Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors

Jike Chong
Youngmin Yi
Arlo Faria
Nadathur Rajagopalan Satish
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-69
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-69.html

May 22, 2008

Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors

Jike Chong Youngmin Yi Arlo Faria Nadathur Satish Kurt Keutzer

University of California, Berkeley

Abstract

Automatic speech recognition is a key technology for enabling rich human-computer interaction in emerging applications. Hidden Markov Model (HMM) based recognition approaches are widely used for modeling the human speech process by constructing probabilistic estimates of the underlying word sequence from an acoustic signal. High-accuracy speech recognition, however, requires complex models, large vocabulary sizes, and exploration of a very large search space, making the computation too intense for current personal and mobile platforms. In this paper, we explore opportunities for parallelizing the HMM based Viterbi search algorithm typically used for large-vocabulary continuous speech recognition (LVCSR), and present an efficient implementation on current many-core platforms. For the case study, we use a recognition model of 50,000 English words, with more than 500,000 word bigram transitions, and one million hidden states. We examine important implementation tradeoffs for shared-memory single-chip many-core processors by implementing LVCSR on the NVIDIA G80 Graphics Processing Unit (GPU) in Compute Unified Device Architecture (CUDA), leading to significant speedups. This work is an important step forward for LVCSR-based applications to leverage many-core processors in achieving real-time performance on personal and mobile computing platforms.

1. Introduction

Automatic speech recognition is a key technology for enabling rich human-computer interaction in many emerging applications. The computation required for effective human-computer speech interaction has been challenging for today's personal and mobile platforms such as laptops and PDAs. The emergence of many-core accelerators for PCs opens up exciting opportunities for the average consumer platform. Programming for many-core platforms, however, requires understanding a very different set of trade-offs. Programming for GPUs, for instance, requires a vast amount of parallelism in applications to hide memory latency. GPUs also minimize threading overhead while restricting thread-to-thread interactions. This paper analyzes an algorithm typically used for large-vocabulary continuous speech recognition (LVCSR), and illustrates important algorithmic trade-offs for an efficient implementation on the NVIDIA Geforce 8800 GTX, an off-the-shelf GPU.

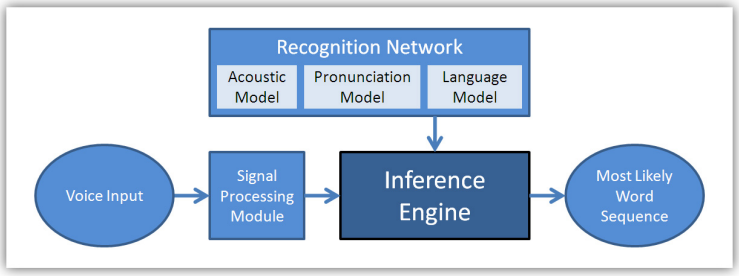

Figure 1. Architecture of a large vocabulary continuous speech recognition system

One effective approach for LVCSR is to use a Hidden Markov Model (HMM) with beam search [9] approximate inference algorithm. It is the standard approach seen in major speech recognition projects such as SPHINX, HTK, and Julius [14, 20, 13]. Figure 1 shows the major components of such a system. This system uses a *recognition network* that is compiled offline from a variety of knowledge sources using powerful statistical learning techniques. Spectral-based speech features are extracted by signal-processing the audio input and the inference engine then computes the most likely word sequence based on the extracted speech features and the recognition network.

Inference engine based LVCSR systems are modular and flexible. They are language independent and robust to various acoustic environments [20, 13]: by using different recognition networks and signal-processing kernels, they have been shown to be effective for Arabic, English, Japanese, and Mandarin, in a variety of situations such as phone conversations, lectures, and news broadcasts.

The inference process recognizes sequences of words from a large vocabulary where the words can be ordered in exponentially many permutations without knowing the boundary segmentation between words. The inference engine implements a search algorithm where the search space consists of all possible word sequences given an array of features extracted from an input audio stream. A 50,000 word vocabulary expands to almost one million hidden states in the HMM in which case a naïve Viterbi search is computationally infeasible, as explained in section 2. The beam search heuristic reduces the search space to a manageable size by pruning away the less likely word sequences [9].

Although there is ample fine grained concurrency inside the inference engine, its top level architecture is quite sequential. As Figure 2 illustrates, the outer loop of the inference engine proceeds in a sequence of iterations with feedback between successive iterations. Even within an iteration, the algorithm follows a rigid sequence of search space evaluation and pruning. The parallelism in the algorithm appears within each pipeline step, where thousands to tens-of-thousands of word sequences can be evaluated and pruned in parallel.

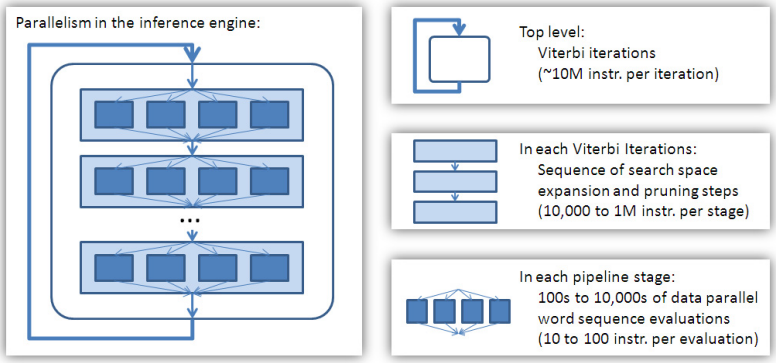

Figure 2. Structure of the inference engine and the underlying parallelism opportunities

There is a large body of prior work in mapping LVCSR to parallel platforms. Ravishankar [18] in 1993 presented a parallel implementation of the beam search algorithm on a multi-processor cluster. Agaram *et al.* [2] in 2001 showed an implementation of LVCSR on a multi-processor simulator. And Dixon *et al.* [6] in 2007 produced a multi-threaded LVCSR system, and attempted to use the GPU as a co-processor accelerator for computing observation probabilities, but were not able to exploit the full potential of the GPU platform. There has also been continuous research in the acceleration of speech recognition in hardware [8, 11]. The proposed hardware speech accelerators are often limited in the size of vocabulary supported and in the flexibility of

the algorithm. More recent implementations such as [11] are similar in architecture to custom many-core accelerators with associated application-specific scheduling and locking mechanisms. A software solution on a general-purpose off-the-shelf chip is more cost-efficient and more readily available today.

Compared to multi-core platforms, many-core platforms take on a different architectural design point. By using many simpler cores, more of the per-chip transistor budget is devoted to computation, and less to on-chip memory hierarchies. Applications taking advantage of this architectural trade-off should be data-parallel and have their memory accesses coalesced. Data-parallel algorithms have mostly independent parallel computation units with minimal slow global synchronizations, and coalesced memory accesses amortize load-store overheads for higher computation throughput.

