Memory-efficient modeling and search techniques for hardware ASR decoders

Michael Price1,2, Anantha Chandrakasan2, James Glass1

1Computer Science and Artificial Intelligence Laboratory
2Microsystems Technology Laboratory
MIT, Cambridge, MA, USA
pricem@mit.edu, anantha@mtl.mit.edu, glass@mit.edu

Abstract

This paper gives an overview of acoustic modeling and search techniques for low-power embedded ASR decoders. Our design decisions prioritize memory bandwidth, which is the main driver in system power consumption. We evaluate three acoustic modeling approaches—Gaussian mixture model (GMM), subspace GMM (SGMM) and deep neural network (DNN)—and identify tradeoffs between memory bandwidth and recognition accuracy. We also present an HMM search scheme with WFST compression and caching, predictive beam width control, and a word lattice. Our results apply to embedded system implementations using microcontrollers, DSPs, FPGAs, or ASICs.

Index Terms: speech recognition, neural networks, fixed-point arithmetic, embedded systems

1. Introduction

Bringing ASR capabilities to small or portable devices requires attention to computational efficiency. Using a more efficient processor, whether a low-voltage DSP or a specialized ASIC, significantly reduces the energy used by arithmetic operations on the chip. There is some memory (i.e. caches) integrated on these chips, but it is not enough to store large ASR models—typically 100+ MB for an acoustic model and search graph. However, accessing an external memory such as DRAM or flash remains comparatively expensive [1]. Any attempt to reduce system power must reduce the amount of data exchanged with this external memory.

This paper explains the most helpful techniques we identified to perform speech recognition in these constrained environments. Acoustic model comparisons (Section 2) show that DNNs provide good accuracy even when the number and precision of parameters are limited. Search techniques that accommodate memory limitations are presented in Section 3.

2. Acoustic model

We are operating within the conventional HMM framework for ASR, with a WFST decoding graph [2]. The acoustic model has to evaluate the likelihood of input features \( y \) with respect to a set of distributions \( p(y | i) \), where \( i \) is the index of an acoustic state or senone. The accuracy of likelihoods directly impacts the search workload and word accuracy of a recognizer.

The general trend in ASR research has been to leverage growing compute resources and datasets to train increasingly elaborate models. Implementers of low-power ASR systems cannot blindly follow this trend. Instead of exclusively pursuing accuracy, we ask a slightly different question.

2.1. What is the most accurate acoustic model with 1 MB of parameters?

Until 2010, most ASR systems used Gaussian mixture models (GMMs) to specify each senone distribution. The community has since shifted to deep neural networks (DNNs), including convolutional and recurrent networks [3]. Povey et al. also developed a subspace GMM (SGMM) framework that improves accuracy over GMMs [4]. It was unclear whether the advantages of DNNs would be maintained in memory-limited implementations, so we characterized the tradeoff between WER and memory bandwidth for GMM, SGMM, and DNN models.

We used Kaldi [5] to train recognizers for several ASR tasks. To model the accuracy and bandwidth of our hardware-oriented algorithm changes, we constructed a separate ASR decoder in C++ and performed comparisons with a speaker-independent recognizer on the WSJ [6] dev93 task. The recognizer’s pruned trigram LM (bdz.gp in the Kaldi recipe) has a vocabulary of 145k words and 1.47M N-grams; the resulting WFSTs have 9.2–9.8M states and 22.2–23.6M arcs, depending on the number of acoustic states.

2.1.1. Gaussian mixture model (GMM)

The GMM models each distribution as a weighted sum of multivariate normal distributions over \( y \). With single-precision floating point values, a typical GMM acoustic model occupies 50 MB; naive evaluation at 100 fps would require 5 GB/s of memory bandwidth. This can be reduced by compressing the GMM, or by not evaluating the entire model on every frame.

