# PARALLEL COMPOSITION OF WEIGHTED FINITE-STATE TRANSDUCERS

*Shubho Sengupta, Vineel Pratap* — Facebook AI Reseearch

*Awni Hannun* — Zoom AI

## ABSTRACT

Finite-state transducers (FSTs) are frequently used in speech recognition. Transducer composition is an essential operation for combining different sources of information at different granularities. However, composition is also one of the more computationally expensive operations. Due to the heterogeneous structure of FSTs, parallel algorithms for composition are suboptimal in efficiency, generality, or both. We propose an algorithm for parallel composition and implement it on graphics processing units. We benchmark our parallel algorithm on the composition of random graphs and the composition of graphs commonly used in speech recognition. The parallel composition scales better with the size of the input graphs and for large graphs can be as much as 10 to 30 times faster than a sequential CPU algorithm.

***Index Terms***— finite-state transducers, parallel algorithms, GPUs

## 1. INTRODUCTION

Finite-state transducers (FSTs) are widely used in speech recognition, natural language processing, optical character recognition, and other applications [1, 2, 3, 4]. A primary function of FSTs is to combine information from different sources at different granularities. In speech recognition, for example, FSTs are used to combine word-level language models, phoneme-level lexicons, and acoustic models operating on sub-phonetic states. This combination is done by composing FSTs representing each source.

Composition is arguably the most important operation used with FSTs. It is also one of the most expensive. If the input FSTs have $V_1$ and $V_2$ nodes, and maximum out-degrees (outgoing arcs per node) of $D_1$ and $D_2$, then the composition scales as $O(V_1V_2D_1D_2)$. Furthermore, the heterogeneity of FSTs makes it difficult to implement a parallel composition.

In speech recognition, FSTs are primarily used for inference in the decoder. In these cases, running time and space usage can be greatly improved by either pre-computing the FST compositions or with on-the-fly or lazy implementations [5, 6]. However, with the advent of automatic differentiation with automata and their use at training time [7, 8], constructing the full composed graph is important. This implies an unmet need for an efficient, eager implementation of FST composition.

We propose a SIMD-style parallel algorithm for FST composition on GPUs. Our algorithm supports arbitrary graph structures and allows for $\epsilon$ transitions. We validate our algorithm on two benchmarks: 1) the composition of two random graphs and 2) the composition of a lexicon graph and an emissions graph. The latter approximates an operation commonly used in speech recognition decoding. For large graphs, the GPU composition is between 10 and 30 times faster than a highly optimized sequential implementation.

## 2. RELATED WORK

This work builds on a large body of research in algorithms for efficient FST composition. In some cases, heuristics such as lazy, on-the-fly, or dynamic composition combine graph search with composition precluding the need to construct the full composed graph [9, 5, 10, 6]. However, when using FSTs to train models the complete graph is usually required. Parallel implementations which construct the full composed graph have been developed for multiple CPUs [11, 12, 13]. However, due to differing constraints and opportunities for parallelism, CPU and GPU devices require different implementations.

This work also builds on research in parallel implementations of core graph operations on GPUs. Examples include breadth-first search [14, 15], single-source shortest path algorithms [16], and all-pairs shortest path algorithms [17]. However, little prior work exists exploring FST composition on GPUs. Argueta and Chiang [18] developed a GPU composition without support for $\epsilon$ transitions and which outperformed sequential baselines on a toy machine-translation task. In contrast, our GPU composition allows for $\epsilon$ transitions and improves over sequential baselines in more practical settings.

## 3. PARALLEL COMPOSITION

A weighted finite-state transducer is a graph which maps input sequences $\mathbf{x}$ to output sequences $\mathbf{y}$ with a corresponding score. We denote by $\mathcal{A}(\mathbf{x}, \mathbf{y})$ the score with which the weighted FST $\mathcal{A}$ transduces $\mathbf{x}$ to $\mathbf{y}$. We assume weights are in the log semiring, hence, the score of a path in a transducer is the sum of the weights along the individual edges. However, the following composition algorithms generalize easily to other semirings.

