Energy-Efficient Speaker Identification with Low-Precision Networks

by

Skanda Koppula

B.S., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author ....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by..
Anantha P. Chandrakasan
Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Thesis Supervisor

Certified by..
James R. Glass
Senior Research Scientist
Thesis Supervisor

Accepted by ....
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Energy-Efficient Speaker Identification with Low-Precision Networks

by

Skanda Koppula

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

In this thesis, I demonstrate an approach for text-independent speaker identification, targeting evaluation on low-cost, low-resource FPGAs. In the first half of this work, we contribute a set of of speaker ID models that build on prior existing small-model state-of-art, and reduce bytesize by >85%, with a ±3% accuracy change tolerance. We employ model quantization and pruning to achieve this size reduction. To the best of our knowledge, this is the first speaker identification model sized to fit in the on-chip memory of commodity FPGAs, allowing us to reduce power consumption. Our experiments allow us to illustrate the accuracy/memory-footprint trade-off for baseline and compressed speaker identification models. Second, I build an RTL design for efficient evaluation of a subset of our speaker ID models. In particular, I design, implement, and benchmark architectures for low-precision fixed point neural network evaluation and ternary network evaluation. Compared to a baseline full-precision network accelerator with the same timing constraints based on designs from prior work, our low-precision, sparsity-cognizant design decreases LUT/FF resource utilization by 27% and power consumption by 12% in simulation [Chen et al., 2017]. This work has applications to the growing number of speech systems run on consumer devices and data centers. A demonstration video and testing logs illustrating results are available at `https://skoppula.github.io/thesis.html`.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Joseph F. and Nancy P. Keithley Professor of Electrical Engineering

Thesis Supervisor: James R. Glass
Title: Senior Research Scientist

Acknowledgments

First and foremost, I would like to thank my two advisors: Professor Chandrakasan and Dr. Glass. Professor Chandrakasan, it has been an honor and incredible experience to work as your student these past three years. Your guiding vision and sterling mentorship these past three years have steered me through the incredible joys and frustrations of research during my SuperUROP and MEng. When I reflect back on my MIT experience, I see that joining your group has been one of the most formative experiences in my career. There are few faculty members I admire more for their vision, work ethic, excellence in their field, and as importantly, sincere support of your students. Dr. Glass, without your gentle guidance and genuine patience these past two years on our floor in SLS, this thesis would not have come to completion. Your level calmness and down-to-earth humor kept me sane during paper crunchtime, and research vision sparked my interest in speech and natural language.

Second, I thank my parents. You were my first teachers, and always there to catch me when I jumped for the next rock. Through the highs and the lows, you always stood beside me. Your love and support is an immeasurable debt.

It has been a pleasure to work with the absolutely brightest set of colleagues and peer mentors, in Ananthagroup and SLS. Chiraag, you have been my graduate student role model. Miaorong, Utsav, Preet, Mohamed, Avishek, Phillip, and the rest of Ananthagroup: you have been amazing friends and always there to help me learn the ropes in digital design. Yonatan, Sree, and other students at SLS: thanks for your guidance and patience during my spastic GPU workloads.

These thesis acknowledgements would not be complete without mentioning those who rounded out my time at MIT outside of research: Kevin Chan, Lisa Ho, Michael Janner, Eric Ponce, Tianye Chen, Irene Guzman, Olivia Zhao, Luke Kulik, and the friends at MIT EVT, B-Entry 2017, and Formula SAE. It's hard to imagine re-doing MIT without you in the story: thank you for the incredible friendships.

Thank you to Foxconn Technology Group for sponsoring this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consumer devices using speech interfaces are growing in number and complexity. Small devices such as wearables, personal assistants, robots, and phones boast extensive voice-controlled interfaces that transact identity-specific data and are linked to personal profiles. Increased reliance on such speech-driven devices presents a few challenges:

- Concerns about the authenticity of overheard speech commands. Relying on a hands-free interface like speech eliminates the possibility of secure, typed passwords, and anyone within speaking distance of the device has the ability to deliver potentially malicious commands, as has been demonstrated before BBC [2017].

- Concerns about the increased computational burden of the signal processing and machine learning required for accurate speech processing. The increased load manifests itself as higher energy costs, beefier processors, and larger memories.

This thesis investigates a solution to both these challenges, presenting an approach for energy-scalable speaker identification (SID). Speaker recognition is the task of identifying the speaker identity given an audio sample of the speaker's voice. This work focuses on optimizations to the speaker ID model structure and representation in memory, and then corresponding design of hardware to efficiently evaluate the proposed model. All work in the experiments and designs that follow are currently available at `https://github.com/skoppula/speaker-id-thesis` and `https://github.com/skoppula/speaker-auth-demo` for reference. A corresponding demonstration of the final system is available for viewing at `https://youtu.be/n1wdkIRCnrU`.

1.1 Outline of Work

In this first chapter, we introduce speaker identification and its relevance (Section 1.2), speaker ID modeling overview (Section 1.3.2), and feature pre-processing (Section 1.3.1). In the second chapter, we introduce our experiments building on prior work to reduce model size by one to two orders of magnitude. We detail benchmarks on candidate baseline architectures in Section 2.1 and procedures to excise normalization operations from the models in Section 2.2.

