INTERSPEECH 2019
September 15–19, 2019, Graz, Austria

GPU-based WFST Decoding with Extra Large Language Model

Daisuke Fukunaga[1], *Yoshiki Tanaka*[1], *Yuichi Kageyama*[1]

[1]Sony Corporation, Japan

Daisuke.Fukunaga@sony.com, Yoshiki.Tanaka@sony.com, Yuichi.Kageyama@sony.com

Abstract

Weighted finite-state transducer (WFST) decoding in speech recognition can be accelerated by using graphics processing units (GPUs). To obtain a high recognition accuracy in a WFST-based speech recognition system, a very large language model (LM) represented as a WFST with more than 10 GB of data is required. Since a GPU typically has only several GB of memory, it is impossible to store such a large LM in GPU memory. In this paper, we propose a new method for WFST decoding on a GPU. The method utilizes the *on-the-fly rescoring* algorithm, which performs the Viterbi search on a WFST with a small LM and rescores hypotheses using a large LM during decoding. We solve the problem of insufficient GPU memory by storing most of the large LM in a memory on the host and copying the data from the host memory to the GPU memory as needed during runtime. Our evaluation of the proposed method on the LibriSpeech test sets using an NVIDIA Tesla V100 GPU shows that it achieves a ten times faster decoding than an equivalent CPU implementation without recognition accuracy degradation.

Index Terms: speech recognition, weighted finite-state transducer, graphics processing unit

1. Introduction

In recent years, large-vocabulary continuous speech recognition (LVCSR) systems have been used in many applications. A fast and high-throughput LVCSR system is required in many use cases, e.g., transcribing massive audio streams in a short time, enabling a voice assistant service to respond to many requests in a timely manner. The primary computation of an LVCSR system consists of acoustic model (AM) inference and weighted finite-state transducer (WFST) decoding. Since a deep neural network (DNN) is often used as an AM (e.g., DNN/HMM hybrid model [1–5]), its inference can be easily accelerated by using a GPU for DNN (or matrix) computation. In contrast, it is difficult to accelerate WFST decoding using a GPU because its algorithm cannot be easily parallelized. Substantial increases in speech have been reported for speech recognition tasks with a WFST network composed of a limited vocabulary when using a GPU [6–9]. However, to obtain a high recognition accuracy for various speeches, large language models (LMs) that include a large vocabulary (e.g., a million words) and an extremely large number of n-gram entries (e.g., billions of n-grams) are required. It is impractical to use WFSTs with such large LMs on a GPU because of the limitation of the GPU memory size. One of the solutions to the memory problem is to use the *on-the-fly rescoring* algorithm [10–12]. This method of WFST decoding performs the Viterbi search on a WFST with a small LM and rescores hypotheses using a large LM during decoding. A series of prior works introduced on-the-fly rescoring with a hybrid GPU/CPU architecture [13–15]. With the hybrid architecture, the GPU searches for a small WFST while the CPU rescores by using a large LM stored in a host memory. Another solution is to generate lattices [16] using a WFST with a small LM on a GPU and rescore the lattices by using a large LM on a CPU. The GPU-accelerated parallel Viterbi search and lattice generation algorithms are introduced in [17]. These algorithms enable the subsequent CPU-based rescoring of the lattices generated using large LMs.

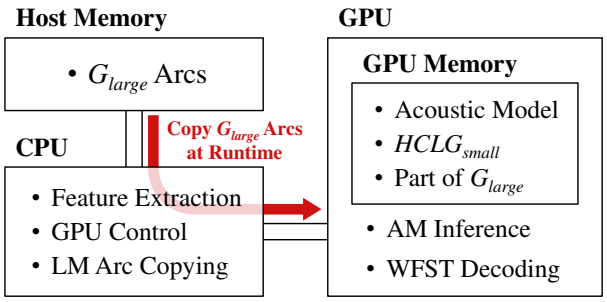

Figure 1: *System overview of our proposed method.*

In this paper, we propose a new method for GPU-based on-the-fly rescoring. In this method, most of the large LM is stored in a host memory and the data in the host memory is copied to the GPU memory as needed during runtime. There are two advantages of this method. Firstly, we can carry out decoding with LMs of more than 10 GB on a GPU by copying the data of the large LM from the host to the GPU as needed. Secondly, since the proposed method executes most of the computation (including the large-LM lookup and hypothesis rescoring) on a GPU, CPU bottlenecks are circumvented without requiring a more powerful CPU.

