Online Speech Decoding Optimization Strategy with Viterbi Algorithm on GPU

Alfonsus Raditya Arsadjaja and Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics Institut Teknologi Bandung
Bandung, Indonesia
raditya1710@gmail.com and imam@informatika.org

Abstract—**Automatic Speech Recognition (ASR) has been popular recently. But the current algorithm for speech recognition is slow and needed the way to recognize faster. One way to achieve it is with GPU, which provides parallel computation; but ASR is hard to parallelize directly.**

This paper describes how to build parallel ASR system, which requires several steps. First, we must convert the data structure to make it compatible with GPU, then we have to make several kernels that equivalent to the serial algorithm in CPU.

We will describe several optimization strategies for make ASR run much faster after we got the correct GPU program. Those strategies are based on profiling result and analysis of the GPU program execution flow.

Best implementation that we had have a speedup around 5.59-6.18 times from the serial CPU implementation.

Keywords-ASR; decoding; GPU; parallel; optimization strategy

I. INTRODUCTION

Automatic Speech Recognition (ASR) has been an active research area for the last five decades, resulting mature core technologies such as Gaussian mixtures models (GMMs), hidden markov models (HMMs), mel-frequency ceptrals coefficients (MFCCs) for ASR has been developed and used in applications that can be found in everyday life, such as for dictating, voice activated commands and even for automatic conference script.

ASR is known to be a challenging problem to parallelize [1] [2], as it has irregular and unpredictable memory accesses. One of the popular algorithm used for decoding is Viterbi search algorithm. There are some number of tools for ASR, and none of them supports the parallel version of Viterbi.

Recently, Graphics Processing Unit (GPU) has been used for general purpose computation, especially for large computation. With correct implementation, GPU can be used to parallelize the process of speech recognition fast and get much speedup. In this paper, we will discuss some strategies with existing library and tools, and how to combine them to produce the parallel Viterbi algorithm in automatic speech recognition.

In this paper, we proposed an online decoding process in the GPU based on the implementation of an opensource ASR tool, Kaldi-ASR and from the implementation of GPU Viterbi in [3]. We implement several strategies to make it faster around 5-6 times of the Kaldi [4] optimized CPU program. Our implementation is provided as an open source in [1].

II. SPEECH DECODING

A. Basic Concepts

Typical architecture of Automatic Speech Recognition is illustrated in Figure 1 below.

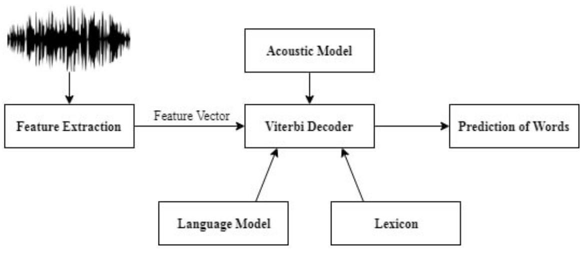

Figure 1 ASR Architecture [5]

The ASR is comprised of Feature Extraction, Viterbi Decoder, with Acoustic Model, Language Model and Lexicon. Acoustic model is a model that stores every signal representation to its equivalent numbers. The model is built using statistical models, such as GMM-HMM and DNN-HMM.

Language model is a model that stores the dependency and relation of each words, so it provides the context about the language used.

Lexicon is a component that stores the sequence of phones that build up a word, for every word in dictionary.

First, the audio signal from file is gone through the feature extraction, to get the feature vectors. These feature vectors are processed to get the probable words from the signal. There are many representation of feature vectors, and Mel Frequency Cepstral Coefficient (MFCC)) is the most frequently used one.

The feature vectors, combined with acoustic model, language model, and lexicon are fed into the decoder, and the decoder begins to make a prediction of words based on the feature vectors.

Speech recognition can be divided into 2 parts, which is training and decoding. This paper only focused on the decoding phase. In this phase, we used Viterbi algorithm for decoding and a graph model called WFST (weighted finite state transducer) proposed by [6] as the representation of the language model and lexicon. WFST consists of 4 parts: H (HMM) that stores parts of the phone, C (Context dependency) that stores the relations between each phone, L (Lexicon), and G (Grammar) that stores the language model. The composition of those 4 parts resulting the graph called as H o C o L o G that ready to use for speech decoding.

