Accelerate Acoustic Likelihood Computations on GPU for Speech Recognition

Yong Liu, Zhen Zhang, Yujing Si, Qingwei Zhao, Yonghong Yan
The Key Laboratory of Speech Acoustics and Content Understanding, Chinese Academy of Sciences
Institute of Acoustics, Chinese Academy of Sciences
Beijing, China
{ liuyong, zhangzhen, siyujing, zhaoqingwei, yyan}@ hccl.ioa.ac.cn

***Abstract*—Acoustic likelihood computations is one of the most computationally intensive part in the large vocabulary continuous speech recognition (LVCSR). In this paper, we give an introduction of accelerating acoustic likelihood computations for graphical processing units(GPUs). According to the optimal method, we achieve 11 × speedup on the acoustic probability evaluation.**

Keywords: Speech Recognition, GMM,GPU

I. INTRODUCTION

After decades of research, the performance of automatic speech recognition (ASR) systems in real usage scenarios lags behind human level performance. The challenges involve speed and accuracy. A lot of researchers use some notable advances and new math or physical models in training. In the field of acoustic model, the Hidden Markov Model was be used[1, 2] and it had a great improvement on the speech recognition. Then, minimum phone error (MPE) [3, 4] were applied to the training of acoustic model. Recently, deep belief network (DBN) [5-7] was successfully introduced into the acoustic model and it shows the obvious performance of decreasing the phone error comparing with the GMM.

In the field of decoder[8] speed for LVCSR, a major achievement is the architecture of weighted finite-state transducers (WFST)[9-11]. WFST have an improvement on the speed than the tree-decoder.

Because of the development of hardware, a lot of researchers pay attention to the GPUs. Base on the architecture of multi-core and Single Instruction Multiple Data (SIMD), the GPU achieves an great speed up in the graph algorithm[12-14]. So, it will be a tendency to use GPU in the intensive computation, like speech recognition.

This paper is organized as follow. Section II gives an introduction to speech recognition and acoustic model. Section III shows the details of accelerate the GMM probability computations, including the basic principle, parallel reduction and matrix multiplication. Section IV gives the experiment results. Concluding remarks and future directions are reported in Section V.

II. SPEECH RECOGNITION AND ACOUSTIC MODEL

Speech recognition is the translation of spoken words into text. For a simple isolated word recognition, each spoken word be represented by a sequence of speech vectors or observations O, defined as $O = o_1,o_2,\ldots,o_T$, where o_t is the speech vector observed at time t. The isolated word recognition problem can be regarded as that of computing

$$\arg\max_i \{P(w_i \mid O)\}$$

Where w_i is the i'th vocabulary word. The probability can be computed by using Bayes' rule gives

$$P(w_i \mid O) = \frac{P(O \mid w_i)P(w_i)}{P(O)}$$

$P(O|w_i)$ is the acoustic model which means a match between an input feature and an element of a database of known phones. In general, the acoustic model is evaluated by Gaussian Mixture Model-Hidden Markov Model (GMM-HMM). The GMM observation probability is a part of acoustic model. In this paper, we only consider the GMM and the HMM state transition probabilities are ignored.

The acoustic likelihood for a GMM is defined as:

$$b_j(\vec{o}_t) = \sum_{c=1}^{C_j} \alpha_{jc} \frac{1}{\sqrt{(2\pi)^d \left|\Sigma_{jc}\right|}} \exp(-\frac{1}{2}(\vec{o}_t - \vec{u}_{jc})' \Sigma_{jc}^{-1} (\vec{o}_t - \vec{u}_{jc}))$$

where $b_j(\vec{o}_t)$ is the probability that distribution j generates the d-dimensional observation vector $\vec{o}_t$ at time t, C_j is the number of Gaussians in the distribution j, α_{jc} is the weight of Gaussian c in distribution j, μ_{jc} and Σ_{jc} are the mean vector and the covariance matrix of Gaussian c in distribution j.

Generally, the GMM probability can be quickened by Intel® Integrated Performance Primitives(IPP). By the Streaming SIMD Extensions (SSE), IPP can be only cost one fifth time than the normal loop instructions (by our experience).

The observation probability is one of the most computationally intensive phase. In the ASR system, it will cost 50%-70% computation time and almost becomes the bottleneck of decoder. In fact, just the active states' probability need to be calculated and we can use the beam pruning to decrease the computations.

There are plenty of methods to decrease the probability evaluation time[7, 15, 16]. In this paper, we use the GPUs and parallel reduction algorithm. Because of the multi-core and SIMD structure[17], more Arithmetic Logic Unit (ALU) and less Control Unit than CPU, GPU can solve more dense and high parallelism task.