In the second half of chapter two, we explore further compression of SID models using new methods of quantization and methods used previously in computer vision: linear and logarithmic quantization (Section 2.3.1), DoReFa-quantization (Section 2.3.2), ternary weight quantization (Section 2.4), and another new contribution to SID, pruning and student-teacher based quantization (Section 2.5). With our optimized SID model, we then propose FPGA designs for evaluation of our speaker ID models (baseline in Section 3.2 and new design in Section 3.3). We discuss our final completed system setup in Section 3.5. Finally, we conclude and outline directions for future work in Sections 4.1 and 4.2.

1.2 Speaker Identification and its Applications

In this work, we focus on closed-set text-independent speaker identification, a variant of the speaker authentication task, in which the objective is to identify the correct speaker among a set of n-possible pre-enrolled speaker identities (as opposed to the more specific binary objective of recognizing a single specific authenticated identity). In this setup, each of the n speakers has a set of enrollment (training) utterances and test utterances. Closed-set speaker ID is also a good proxy for the measure of our model's ability to capture channel variability. Text-dependent SID refers to models trained to recognize persons speaking a specific keyword (e.g. 'OK Alexa'). In contrast, text-independent models are able to distinguish a speaker for any spoken input (or a range of n commands).

Speaker identification systems have real-world applications:

- Securing consumer voice assistants on mobile phones and Internet-of-Things devices with speech interfaces. Voice assistants receive speech commands for execution (e.g. "Send a text message 'Launch the Missiles' to Bob"), and without speaker ID, attackers can easily execute arbitrary commands. This vulnerability has been exploited repeatedly [Price, 2016, Feldman, 2016]. In a particularly egregious example reported by popular press, a crafted TV commercial was able to activate voice-controlled personal assistants and deliver a voice command to execute an online purchase [BBC, 2017].

- Automatic speaker labeling of media: for purposes of categorization and video labeling, SID is often applied on high-content media sites such as YouTube and SoundCloud to identify the particular speaker or artist present in the audio-video clip. Another popular application is automatic speaker labeling while transcribing video conferences.

- Remote biometric authentication: SID has been used for 2nd-factor verification of identity over phone (e.g. to modify to a caller's credit card account). Speaker identity is unique in that the biometric can be verified over the phone, without co-location.

1.3 Speaker Identification: Overview

In our complete SID system, there are two consecutive phases: feature extraction (Section 1.3.1) and model evaluation (Section 1.3.2). An overalls system block diagram detailing processing steps and data flow of the system is shown in Figure 1-1

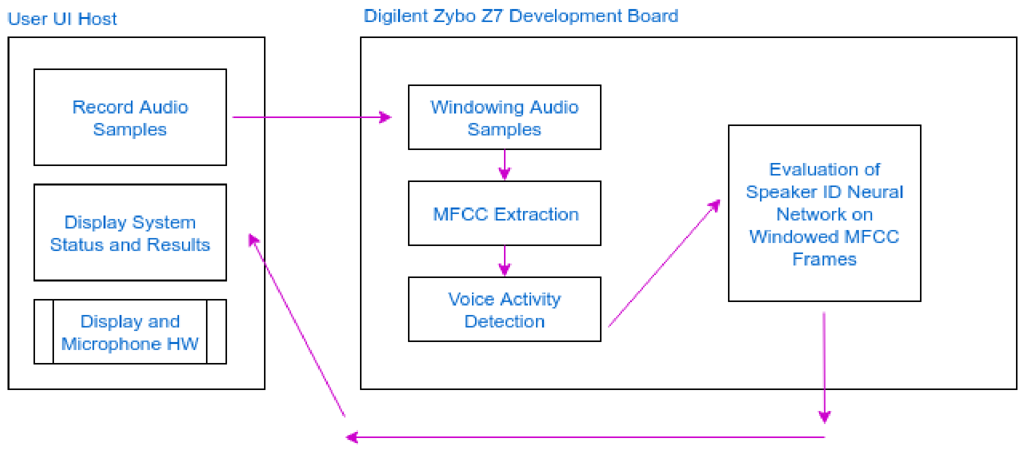

User UI Host
Record Audio Samples
Display System Status and Results
Display and Microphone HW
Digilent Zybo Z7 Development Board
Windowing Audio Samples
MFCC Extraction
Voice Activity Detection
Evaluation of Speaker ID Neural Network on Windowed MFCC Frames

Figure 1-1: Procedure and Data Flow in the Speaker Identification System.

1.3.1 Feature Extraction: MFCC-generation and VAD

As is common in most speech systems, we apply a sequence of signal processing front-end steps to extract what is approximately the envelope of the short-time power spectrum of the input speech sample. We calculate a characterization of this envelope, called Mel-Frequency Cepstral Coefficients (MFCCs), using the following standard steps Prahallad [2011]:

1. We cut the signal into overlapping millisecond-length frames,
2. For each frame, we estimate the power spectrum and apply a Mel filterbank,[1]
3. We integrate to find the energy in each of the filters in the Mel filterbank,
4. We calculate the logarithm of each filterbank energy,
5. We calculate the Discrete Cosine Transform (DCT) of the log energies, and use DCT coefficients as a representation of the frame's power spectrum.

We find that performing this audio pre-processing adds sub-second latency to the system, while significantly reducing the learning burden on the neural network. Preliminary experiments in training networks to operate on raw 8 KHz audio samples

[1]The Mel filterbank translates the actual measured frequency to a human perceived frequency in units of 'mel', given by the formula $m = 2595\ \log_{10}(1 + \frac{f}{700})$.

from the RSR2015 corpus did not yield networks that converged. This audio preprocessing reduces our network size and complexity. In contrast, neural networks designed to operate on raw audio samples are recurrent, very deep, and on the order of hundreds of megabytes for various speech tasks [Chan et al., 2016].