This paper outlines a data-parallel implementation of the LVCSR algorithm on a NVIDIA many-core GPU and illustrates the key algorithmic and data layout decision for coalesced memory access. We briefly introduce the continuous speech recognition process in section 2, provide an overview of many-core architecture properties in section 3, and explain in detail our implementation in section 4.

2. Large vocabulary continuous speech recognition

A speech recognition problem can be defined in terms of the set of possible word hypotheses that can be inferred from an acoustic observation signal. The simplest inference problem is an isolated word recognition task, such as discriminating between a “yes” or “no” in an interactive voice response system; such a task can be solved by many techniques, generally with modest computational effort. By contrast, large vocabulary continuous speech recognition (LVCSR) is a much more difficult problem: for example, the objective might be to provide a transcription to serve as closed captions for a television recording. LVCSR systems must be able to recognize words from a very large vocabulary arranged in exponentially many permutations, without knowing the boundary segmentation between words.

For this task, the Hidden Markov Model (HMM) framework has emerged as the most successful approach. This framework follows the overall architecture of Figure 1, with an HMM used as the recognition network.

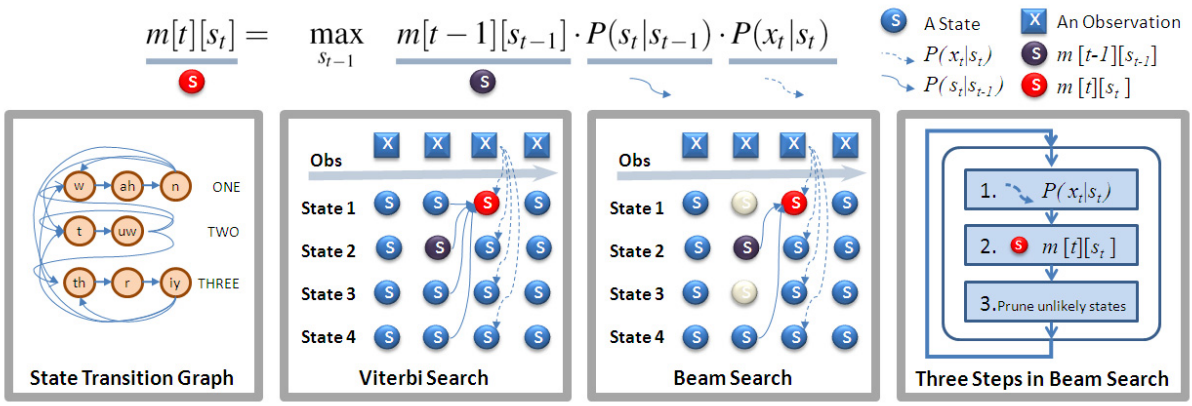

Figure 3. A HMM state transition graph of LVCSR and a comparison of Viterbi and beam search inference algorithms.

2.1. HMM

An HMM represents a discrete-time statistical process in which elements of a sequence of observed random variables are generated by unobserved state variables in a Markov chain. The use of HMMs for sequential pattern recognition has led to various applications besides speech recognition, such as optical character recognition and machine translation. Figure 3 shows an example of an HMM, with the state *transition graph* to the left, and the Viterbi Search *trellis* at center left. The nodes in the transition graph represent the hidden states s of the system with the set of possible transitions (with associated transition probabilities $P(s_t|s_{t-1})$ between them. The emission model $P(x_t|s_t)$ represents the probability of seeing a particular observation x_t at each state s_t of the HMM. The transition probabilities and emission model are learned from training data. A particular path through the trellis has a specific probability of occurrence given the observation sequence. The inference problem is then to find the path with the maximum probability.

The naïve approach is to explore and compute the probability of all paths and select the most probable one. For a network with N states, an observation of length T generates $O(N^T)$ possible paths. For one million states over hundreds of observations, this is clearly computationally infeasible. Fortunately, the Viterbi algorithm was developed to reduce this exponential search space to $O(N^2T)$ by applying dynamic programming. In this algorithm, the most probable path at each time step terminating at a given state can be computed as a recurrence relation maximizing a term involving the most probable paths to each state at the preceding time step.

Let $x_0, \ldots, x_t$ represent a partial observation of the first $t < T$ observation frames and $s_0, \ldots, s_t$ be a corresponding sequence of HMM hidden states that could have generated those observations. We can define the quantity $m[t][s_t]$ as the joint probability of the partial observation sequence of length t and the most likely path terminating at state s_t that generated it: $m[t][s_t] \doteq \max_{s_0,\ldots,s_{t-1}} P(x_0, \ldots, x_t, s_0, \ldots, s_{t-1}, s_t)$

The recurrence relation exploited by the Viterbi algorithm allows us to compute each $m[t][s_t]$ by maximizing over $O(N)$ values at the previous time step as in the equation at the top of Figure 3.

Thus the Viterbi algorithm requires the previously computed joint likelihood at a previous state $m[t-1][s_{t-1}]$ along with the application of the state transition probability $P(s_t|s_{t-1})$ and the observation probability at the current state $P(x_t|s_t)$. We shall see that for LVCSR inference, the computation bottleneck is the evaluation of the observation probability.

Because the size of the state space $|N|$ can be on the order of millions for LVCSR, a common approximation to reduce computation is to employ the *beam search* approach. At each time step, 95-98% of the paths can be safely pruned away as the lower scoring paths are unlikely to become the best path at the end of the observation sequence. Figure 3 shows the nodes traversed in a Viterbi algorithm and a Beam search algorithm. The extent of pruning is usually decided by means of a threshold parameter. The pruning discards any paths with likelihood smaller than the threshold relative to the most-likely path; the threshold is often adjusted dynamically to cap the maximum number of active states that are under consideration at any time step.

2.2. LVCSR recognition network

In this section, we show how the HMM is parameterized for the LVCSR application. The recognition network in LVCSR is a transition graph with a hierarchical combination of HMM acoustic model H, word pronunciation L, and language model G, as depicted in Figure 4. The three models form a hierarchy — the acoustic model defines the phone units of speech, the pronunciation model specifies how these phone units are combined into words, and the language model represents how word-to-word transitions occur. We provide a brief description of these components, and their integration in LVCSR systems.

