Transliterated Mobile Keyboard Input via Weighted Finite-State Transducers

Lars Hellsten[†], **Brian Roark**[†], **Prasoon Goyal**[*], **Cyril Allauzen**[†], **Françoise Beaufays**[†], **Tom Ouyang**[†], **Michael Riley**[†] **and David Rybach**[†]
[†]Google, Inc. [*]New York University
{lhellsten,roark,allauzen,fsb,ouyang,riley,rybach}@google.com
prasoongoyal13@gmail.com

Abstract

We present an extension to a mobile keyboard input decoder based on finite-state transducers that provides general transliteration support, and demonstrate its use for input of South Asian languages using a QWERTY keyboard. On-device keyboard decoders must operate under strict latency and memory constraints, and we present several transducer optimizations that allow for high accuracy decoding under such constraints. Our methods yield substantial accuracy improvements and latency reductions over an existing baseline transliteration keyboard approach. The resulting system was launched for 22 languages in Google Gboard in the first half of 2017.

1 Introduction

The usefulness of weighted finite-state transducers (WFSTs) has been well-documented for speech recognition decoding. Large component WFSTs representing a context-dependent phone sequence model (C), the pronunciation lexicon (L) and the language model (G) can be composed into a single large transducer ($C \circ L \circ G$, or CLG for short) mapping from context-dependent phone sequences on the input to word sequences on the output and optimized for efficient decoding (Mohri et al., 2002). In addition to forming the basis of numerous commercial and research speech recognition engines, this is the approach taken by the widely-used open-source toolkit Kaldi (Povey et al., 2011), which makes use of the OpenFst library (Allauzen et al., 2007) to represent and manipulate the WFSTs. Decoding via such an optimized graph permits the efficient combination of acoustic model scores of context-dependent phone sequences with the scores associated with larger (word n-gram) sequences contributed by the language model.

Speech is not the only uncertain input sequence modality requiring transcription to yield word strings – others include optical character recognition (OCR) and handwriting recognition. WFST-based methods have also recently been applied to soft keyboard decoding (Ouyang et al., 2017), where a sequence of taps or continuous gestures is decoded to the most likely word or word sequence. Unlike typing on a standard physical keyboard (such as a QWERTY keyboard on a laptop), ambiguity arises on a soft (on-screen) keyboard due to the relatively small size of the keyboard (e.g., on a smartphone) and the resulting imprecision of the tap or gesture. In such an approach, the same sort of off-line composition of weighted FSTs provides low latency decoding with a modest memory footprint, which is essential for on-device keyboard functionality. The FST-based decoder described in that paper was launched in the Google Gboard keyboard system in early 2017.

In this paper, we present methods for extending this finite-state decoding approach for mobile keyboard input to transliterated keyboards, where the keyboard representation differs from the output script. One very common scenario of this sort is the use of a standard QWERTY-style soft keyboard for entering text in a language with another writing system, typically because the Latin alphabet is simpler to represent on a compact soft keyboard. Systems for mapping from sequences of symbols in the target script to sequences of Latin symbols are known as *romanization*.

The most widely known romanization system is *Pinyin*, which is a fully conventionalized system for Chinese. For example, the word 水 (water) in Chinese is written as "shuǐ" in Pinyin. Romanization, however, is quite widely used around the world, with such writing systems as Arabic, Cyril-

Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing, pages 10–19,
Umeå, Sweden, 4–6 September 2017.
https://doi.org/10.18653/v1/W17-4002

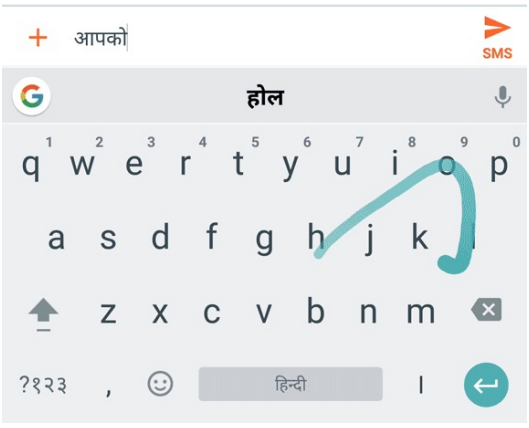

Figure 1: Screen capture of a transliteration keyboard with a user's gesture trace for the target word.

lic, Greek and Thai. In many cases, unlike Chinese, there is no agreed upon standard romanization system, leading to an increase in ambiguity and noise when decoding to the target words in the native script. In this paper, we present a general transliteration approach applied to South Asian languages, transliterated from romanized input to a number of scripts, including Devanagari, Bengali and Perso-Arabic. Figure 1 shows a screenshot of a mobile transliteration keyboard for Hindi with the Devanagari script, showing the trace of the user's input gesture. While the user inputs romanized sequences, suggested word completions, as well as output strings, are in the target script.

