An Automatic Speech Recognition Application Framework for Highly Parallel Implementations on the GPU

Jike Chong
Ekaterina Gonina
Dorothea Kolossa
Steffen Zeiler
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-47
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-47.html

April 26, 2012

An Automatic Speech Recognition Application Framework for Highly Parallel Implementations on the GPU

Jike Chong*, Ekaterina Gonina*, Dorothea Kolossa*†, Steffen Zeiler†, Kurt Keutzer*

*Department of Electrical Engineering and Computer Science, University of California, Berkeley
†Fachgebiet Elektronik und medizinische Signalverarbeitung, Technische Universität Berlin
jike@eecs.berkeley.edu, egonina@eecs.berkeley.edu, d.kolossa@ee.tu-berlin.de, steffen.zeiler@gmx.de, keutzer@eecs.berkeley.edu

Abstract

Data layout, data placement, and synchronization processes are not usually part of a speech application expert's daily concerns. Yet failure to carefully take these concerns into account in a highly parallel implementation on the graphics processing units (GPUs) could mean an order of magnitude of loss in application performance. In this paper we present an application framework for parallel programming of automatic speech recognition (ASR) applications that allows a speech application expert to effectively implement speech applications on the GPU. It is an approach for crystallizing and transferring the often tacit knowledge of effective parallel programming techniques while allowing for flexible adaptation to various application usage scenarios.

The application framework for parallel programming includes an *application context description*, a *software architecture*, a *reference implementation*, and a set of *extension points* for flexible customization. We describe how a speech expert can use the application framework in a parallel application design flow as well as present two case studies that illustrate the flexibility of the framework to adapt to different usage scenarios. The case studies show two examples in extending the framework to an advanced audio-only speech recognition application and an audio-visual recognition application that enables lip-reading in high noise recognition environments. The adaptation to the latter scenario also demonstrates how the ASR application framework has enabled a Matlab/-Java programmer to effectively utilize a GPU to produce an implementation that achieves a 20x speedup in recognition throughput as compared to a sequential CPU implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Automatic Speech Recognition, Algorithms

Keywords Speech Recognition, Application Framework, GPU

1. Introduction

Automatic speech recognition (ASR) is emerging as a critical component in data analytics for a wealth of multimedia data [7]. Speech recognition technology allows multimedia contents to be transcribed from acoustic waveforms into word sequences (Figure 1). Commercial usage scenarios for ASR are appearing in both data centers and portable devices[1]. Being able to efficiently implement speech recognition applications on parallel platforms is increasingly important.

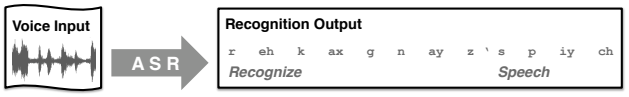

Figure 1. Automatic Speech Recognition (ASR) analyzes a waveform, detects phones, and infers word sequences

Underlying a wide range of ASR applications is a software architecture (shown in Figure 3) which extracts a sequence of features from the acoustic waveform and performs statistical inference on the sequence of features to output a word sequence. An efficient implementation of this software architecture can benefit a whole class of ASR-based applications.

Commercial availability of manycore computing platforms such as the NVIDIA GTX480 brings significant opportunities for ASR applications to perform an order of magnitude faster when efficiently implemented on GPUs as compared to traditional sequential implementations running on a CPU (as demonstrated in [36]). Such dramatic improvements in performance can allow researchers to analyze an order-of-magnitude more data, and allow companies to explore new services and capabilities or provide existing services more profitably.

However, in order to achieve good performance on the GPU, one must have a deep understanding of the architecture characteristics of the parallel implementation platform. With tens of processor cores integrated onto the same chip, the GPU devotes its transistor resources to maximize total instruction throughput[2].

With the GPUs, their dependence on *vector function units* requires carefully designed memory data layouts to effectively utilize the memory bandwidth resources. It has been shown that there can be a 2-10x performance difference between different data layout choices [6]. The GPUs' various *non-coherent caches* and *software-managed scratch-space memories*, as well as the associated memory bandwidth limitations make the mapping of application data working set to the hardware architecture resources of the GPU a form of art. Rearranging program execution to manage data work-

[1] iPhone applications such as Jibbigo are providing speech recognition on the client device as part of a traveler's speech-to-speech translator

[2] Compared with the 102 GigaFLOP/s of peak arithmetic throughput for the Intel i7 960 processor running at 3.2GHz, the NVIDIA GTX480 GPU provides more than 13x the instruction throughput (1392 GigaFLOP/s) running at less than half the frequency (1.45GHz).

ing set to fit in scratch space memory has produced 2-3x speedup for our key computation kernels. On top of that, the multiple *synchronization scopes* with their associated latencies and throughput characteristics and overloadable logic and arithmetic operations make efficient communication among concurrent tasks a challenging design space. By using hardware atomic operations with overloaded arithmetic operations for synchronization, it has been shown in [36] that one can construct software architectures that are 4.5x more efficient than using global barrier based synchronization.

Published results [6] and our experiences [-,-] indicate that failure to carefully design data layout, data placement, and synchronization processes can lead to over an order of magnitude reductions in application performance. Fully exploring each of these factors can be a significant undertaking. For example, {*omitted for blind review*} [-] took five weeks to re-factor and re-verify. While a few ambitious speech experts have parallelized the simpler modules in ASR with limited application level parallel scalability [12]. The many factors to consider put application development for parallel platforms outside of reach of an average application developer.

In order to develop efficient applications, a deep understanding of the application domain is also required. The fine-grained parallelism and fast on-chip cross-core synchronization capabilities of GPUs open up opportunities that often require flexible reorganizations of the application to take advantage of parallel resources. It was observed that transformations of the input data structures while preserving domain-specific invariants can improve the performance of an application by up to 42% [9]. Just being aware of the possible re-organization requires deep application expertise. Furthermore, applying such re-organizations often involves application-specific transformations that are functionally equivalent at the application level but would not produce bit-identical results. These factors make many application re-organizations beyond what an average parallel programmer can attempt.

Thus, we believe that both application domain expertise and parallel programming expertise are required to construct efficient parallel applications. This requirement *limits* the wide spread utilization and deployment of highly parallel microprocessors in academia and industry.

Observing and respecting the need for both application domain and parallel programming expertise for the development of efficient parallel applications, we propose that the few individuals and teams that have expertise in both areas should construct *application frameworks* as a means to aid more application domain experts to effectively develop parallel applications.

An **application framework** is a software environment built around an underlying software architecture (such as the one shown in Figure 3) in which user customizations may only be applied in harmony with the software architecture. The software environment includes an *application context*, a *software architecture*, a *reference implementation*, and a set of *extension points*. A software architecture is defined as a hierarchical composition of parallel programming patterns [20], which are solutions to recurring problems in parallel programming.

The application framework for parallel programming crystallizes in the reference implementation the parallel programming optimizations that are applicable to a class of applications based on their common software architecture. It also provides extension points to allow flexible customizations of the reference implementation with plugins targeting specific application usage scenarios.