In this work, we use 25 ms frames with a 10 ms inter-frame step size. We use the first 20 DCT coefficients, so each MFCC frame is 20-dimensional. The power spectrum is calculated using a 512-point Discrete Fourier Transform (DFT), retaining the first 257 coefficients that we multiply with our 26 Mel filters. These values were used to match prior work for fair comparison [Povey, 2017]. The Kaldi signal processing toolkit was used for extraction of these features. In prior work, the first and second order directions of change for the MFCCs are appended to the feature vector, but remove this step in our final software and FPGA implementations. Our experiments we found a less than 1% improvement in speaker identification accuracy, at the cost of doubling or tripling the input size (and hence first layer network footprint). We discuss implementation of MFCC extraction on our target the Zync-7000 FPGA SoC in the final chapter of this thesis.

Speaker identification models work on the assumption that input is a human voice recording: the speaker identity classification given an input of a dog bark or rustling paper is understandably not interpretable. For this reason, prior to being fed into the speaker identification pipeline, all audio frames are filtered through a Voice Activity Detector (VAD). In this work, we use a simple mean-normalized power-based VAD to filter frames for training of our SID model and during testing time in our final working system [Ekapol Chuangsuwanich and James Glass, 2011].

In particular, we feed calculate the mean value m across all 20-dimensional MFCC frames in an utterance. We then set a fixed signal energy threshold ($t = 5.5$) and mean-scaling factor ($s = 0.5$), and filter audio frames that have mean value less than $sm + t$. This simple non-adaptive VAD filter is both used in practice and inexpensive to compute.

More recent work explores use of a Deep Neural Network-based VAD filter as the front-end of their ASR hardware [Prahallad, 2011, Tashev and Mirsamadi, 2016]. In the last chapter of this thesis, we explore training and deployment of a DNN-based VAD front-end by re-using the fixed-point network accelerator that we implement on the Zybo SoC. Due to time constraints, we did not re-train our SID models on DNN-based VAD-filtered utterances, which could offer system level accuracy improvements.

1.3.2 Model Representation and Training

In this work, we target a model representation that will fit within the fabric of our commodity FPGA, a Xilinx Zybo Zync-7000 FPGA. This imposes a constraint on our model size roughly 0.75 MB, a strong driving factor in our choice of model. This constraint has the added advantage of avoiding costly off-chip DRAM access by fitting within Zybo's BRAM.

Traditionally, state-of-art SID systems use a low-dimensional representation of Gaussian Mixture Models (GMMs) called *i-vectors*. In this setup, a speaker model is formed by shifting a *Universal Background Model* ('UBM', a GMM modeling thousands of speakers) to fit the speaker's data. A common choice is a 2048-mixture GMM of 60 dimensions, yielding a $2048 \times 60 = 122880$ dimensional representation of the speaker (vector of Gaussian means) [Povey et al., 2008, Reynolds, 2015]. This shifting is done using a few iterations of the Expectation-Maximization algorithm. The supervector is transformed into a low-dimensional intermediate representation called the *i-vector* which captures the differences between the UBM and a speaker-specific model[2]. Finally, a cosine distance threshold is commonly used to compare and test a stored i-vector with a test utterance's i-vector[3]. Most prior attempts at embedded speaker identification have used some variant of this GMM-based backend (see prior work, Section 1.5).

In our memory constrained setting, however, we find that a i-vector/GMM-based approach is not particularly suitable. Because of the large supervector dimensionality

[2]We leave a complete description of i-vector training, using sufficient statistics, to Shum [2011] for the interested reader.

[3]An alternative approach to train i-vectors is to use sufficient statistics of neural network posteriors instead of GMM statistics. This has been shown to result in modest accuracy gains. We leave the details to Lei et al. [2014].

(92160) and i-vector dimensionality (200), the i-vector extraction transformation matrix alone requires on the upwards of 70 MB of storage; our target FPGA development board (Section 3.1) has roughly 0.6MB of on-chip memory. The exploding memory requirements was confirmed on models we trained using the default recipes in Kaldi, available in the linked Github code repository `ie`. Even reducing the precision from 32-bit float to 8-bit (at an 8% cost in recognition accuracy), the model size drops to 18 MB, too large to fit in our FPGA accelerator's BRAM blocks.

Instead, we turn to recent work in the last two years that has investigated use of deep neural networks (DNNs) for speaker identification. Our motivation for pursuing this approach is two-fold: (1) significant recent work in computer vision has demonstrated the ability of these networks to compress with insignificant loss of accuracy Iandola et al. [2016], Han et al. [2015] and (2) unlike Expectation-Maximization and i-vector based-inference, the core operations in neural network inference are extremely simple (multiply-and-accumulates and rectification), simplifying FPGA design. In this work, we explore using variants of DNNs such as Fully-Connected Networks, Convolutional Networks, and Locally-Connected Networks to improve the accuracy-size tradeoff. We eschew use of recurrent networks for two reasons: (1) as the time-dependence within the network increases, overall system latency increases and (2) sophisticated RNNs like GRUs and LSTMs generally have 2-4x the number of parameters with negligible benefit to accuracy [Richardson et al., 2015].

