only requires a center and a radius, while a triangle needs three vertices, which reduces memory usage. The radius of each sphere is set to a small constant value (e.g., <0.5) to ensure intersection detection without spatial overlap. Specifically, a transition $q_1 \xrightarrow{a:b/p_1} r_1$ in T_1 maps to a sphere centered at (q_1, b, r_1) . A transition $q_2 \xrightarrow{b:c/p_2} r_2$ in T_2 maps to a sphere centered at (q_2, b, r_2) . The outer labels a and c , together with weights p_1 and p_2 , are stored as metadata associated with the sphere IDs. By encoding the source state, inner label, and destination state into the spatial coordinates, ray intersections capture both state connectivity and label information. The inner-label alphabet Γ includes ϵ , which we encode at a reserved y -coordinate so that ϵ -transitions share the same BVH as regular transitions and require no special data structure. Figure 7 illustrates the 3D representation of FSTs T_1 and T_2 from Figure 2. These primitives are then organized into a BVH, which serves as the acceleration structure for efficient ray traversal and intersection during composition.

Ray-Based Transition Lookup. After embedding FST transitions into the 3D space, RAYFST retrieves outgoing transitions from a given state by emitting rays along the z -axis and detecting intersections with primitives. Each ray queries outgoing transitions from a source state with a specified inner label. For inner label b , rays from q_1 and q_2 retrieve transitions of the forms $q_1 \xrightarrow{a:b/p_1} r_1$ and $q_2 \xrightarrow{b:c/p_2} r_2$, respectively. As shown in Figure 7, a ray emitted from $(1, x, 0)$ intersects with the sphere located at $(1, x, 2)$ in T_1 , indicating a valid transition from state 1 to state 2 with the output

label x . During FST composition, all transitions from each state pair (q_1, q_2) in the two input FSTs will be explored. To cover all possible inner labels, we emit $|\Gamma|$ rays from origin points $(q, i, 0)$ for each $i \in \Gamma$, all directed along the z -axis. Figure 7 illustrates this process for the state pair $(1, 1)$, where each FST emits three rays and four intersections are detected in total. The intersection results and their metadata are written back to global GPU memory. A subsequent GPU kernel then matches the inner labels and constructs new states and transitions for the composed FST (lines 6–15 in Algorithm 1). Among the $|\Gamma|$ rays, the one emitted at the ϵ coordinate retrieves ϵ -transitions. When such a ray reports a hit, the composition kernel follows the standard ϵ -handling rule of FST composition, advancing the state of that transducer alone while keeping the other state unchanged.

Challenges for Ray Tracing. While the ray-based transition lookup leverages hardware traversal for parallel exploration, two key inefficiencies remain: (1) **Separated transition lookup and label matching.** Although CSR-based traversal is replaced by ray tracing, the composition step still requires an additional kernel to compare labels and construct the composed FST. Intersection results and metadata must be written to global memory after the traversal pass and then reloaded by the composition kernel, leading to high memory traffic and extra kernel-launch overhead. (2) **Redundant ray emission and workload imbalance.** Uniformly emitting $|\Gamma|$ rays for every state and inner label leads to redundant computation and unbalanced workloads. Most states have only a few outgoing transitions, so many rays miss all primitives, while highly connected states generate many more intersections and dominate execution time, as in the Grammar and Morph FSTs in Table 3. Next, we present two optimizations, Ray Forwarding and Dynamic Ray Generation, to address these inefficiencies and improve performance.

4.2 Optimization #1: Ray Forwarding

The key idea of Ray Forwarding is to forward rays between T_1 and T_2 to fuse transition lookup and matching into one ray-tracing kernel.

Using Coordinates for Label Matching. In RAYFST, the y -axis encodes the inner label shared by the two FSTs. This design creates a unified label space where transitions with the same inner label are aligned along the y coordinate. Any ray emitted at a given y position therefore intersects only transitions with that inner label, making each intersection test check both state connectivity and label equality. For example, in Figure 7, two rays emitted from $(1, x, 0)$ in T_1 and T_2 can only intersect primitives whose inner label is x , ensuring that each intersection inherently corresponds to matching inner labels in both FSTs.

Label-Aware Ray Forwarding. We integrate label matching into the ray-tracing process by conditionally forwarding rays based on intersection results, as illustrated in Figure 8. When a ray emitted from a state in T_1 intersects a primitive with inner label x , a new ray is forwarded from the corresponding state in T_2 with the same label x . The metadata associated with the hit primitive in T_1 is attached to the forwarded ray and kept in registers. When this ray later intersects a primitive in T_2 , the composed transition is generated immediately, without additional global memory accesses. Rays for ϵ -transitions are not forwarded: a hit on an ϵ -transition

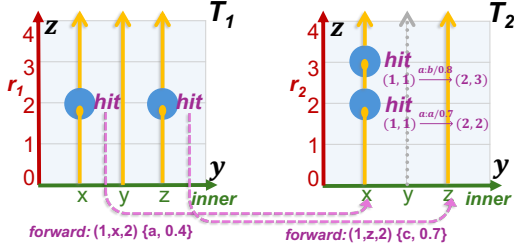


Figure 8: Example of Ray Forwarding from the state pair (1, 1). The figure shows the yOz plane at $x=1$. Rays hitting primitives in T_1 are forwarded to T_2 with the same inner label and carry the hit metadata. Gray arrows indicate rays not issued.