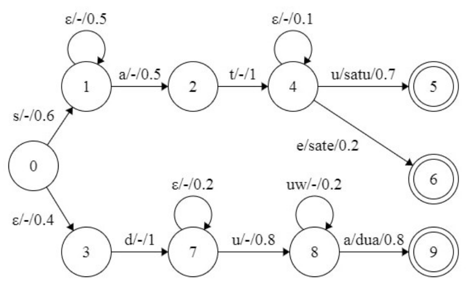

Figure 2 Example of WFST

B. *Viterbi Algorithm*

Viterbi algorithm is an algorithm that have been used for speech decoding for a long time [7] [8]. Basically, this algorithm uses dynamic programming approach to get the most likely sequence in the graph, which later called the Viterbi path. Viterbi algorithm can be used in many type of models, including WFST.

Figure 2 illustrates an example of a WFST used for speech decoding. A vertex is labeled with a state id, and an edge represents input label, output label and a probability of transition between the two states. In WFST, starting from initial state, the algorithm expands the states for every frame to neighboring state using breadth-first-search (BFS) algorithm, and compute the probability of that neighboring state with this equation:

$$prob_{next\_state} = prob_{cur\_state} * weight(arc) * acoustic\_score(frame, input)$$

For notes, acoustic score of a frame with given input label is how likely is the frame's feature vector represent the input label. If there are no input label (e.g. epsilon label), then there are no acoustic score computation for that transition, and we can ignore the last multiplication for above equation.

If more than one transition expands to the same state, then we select the transition that have maximum probability to that state (this is where the dynamic programming approach is used)

After the last frame already decoded, then the best last token (or if one reached final state) are selected to be used for backtracking and get the Viterbi path.

C. *Kaldi CPU Viterbi Algorithm*

Kaldi implements the Viterbi algorithm serially. For every frame, the process is divided into 2 phases; the first is the computation of epsilon arcs, and the second is the computation of non-epsilon arcs. For both phases, all tokens are iterated, and for every token, all transitions from that token are processed sequentially. This is iterated until it reaches the last frame on the input.

Sometimes, the input can be so long, and the number of frames can be large. Therefore, Kaldi provides slightly modified version called *online decoding*. In this version, the decoding process is divided into several batch, and for every batch we processed some number of frames to make the traceback not too long. In every batch, Kaldi use partial traceback function to show the most probable sequence of words for input in that batch, or finish traceback function if it reached end of sentence or end of input.

III. Related Research

Viterbi decoder can be implemented in many ways: serial CPU, multicore CPU, and GPU. Multicore CPU implementation in [9] uses refactorization of the innermost loop of GMM evaluation code (which are known to be costly), exploit the vector unit to store the means and variances, and run them in parallel. It achieves 2.68x speedup from serial implementation.

Our proposed implementation uses GPU to run the intensive GMM evaluation code in parallel. GPU provides much larger amount of core, so ideally it can achieve massive speedup from serial implementation. Basically, the idea is similar with [9] that refactorize the GMM evaluation code, and make a new data structure that uses vector, but also with several optimization that provided by GPU.

There are several researches that implements Viterbi decoder for speech recognition/Natural Language Processing in GPU [10] [11] [12] [2] [3] [13]. [10] is among the first to implement speech recognition system on GPU, and used the GPU to compute observation probabilities, but were not able to exploit the full potential of the GPU platform [11]. [11] [13] [12] proposed to optimize the observation probabilities computations by avoiding the computation of observation probabilities for potential next states with the same label. This is done by employing a lookup table of flags for identifying a set of labels that appear in the set of potential next states. Based on the flags, the computation of observation probabilities only performed for unique label only. We follow the same approach, by employing the *compute_loglikelihoods* kernel to compute the observation probabilities for each unique label and keep the results as the lookup table that will be used in subsequent computation.

The author of [2] implements a GPU implementation based on the idea presented in [14]. The computation is divided into three phases for each iteration of the inference engine: observation probability computation, non epsilon arc traversal, and epsilon arc phase. Our approach has similar phases, but we avoid the state transfer between CPU and GPU in between phases.