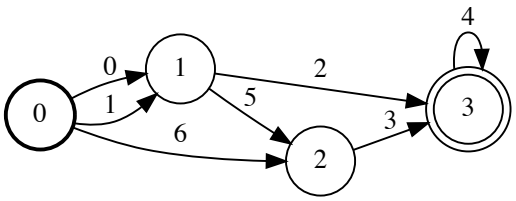



**Fig. 1**. An example FST with edges labeled with their indices into the SoA representation.

### 3.1. Composition

Assume $\mathcal{A}$ transduces $\mathbf{x}$ to $\mathbf{y}$ with score $\mathcal{A}(\mathbf{x}, \mathbf{y})$. Assume also that $\mathcal{B}$ transduces $\mathbf{y}$ to $\mathbf{z}$ with score $\mathcal{B}(\mathbf{y}, \mathbf{z})$. The composition $\mathcal{C}$ of $\mathcal{A}$ and $\mathcal{B}$ transduces $\mathbf{x}$ to $\mathbf{z}$ with the score given by:

$$\mathcal{C}(\mathbf{x}, \mathbf{z}) = \sum_{\mathbf{y}} \mathcal{A}(\mathbf{x}, \mathbf{y}) + \mathcal{B}(\mathbf{y}, \mathbf{z}). \tag{1}$$

Our sequential composition is outlined in algorithm 1. The algorithm yields trim graphs. This means every state is both accessible (*i.e.* reachable from a start state) and co-accessible (*i.e.* can reach an accept state). In order to yield trim graphs, the algorithm proceeds in two stages. The first step is to compute the set of co-accessible states in the composed graph. This is done by calling a subroutine in line 3. The co-accessible subroutine is nearly identical to the main body of algorithm 1, but proceeds backwards from the accept states. The second step is to compute the set of accessible states while only retaining those which are also co-accessible.

### 3.2. Parallel Data Structure

We use a structure of arrays (SoA) layout for the transducer data structure. The SoA layout is more efficient to use with SIMD-style algorithms than the alternative array of structures, which is used for the sequential CPU implementation.

Consider an arbitrary transducer with $V$ nodes and $E$ edges. We store the start and accept status of each node in two arrays with $V$ boolean entries. An example for the graph in figure 1 is below:

```
start   = {T, F, F, F}
accept  = {F, F, F, T}
```

Each node also contains a set of incoming edges and a set of outgoing edges. We store the edge data in five arrays each with $E$ entries containing 1) the input labels, 2) the output labels, 3) the weights, 4) the input node indices, and 5) the output node indices. For each node, the indices of its input and output edges in these five arrays are stored in two more arrays (`inArcs` and `outArcs`). These two arrays are consecutive by node. For example, the array of input arcs starts with the indices of incoming arcs to node 0, followed by the those for node 1, and so on. The starting point for each node in these arrays is stored in two more offset arrays (`inArcOffset` and `outArcOffset`) with $V + 1$ entries. The $v$-th entry states where the $v$-th node's arcs begin, and the entry at $v + 1$ is the index just after where they end (or the start of the entries for node $v + 1$).

An example of how incoming arcs are stored along with the input and output labels for the graph in figure 1 is below:

```
inArcOffset  = {0, 0, 2, 4, 7}
inArcs       = {0, 1, 5, 6, 2, 3, 4}
inLabels     = {a, b, c, d, e, f, g}
outLabels    = {t, u, v, w, x, y, z}
```

Suppose we want to find the input and output labels of the incoming arcs of node 2. We first find the span of the array `inArcs` for node 2 by reading the third and fourth entries of `inArcOffset`. The values of these entries are 2 and 4.

**Algorithm 1** Sequential Composition

1: **Input**: Transducers $\mathcal{A}$ and $\mathcal{B}$
2: Initialize the queue $Q$ and the composed graph $\mathcal{C}$.
3: Compute $R$, the set of co-accessible states in $\mathcal{C}$.
4: **for** $s_a$ and $s_b$ in all start state pairs of $\mathcal{A}$ and $\mathcal{B}$ **do**
5:  **if** $(s_a, s_b)$ is in $R$ **then**
6:  Add $(s_a, s_b)$ to $Q$ and as a start state in $\mathcal{C}$.
7:  **if** $s_a$ and $s_b$ are accept states **then**
8:  Make $(s_a, s_b)$ an accept state in $\mathcal{C}$.
9:  **end if**
10:  **end if**
11: **end for**
12: **while** $Q$ is not empty **do**
13:  Remove the next state pair $(u_a, u_b)$ from $Q$.
14:  **for** all arcs pairs $e_a$ and $e_b$ leaving $u_a$ and $u_b$ **do**
15:  Get the output label $o_a$ of $e_a$.
16:  Get the input label $i_b$ of $e_b$.
17:  **if** $o_a \neq i_b$ **then**
18:  Continue to the next arc pair.
19:  **end if**
20:  Get destination states $v_a$ of $e_a$ and $v_b$ of $e_b$.
21:  **if** $(v_a, v_b)$ is not in $R$ **then**
22:  Continue to the next arc pair.
23:  **end if**
24:  **if** $(v_a, v_b)$ is not in $\mathcal{C}$ **then**
25:  Add $(v_a, v_b)$ as a state to $\mathcal{C}$ and to $Q$.
26:  **if** $v_a$ and $v_b$ are accept states **then**
27:  Make $(v_a, v_b)$ an accept state in $\mathcal{C}$.
28:  **end if**
29:  **end if**
30:  Get the weights $w_a$ of $e_a$ and $w_b$ of $e_b$.
31:  Add an arc to $\mathcal{C}$ from $(u_a, u_b)$ to $(v_a, v_b)$ with label $o_a : i_b$ and weight $w_a + w_b$.
32:  **end for**
33: **end while**
34: **Return**: The composed graph $\mathcal{C}$.