1.4 Datasets and Toolchains

We use the RSR2015 corpus for our experiments [Larcher et al., 2014]. This corpus employs 300 speakers (157 male, 143 female) with 657 utterances per speaker divided across in 9 recording sessions. The average length of each recording is approximately 2 seconds. There are 73 unique phrase texts used for each speaker's 657 utterances. Four different types of recording settings (and channel noise levels) are used during creation of the corpus. Though the RSR corpus is primarily used for text-dependent experiments, the corpus has over 255 unique spoken text prompts. Thus, we use the

RSR corpus because it resembles our target scenario the strongest: a small home speech device recognizing a finite set of commands for control.

We occasionally use the SRE10 corpora set for additional validation of our model and quantization methods [Greenberg et al., 2010]. This set has 7062 utterances divided across 446 speakers. The average length of an SRE10 utterance is 117 seconds. For both corpora, we use a randomly selected 70%/15%/15% training/validation/testing dataset split across the entire set of corpora utterances. We ensured that each dataset split contains utterances from all of the dataset's speakers. Our training, validation, and testing splits of both corpora can be found in the `data_splits` folder in our Github code repository.

We use the speech toolkit Kaldi Audio and the deep learning framework Tensorflow/Tensorpack for SID model training and accuracy evaluation [Povey et al., 2011, Abadi et al., 2015, Wu et al., 2016]. We used Xilinx Vivado and Bluespec Verilog for RTL development and synthesis.

1.5 Prior Work

A number of works have studied the application of end-to-end neural networks for SID. For text-dependent SID, Variani et al. [2014], Snyder et al. [2016], and Zhang et al. [2017] have demonstrated network architectures that achieve comparable accuracy to i-vector systems for noise-free, short-utterances. These networks, however, at minimum exceed 20MB in size, and are not suitable for direct adaptation to our target setting. Some of these models have been deployed to consumer electronics, in the Google Home personal assistant and Windows phones. As per documentation, these devices do not do inference locally; rather, they rely on cloud services for model storage and forward inference.

Efficient hardware evaluation of neural networks has been the focus of various research efforts in the past two years. One avenue of research in the field has focused on building application-specific integrated circuits to efficiently evaluate large pre-trained neural networks [Chen et al., 2017, Han et al., 2016, Price et al., 2017, 2015, 2016b,a, Liu et al., 2015]. Another avenue of research has focused on algorithmic-based model optimization: reducing model complexity by introducing sparsity and Huffman-encoding parameters [Han et al., 2015], student-teacher models to distill knowledge to smaller models [Lu et al., 2017], bit-width reduction [Zhu et al., 2016, Rastegari et al., 2016, Zhou et al., 2016], and kernel-shaping techniques [Howard et al., 2017, Lebedev et al., 2014]. These works have primarily been used for only computer vision tasks, with the exception of [Park and Sung, 2016], which is a limited demonstration of FPGA-based phoneme recognition.

Additionally, a handful of non-DNN based works have studied SID in low-resource FPGA and embedded settings. Sarkar and Saha [2010] implements a very basic SID system in FPGA. The core recognition algorithm uses the cosine distance between downsampled MFCC feature vectors. Accuracy of the system was not benchmarked. Ramos-Lara et al. [2009] replaces the cosine distance classifier with an Support Vector Machine, again reporting no accuracy benchmarks. Both works implement their own DSP front-end. Ehkan et al. [2011], Kan et al. [2010] implement a lossy GMM/UBM SID system on FPGA.

1.6 Key Contributions of this Work

In particular, the key contributions of this work are as follows:

- We demonstrate three methods of compression via model quantization (training-free linear bucketing, trained ternary quantization, and 0-1 fixed point clipping). We elucidate the accuracy vs. model size trade-off of each method using speaker identification models from prior work. Within a $\pm 3\%$ accuracy tolerance, we reduce the SID baseline model size by 85%.

- We demonstrate DNN-based speaker identification on an low-cost, low-resource FPGA. In particular, we implement and benchmark architectures for (1) low-precision fixed point neural network evaluation, (2) ternary network evaluation, and (3) exploiting sparsity in our models. Compared to our baseline network full-precision accelerator, with the same timing constraints, our low-precision, sparsity-cognizant design decreases LUT/FF usage by 27% and estimated power usage by 12% in simulation. This improvement is compared to a baseline design inspired by Chen et al. [2017]; as part of future work, we intend to compare against additional designs from other groups.

Chapter 2

Creating Robust and Low-Overhead Speaker ID Models

2.1 Baselines

At the core of an effective speaker ID system is the underlying neural network classifier. For our baseline models, we build on prior work that tackles DNN-based end-to-end speaker ID. In the upcoming sections, we demonstrate application of various compression and distillation techniques. In particular, we choose our model topology and sizing to improve upon the work of four papers:

1. Snyder et al. [2016] and Bhattacharya et al. [2016] use standard **fully-connected networks** for text-dependent and text-independent speaker verification. Their chosen topologies use four 256-size FC layers (small) and four 504-size FC layers (large), respectively.

2. Variani et al. [2014] proposes use of a **maxout** network for text-dependent speaker identification. The 'maxout' architecture replaces the nonlinearity (typically a ReLU) with a maximum over outputs from replicated, parallel fully-connected layers. Following the specifications in Variani et al. [2014], we use networks with four maxout layers of width 1000 and replication factor 4 (for the large network) and replication factor 2 (for the small network).

