A Parallel Implementation of Viterbi Training for Acoustic Models using Graphics Processing Units

ABSTRACT

Robust and accurate speech recognition systems can only be realized with adequately trained acoustic models. For common languages, state-of-the-art systems are trained on many thousands of hours of speech data and even with large clusters of machines the entire training process can take many weeks. To overcome this development bottleneck, we propose a parallel implementation of Viterbi training optimized for training Hidden-Markov-Model (HMM)-based acoustic models using highly parallel graphics processing units (GPUs). In this paper, we introduce Viterbi training, illustrate its application concurrency characteristics, data working set sizes, and describe the optimizations required for effective throughput on GPU processors. We demonstrate that the acoustic model training process is well-suited for GPUs. Using a single NVIDIA GTX580 GPU our proposed approach is shown to be 94.8x faster than a sequential CPU implementation, enabling a moderately sized acoustic model to be trained on 1000 hours of speech data in under **7 hours**. Moreover, we show that our implementation on a two-GPU system can perform 3.3x faster than a standard parallel reference implementation on a high-end 32-core Xeon server at 1/15th the cost. Our GPU-based training platform empowers research groups to rapidly evaluate new ideas and build accurate and robust acoustic models on very large training corpora at nominal cost.

Index Terms: Continuous Speech Recognition, Acoustic Model Training, Graphics Processing Unit

1. INTRODUCTION

The availability of very large training corpora (1000 hours and more) are empowering speech researchers to achieve ever higher accuracy on challenging speech recognition tasks. However, training acoustic models on these large corpora can take weeks, even on large clusters of workstations. This limits the number of methods and ideas that can be explored and validated in a timely manner. To overcome this bottleneck, in this paper we introduce a novel approach to rapidly train acoustic models using affordable ($500) off-the-shelf graphics processing units (GPU)s. Our platform can train an acoustic model at 94.8x faster than an off-the-shelf sequential CPU implementation. This platform is ideal for accelerating the exploration and validation of ideas for automatic speech recognition.

In state-of-the-art speech recognition systems, Hidden Markov Models (HMM) are used to model acoustic events. Each word within a speech recognition system is represented as a sequence of acoustic events, each state representing a specific phone-like unit as illustrated in Figure 1. During training, each segment of the acoustic signal, or more specifically, Mel-Frequency Cepstral Coefficients (MFCC) features extracted from the acoustic signal, are mapped to an acoustic state and the parameters of each model are updated to better model the acoustic signal observed during training. To further improve speech recognition accuracy, separate models are trained for each phonetic-context in which an acoustic event occurs. The acoustic context is defined by the neighboring phone models. This significantly increases the size of the model and the number of free parameters that must be estimated during training. A moderately-sized speech recognition system for English, for example, generally models 5000 to 10,000 phonetic events and contains tens of millions of free parameters.

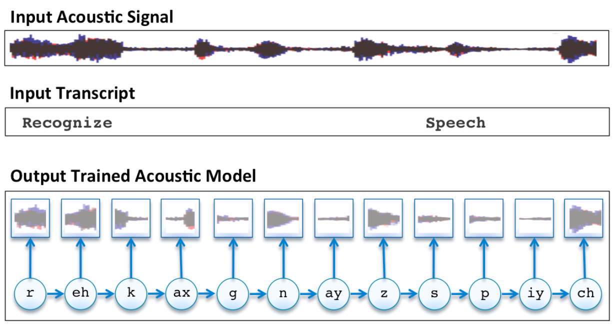

Figure 1: The acoustic model training application

To effectively train such models large amounts of training examples (speech samples) are required. Training of these models is highly data-parallel involving the aggregation of statistics from a large training data set possibly containing millions of utterances. Concurrency exists both between utterances and within an utterance, making the training process highly amenable for parallelization. However, constructing an efficient parallel implementation requires not only extensive application concurrency, but also a deep understanding of the available parallel computation resources.

As the development platform in this work we use the NVIDIA GTX580 GPU which contains 16 cores on-a-chip, two 16-wide single-precision SIMD pipelines in a core, as well as hardware managed cache and software managed memory scratch pad. The GPU is programmed using CUDA [1], a representative data-parallel manycore programming language where an application is organized into a sequential host program that is run on a CPU, and one or more parallel kernels running on a GPU. Each kernel describes a scalar sequential program that will be mapped across a set of par-

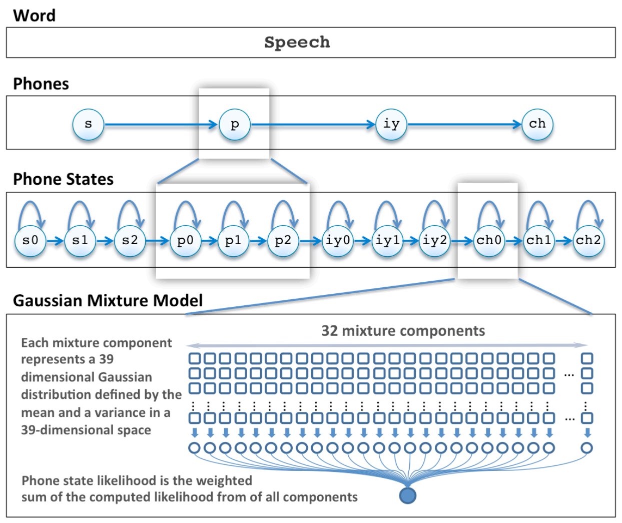
Word
Speech
Phones
s
p
iy
ch
Phone States
s0
s1
s2
p0
p1
p2
iy0
iy1
iy2
ch0
ch1
ch2
Gaussian Mixture Model
32 mixture components
Each mixture component represents a 39 dimensional Gaussian distribution defined by the mean and a variance in a 39-dimensional space
Phone state likelihood is the weighted sum of the computed likelihood from of all components