The HMM acoustic phone model is a model for an elementary unit of speech composed of a three-state, left-

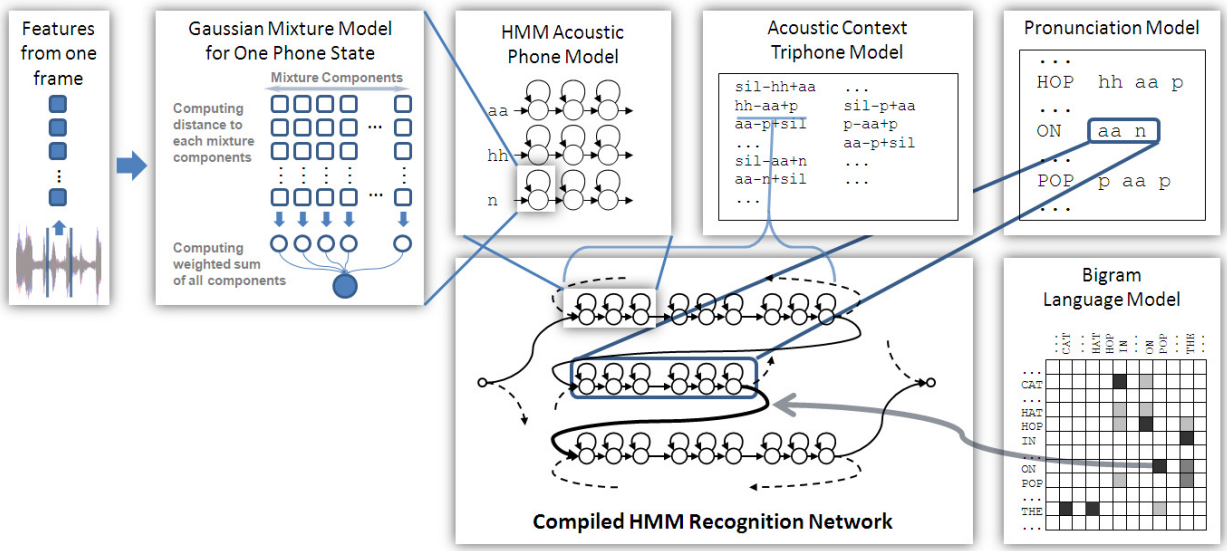

Figure 4. A recognition network composed of acoustic, pronunciation, and language models.

to-right, self-looping transition structure. Each state contains a multivariate Gaussian Mixture Model (GMM) from which the observation probability density function can be computed for the state. Each component in the GMM is a Gaussian model with a vector of means and variances for different features of the input speech. Given an observation of features extracted from an audio frame, we compare the features with the vectors of each Gaussian to obtain a measure of probabilistic match. We then compute the observation probability as a weighted sum of the probabilistic matches to each of the Gaussian mixture components. Since there may be 16 to 128 components present in a typical GMM for an LVCSR system, computing the observation probability density function is often the most compute intensive step in the inference algorithm.

The pronunciation model defines the structure of the state transitions inside each word as a concatenation of the acoustic models for each phoneme of the word. The GMM for different phone units and the transition probabilities within each phone unit are shared among all of its instantiations across many word pronunciations. Most LVCSR systems have a pronunciation dictionary comprising tens of thousands of words, some of which may have multiple possible pronunciations. A typical word has 6-7 phonemes, each of which consists of 3 HMM states. For a 50,000 word language model, this leads to a total of about $7 * 3 * 50,000 =$ a million states, making the inference problem computationally challenging.

Lastly, the recognition network used for decoding a speech utterance must incorporate a language model which constrains the possible sequences of words. The most common language model for a task of this scope is based on bigram probability estimates, relating the likelihood of co-occurrence for two consecutive words in a sequence. Given a set of word HMM models concatenated from phone units, the final recognition network can be constructed by creating bigram-weighted transitions from the end of each word to the beginning of every other word. Due to the sparsity of language model training data, not all bigram transition probabilities can be reliably estimated, so in practice the word-to-word transition matrix is sparse – most words transition only to a small number of following words, and transitions to all other words are modeled with a backoff probability [9].

Performing inference on this model produces the most likely sequence of words while simultaneously determining where the word boundaries lie. We use a beam search algorithm for LVCSR inference, and we exploit parallel processing to evaluate a large number of paths concurrently to reduce overall runtime.

3. Many-core platform characteristics

Current and emerging many-core platforms, like recent Graphical Processing Units (GPUs) from NVIDIA and the Intel Larrabee processor [12], are built around an array of processors running many threads of execution in parallel. These chips employ a Single Instruction Multiple Data (SIMD) architecture. Threads are grouped using a SIMD structure and each group shares a multithreaded instruction unit.

In this work, we map the LVCSR application to the G8x series of many-core GPU architectures from NVIDIA that is already available in the market. We intend to optimize LVCSR for the Intel Larrabee processor when it becomes available.

The G8x series of GPU architectures consists of an array of Shared Multiprocessors (SM) multiprocessors, each of which consists of 8 scalar processors (SP). The G80 NVIDIA GPU, for instance has 16 such SM multiprocessors and is capable of executing up to 128 concurrent hardware threads. Figure 5 shows the block diagram for the NVIDIA G80 chip. GPUs rely mainly on multi-threading to hide memory latency. Each SM multiprocessor is multithreaded and is capable of having 768 different thread contexts active simultaneously.

The GPU is programmed using the CUDA programming framework [16, 17]. An application is organized

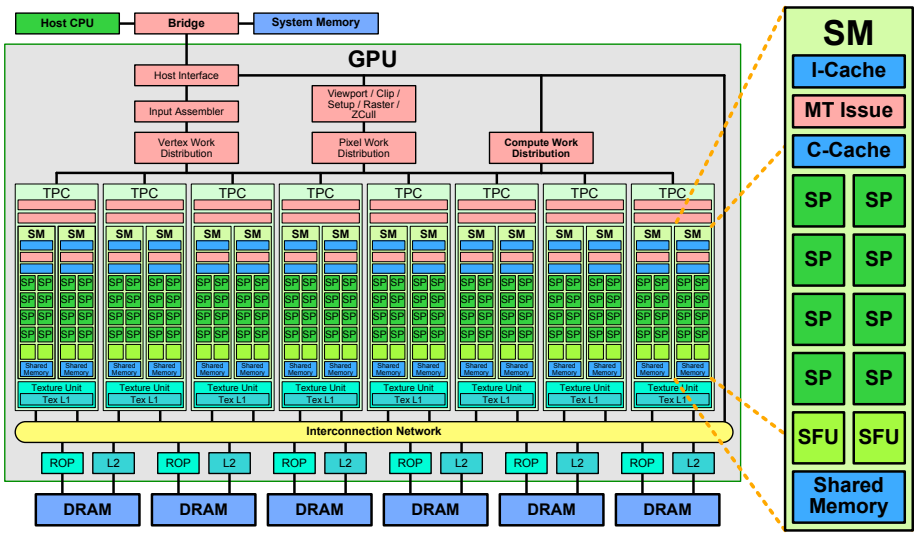

Figure 5. Block diagram of a G80 GPU with 128 scalar SP processors, organized in 16 SM multiprocessors, interconnected with 6 DRAM memory partitions.

into a sequential host program that is run on a CPU, and one or more parallel kernels that are typically run on a GPU (although CUDA kernels can also be compiled to multi-core CPUs [19]).

