Scalable HMM based Inference Engine in Large Vocabulary Continuous Speech Recognition

Jike Chong† Kisun You*‡ Youngmin Yi† Ekaterina Gonina† Christopher Hughes‡ Wonyong Sung* Kurt Keutzer†

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

* School of Electrical Engineering, Seoul National University

‡ Intel Corporation

***Abstract*—Parallel scalability allows an application to efficiently utilize an increasing number of processing elements. In this paper we explore a design space for application scalability for an inference engine in large vocabulary continuous speech recognition (LVCSR). Our implementation of the inference engine involves a parallel graph traversal through an irregular graph-based knowledge network with millions of states and arcs. The challenge is not only to define a software architecture that exposes sufficient fine-grained application concurrency, but also to efficiently synchronize between an increasing number of concurrent tasks and to effectively utilize the parallelism opportunities in today's highly parallel processors. We propose four application-level implementation alternatives we call "algorithm styles", and construct highly optimized implementations on two parallel platforms: an Intel Core i7 multicore processor and a NVIDIA GTX280 manycore processor. The highest performing algorithm style varies with the implementation platform. On 44 minutes of speech data set, we demonstrate substantial speedups of 3.4× on Core i7 and 10.5× on GTX280 compared to a highly optimized sequential implementation on Core i7 without sacrificing accuracy. The parallel implementations contain less than 2.5% sequential overhead, promising scalability and significant potential for further speedup on future platforms.**

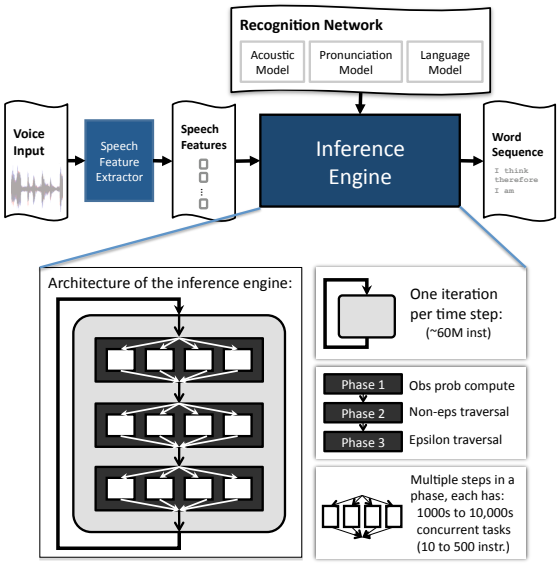

Fig. 1. Architecture of large vocabulary continuous speech recognition

I. Introduction

We have entered a new era where sequential programs can no longer fully exploit Moore's Law and expect a doubling in performance every 18-24 months [1]. Parallel scalability, the ability for an application to efficiently utilize an increasing number of processing elements, is now required for software to obtain sustained performance improvements on successive generations of processors.

Many modern signal processing applications are evolving to incorporate recognition backends that have significant scalability challenges. We examine the scalability challenges in implementing a Hidden-Markov-Model (HMM) based inference algorithm in a large-vocabulary-continuous-speech-recognition (LVCSR) application.

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 *recognition network*, 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 discriminant feature vectors from input audio waveforms. Then the *inference engine* computes the most likely word sequence based on the extracted speech features and the recognition network. In the LVCSR system the common speech feature extractors can be parallelized using standard signal processing techniques [2], [3]. This paper discusses the well-known parallelization challenges for graph traversal [4] in the context of the inference engine [5].

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 feature one time step at a time. The Viterbi search algorithm keeps track of each alternative interpretation as a sequence of states ending in an active state at the current time step and evaluates out-going arcs based on the current-time-step observation to arrive at the set of active states for the next time step. In each time step, it executes in three phases of algorithm steps, keeping track of tens of thousands of alternative interpretations of a speech utterance to select the most likely word sequence [6]. Phase 1 is a compute-intensive phase, and phase 2 and 3 are communication intensive phases. There are significant parallelism opportunities in concurrently evaluating the alternative interpretations of a speech utterance in each algorithm step. However, the inference engine involves a parallel graph traversal through an irregular graph-based knowledge network with millions of states and arcs. It is guided by a sequence of input audio features that continuously changes the data working set at runtime. The challenge is not only to define a software architecture that exposes sufficient fine-grained application concurrency, but also to efficiently synchronize between