Figure 2: Acoustic representation of a word

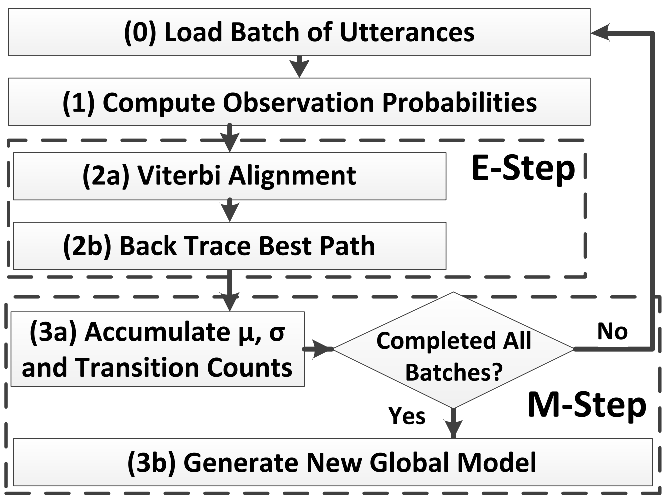
(0) Load Batch of Utterances
(1) Compute Observation Probabilities
(2a) Viterbi Alignment
E-Step
(2b) Back Trace Best Path
(3a) Accumulate μ, σ and Transition Counts
Completed All Batches?
No
Yes
M-Step
(3b) Generate New Global Model

Figure 3: Training flow for one training iteration

allel threads, which are organized into groups called *thread blocks*. The challenge is to effectively organize the training algorithm into threads and thread-blocks and leverage available memory resources and synchronization capabilities to efficiently execute on a manycore computation platform.

2. ACOUSTIC MODEL TRAINING

For any given word, the corresponding acoustic model is synthesised by concatenating phone models to make words as defined by a pronunciation dictionary. The parameters of these phone models are estimated from training data consisting of speech waveforms and their orthographic transcriptions. Each phone is represented by a continuous density HMM of the form illustrated in Figure 2 with transition probability parameters and output observation distributions.

Estimation of HMM parameters is commonly performed according to the the Maximum Likelihood Estimation (MLE) criterion, which maximizes the probability of the training samples with regard to the model. This is done by applying the Expectation-Maximization (EM) algorithm [2], which relies on maximizing the log-likelihood from incomplete data, by iteratively maximizing the expectation of log-likelihood from complete data. As shown in [3], this leads to the Baum-Welch reestimation formulas. The MLE criterion can be approximated by maximizing the probability of the best HMM state sequence for each training sample, given the model, which is known as segmental k-means [4] or Viterbi training. Viterbi training involves much less computational effort than Baum-Welch and in recent work has been shown to be just as effective as the Baum-Welch method [5] for training acoustic models for speech recognition.

2.1 Viterbi Training of Acoustic Models

Figure 3 illustrates the main steps of the Viterbi Training process, which include **(0)** loading training data, **(1)** computing observation probabilities, **(2)** assigning observations to specific states and Gaussian components within the model (E-Step), and **(3)** collecting statistics from the expectation step to reestimate model parameters (M-step).

Given a set of training observations $O^r, 1 \le r \le R$ and an HMM state sequence $1 < j < N$ the observations sequence is aligned to the state sequence via Viterbi alignment. This alignment results from maximizing

$$\phi_N(T) = \max_i \left[\phi_i(T) a_{iN}\right] \tag{1}$$

for $1 < i < N$ where

$$\phi_j(t) = b_j(o_t) \max \begin{cases} \phi_j(t-1)a_{jj} \\ \phi_{j-1}(t-1)a_{j-1j} \end{cases} \tag{2}$$

with initial conditions, $\phi_1(1) = 1$ and $\phi_j(1) = a_{1j} b_j(o_1)$, for $1 < j < N$. When observation likelihoods are modeled as mixture Gaussian densities the output probability $b_j(o_t)$ is as defined as:

$$b_j(o_t) = \sum_{m=1}^{M_j} c_{jm} \mathcal{N}\left(o_t; \mu_{jm}, \Sigma_{jm}\right) \tag{3}$$

where M_j is the number of mixture components in state j, c_{jm} is the weight of the m^{th} component and $\mathcal{N}(\cdot;,\mu,\Sigma)$ is a multivariate Gaussian with mean vector μ and covariance Σ. In Viterbi training, model parameters are updated based on the single-best alignment of individual observations to states and Gaussian components within states. From this alignment, transition probabilities are estimated from the relative frequencies

$$\hat{a}_{ij} = \frac{A_{ij}}{\sum_{k=2}^{N} A_{ik}} \tag{4}$$

where A_{ij} it the total number of transitions from state i to state j. The means and variances of the observation densities are updated using an indicator function $\psi^r_{jm}(t)$ which is 1 if o^r_t is associated with mixture component m of state j and is zero otherwise.

$$\hat{\mu}_{jm} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi^r_{jm}(t) o^r_t}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi^r_{jm}(t)} \tag{5}$$

$$\hat{\Sigma}_{jm} = \frac{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi^r_{jm}(t)(o^r_t - \hat{\mu}_{jm})(o^r_t - \hat{\mu}_{jm})'}{\sum_{r=1}^{R} \sum_{t=1}^{T_r} \psi^r_{jm}(t)} \tag{6}$$

and the mixture weights are computed based on the number of observations allocated to each component

$$c_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\psi_{jm}^{r}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T_r}\sum_{l=1}^{M}\psi_{jl}^{r}(t)} \quad (7)$$

2.2 Prior Works

There has been a number of efforts over the past decades to reduce the time required to train acoustic models for speech recognition. In 1990, Pepper *et al.* experimented with performing training on a set of computers organized in a ring [6]. In 1992, Foote *et al.* introduced an approach to distribute HMM training to a set of five loosely-coupled Armstrong II multi-processor network computers. In 1997, Yun *et al.* mapped the training algorithm to an FPGA infrastrcture [7] and in 2006 Poprescu *et al.* implemented acoustic model training on a MPI-based cluster with three nodes [8]. These prior works all achieved less than 3x speedup over sequential runs and thus have not been widely used.