In this paper, we describe the components of the application framework (Section 2), present a reference implementation of an ASR application framework (Section 2), and utilize its extension points to adapt to different usage scenarios (Section 4). In particular, we utilized the extension points to adapt the reference implementation to two usage scenarios: 1) an advanced audio-only model from SRI used for meeting recording transcription, and 2) an audio-visual speech recognition system that enables lip-reading to increase recognition accuracy in noisy conditions.

2. Application Frameworks

There are four main components in the application framework for parallel programming:

1. An *application context* is a description of the characteristics and requirements for the class of applications
2. A *software architecture* description is a set of concise documentations of the organization and structure of the class of applications described using the software parallel programming patterns [20]
3. A *reference implementation* is a fully functional, efficiently implemented, sample parallel design of the application
4. The *extension points* are a set of interfaces defined to summarize the interactions between the application framework and potential plug-in modules.

The *application context* exposes parallelization opportunities of an application independent of the implementation platform. For application domain experts, it provides the context with which they can understand the motivations of parallelization decisions made in the software architecture of an application framework. Section 3.1 describes the application context for an automatic speech recognition application.

The *software architecture* presents a hierarchical composition of parallel programming patterns that assists in navigating the reference implementation. For application domain experts, the software architecture description allows them to quickly gain an understanding of the opportunities and challenges in the implementation of an application. This helps the domain experts organize their efforts around the fundamental limitations and constraints of implementing the application on highly parallel microprocessors. Section 3.2 describes a software architecture for a speech recognition application in detail.

The *reference implementation* provides a concrete example of how each component in the application framework could be implemented, and how they can be integrated. It is also a proof of the capability of a computing platform for the class of applications the application framework is designed for. For application domain experts, it relieves the burden of constructing functionally-correct baseline implementations before introducing new features. With the sample implementation they can focus on the particular modules that must be designed to meet the needs of specific end-user products. A detailed example of a reference implementation for ASR is described in Section 3.3.

The *extension points* invite developers to customize specific modules using pre-determined interfaces. It allows customizations that would not jeopardize the execution latency or throughput sensitive features in an application framework. For an application domain expert, the extension points provide a flexible interface for implementing plugins while maintaining the efficiency of the final implementation. Examples of extension points for the speech recognition application framework are described in Section 3.4.

Section 5 discusses ways to utilize the application framework components in a parallel application design flow. Briefly, as shown later in Figure 9(b), the application context and software architecture assist an application developer with describing the application and its concurrency opportunities at hand. The sample implementation provides a reference design to help developers evaluate the performance potential on a computing platform. The extension points help abstract away the rest of the application and allow the developer to focus on extending the functionality and implementing new features for an application rather than writing parallel code from scratch.

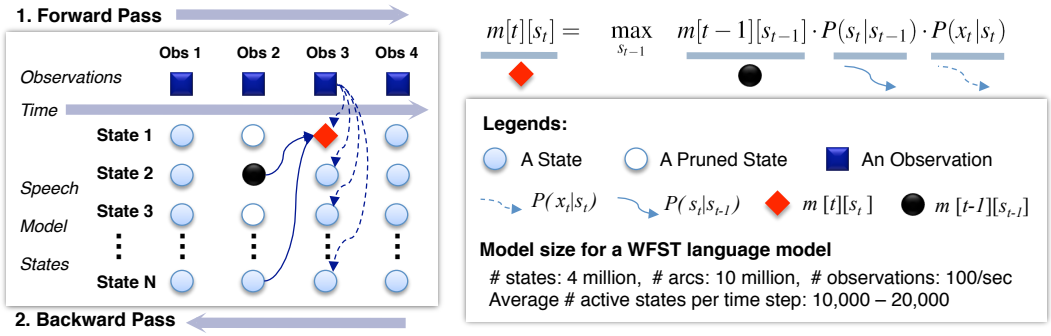

Figure 2. Application characteristics: inner working of the performance critical Viterbi forward and backward pass steps

Application framework for parallel programming is best suited for applications with a relatively mature software architecture, where extension points can be clearly defined [11]. From this perspective, speech recognition is an ideal candidate. For other nascent applications such as computer vision, a programming framework with a set of library of components and application specific data structures may be more appropriate for development assistance.

3. A Speech Recognition Application Framework

A speech recognition application framework helps speech recognition experts efficiently program manycore processors. It is presented as an exemplar of an application framework where we elaborate on the features, functions, and benefits of its components for speech recognition experts. This application framework helps resolve the challenges of implementing a batch mode speech recognition system. In a batch system, the feature extraction component can be trivially parallelized by distributing segments of the input acoustic to parallel computing resources and is not discussed here. This application framework focuses exclusively on the parallelization of the inference process.

3.1 Application Context

Application Characteristics: A Large Vocabulary Continuous Speech Recognition (LVCSR) application analyzes a human utterance from a sequence of input audio waveforms to interpret and distinguish the most likely words and sentences intended by the speaker (see Figure 1). The analysis involves iteratively comparing a sequence of features extracted from the input audio waveform (as observations) to a speech model that has been trained using powerful statistical learning techniques. To adapt to different languages, acoustic environments, domains of vocabularies, only the speech model needs to be replaced and the recognition process stays the same.

The process of recognizing speech is a type of temporal pattern recognition, which is a well-known application of the hidden Markov model (HMM) [30]. The states in the HMM for speech recognition are components of words in a vocabulary. They are hidden because they can only be observed through interpreting features in an acoustic waveform. The Viterbi algorithm [27], with a forward-pass and a backward-pass step, is often used to infer the most likely sequence of words given the observations from the acoustic waveform.

In the forward-pass, as shown in Figure 2, there are two main phases of the algorithm for performing inference. Phase 1, (shown in Figure 2 as dashed arrows between observations and states), evaluates the observation probability of the hidden state. It matches the input information to the available acoustic model elements and only takes into account the instantaneous likelihood of a feature matching acoustic model element. Phase 2, (shown in Figure 2 as the solid arrows between states of consecutive time steps), references the historic information about what are the most likely alternative interpretations of the utterance heard so far, and computes the likelihood of incorporating the current observation given the pronunciation and language models. The computation for each state s_t at time t (with the diamond shaped state as an example) records the state transition from the prior time step $t-1$ that produced the greatest probability $m[t][s_t]$.

Input and Outputs: In a standard speech recognition application, the acoustic inputs are features extracted from acoustic waveforms, typically at 10ms time steps. The number of features used varies among different information sources, languages and acoustic environments in the recognition scenario. For example when recognizing English with single microphone in meeting rooms, 39 features are a common value. Speech models used in this application framework contain information from acoustic models, pronunciation models, and language models. They are combined statically using weighted finite state transducer (WFST) techniques into a monolithic graph structure [24]. Depending on the language models, the speech model graphs often contain millions of states and tens of millions of arcs.