in either transducer directly yields a composed transition that advances the corresponding state, as defined in §4.1. Because only rays that hit a primitive in T_1 are forwarded to T_2 , Ray Forwarding skips traversal in T_2 for inner labels that have no matching transitions in T_1 (e.g., inner label y in T_2 in Figure 8). This avoids unnecessary label comparisons and redundant transition lookup in T_2 . Although a CUDA implementation can similarly fuse the two stages to reduce intermediate memory accesses, Ray Forwarding expresses the lookup in T_2 as a label-constrained BVH traversal. This allows RT cores to prune unrelated primitives before matching intersections.

Traversal Direction. When a ray in one FST hits multiple primitives, Ray Forwarding generates a forwarded ray for each hit in the other FST. These forwarded rays share the same origin and inner label, so they traverse the same regions and tend to serialize the exploration of those transitions. As shown in Figure 8, if the traversal proceeds from T_2 to T_1 , a single ray in T_2 can generate two forwarded rays that both visit the primitive $(1, x, 2)$ in T_1 , causing the same BVH region to be traversed multiple times. To mitigate this effect, RAYFST exposes the traversal direction as a parameter $d \in \{T_1 \rightarrow T_2, T_2 \rightarrow T_1\}$ and selects the direction that reduces repeated traversal and avoids serialized execution.

4.3 Optimization #2: Dynamic Ray Generation

State out-degrees in FSTs are often highly skewed, with many states having only a few outgoing transitions and some states exhibiting much larger fan-out. When we emit a ray, this skew creates two sources of inefficiency. First, many rays have no matching transition and miss all primitives; we refer to such rays as *invalid rays*. Second, primitives are unevenly distributed in space, creating workload imbalance. To address them, we present Dynamic Ray Generation to reduce invalid rays and balance workloads.

Eliminating Invalid Rays. A ray is *invalid* if it misses all primitives, indicating that there is no transition from the corresponding state with that inner label. For example, in Figure 8, a ray emitted from $(1, y, 0)$ in T_1 would be invalid because state 1 has no transitions with inner label y . Emitting such rays wastes computation, so we preprocess the FSTs to determine, for each state, which inner labels have transitions. Rays are generated only when transitions exist. As illustrated in Figure 9, by projecting the primitives onto the $x-y$ plane, we obtain a *ray emission map* that identifies the

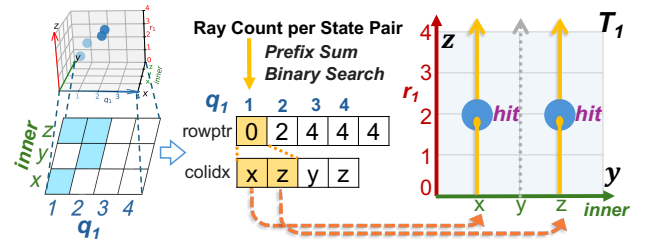


Figure 9: Ray emission map for eliminating invalid rays. For state 1 in T_1 , rays are generated only for inner labels x and z since label y has no transitions. The runtime computes ray counts, applies a prefix sum, and uses binary search to map each ray to its state and inner label.

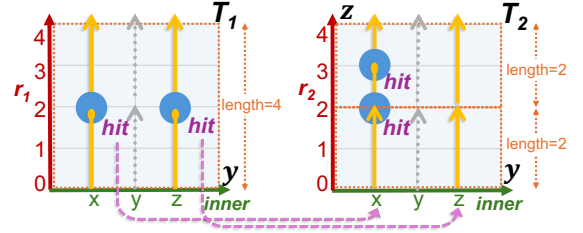


Figure 10: Balancing workloads by partitioning the z -axis. The z -axis of T_2 is split into two equal partitions ($k_2 = 2$) to limit each ray's traversal range and mitigate imbalance.

states and inner labels that have transitions. This map is stored in a lightweight CSR format, which enables efficient detection of state-label combinations that have valid transitions.

When the composition processes state pairs in an iteration, we compute, for each state q in T_i , the number of valid inner labels as $\text{rowptr}[q + 1] - \text{rowptr}[q]$ to determine how many rays it should emit. As shown in Figure 9, state 1 in T_1 emits only two rays. Before launching the ray-tracing kernel, we compute a prefix sum over these counts so that each state knows the starting position of its labels. Each ray then uses its global index and a binary search on the prefix-sum array to find its state and inner label. Because these extra computations add overhead, a parameter m disables the ray emission map when the workload can't saturate the GPU.