In this context, we find that a WFST encoding of transliteration models allows for several optimizations that yield good accuracy under the strict resource and operating constraints of the on-device keyboard decoding. In what follows, we first provide background and preliminaries on Indic languages and their various scripts, as well as finite-state transducer terminology that will be used. We then describe our transliteration modeling approach and a number of WFST optimizations that we perform to achieve the accuracy, latency and memory usage operating points. Finally we present some experiments demonstrating the impact of the optimizations and comparing to an existing baseline. On Hindi and Tamil validation sets, we demonstrate strong word error rate and latency reductions versus an existing baseline. We conclude by presenting the various languages and associated scripts for which transliteration keyboards using this approach have so far been launched in Google Gboard, and by discussing future directions of this work.

2 Background and preliminaries

2.1 Indic scripts

While our approach is general enough to be broadly applicable to any transliteration pair, it is useful to have a running example, and since we later evaluate on South Asian languages, it makes sense to review Indic (or Brahmic) scripts here. We lack space to provide a full treatment of the scripts, but will provide details sufficient to understand our overall approach. We refer the interested reader to Sproat (2003) for further details.

Brahmic scripts are a class of writing systems that have descended from the Brahmi script and hence share certain characteristics. Well-known Brahmic scripts include Devanagari (the standard script for many languages, including Hindi and Sanskrit), Bengali, Gujarati and Tamil. Some South Asian languages have non-Brahmic native scripts, such as those that use the Perso-Arabic script (e.g., Urdu) and others that use modern scripts not descended from Brahmi (e.g., Ol Chicki for Santali). Still, Brahmic scripts are pervasive in South Asian languages, hence a key focus for keyboard entry in these languages.

While the Brahmic scripts can look very distinct, they have some central shared characteristics. The scripts are organized around what is termed an orthographic syllable (*akṣara*), which groups one or more consonants together with associated vowels. Each consonant has an inherent vowel, and other vowels – or different qualities of the vowels, such as nasalization, or also the absence of the inherent vowel – are indicated for that consonant via a mark or diacritic on the consonant. In most Brahmic scripts, vowels that are used independently of a consonant – e.g., word initially – have their own symbols. Finally, sequences of multiple consonants can be combined via ligatures.

For example, take the word "Sanskrit", written संस्कृत in Hindi. The initial syllable सं consists of the consonant स (s) which comes with its inherent vowel (sa), and an *anusvara* diacritic indicating that it is nasalized (san). This is followed by स्कृ (skr) which consists of four pieces combined into a single ligature: (1) a consonant स (s) with (2) a *virama* diacritic canceling its inherent vowel स्, followed by (3) another consonant क (k), which combines into the ligature स्क (sk)[1] plus (4) a vocalic 'r' diacritic, to yield the full glyph. Finally,

[1]Note the ligature omits the now-redundant *virama*.

the consonant त (t) completes the full word. Note that this word is encoded as a string of seven Unicode codepoints: स ◌ं स ◌् क ◌ृ त, and it is up to a rendering system to assemble these symbols into the appropriate two dimensional glyphs. See Sproat (2003) for more details.

2.2 Romanization and transliteration

Transliteration – converting from one writing system to another – is a widespread sequence-to-sequence mapping problem that arises in multiple contexts. For example, proper names must be represented in various writing systems, so transliterating names and places can be very important for translation or querying knowledge bases. While it is true that प्रणब मुखर्जी is the president of India and was born in पश्चिम बंगाल, it is more useful for those who do not read the Devanagari script to be presented with the information that Indian president Pranab Mukherjee was born in West Bengal. Hence, much work in transliteration is focused on translation and information retrieval (Knight and Graehl, 1998; Chen et al., 1998; Virga and Khudanpur, 2003; Haizhou et al., 2004; Gupta et al., 2014).

Knight and Graehl (1998) took pronunciation as a mediating variable in mapping between the two writing systems, and explicitly modeled grapheme-to-phoneme and cross-lingual pronunciation mapping in their model. For example, the probability of mapping to "Sanskrit" from संस्कृत would include probabilities for English pronunciation given the written form (e.g., `S AE N S K R IH T`), the Hindi pronunciation (`S AH N S K R AX T`) given the English pronunciation, and the Devanagari string given the Hindi pronunciation.

Haizhou et al. (2004) took a more direct modeling approach, simply modeling the observed mapping between one writing system representation and the other, with strong gains in accuracy. Under such an approach, transliteration is very similar to grapheme-to-phoneme recognition (g2p), where written representations of words (e.g., "sanskrit") are converted to pronunciations, represented by phone symbols ("`S AE N S K R IH T`" in the ARPAbet representation). This sort of pronunciation modeling is also a well-studied problem, being very important to both text-to-speech and speech-to-text, for deriving pronunciations of items that are out-of-vocabulary. A number of approaches have been used to model this conversion from one symbolic representation (letters) to another (phones) falling in rough monotonic alignment, including log linear models (Wu et al., 2014) and neural sequence models (Rao et al., 2015). Explicit finite-state methods such as Bisani and Ney (2008) and Novak et al. (2013; 2015) have been shown to be very competitive for g2p (Wu et al., 2014; Rao et al., 2015), and we adopt such methods here (see Section 3.1).