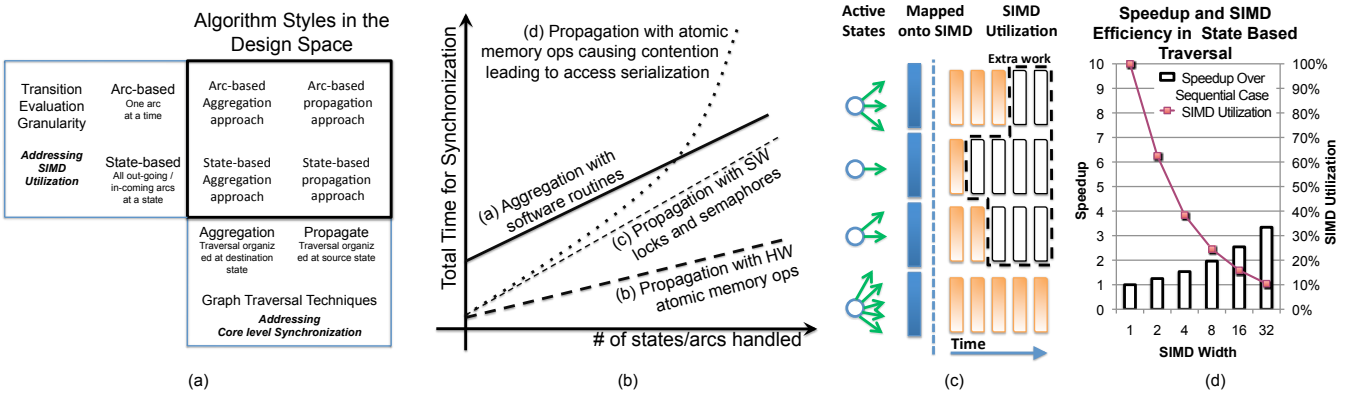

Fig. 2. The algorithmic level design space for graph traversal scalabiliy analysis for the inference engine

an increasing number of concurrent tasks and to effectively utilize the parallelism opportunities in today's highly parallel processors.

We explore two important parallelization challenges for graph traversal in the context of an inference engine: efficient synchronization between concurrent tasks and effective utilization of Single-Instruction-Multiple-Data (SIMD) parallelism. We propose two application-level implementation alternatives for each of the parallelization challenges and compose them to arrive at a design space of four unique algorithm styles . We construct highly optimized implementations of the algorithm styles on two parallel platforms: an Intel Core i7 multicore processor and a NVIDIA GTX280 manycore processor, and present our results here.

II. Related Work

Speech recognition is a challenging computational problem that has attracted many researchers. We highlight related work in software-based acceleration in three categories.

Category 1: Data Parallel, multiprocessor shared memory implementation on multi-processor clusters [7], [8]. These implementations are plagued by high communication overhead in the platform, high sequential overhead in the software architecture, load imbalance among parallel tasks or excessive memory bandwidth requirement, thus limiting the scalability to more parallel platforms. In [9], some of these issues were resolved by using OpenMP as an implementation platform, however, it was based on the tree-lexicon search network, a less efficient approach than the WFST-based approach [10] used in this paper.

Category 2: Task parallel implementation. Ishikawa *et al.* [11] 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 system [12], [13], [14]. [12], [13] focused on speeding up the compute intensive phase, but left the communication intensive phases on the host platform, limiting their scalability. [14] leveraged the simpler structure of a linear-lexicon based recognition network to achieve a 9× speedup compared to a SIMD optimized sequential implementation. It is less efficient than the WFST-based approach.

In contrast, we optimize our software architecture, accomplishing additional speedup in the computation intensive phase on the manycore accelerator, and computing the communication intensive phases in parallel on the multicore and manycore processors. We explore multiple scalable synchronization methods, while traversing the more challenging WFST-based recognition network.

III. Algorithm Styles of the Inference Engine

Given the challenging and dynamic nature of the underlying graph-traversal routines in LVCSR, implementing it on parallel platforms presents two architectural challenges: efficient SIMD utilization and efficient core level synchronization. These challenges are key factors in making the algorithms scalable to increasing number of cores and SIMD lanes. To find a solution to these challenges we explore two aspects of the algorithmic level design space: the graph traversal technique and the arc transition evaluation granularity. Our design space is shown in Fig. 2a.