The availability of general-purpose programmable GPU and data parallel programming models [1] has opened up new opportunities to train speech models at orders of magnitude faster than before. This is further empowered by new algorithms and implementation techniques that focus on parallel scalability [9], which expose the fine-grained concurrency in compute-intensive applications and exploits the concurrency on highly parallel manycore microprocessors.

In cuHMM [10], Liu implemented training of discrete HMMs on GPUs. This generic training engine, although effective for applications such as biological sequence analysis, is not appropriate for acoustic model training as it is unable to handle continuous observation models and cannot take advantage of the special left-right model structure used in speech recognition. In [11], Dixon *et al.* introduced techniques for fast acoustic likelihood computation in the context of a speech recognition decoder, but did not extended the work to the training process and in [12] Pangborn constructed an efficient implementation on the GPU for flow cytometry used in biology and immunology. This approach, however, only trained a single Gaussian mixture model (GMM) and is thus unsuitable for acoustic model training. In this paper we describe an optimized infrastructure for training HMMs, where we leverage the special left-right HMM model structure commonly used in speech recognition while heavily optimizing the observation probability computation.

3. VITERBI TRAINING ON MANYCORE PROCESSORS

Training is a highly data-parallel operation involving the aggregation of statistics from a large training data set possibly containing millions of utterances. Concurrency exists both between utterances and within an utterance, making the training process highly amenable for parallelization. However, constructing an efficient parallel implementation requires not only extensive knowledge in application concurrency, but also a deep understanding of the available parallel computation resources.

3.1 Step 1: Observation Probability Computation

The observation probability computation step implements Equation 3, and contains five levels of concurrency: among features, among mixture components, among phone states, among input observations, and among utterances. Table 1 illustrates the amount of concurrency available, the implied task size and data working set size (data size) if only one level of concurrency is exploited.

Table 1: Observation Probability Concurrency Analysis

Concurrency Opportunity	Degree of Concurrency	Task Size (# IPT[1])	Data Size (# values)
Features	39	2	2
Mixtures	32	100	80
Phone State	100	4000	2560
Observation	360	400K	256K
Utterance	1.1M	144M	270K

3.1.1 *Concurrency*

Concurrency measures how many threads can be working concurrently working on a piece of workload and make meaningful progress in completing the application. The acoustic models we are training are a standard Gaussian mixture acoustic model with 39-dimensions per mixture, with 32 mixtures components. Hence, there is a 39-way concurrency among the 39-dimensional features when computing the likelihood of an observation matching one Gaussian mixture component, and there is a 32-way concurrency when computing the likelihood of an observation, which is a process of comparing an input sample to each of the 32 components of a Gaussian mixture model representing a phone state in an acoustic model.

The input data set we use here transcribed audio segments from the AMI Meeting Corpus[2], which represents natural conversational speech in human interactions recorded in meetings. The audio segments are 3.6 seconds long on average, which consists of transcripts of 5-10 words. The training of one utterance is a process of matching a transcript of 5-10 words, which translates to an average of 30 phones or aproximately 100 phone states. Among the phone states in one utterance, there is a 100-way concurrency in comparing an input sample to the phone states.

Each input sample is a 25ms-window of observation. The observation windows overlap, and there are 100 overlapping window of observations taken each second. With an average of 3.6 seconds long utterances in the corpus we used, there is an average of 360 observations taken per utterance. Hence, we have a 360-way concurrency in computing the processing all the observations at the same time.

One training iteration can utilize as many as 1.1M utterances in a corpus of 1000 hours of audio. The observation probability of all the utterances can be computed concurrently, hence, the 1.1 million way concurrency that could be exploited.

3.1.2 *Task Size*

Task size is measured by the number of instructions that can execute in a thread before a synchronization event or task completion (Instructions per task or IPT) . For example, when a feature is assigned to a thread, each thread can only perform a "weighted-difference" calculation (in two

9[2]http://corpus.amiproject.org

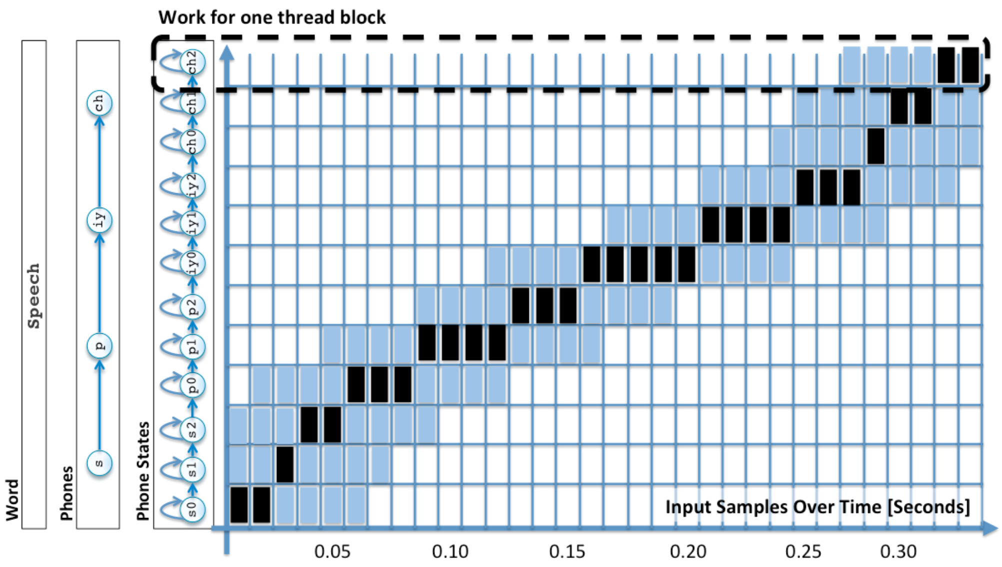

Figure 4: Observation probability calculation over time and phone states

instructions) before synchronizing to sum the "weighted-differences" between threads; for mixture-level parallelism, each thread sums the "weighted-differences" sequentially, and synchronize to calculate the weighted-sum of mixtures.