There are some significant differences between serial and parallel implementation. For parallel implementation, all transitions and states are listed in an array instead of pointers. For every frame, all arcs that meet the conditions from all tokens are listed first, and after that, the corresponding arcs are processed independently in parallel.

The research in [3] is targeted for FST decoding in GPU, not specifically for ASR. They described the implementation in the context of text translator, which does not include the observation probability computation, and epsilon-non epsilon arc traversal. However, the concept of Viterbi search is still same. It proposed new structure for FST, so it can be processed more easily in GPU.

We adopted the implementation in [3] and added the GMM evaluation code to make it can recognize the speech. Then, integrate it with Kaldi. We also make it open source and have a simpler data structure and implementation from [2].

IV. Analysis and Design

A. Design of First GPU Program

Kaldi CPU program have many data structures inside that can't be copied directly to GPU. Many of them have pointers inside, therefore hard to allocate and copied into GPU. The object that use for computing acoustic score in Kaldi have some other data structures within, so it also can't be directly copied to GPU.

The implementation is based on [3] for the FST and some of the GPU Viterbi algorithm, but with some modifications. The main differences with the latter is there are some acoustic model for getting the phoneme in frame probability. There are several modifications while computing the transitions, described below.

First, the compute transition in [3] is divided into 2 phases, which is compute emitting arcs and non-emitting arcs, similar as [2]. For every phase, the techniques are same as the original compute transition kernel, but we separated the epsilon and non-epsilon arcs. Second, the backtracking stage is modified, since there can be some partial traceback, so it doesn't have to backtracking from the final state. Third, in the implementation, all arcs are processed for every phase in every frame (regardless the state whose the arc have been reached or not) so it removes one synchronization problems.

The last one but not least, we modify the GPU implementation so it can be processed with *online decoding*.

With above design, we get the following result:

TABLE I. Profiling Result of First GPU Program

Kernel	Percentage (%)
compute_nonemitting	0.0660
compute_emitting	**99.9276**
compute_max_	0.0002
compute_initial	0.0002
get_path	0.0002
others	0.0058

We can see that *compute_emitting* kernel dominates the time of execution. We will see why it happened with the following table:

TABLE II. Acoustic Score Execution for Every Transition

Operation	Operation type	Number of operations
loglikes = gconsts	Assignment & Data Copy	6 – 16
data_sq = data.pow(2)	Scalar multiplication	40
loglikes = loglikes + means_invvars * data	Matrix multiplication	240 – 640
loglikes = loglikes + -0.5 * inv_vars * data_sq	Matrix multiplication	240 – 640
logsum = LogSumExp(loglikes)	Addition and assignment	6 – 16
TOTAL		**532 – 1,352**

B. Optimization Strategy

From TABLE II, we can see that all transition computation includes the acoustic score calculation. To optimize the execution, below are some of the optimization techniques that we use for our implementation of GPU Viterbi Algorithm.

First, a lot of acoustic score computations are using the same computations. There are ~580,000 transitions in the VoxForge dataset, and every single transition are associated to one GMM each. But there are only ~1,500 GMMs, so there must be a redundancy of acoustic score that calculated twice or more. To remove the redundancy, we precompute all acoustic score of every GMM with another kernel, called *compute-_loglikelihoods* kernel, and the *compute_emitting* kernel only have to use the lookup table of acoustic scores when expanding all transitions in the Viterbi algorithm [12].

Second, there are a lot of allocation and deallocation of device memory used for every single acoustic score computation. To reduce time for allocation and deallocation, we use pre-allocated memory space for intermediate computation needed in acoustic score computation.

Third, we adjust the number of thread per blocks depending on the number of GMMs, so we will maximize GPU utilization with increasing the number of SMs that work in parallel.

Finally, we see that some computation takes so much read and write accesses to GPU global memory, so we change it to make a temporary variable in local memory, and at the end of the all calculation, we update the variable in global memory. This reduces a lot of accesses needed to global memory with same result.

Getting the best previous token for every other state is challenging, because we must update the value and the best previous token at the same time. For solve the problem, we used 64-bit variable to store two 32-bit variable, similar with [3]. The first 32 bits are used to store the probabilities, and the last 32 bits are used to store the best previous token. We modify the last 32 bits from original implementation, so the first 1 bit are used to determine whether it belongs to compute emitting or non-emitting phase, and the rest 31 bits are used to store the previous state.

