GPU-Accelerated Gaussian Clustering for fMPE Discriminative Training
Yu Shi, Frank Seide, and Frank K. Soong
Speech Group, Microsoft Research Asia, Beijing, China
{yushi,fseide,frankkps}@microsoft.com

Abstract
The Graphics Processing Unit (GPU) has extended its applications from its original graphic rendering to more general scientific computation. Through massive parallelization, state-of-the-art GPUs can deliver 200 billion floating-point operations per second (0.2 TFLOPS) on a single consumer-priced graphics card. This paper describes our attempt in leveraging GPUs for efficient HMM model training. We show that using GPUs for a specific example of Gaussian clustering, as required in fMPE, or feature-domain Minimum Phone Error discriminative training, can be highly desirable. The clustering of huge number of Gaussians is very time consuming due to the enormous model size in current LVCSR systems. Combining an NVidia GeForce 8800 Ultra GPU against an Intel Pentium 4 implementation, we find that our brute-force GPU implementation is 14 times faster overall than a CPU implementation. Overall execution time of the Gaussian clustering is reduced from more than 17 times compared with a CPU implementation. Overall execution time of the Gaussian clustering is reduced from more than 24 hours to 100 minutes, a 14 times speedup.

This paper is organized as follows: Section 2 gives a brief overview on the GPU hardware and programming environment, Section 3 briefly reviews fMPE method; The Gaussian clustering method is described in Section 4, followed by the GPU implementation in Section 5. Runtime comparisons of GPU vs. CPU-based Gaussian clustering are given in Section 6.

1. Introduction
In today’s speech recognition systems, acoustic models can easily have a million of Gaussians or more, which makes the training procedure very time consuming. One solution is that of distributed architectures. The training data is split into several subsets and processed in parallel on a large number of computing servers. After all jobs have finished, the accumulators are picked up and merged for updating the model. This is repeated until the acoustic models obtained.

An emerging alternative solution is the use of GPUs (Graphics Processing Units). GPUs are highly specialized architectures designed for graphics rendering. Driven by the computer gaming industry, graphics cards now achieve a vast 200 GFLOPS through massive parallelism, while remaining consumer-priced commodity hardware. This has led to a trend to leverage GPUs to accelerate computationally intensive applications, including scientific applications, referred to as GPGPU (General-Purpose computation on the GPU).

This paper represents our first step towards speeding up HMM model training using GPUs. A baby step actually, aimed at gaining an understanding how GPUs work and can be leveraged. Therefore, we focus on one very narrow, specific part—the Gaussian clustering procedure required for the discriminative training process called feature-domain Minimum Phone Error training (fMPE, [1]). fMPE significantly improves word error rates (WER) compared to maximum-likelihood (ML) training—by 6% relative in our tasks of recognizing conversational telephone speech (CTS) and voicemails. Practical use of fMPE requires all Gaussians of all HMM states, possibly hundreds of thousands, to be clustered into a smaller set of Gaussians, such as 40k. We chose to focus on accelerating this step because it is slow and difficult to be run on a distributed system while its GPU implementation is relatively simple yet exposes us to the relevant elements of GPU programming.

GPU-accelerated and CPU-only runtimes of the clustering are compared for commercial state-of-the-art hardware: an NVidia GeForce 8800 Ultra graphics card and a 3 GHz Intel Pentium 4. The graphics card has 128 stream processors clocked at 1.5 GHz, and can, across all processors, execute 192 billion multiply-and-accumulate operations in single-precision floating point per second. We developed our GPU code using NVidia’s C-language programming API “CUDA” (Compute Unified Device Architecture) version 1.0 [2].

Our experiments show that all steps in the Gaussian clustering can be significantly accelerated using GPUs, by as much as 17 times compared with a CPU implementation. Overall execution time of the Gaussian clustering is reduced from more than 24 hours to 100 minutes, a 14 times speedup.

This paper is organized as follows: Section 2 gives a brief overview on the GPU hardware and programming environment, Section 3 briefly reviews fMPE method; The Gaussian clustering method is described in Section 4, followed by the GPU implementation in Section 5. Runtime comparisons of GPU vs. CPU-based Gaussian clustering are given in Section 6.