Published by Atlantis Press, Paris, France.

III. ACCELERATE GMM LIKELIHOOD COMPUTATIONS ON GPU

According to Equation GMM probability, the natural logarithm likelihood of one Gaussian can be expressed as:

$$b_{jc}(\vec{o}_t) = \ln \alpha_{jc} - \frac{1}{2}\ln((2\pi)^d |\Sigma_{jc}|) - \frac{1}{2}\vec{u}'_{jc}\Sigma^{-1}_{jc}\vec{u}_{jc} + \vec{u}'_{jc}\Sigma^{-1}_{jc}\vec{o}_t - \frac{1}{2}\vec{o}_t'\Sigma^{-1}_{jc}\vec{o}_t$$

The first three terms are constant for a specific-GMM can be pre-computed. Denoting this term by h_{jc}, it is:

$$h_{jc} = \ln \alpha_{jc} - \frac{1}{2}\ln((2\pi)^d |\Sigma_{jc}|) - \frac{1}{2}\vec{u}'_{jc}\Sigma^{-1}_{jc}\vec{u}_{jc}$$

For the other two terms, we can re-denote mean and covariance:

$$U_{jc} = \vec{u}'_{jc}\Sigma^{-1}_{jc}$$

$$V_{jc} = Diag\ (-\frac{1}{2}\Sigma^{-1}_{jc})$$

where Σ^{-1}_{jc} is the diagonal covariance matrix. The likelihood for a single Gaussian can be expressed as:

$$b_{jc}(\vec{o}_t) = h_{jc} + U_{jc}\vec{o}_t + V'_{jc}\vec{o}_t^{\,2}$$

This equation means the computation can be expressed by a dot-product:

$$\vec{obs} = (1, o_1, o_2, ..., o_n, o_1^2, o_2^2, ..., o_n^2)$$

$$\vec{M} = (h, u_1\sigma^{-1}_{11}, ..., u_n\sigma^{-1}_{nn1}, -\frac{1}{2}\sigma^{-1}_{11}, ..., \frac{1}{2}\sigma^{-1}_{nn})$$

Then the GMM probability will be expressed:

$$\ln b_j(\vec{o}_t) = \log \overset{C}{\underset{c=1}{Add}}(\vec{obs} \bullet \vec{M}_{jc})$$

$$\log Add(x + y) = \ln(\exp(x) + \exp(y))$$

Because of the structure of GPU, it will be fit for the dot-multiply. The final logAdd operation can be finished by basic parallel reduction algorithm.

A. *Parallel Reduction*

Based on the above description, we can store the GMM model as followed:

where n is the dimensional of observation vector and the mix number of Gaussian is c. Similarly, the observation vector will be stored like this:

Assumed the warp num of GPU is 32 (the num of one SIMD group, which means 32 threads will be run for one instruction synchronous) and a block contains 256 threads. Besides that, the GMM contains STATENUM states, the num of Gaussians is 16 and the frame num is FRAMENUM for once processing. We will get the follow algorithm. The CPU function likes Fig. 1, then the GPU device kernel function likes Fig. 2. The GPU code has two part: host function and device kernel function. Host function will run on CPU and device function will run on GPU device.

```
dim3 grid(STATENUM,FRAMENUM);
dim3 block(256);
ReducationKernel<<<grid,block,1024>>>(model,
stateNum, dimMixGaussian, obv, dimObv, outProb)
```

Figure 1. Reduction algorithm for host.

```
__shared__ data[1024];
stateIndex←blockIdx.x;
frameIndex←blockIdx.y;
modelStride←(dimObv * 2 + 1) * dimMixGaussian;
baseModel←stateIndex * modelStride;
baseObv ←frameIndex * modelStride;
baseOutProb ← frameIndex * stateNum;
i←threadIdx.x;   x←threadIdx.x;
data[i] ←0;
while(i < modelStride)
{
      data[x] += model[baseModel + i] *
 obv[baseObv + i];
      i += 256;
}
__syncthreads();
if( x < 128 ) then
{  data[x] += data[x+128]; __syncthreads();}
if(x < 64) then
{  data[x] += data[x+64]; __syncthreads(); }
if(x < 32 ) then
{
      data[x] += data[x+32];
      data[x] += data[x+16];
      data[x] ← logAdd(data[x],data[x+8]);
      data[x] ← logAdd(data[x],data[x+4]);
      data[x] ← logAdd(data[x],data[x+2]);
      data[x] ← logAdd(data[x],data[x+1]);
}
if(0 == x) then
          outProb[baseOut + x] ← data[0];
```

Figure 2. Reduction kernel algorithm for GPU device

A block of a GPU will output the probability of a state for a frame and a grid will get FRAMENUM frames probability for STATENUM states. At first, the kernel will get the dot-multiply of model and observation, then loop accumulation until the dimension less than 256. Secondly, the kernel will run the parallel reduction for addition. Finally, the logAdd function will be called when the dimension equal the mix num of Gaussian-16.

