GPU-accelerated HMM for Speech Recognition

Leiming Yu, Yash Ukidave and David Kaeli
Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA
{ylm, yukidave, kaeli}@ece.neu.edu

Abstract—Speech recognition is used in a wide range of applications and devices such as mobile phones, in-car entertainment systems and web-based services. Hidden Markov Models (HMMs) is one of the most popular algorithmic approaches applied in speech recognition. Training and testing a HMM is computationally intensive and time-consuming. Running multiple applications concurrently with speech recognition could overwhelm the compute resources, and introduce unwanted delays in the speech processing, eventually dropping words in the process due to buffer overruns. Graphics processing units (GPUs) have become widely accepted as accelerators which offer massive amounts of parallelism. The host processor (the CPU) can offload compute-intensive portions of an application to the GPU, leaving the CPU to focus on serial tasks and scheduling operations. In this paper, we provide a parallelized Hidden Markov Model to accelerate isolated words speech recognition. We experiment with different optimization schemes and make use of optimized GPU computing libraries to speedup the computation on GPUs. We also explore the performance benefits of using advanced GPU features for concurrent execution of multiple compute kernels. The algorithms are evaluated on multiple Nvidia GPUs using CUDA as a programming framework. Our GPU implementation achieves better performance than traditional serial and multi-threaded implementations. When considering the end-to-end performance of the application, which includes both data transfer and computation, we achieve a 9x speedup for training with the use of a GPU over a multi-threaded version optimized for a multi-core CPU.

Index Terms—Speech Recognition, Hidden Markov Model, GPUs.

I. INTRODUCTION

Speech recognition is used to translate spoken words into a machine-readable format. It is used in a wide range of applications including guidance and control of automobiles and aircraft, voice dialing systems, and digital medical dictation. Speech recognition is also used to provide accessibility features in mobile phones, banking and computer-unwanted delays in the speech processing, eventually dropping words in the process due to buffer overruns. Graphics processing units (GPUs) have become widely accepted as accelerators which offer massive amounts of parallelism. The host processor (the CPU) can offload compute-intensive portions of an application to the GPU, leaving the CPU to focus on serial tasks and scheduling operations. In this paper, we provide a parallelized Hidden Markov Model to accelerate isolated words speech recognition. We experiment with different optimization schemes and make use of optimized GPU computing libraries to speedup the computation on GPUs. We also explore the performance benefits of using advanced GPU features for concurrent execution of multiple compute kernels. The algorithms are evaluated on multiple Nvidia GPUs using CUDA as a programming framework. Our GPU implementation achieves better performance than traditional serial and multi-threaded implementations. When considering the end-to-end performance of the application, which includes both data transfer and computation, we achieve a 9x speedup for training with the use of a GPU over a multi-threaded version optimized for a multi-core CPU.

Speech recognition imposes strict performance deadlines, and so can be considered a real-time application. When speech recognition is run on CPU, we can encounter significant latency. A speaker has to wait for an average of a few seconds for a response from the system. A high degree of data-level and task-level parallelism is present in most HMM models used for speech recognition. GPUs are well-suited to exploit both data-level and task-level parallelism.

In this paper, we implement a GPU-accelerated HMM targeted for isolated-word based recognition. The implementation is evaluated using two high-end GPUs belonging to Nvidia’s Kepler architecture. Our implementation accounts for the execution time of the kernels, the data transfer time to and from the GPU, and also considers the time required for work scheduling and data setup. The evaluation is carried out while maintaining a 100% recognition rate.

The contributions of this paper include:

- parallelization of continuous HMMs targeting GPU hardware new acceleration features,
- evaluation of the Nvidia HyperQ technology recently introduced on the Nvidia Kepler architecture, and
- presentation of a highly-accurate and functional heterogeneous framework for isolated-word speech recognition.