Balancing Workloads Across Rays. The spatial distribution of primitives is often non-uniform, so some rays traverse many primitives while others terminate quickly. On SIMT GPUs, threads in a warp execute in lockstep, so the warp cannot finish until the longest-running ray completes, leading to load imbalance. One way to mitigate this is to remap primitives in 3D space to obtain a more uniform distribution [43], but such remapping requires runtime analysis of primitive coordinates and incurs substantial overhead, which is unacceptable for FST composition. Instead, we use a lightweight balancing strategy that assigns each transducer T_i a partition count k_i along the z -axis. By shifting ray starting positions and limiting their traversal ranges according to k_i , we reduce workload imbalance with minimal overhead while maintaining high

Table 1: Key parameters and statistics for adaptive parameter selection.

Type	Notation	Description
Parameter	d	Traversal direction in Ray Forwarding.
Parameter	m	Ray-emission map switch in Dynamic Ray Generation.
Parameter	k_i	Number of z-axis partitions for transducer T_i in Dynamic Ray Generation.
Statistic	$ W $	The size of the worklist W of state pairs processed in the current iteration.
Statistic	$\Lambda_i(q)$	Inner labels of outgoing transitions from state q in T_i .
Statistic	$\deg(q)$	Out-degree of state q in T_i .
Statistic	Θ_{gpu}	Maximum number of concurrent rays on the target GPU.

occupancy. Figure 10 illustrates this strategy: T_1 and T_2 are divided into $k_1 = 1$ and $k_2 = 2$ partitions along the z-axis, respectively. For each inner label in T_2 , two rays are emitted from different starting positions (e.g., $z = 0$ and $z = 2$), and each ray traverses a z-range of length 2.

4.4 Adaptive Parameter Selection

RAYFST selects three parameters at runtime based on the input FSTs and the runtime conditions in the current iteration: the traversal direction d , the ray-emission map switch m , and the partition count k_i for each transducer. This strategy uses lightweight statistics to capture forwarding pressure, pruning opportunity, and available parallelism. Table 1 lists the key notation.

Traversal Direction d . Ray Forwarding forwards every hit in the first transducer to the second one. If the first traversal produces many hits, many forwarded rays with the same origin and inner label are generated. These rays tend to revisit the same BVH regions in the second transducer, which increases repeated traversal and serialized execution. To decide the traversal direction that reduces repeated traversal for each transducer T_i , we measure

$$\rho_i = \frac{\sum_{q \in Q_i} |\Lambda_i(q)|}{|Q_i| \cdot |\Gamma|}, \quad S_i = \rho_i \cdot \max_{q \in Q_i} \deg(q),$$

where Q_i is the state set of T_i , and Γ is the inner-label alphabet. ρ_i measures how often a state-label pair is active, and $\max_{q \in Q_i} \deg(q)$ captures the largest fan-out after a hit. Their product S_i estimates the forwarding pressure when T_i is traversed first. We therefore set

$$d = \begin{cases} T_1 \rightarrow T_2, & S_1 \leq S_2, \\ T_2 \rightarrow T_1, & S_2 < S_1. \end{cases}$$

This choice starts from the transducer with the smaller forwarding pressure and reduces repeated traversal in the second BVH.

Ray-Emission Map Switch m . Enabling the ray-emission map filters invalid rays before traversal, but it also adds prefix-sum and index-lookup overhead. We enable it only when the unfiltered workload is large enough that removing rays is more beneficial than preserving occupancy. For the worklist W of state pairs processed in the current iteration, the unfiltered ray count is $|W| \cdot |\Gamma|$. When the unfiltered ray count is smaller than Θ_{gpu} , the GPU cannot be fully occupied. In this case, filtering rays would shrink the workload

Table 2: Parameter configurations of RAYFST used in the evaluation.

Config.	Traversal direction: d	Invalid-ray elimination: m	Partition count: k_i
RAYFST-O0	$\backslash$	$\backslash$	$\backslash$
RAYFST-O1	$T_1 \rightarrow T_2$	$\backslash$	$\backslash$
RAYFST-O2	$T_1 \rightarrow T_2$	on	2
RAYFST	Adaptive Parameter Selection		
RAYFST-TUNED	$\{T_1 \rightarrow T_2, T_2 \rightarrow T_1\}$	$\{\text{on}, \text{off}\}$	$\{1, 2, 4, 8\}$

further and may underutilize the GPU. We therefore choose

$$m = \begin{cases} \text{off}, & |W| \cdot |\Gamma| < \Theta_{\text{gpu}}, \\ \text{on}, & \text{otherwise.} \end{cases}$$

Partition Count k_i . Partitioning the z-axis increases the number of rays and shortens each ray's traversal range. We estimate the ray count R and apply partitioning only when R cannot saturate the GPU. Assuming $d = T_1 \rightarrow T_2$ (the other direction is symmetric), rays are emitted from T_1 , so

$$R = \begin{cases} |W| \cdot |\Gamma|, & m = \text{off}, \\ \sum_{(q_1, q_2) \in W} |\Lambda_1(q_1)|, & m = \text{on}. \end{cases}$$