A kernel executes a scalar sequential program across a set of parallel threads. The programmer organizes these threads into *thread blocks*; a kernel consists of a grid of one or more blocks. A thread block is a group of threads that can coordinate by means of a barrier synchronization. They may also share a shared memory space private to that block. The programmer must specify the number of blocks in the kernel and the number of threads within a block when launching the kernel. A single thread block is executed on one SM multithreaded processor. While synchronization within a block is possible using barriers in the shared memory, global synchronization across all blocks is only performed at kernel boundaries. The SIMD structure of the hardware is exposed through thread warps. Each group of 32 threads within a thread block executes in a SIMD fashion on the scalar processors within an SM, where the scalar processors share an instruction unit. Although the SM can handle the divergence of threads within a warp, it is important to keep such divergence to a minimum for best performance.

Each SM is equipped with a 16KB on-chip scratchpad memory that provides a private per-block shared memory space to CUDA kernels. This shared memory has very low access latency and high bandwidth. Along with the SM's lightweight barriers, this memory is an essential ingredient for efficient cooperation and commu-

nication amongst threads in a block.

Threads in a warp can perform memory loads and stores from and to any address in memory. However, when threads within a warp access consecutive memory locations, then the hardware can coalesce these accesses into an aggregate transaction for higher memory throughput. On current architectures a warp of 32 threads accessing consecutive words only issues two memory accesses while a warp performing a gather or scatter issues 16 memory accesses.

Code optimizations for the GPU platform using CUDA usually involves code reorganization into kernels, each of which has thread blocks with warps of threads in them. The synchronization available at each step is different. There is an implicit global barrier between different kernels. Threads within a single thread block in a kernel can synchronize using a local barrier, but this is not available to threads across different blocks. It is a good general policy to limit or eliminate routines that require global coordination between all threads. Code optimizations for the GPU may also involve changing the layout of the data structures for getting good coalescing of memory accesses across threads in a warp.

4. Inference engine implementation

The inference engine implements the beam search algorithm and iterates over three main steps of execution as shown in figure 3 on the right. Each iteration begins with a set of active states that represent the set of most likely sequence of words up to the current observation. The first step computes the observation probabilities of all potential next states, the second step computes the next state likelihoods, and the third step selects the most likely next state to retain as the set of new active states for the next iteration. This section explains the implementation of a beam search iteration.

One important challenge of effectively using the GPU as an accelerator with a separate memory space is to avoid copying intermediate results back and forth between the CPU and GPU. To avoid this transfer we keep all computations and intermediate results on the GPU. However, we must manage intermediate result data layout to maximize coalesced memory accesses, and avoid expensive global coordination in the implementation. We explain how we implement each of the three steps of execution in the following subsections and present the final implementation at the end of this section.

4.1 Observation probability computation

The first step of the beam search inference engine is to compute the observation probability of *all* potential next states. For LVCSR, as shown in figure 6, many states in the recognition network share the same label. All states with the same label have the same observation probability, thus the naïve approach of doing one observation probability computation per potential next state would greatly duplicate necessary computation. Experiments show that 97-98% of the potential next states have duplicate labels. In the sequential implementation, observation probability computation is the major bottleneck, taking 98% of the total runtime. By doing the computation on only the *unique* labels, we can achieve a 40X reduction in runtime for this step. Table 1 illustrates the advantages of finding the unique labels.

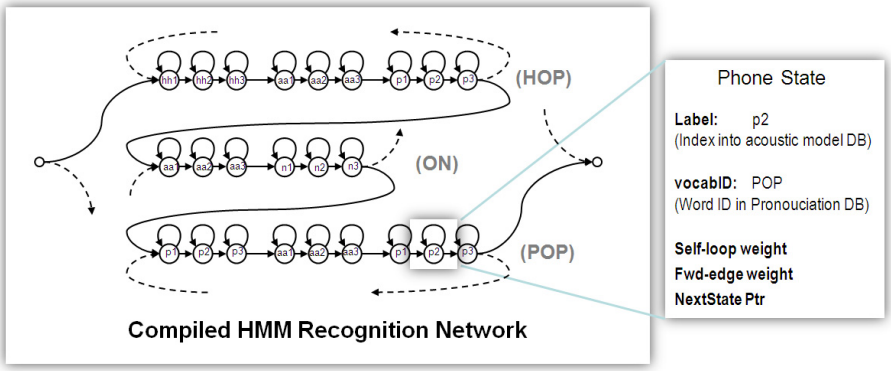

Figure 6. A phone state in the recognition network with its label

Table 1. Observation probability computation for a set of 23 speech segments

Algorithm	# Labels Computed	Execution Time	% Runtime
All next state labels	398 million	3092 sec	98.13%
Only next state unique labels	9 million	72 sec	55.00%

For this reason, we add a step to the implementation, where step 0 uses an efficient data-parallel find-unique algorithm to remove duplicates among the labels of potential next states. Step 1 then only needs to compute the observation probabilities for the unique labels.

4.1.1 Step 0: Finding unique labels

Given a set of labels, we want to find the set of unique labels for which to compute the observation probabilities. Finding the set of unique labels is straightforward on a CPU, usually implemented with a "sort" followed by "compaction". However, doing the same routines in a data-parallel fashion is much more involved [3, 5].

By considering the application-specific characteristics and platform capabilities, we are able to implement a much more efficient find-unique routine for LVCSR on the NVIDIA GPU. In our implementation of LVCSR, the set of all possible labels is known *a priori*, as our recognition network is statically compiled. This enables us to make a lookup table of flags of the size of all possible labels, and set the flag for the labels that have appeared in the set of potential next states. To produce a list of unique labels, we compact this lookup table to only include elements whose flags are set. This reduces an expensive sorting problem to a parallel hashing problem, see table 2. The lookup table memory footprint is not significant. For our 50k-word vocabulary, the lookup table only needs to support 78441 flags.

The flag setting operation is parallelized by having each thread look at one potential next states, and set the flag for the label of that state. This causes memory write conflicts, as different threads may try to modify a particular flag at the same time. Although NVIDIA GPUs provide atomic memory accesses to manage write conflicts, the atomic access implementation exposes the long memory latency to device memory and severely impacts performance.

Table 2. Step 0: Find-unique implementation execution time for a segment of speech

Algorithm	Runtime
Sorting	2420.0 ms
Lookup table with atomic memory ops	157.1 ms
Lookup table with non-atomic memory ops	21.4 ms

Algorithmically, the flag setting step only requires the invariant that flags for labels appearing in potential next states are set. We leverage the non-atomic memory write semantics of the NVIDIA GPUs, which states that at least one conflicting write to a device memory location is guaranteed to succeed [17]. The success of any one thread in write conflict situations can set the flag and satisfy the invariant; thus we can use non-atomic operations to set the flags. As shown in table 2, this dramatically improves the performance for our find-unique implementation.