The rest of paper of the paper is organized as follows. In Section II we provide a detailed introduction to HMMs. In Section III we discuss the potential benefits of a GPU. Section IV presents our experimental results. We discuss related work in Section V. In Sections VI we conclude the paper and in Section VII, we consider directions for future work.
II. HIDDEN MARKOV MODEL

A Hidden Markov Model is a two-stage stochastic process. For the first stage, it defines a finite-state space which describes causal transitional probabilities between states. The current process’s behavior depends only on the previous state, as shown in Equation 1, where \( P \) stands for the probability and \( S \) stands for the state.

\[
P(S_t | S_1, S_2, \ldots S_{t-1}) = P(S_t | S_{t-1})
\]  

(1)

For the second stage, the observed emission \( O_t \) is generated with a probability that only depends on the current state \( S_t \) and not on any predecessor state, as shown in Equation 2.

\[
P(O_t | O_1 \ldots O_{t-1}, S_1 \ldots S_t) = P(O_t | S_t)
\]  

(2)

The above sequence \( O_t \) is the only output that can be observed from the model. Meanwhile, the state sequence that is generated during the process is hidden. This is the reason why this model is called a Hidden Markov Model.

A first-order HMM usually consists of the following three important parameters [8], the initial probability \( \pi \), the state transition matrix \( A \) and the emission matrix \( B \). A simple three-state HMM example is illustrated in Figure 1.

- **Initial State Probability Vector**
  \[ \pi = \{\pi_i \mid \pi_i = P(S_1 = i)\} \]

- **State Transition Probability Distribution Matrix**
  \[ A = \{a_{ij} \mid a_{ij} = P(S_t = j \mid S_{t-1} = i)\} \]

- **Emission Probability Distribution Matrix**
  \[ B = \{b_{jk} \mid b_{jk} = P(O_t = o_k \mid S_t = j)\} \]

A. The Forward Algorithm

Our Forward algorithm computes the likelihood of an observation sequence according to the transition probability matrix. During the implementation, the sequence of observations \( O \) is represented by a \( T \times D \) observation matrix, where \( T \) is the number of speech segmentation windows and \( D \) is the length of extracted feature vectors. The D-variant Normal distribution density matrix (i.e., observation probability) \( B(T, N) \) for each state along \( T \) windows, becomes the input to calculate the likelihood of the observations.

To compute the likelihood, the accumulated state probability matrix \( \alpha \) is constructed with size \( T \times N \) for the sequence of \( T \) observations, with \( N \) underlying states. Each cell of \( \alpha(T, N) \) contains the probability of being in the corresponding hidden state \( N \) at a particular time \( T \), from the start of observation \( T_0 \). Eventually, the likelihood for a specific observation sequence can be calculated by accumulating the probability of \( \alpha(T, N) \) for the entire sequence, as shown in Figure 2.

1: **procedure** FORWARD()
2: \( \alpha(1, N_i) \leftarrow B(1, N_i) \times Prior(N_i) \)
3: loglikelihood \( \leftarrow \log 10(\sum \alpha(1, N_i)) \)
4: \( \alpha(1, N_i) \leftarrow \text{normalize} \alpha(1, N_i) \)
5: for \( t = 2 : T \) do
6: \( \alpha(t, N_i) \leftarrow B(t, N_i) \times (A \times \alpha(t - 1, N_i)) \)
7: loglikelihood \( \leftarrow \log \text{likelihood} + \log 10(\sum \alpha(t, N_i)) \)
8: \( \alpha(t, N_i) \leftarrow \text{normalize} \alpha(t, N_i) \)
9: end for
10: **return** loglikelihood
11: **end procedure**

![Fig. 2. The Forward Algorithm](image)

We assume that an infinite number of parallel units exist for the parallel implementations. Then the complexity \( O(N) \) of Line 2, computing the first row of \( \alpha \), can be reduced to constant time \( O(1) \). The summation of \( \alpha \) and element-wise log computation on Lines 3 and 7 can be reduced from \( O(N) \) to \( O(\log_2(N)) \). For parallelization on Lines 4 and 8, the complexity of parallel reduction for inner product of \( \alpha \) and element-wise division becomes \( O(\log_2(N)) \). From Line 5 to Line 9, the algorithm loops over the rest of observations \((T - 1)\), where Line 6 operates a matrix-vector multiplication \( A \times \alpha(t - 1, N_i) \), followed by an element-wise multiplication with \( B \). In practice, the complexity of Line 6 is \( \frac{N}{K} O(\log_2(N)) \), which is limited by the number of row-wise parallel units \( K \). Ideally, the parallel matrix-vector
Serial multiplication takes $O(\log_2(N))$ where $K = N$, and element-wise matrix multiplication takes $O(1)$.