Alternatively, if each thread is responsible for computing the observation probability of one input sample with one phone state, it will be a 4000-instruction task.

Since there are on average 100 phone states per utterance, if one thread is responsible for computing the observation probability of all phone state for an utterance for one input sample, it be executing a 400k-instruction task.

If a thread is assigned to handle all obervation probability calculation for one utterance, it will be executing 144M instructions.

3.1.3 *Data Size*

Data size is the number of values the data working set contains in order to perform the task at each concurrency level within each thread. It is the number of values rather than bytes to make the metrics agnostic to the precision of the value used in a particular implementation.

For feature-level concurrency, each thread only require the mean and variance of a feature dimension, and thus only has a data size of two values before synchronization.

For mixture level concurrency, each thread requires 39 mean, 39 variance, as well as a likelihood constant and weight of the mixture, for a total of 80 values for the computation.

For phone state level concurrency, each thread requires access to all 32 mixtures in the existing acoustic model to compute the obervation probability between one observation and one phone state. This sums up to a total of 2560 values for the computation.

For observation level concurrency, each thread is responsible to compare one imput sample to all phone states in a training utterance. The data working set would involve an average of 100 phone states per utterance, which adds up to 256000 values representing 100 phone states.

For utterance level concurrency, each thread is responsible for all computations corresponding to one utterance, which include the phone state models as well as an average of 360 (39-value) input samples for utterances with an average of 3.6-second long. This set of data totals to 270 thousand values.

3.1.4 *Design Decisions*

For efficient implementation on the GPU, we choose the level of concurrency to maximize task granularity while allow the data working size to be small enough to fit into fast hardware-managed cache and software-managed memory scratch space.

Figure 4 illustrates the parallelization of the observation probability calculation. The goal is to select a level of concurrency that can be efficienty implemented on a GPU with a memory hierarchy similar to that of the NVIDIA Fermi architecture.

In Figure 4, we have the transcript represented as a sequence of phone states on the y-axis, and input acoustic samples represented as a sequence of extracted acoustic features over time on the x-axis. This creates a matrix of results that are computed to determine the best alignment between the transcript and the input utterance, so that the corresponding input samples can be used to improve the characteristics of the phone state.

We utilizd the phone state level concurrency with a data size of 2560 values represent 10KB of data working set of acoustic models per thread block. 10KB of shared data allows multiple thread blocks to execute on a multiprocessor with limited shared memory size of less than 50kB. The threads are organized to compute the observation samples for one phone state. This maximizes the phone state model data reuse from the on-chip shared memory in the calculation of the observation probabilities to construct a code-book for the E-step.

The input sample data for each utterance is transposed such that there are is one array for each feature dimension over time steps. This allows each thread in a *thread block* to have coalesced memory accesses when accessing the input sample features.

3.1.5 Applying Pruning

Pruning is an implementation optimization that focus on computing the observation probability only for the more likely candidate cells in the matrix of result illustrated in Figure 4. We studied two approaches to computing the observation probability: with or without pruning.

One approach is to compute the observation probabilities *without pruning* for all input samples, which corresponds to computing a result for all cells in each row in Figure 4. This is the most straight-forward implementation and is illustrated for the "ch2" phone state with the row enclosed in the dotted line. There is significate redundancies in this process, as the white cells in the rows have very low likelihoods of being part of the path through result matrix that represent an alignment between the transcript and the utterance, as illustrated with the cells in black.

The other approach is to only compute the obseration probability for the more likely candidate cells, which is often referred to an implementation "with pruning". Pruning in alignment is often performed by exploring the input samples sequentially over time. With our parallel approach, exploring input samples sequentially introduces sequential dependencies that requires expensive synchronizations between threads and thread blocks.

We propose an alternative approach for pruning. We observe that the acoustic model training flow involves the consideration of a transcript-input utterance pair more than 20 times. As the acoustic model improves in accuracy, we expect only small flunctuations in transcript to input utterance alignment. As a result, we compute only a margin of cells around the best alignment from the previous training iteration, illustrated in the shaded cells in Figure 4. Our alternative pruning approach is able to preserve all benefits of the parallelization design decisions, and has achieved more than 10x improvements for the observation probability computation step for utterances greater than 10 seconds long.

3.2 Step 2a: Alpha Computation

The Alpha computation is the forward pass of the Viterbi algorithm that uses dynamic programming to arrive at an optimal estimation of match likelihood between the transcript and the acoustic input (Equation 2). Computation is performed over each phone state and each time step as illustrated in Figure 5, and keeps track of the path of best alignment between the transcript and the acoustic input seen before that phone state and time step.

This computation contains two levels of concurrency: among phone states and among utterances. There exist data dependency between input observations over time, along the x-axis in Figure 5, which is illustrated with the white arrows for one time step. We observe that the computation of state likelihood at each time step depends on the results of the result from the previous observation. Table 2 illustrates the amount of concurrency available, the implied task size and data working set size (data size) if only one level of concurrency is exploited.

Table 2: Alpha Calculation Concurrency Analysis

Concurrency Opportunity	Degree of Concurrency	Task Size (# IPT[3])	Data Size (# values)
Phone State	100	12	5
Utterance	1.1M	432k	36.5K

3.2.1 Concurrency

The audio segments are on average 3.6 seconds long in the conversational audio corpus we use, with each audio segment transcribing to 5-10 words utterances. This translates to an average of 30 phones or approximately 100 phone states. The alpha calculation, compute the values at the current time step based on the values in the previous time step. The application concurrency at the phone state level is approximately a 100-way concurrency.

As with the observation probability calculations, one training iteration can utilize as many as 1.1M utterances in a corpus of 1000 hours of audio. The alpha values for multiple utterances can be computed concurrently, hence, the 1.1 million way concurrency can be exploited.