2. GPU Architecture
2.1. The GPU hardware
The GeForce 8800 Ultra is comprised of 16 “multiprocessors” (MP), where each MP has 8 streaming processors (SPs). The SPs, although originally designed for graphics rendering, are nearly full-fledged processors providing a IEEE single-precision ALU (Arithmetic Logical Unit), pointers, and conditional execution/branching—but no stack and therefore no subroutine calls. All 8 SPs execute the same instruction, but on distinct set of registers and data (SIMD, Single Instruction, Multiple Data). Running at 1.5 GHz, the total of 128 SPs allows for 192 billion operations per second (0.2 TFLOPS counting multiply-and-accumulate as one instruction). All SPs in one MP share one 16 KB fast-access level-1 (L1) cache that allows sharing of data between threads. Unlike common CPU caches, this “cache” has its own dedicated address space, and the programmer must manually move data between it and the main GPU DRAM. This is critical—DRAM accesses can be over 100 times slower. There is no sharing, communication, or synchronization between threads executing on different MPs.

2.2. Programming the GPU
NVidia provides a programming environment CUDA (Compute Unified Device Architecture) [2]. CUDA targets the use of GPUs for computational purposes rather than graphics rendering, and allows to write GPU programs in a variant of the C programming language.
In CUDA, the GPU is viewed as a computer device that executes a very high number—thousands or even millions—of threads in parallel. A function can be compiled to the instruction set, and the resulting program, called a kernel, can be downloaded to the device. Both the host (the main CPU) and the device maintain their own DRAM, and CUDA provides APIs for copying data from one DRAM to the other.

The batch of threads that executes a kernel is organized as a grid of thread blocks as illustrated in Fig. 1. A thread block is a batch of threads that execute on the same multiprocessor and as such can efficiently share data through the fast shared L1 cache and synchronize execution. Each thread can identify itself through its thread id within the block. Due to its heritage from graphics rendering, a thread id can be a two or three-dimensional index.

Thread blocks of same dimensionality and size that execute the same kernel can be batched together into a grid of blocks to allow parallel execution on the 16 multiprocessors.

Programming the GPU is, despite the familiar programming language, very different. First, one must refactor the task into (thousands of) near-identical threads. E.g., GPU threading is so efficient that evaluating a single Gaussian can be parallelized by including a maximum of 16 threads in the L1 cache. Another is to “coalesce” memory accesses across multiple threads—reading a single byte in a single thread takes the same number of cycles as reading 16 bytes each from 16 parallel threads—256 bytes at once—if the addresses are aligned to DRAM page boundaries in a certain way.

3. fMPE

fMPE [1] is a technique in speech recognition where speech feature vectors are modified through a transform that is trained discriminatively using the Minimum Phone Error (MPE) criterion. Modified feature vectors y(t) are obtained from the original features x(t) as:

\[ y(t) = x(t) + M \cdot h(t) \]  
\[ h_i(t) = \frac{b_i(x(t))}{\sum_j b_j(x(t))} \]

where \( t \) is the (discrete) time, \( h(t) \) a vector of posterior probabilities, and \( b \) the likelihood of a Gaussian. In many implementations, including ours, \( h(t) \) is further expanded by including left and right neighbor frames. I.e., the vector of posterior probabilities from a large base of Gaussians is projected into the feature space and added to the original feature. The projection matrix \( M \) is estimated by a linear method. Each iteration of fMPE training involves three passes over the data: (1) accumulate normal MPE statistics; (2) accumulate fMPE statistics, and (3) ML update with the newly transformed data via a single-pass retraining.

In the simplest case, the set of Gaussians is just the set of all Gaussians of the acoustic model. However, with acoustic models of many, sometimes over hundreds of thousands of Gaussians, the above transform becomes infeasible in practice. Thus, we need to cluster Gaussians to a smaller set, in our case 40k.