2. GPU-based on-the-fly rescoring

A statically composed WFST comprising an *HCLG* (hidden Markov model *H*, context dependence model *C*, pronunciation lexicon *L*, and n-gram language model *G*) performs efficient speech recognition [18]. However, in the case of using a large vocabulary and a large number of n-grams for LMs, it is infeasible to decode with a single composed *HCLG* because an extremely large *HCLG* is generated by its composition. On-the-fly rescoring enables decoding with a large LM by dividing an LM into a small LM G_{small}, e.g., a pruned trigram, and a large LM G_{large}, e.g., an unpruned 4-gram (deviation from G_{small}). Hypotheses generated by the Viterbi search with $HCLG_{small}$ are rescored by G_{large} during decoding. With our proposed method, WFST decoding using on-the-fly rescoring is executed efficiently on a GPU. In this section, the proposed method is described in detail.

2.1. Basic strategy

One of the purposes of our proposed method is to execute the LM lookup on a GPU. However, for a speech recognition task with a very large vocabulary, the size of G_{large} becomes more

http://dx.doi.org/10.21437/Interspeech.2019-2101

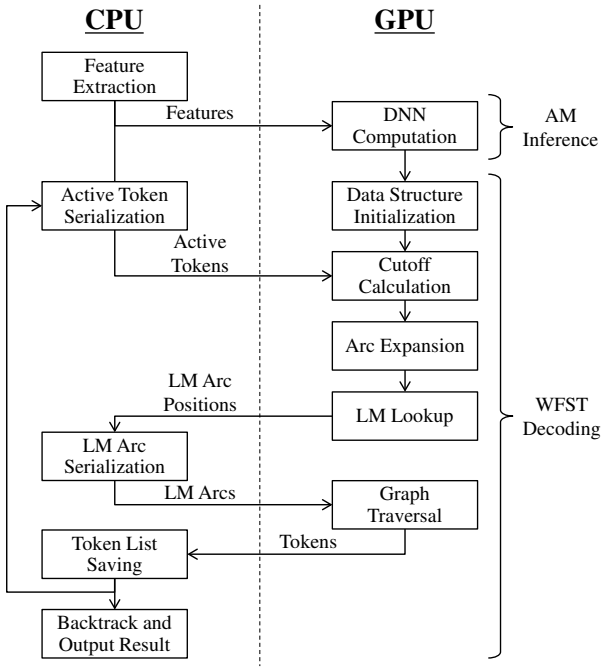

Figure 2: *Implemented pipeline of our proposed method.*

than 10 GB, whereas a GPU typically has several GB of memory at most. Therefore, we store only the data required for the lookup of G_{large} in the GPU memory and store the remaining data in the host memory. The data stored in the host memory are copied from the CPU to the GPU as needed during runtime. The basic strategy is to execute almost all the processes, including the LM lookup, on the GPU while minimizing the data stored in the GPU memory.

The LM lookup is a process of finding the arc in G_{large} corresponding to a transition in $HCLG_{small}$ using the state in G_{large} and the output label of $HCLG_{small}$. To execute the LM lookup on a GPU, the labels of all arcs and the backoff states of each state in G_{large} are stored in the GPU memory. The positions of the first label of each state are also stored in the GPU memory to help identify which state each label belongs to. The labels are arranged in an array in order from the first state to the last state. The labels in the same state are sorted so that arc positions can be calculated by binary searching. The positions of the first labels and the backoff states are also arranged in arrays. To obtain the number of outgoing arcs from a state, we can subtract the position of the first label of the state from that of the next state. The remaining data of G_{large} are arc information (label, weight, and destination state). They are stored in the host memory. About 30% of the data of G_{large} are stored in the GPU memory using this data structure. As an example, provided that a GPU has 16 GB of memory, it is possible to use G_{large} up to about 40 GB considering the amount of memory used by AMs, $HCLG_{small}$, and the working space.