When $m = \text{off}$, every state pair in W emits one ray per inner label, giving $|W| \cdot |\Gamma|$ rays. When $m = \text{on}$, the ray-emission map restricts each state pair (q_1, q_2) to its $|\Lambda_1(q_1)|$ labels with valid transitions in T_1 . If R reaches the maximum number of concurrent rays on the target GPU ($R \geq \Theta_{\text{gpu}}$), we keep a single partition. Otherwise, we use the average out-degree $(\sum_{q \in Q_i} \deg(q)) / |Q_i|$ to decide how aggressively to split transducer T_i . A larger average out-degree usually means denser z-ranges and longer ray traversals, so finer partitioning is more useful. We limit the candidates to a small set $K = \{1, 2, 4, 8\}$ and choose

$$k_i = \begin{cases} \arg \min_{k \in K} |k - (\sum_{q \in Q_i} \deg(q)) / |Q_i||, & R < \Theta_{\text{gpu}}, \\ 1, & \text{otherwise,} \end{cases}$$

5 Evaluation

5.1 Experimental Setup

System Configuration. We evaluate our RAYFST on a computer with an NVIDIA RTX 3090 (Ampere architecture, 24 GB memory, 6 MB L2 cache, and 82 SMs). The GPU is equipped with second-generation ray tracing cores for hardware acceleration [3]. The system runs Linux on a 12-core Intel Xeon 4214R CPU and 128 GB memory. All programs are compiled with CUDA 13.0 and GCC 13.2 with the compilation options `-O3 -arch=sm_86`. The ray-tracing kernels in RAYFST are implemented using NVIDIA OptiX 9.0 APIs [6]. The spatial mapping and optimizations in RAYFST rely on hardware-accelerated BVH traversal within a programmable ray-tracing pipeline. The same design can target other RT-enabled architectures, such as AMD GPUs [4, 31] and Intel GPUs [7, 42]. We measure the FST composition time in milliseconds, including pre-processing on the input FSTs for all evaluated schemes, CPU-GPU data transfer for GPU schemes and BVH construction for RAYFST.

Evaluated Schemes. We compare RAYFST against three state-of-the-art FST libraries, including two CPU baselines and one GPU baseline. The CPU baselines are: (1) OpenFst [9], a widely used

Table 3: Characteristics of the real-world FST benchmarks. Each benchmark consists of two transducers, denoted L (left) and R (right). The table reports the number of states and transitions, the input and output alphabet sizes, and the average and maximum per-state-label out-degrees for both transducers.

Benchmark	#States		#Transitions		Input alph.		Output alph.		Avg deg.		Max deg.	
	L	R	L	R	L	R	L	R	L	R	L	R
Edit	2	76,867	158	198,284	29	27	27	27	2.3	1.0	3	1
Grammar	111,591	161	239,845	5,032	128,256	73	73	98	1.6	1.0	59,082	2
Morph	140	98,663	7,064	129,502	74	181	79	171	1.0	1.1	1	28
Speech	251	27,397	17,250	37,703	69	69	69	10,001	1.0	1.0	1	4

CPU library that provides a variety of FST operations. (2) k2 [5], an FST library for CPU and GPU execution that composes FSTs by inverting one FST and intersecting sorted transitions. Since it does not support composition with ϵ -transitions on GPU, we use its CPU routine. The GPU baseline is: (3) GTN [40], a GPU-based FST library that performs parallel composition on a CSR-like representation. We do not compare against the GPU implementation from prior work [10] because it could not complete most benchmarks within 10 minutes. In this section, RAYFST denotes the default configuration with both optimizations enabled and adaptive parameter selection. We use the other configurations of RAYFST to study parameter choices and the effect of each optimization, summarized in Table 2. For real-world FSTs, we report RAYFST-TUNED, which evaluates all 64 parameter combinations and keeps the fastest run under a 30-minute search limit per benchmark. For synthetic FSTs, we report RAYFST-O0, RAYFST-O1, and RAYFST-O2 to evaluate the two optimizations separately. To ensure correctness, we verify every composed FST with OpenFst’s RandEquivalent tool.

5.2 Performance on Real-World FSTs

FST Benchmarks. We construct a set of benchmarks consisting of FSTs extracted from four real-world workloads and evaluate the performance of their composition operations. These workloads cover error-tolerant recognition, constrained LLM decoding, morphological translation, and speech recognition: (1) Edit simulates error-tolerant word recognition [17] by composing an edit-distance transducer with a lexicon FST built from the LibriSpeech corpus [38]. (2) Grammar is used for constrained decoding in large language models by composing a lexicon FST over Llama-3.2 [41] tokens with a Python grammar transducer for token-to-terminal mapping [25, 39]. (3) Morph performs morphological analysis and translation by composing the English morphological analyzer with the English-Spanish bilingual transfer transducer [13]. (4) Speech represents speech-recognition decoding by composing a LibriSpeech lexicon [38] with an emissions graph that encodes the alignment between phonemes and acoustic frames. Table 3 summarizes the characteristics of the FST pairs in these workloads.