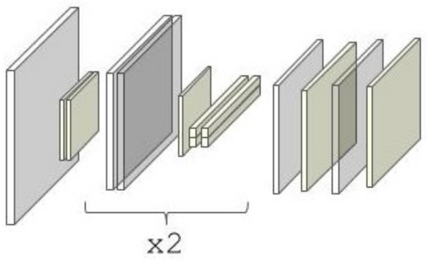


Figure 2-1: Depth-Separable Convolution (DSC) network adapted from Howard et al. [2017]. We test usage of this kernel in our SID task. In a DSC topology, the 3D convolution kernel is decomposed into a flat 2D kernel, and a subsequent 1D kernel to reduce parameter and multiplication count.

3. Chen et al. [2015] proposes an improvement on Variani et al. [2014], using of **locally-connected** (LCN) and **convolutional** network (CNN) topologies for text-dependent speaker identification. The differences in these topologies are described in Figure 2-2. Matching Chen et al. [2015], the LCN network uses a locally-connected first layer with 10×10 kernels, followed by three 256-size fully-connected layers. Similarly, the CNN network uses a 5×5 convolutional kernel, followed by three 256-size fully-connected layers.

4. Torfi et al. [2017] and Howard et al. [2017] use variants of convolutional networks for speaker ID and vision applications. In particular, these papers use **depth-seperable convolution** (Figure 2-1) kernels, followed by three 256-size fully-connected layers.

All models use full-precision 32-bit floating point weights and activations.

In our full-precision baseline models, we insert batch normalization in between layers to improve training and decaying L2 regularization on all the network parameters. We excise these layers in subsequent full-precision and quantized models, in order to reduce parameter count and simplify computation.

In order to track multiplications, custom layers were implemented in Python to exactly track the computations added to the Tensorflow network graph in every layer. Multiplications, additions, and parameter counts were tracked in our custom batch normalization, fully/locally connected, and (depth-separable) convolutional layers.

Our following two formulas capture the key metrics of interest, aggregating them

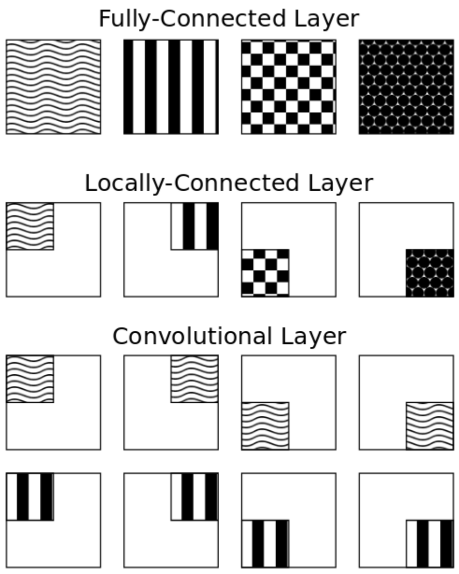


Figure 2-2: Differences in network topologies used in three of our baseline SID models. In the fully-connected network (FCN), each layer's kernel (pattern) covers the entire area of input. In a locally-connected (LC) topology, each kernel is applied to a local patch of the input. In a convolutional network (CNN), each kernel slides across the entire area of input [Chen et al., 2015].

into a simple-to-interpret score:

$$\texttt{score} = \log_{10}(m \times e \times b) \tag{2.1}$$

where m is the number of multiplications required for one evaluation of the network, e is the validation error of the model, and b is the bytesize of the model. A lower score is indication of a better model.

Experimental Procedure: As previously mentioned, we use the RSR2015 70% training set to train the model, the 15% development set to tune last layer dropout (ranging between 0 to 15% dropout, in 5% increments), and our 15% testing set for our final assessment of error. We train using simple cross-entropy loss, comparing the network output with a one-hot vector representing which of 255 speakers the utterance belongs to (the closed-set speaker ID task). The final error that we report in the tables that follow is the utterance-level closet-set speaker ID error on the testing dataset. To calculate this, we take the majority vote among the speaker classifications

Model	Error (%)	Mults	Params	Bytesize	Score
Fully Connected, Small	9.39	519K	521K	2.08MB	11.0079
Fully Connected, Large	2.97	1433K	1435K	5.74MB	11.3882
Convolutional	12.574	265K	260K	1.04MB	10.5411
Locally-Connected	11.32	287K	288K	1.15MB	10.5748
Maxout, Large	26.32	2133K	2136K	8.54MB	12.6812
Maxout, Small	36.65	1821K	1826K	7.3MB	12.6881
Depth-Separable Conv., Small	29.56	360K	332K	1.32MB	11.151
Depth-Separable Conv., Large	11.14	1233K	1221K	4.88MB	11.827

Table 2.1: Speaker Identification Error of Baseline Models on RSR2015, using batch norm and L2 regularization. The cells highlighted in blue correspond to the best baselines (lowest metric score).

on the sliding window input frames. We repeat this procedure for all models that we train, in the following experiments.

We implement all networks and training code in Tensorflow, using the Kaldi toolkit for signal pre-processing. We use the Adam stochastic optimization algorithm [Kingma and Ba, 2014] and train for a fixed 30 epochs (ensuring manually that the loss function reaches a reasonable plateau). We use a decaying learning rate schedule across the epochs starting at 0.1 and reaching below 0.001 by the end of training.

To train this set of models, we use batch normalization and regularization. The lower scores in Tables A.2 and A.1 (Appendix A) show that both these modifications are required for optimal performance.

Table 2.1 describes the performance of the baseline models on the RSR2015 corpus. Surprisingly, the simplest set of models – the fully-connected, convolutions, and locally-connected architectures – report the lowest score, boosted by their comparatively low error. Table A.3 in the Appendix details the performance on the SRE dataset, which confirms the same performance/efficiency trends. Accordingly, in the subsequent sections, we focus on optimizing these three architectures.