2.2. System overview

Figure 1 shows an overview of a speech recognition system employing our proposed method. An AM, $HCLG_{small}$, and part of G_{large} are stored in the GPU memory. The arcs of G_{large} are stored in the host memory. The AM inference and WFST decoding are executed on the GPU, while the feature extraction, GPU control, and copying of G_{large} arcs are executed on the CPU.

Figure 2 shows the implemented pipeline of our proposed method. After feature extraction is executed on the CPU, AM inference is executed on the GPU. The output of AM inference is subsequently used for WFST decoding on the GPU. WFST decoding uses the token passing algorithm [19]. First, the active tokens saved in the host memory are copied to the GPU memory. After the data structure on the GPU is initialized, the cutoff computation, the arc expansion of $HCLG_{small}$, and the LM lookup are carried out in this order. The positions of the required G_{large} arcs (calculated by the LM lookup) are copied to the CPU. Then, the G_{large} arcs are copied from the host memory to the GPU memory. Next, the graph traversal is executed along with the rescoring of hypotheses on the GPU using the copied G_{large} arcs. A token list after the transitions is copied to the CPU and saved in the host memory. These processes are repeated until all audio frames are consumed. Finally, a recognition result is extracted on the CPU.

In the rest of this section, the implementation of our proposed method is described in detail. The procedure for copying G_{large} arcs efficiently is described in Section 2.3. The parallel implementation of the arc expansion and graph traversal is described in Section 2.4.

2.3. Language model arc transfer between CPU and GPU

It is important to efficiently copy arcs of G_{large} stored in the host memory to the GPU. For efficiency, the arcs copied to the GPU should be gathered with no duplication. The LM lookup is carried out to specify the arc of the next transition in G_{large} by the output word label from $HCLG_{small}$ and the current state in G_{large} when a transition on $HCLG_{small}$ outputs a word. Arcs of G_{large} are stored in an array in the host memory. The positions of the arcs required for the next transitions are calculated on the GPU. Different tokens in the same frame may correspond to the same arc of G_{large}; therefore, duplications must be omitted to minimize the data size to be copied. Since each arc of G_{large} used in the next transitions is determined only by the current state and label, it is possible to avoid redundancy by using a hash table whose key is a pair of a state and a label. Algorithm 1 shows the procedure of searching for elements using a hash table on a GPU. Since the hash table uses linear probing, if the destination slot for the inserted element is already occupied, the next open slot is used. If the maximum number of elements stored in the hash table is known beforehand, hash table expansion is unnecessary. For the LM lookup, as shown in Algorithm 2, whether the position of an arc of G_{large} has already been calculated by

Algorithm 1 Element Search using Hash Table on GPU

```
procedure HASHTABLEFIND(key)
    index ← HashFunction(key) % tableSize
    loop        ▷ Loop must end if numElements ≤ tableSize.
        old ← atomicCAS(&tableKeys[index], EMPTY, key)
            ▷ tableKeys are initialized to EMPTY beforehand.
        if old = key then                          ▷ Element is found.
            found ← true
            return found, index
        else if old = EMPTY then                   ▷ Element is not found.
            found ← false
            return found, index
        index ← (index + 1) % tableSize
```

Algorithm 2 Language Model Lookup with Hash Table

```
procedure LOOKUP(state, label)
    found, index ← HashTableFind(Pack(state, label))
    if not found then
        arcPosition ← GetLmArcPosition(state, ilabel)
        if not backoff transition then
            arcId ← atomicAdd(count, 1)
                        ▷ count is initialized to 0 beforehand.
            tableValues[index] ← arcId
            arcPositions[arcId] ← arcPosition
        else   ▷ If backoff transition, call Lookup recursively.
            arcId ← Lookup(state, 0)
            tableValues[index] ← arcId
            Lookup(backoffStates[state], label)
    return tableValues[index]
```