4.1.2 Step 1: Parallelizing observation probability computation

As shown in table 1, observation probability computations continue to dominate the beam search sequential runtime even when we compute only the set of unique labels. Fortunately, the computation for each unique label is independent, and is thus trivially parallelizable. However, the number of unique labels varies drastically across beam search iterations, from 500 to over 3000, as illustrated in figure 7. To efficiently utilize the G80 architecture, one needs a minimum of around 5000 thread context to be active concurrently. We thus explore another level of parallelism in the GMM computation, where each mixture model contains 16 to 128 mixtures that can be computed in parallel, as shown in figure 4.

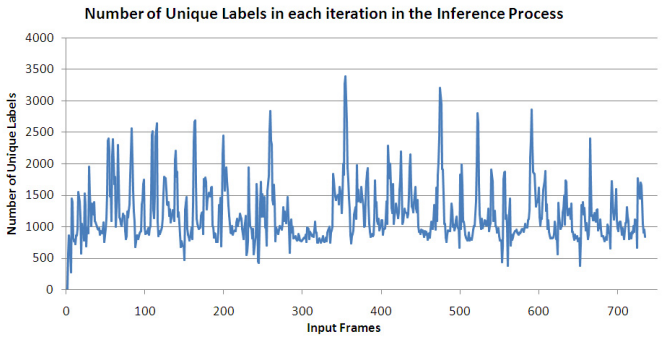

Figure 7. Graph of number of active states over observation frames

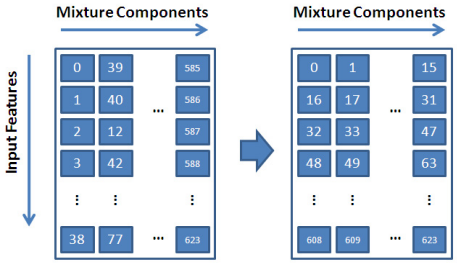

Figure 8. Gaussian mixture model data layout re-formatting for coalesced access

The NVIDIA hardware granularity fits the GMM computation well. Our implementation maps each GMM computation to a thread block on the GPU, where each thread in the block computes the distance between one Gaussian mixture and the observation feature space. It is crucial that threads in a thread block be able to synchronize, as the observation probability is computed by summing the weighted distances from each Gaussian in the mixture. The summation is done using a tree-based reduction scheme as in [1] using within-block synchronization.

Mapping the Gaussian mixtures to different threads in a block also has implications on the data structures for storing the Gaussian mixture means and variances. For coalesced memory access, we reorganized each GMM's mean and variance arrays (one element per Gaussian in the GMM) to be indexed by features, rather than by mixtures. Figure 8 shows the memory reorganization for each GMM. This reorganization improves performance of this kernel by a factor of 3.

4.2 Step 2: Next state likelihood computation

There are two types of next state likelihood computations: those using within-word transitions, and those using word-to-word transitions. The within-word transitions connect an active state to a small number of next states, whereas word-to-word transitions connect an active state to the first state of each word in the vocabulary.

Let us refer to an "active state at the end of a word" as an "active-end-state", and refer to the "first state in each word" as "first state". An active-end-state is present in 80-99% of the beam search iterations depending on beam search configurations. In these iterations all first states must be examined, leading to a strong incentive to arrange the underlying data structure to facilitate coalesced memory accesses. In the LVCSR language model, the transition between the states in the transition graph is determined by the vocabulary, but the layout of the states in memory can be optimized for faster execution. In our implementation, as illustrated in Figure 9, we pre-arrange the first states of each word in the transition graph in consecutive memory locations. In the beam search, when we limit the beam width to be at most 2% of the total number of states, the number of the first states being evaluated accounts for more than half of the total number of next states evaluated.

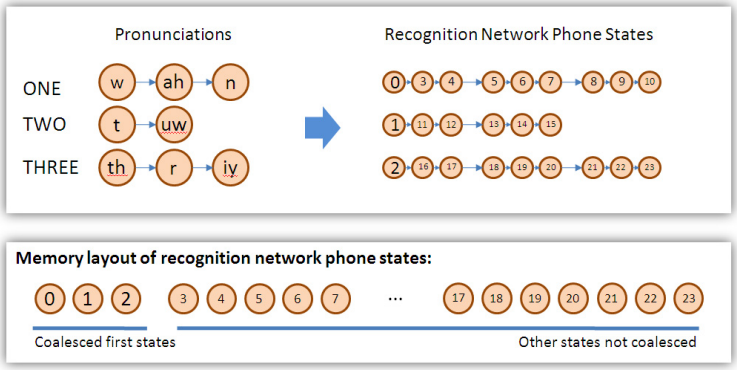

Figure 9. Illustration of memory layout used for transition states on a small three word example

Handling the word-to-word transitions is a complex process that requires significant communication. For a 50k-word vocabulary there are on average 10 active-end-states in a beam search iteration, each of which can transition to any of the 50k words in the vocabulary. For each the 50k times 10 = 500,000 transitions, one must check if there is a word-to-word transition probability stored in the bigram model. If the transition probability is not available, a default probability is computed with the unigram probability of the first state and the backoff

constant of the active-end-state, as explained in [9].

A straight forward approach to implement this transition is to parallelize over the first states, where a thread performs a lookup in the bigram model for word-to-word transitions starting from every active-end-state to the first state associated with the thread. This kind of lookup, however, must access the entire bigram transition table in uncoalesced fashion each beam search iteration, creating a memory bottleneck.

Our solution is an iterative two-step process, where we iterate over the set of active-end-states and parallelize over the first state within each iteration. In the first step, we convert the sparse matrix representation of bigram entries corresponding to the active-end-state of in an iteration to dense matrix representation. In the second step, we parallelize over the first states by looking up bigram word-to-word transition probability in dense matrix format and performing default case computation if necessary. In this step, the dense matrix format is instrumental in allowing this step to be efficiently executed in a data parallel fashion. This approach also reduces the number of uncoalesced memory accesses to about 1/1000 of the size of the bigram transition table.

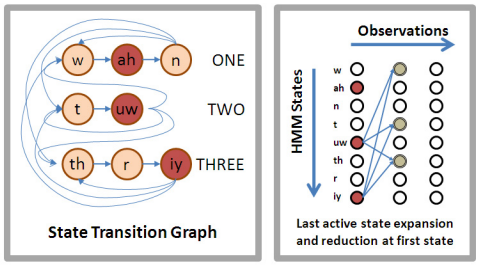

Figure 10. Illustration of state transition

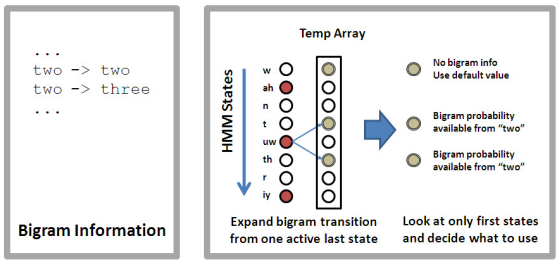

Figure 11. Illustration of bigram computation

4.3 Next state pruning

The pruning process limits the number of active states to the most promising states. We prune based on the beam threshold as defined in [9] and also cap the number of active states below a hard limit to fit in fixed size arrays. This is necessary to avoid memory allocation in the timing sensitive loops.