Model	Error (%)	Mults	Params	Size	Score	Baseline
Fully Connected, Small	9.39	517K	519K	2.07MB	11.0045	11.0079
Fully Connected, Large	2.97	1428K	1431K	5.72MB	11.3858	11.3882
Convolutional	12.60	263K	258K	1.03MB	10.537	10.5411
Locally-Connected	11.41	285K	286K	1.14MB	10.572	10.5748

Table 2.2: Speaker Identification Error of Baseline Models on RSR2015, after normalization excision. Marginal increases in error are offset by model size reductions. All four models have lower metric scores: an average, very small 0.002 improvement from baseline.

2.2 Normalization Excision

The first optimization applied on these three models cuts out the normalization during inference. Batch normalization, while useful for training, adds a number of parameters to every single layer (proportional to the width of the layer). Instead, we can achieve the same effect by folding normalization parameters into the next kernel parameters. In particular, input activations x go through a normalization and fully-connected layer:

$$\begin{aligned} x_{norm} &= \gamma \frac{x-\mu}{\sqrt{\sigma}} + \beta \\ y &= f(W x_{\text{norm}} + b) \end{aligned} \tag{2.2}$$

where β, γ, σ, μ are learned batch normalization constants and W, b are the regular layer parameters.

This is equivalent to performing the operation with a new, pre-computed W' and b':

$$\begin{aligned} W' = \gamma \frac{W}{\sqrt{\sigma}} \qquad b' &= b + W\big(\beta - \frac{\gamma\mu}{\sqrt{\sigma}}\big) \\ y &= f(W'x + b') \end{aligned} \tag{2.3}$$

Table 2.2 summarizes the model performance post-normalization folding. We see marginal increases in error offset by reductions in the number of multiplications and paremeters. This results in a decrease in score for three of the four models.

Surprisingly, we encountered significant reductions in accuracy if we attempted to re-train the networks after folding the batch normalization parameters. We hypothesize that this is because normalization folding induces a ‘fragility’ in the network parameters: an isolated local minimum, around which small perturbations in the weights cause large increases in loss.

2.3 Reducing Precision

To take these speaker ID models to hardware, we need fixed precision. In the following two sections, we discuss our work quantizing parameters to reduce the model size and simplify the underlying hardware required for evaluation.

2.3.1 Training-Free Linear Quantization

One of the most common approaches for quantization of floating point numbers to a fixed-precision is uniform linear quantization [Hubara et al., 2016, Jacob et al., 2017, Xichen, 2017, Lin et al., 2016]. This form of quantization has the advantage that it is straightforward to implement, and generally does not necessitate post-quantization training. In linear quantization, the maximum and minimum values W_{max}, W_{min} are recorded for each kernel W. To compensate for erroneous overflow/underflow outliers, usually W_{max}, W_{min} are selected to be the 99.5th percentile largest and smallest value. The number of bits required to distinguish numbers in this range is given by $b = \lceil \log_2 |W_{max} - W_{min}| \rceil$.

If we have k-bit quantization, the quantized values are integers in the range $[-2^{k-1}, 2^{k-1}]$. This means that to scale numbers in the original range to the new range, we must divide parameters by scaling factor $\delta = 2^{k-b-1}$. Scaling factor δ can be intuitively thought of as the range in the floating point world corresponding to one quantization interval. This means we have our complete k-bit quantization conversion for kernel W, stochastically rounding the scaled value:

$$W_q^{ij} = \lfloor W^{ij} / 2^{k-b-1} \rceil$$

Activations are quantized in a similar fashion. Over the course of floating point training, the minimum and maximum activations are tracked so that the corresponding scaling factors can be applied during evaluation on hardware. Note that this division by a constant power of two (δ) during real-time evaluation is straightfoward to implement in digital logic.

Note that the scaling factor for the biases $\delta_{b,l}$ of layer l are chosen carefully based on the scaling factors for the weights $\delta_{w,l}$ and activations $\delta_{a,l}$ of that layer: $\delta_{b,l} = \delta_{w,l} \cdot \delta_{a,l}$. This is to preserve correctness: the original output of a layer would be $Wx + b$. The new quantized output would be $\frac{1}{\delta_{b,l}}(Wx + b)$, with the scaling factors accumulated and factored out by the end of the network. The final quantized network outputs a constant scaling of the original outputs. In a classification task, like in speaker ID, multiplying each of the classes by a positive scaling factor does not effect the classification outcome.

Figure 2-3 shows an example of the distribution of parameters before and after quantizing the weight kernels in our large fully connected network to 6-bit values.

Prior work has used uniform linear scaling to reasonable success on vision tasks using very-large, very-deep vision networks (e.g. AlexNet, ResNet-17, or VGG-16), where there is significant redundancy in the information encoded in the parameters [Jacob et al., 2017].

However, when implementing this quantization on our speaker ID networks, interestingly, we obtain significantly worse results than baseline ($15\times$ increase in classification error).