Taking into account the loop dependency, the algorithmic complexity of a serial implementation is $O(N^2T)$, and $O(\log_2(N)T)$ for a parallel implementation. The complexities for both implementations are illustrated in Table I.

<table>
<thead>
<tr>
<th>Computation Step</th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Line 3</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 4</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 6</td>
<td>$O(N^2)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 7</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 8</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
</tbody>
</table>

TABLE I
Complexity Comparison For Forward Algorithm

**B. The Backward Algorithm**

Backward algorithm is used to compute the backward variable $\beta$. Instead of calculating the probability for all observations, it traces back through the hidden state probability path, guided by the state transition and observation probability matrices. The pseudocode is shown in Figure 3.

1: **procedure** BACKWARD()
2: $\beta(T, N_i) \leftarrow 1$
3: for $t = T - 1 : 1$ do
4: $\beta(t, N_i) \leftarrow A \times (\beta(t+1, N_i) \times B(t+1, N_i))$
5: end for
6: **return** $\beta(T, N_i)$
7: **end procedure**

Fig. 3. The Backward Algorithm

For the serial version, the element-wise multiplication $\beta \times B$ and matrix-vector multiplication on Line 4, and the normalization on Line 5 have the complexity of $O(N)$, $O(N^2)$ and $O(N)$ respectively. For the parallel version, the complexities for Lines 4 and 5 are reduced to $O(\log_2(N))$, as shown in Table II. Considering the loop-carried dependency, the parallel version achieves a complexity of $O(\log_2(N)T)$ compared with $O(N^2T)$ for its serial counterpart.

<table>
<thead>
<tr>
<th>Computation Step</th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 4</td>
<td>$O(N^2)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 5</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
</tbody>
</table>

TABLE II
Complexity Comparison For Backward Algorithm

**C. The Iterative Expectation-Maximization Algorithm**

The Iterative Expectation-Maximization (EM) algorithm finds the local optimal solution from the initial values by updating the following parameters: the prior probability $Prior$, the state transition matrix $A$, and the mean $\mu_s$ and covariance $\Sigma_s$ of the multi-variate Gaussian density. In our implementation, fifteen iterations are adequate to obtain a 100% recognition rate[12].

To describe this estimation procedure, we define $\xi$ as the state transition probability matrix, $\gamma$ as the probability in state $S_t$ at time $t$. Besides, $\chi$ stores the intermediate $N \times N$ matrix for $\xi$, $\gamma_s$ sums up the probability across time $T$ for each state $S_t$, and $\gamma_0$ presents the speech features’ probabilities for each state $S_t$. The stages involved in Forward-Backward-Expectation are described in Figure 4.

1: **procedure** EM()
2: **Initialize values**
3: for $iter = 1 : MaxIter$ do
4: $likelihood \leftarrow$ Forward()
5: $\beta(T, N_i) \leftarrow$ Backward()
6: $\gamma(T, N_i)$
7: $\xi \leftarrow$ Normalise($\gamma$)
8: $\gamma$ for each state
9: $E(\pi) \leftarrow$ Normalise($\gamma$)
10: $E(\mu_s)$
11: $E(\Sigma_s)$

Fig. 4. The Expectation-Maximization Algorithm

The complexity of the inner steps for both serial and parallel implementations are shown in Table III. Assuming that $D$ and $T$ are much smaller than $N$, the parallel implementation can lower the upper bound of the run time from $O(N^2T)$ to $O(NDT)$. 