Romanization is the special case of mapping from other writing systems to the Latin script, often to permit easier keyboard input. As we can see from the Hindi example in the last section, the number of possible symbols in Brahmic scripts is large, due to the combinations of consonants and vowels via diacritics, and the multi-symbol ligatures. While keyboard layouts do exist for these scripts, transliteration from romanized input to the target Brahmic script is a common form of keyboard input, especially for mobile devices. As mentioned earlier, there are many romanization systems for these languages (in contrast to broadly conventionalized Pinyin systems), making this sort of transliteration keyboard challenging for Indic languages (Ahmed et al., 2011).

2.3 Weighted finite-state transducers

A weighted finite-state transducer $T = (\Sigma, \Delta, Q, I, F, E, \mathbb{K})$ consists of: input (Σ) and output (Δ) vocabularies; a finite set of states Q, of which (without loss of generality) one is the initial state I, and a subset of states $F \subseteq Q$ are final states; a weight semiring $\mathbb{K}$; and a set of transitions $(q, \sigma, \delta, w, q') \in E$, where $q, q' \in Q$ are, respectively, the source and destination states of the transition, $\sigma \in \Sigma$, $\delta \in \Delta$ and $w \in \mathbb{K}$. A weighted finite-state automaton is a special case where $\Sigma = \Delta$ and, for every transition $(q, \sigma, \delta, w, q') \in E$, $\sigma = \delta$. For the work in this paper, we make use of the OpenFst library (Allauzen et al., 2007) to encode and manipulate WFSTs, and, unless otherwise stated, use the tropical semiring for weights.

Encoding n-gram language models as WFSTs involves the use of failure-transitions (Allauzen et al., 2003) with a particular 'canonical' structure that corresponds to backoff smoothing. We use such an encoding for building word-based language models in the target language of the keyboard application, and also as an intermediate representation in the training of our transliteration

transducer, as described in Section 3.1. For this language model training and encoding, we make use of the OpenGrm n-gram library (Roark et al., 2012), which provides counting, smoothing and pruning functions resulting in an OpenFst encoded model.

In speech recognition, a pronunciation lexicon, consisting of words found in the vocabulary of the n-gram model along with their pronunciations, can be compiled into a WFST with input vocabulary Σ of phones and output vocabulary Δ of words. When this lexicon L is composed with the n-gram model G, various optimizations can be carried out on the resulting transducer, to share structure and accrue costs as early as possible (Mohri et al., 2002). Similar optimizations are possible for keyboard models, where the input vocabulary Σ of the lexicon L is the letters that spell the words (see section 3.3).

2.4 WFST-based keyboard decoding

There are some differences between speech and keyboard decoding, which are presented in detail in Ouyang et al. (2017), and which we summarize here. The need for decoding arises in keyboard due to so-called 'fat finger' errors, where the actual point on the keyboard that was touched is not within the bounding box of the intended letter. The term for the sequence of letters corresponding to actual touch points is 'literal'. For example, on a mobile device, with a relatively small QWERTY keyboard, the literal string typed may be "sabsjriy" when the intended word was "sanskrit". In that example, three letters (b,j,y) adjacent to the intended letters (n,k,t) were touched. For possible intended keys, a spatial model provides likelihoods for touch points, via, for example, a Gaussian for each key centered at its central point, or perhaps a neural network learned from real touch data. As with speech, there is contextual dependency between adjacent keys, e.g., the distribution over touch points for a given key may depend on the previous key typed. The decoder takes as input a sequence of touch inputs on the device and combines the context dependent spatial model scores (analogous to the acoustic model in speech) with language model scores to determine the most likely intended word.

One key difference between speech and keyboard decoding is the privileged nature of the literal string in keyboard decoding. Free entry of text via a keyboard is by its nature an open-vocabulary interface, i.e., the user should be able to type whatever they want. The keyboard decoder must decide whether the most likely word according to its models is sufficiently more likely than the literal to warrant auto-correction. If the margin between the score of the best scoring candidate and the literal score falls below the margin required for correction, the literal is output, thus allowing for out-of-vocabulary (OOV) items to be typed.

In addition to touch typing, the WFST-based keyboard decoder also permits gesture input (Zhai and Kristensson, 2003), which involves tracing a path from the first letter of a word through all the letters to the final letter before lifting the finger. With gesture, there is no natural notion of literal string, since multiple words may have the same path, e.g., 'or' and 'our' on a QWERTY keyboard. As a result, OOV processing is restricted, much as with speech, and users must revert to touch typing to input OOVs. The same decoder, however, is used to combine scores from a gesture spatial model with the language model to find the most likely word string.

We train a weighted transducer to provide likely transliterations for unseen words and permit non-canonical transliterations for existing words, increasing uncertainty in the decoder. This is a particular challenge given the strict constraints that come with on-device keyboard modeling: latency can be no more than 20 msec, models must, in aggregate, be on the order of 10MB in size, and memory usage during decoding is strictly constrained.