3.2.2 Task Size

If each thread is responsible for computing the alpha value of one input sample with one phone state, it will be a 12-instruction task, which involves the computation illustrated with the white arrows in Figure 5, and described in Equation 2.

If a thread is assigned to handle all alpha calculations for one utterance, it will be executing 432K (12 instructions per state-observation $\times$ 100 phone states $\times$ 360 input observations) instructions before a thread synchronization is required.

3.2.3 Data Size

For phone state level concurrency, each thread requires access to five values. The model we consider (y-axis in Figure 5) allows the alignment between the transcript and acoustic input to stay in one phone state or move forward in time by one phone step. Each phone state only have states that only connect to themselves and to their next states. This limit the data working set of each thread to: two alpha values of the previous time step, two transition probabilities, and one observation probability (computed in Step 1).

For utterance level concurrency, each a thread is responsible for all computations corresponding to one utterance. The data working set involve keeping track of the data working set of iterating through the time steps, as well as retaining the backtrack information for extracting the optimal alignment. For iterating through the time steps, we have 100 alpha values from the previous time step, 200 transition probabilities (white arrows in Figure 5), 100 observation probability (computed in Step 1), and 100 computed alpha values from the current time step, which adds up to 500 values.

For retaining the backtrack information, we must store the back pointer for each phone state and each time step, which is 36k values (100 phone states $\times$ 360 input observations).

For utterance level concurrency, we have a data working set of 37k values (500 frequently accessed values + 36k backtrack log).

3.2.4 Design Decisions

As with the observation probability calculations, we choose the level of concurrency to maximize task granularity while allow the data working size to be small enough to fit into fast hardware-managed cache and software-managed memory scratch space. Thus the goal is to select a level of concurrency that can be efficiently implemented on a GPU with a memory hierarchy similar to that of the NVIDIA

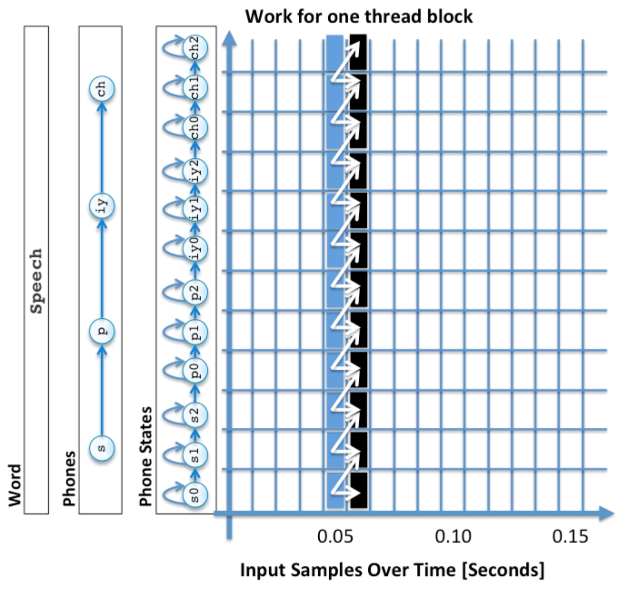

Figure 5: Alpha calculation over time and phone states

Fermi architecture.

In Figure 5, we have the transcript represented as a sequence of phone states on the y-axis, and input acoustic samples represented as a sequence of extracted acoustic features over time on the x-axis. This creates a matrix of results which represent the best alignment between the transcript and the input utterance upto that point in time.

The computation is implemented as a time synchronous operation, utilizing phone state concurrency, where the computation for one acoustic input sample is dependent on the results from the previous input sample. Each thread block then iterates sequentially over the time steps, with each thread handling the computation for one phone state.

We implement double buffering of the intermediate results (100 previous alpha values and 100 current alpha values) in shared memory, cache model transition probabilities (200 values) in shared memory, and stream in the observation probabilities (100 values) to enable fast iteration of computation without incurring excessive memory operations to off-chip memory. The backtrack information is streamed out as output to global memory, as it is required by Step 2b.

The time synchronous steps are implemented with a local _syncthreads() at the thread block level to minimize the overhead of synchrization for each time step.

3.3 Step 2b: Backtracking Computation

The backtrack computation traces the one-best path that demonstrates the best alignment of transcript to the input utterance. Backtrack starts from the end of the phone state sequence representing the transcript and the end of the acoustic input utterance, and backtracks step by step to the beginning of the phone state sequence. Backtrack is a highly sequential process over the input observations and phone states, and is identical to pointer chasing. Utterance level concurrency is often considered the only level of concurrency in the application. Table 3 illustrates the amount of concurrency available, the implied task size and data working set size (data size) if only one level of concurrency is exploited.

Table 3: Backtrace Calculation Concurrency Analysis

Concurrency Opportunity	Degree of Concurrency	Task Size (# IPT[4])	Data Size (# values)
Utterance	1.1M	720	36k

3.3.1 *Concurrency*

As with the observation probability calculations and the alpha computation, one training iteration can utilize as many as 1.1M utterances in a corpus of 1000 hours of audio. Backtracking for multiple utterances can be computed concurrently, hence, the 1.1 million way concurrency can be exploited.

3.3.2 *Task Size*

Calculating the backtrace through a single phone state and input observation pair requires just 2 instructions. If single thread is responsible for computing the backtrace for an utterance with an average of 360 input observations and diagonal backtrace, would have approximately 720-instruction task.

3.3.3 *Data Size*

Data size is the number of values the data working set contains in order to perform the task at each concurrency level within each thread. Althrough the backtrack only touchs a small percentage of the grid of back pointer values produced in Step 2a, the entire grid back pointers could be touched depending on the pointers. Since the entire grid must be available to a single thread, the average data size is 36K (100 phone states $\times$ 360 input observations).

3.3.4 *Design Decisions*

From Table 3 it is clear that the only feasible currency is at the utterance level. Implementing an utterance level parallelism naively causes a severe performance bottleneck. The pointer chasing operation will involve two memory round trips per time step, which can take 100,000s processor cycles to backtrack 360 steps.