Overall Performance. Figure 11 shows the performance of all schemes on four representative real-world FST benchmarks. RAYFST-TUNED outperforms both CPU and GPU schemes, with average speedups of 7.4 $\times$ over OpenFst, 14.0 $\times$ over k2, and 11.3 $\times$ over GTN. Without parameter tuning, RAYFST still achieves 5.8 $\times$, 11.0 $\times$, and 8.9 $\times$ speedups over the three baselines. On three of four benchmarks, RAYFST remains close to RAYFST-TUNED. On Grammar,

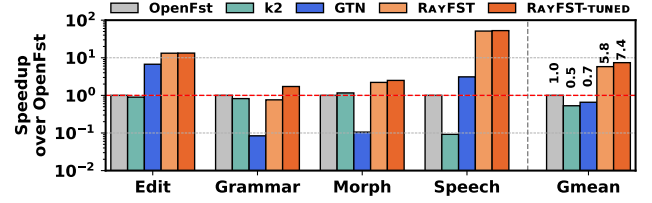


Figure 11: Speedup of the evaluated schemes on real-world FST benchmarks, normalized to OpenFst.

Table 4: Execution time on real-world FST benchmarks (ms). SpdUp: speedup of RAYFST-TUNED over each scheme.

Benchmark	OpenFst		k2		GTN		RAYFST		RAYFST-TUNED
	Time	SpdUp	Time	SpdUp	Time	SpdUp	Time	SpdUp	Time
Edit	173.1	13.3 $\times$	192.7	14.8 $\times$	25.7	2.0 $\times$	13.1	1.0 $\times$	13.0
Grammar	122.9	1.7 $\times$	149.7	2.1 $\times$	1461.9	20.5 $\times$	161.5	2.3 $\times$	71.4
Morph	28.3	2.5 $\times$	24.4	2.2 $\times$	268.3	23.7 $\times$	12.8	1.1 $\times$	11.3
Speech	4546.4	52.7 $\times$	49374.3	572.8 $\times$	1462.3	17.0 $\times$	88.6	1.0 $\times$	86.2
Gmean	—	7.4 $\times$	—	14.0 $\times$	—	11.3 $\times$	—	1.3 $\times$	—

the 2.3 $\times$ gap to the offline-tuned oracle results from its highly skewed degree distribution, although RAYFST remains 9.1 $\times$ faster than GTN. The execution times are summarized in Table 4, where RAYFST-TUNED reduces the runtime from several thousand milliseconds to under one hundred for several benchmarks.

Comparison with CPU Schemes. RAYFST-TUNED consistently outperforms OpenFst on all benchmarks, achieving up to 52.7 $\times$ speedup on Speech. OpenFst performs label matching by sequentially scanning transitions, which becomes costly in Edit and Speech due to the large number of transitions. k2 uses an invert-sort-intersect pipeline and performs well on Morph, but it slows down on Speech because the large transition sets make sorting and intersection expensive. For RAYFST, the speedup on Grammar is lower because its FSTs have uneven out-degrees and the CPU is less sensitive to this irregularity. Even so, RAYFST-TUNED still achieves 1.7 $\times$ and 2.5 $\times$ speedups over OpenFst on Grammar and Morph by generating more rays to balance workloads.

Comparison with GPU Schemes. GTN accelerates composition with thread-level parallelism, but its performance is constrained by irregular memory access and redundant label-matching computation. These costs increase when the inner alphabet is large or when out-degrees vary widely across states. In Edit and Speech, GTN improves over the CPU scheme, while RAYFST gains additional speedup from ray-based transition lookup, which leverages hierarchical BVH traversal. In Grammar and Morph, the large alphabet and uneven out-degrees increase GTN’s matching cost and cause workload imbalance, making it slower than the CPU baseline. RAYFST-TUNED improves performance by generating rays dynamically based on FST characteristics, achieving an average 11.3 $\times$ speedup over GTN.

Runtime Breakdown of RAYFST. We break down the runtime of RAYFST on real-world FSTs to understand where time is spent. We partition execution into four components: FST Preprocessing collects input-FST statistics and prepares the ray-emission map for Dynamic Ray Generation. BVH Construction builds the BVHs for

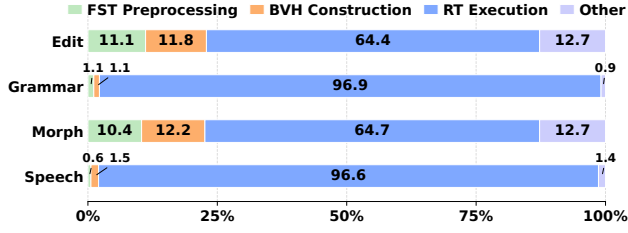


Figure 12: Runtime breakdown of RayFST on real-world FST benchmarks.