3 Methods

3.1 Pair language models

Due to the latency and memory constraints of our keyboard application, we pursued methods that resulted in models that could be encoded in relatively compact WFSTs. As mentioned in section 2.2, explicit finite-state methods used for grapheme-to-phoneme conversion, such as Bisani and Ney (2008) and Novak et al. (2013; 2015) are very competitive. The starting point for all such methods is a simple per-symbol alignment of both the input string and the output string. Thus, for example, the word "phlegm" is pronounced `F L EH M` (again using the ARPAbet representation), and one natural alignment between the grapheme and phoneme sequences is: p:ϵ h:`F` l:`L` e:`EH` g:ϵ m:`M`

For transliteration, we can align a Devanagari

script Hindi word (संस्कृत) with its romanization (sanskrit or sanskrt), where we make use of the Unicode symbol sequence (स◌ंस◌्क◌ृत):

s:स a:ϵ n:◌ं s:स ϵ:◌् k:क r:◌ृ i:ϵ t:त

Note that symbols on either the input or the output may not directly correspond to a symbol on the other side (such as 'a', 'i' and ◌् in the above example), which we represent with an ϵ on the other side of the transduction. Given a lexicon of words and their pronunciations or transliterations (e.g., that 'sanskrit' is a romanization of संस्कृत), expectation maximization (EM) can be straightforwardly used to learn effective alignments of this sort.

Given an aligned sequence of input:output 'pair' symbols such as e:EH, we can build an n-gram model to produce joint probabilities over sequences of such pairs. We refer to these models as pair language models, though they are alternatively called joint multi-gram models (Bisani and Ney, 2008). By conditioning the probability of these input:output mappings on the prior context, the model appropriately conditions the probability of h:F on whether the previous mapping was p:ϵ. As stated above, results have shown these models yield very similar performance to more complex and compute-intensive modeling methods (Wu et al., 2014; Rao et al., 2015), and they can be directly encoded as WFSTs (Novak et al., 2013; Wu et al., 2014), making them excellent candidates for low-resource, low-latency keyboard decoder models.

One useful variant of these models are those that reduce or eliminate input and/or output ϵ symbols (insertions/deletions), by allowing the merger of multiple symbols on either the input or the output side. For example, for our pronunciation example above, we may derive an alignment ph:F instead of p:ϵ h:F, which can also be learned using EM (Novak et al., 2013; Novak et al., 2015). Given the structure of WFST-based language models (see section 2.3), which make use of ϵ-transitions to encode backoff, reducing the number of deletions and insertions is important when reducing epsilon cycles in the WFST.

3.2 Transliteration WFST optimization

Figure 2 shows a fragment of the bigram pair language model encoded as a WFST, with model weights not shown for clarity reasons. It is a bigram model, so the incoming arcs into any state are labeled with the same pair symbol, e.g., state 5 has two incoming arcs, both labeled with k:क. The weights on each arc are the (typically negative log) probabilities of the pair symbols on the arcs given the state.

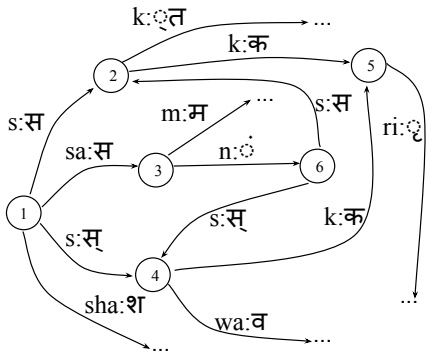

Figure 2: Fragment of the topology of a bigram pair language model, weights omitted for clarity.

This WFST is an automaton over pair symbols, but we need a transducer mapping between Unicode codepoints on the input (QWERTY keyboard letters) and output (target script). If these were single symbol pairs, then converting to a transducer would be trivial: just change from a pair (encoded) symbol (k:क) to an input symbol (k) and an output symbol (क). However, as stated earlier, we merge symbols to reduce the number of insertions or deletions[2], as well as to increase the parameterization of the model, so the conversion to transducer is slightly more involved.

We illustrate our approach by considering the four arcs leaving state 1 in the pair language model automaton in Figure 2, all of which start with an 's' on the input side. Two arcs leaving state 1 have symbols consisting of multiple input Unicode symbols ('sa' and 'sha'); and one of the arcs leaving state 1 has a symbol consisting of multiple output Unicode symbols (स् which is two symbols स and ◌्). Figure 3 shows transducers that map these four pair symbols to their corresponding input and output strings. Figure 3a shows a transducer (call it I) with pair language model symbols (disambiguated with parentheses just for ease of visualization) on the output side and QWERTY key sequences on the input side. Figure 3b shows a transducer (call it O) with pair language model symbols on the input side and Devanagari Unicode symbols on the output side. For a pair language model P, we can derive our transliteration transducer T

[2]We also restrict the transducer to one insertion or deletion in a row, to avoid epsilon cycles.