The kernel has a problem of modifying. When the num of Gaussian is not 16, like 32 or 128, the code must be revised. When it is not the power of two, it will have more troubles.

In the algorithm of Fig. 2, the parallel reduction algorithm is used. The reduction can decrease the complexity of o(n) to o($\log_2 n$). It is very common for multi-core computation. The basic principle is following:

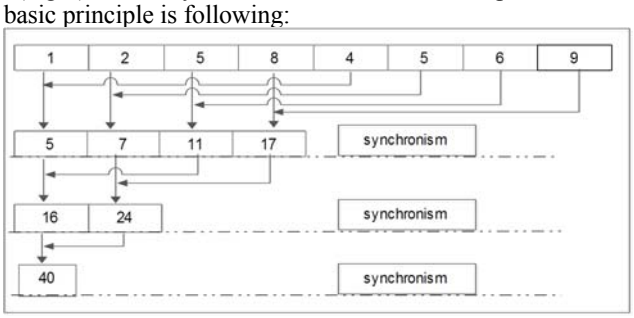
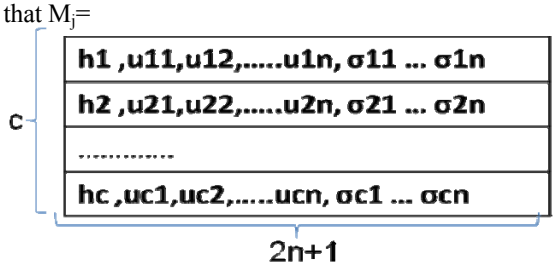


Figure 3. Parallel reduction algorithm

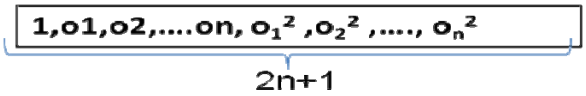
B. *Matrix Multiplication*

Because of the extend limit of the previous idea, we can change the store pattern of model and observation. Besides that, we will make most use of matrix multiply implementation [18]. Always, the basic matrix multiply library will effectively use the computation of process than the kernels by ourselves. Moreover, the matrix multiplication will be fit for the volatile mix num of Gaussian.

In our designer, the i-th state of the model will be saved like that M_j=



It is a c rows and (2n+1) cols matrix which c is the mix num of Gaussian and n is the dimension of observation. So, the j-th frame observation is saved as O_i=:



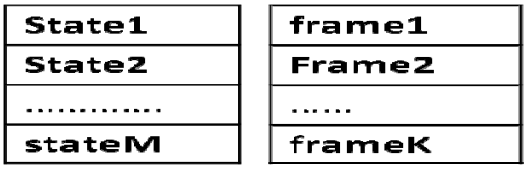
For a state, the probability of every Gaussian MP(c,i,j) is result of the matrix M(i) multiplying the matrix O(j). After the matrix multiply, the probability of a state P(i,j) is the logAdd of every dimension MP(c,i,j).

$$\mathrm{MP}(c, i, j) = M(i) * (O(i))^T$$

$$P(i, j) = \log \overset{C}{\underset{c=1}{Add}} (MP(c, i, j))$$

where MP(c, i, j) is the c-th mix probability of j-th frame for the i-th state and P(i, j) is the j-th frame probability for the i-th state.

When we deal with multi-state and multi-frame, the store pattern is :



Left is the model and right is the multi-frame observation. The model is a matrix of c* StateNum rows and (2n+1) cols. The observation is a matrix of frameNum rows and (2n+1) cols. The probability of every mix Gaussian MP is matrix M multiply the transposition of matrix O. Finally, the logAdd function will be called to deal the every c-dimension for the matrix MP. The algorithm is followed:

```
//Host code:
cublasSgemm (Model,Obv,MixProb);
dim3 block(256)
dim3 grid( (stateNum + 255)/256);
LogAddKernel<<<grid,block>>>(MixProb,  stateNum,
firstMixIndex, mixNumArray, outProb);

//Device code:
stateIndex←blockIdx.x * blockDim.x + threadIdx.x;
x←threadIdx.x;
firstMixLoc ← firstMixIndex[x];
mixNum← mixNumArray[x];
p ← MixProb[firstMixLoc];
for i=2:mixNum
    p = logAdd(prob, MixProb[i + firstMixLoc ]);
outProb[stateIndex]=p;
```

Figure 4. Host function and device kernel for matrix GMM method

In fact, the logSum for the mix probability is a highly parallel implementation by the general parallel case, however, we must consider the variable Gaussian mix num. So we use the loop operation which loses part of effective.