Working Set Size: When using the speech model during inference, it is observed that one can achieve good accuracy by tracking a small percentage of the total number of states representing the most likely alternative interpretations of the utterances. In our experiments, we found that tracking more than 1% of the most likely alternative interpretations provides diminishingly small improvements in accuracy while requiring a linear increase in execution time. Thus the working set of active states is kept below 1% of the total number of states in the speech model, which is on average 10,000 to 20,000 active states.

Concurrency: There are four main levels of concurrency in the inference process. We provide an application description and highlight the amount of concurrency available at each level. The opportunities and challenges posed by these levels of concurrency in an implementation will be explored in detail in Section 3.2.

1. Different speech utterances can be distributed to different machines for processing. A typical conversational utterance can be 5-30 seconds long, and a one-hour audio input can be distributed to hundreds of machines to process. Each inference process can take billions of instructions and last a few seconds.
2. For a set of utterances, if the forward and backward passes in the Viterbi algorithm are handled by dedicated computing resources such as different cores or different processors, the two passes can be pipelined. When utterance A has completed the forward-pass and proceed to compute the backward pass, utterance B can initiate its forward-pass.
3. In the forward-pass, if Phase 1 and Phase 2 are handled by dedicated computing resources, the two phases can be pipelined: i.e. one time step can be in Phase 2 while another time step performs Phase 1 on another computing resource.

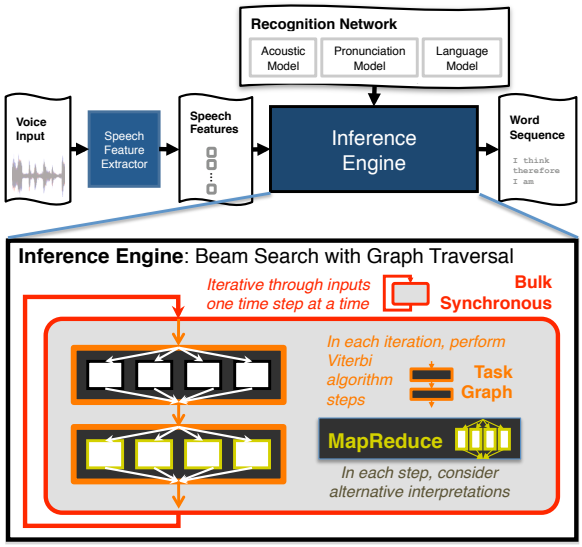

Figure 3. Software architecture of a large vocabulary continuous speech recognition application

4. Within each of the functions in Phase 1 and Phase 2 of the forward pass, there are thousands of observation probabilities and tens of thousands of alternative interpretation of the utterance to track. Each can be tracked independently with some amount of synchronization required after each function. Each unit of work is usually no larger than tens to hundreds of instructions.

Performance Constraints: The goal of automatic speech recognition is to transcribe a sequence of utterances as fast as possible with as high an accuracy as possible. For commercial applications, there is usually an accuracy threshold which makes the usage scenario realistic and practical. For example, for in-car command and control applications, one may tolerate a 5% command error rate in the interpretation of non-essential commands. For data analytics, where one searches for the keywords in a recorded telephone conversation, a 50% word error rate may be tolerable and still yield useful results.

3.2 Software Architecture

A software architecture is the organization of a software program expressed as hierarchical composition of patterns [20]. In the application framework, the software architecture expresses the composition of the reference implementation and reflects decisions made when mapping application concurrency to parallel hardware resources.

The hardware resources targeted in our ASR application framework is the GPU, which is an offload device with the CPU acting as as the host engine. Programs running on the GPU are written in CUDA [28], a language based on C++, with minor keyword extensions. CUDA programs are invoked from the host CPU code, and operands must be explicitly transferred between the CPU and the GPU. Recent GPUs such as the GTX480 contain 15 cores each with dual issue 16-wide SIMD units, with non-coherent caches and scratch space memories available in each core. In order to efficiently program the GPU, one must efficiently leverage the wide vector units, the GPU memory hierarchy, and the synchronization primitives within and between cores.

Concurrency Exploited: In our ASR application framework for highly parallel implementations on manycore processors, we have selected to exploit the fourth type of concurrency: the fine-grained parallelism within each of the functions in Phase 1 and Phase 2 of the forward pass of the inference algorithm. This choice may be the most complex to implement but provides the most efficient utilization of the manycore platform.

When mapping the application onto a manycore platform, the following thought experiments were performed to help eliminate choosing any of the first three types of concurrency:

1. Concurrency among speech utterances is the low hanging fruit. It can be exploited over multiple processors and is complementary to the more challenging fine-grained concurrency explored among cores on a chip and among vector lanes within a SIMD unit. Exploiting concurrency among speech utterances among cores and vector lanes, however, is not practical. With tens of cores sharing the same memory sub-system, the available memory capacity and memory bandwidth in a GPU cannot accommodate the working set sizes of tens of concurrent speech inference processes.
2. When different forward and backward passes are mapped onto different resources, referring to the pipe-and-filter computational parallel programing pattern, load balancing becomes the most significant factor in achieving efficient utilization. Backward pass performs less than 1% of the work done by the forward pass, thus the source of concurrency is not suitable for exploitation.
3. Depending on the model parameters used, Phase 1 and Phase 2 of the forward pass do similar amounts of work. However, referring to the pipe-and-filter computational parallel programming pattern, communication between the "filters" along the "pipes" may limit performance. In this case, if Phase 1 and Phase 2 are implemented on GPU and CPU, the amount of intermediate results that must be transferred between them can become a performance bottleneck. This is indeed observed in [12].

Architecture Representation: The software architecture for the ASR application framework is a hierarchical composition of parallel programming patterns, as shown in Figure 3. The top level of the inference engine can be associated with the Bulk Synchronous Structural Pattern [20], where each iteration is handling one input feature vector corresponding to one time step. The computation for the entire iteration is mapped onto the GPU device, such that the computation throughput will not be bottlenecked by intermediate results transferring between the CPU and the GPU. The work within each iteration can be associated with the Task Graph Structural Pattern [20], where different functions in two phases of execution take place. Each function in each of the two phases can be associated with the MapReduce Structural Pattern [20], where thousands of observation probabilities are computed in parallel, and tens of thousands of alternative interpretations of a speech utterance are tracked in parallel.

Challenges: While this software architecture maps well on to the many computing resources on the manycore GPU devices, it also presents significant challenges in global synchronizations between different algorithm steps. For example, in Phase 2 of the forward pass of the Viterbi algorithm, the inference process is based on parallel graph traversal, a known hard problem in parallel computing [22], especially in the context of speech recognition [19]. The parallel graph traversal operates on an *irregular subset* of 10,000 states of the speech model representing the most likely alternative interpretations of the utterance, and frequently updates *conflicting memory locations*. Correctly implementing such challenging tasks in parallel while optimizing for metrics such as memory bandwidth are often beyond what most application domain experts would like to undertake. Section 3.3 elaborates on how this challenge can be resolved by the application framework, which provides a reference implementation that encapsulates an efficient solution to these implementation challenges as suggested in [36].