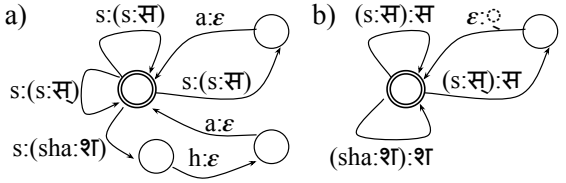

Figure 3: Fragments of transducers mapping a) from input Unicode symbols to symbols in the pair language model, and b) from symbols in the pair language model to output Unicode symbols. Parentheses in symbols are just to assist disambiguation in the figure.

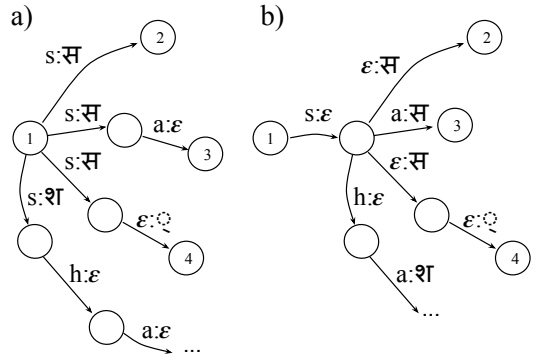

Figure 4: Illustration of conversion of pair language model to Unicode symbol transducer, showing state 1 of Figure 2: a) shows a direct conversion into single Unicode symbol transducer; b) shows a determinized structure which allows structure sharing and proper weight pushing. Numbered states are the same as in Figure 2 and unnumbered states are new.

by composing these together: $T = I \circ P \circ O$. The resulting transducer looks like the fragment shown in Figure 4a, where the original destination states from the pair language model in Figure 2 retain their numbers and new states are unnumbered. This is a simple conversion, with no structure sharing among arcs leaving the state, and the original model weights are accrued upon leaving state 1.

An alternate construction can be used, resulting in a transducer with structure like that shown in Figure 4b. This involves determinizing the input side of the transducer in Figure 3a mapping between input strings and pair language model symbols. As there are two paths with "s" on the input side (which is also a prefix of the other strings) this is not determinizable as is, much like pronunciation lexicons are not generally determinizable. We use a variant of the method presented in Mohri et al. (2002) to permit determinization in these cases. In their algorithm, they added an auxiliary phone symbol to the end of each pronunciation that indicated the word identity. In our case, as seen in Figure 5a, we create a transducer with each input symbol paired with ϵ, followed by an extra arc that pairs an input ϵ with the full pair symbol. We then encode the transducer (treating the input:output as a a single symbol), determinize it, then decode it again to a transducer, resulting in the transducer in Figure 5b. We further reduce the size of this by combining sequential arcs labeled with x:ϵ and ϵ:y into a single arc x:y in cases where the intermediate state has only one incoming and one outgoing arc. This results in the transducer in Figure 5c, which we can use in lieu of the transducer in Figure 3a to produce the final transducer, as illustrated in 4b.

While the structure sharing that results from this weighted determinization is important, its most important feature is the weight pushing that results. When building the encoded automaton in Figure 5a, we put the pair language model weight from the model on the first arc of each path, i.e., the arcs labeled S:ϵ in this example. When that automaton is determinized, the weights get pushed along the path, leaving only the minimum cost over all paths for which the transition is a prefix. This correct weight pushing of the transliteration cost yields major benefit during decoding.

One complication to keep in mind is that the transliteration cost provided by this model is based on a joint probability over input/output relations, not a conditional probability of the input given the output, which is what is needed (see Section 3.4).

3.3 L ∘ G optimization

In speech recognition, the $L \circ G$ transducer has phones on the input side and words on the output side. Unlike speech, however, in the current case the input symbols are individual Unicode symbols in the target script, the concatenation of which spells out the target word. Since the words can be inferred from the input symbols, we do not need to store the word labels explicitly; we project $L \circ G$ to its input labels (with an appropriate word boundary symbol to determine where such boundaries occur), and the decoder outputs concatenations of character label strings. Fig. 6 shows the structure of the optimized $L \circ G$ automaton.

The construction of $L \circ G$ is as follows. Starting with G, replace each non-backoff arc – i.e., arcs with word labels rather than ϵ – with a linear path corresponding to the word's symbols followed by the word end symbol (here </w>). Common prefixes of each such path leaving a state are merged to obtain a trie, backoff (ϵ) arcs are retained un-

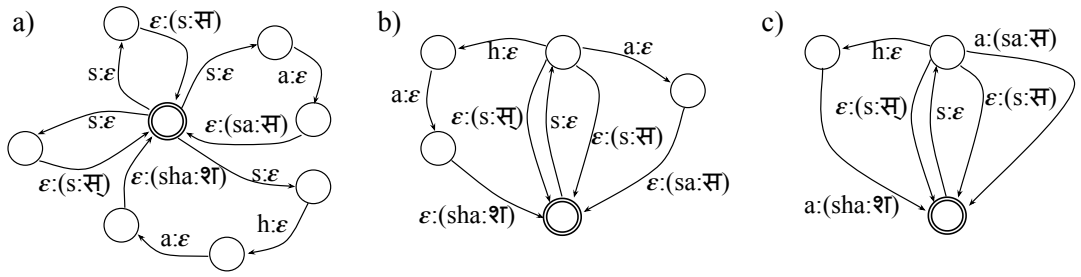

Figure 5: Fragments of transducers mapping from input Unicode symbols to symbols in the pair language model: a) with extra transition allowing for determinization; b) determinized on the encoded symbols; and c) combining sequential output and input ϵ transitions. Parentheses are just to assist disambiguation in the figure.