To understand why, we examined the behavior quantizing each of the layers individually. The results are shown in Figures 2-4 and 2-5. Despite deviating from floating point weight by $< 10^{-4}$ (less than 0.1% of the original value), the quantized versions of the first linear (`linear0/W`) induced a 20.6% increase in classification error. On the flip side, layer `linear2/b` deviated from the floating point weight by a magnitude of 0.3 (4% of the original value), yet increased error by 1.43% when quantized. We find that some layers are significantly more sensitive to quantization (particularly, the first few layers). This finding is also true of our other models as

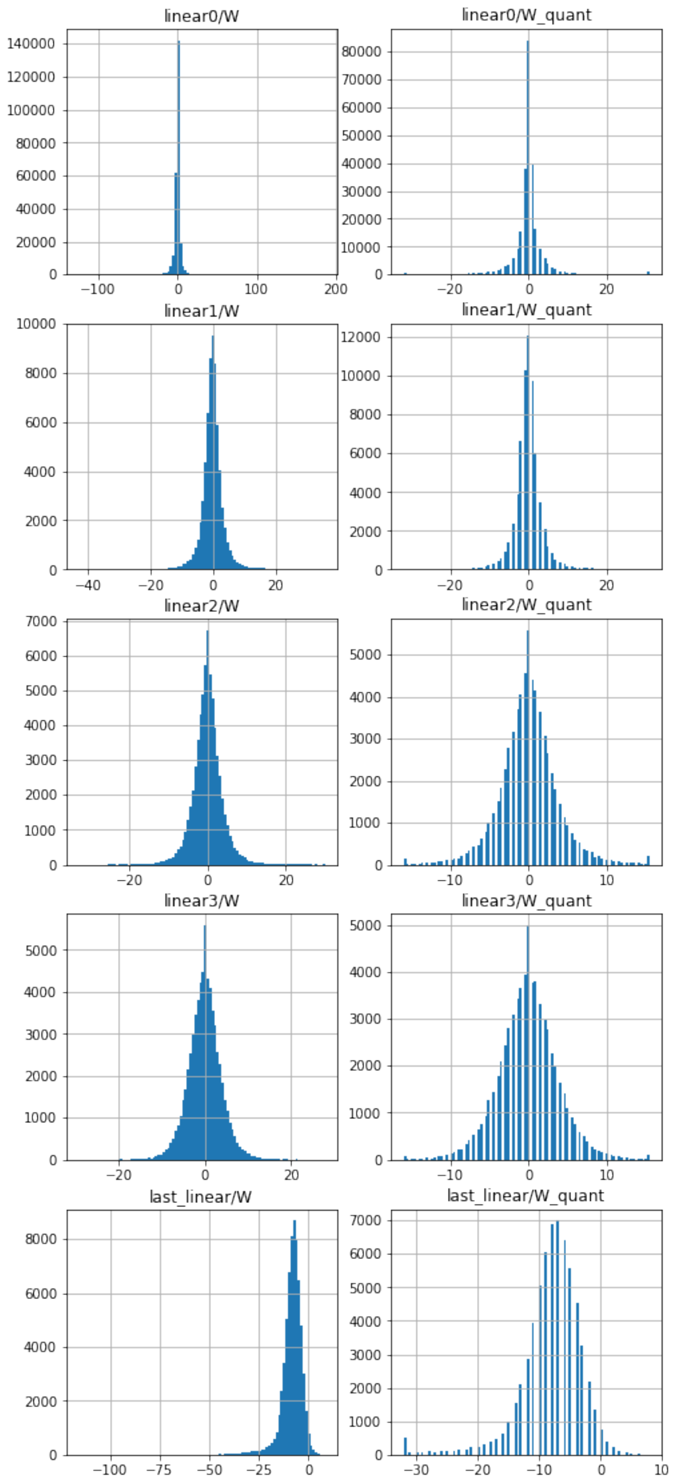


Figure 2-3: Visualization of the parameter distribution of the large fully-connected SID network before (left) and after (right) applying 6-bit min-max linear quantization.

Model	Error (%)	Mults	Params	Bytesize	Score	Baseline
Fully Connected, Large	7.43	1428K	1431K	2.86MB	11.4827	11.3882
Convolutional	20.40	263K	258K	517KB	10.4451	10.5411

Table 2.3: Speaker Identification Error of 16-bit Min-Max Linearly Weight-Quantized Models on RSR2015. While model size is up to 30% less, scores are not significantly better (and for the FCN, worse) because of the increase in error.

well; similar results for our convolutional models can be viewed in Appendix Figures A-1 and A-2. These experiments were performed with full 32-bit activations.

An interesting trend we noticed when scaling the bias for quantization is explosion of the bias magnitude. As noted before, the bias is divided by the nearest power of two approximation of $\delta_{b,l} = \delta_{w,l} * \delta_{a,l}$. If $\delta_{w,l}$ and $\delta_{a,l}$ are extremely small, dividing by $\delta_{b,l}$ will increase the bias significantly, reaching magnitudes on the order of 10^{12}. Such large values are susceptible to overflow in bit-constrained settings. A concrete example of this for the large fully-connected network is shown in Table A.4.

Under this quantization scheme, we found that we had to leave the first layer in 32-bit fixed-point during inference, 24 bits for the integer portion, and 8 bits for the decimal. With this modification, we obtain Table 2.3 showing results with 16-bit parameter quantization using training-free min-max linear quantization.

Notice that the scores are slightly worse than the original non-quantized models, because of the drop in accuracy. With even lower precision, the increases in error offset the improvements because of smaller models. Thus, we seek alternative methods of quantization that could better preserve the small error of the original models.

2.3.2 [0,1] Fixed Point Quantization

To resolve the exploding bias problem previously highlighted, we switch our quantization to limit the range of weights, activations, and biases to $[-1, 1]$ represented as a k-bit fixed point number.