V. Experiment And Testing

A. Dataset Description and Environment

We conduct the experiments using the **Online Demo** of **Voxforge dataset**, using the **tri2b-mmi** model[2], which contains 7,021 words, 237,925 hidden states, an acoustic model that contains 1,532 GMMs, and 580,605 transitions. We think that this model already represented many of the larger size model with the same structure. To see the results of each of the optimization described in the previous section, we run 5 set of experiments. These are results of all scenarios that are tested. We will discuss the major bottleneck in every scenario and how to deal with it.

We will test it on Intel CPU Xeon E5606 2.31 GHz with 8 GB RAM and NVidia GeForce GTX 1080 GPU with 8 GB device memory.

B. Scenarios

We will run experiment based on different combination of strategy that already described in previous chapter.

In 1st scenario, we will run the correct GPU program without any strategy. This is intended to make sure that the output is correct.

In 2nd scenario, we use the 1st strategy, with hypothesis that the number of redundant computation of acoustic score will be eliminated.

In 3rd scenario, we use 1st and 2nd strategy, with hypothesis that the number of allocation is reduced so much and eliminate unnecessary allocation and deallocation.

In 4th scenario, we use 1st, 2nd, and 3rd strategy, with hypothesis that the number of SMs used will be maximized and increase GPU utilization significantly.

In 5th scenario, we use all optimization strategy, with hypothesis that the time for I/O are smaller because we use help of local memory instead of global memory.

C. Experiment Results

To show that the optimization does not result in worse decoding, we display the WER (Word Error Rate) of each of the scenario in the table below.

TABLE III. Performance of Experiments

Scenario	Sound File 1			Sound File 2			Sound File 3		
	Time (s)	Speedup	WER (%)	Time (s)	Speedup	WER (%)	Time (s)	Speedup	WER (%)
Baseline	20.17	1	10.38	28.76	1	16.89	22.42	1	21.43
1st scenario	1,288.55	0.016	12.26	1,761.95	0.016	17.57	1,291.38	0.017	21.43
2nd scenario	10.68	1.889	12.26	14.61	1.969	17.57	10.68	2.100	21.43
3rd scenario	6.87	2.935	12.26	9.45	3.044	17.57	6.90	3.249	21.43
4th scenario	5.07	3.977	12.26	6.98	4.121	17.57	5.09	4.407	21.43
5th scenario	3.61	5.592	12.26	4.97	5.780	17.57	3.63	6.176	21.43

TABLE IV. Profiling Result Of Different Scenario

Kernel	Percentage (%)				
	1st scenario	2nd scenario	3rd scenario	4th scenario	5th scenario
compute_loglikelihoods	-	**82.8122**	**72.0691**	**60.1254**	**37.2152**
compute_nonemitting	0.0660	8.7481	14.2135	20.2098	31.8562
compute_emitting	**99.9276**	7.5995	12.3302	17.6447	27.8995
compute_max	0.0002	0.0274	0.0449	0.0649	0.1001
compute_initial	0.0002	0.0228	0.0369	0.0534	0.0102
get_path	0.0002	0.0279	0.0457	0.0660	0.1018
others	0.0058	0.7261	1.2597	1.8158	2.8170

The baseline is the Kaldi CPU Viterbi Algorithm, that described on **Online Wav GMM Decoder**[3] file.

From the profiling result of each scenario, we can see that *compute_loglikelihoods* kernel have the biggest contribution, for almost every scenario. There is an exception for the 1st scenario because the acoustic score calculation is computed inside *compute_emitting* kernel.

Based on the performance of each sound file, we can see that each strategy has significant impact. The biggest speedup is obtained from the 1st strategy, which is to precompute all the acoustic score for every GMM.

[2] https://github.com/kaldi-asr/kaldi/blob/master/egs/voxforge/online_demo

[3] https://github.com/kaldi-asr/kaldi/blob/master/src/onlinebin/online-wav-gmm-decode-faster.cc

It is caused by acoustic score computation which is costly (based on TABLE II) and by applying that strategy, we reduced the number of *work* needed by over 99%.