<table>
<thead>
<tr>
<th>Computation Step</th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward()</td>
<td>$O(N^2)$</td>
<td>$O(\log_2(N)T)$</td>
</tr>
<tr>
<td>Backward()</td>
<td>$O(N^2)$</td>
<td>$O(\log_2(N)T)$</td>
</tr>
<tr>
<td>Line 4 - 12</td>
<td>$O(N^2)T$</td>
<td>$O(\log_2(N)T)$</td>
</tr>
<tr>
<td>Line 13 - 17</td>
<td>$O(N^2)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Line 18 - 22</td>
<td>$O(ND^2T)$</td>
<td>$O(NDT)$</td>
</tr>
</tbody>
</table>

TABLE III
Complexity Comparison For Iterative EM Algorithm
III. ACCELERATION USING GPUs

CPUs are efficient for processing complex serial tasks containing irregular control flow. They utilize techniques such as pipelining, deep memory hierarchies, superscalar execution and branch prediction to improve execution performance. Multi-core CPUs can execute multi-threaded applications effectively. But CPU resources are not being leveraged effectively while executing massively parallel workloads. Such computations can be accelerated by moving them to a GPU.

Nvidia provides the CUDA framework for programming compute-intensive applications on their GPUs. CUDA adopts the Single Instruction Multiple-Thread (SIMT) execution model for all the threads executing on the GPU. The program executing on the GPU is called a kernel. CUDA provides the user with limited shared memory management between threads and barrier synchronization for global communication. Modern CUDA architectures provide improved double-precision performance and allow for concurrent kernel execution of kernels. The latest CUDA-capable GPUs belonging to Kepler architecture provide support for Dynamic Parallelism and HyperQ [13] [14]. Specifications for the two GPUs used in our evaluation of our HMM implementation are shown in Table IV.

<table>
<thead>
<tr>
<th>Component</th>
<th>GTX 680</th>
<th>GTX TITAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Capability</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>GPU Clock rate</td>
<td>1059 MHz</td>
<td>876 MHz</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>256-bit</td>
<td>384-bit</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>512KB</td>
<td>1.5MB</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>1536</td>
<td>2688</td>
</tr>
<tr>
<td>Global Memory</td>
<td>2GB</td>
<td>6GB</td>
</tr>
<tr>
<td>Unified Addressing</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

TABLE IV
GTX 680 AND GTX TITAN SPECIFICATIONS

A. Nvidia Kepler Architecture

The GeForce GTX 680 was the first Kepler GPU released by Nvidia. This GPU has special stream-multiprocessor (SMX) instructions, SHFL and ATOM, provided to reduce the overhead of data exchange and atomic operations [14]. Equipped with these new hardware features, Nvidia’s Dynamic Parallelism and HyperQ mechanisms can boost application performance. Dynamic Parallelism allows the GPU to dynamically launch child kernels to further increase device utilization. Dynamic Parallelism can support a maximum of 24 levels of recursion. Data streaming capabilities can be greatly enhanced by using HyperQ. Up to 32 tasks can be run concurrently on the Kepler [13].

B. Programming Model

Data-Level Parallelism (DLP) and Task-Level Parallelism (TLP) can be implemented utilizing the SIMD capabilities of SMX and CUDA streams. When starting to run an application, the CPU offloads tasks and transfers data to the GPU. The programmer develops a CUDA kernel to execute a grid of threads. These threads are grouped into blocks, a block of threads are then mapped onto a SMX, where a group of 32 threads is called a warp. These warp threads are executed concurrently.

On a GPU device, global memory (read/write) and constant memory (read-only) store data from the host side. These memory regions are accessible to all threads. The local registers and shared memory are only accessible locally, and are meant to improve data access performance and reduce traffic to global memory, as shown in Figure 5. The stored data are updated when the current block becomes inactive and a new block of threads is dispatched. Kepler can allocate 64 active warps per SMX. If there is divergence inside a warp, the GPU will serialize execution. At the final stage of kernel execution, the results on device memory are transferred back to the host. If the GPU can overlap computation and communication effectively, the impact of the transfer due to PCIe latency can be hidden and performance should not be impacted.