A. Traversal Techniques: Aggregate or Propagate

The two graph traversal techniques are traversal by *propagation* and traversal by *aggregation*. During the graph traversal process, each arc has a source state and a destination state. Traversal by *propagation* organizes the traversal process at the source state. It evaluates the outgoing arcs of the active states and *propagates* the result to the destination states. As multiple arcs may be writing their result to the same destination state, this technique requires write conflict resolution support in the underlying platform. The programmer declares certain memory operations as atomic, and the implementation platform resolves the potential write conflicts.

Traversal by *aggregation* organizes the traversal process around the destination state. The destination states update their own information by performing a reduction on the evaluation results of their incoming arcs. The programmer explicitly manages the potential write conflicts by using additional algorithmic steps such that no write-conflict-resolution support is required in the underlying platform.

The choice of the traversal technique has direct implications on the cost of core level synchronization. Efficient synchronization between cores reduces the management overhead of a parallel algorithm and allows the same problem to gain additional speedups as we scale to more cores. Fig. 2b outlines

the trade-offs in the total cost of synchronization between the aggregation technique and the propagation technique. The qualitative graph shows increasing synchronization cost with increasing number of concurrent states or arcs evaluated.

The fixed cost for the aggregation technique (Y-intercept of line (a) in Fig. 2b) is higher than that of the propagation technique, as it requires a larger data structure and a more complex set of software routines to manage potential write conflicts. The relative gradient of the aggregation and propagation techniques depends on the efficiency of the platform in resolving potential write conflicts. If efficient hardware-supported atomic operations are used, the variable cost for each additional access would be small, and the propagation technique should scale as line (b) in Fig. 2b. If there is no hardware support for atomic operations, and sophisticated semaphores and more expensive software-based locking routines are used, the propagation technique would scale as line (c). In addition, if the graph structure creates a scenario where many arcs are contenting to write to a small set of next states, serialization bottleneck may appear and the propagation technique could scale as line (d).

In order to minimize the synchronization cost for a given problem size, we need to choose the approach corresponding to the lowest-lying line in Fig. 2b. For small number of active states or arcs, we should choose the propagation technique. For larger number of arcs, however, the choice is highly dependent on the application graph structure and the write-conflict-resolution support in the underlying implementation platform.

B. Evaluation Granularity: Arc-Based or State-Based

We also explore two recognition network evaluation granularities: *state-based* evaluation and *arc-based* evaluation. In a parallel implementation we must define units of work (or tasks) that can be done concurrently. *State-based* evaluation defines a unit of work as the evaluation of all outgoing or incoming arcs associated with a *state*, with the majority of states having one or two outgoing or incoming arcs. *Arc-based* evaluation defines a unit of work as the evaluation of a single *arc*. The fine granularity of tasks allows the workload to scale to increasingly parallel implementation platforms. Each fine-grained task, however, has little instruction-level parallelism and can not fully utilizing a growing number of SIMD lanes in a core. We must efficiently map tasks to SIMD lanes to gain higher SIMD utilization, such that the algorithm can scale to even wider SIMD units in future processors.

SIMD operations improve performance by executing the same operation on a set of data elements packed into a contiguous vector. Thus, SIMD efficiency is highly dependent on the ability of all lanes to synchronously execute useful instructions. When all lanes are fully utilized for an operation, we call the operation "coalesced". When operations do not coalesce, the SIMD unit becomes under-utilized.

For the state-based approach, we see in Fig. 2c that the control flow diverges as some lanes are idle, while others are doing useful work. In our recognition network, the number of outgoing arcs of the active states ranges from 1 to 897. The bar chart in Fig. 2d shows that the state-based evaluation granularity incurs significant penalties with increasing SIMD width. A 32-wide SIMD achieves only 10% utilization and gets only a 3.3× speedup over a sequential version.