4. Gaussian Clustering Algorithm

The objective of Gaussian clustering is to use a smaller set of Gaussians in place of the original infeasibly high set of Gaussians. We build a binary tree with a specified depth such that the set of leaf nodes of the tree represents a partition of the set of the Gaussians in the acoustic space. We adopt a clustering similar to the LBG algorithm [3] to construct such a tree, but using a symmetric divergence measure between two Gaussians as the distortion measure [4]. The generic algorithm is summarized in Fig. 2.

![Figure 2: Gaussian clustering algorithm.](image)

The symmetric divergence between two Gaussians, say \( \mathcal{N}(x; \mu_1, \Sigma_1) \) and \( \mathcal{N}(x; \mu_2, \Sigma_2) \) used in [4] is expressed as:

\[
D(i,j) = \frac{1}{2} tr[ (\Sigma_i^{-1} - \Sigma_j^{-1}) ] + \frac{1}{2} tr[ (\Sigma_i^{-1} + \Sigma_j^{-1}) (\mu_i - \mu_j)(\mu_i - \mu_j)^T ]
\]

(3)

where \( tr[\cdot] \) denotes the trace of a matrix. In diagonal case the divergence measure is given by:

\[
D(i,j) = \frac{1}{2} \sum_{l=1}^{d} \left[ \sigma_{il}^2 - \sigma_{jl}^2 \sigma_{il}^2 - \sigma_{jl}^2 \right] + \frac{1}{2} \sum_{l=1}^{d} \left[ \sigma_{il}^2 + \sigma_{jl}^2 \right] (\mu_i - \mu_j)^2
\]

(4)

where \( d \) is the dimension of the Gaussians. If we fix the partition, then the centroid \( \mathcal{N}(x; \mu_1, \Sigma_1) \) of cluster \( c \) which has a set of Gaussians \( \{ \mathcal{N}(x; \mu_i, \Sigma_i) \}, i = 1, 2, \cdots, n_c \) is calculated on the basis of minimizing the divergence as:

\[
\mu_c = \frac{1}{n_c} \sum_{i=1}^{n_c} \mu_i
\]

(5)

\[
\Sigma_c = \frac{1}{n_c} \sum_{i=1}^{n_c} (\mu_i - \mu_c)(\mu_i - \mu_c)^T
\]

(6)

In our model, diagonal covariance is used and they are:

\[
\mu_{cl} = \frac{1}{n_c} \sum_{i=1}^{n_c} \mu_i l
\]

(7)

\[
\sigma_{ll}^2 = \frac{1}{n_c} \sum_{i=1}^{n_c} (\mu_i l - \mu_{cl})^2
\]

(8)

with \( l = 1, \cdots, d \).

In the terminating step, the distortion is

\[
D = \sum_{i=1}^{n_c} D(i, c(i))
\]

(9)

where \( c(i) \) denotes the cluster membership of Gaussian \( \mathcal{N}(x; \mu_1, \Sigma_1) \) obtained in the partitioning step. If the relative change of \( D \) compared with last iteration is less than a specified threshold, then a convergence is observed and the iterations need to be terminated.

The binary splitting of a cluster centroid \( \mathcal{N}(x; \mu_c, \Sigma_c) \) is constructed by splitting the mean vector \( \mu_c \) into

\[
\mu_{c1} = \mu_c (1 + \lambda), \quad \mu_{c2} = \mu_c (1 - \lambda)
\]

(10)

where \( \lambda \) denotes a small perturbation factor.
5. Gaussian Clustering on the GPU

In Gaussian clustering, the partitioning step dominates the execution time. The naive implementation requires $O(nk)$ evaluations of the distortion metric per iteration to test each of the $n$ Gaussians for possible membership in each of the $k$ clusters. This is independent for each Gaussian, providing significant opportunities for parallel processing. In addition, the metric evaluation is not data-dependent, so the parallelism is in fact SIMD. This form of all-pairs testing can easily map to the streaming architecture of the GPU. As a consequence, it is feasible to avoid approximate speed-up heuristics and indeed use the brute force method incurring $O(nk)$ evaluations.

The remaining computationally expensive portions are fitting and terminating. These require $O(n)$ accumulations per iteration to update model parameters and to evaluate the total distortion, respectively. They are inherently “reduce” operations and are also suited for GPU. Some recent work has been proposed to use GPU for $k$-means clustering, e.g., [5] and [6]. In our implementation, all these steps are performed on the GPU.