C. Optimization

To create a kernel for the GPU, the programmer must parallelize the original algorithm to exploit task-level and data-level parallelism.

1) Task Level Parallelism: To handle independent tasks, CUDA streams are efficient for assigning workloads to multiple GPUs or executing concurrent kernels on a single GPU. The syntax is illustrated in Listing1.

Listing 1. Asynchronous Transfer with Two Streams
```
cudaMemcpyAsync(dev, host, size, H2D, stream0);
cudaMemcpyAsync(dev, host, size, H2D, stream1);
kernel<<<grid, block, stream0>>>(...);
kernell<<<<grid, block, sm, stream0>>>\{\...\};
cudaMemcpyAsync(host, dev, size, D2H, stream0);
cudaMemcpyAsync(host, dev, size, D2H, stream1);
```

Copying data using pinned memory on the host occurs asynchronously and can potentially overlap transfer time and computation. The data sets need to be independent to enable this overlap. Figure 6 shows 3-way concurrency, using two
GPU kernels and one CPU kernel. If we can achieve this degree of overlap, performance can potentially increase by 3x.

Fig. 6. Three Way Way Concurrency

2) Data Level Parallelism: Many well-established GPU libraries have been developed to distribute SIMD computation across multiple processors, among which MAGMA [15], Thrust [16], cuBLAS [17], etc. are included. Profiling tools such as Nvidia’s `nvprof` [18] and AMD’s CodeXL [19] are helpful for developers to detect hot spots in their programs. These codes are available for GPU programmers to increase productivity and performance.

Besides these fixed library templates, programmers often need to develop their own customized kernels. There are several essential and widely used techniques, such as Tiling, Binning and workload regularization, as described in [20].

For example, Line 20 in Figure 4 calculates a row sum on gamma observations, and then normalizes the value row-wise. This computation involves both a parallel reduction step and an element-wise division step, in order to update the expected mean. Figure 7 presents an optimized kernel execution diagram that utilizes constant memory and shared memory.

Fig. 7. Kernel Optimization for the Expected Mean

Here, parallel blocks of the kernel keep caching and adding the global data to shared memory. After performing reduction in the shared memory, the data are scaled using the values stored in constant memory. The gamma_state_sum is cached to provide fast access speed and reduced memory contention.

IV. Execution Results

We present the results of running our Speech Recognition algorithm on two different Nvidia platforms. The specifications of the host machine are shown in Table V. Performance is evaluated by comparing three different implementations, a single-thread CPU version, a optimized multi-thread CPU version with SSE instructions and the compiler optimization -O3 turned on, and a parallelized CUDA version.

Before applying our HMM to the input sequence, the input needs to be segmented. If the sampling frequency is 8KHz and the window size is 128-points with a 50% overlap, a one-second word pronunciation can produce 64 windows, which corresponds to the segmentation number $T$. Then the MFCC feature, a 39 coefficients vector for each segmentation, is extracted. The feature dimension $D$ is set to 64 with zeros padded at the end. The hidden state number $N$ is varied from 64 to 4096 for testing. During the evaluation, the elapsed time reported takes into account both transfer and computation times. We report the minimum runtime after 100 test runs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Ubuntu 13.04</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i7-920 @ 2.67GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>8GB DDR3 1066MHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia GeForce GTX 680</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Table V

A. Data Transfer

As the hidden state number $N$ increases, the time spent copying data between host and device grows, as shown in Figure 8 and Figure 9. Instead of using pageable memory, pinned memory can prevent data from being swapped out and can preserve memory bandwidth. Here, the Forward Algorithm is used as an example.

For each API call, the elapsed time shown on top of each bar is reported in millisecond. As seen in the figures, using pinned memory can speedup the total runtime when $N$ is equal to or greater than 1024. It indicates that the data transfer should be at least 4.2 MB to utilize pinned memory that includes all HMM parameter matrices such as transition and emission matrices.