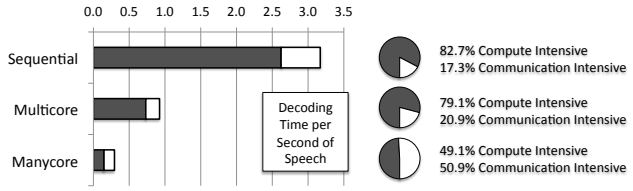

Fig. 3. Ratio of computation intensive phase of the algorithm vs communication intensive phase of the algorithm

We can eliminate this kind of control flow divergence by using the arc-based approach, as each arc evaluation presents a constant amount of work. However, such fully coalesced control flow requires extra storage overhead. In order for each arc evaluation to be an independent task, the arc must have a reference to its source state. We must store this information for every arc we evaluate. For a small SIMD width, this overhead may eliminate any gains from coalesced control flow.

IV. Evaluation of the Inference Engine

We explore the traversal techniques and evaluation granularities on two hardware platforms: the Intel Core i7 920 multicore processor with 6GB memory and the NVIDIA GTX280 manycore processor with a Core2 Quad based host system with 8GB host memory and a GTX280 graphics card with 1GB of device memory. The Core i7 was programmed using the task queue abstraction [15], and the GTX280 was programmed using CUDA [16]. The results are shown in Table II.

Fig. 3 highlights the growing importance of optimizing the communication intensive phases on parallel platforms. Over 80% of the time in the sequential implementation is spent in the compute intensive phases of the application. While the compute intensive phase achieved a 4-20× speedup in our highly parallel implementations, the communication intensive phases incurred significant overhead in managing the parallelism. Even with a respectable 3× speedup, the communication intensive phases became proportionally more dominant, taking 49% of total runtime in the manycore implementation. This motivates the need to examine in detail the parallelization issues in the communication intensive phases of our inference engine. We mainly analyze its synchronization efficiency and effectiveness of using SIMD parallelism.

TABLE I
Accuracy, word error rate (WER), for various beam sizes and corresponding decoding speed in real-time factor (RTF)

Avg. # of Active States		32820	20000	10139	3518
WER		41.6	41.8	42.2	44.5
RTF	Sequential	4.36	3.17	2.29	1.20
	Multicore	1.23	0.93	0.70	**0.39**
	Manycore	**0.40**	0.30	0.23	0.18

The speech models were taken from the SRI CALO real-time meeting recognition system [17], trained using the data

TABLE II
RECOGNITION PERFORMANCE NORMALIZED FOR ONE SECOND OF SPEECH FOR DIFFERENT ALGORITHM STYLES WITH AVERAGE OF 20,000 ACTIVE STATES EACH ITERATION. SPEEDUP REPORTED OVER SIMD-OPTIMIZED SEQUENTIAL VERSION. RESULTS EXPLAINED ARE IN BOLD.

	Core i7	Core i7			GTX280			
Seconds (%)	Sequential Prop. by states	Prop. by states	Prop. by arcs	Aggr. by states	Prop. by states	Prop. by arcs	Aggr. by states	Aggr. by arcs
Phase 1	2.623 (83%)	0.732 (79%)	0.737 (73%)	0.754 (29%)	0.148 (19%)	0.148 (49%)	0.147 (12%)	0.148 (16%)
Phase 2	0.474 (15%)	0.157 (17%)	0.242 (24%)	1.356 (52%)	0.512 (66%)	0.103 (34%)	0.770 (64%)	0.469 (51%)
Phase 3	0.073 (2%)	0.035 (4%)	0.026 (3%)	0.482 (19%)	0.108 (15%)	0.043 (14%)	0.272 (23%)	0.281 (31%)
Sequential Overhead	-	0.001	0.001	0.001	0.008(1.0%)	**0.008(2.5%)**	0.014(1.2%)	0.014(1.6%)
Total	3.171	0.925	1.007	2.593	0.776	0.301	1.203	0.912
Speedup	1	**3.43**	3.15	1.22	4.08	**10.53**	2.64	3.48

and methodology developed for the SRI-ICSI NIST RT-07 evaluation [18]. The acoustic model includes 128-component Gaussians. The pronunciation model contains 59K words and 80K pronunciations. The recognition network contains 4.1M states and 9.8M arcs and is composed using WFST techniques.