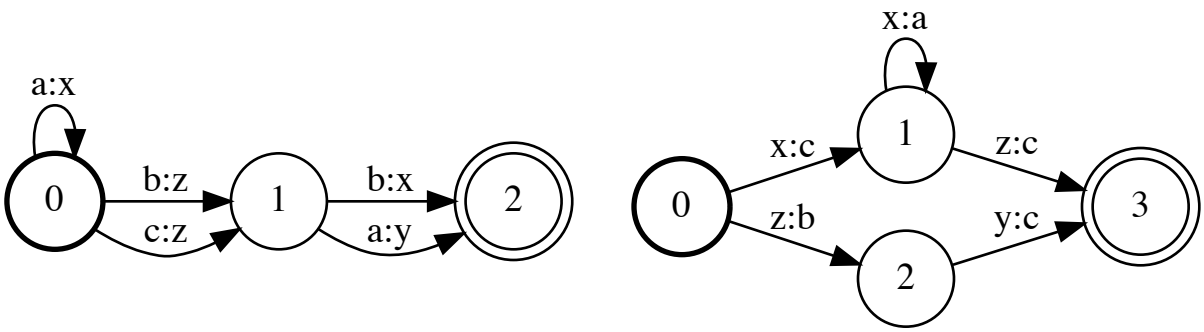



**Fig. 2**. An example of the composition of two transducers. On the first step the queue contains the start state $Q = [(0, 0)]$, and 6 arcs are explored. On the second step, $Q = [(0, 1), (1, 2)]$, and 8 arcs are explored. On the third step, $Q = [(1, 3), (2, 3)]$, and 0 arcs are explored, after which the algorithm terminates. Notice that from the second to third step, the state $(0, 1)$ is reachable, but we do not add it to $Q$ as it has already been considered.

This means that the indices of the input arcs for the second node start at index 2 in `inArcs` and end just before index 4. So we know the input arcs for node 2 have indices 5 and 6 in the arrays storing the input labels, the output labels, and the weights. The input labels for these arcs are `f` and `g`, and the output labels are `y` and `z`.

### 3.3. Parallel Algorithm

At a high-level the primary distinction between the sequential and parallel compose are the loops in lines 12 and 14 of algorithm 1. At each iteration of algorithm 1, the next state pair from the queue $Q$ is removed (line 12). The next loop (line 14) is over the cross product of outgoing arcs for that state pair. The parallel algorithm simultaneously explores all arc pairs for all state pairs currently in the queue. A separate thread is assigned to each arc pair to be explored. The co-accessible subroutine (line 3) is made parallel in the same way. An example demonstrating the states and arcs explored at each iteration of the parallel algorithm is given in figure 2.

Many threads can attempt to modify any of the arrays described in section 3.2 while the composed graph is being constructed. We modify the algorithm to avoid these potential race conditions. After computing the coaccessible states, the parallel implementation performs two passes over all possible accessible states of the composed graph. In the first pass, we compute the number of new nodes in the composed graph along with the number of input and output arcs for each. The offset into the arrays containing edge data for each node is known at this point. On the second pass, the algorithm fills in the correct values for the new arcs in the correct location. In the first pass over accessible states the number of input and output arcs for a node is incremented atomically. Similarly, in the second pass the arc index being written for a given node is also incremented atomically. Access to all other data structures is thread safe and does not require atomic operations or other synchronization primitives.