B. HMM Algorithms

In our HMM algorithm, we use the cuBLAS library [17] for performing vector-vector, matrix-vector and matrix-matrix operations. The Nvidia CUDA command-line profiler, `nvprof`, was used to identify the bottlenecks in this application. For example, the `cublasSdot()` function that computes the dot product of two vectors, by default returns a value from the device to the host. Through profiling, we detected frequent communications between host and device, sharing this intermediate value. We configured the cuBLAS handler to use `CUBLAS_POINTER_MODE_DEVICE` mode so that the dot product is kept on the device without transferring intermediate data to the host.

Several additional optimizations have been applied here. For example, we merged kernels that are independent from
1) The performance of the Forward Algorithm is shown in Figure 10. The single-thread CPU version is faster than the others when $N$ is smaller than 256. But as $N$ grows, the overhead of pthread creation and joins can be hidden by computation on the 4 CPU cores when using 4 threads. When run on GTX 680 with $N$ equal to 4094, the GPU implementation can outperform the optimized CPU version by 5.6x.

2) The Backward Algorithm is not as computationally intensive as the other two. As shown in Figure 11, the single-thread CPU version runs fastest among the three versions until the hidden state number grows larger than 256. The best performance that CUDA can achieve is 5.4x compared to the optimized CPU version, 20.5x compared to the single-thread CPU version.

3) The Expectation Step is most time-consuming step for the Iterative EM algorithm because of the repeated process to compute and update the expected values of the prior probability $\text{Prior}$, the transition matrix $A$, the mean $\mu_s$ and covariance $\sum_s$. Compared to the single-thread CPU version, the GPU code achieves a 3x speedup for the smallest dataset, a 31x speedup for the largest. Meanwhile, the GPU version accelerates the optimized CPU version by 3x for the smallest case and by 9x for the largest case. Results are shown in Figure 12.

4) Combining the Forward, Backward and Expectation steps into an iterative training loop, this EM algorithm can help us find the local optimum parameters for our HMM. A well-trained model should maintain a high recognition rate. We found that training with more than 15 iterations is unnecessary given the speech samples used in [12]. As shown in Figure 13, the CUDA version continues to enjoy double-digit speedups. The GPU speedups versus the single-thread and multi-thread CPU versions are 43.4x and 9.2x respectively, when working with the largest test case.

C. HyperQ

HyperQ allows multiple CPU cores or processes to dispatch parallel work to the GPU. Between the host and a single GPU, a maximum of 32 simultaneous connections can be managed by the hardware. This Nvidia GPU feature is presently only supported on high-end GPUs with Capability 3.5 or higher, such as the GTX Titan.

Our evaluation of Multi-streaming using HyperQ was done using the GTX Titan. The performance impact of this new feature on multi-word log-likelihood calculations using the Forward Algorithm is shown in Table VI, where time is reported in milliseconds. The results show that the execution time can be reduced by 19% on average with 2 concurrent streams and by 32% on average using 3 concurrent streams. For the best case, the Forward algorithm can achieve a 44% and 74% improvement in execution time by using 2 and 3 concurrently executing streams, respectively.

Running concurrent word recognition streams requires transferring data simultaneously, so the number of DMA engines becomes the bottleneck for our application. For the worst case, when $N$ equals 4096, the kernels spend most of their time waiting for the data, so we can only achieve a 7% speedup with HyperQ.

The HyperQ streaming feature can also be applied to the iterative EM algorithm during the Expectation stage, as shown in Line 18 through Line 22 in Figure 4. Computing and
Fig. 10. Forward Performance

Fig. 11. Backward Performance

Fig. 12. Expectation Performance

Fig. 13. Iterative EM Performance

updating the expected mean and covariance for each hidden state can be executed concurrently. We were excited to find that we could achieve a 2x speedup when running two concurrent streams. However, launching more than two streams did not achieve good speedup, as seen in Figure 14.