5.1. Data representation

Considering all three tasks of partitioning, fitting, and terminating, we specify the thread blocks as 2-D arrays of equal size. The block dimension is chosen as 16 for memory-alignment reasons.

To cluster Gaussians on GPU, the Gaussians need to be stored in the device. As this data does not change throughout the process, it only needs to be transferred across the bus once. The Gaussians are organized in a 2-D array of float2 types with each row representing one Gaussian and each column representing one dimension. float2 is a built-in vector type of CUDA and is used here to store the mean and variance of a Gaussian. The row is padded with zeros to ensure aligned addresses for the runtime-critical coalescing of memory accesses.

5.2. Partitioning kernel

In partitioning, the GPU calculates the distortion of each Gaussian to each cluster in parallel. The algorithm is further parallelized by finding the partial minimum for the threads in each block. Then the CPU takes over a final reduction step to find the global minimum distortion for each Gaussian over the partial minima.

Similar to the Gaussians, the clusters are also organized in a 2-D array of float2s with each row representing one cluster model and each column representing one dimension. The row is also padded with zeros. Unlike the original unclustered Gaussians, the clustered models must be transferred to device in every iteration.

The task of computing the distortion matrix between the Gaussians and clusters is split among several threads in the following way:

- Each thread block is responsible for computing one square sub-matrix of the distortions;
- Each thread within the block is responsible for computing one distortion.

In the final tree split, 372k Gaussians are assigned to 40k clusters. The whole distortion matrix with floating point values needs about 60GB memory. Even if we do a first reduction within each block, i.e. store the partial minima instead of all distortions, the total memory is still close to 4GB. To further reduce the required memory, we partition the clusters into several groups and iterate over them. For each cluster group we load their models into the GPU memory and compute the partial minimum distortion and the corresponding membership from each thread block for each Gaussian.

In order to fit everything in the limited resources of GPU, the Gaussian dimension is divided into as many segments of block size as necessary, and the total distortion is computed as the sum of the segment distortions. Each of these partial distortions is computed by first loading the corresponding dimension segments of Gaussians and clusters from global GPU memory to the shared L1 cache, with one thread loading one dimension of each subset, and then by having each thread compute one dimension of the distortion. Each thread accumulates the result of each of these partial distortions into a register. Once done we perform a reduce operation within each block to find the partial minimum distortion and corresponding membership for each Gaussian and write the result to the global GPU memory. The process is finalized by the CPU: It reads back the partial minima and computes the membership of each Gaussian via a further reduction step.

5.3. Fitting kernel

To update the cluster models, we create a vector of Gaussian IDs for each cluster. The ID vectors are then padded with -1 to a multiple of the block size before being concatenated into a large chunk of consecutive memory. This can ensure that each thread block accumulates the Gaussians coming from the same cluster and each cluster occupies as few blocks as possible. Then both the Gaussians and the entire ID buffer are transferred to the global GPU DRAM. The first thread of each row of blocks is responsible for copying the Gaussian ID to the shared L1 cache. Each column is responsible for calculating the partial accumulations for one dimension of the Gaussians. Once done, the partial accumulations will be transferred back to the CPU, and the CPU adds them up for each cluster to calculate a new centroid. Below is the primary steps in the model update kernel.

- Copy Gaussian IDs from global GPU DRAM to shared L1 cache (in parallel by a subset of threads);
- Copy Gaussians from global GPU DRAM to shared L1 cache, where each thread copies one dimension, and then synchronize all threads;
- Calculate partial accumulations in shared L1 cache, and then synchronize all threads;
- Write result to global GPU DRAM.

5.4. Terminating kernel

Besides the Gaussians and the Gaussian ID vector, the terminating kernel also needs the newly updated cluster models and a cluster ID vector. Similar to the fitting kernel, the terminating kernel has the following main processing steps:

- Copy Gaussian IDs from global GPU DRAM to shared L1 cache;
- Copy Gaussians to shared L1 cache;
- Copy the model for this block L1 cache;
- Compute distortions in L1 cache (one dimension per thread);
- Calculate partial distortions in L1 cache;
- Write result back to global GPU DRAM.