Figure 6 illustrate a prefetch-based implementation optimization we implemented for this step. The shaded block of data illustrates the blocks of backtrack information that is being prefetched in to shared memory. By pre-fetching 32 time steps for 4 phone states of potentially accessible values as a batch into the shared memory on the GPU and perform the backtrack from shared memory, we fully utilize the load bandwidth and minimized memory latency caused by the pointer chasing operations in the backtrack process to global memory[5].

3.4 Step 3: Maximization Step

The maximization step takes the aligned and labeled input observations to update the aggregated statistics in the acoustic model according to equations 4, 5, 6 and 7. The process is an instance of histogram generation, where we compute the statistical distribution of the training data set.

The histogram bins are the Gaussian mixtures in the GMMs representing triphone states in the acoustic speech model. An acoustic speech model typically contains 10,000 triphone states in an acoustic codebook. For a 32 mixture Gaussian model, there can be as many as 320,000 histogram

9[5]The prefetch optimization could also be applied to the CPU implementation but is not included in this comparison.

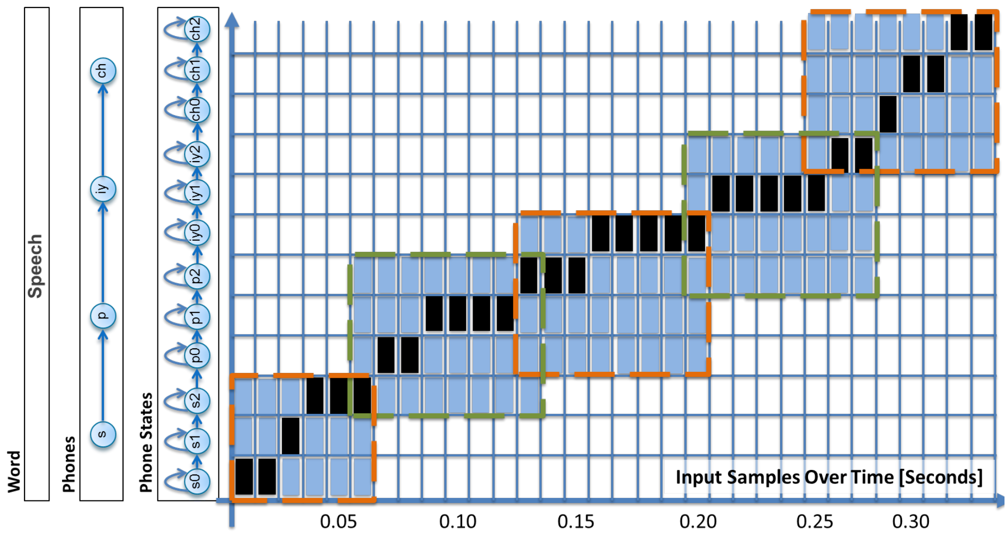

Figure 6: Backtrace calculation over time and phone states

bins to accumulate to, each represented by a 39-dimensional mean and variance value-pair. This number of bins would be too large to fit in the last level cache of today's microprocessors, making an efficient histogram generation implementation challenging.

To further compound the issue, a typical training data set for speech recognition set may involve thousands of hours of audio, stored as segments of audio that represents speech utterances separated by silences, taking tens to hundreds of gigabytes of storage. With such a large amount of training data, the histogram counts run into issues of overflow. The issue of overflow can be avoided by continuously recalculating transition probabilities of the HMMs and the means and variances of each of the Gaussians in the final acoustic model. Unfortunately this solutions run into issues of underflow once large amounts of data has been processed.

To counter these deficiencies we use a hybrid local-global accumulation method to efficiently aggregate the statistics in the histogram generation process. In the local accumulation step, batches of utterances are processed in parallel, and the histogram counts for each batch are accumulated. The batch sizes, ranging from 50 - 200 utterances, prevent histogram counts from overflowing. Once the histograms from a batch are accumulated they are merged into a global model.

Given the training data set and histogram characteristic, we map each utterance to a thread block on the GPU, and first aggregate the histogram information within an utterance locally, then merging the results from each thread block to the main histogram globally[6]. While this does not solve the unchangeably large number of histogram bins, it does alleviate the potential sequentialization bottlenecks at some histogram bins when thousands of thread context concurrently performing memory operations on a popular triphone state. For the aggregation of floating-point values, our implementation extensively leverages floating-point atomic additions to global memory in the NVIDIA Fermi architecture.

4. RESULTS

4.1 Experimental Evaluation

We evaluated the effectiveness of our proposed GPU-based acoustic model training procedure by first comparing the time required to perform one iteration of Viterbi training to a single-thread CPU implementation. The speech corpora used in this evaluation consisted of 122hrs and approximately 150k utterances of speech collected from headset, lapel and far-field microphones from 168 sessions from the AMI Meeting Corpus[7]. This data was replicated to generate the larger training sets up to 10,000 hours. An initial cross-word, context dependent triphone model was trained on this corpora using HTK [13] and the resulting model consisted of 8000 codebooks with a maximum of 32 Gaussian components per codebook. Each model consisted of a three-state left-to-right hidden Markov model (HMM) with two transitions per state, a local transition to the current state and a transition to the neighboring state to the right. Models were trained using 39-dimension acoustic features, which combined 13-dimension MFCCs with its delta and delta-delta components.

For evaluation we used the NVIDIA GTX580 GPU in an Intel Core i7-2600k CPU-based host platform. The GPU has 16 cores each with dual issue 16-way vector arithmetic units running at 1.54GHz. Its processor architecture allows a theoretical maximum of two single-precision floating point operations (SP FLOP) per cycle, resulting in a maximum of 1.58 TeraFLOP of peak performance per second. The GPU device has 1.5GB of GDDR5 memory. For CPU runs, we used a 4-socket Intel Xeon X7550 Server running at 2.00GHz with an aggregate of 32 cores and 128GB of memory. For compilation, we used g++ 4.5.0 and NVCC 3.2 targeting GPU Compute Capability 2.0.