We use one of the most advanced GPU (NVidia GeForce GTX 1080) that has 8GB of device memory. The latest GPU available as this is written (NVidia Titan V) have 12GB of device memory. Our implementation uses ~400 MB of GPU device memory, and mostly used for saving the probabilities of all states for every frame. So it can be scaled up to process FST that has 20-30x size of the current FST. More than that, and it can be the interesting research in the future.

The best scenario has a speedup from Kaldi CPU program for around 5.59-6.18 times.

VI. CONCLUSION

In this paper, we have discussed about basic of WFST, Viterbi algorithm, and Implementation of Viterbi algorithm in Kaldi. We also have implemented Viterbi decoding and their data structure in GPU and prove their correctness with the WER. To make the implementation run faster, we also have applied several optimization, i.e. that obtained from profiling result. Our implementation has a slightly higher WER, but have a performance speedup about 5.59-6.18x than Optimized Kaldi CPU implementation.

VII. FUTURE WORK

There are some improvement that can be applied from this paper, including:

1. Apply silence phone to detect end of utterance (end of sentence).
2. Apply traceback from previous Viterbi call, so the first frame decoded in that Viterbi call doesn't necessarily the first word of the sentence.
3. Apply the acoustic model to DNN-HMM (this paper only uses GMM-HMM).
4. Make a new structure so don't have to excess the memory usage while computing Viterbi probability, and can process much larger WFST.
5. Implement the Viterbi beam search and pruning so it reduces the space and amount of work needed when decoding.

REFERENCES

[1] A. L. Janin, "Speech Recognition on Vector Architectures," University of California at Berkeley, Berkeley, 2004.

[2] A. Segura Salvador, Characterization of Speech Recognition Systems on GPU Architectures, Barcelona: Facultat d'Informàtica de Barcelona, 2016.

[3] A. Argueta and D. Chiang, "Decoding with Finite-State Transducers on GPUs," 2017.

[4] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motl´ıcek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer and K. Vesely´, "The Kaldi speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, 2011.

[5] S. Karpagavalli and E. Chandra, "A Review on Automatic Speech Recognition Architecture and Approaches," *International Journal of Signal Processing, Image Processing and Pattern Recognition,* 2016.

[6] M. Mohri, F. Pereira and M. Riley, "Weighted Finite-State Transducers in Speech Recognition," *Computer Speech & Language,* 2002.

[7] N. Hermann and S. Ortmanns, "Dynamic programming search for continuous speech recognition," *IEEE Signal Processing Magazine,* vol. 16, no. 5, pp. 64-83, 1999.

[8] G. D. Forney, "The Viterbi Algorithm," *Proceedings of the IEEE,* vol. 61, pp. 268-278, 1973.

[9] H. Tabani, J.-M. Arnau, J. Tubella and A. González, "Performance Analysis and Optimization of Automatic Speech Recognition," *IEEE Transactions on Multi-Scale Computing Systems,* 2017.

[10] P. R. Dixon, D. A. Caseiro, T. Oonishi and S. Furui, "The Titech large vocabulary WFST speech recognition system," in *Automatic Speech Recognition & Understanding*, 2007.

[11] J. Chong, Y. Yi, A. Faria, N. Satish and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *the 1st Annual Workshop on Emerging Applications and Many Core Architecture*, 2008.

[12] J. Chong, E. Gonina and K. Keutzer, "Efficient Automatic Speech Recognition on the GPU," in *GPU Computing Gems Emerald Edition*, 2011.

[13] J. Chong, E. Gonina, Y. Yi and K. Keutzer, "A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit," in *INTERSPEECH*, 2009.

[14] 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.

[15] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein and I. Stoica, "Highly Available Transactions: Virtue and Limitations," in *VLDB*, 2013.

[16] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasiblity of consistent, available, partition-tolerant werb services," *SIGACT News,* vol. 33, no. 2, pp. 51-59, 2002.

[17] A. Lumsdaine, D. Gregor, B. Hendrickson and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters,* 2007.

[18] V. Narasiman, M. Shebanov, C. J. Lee, R. Miftakhutdinov, O. Mutlu and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *IEEE/ACM International Symposium on Microarchitecture*, 2011.