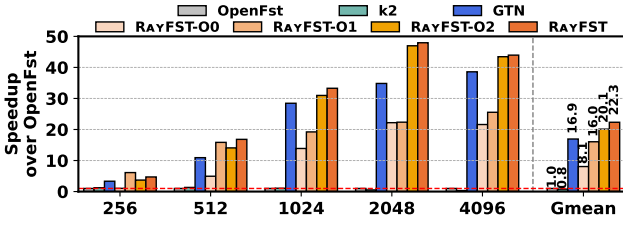


Figure 13: Performance on synthetic FSTs with varying node counts. All FSTs have out-degree 4 and alphabet size 10.

both FSTs. RT Execution runs the ray-tracing kernels for transition lookup and label matching. Other includes buffer allocation, host-device transfers, parameter packing, and synchronization. We find that RT execution is the dominant cost in all workloads, accounting for 64.4%–96.9% of total runtime, as shown in Figure 12. On the larger Grammar and Speech workloads, RT execution exceeds 96% of total time, leaving only 3.1%–3.4% for all non-RT costs. On the smaller Edit and Morph workloads, fixed costs account for a larger share, and FST preprocessing and BVH construction together take about 23% of runtime. Even in these cases, RT execution remains the largest component at about 64%. We observe the same trend as in Figure 11: as workload size grows, setup costs are amortized, and performance depends mainly on RT traversal. We also measure the memory footprint of the BVHs constructed for the two input FSTs. For each FST pair, the two BVHs occupy 2.8, 12.5, 10.1, and 7.0 MB on Speech, Grammar, Edit, and Morph, respectively.

5.3 Sensitivity Studies on Synthetic FSTs

Synthetic FSTs Setup. We generate synthetic FSTs to study how FST structure affects composition performance. We vary the state count, out-degree, and alphabet size one at a time, and evaluate schemes under the same setting. For each setting, we generate a random FST pair in which all states have the same out-degree and all transitions are uniformly distributed over the alphabet. We also study the effect of our optimizations by enabling them incrementally on synthetic FSTs.

Sensitivity to State Count. Figure 13 shows how performance changes as the number of states increases from 256 to 4096. CPU schemes such as OpenFst and k2 improve only slightly because their performance is limited by sequential transition processing. RayFST and GTN achieve higher speedups because larger FSTs expose more

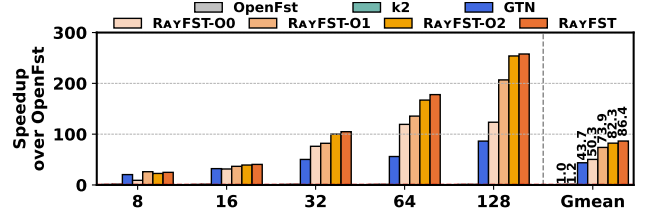


Figure 14: Performance on synthetic FSTs with varying out-degrees. All FSTs have 256 states and alphabet size 10.

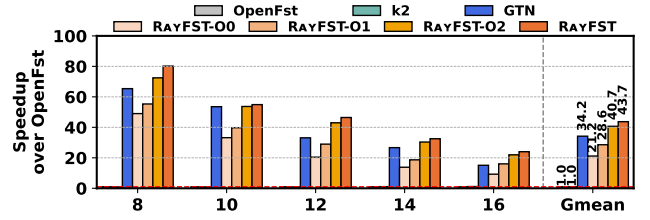


Figure 15: Performance on synthetic FSTs with varying alphabet size. All FSTs have 1024 states and out-degree 4.

parallelism for the GPU. Across all sizes, RayFST remains consistently faster than GTN. This advantage comes from replacing CSR transition lookup with BVH-accelerated ray-primitive intersections, which improve locality and reduce redundant work.

Sensitivity to Out-Degree. Figure 14 shows how performance changes as the out-degree increases from 8 to 128. Larger degrees expand each state’s transitions and increase label-matching cost for CPU schemes such as OpenFst and k2, leading to more execution time. Both GPU schemes improve with higher degrees, but RayFST scales better because Ray Forwarding and Dynamic Ray Generation reduce invalid rays and balance workloads. At degree 128, RayFST reaches about 3.0× speedup over GTN.

Sensitivity to Alphabet Size. Figure 15 shows how performance changes as the alphabet size increases from 8 to 16. As the alphabet grows, label matches become sparser, which reduces the number of composed transitions and lowers execution time for CPU schemes such as OpenFst and k2. For GPU schemes, GTN remains sensitive to redundant query dispatch under sparse matches, while RayFST keeps more stable performance by pruning unmatched candidates during BVH traversal. Across all tested alphabet sizes, RayFST remains consistently faster than GTN.

Optimization Analysis. Figure 13, Figure 14 and Figure 15 show the performance of RayFST on synthetic FSTs with optimizations enabled incrementally. Our observations are as follows: (1) RayFST-O1 outperforms RayFST-O0 on all synthetic benchmarks. By fusing transition lookup and label matching into a single ray-tracing kernel, Ray Forwarding reduces substantial global memory traffic. (2) The benefit of Dynamic Ray Generation depends on FST size. With small state counts or degrees (e.g., 256 states or degree 8), RayFST-O2 can be slightly slower than RayFST-O1 due to preprocessing and binary-search overhead. For larger FSTs, however, the ray-emission map removes many redundant rays, allowing

Table 5: Total FST composition time and end-to-end training-step latency on two FST applications.

Application	Total Composition (ms)			E2E Train Step (ms)		
	GTN	RAYFST	Speedup	GTN	RAYFST	Speedup
ASR	372.8	171.8	2.17×	656.0	455.0	1.44×
HWR	488.3	180.4	2.71×	814.9	507.1	1.61×

RAYFST-O2 to outperform RAYFST-O1. (3) RAYFST achieves performance close to RAYFST-TUNED without tuning. It selects traversal direction and ray-generation strategy at runtime based on FST characteristics and GPU utilization, delivering strong performance across all settings.