4.2 Performance Analysis

Figure 7 illustrates the training time used for various training set sizes. As expected, training time scales linearly with increase in training set size. The CPU run-times were measured with and without pruning using Viterbi training. Using one hour of training time, we can process 7.2 hours of

9[6]Numerical issues that usually plague likelihood accumulations are handled when merging local values with global values.

9[7]http://corpus.amiproject.org

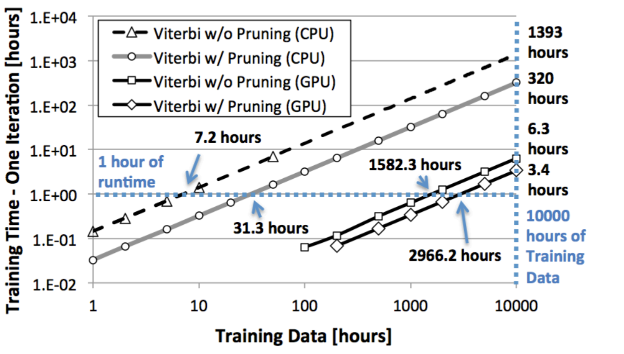

Figure 7: Training time for one training iteration of 32-component GMM models

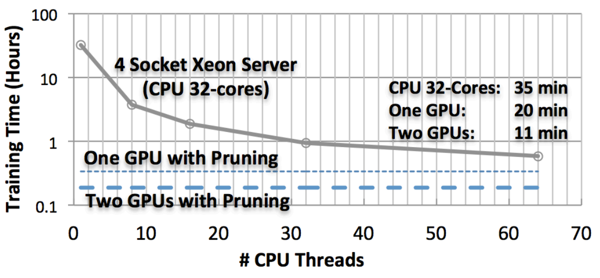

Figure 8: Training time for one iteration with 32 component Gaussian on a 1000-hour AMI corpus

training data with no pruning, and 31.3 hours of data with pruning. With GPU-based implementation, we can process 1582.3 hours of data with no pruning and 2966.2 hours with pruning using an equivalent manycore Viterbi training routine. The GPU-based implementation with pruning is 94.8x faster a similar algorithm with pruning running sequentially on the Xeon server CPU. Given a 10,000-hour training corpus, a single GPU is able to perform one iteration of Viterbi training in just 3.4 hours. The same 10,000-hour training corpus would take 1393 hours without pruning and 320 hours for the CPU case when pruning is enabled. With a multi-GPU setup, our platform enables training on very large corpora even with limited computational resources.

One approach to train a full acoustic model involves iteratively increasing the number of Gaussian components from a single Gaussian to a much large number via *mixture splitting*. This process is computationally expensive and involves performing 3-5 EM training iterations for acoustic models with increasingly large observation models (GMMs). For example, to train a 32-component GMM the training time required on a single GPU for a 1000-hour training set is illustrated in Table 4. We assume an average of four EM iterations are used per mixture count, and accumulate the measured execution time for our Viterbi training routine on the GPU. Taking into account the time necessary for performing mixture splitting, we expect to be able to train an acoustic model with 1000-hour of data over night in less than 6.85 hours, with 0.37 hours accounted for mixture splitting and cluster tying overhead. Our rapid training system can also allow new ideas and concepts to be tested on a 100-hour training set in less than one hour.

Figure 8 illustrates the training time achievable with a four-socket 32-core Xeon server costing over $30,000. Given a 1000-hour training set, the utterances are dynamically scheduled onto a varying number of CPU threads. We see that a single GPU system out performs the 32-core Xeon server by 82%, and that a two-GTX580 desktop system that cost less than $2,000 can be 3.3x faster in performing one EM iteration with 32-component GMMs on a 1000-hour training set.

Figure 10 illustrates benefit of pruning with the observation probability computation time measured for the implementations without pruning and with various degrees of pruning. We see a fixed 0.2ms overhead for performing the observation probability computation, with include setup processes such as copying the acoustic models of the phone states into the shared memory. The computation time without pruning increases quadratically with respect to the input utterance length, and computation time with pruning scales linearly with respect to the input utterance length. While the actual computation times fluctuates for different utterance over the range of utterance length, the line of best fit and the quadratic and linear relationships can be clearly observed as the slop of log-log plot. The benefit of pruning increases to more than a order of magnitude speedup for utterances longer than 10 seconds long.

Figure 10 also illustrates that having pruning sample margins of less than 64 samples does not result in significantly more speedup. A margin of 64 samples means that we anticipate shifts of alignment of more than half a second long, we consider wide enough to accommodate small variations created by updates to the acoustic models in their training process.

Figure 9 illustrates the step-wise timing break down of one training iteration of a 1000-hour training set. The runs are separated into batches to allow the data working set within a batch to fit on the GPU global memory.

Without pruning, as shown in the top pie chart in Figure 9, the most compute-intensive observation probability computation uses 68.4% of the total execution time. We have achieved 1.23 instructions per cycle (IPC) on the Fermi GPU architecture, which is 61.5% of the peak execution efficiency. The E-Step includes the highly sequential backtracking process, with the various technique described in section 3.2 and 3.3, only 12.9% of the time is spent here. By using the local-global reduction technique and leveraging efficient floating-point atomic-add capability supported by the hardware, the M-Step takes only 6.6% of total execution time. The utterance load step is taking up a quite significant 12% of the total executing time.

With pruning, as shown in the bottom pie chart in Figure 9, the most compute-intensive observation probability computation uses 32.1% of the total execution time. The E-Step includes the highly sequential backtracking process, with the various technique described in section 3.2 and 3.3, 27.8% of the time is spent here. By using the local-global reduction technique and leveraging efficient floating-point atomic-add capability supported by the hardware, the M-Step takes only 14.0% of total execution time. The utterance load step is taking up a quite significant 26.1% of the total executing time. This step is expected to be significantly faster after an re-factoring the code base and is on the list of future work.