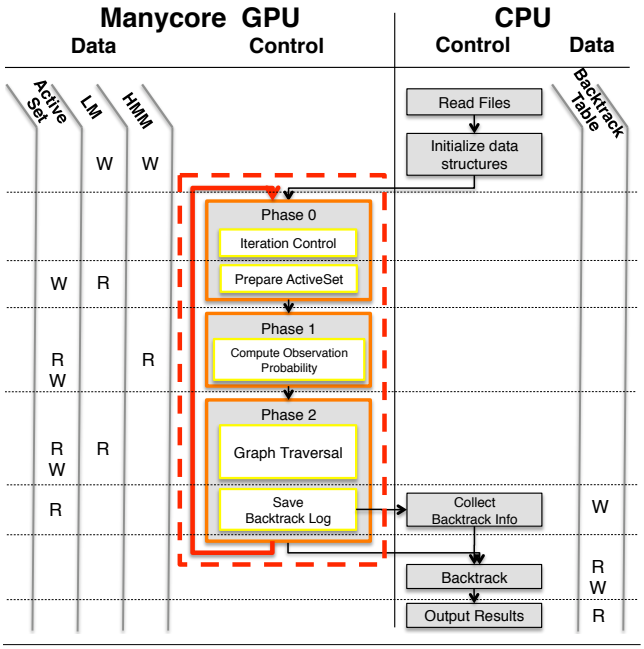

Figure 4. Summary of the data structure access and control flow of the inference engine on the manycore platform

3.3 Reference Implementation

Figure 4 illustrates the reference implementation of the speech inference engine. As specified in the software architecture, the forward pass inference loop is implemented on the GPU accelerator, and the backward pass is implemented on the CPU.

The main data structures can be classified in two types: read-only model data structures and read/write runtime data structures. Referring to Figure 4, the read-only model data structures include the acoustic model parameters (shown as "HMM" for hidden Markov model) and the WFST graph structure (shown as "LM" for language model). Read/write runtime data structures include storage for intermediate buffers for the inference process (shown as "Active Set" for the set of most likely alternative interpretations actively tracked at runtime) and backtrack information (shown as "Backtrack Table"), which is a log of the forward pass of the Viterbi algorithm used for the backward pass.

In terms of the program flow, the models and inputs are read from several files and placed into the memory on the GPU. The forward pass of the Viterbi algorithm is computed on the GPU and the logs are sent back to the CPU. When the forward pass is complete, the CPU performs the backtrack operations from the overall most likely path, and outputs the results to file.

On the GPU, the forward pass occurs in three phases. An additional Phase 0 is inserted to implement performance optimization techniques. Phase 1 evaluates the observation probability, and Phase 2 tracks the most likely interpretations of the input utterance over time.

In Phase 0, the *Iteration Control* step is introduced to manage the memory pools allocated in the ActiveSet data structure. In highly parallel shared memory architectures, any memory allocation and freeing action will introduce a point of serialization in shared memory management. The technique to avoid serialization is to allocate a memory pool at the beginning of a program, and carefully manage program behavior such that memory usage stays within the pre-allocated space. In speech recognition, we are tracking a set of active states which represents the set of most likely alternative interpretations in each timestep at run time. The decision of whether a state is active is based on a *pruning threshold*. States whose likelihood is greater than the threshold are tracked, others are pruned away in Phase 2. The challenge is that at the beginning of a time step in Phase 0, it is not clear what pruning threshold will allow the right amount of states to be active. The *Iteration Control* step makes a prediction based on the number of active states in the past few time steps, as well as the threshold of the past few time steps to predict what threshold to set in the current time step to keep the traversal process within the pre-allocated space.

During the *Prepare ActiveSet* step in Phase 0 we populate runtime data buffers to maximally regularize data accesses. The recognition network is irregular and the traversal through the network is guided by user input available only at runtime. To maximally utilize the memory bandwidth, the data required in each iteration is gathered into consecutive vectors acting as runtime data buffers, such that the algorithmic steps in the iteration are able to load and store results one cache line at a time. This maximizes the utilization of the available memory bandwidth.

The *Compute Observation Probability* step in Phase 1 is a compute intensive step. It involves matching the input waveform feature vector at the current time step to a large set of acoustic model parameters in the form of Gaussian mixture models (GMM). Depending on types of features used and the corresponding acoustic model used, the computation performed at this step may be different across different usage scenarios.

Phase 2 is a performance critical phase. There are two steps in this phase, the *Graph Traversal* step and the *Save Backtrack Log* step. The *Graph Traversal* step involves data-parallel graph traversals of the irregular WFST graph structure, where the working set is guided by inputs known only at runtime. The graph traversal routines execute in parallel on different cores and frequently have to update the same memory locations. Such frequent write conflicts between cores must be resolved efficiently. We employed four techniques to optimize the graph traversal for speech inference on GPUs, with each of them discussed in details in [-]: {*omitted for blind review*}

1. Constructing efficient dynamic vector data structures to handle irregular graph traversals
2. Implementing an efficient find-unique function to eliminate redundant work by leveraging the GPU global memory write-conict-resolution policy
3. Implementing lock-free accesses of a shared map leveraging advanced GPU atomic operations to enable conflict-free reduction
4. Using hybrid local/global atomic operations and local buffers for the construction of a global queue to avoid sequential bottlenecks in accessing global queue control variables

These techniques allow the graph traversal step with *irregular* data access patterns and *irregular* task synchronization patterns to be implemented on a data parallel platform. This way the application will not be bottlenecked by sharing of intermediate data between the CPU and the GPU in the inner loop of the speech inference algorithm in a hybrid GPU-CPU implementation. The implementation of these techniques also includes basic capabilities for *introspections* on the dynamically changing data working set size induced by input data. The framework is able to *automatically adapt* the routines' runtime parameters by adjusting their task distribution parameter (the thread block size) based on the amount of work available.

The *Save Backtrack Log* step in Phase 2 transfers traversal data from the GPU to the CPU. This step incurs a sequential overhead of 13% of the total execution time after parallelization.