6. Results

6.1. Experimental Setup

In our fMPE experiments, training is done on the 300h “Switchboard” speech corpus. No adaptation is used. System is speaker independent (SI) and gender independent (GI), with 9,300 states and 372k Gaussians. Models have cross-word triphone phonetic context. Features are 39-dimensional PLP. Lattice generation for fMPE uses a unigram LM. fMPE posterior vectors are
computed on 40k Gaussians and are further expanded to 280k dimensions based on context frames. The transform is trained with all of the training data. Testing is on both the “Switchboard” eval2000 conversational telephone speech (CTS) test set and a set 250 voicemails from the LDC Voicemail collection.

The CPU code for Gaussian clustering is compiled using the Microsoft Visual Studio 2005 compiler with optimization flag O2. Unlike our GPU implementation, approximate speed-up heuristics must be used to make the process feasible—without, the last iteration alone would take 9 hours. Therefore, the membership of each Gaussian is computed using only the closest clusters, in our case 20, to the original cluster as an approximation. The CPU code is further sped up using partial distance short-cut evaluation in metric evaluations.

6.2. Results for GPU Speed-up

We compare the GPU implementation results against the CPU results under Windows XP. Given the same input acoustic model, the execution time required by CPU, GPU (w/ overhead), and GPU (kernel only) for each step were recorded. This is done by sampling a clock-cycle counter at the beginning and at the end of each process and each kernel. “GPU (with overhead)” also includes the pre- and post-processing time, e.g., bus transfer in both directions, final reduction on CPU, and so on.

In Fig. 3, we compare the runtime of the partitioning step on GPU and CPU. We measure average execution time over all iterations at each tree split, i.e., each number of clusters. In the figures, “GPU (kernel only)” represents the execution time by the device, “GPU (w/ overhead)” is the GPU kernel time plus the pre- and post-processing time, while “CPU” has the same time counter position as “GPU (w/ overhead)”. Using the GPU, the partitioning is significantly accelerated, especially when the number of clusters becomes very large. The brute-force CUDA program on the G80 for partitioning of 215 clusters achieves more than 17 times speedup, compared with the Pentium 4 (which in addition uses speed-up heuristics).

Fig. 4 shows the execution time of model updates. In comparison with the total GPU time, the GPU kernel time is significant smaller. As the GPU is only responsible for calculating the partial accumulations, the kernel requires roughly the same execution time for different number of clusters. On the other hand, in postprocessing the CPU executes the merging and model update process, and more time is needed for more clusters, which makes the GPU less efficient as in partitioning. This can be improved by performing model updates on the GPU with additional transfer of the entire accumulations. Since the performance of model updates relatively uncritical compared to the partitioning, we did not do this improvement. In the terminating step, we observed a behavior similar to the model updates (not shown).

With all of the partitioning, fitting, and terminating portions implemented on the GPU, the overall execution time of Gaussian clustering was reduced from more than 24 hours to 100 minutes, or a 14 times speedup approximately.

6.3. fMPE Results

Fig. 5 shows the word-error rate (WER) improvements from fMPE. They are the same for the GPU and CPU implementation. On our two test sets—the eval2000 conversational telephone speech test set and the LDC voicemails—the baseline maximum-likelihood trained model obtains WERs of 34.3% and 38.6%, respectively. With fMPE, WERs were reduced to 32.3% and 36.2%—a relative 6% improvement.

7. Conclusions

In this work we investigated the use of a GPU to accelerate Gaussian clustering in fMPE acoustic model training. We compared start-of-the-art commercial products for both GPU and CPU: an NVidia GeForce 8800 Ultra GPU and an Intel Pentium 4 CPU. Our brute-force GPU implementation demonstrated 14 times faster overall than CPU implementation that did employ approximate speed-up technique. The GPU implementation required significant special consideration to adapt to the specific, somewhat peculiar GPU hardware architecture, including massive multi-threading and parallelized memory-access patterns. This was our first step towards leveraging GPUs for speech model training, and we got promising results in this task. We will now focus on accelerating the maximum likelihood model training.

8. References