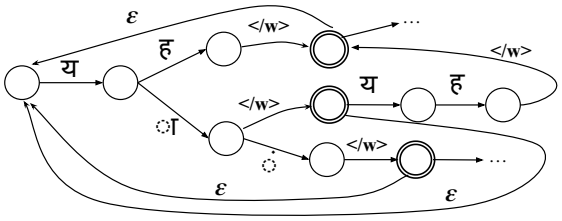

Figure 6: Illustration of the topology of $L \circ G$, for a fragment beginning at the unigram state containing the words या, यां, and यह, with the word end delimiter </w>. Weights are omitted.

changed, and weights are 8-bit quantized. Finally, the automaton is encoded using a variation of the LOUDS-based representation in Sorensen and Allauzen (2011). The topology of $L \circ G$ does not permit the use of the context trees from that representation. Instead, we store explicit 32-bit next state IDs for each backoff arc, indexed by rank in an indicator bit vector over states. We also store a bit vector indicating which states have a context transition arc, which must be an outgoing arc with the word end label. To find the destination state for these arcs, we recursively follow backoff arcs from the most recent word boundary state in the $L \circ G$ traversal until we're able to successfully find a context match.

Note that this structure is suboptimal in state/arc size, since common suffixes (tails) are not merged. But the topology allows for the LOUDS-based encoding which is more compact than a general compaction method could achieve on a more minimal but less regular topology.

3.4 Transliteration cost normalization

One key complication is the need for a conditional (not joint) transliteration model. The overall joint model of the words, transliterations, and touch points must be broken down into probabilities of: (1) the words (language model G); (2) the transliteration *given* the words; and (3) the touch points given the romanized symbols. Yet the transliteration model T is a joint model of aligned romanized and word strings. Thus each word requires an additional normalization factor.

We achieve this by dividing the probability of each w output by $L \circ G$ by the marginalization sum:

$$\mathcal{N}_w = \sum_{\boldsymbol{s},\boldsymbol{t} \in T | \mathbf{concat}(\boldsymbol{t}) = w} P_T(\boldsymbol{s}, \boldsymbol{t})$$

where $\boldsymbol{s}$ and $\boldsymbol{t}$ are input and output sequences in the source and target scripts, respectively. Notice that in the log semiring, $\mathcal{N}_w$ is simply the weight of the shortest path in T that outputs an encoding of w. Thus, we project $T \circ L$ to output labels, determinize, invert weights, and compose the result with L to get a "conditional lexicon" L', such that $T \circ L' \circ G$ models the joint probability of a sequence of words and a corresponding transliteration.

We distribute this normalization factor via weighted determinization when building the trie leaving each language model state, as discussed in Section 3.3 and shown in Figure 6. Similar to the weight pushing that we get when determinizing the transliteration transducer, we put the negative normalization cost on the first arc of each path, then determinize, which distributes the weights correctly, and finally take the negative again to yield the correct normalization factor, which is now accrued incrementally at each character. Finally, to achieve full weight pushing in the $L \circ G$, we push the language model weights.

In addition to decoding sequences of taps and gestures, our keyboard application implements functionality that predicts the most likely completions and next word predictions (Ouyang et al., 2017). These require LM probabilities, which we can extract from $L \circ G$ by traversing arcs recursively until an arc with the word end symbol is en-

countered. However, note that the $L' \circ G$ FST described above has the transliteration normalization factor built into the cost. Thus we require a further modification to the next word prediction algorithm to remove the normalization factors: we store the normalization factors in a LOUDS trie (Delpratt et al., 2006) over the lexicon, and adjust the next word prediction costs correspondingly.

3.5 Decoder optimization

For conventional models, our keyboard decoder (Ouyang et al., 2017) uses on-the-fly composition of $(C \circ L)$ (statically composed) and G using look-ahead composition filters (Allauzen et al., 2009). Because of the ambiguity of transliteration, a statically composed $C \circ T \circ L$ is typically too large for a mobile application. For example, in a conventional Hindi model, $C \circ L$ contained fewer than 500k arcs, while incorporating T resulted in 35M arcs, requiring over 800 MB of storage. We addressed this by refactoring the composition so that a static $C \circ T$ is composed on-the-fly with a static $L \circ G$. With the optimizations discussed in previous sections, the former consists of approximately 100k arcs (1.7 MB), and the latter approximately 3M arcs (6.7 MB), which both fit within our constraints.

While the decoder graph optimizations achieve good model size characteristics, the overall number of states and arcs is much higher than in conventional models. If we use the same decoder meta-parameters as we do for a non-transliteration model, the decoder will search many more states, increasing latency and memory use. We employed the following optimizations, relative to non-transliterated models, to remain within our desired constraints:

Reduced beam width. The decoder prunes low-scoring hypotheses more aggressively.