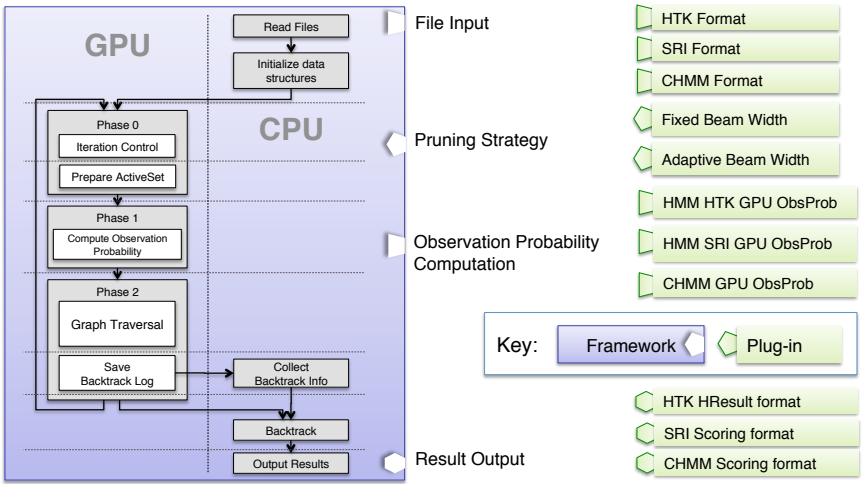

Figure 5. An application framework for automatic speech recognition with extension points

3.4 Extension Points

Based on the optimized reference implementation, we can construct extension points to allow the implementation to be flexibly adapted to a variety of usage scenarios. The application framework with its extension points and many plugins is presented in Figure 5. The extension points are illustrated as notches in the application framework. Their position is aligned with the functions in the program flow they affect. The shapes of extension points, or the notches, correspond to the associated plugins that match them.

An *extension point* is an interface for creating families of functions that extend the capability of the application framework for a specific application. It is implemented as an instance of an *Abstract Factory* creational object-oriented programming pattern as specified in [15]. This pattern involves an abstract class definition that specifies a set of methods with names, a certain number of parameters, and the types of the parameters.

There are three extension points implemented in the ASR application framework: Observation Probability Computation, Pruning Strategy, and Output File Format.

The **Observation Probability Computation extension point** is complex. It is used to manage not only the computation of the observation probability, but also the data structure of the acoustic model associated with the computation. As different acoustic models may use very different data structures, the data structure itself is not specified at the interface. This is also the reason why the File Input extension point is part of this interface. The interface is specified as follows:

```
1  string getName();
2  void   build_model(const Runopt * options);
3  void   save_model (const char *filename);
4  void   load_model (const Runopt * options);
5  void   free_model ();
6
7  void   observationProbComputation(
8          const int         frame,
9          const Utterance *in,
10         const int        *LabelFlagHash,
11               float      *LabelProbHash);
```

The interface on line 1 is a self identifying function for the plugin function. The *build_model* interface on Line 2 reads a text format input file into the internal data format. The filename to read is specified in the run options in the parameter. The *save_model* interface on Line 3 dumps the internal format to a binary file for fast loading in future runs. The *load_model* interface on Line 4 loads previously dumped internal format from a binary file. The *free_model* interface frees a model from memory.

The *observationProbComputation* interface on line 7 is where Phase 1 of the inference engine takes place. It has four parameters. The *frame* parameter specifies what frame, or time step, the Phase 1 calculations are targeting. The *in* parameter provides the input data. The *LabelFlagHash* parameter provides a vector of "0"s and non-"0"s as flags, where a "0" means the calculation should be skipped. The *LableProbHash* is a vector of output where for each non-"0" in the input flag vector, a valid observation probability is expected.

The **Pruning Strategy extension point** allows customization of the algorithm for specifying the pruning threshold. The interface is as follows:

```
1  string getName();
2  void   iteration_control(
3          const int         frame,
4                iControl *history,
5                float     pruneThreshold);
```

The interface on line 1 is a self identifying function for the plugin function. The *iteration_control* interface has three parameters. The *frame* parameter specifies what frame, or time step, the plugin is working on. The *history* parameter provides the history of past active state count and pruning threshold used. The *pruneThreshold* is the predicted pruning threshold to be used for the current frame.

The **Result Output extension point** allows the result of the inference process to be displayed in any format. The interface is:

```
1  string getName();
2  void   Result_Output(
3                FILE          *outFile,
4          const gpu_History   *hist,
5          const WordTable     *Wordtable,
6          const SegmentList   *segmentList);
```

The interface on line 1 is a self identifying function for the plugin function. The *Result_Output* interface has four parameters. The *outFile* parameter indicates the file pointer if the results are to be written to a file. The *hist* parameter provides the results of the backward pass in the Viterbi algorithm. The *Wordtable* parameter provides the word table to look up word IDs and print them out as words. The *segmentList* parameter provides the list of filenames being analyzed.

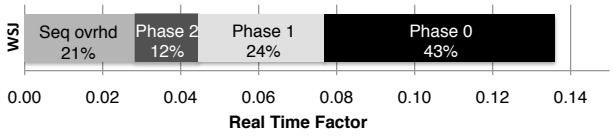

Figure 6. Recognition speed in real time factor demonstrated with the Wall Street Journal corpus at 8.0% WER

4. Sample Usages and Results

The reference implementation is provided with a small 5000 word model based on the Wall Street Journal corpus trained on broadcast news. The acoustic model was trained by HTK [35] with the speaker independent training data in the Wall Street Journal 1 corpus. The frontend uses 39 dimensional features that have 13 dimensional MFCC, delta and acceleration coefficients. The trained acoustic model consists of 3,000 16-mixture Gaussians. The WFST network is an $H \circ C \circ L \circ G$ model compiled and optimized offline with the dmake tool described in [1]. There are 3.9 million states and 11.4 million arcs in the WFST graph representing the language model. The test set consists of 330 sentences totaling 2,404 seconds from the Wall Street Journal test and evaluation set. The serial reference implementation using the LLM representation has a word error rate (WER[3]) of 8.0% and runs with a 0.64 real time factor.

For the implementation platform, we used the NVIDIA GTX480 (Fermi) GPU with a Intel Core i7 920 based host platform. GTX480 has 15 cores, each with dual issue 16-way vector arithmetic units running at 1.45GHz. Its processor architecture allows a theoretical maximum of two single-precision floating point operations (SP FLOP) per cycle, resulting in a maximum of 1.39 TeraFLOP of peak performance per second. For compilation, we used Visual Studio 2008 with nvcc 3.1.

To achieve the same 8.0% WER, the framework's reference implementation achieved 0.136 real time factor, or 4.7x speed up. The pruning threshold was set at 20,000 states, and the resulting run traversed an average of 20,062 states and 141,071 arcs per time step. From Figure 6, we observe that the execution time is dominated by Phase 0 for data gathering. This is necessary to align operands for Phase 2, the graph traversal phase. For the sequential overhead, 65% is used for transferring the backtrack log from the GPU to the CPU and 35% is for backtrack, file input and output on the CPU.

4.1 Adapting to the SRI Meeting Model

The SRI Meeting model is produced for the accurate automatic transcription of multi-party meetings. It aims to construct an interactive agent that provides online and offline assistance to meeting participants.

We used the speech models from the SRI CALO realtime meeting recognition system [34]. It is produced with an advanced frontend that uses 13 dimensional perceptual linear prediction (PLP) features with 1st, 2nd, and 3rd order differences, is vocal-tract-length-normalized and is projected to 39 dimensions using heteroscedastic linear discriminant analysis (HLDA). The acoustic model is trained on conversational telephone and meeting speech corpora, using the discriminative minimum-phone-error (MPE) criterion. The language model is trained on meeting transcripts, conversational telephone speech, and web and broadcast data [33]. The acoustic model includes 52K triphone states which are clustered into 2,613 mixtures of 128 Gaussian components. The acoustic model uses tied Gaussian Mixture Model (GMM) weights [33] and requires a slightly more complex Phase 1 module for observation probability computation.