One way to shrink a GMM is to use fixed-point mean and variance coefficients with reduced precision. With a nonlinear quantizer, acceptable accuracy can be obtained with as few as 5 bits for the mean and 3 bits for the variance [7]. Some hardware architectures can also evaluate the distributions for multiple frames in parallel [8], in exchange for longer latency: the first frame is not evaluated until features for a batch of frames have been loaded. The local memory required by a GMM is then dominated by the storage of likelihood results. Assuming a fixed memory size (in this case 1 Mb) and precision (32 bits), there is a tradeoff between the number of distributions and the number of frames that can be evaluated in parallel. Taking this into account, the performance of several hyperparameter combinations is shown in Figure 1. There is a steep tradeoff between memory bandwidth and recognition accuracy; this is generally true for all three modeling frameworks.

1Unless the feature space has been whitened, separate quantizers must be estimated for each feature dimension.
2.1.2. Subspace Gaussian mixture model (SGMM)

The SGMM is a collection of GMMs whose parameters vary in a low-rank vector space relative to a universal background model, or UBM [4]. Each distribution is modeled as follows:

\[
P(y_t | i) = \sum_{m=0}^{M-1} \sum_{c=0}^{C-1} k_{im} \, w_{cim} N(y_t; \mu_{cim}, \Sigma_{c})
\]

\[
\mu_{cim} = M_c v_{im}
\]

\[
w_{cim} = \exp \sum_{c'=0}^{C-1} \exp w_{c'} v_{im}
\]

where the vectors \(v_{im}\) describe the location of parameters for distribution \(i\) in a low-rank subspace, and \(M_c\) and \(w_c\) define the subspace for UBM component \(c\). The second level of mixing, performed by the weights \(k_{im}\), accounts for different ways that a single phonetic unit can be realized (i.e., depending on context). We exploit two opportunities for bandwidth reduction:

1. It is possible to store global parameters (i.e., UBM) locally and fetch a subset of the state-specific parameters for each evaluation.
2. “Gaussian selection” can be performed to prune the set of mixture components for each frame.

The quantization and parallelization techniques that we used for GMMs also apply to SGMMs. For a fair comparison, the 1 Mb local memory is split between likelihoods and intermediate results, reducing the number of frames that can be evaluated in parallel. Figure 2 shows the bandwidth/accuracy tradeoff for different model dimensions. All models used a 400-component UBM and selected the top 3 components per frame.

2.1.3. Neural network (NN)

A neural network is a sequence of layers that operate on real-valued vectors. Each layer \(l\) performs an affine transformation of its input followed by an element-wise nonlinearity:

\[
x_{l+1} = g(W_l x_l + b_l)
\]

We use a feed-forward, fully-connected NN that operates on stacked MFCC feature vectors. Most of the time and bandwidth is spent multiplying the layers’ weight matrices by their input vectors. To save bandwidth, the weights can be quantized and multiple frames can be evaluated in parallel, as with GMMs. We also train sparse matrices and store only the nonzero weights [9]; similar gains could be made with low-rank approximations [10]. Our model performs a piecewise Chebyshev polynomial approximation to the sigmoid function.

NN evaluation requires working memory proportional to the width of the largest layer. This memory is used most efficiently if all of the layers, including the output layer, have approximately the same number of outputs. This is in contrast to most software implementations where the output layer is large (2k–16k nodes) relative to the hidden layers (512–2k nodes).

Figure 3 shows results for a variety of NN sizes and sparsities. For this task, small NNs perform almost as well as large NNs. A quantized sparse NN with 512 nodes per layer requires about 4 MB/s of memory bandwidth to evaluate, with 10% relative WER degradation compared to the best model tested.

2.2. Model comparison and discussion

Figure 4 shows the Pareto optimal subset of results from each model framework. This allows us to see the tradeoff between WER and memory bandwidth, assuming that hyperparameters such as quantizer depth and model dimensions have been optimized. Only the acoustic model’s contribution to bandwidth is included.

We selected the DNN framework for our implementation because it offers the best accuracy, even when memory bandwidth is strictly limited. We hope that further improvements developed by the ASR research community can be ported to hardware with incremental changes in architecture.
3. Search

Our recognizers use Kaldi’s HCLG WFST construction. The search subsystem initializes a set of hypotheses for the HMM’s hidden state and propagates them forward in time. Our starting point is the architecture presented in [11], which moves hypotheses between two “active state lists” in local memory.