Stricter error correction model. The decoder supports correction of insertion, substitution, and deletion errors (Ouyang et al., 2017). This model is assumed to be independent from the transliteration model, whereas in fact there is some overlap. For instance, doubling or interchanging vowels is one common source of transliteration ambiguity. We lowered probabilities in the edit model to help offset this.

Stricter spatial model. We used a slightly lower Gaussian variance in the tapping model, optimized for WER by sweeping over settings.

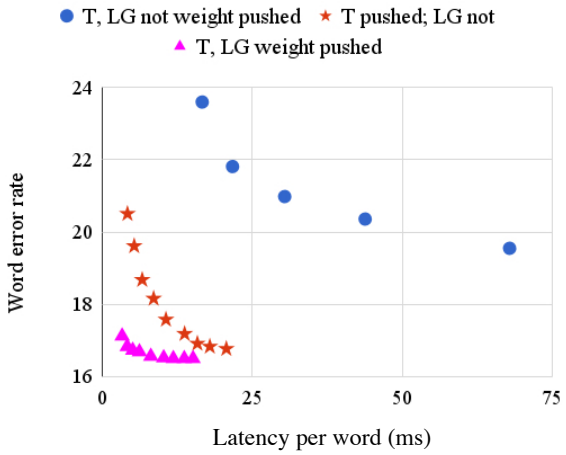

Figure 7: Sweep over beam sizes on Hindi dev set before and after transliteration transducer T weight pushing and $L \circ G$ weight pushing.

4 Experiments

4.1 Data

As is typical with non-transliterated systems, we use low order n-gram language models over a limited vocabulary. For the current study, vocabulary was 150k words and models were pruned trigrams with 750k n-grams.

Transliteration model training data consists of individual word transliterations for each native word in the vocabulary, collected from a small number (approximately 5) of speakers drawn from a larger overall pool of speakers. Each pair of (native word, transliteration) constituted one training instance used by the pair LM method described in Section 3.1. We ensured, using random sampling where necessary, that all words in the native vocabulary had the same number of training instances to avoid introducing bias.

Performance was evaluated on blind test datasets of tapped and gestured sentences in Hindi and Tamil (over 3000 words total for each input modality and language). In the data collection study, participants tapped and gestured sentences written in the native script on a smartphone QWERTY keyboard that produced no output, using their preferred transliteration. The prompts were sentences sampled from transcribed speech interactions. Participants were asked to type at a natural pace and not worry about fixing any mistakes, to collect natural errors and test the correction capability of the decoder and robustness of the transliteration modeling. We evaluate word error rate (WER), which is the number of errors divided by the number of reference words,

	$L \circ G$ size (MB)		
Language	standard	compact	LOUDS
Hindi	85.3	26.3	6.72
Tamil	126.5	37.1	9.80

Table 1: Size savings via LOUDS $L \circ G$ construction.

	Hindi		Tamil	
System	WER	LPC	WER	LPC
Baseline HMM system	19.5	3.0	26.2	2.1
WFST-based	16.4	0.5	22.6	0.2

Table 2: Word error rate (WER) and latency per character (LPC) results on blind test sets.

Language	Script	Language	Script
Assamese	Bengali	Manipuri	Manipuri
Bengali	Bengali	Marathi	Marathi
Bodo	Bengali,	Nepali	Nepali
	Devanagari	Odia	Odia
Dogri	Devanagari,	Punjabi	Gurmukhi,
	Perso-Arabic		Perso-Arabic
Gujurati	Gujurati	Sanskrit	Devanagari
Hindi	Devanagari	Santali	Ol Chiki
Kannada	Kannada	Sindhi	Devanagari,
Kashmiri	Devanagari,		Perso-Arabic
	Perso-Arabic	Tamil	Tamil
Konkani	Devanagari	Telugu	Telugu
Maithili	Devanagari	Urdu	Perso-Arabic
Malayalam	Malayalam		

Table 3: Languages and scripts launched in GBoard using the methods presented in this paper.

presented as a percentage.

The baseline system in our experiments comes from the Google Indic Keyboard, an existing product with between 50 and 100 million installations from the Google Play Store. It uses an HMM-based decoder, with a standard n-gram language model and a transliteration component consisting of conditional probability distributions $P(\boldsymbol{s} \mid \boldsymbol{t})$ trained using the EM algorithm over a set of sub-word (primarily syllable) transliteration rules. The sub-word transliteration probabilities are assumed to be independent, in contrast to our approach. For this study, its language model and transliteration training data were identical to our system.

4.2 WFST optimizations

Table 1 presents the size in megabytes of the $L \circ G$ compiled in the standard way, with a general method of compression (compact), and with the specialized LOUDS format.

Figure 7 presents a sweep over beam search parameters on the Hindi dev set, for identical models compiled with and without weight pushing. As can be seen from that plot, with both transliteration transducer optimization (section 3.2) and $L \circ G$ optimization (section 3.3), very large latency reductions were achieved versus the unoptimized system, yielding the lowest word error rate along with the lowest latencies.[3] Due to these optimizations, low word error rates are achieved within the latency constraints imposed by the application.