The test set consisted of 44 minutes of segmented audio from 10 speakers in NIST conference meetings. The recognition task is very challenging due to the spontaneous nature of the speech. The ambiguities in the sentences require a larger number of active states to keep track of alternative interpretations which leads to slower recognition speed.

As shown in Table I, the multicore and manycore implementations can achieve significant speedup for the same number of active states. More importantly, for the same real time factor (RTF), parallel implementations provide a higher recognition accuracy. For example, for an RTF of 0.4, accuracy improves from 44.5% to 41.6% WER going from a multicore implementation to manycore implementation.

Our implementations are structured to be scalable. As shown in Table II the sequential overhead in our implementations was measured to be less than 2.5% even for the fastest implementation. Also seen in Table II, the fastest algorithm style differed for each platform. Synchronization using the aggregation technique has an overwhelming overhead, despite using highly optimized software routines. The propagation technique, in comparison, had strictly better results. However, our first propagation implementation on GTX280 using global atomic operations had severe atomic-memory-access conflicts and performed worse than Core i7. The issue was resolved by using local atomics operations [16].

The GTX280 has a logical SIMD width of 32. The implementation on GTX280 benefited significantly from evaluation by arc, re-gaining the lost efficiencies seen in Fig. 2d, making propagation by arc the fastest algorithm style. However, SIMD has not been applied to the implementation on Core i7, since the overhead of coalescing control flow exceeds the benefit comes from 4-wide SIMD. Thus, the fastest algorithmic style on Core i7 was propagation by state.

V. CONCLUSIONS

We explored two important aspects of the algorithmic level design space for application scalability to account for differing support and efficiency of concurrent task synchronization and SIMD utilization on multicore and manycore platforms. While we achieved significant speedups compared to highly optimized sequential implementation: 3.4× on an Intel Core i7 multicore processor and 10.5× on a GTX280 NVIDIA manycore processor, the fastest algorithm style differed for each platform. Application developers must take into account underlying hardware architecture features such as synchronization operations and the SIMD width when they design algorithm for parallel platforms.

Automatic speech recognition is a key technology for enabling rich human-computer interaction in emerging applications. Parallelizing its implementation is crucial to reduce recognition latency, increase recognition accuracy, enabling the handling more complex language models under time constrains. We expect that an efficient speech recognition engine will be an important component in many exciting new applications to come.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[2] A. Obukhov and A. Kharlamov, "Discrete cosine transform for 8x8 blocks with CUDA," *NVIDIA white paper*, October 2008.

[3] V. Podlozhnyuk, "FFT-based 2D convolution," *NVIDIA white paper*, June 2007.

[4] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, 2007.

[5] A. Janin, "Speech recognition on vector architectures," Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, 2004.

[6] H. Ney and S. Ortmanns, "Dynamic programming search for continuous speech recognition," *IEEE Signal Processing Magazine*, vol. 16, pp. 64–83, 1999.

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

[8] S. Phillips and A. Rogers, "Parallel speech recognition," *Intl. Journal of Parallel Programming*, vol. 27, no. 4, pp. 257–288, 1999.

[9] K. You, Y. Lee, and W. Sung, "OpenMP-based parallel implementation of a continous speech recognizer on a multi-core system," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009.

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

[11] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Toulouse, France, 2006.

[12] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009.

[13] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU accelerated acoustic likelihood computations," in *Proc. Interspeech*, 2008.

[14] J. Chong, Y. Yi, N. R. S. A. Faria, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *Proc. Intl. Workshop on Emerging Applications and Manycore Architectures*, 2008.

[15] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. Intl. Symposium on Computer Architecture (ISCA)*, 2007.

[16] *NVIDIA CUDA Programming Guide*, NVIDIA Corporation, 2009, version 2.2 beta. [Online]. Available: http://www.nvidia.com/CUDA

[17] G. T. et al, "The CALO meeting speech recognition and understanding system," in *Proc. IEEE Spoken Language Technology Workshop*, 2008, pp. 69–72.

[18] A. Stolcke, X. Anguera, K. Boakye, O. Cetin, A. Janin, M. Magimai-Doss, C. Wooters, and J. Zheng, "The SRI-ICSI spring 2007 meeting and lecture recognition system," *Lecture Notes in Computer Science*, vol. 4625, no. 2, pp. 450–463, 2008.