The forward pass of Viterbi search is divided into two phases (based on the input label of WFST arcs): the “non-$\epsilon$” phase and the “$\epsilon$” phase. To avoid recursion in the $\epsilon$ phase, we preprocess the WFST so that all paths of only $\epsilon$ arcs are bypassed by a single $\epsilon$ arc having the same total weight [12]. Some paths may traverse multiple arcs with non-$\epsilon$ output labels, so we create multi-word output symbols as necessary to bypass these paths; see Figure 5.

The following sections describe our efforts to implement a WFST search supporting large models on a processor with limited local memory and off-chip memory bandwidth.

3.1. WFST

Search initialization, forward passes, final weight updates, and backtracks all require access to a WFST. Arcs expanded in the forward pass account for significant memory bandwidth (10–100 MB/s) in a typical decoder. We apply both compression and caching to reduce WFST memory bandwidth. These work hand in hand: compression makes the cache look larger, relative to the information it is storing. Figure 6 shows the cache hit rate and WFST-related memory bandwidth for decoding a typical utterance with different beam widths.

Our WFST encoding is based on [7]. It preserves graph structure but quantizes weights (arc weights and state final weights) because high accuracy is unnecessary. Each state is stored as a state header following by a sequence of arcs. The data structure is illustrated in Figure 7.

Figure 5: WFST preprocessing with multi-word output symbols. Left: Adding $\epsilon$ arcs to bypass arbitrary length chains of only $\epsilon$ arcs [12]. Right: Compound word labels are created for arcs that bypass multiple non-$\epsilon$ output labels.

The following sections describe our efforts to implement a WFST search supporting large models on a processor with limited local memory and off-chip memory bandwidth.

3.2. Beam width control

ASR systems keep runtime in check by using a “beam search” variant of the Viterbi algorithm, pruning unlikely hypotheses during each forward pass [13]. The pruning can be relative (based on likelihood) or absolute (based on a maximum number of hypotheses). The desired number of hypotheses varies naturally over time; for example, there is a high branching factor near the beginning of a word. Figure 8 demonstrates this effect when decoding with relative pruning.

Hardware implementations need a combination of relative and absolute pruning to operate efficiently with a fixed local memory. We have a configurable hard limit for the number of active states; when this limit is reached during search, we reduce the beam width and prune the hypotheses in-place. This introduces search errors: it changes the beam width within a single frame, and the beam width is reduced in fixed (relatively large) increments to avoid repeating the process many times.

Performance can be improved over this crude approach by adding a “soft” cutoff inspired by Kaldi [14]. When the soft cutoff is exceeded, we preemptively reduce the beam width. This is done by storing a histogram of relative log-likelihoods, as in [15], and estimating (via linear interpolation of the cumulative sum) the beam width needed to hit a target state count. This is illustrated in Figure 9. We use the histogram to compute a
pruning threshold for the next frame, rather than retroactively pruning the current hypotheses. This feature is only engaged if the soft cutoff is lower than the hard cutoff.

The performance impact of these pruning mechanisms is visible in Figure 10. These results were obtained on the WSJ dev93 task (configured as in section 2) using a sparse 6-layer DNN acoustic model with 1024 nodes per hidden layer. Generally, lowering the cutoffs speeds up decoding at higher beam widths. Using a soft cutoff at 2k states (compared to the default of 5k) saves almost as much time as a hard cutoff, but doesn’t degrade accuracy.

The word lattice must be coordinated with the state list. We opted to store the WL state ID associated with every hypothesis in the state list. This requires more local memory, but the state list only includes hypotheses for 2 frames, whereas the word lattice covers the entire utterance.

Figure 12 shows the effect of the word lattice structure on external memory writes. Heavy workloads force more frequent word lattice snapshots, but there is still a 7–8x bandwidth savings because not all arcs have word labels.

4. Conclusions
This paper has laid the groundwork for memory-efficient ASR decoders with a bandwidth requirement on the order of 10 MB/s. We will provide more details about our hardware architecture and implementation in a future publication. Future work will also consider recent developments such as CTC-trained LSTM models [17] and split-VQ compression [18].

5. Acknowledgements
This work was funded by Quanta Computer via the Qmulus Project.
6. References