The maximum likelihood in each beam search iteration is computed with an efficient two level reduction. We use the beam threshold to prune based on likelihood, as thresholding obviates the need for sorting. When there are more states than the capacity of fixed size array, we iteratively reduce the beam threshold until the pruned solution fits within a fixed array limit.

4.4 Final algorithm

The final implement is a four-step algorithm optimized for data parallel execution, shown in Figure 12. Step 0 and step 1 implements the observation probability computation, step 2 implements the next states likelihood computation, and step 3 implements the max reduction and state pruning phase.

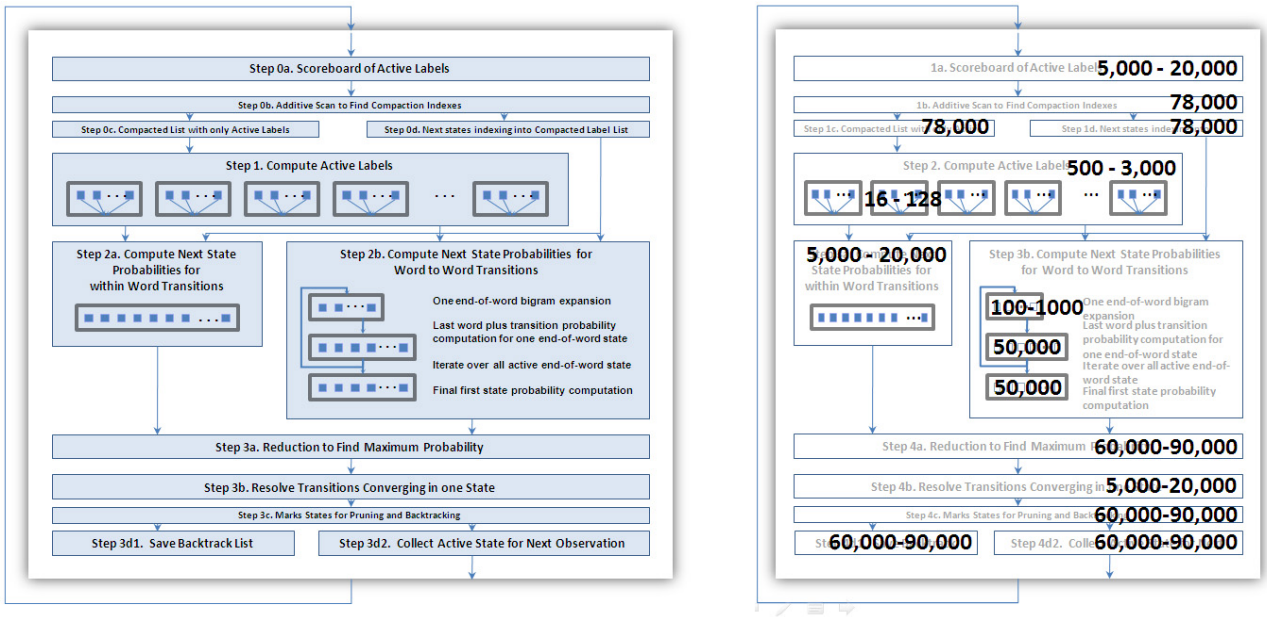

Figure 12. Detailed task graph of the final implementation, and step-wise data-parallelism

There exists some task-level parallelism in a beam search iteration, but the most significant parallelization potential lies in data level parallelism. All steps in our implementation use algorithms and data structures parallelizable to thousands of processors, enabling the continued scaling of the algorithm for future generations of many-core processors.

5. Results

We performed our experiments on an NVIDIA GeForce 8800 GTX running in a PC with a 2.66GHz Intel Core2 Duo, 8 GB of main memory and using a Linux 2.6.22 kernel. The main properties of this GPU are listed in table 3. The GPU used is a widely available $500 consumer platform that can be installed in any desktop PC.

The inference engine recognizes human speech by processing sequences of 10ms speech frames one frame at a time and producing probabilistic estimates of the underlying word sequence based on the language knowledge sources encoded in the recognition network. We used the standard Mel-frequency cepstral coefficients (MFCC)

and their derivatives [9] as the feature vector to associate with each frame.

There are two language models used to test our inference routine. The properties of the language models are shown in Table 4. The large-vocabulary acoustic models were trained on about 250 hours of conversational telephone speech data from the NIST Hub-5 English Large Vocabulary Continuous Speech Recognition evaluation. The test set was taken from the NIST 2001 Speech Transcription Workshop. The small-vocabulary models were trained and tested on the OGI Numbers95 database of spoken digit sequences.

Table 3. GPU parameters

Properties	8800 GTX
Number of multiprocessors (SMs)	16
Multiprocessor width	8
Local memory per SM	16kb
# of processors	128
Off-chip memory	768 MB

Table 4. Test case properties

Properties	Small Case	Large Case
# of words	41	50710
# of states	525	996903
# of mixtures in GMM	128	16
# of unique phones	84	78438
# of bigram edges	600	552531

We validated our inference engine by comparing it with the results of HVite [20], an inference engine in the HTK package. Table 5 shows that both inference engines achieved an error rate of around 16%, and the accuracy difference between the two is within 1%.

Table 5. Accuracy validation

Parameters	Hvite	Our Work
Total error	15.52	15.88
Substitution error	9.05	9.44
Deletion error	1.29	0.86
Insertion error	5.17	5.58

The inference engine was tested on the two recognition models of table 4. The first of these language models is a small example which is much faster than real-time even on a standard desktop 2.66 GHz Intel Core 2 CPU. Thus it is unnecessary to speed it up on a GPU. We only show the performance of the GPU implementation on the larger language model. We tested our engine on a set of 42 speech segments from the NIST 2001 Speech

Table 6. Performance result for 42 utterances totaling 162.9 seconds in length

Properties	CPU runtime	%	CPU+GPU runtime	%	Speedup
Step 0: Collect Unique Active Labels	12590.6 ms	5%	3924.8 ms	15%	3.21x
Step 1: Compute Observation Probabilities	114477.4 ms	49%	5984.6 ms	23%	19.13x
Step 2: Update Next State Likelihoods	82609.6 ms	35%	6989.6 ms	27%	11.82x
Step 3: Reduce and Prune States	24981.7 ms	11%	7817.0 ms	30%	3.20x
Sequential (Loop + Control) Overheads	214.5 ms	0.1%	1338.0 ms	5%	
Total	234874.8 ms		26054.0 ms		9.01x
(Utterance Length / Recognition Time)	0.69x		6.25x		

Transcription Workshop. Table 6 shows the performance of the LVCSR inference engine running as a single thread compiled with Intel C++ Compiler (ICC) version 10.1.008 on the CPU, and compares it with a parallel version running on the NVIDIA 8800 GTX GPU. The test was run with a beam threshold of 150 (as defined in [9]) and an active state cap of 20,000 states.