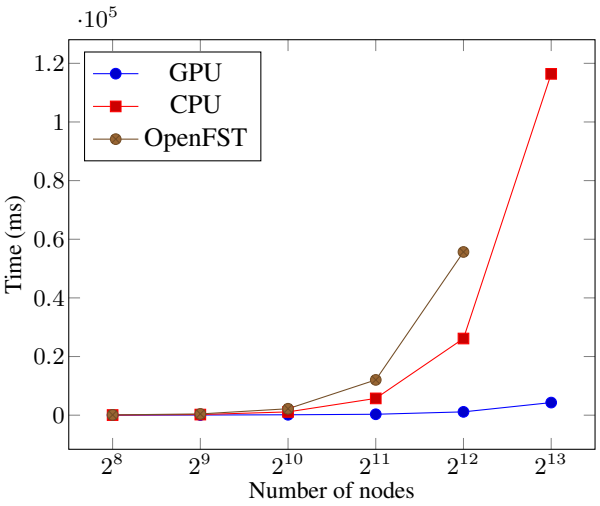
## 4. EXPERIMENTS

In the following, we compare the GTN GPU and CPU implementations as well as an OpenFst CPU implementation [19]. Code to reproduce these benchmarks is open source and available at `https://github.com/awni/parallel_compose`. The CPU and GPU implementation are open source as part of the GTN framework available at `https://github.com/gtn-org/gtn`. The GPU benchmarks are performed on 32GB Nvidia V100 GPUs. The CPU benchmarks are single-threaded and performed on 2.20GHz Intel Xeon E5-2698 CPUs.

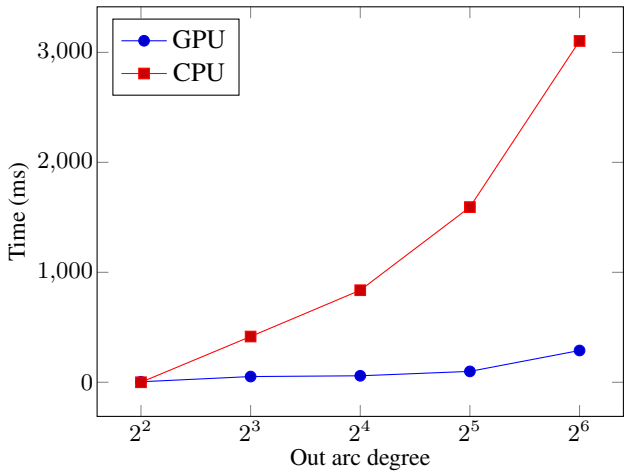
### 4.1. Random Graphs

The random graphs are constructed by first specifying the number of nodes. Each graph has a single start and a single accept state. Outgoing edges from each node are added such that every node in the graph has the same outgoing arc degree. The destination node of each arc is randomly chosen from any of the nodes in the graph including the source node; hence self-loops are allowed. The arc label is randomly chosen from a predefined token set.

Figure 3 compares the performance of the OpenFst CPU and the GTN CPU and GPU compose implementations. In figure 3a, we vary the number of nodes in the input graphs from 256 to 8,192 while keeping the outgoing arc degree and token set size fixed at 5 and 10 respectively. In figure 3b, the outgoing arc degree of each node is increased from 4 to 64 while keeping the number of nodes fixed. Each point in the figures is the mean over several trials, the exact number depending on the size of the graphs. Both the CPU and GPU implementations ultimately scale quadratically in the number of nodes. However, the GPU implementation is much faster at larger graph sizes. When the input graphs have 8,192 nodes, the GPU implementation is nearly 30 times faster than the GTN CPU implementation.




(a) Varying nodes.




(b) Varying arc degree.

**Fig. 3**. A comparison of OpenFst CPU and GTN CPU and GPU composition on random graphs. (a) The number of nodes in the graph are increased while the outgoing arc degree and token set sizes are fixed at 5 and 10 respectively. (b) The outgoing arc degree is increased while keeping the number of nodes fixed at 256. In (b) the number of tokens is set to twice the outgoing arc degree to ensure the composition is not empty.

### 4.2. Composition with a Lexicon

To emulate a more realistic computation, we compose an emissions graph with a lexicon. The lexicon is derived from the LibriSpeech corpus [20] and contains 200,000 word-to-phoneme mappings using 69 phonemes. We intersect this with an emissions graph designed to emulate the output of an acoustic model. The emissions graph is linear with 251 nodes and 69 arcs between each node. The arcs between each node represent the phoneme scores for the corresponding frame. We use 251 nodes as this corresponds to 10 seconds of audio assuming a 10 millisecond step size for acoustic features and a factor of four frame rate reduction in the network. We compose the closure of a graph representing the lexicon with the linear emissions graph. The closure of the lexicon adds $\epsilon$ transitions, so this benchmark requires support for $\epsilon$ in the composition.

Figure 4 compares the CPU and GPU composition while increasing the number of words in the lexicon. The words are randomly sampled without replacement. For a small number of words (1,000) the CPU and GPU run times are comparable, but as we increase the number of words, the GPU composition is much faster. At 32,000 words, the GPU implementation is more than 10 times faster than the CPU implementation.