During the Expectation stage, since the required data are available before launching the computation for the expected mean and covariance, the kernels can run in parallel using two streams and achieve a 2x speedup. But running a third concurrent stream requires more partitioned resources on the device, particularly shared memory in our case. Based on the available resources, a limited number of kernels can run simultaneously. Therefore, four streams can only achieve an average of 15% speedup over two streams. Kernel execution can not achieve perfect concurrency due to resource contention. As illustrated in Figure 14, using over 4 streams can even decrease performance in some cases.

V. RELATED WORK

There has been a significant number of successful speech recognition systems delivered. The Harpy system [21] sponsored by Defense Advanced Research Projects Agency, can recognize 90% of naturally spoken language from a dictionary set containing 1000 words. The CMU SPHINX-4 speech recognition system, developed in Java, can achieve a recognition speed of 1.29 seconds/word, with a 3.9% word error rate [22]. The DynaSpeak [23] system uses a HMM acoustic model and achieves a 92% average word recognition rate across different acoustic scenarios.

GPU implementations started to appear recently. In 2009, Liu described a CUDA implementation to accelerate HMM Training and Classification, and achieved a 100x speedup [10]. During the same year, Zhang et al. presented a GPU-accelerated Viterbi algorithm, and compared their work to a parallelized HVite system [24]. They obtained a 3x speedup on keyword spotting from a 3000 word dataset. In 2011, Hymel improved Liu’s work to target HMMs for wireless applications [11]. He found that in general, GPU implementations outperform CPU implementations when the number of hidden states grows larger than 400. Chong et al. [25] also implemented an efficient automatic speech recognition engine on the GPU that achieves 10.6x speedup over an optimized sequential program. More recently, a hybrid heterogeneous speech recognition engine named HYDRA [26] has reported a 30x speedup over a CPU when running over a billion parameters using a large language model.

While all of these previous works have made significant progress in accelerating Speech Recognition, we feel our
TABLE VI

<table>
<thead>
<tr>
<th>N</th>
<th>CUDA</th>
<th>HyperQ(x2)</th>
<th>HyperQ(x3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.80</td>
<td>1.62</td>
<td>1.60</td>
</tr>
<tr>
<td>128</td>
<td>1.86</td>
<td>1.67</td>
<td>1.63</td>
</tr>
<tr>
<td>256</td>
<td>2.12</td>
<td>1.89</td>
<td>1.67</td>
</tr>
<tr>
<td>512</td>
<td>2.66</td>
<td>2.34</td>
<td>2.04</td>
</tr>
<tr>
<td>1024</td>
<td>6.27</td>
<td>4.35</td>
<td>3.61</td>
</tr>
<tr>
<td>2048</td>
<td>12.48</td>
<td>9.19</td>
<td>7.84</td>
</tr>
<tr>
<td>4096</td>
<td>34.46</td>
<td>32.30</td>
<td>32.02</td>
</tr>
</tbody>
</table>

Fig. 14. Expectation Performance Using HyperQ

The study on current Nvidia platforms consider the end-to-end performance of the algorithm and provide a number of new insights.

VI. CONCLUSION

As GPU vendors continue to provide more programmable processors featuring high bandwidth and massive data processing performance, we will continue to see new classes of applications accelerated by utilizing these powerful platforms. In this paper we described our GPU-accelerated HMM and showed that this implementation can improve performance by 9.2x and 7.9x, as compared to an optimized multi-thread CPU version during training and testing stages, respectively. This work shows the potential of using a GPU as a speech recognition platform for real-time implementations. New hardware-supported programming features such as HyperQ can further increase the performance, but the performance benefits can be limited by the memory bandwidth and available resources in the hardware. We also found that the GPU version can be slower than a single-thread CPU version when the number of hidden states is smaller than 256 for the Forward Algorithm. Thus, the combination of CPU-based testing and GPU-based training fits the performance requirements for isolated word speech recognition when dealing with a limited dataset.

VII. FUTURE WORK

Currently, feature extraction remains a single-thread task that is run serially on the CPU. We would like to offload the entire preprocessing step to the GPU, leaving the CPU to initialize and finalize the computation. We also would like to implement a GPU-accelerated noise cancellation algorithm to make the recognition system robust to background noise.

REFERENCES