A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit

Jike Chong, Ekaterina Gonina, Youngmin Yi, Kurt Keutzer

Department of Electrical Engineering and Computer Science, University of California, Berkeley

{ jike, egonina, ymyi, keutzer}@eecs.berkeley.edu

Abstract

Tremendous compute throughput is becoming available in personal desktop and laptop systems through the use of graphics processing units (GPUs). However, exploiting this resource requires re-architecting an application to fit a data parallel programming model. The complex graph traversal routines in the inference process for large vocabulary continuous speech recognition (LVCSR) have been considered by many as unsuitable for extensive parallelization. We explore and demonstrate a fully data parallel implementation of a speech inference engine on NVIDIA’s GTX280 GPU. Our implementation consists of two phases - compute-intensive observation probability computation phase and communication-intensive graph traversal phase. We take advantage of dynamic elimination of redundant computation in the compute-intensive phase while maintaining close-to-peak execution efficiency. We also demonstrate the importance of exploring application-level trade-offs in the communication-intensive graph traversal phase to adapt the algorithm to data parallel execution on GPUs. On 3.1 hours of speech data set, we achieve more than $11 \times$ speedup compared to a highly optimized sequential implementation on Intel Core i7 without sacrificing accuracy.

Index Terms: Data parallel, Continuous Speech Recognition, Graphics Processing Unit

1. Introduction

Graphics processing units (GPUs) are enabling tremendous compute capabilities in personal desktop and laptop systems. Recent advances in the programming model for GPUs such as CUDA [1] from NVIDIA have provided an implementation path for many exciting applications beyond graphics processing such as speech recognition. In order to take advantage of high throughput capabilities of the GPU-based platforms, programmers need to transform their algorithms to fit the data parallel model. This can be challenging for algorithms that don’t directly map onto the model, such as graph traversal in a speech inference engine.

In this paper we explore the use of GPUs for large vocabulary continuous speech recognition (LVCSR) on NVIDIA GTX280 GPU. A LVCSR application analyzes a human utterance from a sequence of input audio waveforms to interpret and distinguish the words and sentences intended by the speaker. Its top level architecture is shown in Fig. 1. The recognition process uses a Weighted Finite State Transducer (WFST) based recognition network [2], which is a language database that is compiled offline from a variety of knowledge sources using powerful statistical learning techniques. The speech feature extractor collects feature vectors from input audio waveforms, and then the Hidden-Markov-Model-based inference engine computes the most likely word sequence based on the extracted speech features and the recognition network. In the inference process, we focus on large vocabulary continuous speech recognition (LVCSR) on NVIDIA’s GTX280 GPU. Our implementation consists of two phases - compute-intensive observation probability computation phase and communication-intensive graph traversal phase. We take advantage of dynamic elimination of redundant computation in the compute-intensive phase while maintaining close-to-peak execution efficiency. We also demonstrate the importance of exploring application-level trade-offs in the communication-intensive graph traversal phase to adapt the algorithm to data parallel execution on GPUs.

LVCSR system the common speech feature extractors can be parallelized using standard signal processing techniques. On the other hand, the graph traversal routines have been considered unsuitable for extensive parallelization [3, 4]. This paper discusses the application-level trade-offs that need to be made in order to efficiently parallelize the graph traversal process in LVCSR and illustrates the performance gains obtained from effective parallelization of this portion of the algorithm.

A parallel inference engine traverses a graph-based knowledge network consisting of millions of states and arcs. As shown in Fig. 1, it uses the Viterbi search algorithm to iterate through a sequence of input audio features one time step at a time [5]. The Viterbi search algorithm keeps track of each alternative interpretation of the input utterance as a sequence of states ending in an active state at the current time step. It evaluates outgoing arcs based on the current-time-step observation to arrive at the next set of active states. Each time step consists of two phases: Phase 1 - observation probability computation and Phase 2 - graph traversal computation. Phase 1 is compute-intensive while Phase 2 is communication-intensive.

The inference engine is implemented using CUDA, which requires the computation to be organized into a sequential host program on a CPU calling parallel kernels running on the GPU. A kernel executes a scalar sequential program across a set of parallel threads where each thread operates on a different piece of data. The CPU and the GPU have separate memory spaces and there is an implicit global barrier between different kernels, as illustrated at the bottom of Fig. 1.

Managing aggressive pruning techniques to keep LVCSR computationally tractable requires frequent global synchronizations. We are keeping track of on average 0.1-1.0% of the total state space and must communicate the pruning bounds every time step. There exist significant parallelism opportunities in each time step of the inference engine. For example, we can...
evaluate thousands of alternative interpretations of a speech utterance concurrently. At the same time, the inference engine involves a parallel graph traversal through a highly irregular knowledge network. The traversal is guided by a sequence of input audio features that continuously changes the working set at run time. The challenge is to not only define a software architecture that exposes sufficient fine-grained application concurrency (Section 3.1), but also to extract close-to-peak performance on the GPU platform (Section 3.2). We also explore alternatives in the recognition network structure for more efficient execution on a data parallel implementation platform (Section 3.3).

2. Related Work

There have been many efforts in paralleling LVCSR. We highlight three categories of efforts in software-based acceleration.

Category 1: Data Parallel. Ishikawa et al. [9] exploited pipelined task-level parallelism on three ARM cores. Here, scaling requires extensive redesign effort.


Category 3: Data Parallel implementation on manycore accelerator in CPU-based host systems [10, 11, 12]. [10, 11] focused on speeding up the compute intensive phase but left the communication intensive phases on the host platform, thereby limiting their scalability. [12] leveraged the simpler structure of a linear-lexicon based (LL) recognition network to achieve a 9× speedup compared to a highly optimized sequential implementation. However, LL-based recognition networks are less efficient than the WFST-based recognition networks [13, 14].