Algorithm 3 Parallel Arc Expansion

```
procedure EXPAND(states, tokens, cutoff)
    state ← states[threadId]
    token ← tokens[threadId]
    if token.weight < cutoff then        ▷ Token is not pruned.
        numArcs ← GetNumOutgoingArcs(state)
    else                  ▷ If token is pruned, set numArcs to 0.
        numArcs ← 0
    numArcsArray[threadId] ← numArcs
    Synchronize all threads
    startPosition ← PrefixSum(numArcsArray, threadId);
    for i ← 0 to numArcs - 1 do
        arc ← GetArc(state, i)
        arcArray[startPosition + i] ← arc
```

Algorithm 4 Token Recombination with Hash Table

```
procedure RECOMBINE(states, tokens)
    state ← states[threadId]
    token ← tokens[threadId]
    _, index ← HashTableFind(state)
    atomicMin(&tableValues[index], token.weight)
      ▷ tableValues are initialized to FLT_MAX beforehand.
    Synchronize all threads
    if tableValues[index] = token.weight then
        tokenId ← atomicAdd(count, 1)
                        ▷ count is initialized to 0 beforehand.
        tokenList[tokenId] ← token
```

using the hash table is examined. The position of the arc is calculated and inserted into the hash table only if it does not exist. For backoff transitions, several different pairs of a state and a label may correspond to the same arc. Therefore, such arcs are inserted in the hash table with the label set to 0 to eliminate duplications. By doing this, we can obtain a list of the positions of the arcs of G_{large} used for the next transitions without duplications. After finishing the calculations of the positions for all tokens, the list is copied to the CPU.

Arcs of G_{large} copied to the GPU are cached in the GPU memory after having been used to reduce the amount of data transferred between the CPU and GPU and to calculate the positions of the arcs. We also use a hash table for caching the arcs of G_{large}. Cached arcs in the hash table can be located by hashes computed with input pairs of a state and a label. If hashes conflict when writing a cache, the old cache is overwritten; linear probing is not used for this hash table. Caching arcs of G_{large} in the GPU memory are potentially very effective for improving the decoding time.

2.4. Parallel Viterbi search on GPU

One of the issues of the Viterbi search on a GPU is how to deal with different numbers of outgoing arcs in parallel. In [20], the data used in the next transitions are gathered in an array in advance for efficient memory access. In [17], active tokens are dynamically load-balanced into groups of 32 threads with all outgoing arcs of each token processed in the thread group. In our proposed method, we expand all arcs of $HCLG_{small}$ used in the next transitions to an array in advance. As shown in Algorithm 3, the arcs are expanded in parallel. The numbers of outgoing arcs are loaded on each thread as one GPU thread is assigned to one active state. Next, prefix sums are calculated for each number of arcs. The prefix sums are the destination indices of the array used to write the data. Using the prefix sums, the data of the arcs are written to the array in parallel. If a token is pruned by the cutoff, the number of outgoing arcs of its state is set to 0 to avoid redundant computation. In the subsequent processes, one GPU thread is assigned to one arc. By expanding the arcs of $HCLG_{small}$ in advance, the exact number of parallelism is known beforehand. This information enables us to execute GPU processes efficiently.

Token synchronization is another issue. If tokens reach the same state in the same frame, only the token with the smallest weight remains. This mechanism is called *token recombination*. In [17, 20, 21], the *atomicMin*[1] operation is applied to an array whose size is the number of states or arcs for the token recombination. Using this procedure, the array consumes a large amount of GPU memory when a WFST has a large number of states and arcs. This is critical, especially when processing multiple utterances simultaneously. We use a hash table instead of an array to reduce GPU memory consumption because most of the elements in the array are actually unused. Algorithm 4 shows the token recombination procedure using a hash table. The hash table uses the linear probing mechanism described in Section 2.3. We should set the size of the hash table to several times larger than the total number of transitions of the frame to enable processing of the token recombination with sufficiently small insertion overheads. Even then, the memory usage of the hash table is considerably smaller than that of the array, whose size is the number of states or arcs.