The Core2 Duo CPU has 4-wide SIMD running at 2.66GHz, and the GeForce 8800 GTX GPU has 128 hardware threads running at 1.35GHz. Adjusting for frequency and hardware thread count, the GPU has $(128 * 1.35)/(4 * 2.66) = 17.5$x the peak computation throughput of the CPU. Assuming that both CPU and GPU fully utilize their respective SIMD units, we should see a 17.5x boost in performance for compute limited kernels.

In table 6, we see a 19x speedup for step 1, the most computationally intensive step, which is higher than expected. Although we enabled the vectorization option for SSE3 in ICC and design the data structures to be vectorizable, we found that ICC is not as effective in utilizing the CPU 4-wide SIMD unit as the CUDA compiler in utilizing the 128 GPU hardware threads.

In our data parallel implementation, we minimized the sequential overhead to be less than 0.1% of the total sequential runtime. This gives an upper bound of 1000x on the speedup over sequential code, such that the application is not bottlenecked by Amdahl's law. In the parallel implementation we see steps 1 and 2, the two most significant steps in the single thread implementation, were significantly improved. The sequential overhead in the CPU+GPU parallel implementation increased slightly, as small amounts of data are copied between GPU and CPU.

Figure 13 shows the detailed break down of the kernel execution times collected using NVIDIA CUDA Visual Profiler 1.0.04. Each row illustrates data collected for a CUDA kernel, and the execution time and speedup numbers are analyzed with respect to a variety of metrics to help explain why the speedup observed may not be the expected 17.5x. IPC is instructions per cycle, computed from the number of dynamic instructions logged during the execution time of the kernel. The kernel overhead is the overhead incurred when launching a kernel, and is measured by the CUDA Profiler. The speedup without accounting for kernel overhead indicates the effectiveness of our data-parallel implementation, without the artifacts of the GeForce 8800 GTX platform. The percentage of coherent memory accesses is computed from the number of coherent/incoherent loads/stores

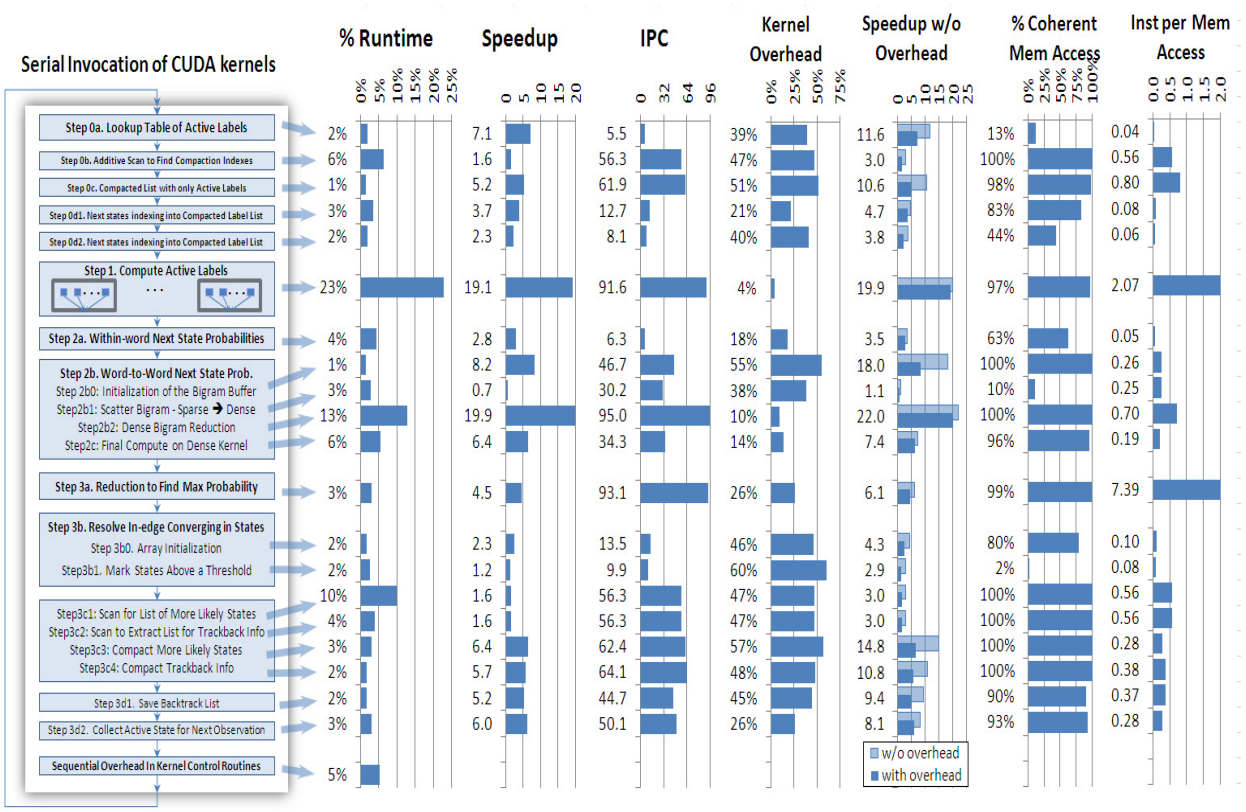

Figure 13. Detailed characterization of the data-parallel LVCSR implementation

provided by the CUDA Profiler. The instructions per memory access column shows how memory intensive the kernel implementation is. We now analyze the kernels using these metrics.

Step 1 is the largest kernel by execution time. We obtained an IPC of more than 90, despite 3% of the memory accesses being uncoalesced. This is possible because the step has a high ratio of instructions per memory access (IPA), such that the long memory latencies can be hidden with computation. Step 2b2 also reached a high IPC of 95. Although it has a relatively low ratio of IPA, the high IPC is achieved because it has fully coalesced memory accesses, which helps to prevent memory bandwidth from becoming a bottleneck.

The scan steps in 0b, 3c1 and 3c2 make up 20% of total execution time. The data-parallel implementation only achieved 1.6x speedup because of three reasons: 1) the sequential version is very work-efficient and touches every element to be scanned just once, where as the data-parallel version leverages the scan implementation from the CUDA Performance Primitive (CUDPP) library 1.0a, which uses an O(n) algorithm that requires a lot more than n operations for n inputs; 2) the need for global synchronization in a scan routine requires mul-

tiple kernel invocations, thus adding significant kernel call overhead penalties; and 3) the implementation of scan in CUDPP is still a subject of research, and the current implementation may not the optimal.