To adapt the ASR application framework to the SRI Meeting Model, a new Observation Probability Computation plugin was developed to handle the tied Gaussian Mixture Model with a two step computation. This extension also involved a new component to read in SRI's model file format. A new Result Output plugin was developed to produce files for SRI's accuracy scoring script. the Pruning Strategy plugin was kept the same.

The test set consisted of excerpts from NIST conference meetings, taken from the "individual head-mounted microphone" condition of the 2007 NIST Rich Transcription evaluation. The segmented audio files total 44 minutes in length and comprise 10 speakers. For the experiment, we assumed that the feature extraction is performed offline so that the inference engine can directly access the feature files. The meeting recognition task is very challenging due to the spontaneous nature of speech. The ambiguities in the sentences require larger number of active states to keep track of alternative interpretations which leads to slower recognition speed.

Table 1 shows at various pruning thresholds the accuracy in WER, speed in real time factor and speedup on the GTX480 GPU (compared to an optimized sequential version implemented on the CPU). By leveraging the optimizations in the ASR framework, we were able to achieve up to 14.2x speedup GPU compared to a sequential version run on the CPU.

Table 1. Accuracy, word error rate (WER), for various beam sizes and corresponding decoding speed in real-time factor (RTF)

# of Active States	30k	20k	10k	3.5k
WER	41.6	41.8	42.2	44.5
Sequential RTF	4.36	3.17	2.29	1.20
Manycore RTF	0.37	0.22	0.18	0.13
Speedup	11.6x	14.2x	13.0x	9.10x

4.2 Using Coupled HMMs for Robust ASR

Robustness of speech recognition can be significantly improved by multi-stream and especially by audio-visual speech recognition (Figure 7). This is of interest for example for human-machine interaction in noisy reverberant environments, and for transcription of or search in multimedia data. The most robust implementations of audiovisual speech recognition often utilize coupled hidden Markov models (CHMMs), which allow for both input streams to be asynchronous to a certain degree. In contrast to conventional speech recognition, this increases the search space significantly, so current implementations of CHMM systems are often not real-time capable. Thus, for real-time constrained applications such as online transcription of VoIP communication or responsive multi-modal human-machine interaction, using current multiprocessor computing capability is vital.

4.2.1 Model Architecture of Coupled HMMs

Multistream and audiovisual speech recognition both use a number of streams of audio and/or video features in order to significantly increase robustness and performance ([26, 29]). Coupled hidden Markov models (CHMMs), with their tolerance for stream asynchronicities, can provide a flexible integration of these streams and have shown optimum performance in a direct comparison of alternative model structures in [25].

In CHMMs, both feature vector sequences are retained as separate streams. As generative models, CHMMs describe the probability of both feature streams jointly as a function of a set of two discrete, hidden state variables, which evolve analogously to the single state variable of a conventional HMM.

[3] The Word Error Rate is computed by summing the substitution, insertion and deletion errors of a recognition result after it is matched against a golden reference using the longest common subsequence matching algorithm.

Thus, CHMMs have a two-dimensional state $\mathbf{q}$ which, for audiovisual recognition, is composed of an audio and a video state, q_a and q_v, respectively, as shown in Fig. 7.

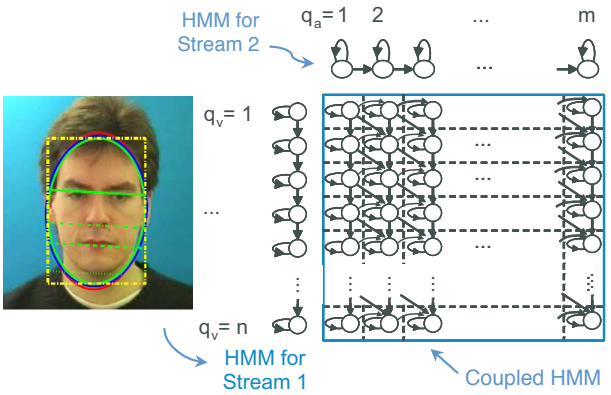

Figure 7. A coupled HMM consists of a matrix of interconnected states, which each correspond to the pairing of one audio- and one video-HMM-state, q_a and q_v, respectively.

Each possible sequence of states through the model represents one possible alignment with the sequence of observation vectors. To evaluate the likelihood of such an alignment, each state pairing is connected by a transition probability, and each state is associated with an observation probability distribution.

The transition probability and the observation probability can both be composed from the two marginal HMMs. Then, the coupled transition probability becomes

$$\begin{aligned} \mathrm{p}\Big(q_a(t+1) \quad &= j_a, q_v(t+1) = j_v | q_a(t) = i_a, q_v(t) = i_v\Big) \\ &= a_a(i_a, j_a) \cdot a_v(i_v, j_v) \end{aligned} \tag{1}$$

where $a_a(i_a, j_a)$ and $a_v(i_v, j_v)$ correspond to the transition probabilities of the two marginal HMMs, the audio-only and the video-only single-stream HMMs, respectively. This step of the computation, the so-called *propagation step*, is memory intensive due to the need for transition probability lookup in a large and irregularly structured network.

For the observation probability, both marginal HMMs could be composed similarly, to form a joint output probability by

$$\mathrm{p}(\mathbf{o}|\mathbf{i}) = b_a(o_a|i_a) \cdot b_v(o_v|i_v), \tag{2}$$

where $b_a(o_a|i_a)$ and $b_v(o_v|i_v)$ denote the output probability distributions for both single streams.

Such a formulation, however, does not take into account the different reliabilities of the two feature streams. Therefore, Eq. (2) is commonly modified by an additional stream weight γ as follows

$$\mathrm{p}(\mathbf{o}|\mathbf{i}) = b_a(o_a|i_a)^{\gamma} \cdot b_v(o_v|i_v)^{1-\gamma}. \tag{3}$$

Finally, computation of the marginal HMM state probabilities can be implemented in the same way as for a standard single HMM system, e.g. as an M-component Gaussian mixture model (GMM)

$$b(o|i) = \sum_{m=1}^{M} \gamma_m \mathcal{N}\left(o \mid \mu_{i,m}, \Sigma_{i,m}\right). \tag{4}$$

$\mathcal{N}(o|\mu, \Sigma)$ stands for a multivariate Gaussian distribution evaluated for the vector-valued observation o with mean μ and covariance matrix Σ. The covariance matrix may be either a full or a diagonal covariance matrix, where the latter implies that feature vector components are either independent or that their dependencies may be neglected.

The steps given by (4) and (3) will be referred to as the *likelihood computation* and the *likelihood combination* steps, respectively. These steps, especially the *computation*, have a much greater compute-to-memory-access ratio than the propagation step, due to the computational effort involved in GMM evaluation.