We also employ some other techniques. All tokens after the token recombination are stored in the host memory to reduce GPU memory usage. Since calculating the best path is an insignificant process, it is calculated directly on the CPU to avoid having to copy the data back to the GPU. For the cutoff calculation, we use the radix selection algorithm to find the token with the k-th smallest weight in parallel. We also convert $HCLG_{small}$ into an epsilon-free WFST as described in [7] to reduce the number of GPU kernels.

3. Experiments

We evaluated the effectiveness of our proposed GPU-based on-the-fly rescoring method on LibriSpeech test sets, which are a corpus derived from audiobooks [22]. The LibriSpeech test sets

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmin

Table 1: *Results of the measurements ("other", beam=16)*

Method		p2.xlarge		p3.2xlarge	
	Thread	RTF	Δ	RTF	Δ
CPU	Single	0.6883	1.0×	0.4594	1.0×
	Multi	0.2645	2.6×	0.1007	4.6×
GPU (Proposed)	Single	0.1314	5.2×	0.0453	10.1×
	Multi	0.0458	15.0×	0.0122	37.7×

include 5.4 hours of "clean" utterances and 5.1 hours of "other" utterances. We trained an AM and an LM based on the training recipe from the Kaldi toolkit [23] using a 960-hour variant of LibriSpeech as the training data set.

We created $HCLG_{small}$ with the pruned trigram LM and G_{large}, which is the unpruned 4-gram LM. $HCLG_{small}$ has 8,944,726 states and 29,677,775 arcs, and G_{large} has 27,370,770 states and 167,448,530 arcs. The LM has a vocabulary of 200k words and G_{large} has 145M n-gram entries.

The evaluations were performed on the *p2.xlarge* instance and *p3.2xlarge* instance of AWS EC2[2]. The p2.xlarge has an Intel Xeon *E5-2686 v4* 2-core (4-thread) CPU and an NVIDIA Tesla K80 GPU. The p3.2xlarge has an Intel Xeon *E5-2686 v4* 4-core (8-thread) CPU and an NVIDIA Tesla V100 GPU.

We compared our proposed method with an equivalent CPU implementation. In this evaluation, both the CPU implementation and the proposed method executed AM inferences on a GPU. Only WFST decoding was executed differently depending on the implementation. We used the real time factor (RTF), which indicates the ratio of the decoding time to the utterance length, for the evaluation of the decoding time. The measurements were carried out with various beam widths (8, 10, 12, 14, and 16). The word error rate (WER) of the proposed method is generally equivalent to that of the CPU implementation since the implemented search algorithm in our proposed method is almost identical to that of the CPU implementation.

Table 1 shows RTFs of the 1-best search using the "other" test set. Our proposed method is 5.2 times faster on p2.xlarge and 10.1 times faster on p3.2xlarge than the CPU implementation in the single-thread measurement (processing utterances sequentially). RTFs can be further reduced by processing multiple utterances simultaneously using multithreading. We obtained a 16.0 times speedup on p2.xlarge with 4 threads and a 37.7 times speedup on p3.2xlarge with 8 threads.

Figure 3 shows the relationship between WER and RTF measured with different beam widths using the "other" test set. The effectiveness of the proposed method increases proportionally to the beam width. The ratio between the overhead of the GPU computation and the overall computation duration scales inversely proportional to the beam width (the number of processed tokens). Figure 4 shows the result of multithreading. Although the CPU implementation is scaled on a multi core CPU, the GPU implementation of the proposed method is also scaled. It is assumed that GPU resources are not exhausted from just one utterance because the number of active tokens fluctuates from a few to several thousand. Therefore, to maximize the performance of the proposed method, it is necessary to process multiple utterances simultaneously.

In multithreaded processing, while the CPU utilization of the CPU implementation is almost 100%, the CPU utilization of the proposed method is only about 25%. Since the computation is mainly executed on a GPU in the proposed method, CPU utilization is not the bottleneck.