IV. EXPERIMENTS

We have several experiments for GMM computations speed between GPU and CPU. The feature of speech is 39-dimension, including The basic 12 MFCCs, the energy, the corresponding first and second derivatives. The configure of machine is the Intel(R) Xeon(R) X5650 CPU and NVIDIA Tesla C2050 GPU with the Windows Server 2008 OS.

At first, we have an small test for 12 frames and 5000 same states at a time, the probability computation of these states will be executed for 10 times. The comparison time shows in the Tab. 1.

Tab.1 displays part of problems. Because of the multi-times same data implementations, the cache of CPU and GPU will have a great effect on the speed. So, Tab. 1 just show the GPU is more fast than CPU, but the speed is not precise.

TABLE I. BASIC SPEED TEST

	Cost time	Speedup
IPP	15ms	base

GPU(Reduction)	3.92ms	3.8×
GPU(Matrix)	1.97ms	7.6×

Then, we have another experiment for a 181 frames speech feature file and 5786 states GMM model. The GPU will output the probability of 12 frames and 5786 states by matrix multiplication at a time, meanwhile the CPU only calculates a states and 12 frames once and then make double loops. Also, we have the single frame experiments.

TABLE II. SPEED COMPARISON EXPERIMENT

	Cost time	Speedup
IPP(Single frame)	20.5second	base
GPU(Single frame)	2.77second	7.4×
IPP(12-frame)	2.38second	base
GPU(12-frame)	0.205second	11.6×

Because of the multi-frame and multi-state, the delay of different warp for GPU was masked. So 12-frame computation could have a more speedup than the single.

V. CONCLUSION AND FUTURE WORK

Tab.2 show the GPU have 11.6× speedup. But we ignore the fact that the CPU could implement the beam pruning. After the pruning operation, only one filth states need to evaluate the probability according to the other decoder experiments. However, the GPU kernel function is very hard to prune. Despite the fact, the GPU also have a 2× speedup than IPP.

In fact, there is another method to calculate the acoustic model: using the artificial neural network (ANN) to replace the Gaussian Mixture Model. [5, 6] shows the deep belief networks have a better recognition accuracy. Besides, [6] displays the ANN training will have a 30 times speedup on GPU than the CPU. So the acoustic model of neural network is a tendency of speech recognition.

On the other hand, [19-21] give the potential speed for the decoder on GPU. The decoder on GPU will become a tendency to client-service system on speech recognition in order to get a quicker response. Combining with ANN acoustic model, the decoder will have a more fast speed. If there is a great enhancement in decoding, the speech recognition of actual time will likely come true.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China (Nos. 10925419, 90920302, 61072124, 11074275, 11161140319, 91120001) and the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant Nos. XDA06030100, XDA06030500).

REFERENCES

[1] K. F. LEE, "ON LARGE-VOCABULARY SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION," *SPEECH COMMUNICATION*, VOL. 7, PP. 375-379, 1988.

[2] L. R. RABINER, "A TUTORIAL ON HIDDEN MARKOV MODELS AND SELECTED APPLICATIONS IN SPEECH RECOGNITION," *PROCEEDINGS OF THE IEEE*, VOL. 77, PP. 257-286, 1989.

[3] D. POVEY, "DISCRIMINATIVE TRAINING FOR LARGE VOCABULARY SPEECH RECOGNITION," *CAMBRIDGE, UK: CAMBRIDGE UNIVERSITY*, 2004.

[4] D. POVEY AND P. C. WOODLAND, "MINIMUM PHONE ERROR AND I-SMOOTHING FOR IMPROVED DISCRIMINATIVE TRAINING," 2002, PP. I-105-I-108.

[5] A. MOHAMED, G. DAHL, AND G. HINTON, "ACOUSTIC MODELING USING DEEP BELIEF NETWORKS," *AUDIO, SPEECH, AND LANGUAGE PROCESSING, IEEE TRANSACTIONS ON*, PP. 1-1, 2010.