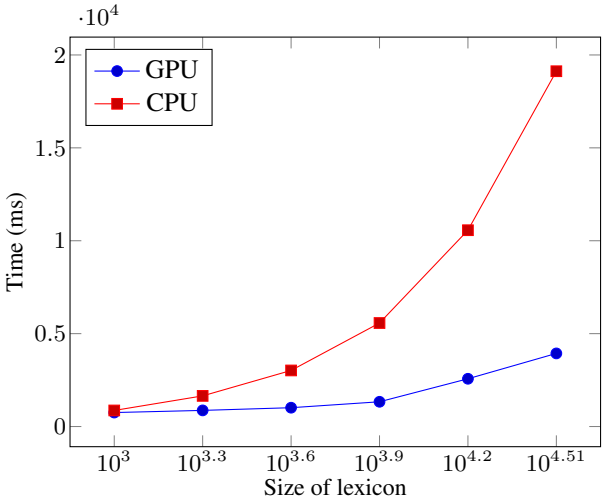



**Fig. 4**. A comparison of the CPU and GPU composition with a lexicon and emissions graph for increasing number of words.

## 5. CONCLUSION

We presented an algorithm for parallel composition on GPUs. Our algorithm handles general FSTs including $\epsilon$ transitions. For large graphs, the parallel composition can be as much as 10 to 30 times faster than a highly optimized sequential algorithm running on the CPU. We intend to continue to refine and optimize the parallel composition. Developing parallel algorithms which can handle $N$-way composition instead of just two inputs may yield further improvements. Other important FST operations include determinize, minimize, epsilon removal, and shortest path algorithms. A fully featured framework for operations on FSTs which can run on GPUs will require parallel implementations of these algorithms. Such a framework has the potential to open the door for new and impactful machine-learning models built from FSTs.

## 6. REFERENCES


[1] Thomas M Breuel, "The OCRopus open source OCR system," in *Document recognition and retrieval XV*. International Society for Optics and Photonics, 2008, vol. 6815, p. 68150F.

[2] Kevin Knight and Jonathan May, "Applications of weighted automata in natural language processing," in *Handbook of Weighted Automata*, pp. 571–596. Springer, 2009.

[3] Mehryar Mohri, "Finite-state transducers in language and speech processing," *Computational linguistics*, vol. 23, no. 2, pp. 269–311, 1997.

[4] Mehryar Mohri, Fernando Pereira, and Michael Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, vol. 16, no. 1, pp. 69–88, 2002.

[5] Takaaki Hori and Atsushi Nakamura, "Generalized fast on-the-fly composition algorithm for wfst-based speech recognition," in *Ninth European Conference on Speech Communication and Technology*, 2005.

[6] Mehryar Mohri, Fernando Pereira, and Michael Riley, "Speech recognition with weighted finite-state transducers," in *Springer Handbook of Speech Processing*, pp. 559–584. Springer, 2008.

[7] Awni Hannun, Vineel Pratap, Jacob Kahn, and Wei-Ning Hsu, "Differentiable weighted finite-state transducers," *arXiv preprint arXiv:2010.01003*, 2020.

[8] "k2," https://github.com/k2-fsa/k2, 2020.

[9] Octavian Cheng, John Dines, and Mathew Magimai Doss, "A generalized dynamic composition algorithm of weighted finite state transducers for large vocabulary speech recognition," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*. IEEE, 2007, vol. 4, pp. IV–345.

[10] Andrej Ljolje, Fernando Pereira, and Michael Riley, "Efficient general lattice generation and rescoring," in *Sixth European Conference on Speech Communication and Technology*, 1999.

[11] Minyoung Jung, Jinwoo Park, Johann Blieberger, and Bernd Burgstaller, "Parallel construction of simultaneous deterministic finite automata on shared-memory multicores," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 271–281.

[12] Bryan Jurish and Kay-Michael Würzner, "Multi-threaded composition of finite-state-automata," in *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing*, 2013, pp. 81–89.

[13] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 529–542.

[14] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[15] Duane Merrill, Michael Garland, and Andrew Grimshaw, "Scalable gpu graph traversal," *Acm Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[16] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 349–359.

[17] Pawan Harish and Petter J Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[18] Arturo Argueta and David Chiang, "Composing finite state transducers on gpus," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 2697–2705.

[19] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri, "OpenFst: A general and efficient weighted finite-state transducer library," in *International Conference on Implementation and Application of Automata*. Springer, 2007, pp. 11–23.

[20] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2015, pp. 5206–5210.