[2] https://aws.amazon.com/ec2/

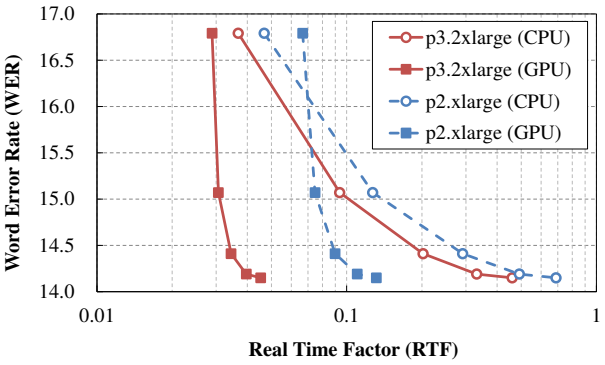

Figure 3: *Relationship between WER and RTF (single thread).*

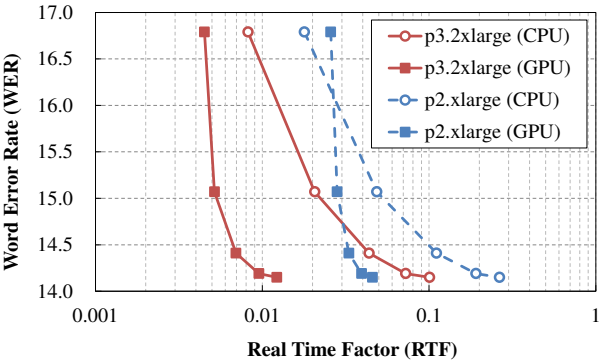

Figure 4: *Relationship between WER and RTF (multithread).*

The size of G_{large} used in this evaluation was 3 GB, 0.9 GB of which was stored in the GPU memory in the proposed method. Since the Tesla V100 GPU on p3.2xlarge has 16 GB of memory and the memory usage of $HCLG_{small}$ and the working space was about 2 GB, the proposed method can be applied with G_{large} up to around 40 GB. Additionally, we can use a larger G_{large} by using the label compression method as described in [24] to reduce GPU memory usage.

4. Conclusions

In this paper, we describe a new method for WFST-based decoding with the on-the-fly rescoring algorithm. The method stores most of the large LM in the host memory and copies the arcs of the large LM to the GPU memory as needed during runtime. By storing most of the data of the large LM in the host memory, it is possible to decode with a huge LM on a GPU with a limited memory capacity. Moreover, by implementing efficient data transfer between the CPU and the GPU and implementing a parallel processing algorithm on the GPU, we significantly reduce the decoding time. In the evaluation using LibriSpeech test sets, the decoding of our proposed method was 10.1 times faster than the CPU implementation. Since the computation is mainly executed on a GPU in the proposed method, CPU utilization is not the bottleneck of WFST decoding. A more powerful CPU is not necessary to accelerate speech recognition tasks with an extra-large language model. Instead, it is possible to achieve speedups using only a GPU and a large host memory.

5. References

[1] D. Yu, L. Deng, and G. E. Dahl, "Roles of Pre-Training and Fine-Tuning in Context-Dependent DBN-HMMs for Real-World Speech Recognition," in *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning*, 2010.

[2] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition," in *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30-42, 2012.

[3] F. Seide, G. Li, and D. Yu, "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks," in *INTERSPEECH 2011 – 12th Annual Conference of the International Speech Communication Association*, 2011, pp. 437-440.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," in *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82-97, 2012.

[5] O. Abdel-Hamid, A. Mohamed, H. Jiang and G. Penn, "Applying Convolutional Neural Networks concepts to hybrid NN-HMM model for speech recognition," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 4277-4280.

[6] K. You, J. Chong, Y. Yi, E. Gonina, C. J. Hughes, Y.-K. Chen, W. Sung, and K. Keutzer, "Parallel Scalability in Speech Recognition," *IEEE Signal Processing Magazine*, vol. 26, no. 6, 2009.

[7] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A Fully Data Parallel WFST-based Large Vocabulary Continuous SpeechRecognition on a Graphics Processing Unit," in *INTERSPEECH 2009 – 10th Annual Conference of the International Speech Communication Association*, 2009, pp. 1183-1186.