4.2.2 How was the framework used?

A coupled HMM can be compiled into a WFST that conforms to the required format in two steps:

- First, the 2-dimensional state index needs to be converted into one linear index for each of the involved word models. These linearized word models can then be stored in an applicable format, in this case, the OpenFST [2] input format.
- Secondly, the word models need to be composed into the sentence level WFST. This compilation, and a subsequent minimization, were carried out using OpenFST, which resulted in an overall network size of 3167 states and 12751 arcs for the GRID grammar [10].

Once the WFST network is available, the only relevant change with respect to a "regular" HMM is the observation probability computation according to Eq. (3). Therefore, the significant extension point for enabling CHMM-based audiovisual and multistream ASR was the observation probability computation step, which had to be adapted for coupled HMMs. For this purpose, CUDA kernels were implemented for Equations (4), the *likelihood computation*, and (3), the *likelihood combination* step.

The likelihood computation was optimized especially for the use with full covariance matrices, which can often result in substantial performance improvements; the relevant optimizations are shown in some detail in [-] . For the likelihood combination step, a simple kernel was designed that is parallelized over all those coupled states that are in the active set at the given time frame.

4.2.3 Performance results

The WFST decoder was used for multi-stream speech recognition in the following experiment. The two combined marginal HMMs were a 39-dimensional full-covariance Mel-frequency Cepstrum model and a 31-dimensional diagonal covariance RASTA-PLP model. The accuracy remained precisely the same for the C++ reference implementation and the GPU version of decoder, reaching 99.3% for the best-performing, speaker-independent model on clean data.

Fig. 8 shows the runtime per file, averaged over 47 utterances, where the decoder was running sequentially on a Intel Core i7 920 CPU or on a NVIDIA GTX480 GPU.

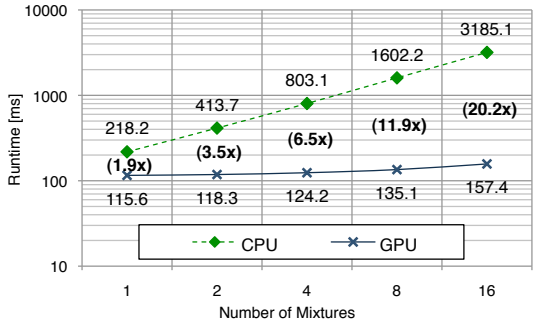

Figure 8. Runtime in ms per file of 3s length for $M = 1, 2, \ldots, 16$ mixture components used in Eq. (4). The speedup factor S is given in parentheses.

As can be seen, the speedup grows almost linearly with model complexity. For the GPU version, system overhead for calling the accelerator dominates the overall runtime for models with less than 4 mixture components. The likelihood computation starts dominating the runtime for more complex models with 8 and 16 mixture components.

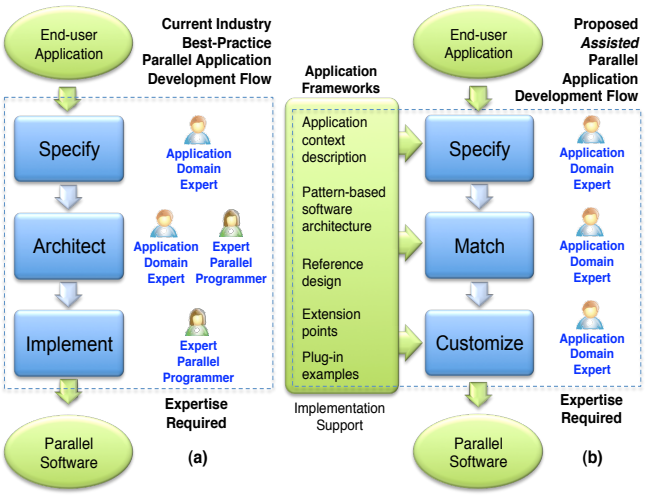

Figure 9. Parallel application development flow with and without assistance from application frameworks

5. Discussions

5.1 Parallel Programming Design Flow Implications

Our ASR application framework is developed to help speech application domain experts more effectively utilize highly parallel microprocessors. It allows them to extend the functions of an application implementation without having to implement the most complex and performance-sensitive parts of an ASR based application.

Without the application framework, the current industry best-practice for developing parallel software can be viewed as a three-step process involving *Specify*, *Architect* and *Implement* steps, as shown in Figure 9(a).

- The *Specify Step* highlights the application characteristics in terms of the type, size and requirements of the computation and any performance goals that must be met or would be nice to meet. It also exposes all the parallelization opportunities in the application, as well as the amount of parallelism each opportunity entails.
- The *Architect Step* defines the design space which is a set of alternative implementations of the solution that solves end-user's problem. The design space is associated with the parallelism opportunities exposed in the Specify step. In architect step, we also explore and prototype the potential performance bottlenecks. The end result is a set of data structures and their layouts, as well as application programing interfaces (APIs).
- The *Implement Step* implements the functions of the application by translating the high level descriptions of the application into software code, defines and deploys unit tests to verify functional correctness and performance requirements.

In this process, parallel programming experts are involved in the Architect and Implement step of every application development project. There are many application domains that can benefit from parallel implementations. If parallel programming experts have to be involved in developing every parallel application, the deployment of highly parallel microprocessors will be severely limited.

With the application framework, the process now has three steps of Specify, Match, and Customize, as shown in Figure 9(b).

- The *Specify Step* is the same as above, where one describes the application characteristics, exposes application concurrency (or parallelization opportunities), and defines invariants.
- The *Match Step* is where one selects the application framework to use, checks for potential performance bottlenecks, and gets an understanding of the data types and APIs of the extension points.
- The *Customize Step* is where one develops the plug-in modules to obtain new functions for the end application.

The parallel programming expert is still required to develop the application framework. Once the framework is available, the application domain expert can perform the *Specify*, *Match* and *Customize* steps independent of parallel programming experts, allowing more applications to benefit from the capabilities of highly parallel microprocessors.

5.2 Related Work

Application frameworks are developed as a tool for propagating proven software designs and implementations in order to reduce cost and improve quality of software. This concept has been discussed since the 1980s [31] and early application frameworks were built with macroprocessor and conditional compilation [5]. In the 1990s, with the rise of object-oriented programming (OOP) and the development of programming patterns based on OOP [15], there was a plethora of frameworks developed to define proven software designs in a variety of application domains. A large collection of such application frameworks in various domains has been presented in [13, 14].

As the computing platforms shift from sequential to parallel, the implementation challenges of software development are shifting towards exposing and exploiting application concurrency. We focus on these challenges by basing the application framework on the application's software architecture, which are based on a foundation of parallel programming patterns [20, 23]. This was inspired by the success of Ruby on Rails [4], which was built around customizations to the Model-View-Controller architecture [21].