5.4 End-to-End Performance on Real Applications

Evaluated Applications. We evaluate RAYFST on two differentiable sequence-to-sequence training tasks from the GTN library [1, 40], where FST composition is executed at every training iteration: (1) An automatic speech recognition (ASR) model trained on LibriSpeech [38]. The encoder is a Time-Depth Separable (TDS) convolutional network [18]. It is trained with a transducer loss that requires four composition operations per step to align the target word sequence with acoustic emissions. The largest intermediate FST in this pipeline reaches up to 255K states and 2.2M transitions. (2) An offline handwriting recognition (HWR) model trained on the IAM database [30]. The encoder is a 2D-TDS network [18] over a 78-character alphabet with a bigram transition model encoded as a compact FST. Each training iteration requires seven composition operations in total: five to construct alignment graphs and normalize output distributions, and two more during decoding to find the most likely character sequence. The largest intermediate FSTs reach up to 44K states and 3.5M transitions. We report the average per-iteration latency (i.e., the execution time of a single training step on a mini-batch) over multiple runs after warm-up.

End-to-End Speedup. Table 5 reports end-to-end per-iteration latency and total FST composition time per iteration for GTN and RAYFST. RAYFST reduces total FST composition time by 2.17× on ASR, from 372.8 ms to 171.8 ms, and by 2.71× on HWR, from 488.3 ms to 180.4 ms. This reduction lowers end-to-end per-iteration latency by 1.44× on ASR, from 656.0 ms to 455.0 ms, and by 1.61× on HWR, from 814.9 ms to 507.1 ms. As Figure 1 shows, FST composition accounts for 56.4% of training time in ASR and 61.7% in HWR. As a result, accelerating FST composition delivers end-to-end speedups and saves more than 200 ms per iteration on both applications.

6 Related Work

This section provides an overview of related work on GPU execution of finite-state automata, and the broader use of GPU ray tracing cores in general-purpose computing.

Finite-State Automata Execution on GPUs. Finite-state transducers extend finite-state automata by adding output labels to transitions. They both face irregular parallelism and load imbalance because the active worklist changes across steps. Previous GPU automata engines map threads to states or transitions, which can leave many threads idle when the worklist is sparse or imbalanced [26, 46].

ngAP improves efficiency with symbol-parallel execution and worklist scheduling that increases locality [14]. Automata processing matches each input symbol against transitions in one machine. FST composition instead matches outgoing transitions from two machines by shared labels. RAYFST offloads this label matching to RT cores through geometric mapping and ray traversal.

Leveraging RT Cores for Non-Graphics Applications. RT cores have recently been repurposed for non-graphics workloads such as neighbor search [27, 29, 35, 45], database indexing [20], spatial indexing [16], spatial join queries [15], and scan [28]. These efforts show that BVH traversal can accelerate irregular search and filtering in non-graphics domains. Xiao et al. [43] further study graph workloads such as BFS and triangle counting, and show that BVH construction cost can limit end-to-end gains. Our work shares the same high-level goal of repurposing RT cores for irregular GPU workloads. However, RAYFST targets label-constrained state expansion rather than generic spatial filtering. RAYFST maps transitions to geometric primitives so RT traversal directly performs candidate matching during composition.

7 Conclusion

This paper presents RAYFST, a ray-tracing-based approach for accelerating FST composition. Existing GPU methods are limited by irregular memory access and redundant label matching. RAYFST addresses these challenges by using RT-core BVH traversal for parallel transition lookup and introducing Ray Forwarding and Dynamic Ray Generation to reduce redundancy and balance workloads. Without offline tuning, RAYFST achieves a geometric-mean speedup of 5.8× over OpenFst, 11.0× over k2, and 8.9× over GTN across four real-world FST benchmarks. On synthetic FSTs, RAYFST reaches up to 3.0× speedup over GTN at high out-degrees by exploiting BVH-based pruning. For end-to-end training, RAYFST reduces total FST composition time by 2.2× on ASR and by 2.7× on HWR, translating into 1.4× and 1.6× per-iteration speedups.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, which significantly improved the quality of this paper. Tianao Ge and Xiaowen Chu were partially supported by the Guangzhou Municipal Joint Funding Project with Universities and Enterprises under Grant No. 2024A03J0616. Corresponding authors: Hongyuan Liu and Xiaowen Chu.

References

- [1] 2018. GTN Applications. https://github.com/facebookresearch/gtn_applications.
- [2] 2018. NVIDIA TURING GPU ARCHITECTURE. <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>.
- [3] 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE. <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf>.
- [4] 2023. RDNA3 Instruction Set Architecture. https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf.
- [5] 2025. k2: FSA/FST toolkit. <https://github.com/k2-fsa/k2>.
- [6] 2025. NVIDIA OptiX 9.0. <https://raytracing-docs.nvidia.com/optix9/index.html>.
- [7] 2026. Embree ray tracing kernels repository. <https://github.com/RenderKit/embree>.
- [8] Cyril Allauzen, Michael Riley, and Johan Schalkwyk. 2009. A generalized composition algorithm for weighted finite-state transducers. In *Proceedings of the 10th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. 1203–1206.