[8] J. Chong, E. Gonina, K. You, and K. Keutzer, "Exploring Recognition Network Representations for Efficient Speech Inference on Highly Parallel Platforms," in *INTERSPEECH 2010 – 11th Annual Conference of the International Speech Communication Association*, 2010, pp. 1489-1492.

[9] J. Kim, J. Chong, and W. Sung, "H- and C-level WFST-Based Large Vocabulary Continuous Speech Recognition on Graphics Processing Units," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011, pp. 1733-1736.

[10] T. Hori, C. Hori, and Y. Minami, "Fast On-The-Fly Composition for Weighted Finite-State Transducers in 1.8 Million-Word Vocabulary Continuous Speech Recognition," in *INTERSPEECH 2004 – 8th International Conference on Spoken Language Processing*, 2004, pp. 289-292.

[11] T. Hori and A. Nakamura, "Generalized Fast On-The-Fly Composition Algorithm for WFST-based Speech Recognition," in *INTERSPEECH 2005 – 9th European Conference on Speech Communication and Technology*, 2005, pp. 557-560.

[12] T. Hori, C. Hori, Y. Minami and A. Nakamura, "Efficient WFST-Based One-Pass Decoding With On-The-Fly Hypothesis Rescoring in Extremely Large Vocabulary Continuous Speech Recognition," in *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 15, no. 4, pp. 1352-1365, 2007.

[13] J. Kim, J. Chong, and I. Lane, "Efficient On-The-Fly Hypothesis Rescoring in a Hybrid GPU/CPU-based Large Vocabulary Continuous Speech Recognition Engine," in *INTERSPEECH 2012 – 13th Annual Conference of the International Speech Communication Association*, 2012, pp. 1034-1037.

[14] J. Kim and I. Lane, "Accelerating Large Vocabulary Continuous Speech Recognition on Heterogeneous CPU-GPU Platforms," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 3291-3295.

[15] J. Kim and I. Lane, "Accelerating Multi-User Large Vocabulary Continuous Speech Recognition on Heterogeneous CPU-GPU Platforms," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 5330-5334.

[16] D. Povey, M. Hannemann, G. Boulianne, L. Burget, A. Ghoshal, M. Janda, M. Karafiát, S. Kombrink, P. Motlíček, Y. Qian, K. Riedhammer, K. Veselý, and N. T. Vu, "Generating exact lattices in the WFST framework," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 4213-4216.

[17] Z. Chen, J. Luitjens, H. Xu, Y. Wang, D. Povey, and S. Khudanpur, "A GPU-based WFST Decoder with Exact Lattice Generation," in *INTERSPEECH 2018 – 19th Annual Conference of the International Speech Communication Association*, 2018, pp. 2212-2216.

[18] M. Mohri, F. Pereira, and M. Riley, "Weighted Finite-State Transducers in Speech Recognition," *Computer Speech & Language*, vol. 16, no. 1, pp. 69-88, 2002.

[19] S. J. Young, N. H. Russell, and J. H. S Thornton, "Token Passing: a Simple Conceptual Model for Connected Speech Recognition Systems," Cambridge University Engineering Department Technical Report, Cambridge, UK, 1989.

[20] J. Chong, E. Gonina, and K. Keutzer, "Efficient Automatic Speech Recognition on the GPU," *GPU Computing Gems Emerald Edition*, 2011, pp. 601-618.

[21] A. Argueta and D. Chiang, "Decoding with Finite-State Transducers on GPUs," in *Proceedings of EACL*, 2017, pp. 1044-1052.

[22] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An ASR Corpus Based on Public Domain Audio Books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5206-5210.

[23] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlíček, Y. Qian, P. Schwarz, J. Silovský, G. Stemmer, and K. Veselý, "The Kaldi Speech Recognition Toolkit," in *2011 IEEE Workshop on Automatic Speech Recognition and Understanding*, 2011.

[24] D. Caseiro, "WFST Compression for Automatic Speech Recognition," in *INTERSPEECH 2010 – 11th Annual Conference of the International Speech Communication Association*, 2010, pp. 1493-1496.