4.3 Blind test evaluation

After setting meta-parameters on the development set for Hindi, we ran the decoder on blind test sets for both Hindi and Tamil. Table 2 presents a comparison between the WFST-based decoder and a baseline system.

5 Summary and future directions

We have presented a WFST-based approach to transliteration support in keyboards on mobile devices, with specific results in South Asian languages. The presented WFST optimization methods yielded very large model size reductions, critical for on-device models. We also presented weight pushing approaches that yielded large speedups in decoding, with a commensurate reduction in active memory usage. The resulting system achieves large accuracy improvements in addition to strong latency reductions.

This approach has been used to build transliteration keyboard systems for the 22 languages shown in Table 3, some of which can be transliterated to different scripts. These keyboards were launched as part of Google Gboard in the first half of 2017. We continue to investigate new methods for training such models, including jointly training target script language models and transliteration models.

Acknowledgments

Thanks to Daan van Esch, Elnaz Sarbar and Cory Massaro for invaluable support with data curation; to Richard Sproat for useful guidance at the outset of this project and helpful perspectives throughout; and to Yuanbo Zhang for helping understand the Indic IME approach and facilitating direct comparisons of the approaches.

[3] Active memory usage is highly correlated with latency – being due to decoder search – hence we do not show that plot, though it shows a similar trend.

References

Umair Z Ahmed, Kalika Bali, Monojit Choudhury, and Sowmya VB. 2011. Challenges in designing input method editors for Indian languages: The role of word-origin and context. *Workshop on Advances in Text Input Methods (WTIM 2011)*, pages 1–9.

Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2003. Generalized algorithms for constructing statistical language models. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 40–47.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata*, pages 11–23. Springer.

Cyril Allauzen, Michael Riley, and Johan Schalkwyk. 2009. A generalized composition algorithm for weighted finite-state transducers. In *Proceedings of Interspeech*, pages 1203–1206.

Maximilian Bisani and Hermann Ney. 2008. Joint-sequence models for grapheme-to-phoneme conversion. *Speech Communication*, 50(5):434–451.

Hsin-Hsi Chen, Sheng-Jie Hueng, Yung-Wei Ding, and Shih-Chung Tsai. 1998. Proper name translation in cross-language information retrieval. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*, pages 232–236. Association for Computational Linguistics.

O'Neil Delpratt, Naila Rahman, and Rajeev Raman. 2006. Engineering the louds succinct tree representation. In Carme Àlvarez and María Serna, editors, *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings*, pages 134–145. Springer Berlin Heidelberg, Berlin, Heidelberg.

Parth Gupta, Kalika Bali, Rafael E Banchs, Monojit Choudhury, and Paolo Rosso. 2014. Query expansion for mixed-script information retrieval. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 677–686. ACM.

Li Haizhou, Zhang Min, and Su Jian. 2004. A joint source-channel model for machine transliteration. In *Proceedings of the 42nd Annual Meeting on association for Computational Linguistics*, page 159. Association for Computational Linguistics.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4):599–612.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Josef R Novak, Nobuaki Minematu, and Keikichi Hirose. 2013. Failure transitions for joint n-gram models and g2p conversion. In *Proceedings of Interspeech*.

Josef Robert Novak, Nobuaki Minematsu, and Keikichi Hirose. 2015. Phonetisaurus: Exploring grapheme-to-phoneme conversion with joint n-gram models in the WFST framework. *Natural Language Engineering*, pages 1–32.

Tom Ouyang, David Rybach, Françoise Beaufays, and Michael Riley. 2017. Mobile keyboard input decoding with finite-state transducers. *arXiv preprint arXiv:1704.03987*.

Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. 2011. The Kaldi speech recognition toolkit. In *IEEE workshop on automatic speech recognition and understanding (ASRU)*.

Kanishka Rao, Fuchun Peng, Haşim Sak, and Françoise Beaufays. 2015. Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *Proceedings of ICASSP*, pages 4225–4229.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66.

Jeffrey Sorensen and Cyril Allauzen. 2011. Unary data structures for language models. In *Proceedings of Interspeech*, pages 1425–1428.

Richard Sproat. 2003. A formal computational analysis of Indic scripts. In *International symposium on Indic scripts: past and future*, Tokyo.

Paola Virga and Sanjeev Khudanpur. 2003. Transliteration of proper names in cross-lingual information retrieval. In *Proceedings of the ACL 2003 workshop on Multilingual and mixed-language named entity recognition-Volume 15*, pages 57–64. Association for Computational Linguistics.

Ke Wu, Cyril Allauzen, Keith Hall, Michael Riley, and Brian Roark. 2014. Encoding linear models as weighted finite-state transducers. In *Proceedings of Interspeech*.

Shumin Zhai and Per-Ola Kristensson. 2003. Shorthand writing on stylus keyboard. In *Proceedings of the ACM SIGCHI conference on Human factors in computing systems*, pages 97–104.