- [9] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA) (Lecture Notes in Computer Science, Vol. 4783)*.
- [10] Arturo Argueta and David Chiang. 2018. Composing Finite State Transducers on GPUs. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2697–2705.
- [11] Paul R. Dixon, Diamantino Caseiro, Tasuku Oonishi, and Sadaoki Furui. 2007. The Titech large vocabulary WFST speech recognition system. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*.
- [12] Bilal Elghadyry, Faissal Ouardi, and Sébastien Vérel. 2021. Composition of weighted finite transducers in MapReduce. *J. Big Data* 8, 1 (2021), 1–15.
- [13] Mikel L. Forcada, Mireia Ginesti-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M. Tyers. 2011. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation* 25, 2 (2011), 127–144.
- [14] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngap: Non-blocking large-scale automata processing on GPUs. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 268–285.
- [15] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS)*. 124–136.
- [16] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2025. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 396–411.
- [17] Kyle Gorman. 2016. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*. 75–80.
- [18] Awni Hannun, Ann Lee, Qiantong Xu, and Ronan Collobert. 2019. Sequence-to-Sequence Speech Recognition with Time-Depth Separable Convolutions. In *Proceedings of the 20th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. 3785–3789.
- [19] Awni Hannun, Vineel Pratap, Jacob Kahn, and Wei-Ning Hsu. 2020. Differentiable weighted finite-state transducers. *arXiv preprint arXiv:2010.01003* (2020).
- [20] Justus Henneberg and Felix Schuhknecht. 2023. RTIndex: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16 (2023).
- [21] Takaaki Hori and Atsushi Nakamura. 2005. Generalized fast on-the-fly composition algorithm for WFST-based speech recognition. In *Proceedings of the 6th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. 557–560.
- [22] Bryan Jurish and Kay-Michael Würzner. 2013. Multi-threaded composition of finite-state-automata. In *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing (FSMNL)*. 81–89.
- [23] Lila Kari, Stavros Konstantinidis, and Steffen Kopecki. 2018. Transducer descriptions of DNA code properties and undecidability of antimorphic problems. *Inf. Comput.* 259, 2 (2018), 237–258.
- [24] Lauri Karttunen. 2000. Applications of Finite-State Transducers in Natural Language Processing. In *Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA) (Lecture Notes in Computer Science, Vol. 2088)*. 34–46.
- [25] Terry Koo, Frederick Liu, and Luheng He. 2024. Automata-based constraints for language model decoding. In *Proceedings of the 1st Conference on Language Modeling (COLM)*.
- [26] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are slow at executing NFAs and how to make them faster. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 251–265.
- [27] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 549–565.
- [28] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yanan Jing, and X. Sean Wang. 2024. RTScan: Efficient Scan with Ray Tracing Cores. *Proc. VLDB Endow.* 17, 6 (2024), 1460–1472.
- [29] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS)*. 14–25.
- [30] Urs-Viktor Marti and Horst Bunke. 1999. A Full English Sentence Database for Off-line Handwriting Recognition. In *Proceedings of the 5th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE Computer Society.
- [31] Daniel Meister, Paritosh Kulkarni, Aaryaman Vasishta, and Takahiro Harada. 2024. HIPRT: A ray tracing framework in hip. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 3 (2024), 1–18.
- [32] Yajie Miao, Mohammad Gawayyed, and Florian Metze. 2015. EESN: End-to-end speech recognition using deep RNN models and WFST-based decoding. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. 167–174.
- [33] Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Comput. Speech Lang.* 16, 1 (2002), 69–88.
- [34] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [35] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th ACM International Conference on Supercomputing (ICS)*. 289–300.
- [36] Tri Nguyen and Michela Becchi. 2024. A Transducers-based Programming Framework for Efficient Data Transformation. In *Proceedings of the 33rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 66–77.
- [37] Yinfei Pan, Ying Zhang, and Kenneth Chiu. 2008. Simultaneous transducers for data-parallel XML parsing. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–12.
- [38] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: an ASR corpus based on public domain audio books. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210.
- [39] Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and Efficient Grammar-Constrained Decoding. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*.
- [40] Shubho Sengupta, Vineel Pratap, and Awni Y. Hannun. 2022. Parallel Composition of Weighted Finite-State Transducers. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 6542–6546.
- [41] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [42] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.
- [43] Zhixiong Xiao, Mengbai Xiao, Yuan Yuan, Dongxiao Yu, Rubao Lee, and Xiaodong Zhang. 2025. A Case Study for Ray Tracing Cores: Performance Insights with Breadth-First Search and Triangle Counting in Graphs. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 2 (2025), 1–25.
- [44] Reza Yazdani, José-Maria Arnau, and Antonio González. 2017. UNFOLD: a memory-efficient speech recognizer using on-the-fly WFST composition. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 69–81.
- [45] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 76–89.
- [46] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 129–140.