Application frameworks for parallel programming have been proposed for scientific computing on large computing clusters. Examples include Cactus [17] for structured grids based applications, and CHARMS [18] for computational chemistry applications. Their development often involved multi-year efforts by a large team of developers. As multi-GPU workstations today often pack the compute throughput common in computing centers a few years ago, the demand of application frameworks is shifting towards desktop applications in a more diverse set of application domains. As such, we are already seeing frameworks emerging for GPU-based computing targeting machine learning and computer vision applications [3, 8]. By providing reference implementations and defining extension points on top of proven software designs, we hope to create light-weight application frameworks that are easy to construct and easy to use. We expect this type of application frameworks to become more important as more computer-literate application domain experts develop application for high performance workstations or even portable devices.

For speech recognition application, application frameworks have been proposed in [16, 32]. They focused on the software architecture to enable the interactive nature of speech recognition in dictation, command-and-control, or meeting transcription usage scenarios. They have not considered the parallel programming component of the application needs. Our framework focuses on exposing and exploiting the underlying application concurrency to enable high throughput recognition, which is complimentary to the existing work.

6. Conclusion

Our application framework for parallel programming is developed to help application domain experts effectively utilize highly parallel microprocessors. We have demonstrated that the automatic speech recognition (ASR) application framework has enabled a Matlab/-

Java programmer to achieve 20x speedup in her application by extending an audio-only speech recognition reference implementation to an audio-visual speech recognition application. This was achieved through the use of extension points to plug in new components in audio-video speech recognition that enables lip-reading in high noise environments to improve recognition accuracy.

The ASR application framework allowed the speech expert to leverage 12 months of prior research in software architecture, application design space, and implementation techniques of speech recognition on highly parallel computing platforms. With the proliferation of highly parallel computation from servers to work stations to laptops and portable devices, there will be increasing demand for adapting business and consumer applications to specific usage scenarios. We have shown that our application framework, with its application context description, application software architecture, reference implementation, and extension points, is an effective approach for transferring tacit knowledge about efficient, highly parallel software design for use by application domain experts. We believe application frameworks for parallel programming will be an important force for accelerating the adoption of highly parallel microprocessors.

References

[1] C. Allauzen, M. Mohri, M. Riley, and B. Roark. A generalized construction of integrated speech recognition transducers. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 761–764, 2004.

[2] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings CIAA 2007*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 2007.

[3] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. Gpucv: A gpu-accelerated framework for image processing and computer vision. *Advances in Visual Computing, Lecture Notes in Computer Science*, 5359:430–439, 2008.

[4] M. Bachle and P. Kirchberg. Ruby on rails. *IEEE Software*, 24:105–108, 2007. ISSN 0740-7459.

[5] D. S. Batory. Construction of file management systems from software components. Technical report, Austin, TX, USA, 1988.

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8.

[7] Business Wire. Impact 360 speech analytics software from verint witness actionable solutions enhances performance and customer experiences at telefnica o2 ireland. *Financial Tech Spotlight*, July 19, 2010. URL `http://financial.tmcnet.com/news/2010/07/19/4907090.htm`.

[8] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, April, 2008.

[9] J. Chong, E. Gonina, Y. Yi, and K. Keutzer. A fully data parallel wfst-based large vocabulary continuous speech recognition on a graphics processing unit. In *Proceeding of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1183–1186, September 2009.

[10] M. Cooke, J. Barker, S. Cunningham, and X. Shao. An audio-visual corpus for speech perception and automatic speech recognition. *J. Acoust. Soc. Am.*, 120(5):2421–2424, 2006.

[11] L. Deutsch. Design reuse and frameworks in the smalltalk-80 system. *Software Reusability, Volume II, Applications and Experiences, Biggerstaff and Perlis, editors. Addison-Wesley*, pages 57–71, 1989.

[12] P. R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Comput. Speech Lang.*, 23(4):510–526, 2009. ISSN 0885-2308.

[13] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley Computer Publishing, New York, NY, 1999. ISBN 0471248754.

[14] M. Fayad, D. Schmidt, and R. Johnson. *Implementing Application Frameworks: Object-Oriented Framework at Work*. Wiley Computer Publishing, New York, NY, 1999. ISBN 0471252018.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995. ISBN 0201633612.

[16] P. N. Garner, J. Dines, T. Hain, A. E. Hannani, M. Karafiat, D. Korchagin, M. Lincoln, V. Wan, and L. Zhang. Realtime asr from meetings. *Proceeding of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 2119–2122, September, 2009.

[17] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. *High Performance Computing for Computational Science - VECPAR 2002, Lecture Notes in Computer Science*, 2565:15–36, 2003. ISSN 0302-9743.

[18] E. Grinspun, P. Krysl, and P. Schröder. Charms: a simple framework for adaptive simulation. *ACM Trans. Graph.*, 21(3):281–290, 2002. ISSN 0730-0301.

[19] A. Janin. *Speech recognition on vector architectures*. PhD thesis, University of California, Berkeley, Berkeley, CA, 2004.

[20] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal, Addressing the Challenges of Tera-scale Computing*, 13(4):6–19, 2009.

[21] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:0118, 2001.

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

[23] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison Wesley, New York, NY, 2004. ISBN 0321228111.

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

[25] A. Nefian, L. Liang, X. Pi, X. Liu, and K. Murphy. Dynamic bayesian networks for audio-visual speech recognition. *EURASIP Journal on Applied Signal Processing*, 11:1274–1288, 2002.

[26] C. Neti, G. Potamianos, J. Luettin, I. Matthews, H. Glotin, D. Vergyri, J. Sison, A. Mashari, and J. Zhou. Audio-visual speech recognition. Technical report, Johns Hopkins University, CLSP, 2000.

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

[28] *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, March 2009. URL `http://www.nvidia.com/CUDA`. Version 2.2 beta.

[29] E. Petajan, B. Bischoff, and D. Bodoff. An improved automatic lipreading system to enhance speech recognition. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, 1988.

[30] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989. ISSN 0018-9219.

[31] C. C. W. Smithson. A lego approach to financial engineering: An introduction to forwards, futures, swaps and options. *Midland Corporate Financial Journal.*, pages 16–28, Winter, 1987.

[32] S. Srinivasan and J. Vergo. Object oriented reuse: experience in developing a framework for speech recognition applications. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 322–330, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8368-6.

[33] 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*, 4625(2):450–463, 2008.

[34] G. Tur, A. Stolcke, L. Voss, J. Dowding, B. Favre, R. Fernandez, M. Frampton, M. Frandsen, C. Frederickson, M. Graciarena,

D. Hakkani-Tr, D. Kintzing, K. Leveque, S. Mason, J. Niekrasz, S. Peters, M. Purver, K. Riedhammer, E. Shriberg, J. Tien, D. Vergyri, and F. Yang. The CALO meeting speech recognition and understanding system. In *Proc. IEEE Spoken Language Technology Workshop*, pages 69–72, 2008.

[35] 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*, 2:125–128, Apr. 1994.

[36] K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y. Chen, W. Sung, and K. Keutzer. Parallel scalability in speech recognition: Inference engine in large vocabulary continuous speech recognition. In *IEEE Signal Processing Magazine*, number 6, pages 124–135, November 2009.