In contrast, we optimized our software architecture by implementing data parallel versions of both of the observation probability computation and the graph traversal phases for a GPU platform. We used the more challenging WFST-based recognition network and achieved greater speedups in each of the two phases.

3. Data Parallel Inference Engine

Among the two phases of the inference engine shown in Fig. 1, the compute-intensive phase involves using Gaussian Mixture Models (GMM) to estimate the likelihood that an input audio feature matches a triphone state. This phase maps well to highly parallel platforms such as GPUs. The communication-intensive phase involves traversing through a highly irregular recognition network, while managing a dynamic working set that changes every time step based on input audio features. Although this phase is highly challenging to implement on parallel platforms, we demonstrate that with carefully managed application-level trade-offs, significant speedups can still be achieved.

3.1. Overall Optimizations

Implementing both phases on the GPU has significant advantages over separately implementing the compute-intensive phase on the GPU and the communication-intensive phase on the CPU. A split implementation incurs high data-copying overheads between the CPU and the GPU for transferring large amounts of intermediate results. It is also less scalable as the transfers become a sequential bottleneck in the algorithm. Implementing all phases to run on the GPU eliminates data transfers between the CPU and the GPU and allows for more scalable parallel pruning routines.

Both of the phases extensively use the vector units on the GPU, which require coalesced memory accesses and synchronized instruction execution. Memory accesses are coalesced when data is referenced from consecutive memory locations so it can be loaded and used in a vector arithmetic unit directly without rearrangement. The kernels are written to have synchronized instruction control flow so that all lanes in a vector unit are doing useful work while executing the same operation, i.e., the Single-Instruction-Multiple-Data (SIMD) approach.

To maximize coalesced memory accesses, we create a set of buffers to gather the active state and arc information from the recognition network at the beginning of each time step for all later references in that time step. In addition, we use arc-based traversal where each SIMD lane is assigned to compute one out-going arc. Since the amount of computation is the same for all out-going arcs, all SIMD lanes are synchronized during this computation. This approach yields more efficient SIMD utilization and results in 5× performance gain for the communication intensive phase compared to traversing the graph with one state per SIMD lane, where each lane has different amount of work depending on the number of out-going arcs a state has.

To coordinate the graph traversal across cores, we extensively use atomic operations on the GPU. When computing the arc with the most-likely incoming transition to a destination state, each arc transition updates a destination state atomically. This efficiently resolves write-conflicts when multiple cores compute arcs that share the same destination state.

3.2. Compute-intensive Phase Optimization

In the compute-intensive phase, we compute the observation probability of triphone states. This involves two steps: (1) GMM computation and (2) logarithmic mixture reduction. Our implementation distributes the clusters across GPU cores and uses parallel resources within-core to compute each cluster’s mixture model. Both steps scale well on highly parallel processors and the optimization is in eliminating redundant work.

A typical recognition network has millions of arcs, each labeled with one of the approximately 50,000 triphone states. Furthermore, the GMM for the triphone states can be clustered into 2000-3000 clusters. In each time step, on average only 60% of the clusters and 20% of the triphone states are used.

We prune the list of GMM and triphone states to be computed in each time step based on the lexicon model compiled into the WFST recognition network. We remove the redundant GMM and triphone states from consideration for each time step, thereby reducing the computation time for this phase by 70%.

3.3. Communication-intensive Phase Optimizations

The communication-intensive phase involves a graph traversal process through an irregular recognition network. There are two types of arcs in a WFST-based recognition network: arcs with an input label (non-epsilon arcs), and arcs without input labels (epsilon arcs). In order to compute the set of next states in a given time step, we must traverse both the non-epsilon and all the levels of epsilon arcs from the current set of active states. This multi-level traversal can impair performance significantly.

We explore the modification of flattening the recognition network to reduce the number of levels of arcs that need to be traversed and observe corresponding performance improvements. To illustrate this, Fig. 2 shows a small section of a WFST-based...
4.4. Communication-intensive Phase

Parallelizing this phase on a quadcore CPU achieves a 2.85 × performance gain [15] and incurs intermediate result transfer overhead. We achieved a 3.84 × performance gain with an equivalent configuration on the GPU and avoided intermediate

1A single-pass time-synchronous Viterbi decoder from SRI using lexical tree search achieves 37.9% WER on this test set
result transfer overhead. Despite the better speedup on the GPU, this phase became more dominant as shown in Fig 3.

Table 1 demonstrates the trade-offs of recognition network augmentation for efficient data parallel traversal in our inference engine. The augmentation for Two-Level setup resulted in a 2.0% increase in arc count and the augmentation for One-Level setup resulted a 32.2% increase. The dynamic number of arcs evaluated increased marginally for the Two-Level setup. However for the One-Level solution it increased significantly by 48-62%, as states with more arcs were visited more frequently.

Fig 4 shows the run time for various pruning thresholds. The network modifications are described in Section 3.3. Without network modifications, there is significant performance penalty as multiple levels of epsilon arcs must be traversed with expensive global synchronization steps between levels. With minimal modifications to the network, we see a 17-24% speedup for this phase. An additional 8-29% speedup can be achieved by eliminating epsilon arcs completely, saving the fixed cost of one level of global synchronization routines, but this comes at the cost of traversing 48-62% more arcs.

5. Conclusion

We presented a fully data parallel speech inference engine with both observation probability computation and graph traversal implemented on an NVIDIA GTX280 GPU. Our results show that modifications to the recognition network are essential for effective implementation of data parallel WFST-based LVCSR algorithm on GPU. Our implementation achieved up to 11.7x speedup compared to highly optimized sequential implementation with 5-8% sequential overhead without sacrificing accuracy. This software architecture enables performance improvement potentials on future platforms with more parallelism.

6. References