4.3 Discussion

Table 4: Training time for a 1000-hour training set for models with different numbers of Gaussian mixtures

# components in GMM	1	2	4	8	16	32
One Iteration (hours)	0.247	0.248	0.252	0.267	0.282	0.323
Four Iterations (hours)	0.987	0.994	1.010	1.067	1.130	1.293
Total (hours)	6.850 = 6.480 + 0.370(overhead)					

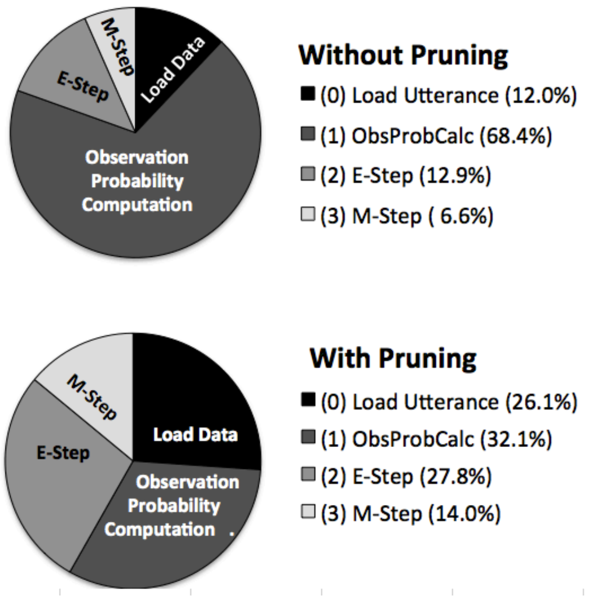

Figure 9: GPU implementation component-wise timing breakdown (1000-hour AMI corpus)

The AMI Meeting Corpus consists of relatively short conversational utterances that are on average only 3.6 seconds long. 69% of the utterances are less than 2 seconds long. With short utterances, performing force-alignment without pruning incurs little overhead.

5. CONCLUSION

We presented a new framework for rapid training of acoustic models using the GPU. We focus on Viterbi training and shown that using a single GPU, our proposed approach is 94.8x faster than a sequential CPU implementation. Training an acoustic model with 8000 codebook of 32-component Gaussian mixtures on 1000 hours of speech would take just under **7 hours**. Using a scalable pruning algorithm, a single GPU system can perform 82% faster and a two-GPU system can be 3.3x faster when compared to an off-the-shelf acoustic model training engine running on a high-end 32-core Xeon server. Our GPU-based training platform empowers research groups to rapidly evaluate new ideas and build accurate and robust acoustic models on very large training corpora.

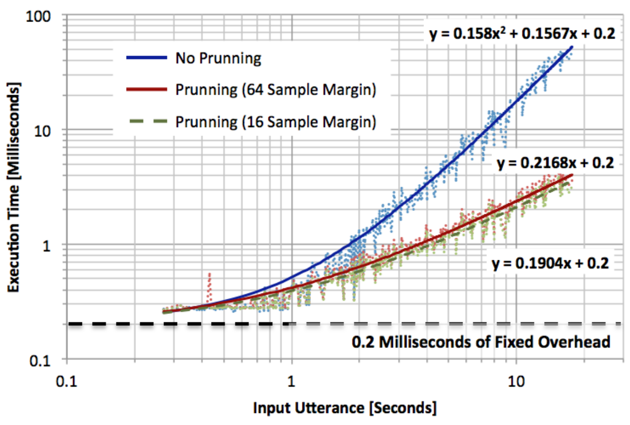

Figure 10: Observation probability computation time for various utterance lengths

6. REFERENCES

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, pp. 40–53, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[3] X. Huang, Y. Ariki, and M. Jack, *Hidden Markov Models for Speech Recognition.* New York, NY, USA: Columbia University Press, 1990.

[4] B. Juang and L. Rabiner, "The segmental K-means algorithm for estimating parameters of hidden Markov models," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 38, no. 9, pp. 1639 – 1641, Sep. 1990.

[5] L. Rodríguez and I. Torres, "Comparative study of the baum-welch and viterbi training algorithms applied to read and spontaneous speech recognition," in *Pattern Recognition and Image Analysis*, ser. Lecture Notes in Computer Science, F. Perales, A. Campilho, N. de la Blanca, and A. Sanfeliu, Eds. Springer Berlin / Heidelberg, 2003, vol. 2652, pp. 847–857, 10.1007/978-3-540-44871-6-98.

[6] D. Pepper, I. Barnwell, T.P., and M. Clements, "Using a ring parallel processor for hidden Markov model training," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 38, no. 2, pp. 366 –369, Feb. 1990.

[7] H.-K. Yun, A. Smith, and H. Silverman, "Speech recognition HMM training on reconfigurable parallel processor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1997, pp. 242 –243.

[8] V. Popescu, C. Burileanu, M. Rafaila, and R. Calimanescu, "Parallel training algorithms for continuous speech recognition, implemented in a message passing framework," in *14th European Signal Proc. Conf. (EUSIPCO'06), Florence, Italy*, Sept. 4-8 2006.

[9] K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y.-K. Chen, W. Sung, and K. Keutzer, "Parallel scalability in speech recognition," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 124 –135, 2009.

[10] C. Liu, "cuHMM: a CUDA implementation of hidden

Markov model training and classification," http://code.google.com/p/chmm, May 2009, online.

[11] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 2009*, 2009, pp. 4321 –4324.

[12] A. D. Pangborn, "Scalable data clustering using GPUs," Master's thesis, Rochester Inst. of Tech., Rochester, New York, May 2010.

[13] P. C. Woodland, C. J. Leggetter, J. J. Odell, V. Valtchev, and S. Young, "The 1994 HTK large vocabulary speech recognition system," *Proc. of ICASSP 95*, pp. 73 – 76, May 1995.