Step 3a uses a two-level reduction approach which is a different algorithm compared to the sequential implementation. The second level is implemented sequentially on the CPU to avoid kernel overhead penalty, but this involves transfer of the intermediate results back to the CPU, reducing the effective speedup to 4.5X.

The rest of the kernels fall under two groups: 1) limited by kernel overhead, and 2) limited by uncoalesced memory accesses. Steps 0a, 0c, 2b0, 3c3, 3c4, 3d1, and 3d2 falls in the first group. These are simple kernels that are only a few lines long, where we would obtain speedups of around 10X if we discount the kernel overhead. The second group of kernels include steps 0d1, 0d2, 1a, 2b1, 2c, 3b0 and 3b1, where the kernels have low IPA ratio and are hence memory bound, while at the same time the algorithm requires uncoalesced memory accesses, aggravating the memory bottleneck.

We obtained an overall speedup of 9.0X for the GPU over the CPU. Our key result is that the GPU implementation is about 6.25X better than the performance required for real-time recognition.

Pushing the performance beyond that is required for real-time recognition is crucial since this allows us to handle more complex language models and increase the accuracy of recognition by using multi-pass inference algorithms under real time constraints. Further, speech recognition could be just one of the tasks in an application like language translation with spoken inputs which have real-time requirements.

6. Discussions on further optimizations

In this paper, we establish a baseline infrastructure for data-parallel beam search in LVCSR on a standard HMM implementation. We found that the most significant computation bottleneck is in the GMM computation for observation probabilities, and the most significant communication bottleneck is in the bigram transition computation. These are the same performance bottlenecks seen in sequential implementations. There is a large body of prior work on optimization techniques to mitigate these bottlenecks on sequential platforms. We discuss their applicability for a data-parallel implementation here.

To reduce the amount of GMM computation, techniques proposed in literature include Bucket Box Intersection (BBI)[7], Subspace Distribution Clustering for Continuous observation density HMM (SDCHMM)[4],

and Gaussian selection[10]. From a high level point of view, these techniques trade off computation with decision trees or lookup tables to reduce execution time. While these are valid trade-offs for current sequential platforms, implementing these optimizations on a data-parallel platform with SIMD execution units would introduce uncoalesced memory accesses and divergent branches in instruction streams, severely limiting overall instruction throughput. The benefit of these techniques are reported to be limited to 2-5x reduction in the GMM computation. Since we find in our experiments that execution time on SIMD-based data-parallel platforms can vary by an order of magnitude in the presence of uncoalesced memory accesses and divergent branches, it does not seem promising that these optimizations will lead to runtime improvements.

To improve the recognition network representation for more effective most-likely word sequence searches, the Weighted Finite State Transducer (WFST)[15] model has been recently proposed to allow a set of FSM optimization techniques such as factorization and determinization to be used to minimize the state space. This is achieved by using a Mealy machine representation for the state transition graph, where word sequences are emitted from the state transition edges, rather than the standard Moore machine representation used in our work, where word sequences are emitted from the states. Since the recognition network optimizations are performed at compile time, the optimization techniques are transparent to the inference engine, and will equally benefit a sequential and a data-parallel implementations. We are pursuing WFST as a promising extension of our work.

7. Conclusions

Automatic speech recognition is a key technology for enabling rich human-computer interaction in emerging applications. This paper explores the concurrency present in continuous speech recognition with large vocabulary models and presents an inference engine implementation on an NVIDIA many-core GPU using the CUDA programming language.

We establish a baseline infrastructure for data-parallel beam search in LVCSR on a standard HMM implementation. Our results show a 9.01X speedup over a sequential implementation on CPU. This corresponds to going from 0.69X the real-time requirement to over 6.25X the real time requirements. Pushing the performance beyond real time is crucial for speech recognition since we can then handle more complex language models and can incorporate more accurate multi-pass inference engines for more accurate recognition. Further, we expect

that speech recognition will be just one step in exciting new applications with real-time requirements such as language translation.

We are exploring an extension to handle WFST-based recognition networks and would like to explore the potential speedups on other manycore processor as they become available.

References

[1] Optimizing Parallel Reduction with CUDA. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[2] K. Agaram, S. Keckler, and D. Burger. A characterization of speech recognition on modern computer systems. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 45–53, 2 Dec. 2001.

[3] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[4] E. Bocchieri and B. Mak. Subspace distribution clustering for continuous observation density hidden markov models. In *Proc. Eurospeech '97*, pages 107–110, Rhodes, Greece, 1997.

[5] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`.

[6] P. Dixon, D. Caseiro, T. Oonishi, and S. Furui. The titech large vocabulary wfst speech recognition system. *Automatic Speech Recognition & Understanding, 2007. IEEE Workshop on*, pages 443–448, 9-13 Dec. 2007.

[7] J. Fritsch and I. Rogina. The bucket box intersection (BBI) algorithm for fast approximative evaluation of diagonal mixture gaussians. In *Proc. ICASSP '96*, pages 837–840, Atlanta, GA, 1996.

[8] H.-W. Hon. A survey of hardware architectures designed for speech recognition. Technical report, CMU, 1991.

[9] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice-Hall, 2001.

[10] K. Knill, M. Gales, and S. Young. Use of gaussian selection in large vocabulary continuous speech recognition using hmms. *Proc. Fourth Intl. Conf. on Spoken Language, ICSLP 96.*, 1:470–473 vol.1, 3-6 Oct 1996.

[11] R. Krishna, S. Mahlke, and T. Austin. Architectural optimizations for low-power, real-time speech recognition. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 220–231, New York, NY, USA, 2003. ACM.

[12] Intel plans powerful multicore x86 cpu. `http://www.pcworld.com/businesscenter/article/130815/intel_plans_powerful_multicore_x86_cpu.html`.

[13] A. Lee, T. Kawahara, and K. Shikano. Julius — an open source real-time large vocabulary recognition engine. In *Proc. European Conf. on Speech Communication and Technology (EUROSPEECH)*, pages 1691–1694, 2001.

[14] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *Acoustics, Speech, and Signal Processing, IEEE Transactions on*, 38(1):35–45, Jan 1990.

[15] M. Mohri, F. Pereira, and M. Riley. Weighted finite state transducers in speech recognition. *Computer Speech and Language*, 16:69–88, 2002.

[16] J. Nickolls, I. Buck, K. Skadron, and M. Garland. CUDA: Scalable parallel programming. *ACM Queue*, Apr 2008.

[17] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, Nov. 2007. Version 1.1.

[18] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition, 1993.

[19] J. A. Stratton, S. S. Stone, and W. W. Hwu. M-CUDA: An efficient implementation of CUDA kernels on multi-cores. IMPACT Technical Report IMPACT-08-01, UIUC, Feb. 2008.

[20] P. Woodland, J. Odell, V. Valtchev, and S. Young. Large vocabulary continuous speech recognition using HTK. *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, ICASSP 1994*, 2:125–128, Apr. 1994.