[6] G. E. DAHL, D. YU, L. DENG, AND A. ACERO, "CONTEXT-DEPENDENT PRE-TRAINED DEEP NEURAL NETWORKS FOR LARGE-VOCABULARY SPEECH RECOGNITION," *AUDIO, SPEECH, AND LANGUAGE PROCESSING, IEEE TRANSACTIONS ON*, VOL. 20, PP. 30-42, 2012.

[7] G. E. HINTON, S. OSINDERO, AND Y. W. TEH, "A FAST LEARNING ALGORITHM FOR DEEP BELIEF NETS," *NEURAL COMPUTATION*, VOL. 18, PP. 1527-1554, 2006.

[8] S. YOUNG, "A REVIEW OF LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION," *SIGNAL PROCESSING MAGAZINE, IEEE*, VOL. 13, P. 45, 1996.

[9] D. MOORE, J. DINES, M. DOSS, J. VEPA, O. CHENG, AND T. HAIN, "JUICER: A WEIGHTED FINITE-STATE TRANSDUCER SPEECH DECODER," *MACHINE LEARNING FOR MULTIMODAL INTERACTION*, PP. 285-296, 2006.

[10] M. MOHRI, F. PEREIRA, AND M. RILEY, "SPEECH RECOGNITION WITH WEIGHTED FINITE-STATE TRANSDUCERS," *HANDBOOK ON SPEECH PROCESSING AND SPEECH COMMUNICATION*, 2008.

[11] M. MOHRI, F. PEREIRA, AND M. RILEY, "WEIGHTED FINITE-STATE TRANSDUCERS IN SPEECH RECOGNITION," *COMPUTER SPEECH & LANGUAGE*, VOL. 16, PP. 69-88, 2002.

[12] S. HONG, S. KYUN KIM, T. OGUNTEBI, AND K. OLUKOTUN, "ACCELERATING CUDA GRAPH ALGORITHMS AT MAXIMUM WARP," *SIGPLAN NOTICES*, VOL. 46, P. 267, 2011.

[13] P. HARISH AND P. NARAYANAN, "ACCELERATING LARGE GRAPH ALGORITHMS ON THE GPU USING CUDA," *HIGH PERFORMANCE COMPUTING–HIPC 2007*, PP. 197-208, 2007.

[14] S. HONG, T. OGUNTEBI, AND K. OLUKOTUN, "EFFICIENT PARALLEL GRAPH EXPLORATION ON MULTI-CORE CPU AND GPU," IN *PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT)*, 2011, PP. 78-88.

[15] P. CARDINAL, P. DUMOUCHEL, G. BOULIANNE, AND M. COMEAU, "GPU ACCELERATED ACOUSTIC LIKELIHOOD COMPUTATIONS," IN *INTERSPEECH*, 2008.

[16] J. VANEK, J. TRMAL, J. V. PSUTKA, AND J. PSUTKA, "OPTIMIZED ACOUSTIC LIKELIHOODS COMPUTATION FOR NVIDIA AND ATI/AMD GRAPHICS PROCESSORS," *AUDIO, SPEECH, AND LANGUAGE PROCESSING, IEEE TRANSACTIONS ON*, VOL. 20, PP. 1818-1828, 2012.

[17] NVIDIA. (2012). *CUDA PROGRAMMING GUIDE*. AVAILABLE: HTTP://DEVELOPER.DOWNLOAD.NVIDIA.COM/COMPUTE/DEVZONE/DOCS/HTML/C/DOC/CUDA_C_PROGRAMMING_GUIDE.PDF

[18] NVIDIA. (2012). *CUDA TOOLKIT*. AVAILABLE: HTTP://DEVELOPER.NVIDIA.COM/CUDA/NVIDIA-GPU-COMPUTING-DOCUMENTATION

[19] J. CHONG, Y. YI, A. FARIA, N. SATISH, AND K. KEUTZER, "DATA-PARALLEL LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION ON GRAPHICS PROCESSORS," IN *PROCEEDINGS OF THE 1ST ANNUAL WORKSHOP ON EMERGING APPLICATIONS AND MANY CORE ARCHITECTURE (EAMA)*, 2008, PP. 23-35.

[20] J. CHONG, E. GONINA, AND K. KEUTZER, "EFFICIENT AUTOMATIC SPEECH RECOGNITION ON THE GPU," *CHAPTER IN GPU COMPUTING GEMS EMERALD EDITION, MORGAN KAUFMANN*, 2011.

[21] 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," PRESENTED AT THE INTERSPEECH, 2009.