In particular, during training, an underlying floating point weight kernel x is quantized to matrix x_{quant} on the fly using the transformation:

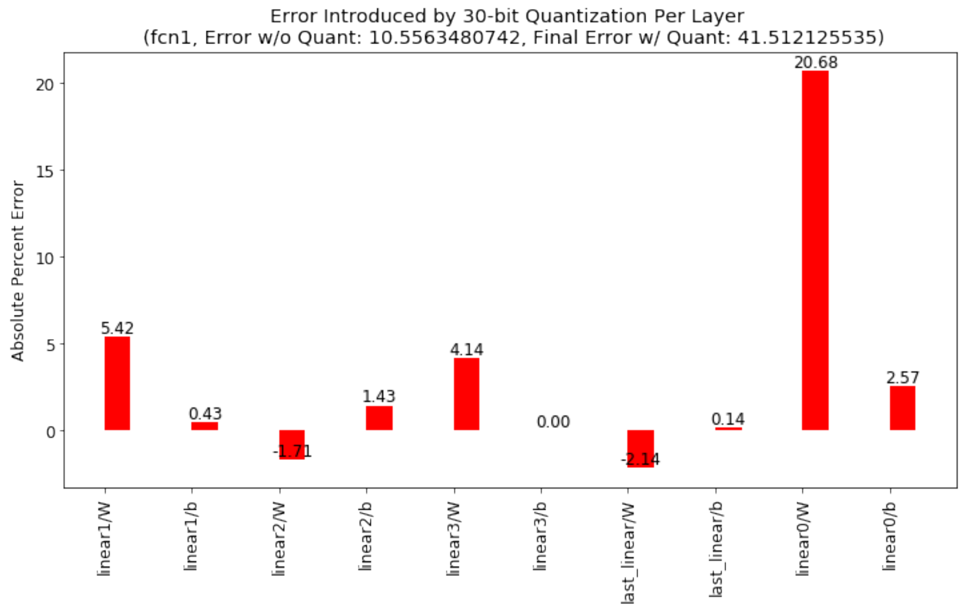


Figure 2-4: Increase in error after quantizing each layer in the large fully-connected SID network (30-bit min-max linear quantization).

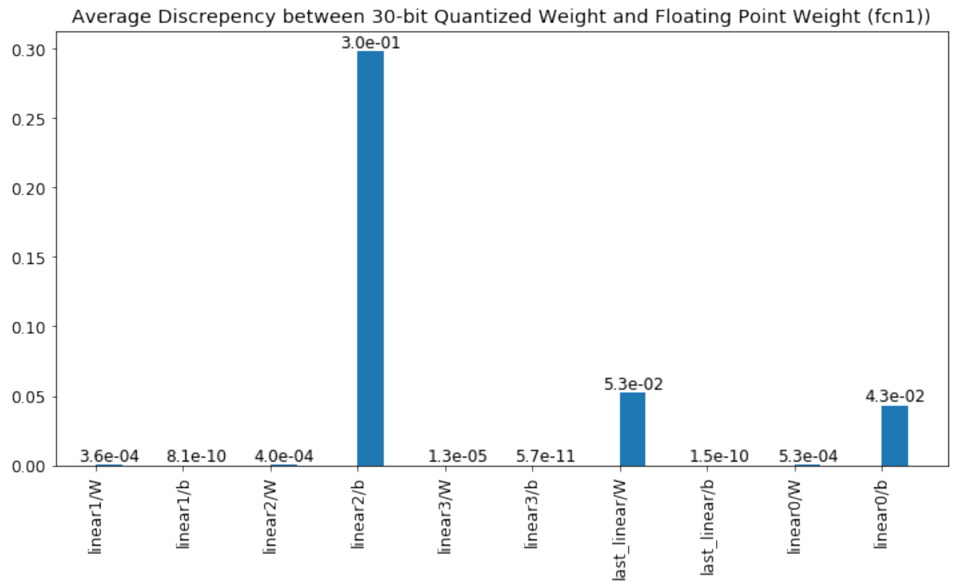


Figure 2-5: Maximum magnitude of discrepency between the quantized and non-quantized parameters of each layer in the large fully-connected SID network (30-bit min-max linear quantization).

$$\begin{aligned} x_{[0,1]} &= \tfrac{x}{2\max|x|} + \tfrac{1}{2} \\ x_{quant} &= 2\ \texttt{quant}(x_{[0,1]}) - 1 \end{aligned} \tag{2.4}$$

$x_{[0,1]}$ is x squashed into the the range $[0,1]$. The `quant` function takes in a real value $x \in [0,1]$ and outputs a k-bit fixed point $x \in [0,1]$:

$$\texttt{quant}(x) = \frac{1}{2^k}\ \texttt{round}(2^k \times x)$$

Importantly, note two things: (1) during training, the floating point weights are stored and updated during backpropogation but only the quantized versions are used to compute loss (2) during inference, we store only x_{quant}, removing the need to compute expensive max functions on the fly.

Quantized activations a (after the layer computation $a = Wx + b$) are limited to the range $[0,1]$ through use of a rectifiying non-linearity clipped to a maximum 1.

This quantization scheme is based work from Zhou et al. [2016], with two key modifications: (1) we remove the original scheme's use of `tanh` during quantization, which we found squashes weights that grow large, losing information capacity and (2) we replace the squashing non-linearity `tanh` with the clipped ReLU, which we found to improve performance and increase transferability to hardware.

Results

While these networks take roughly 1.2x longer to train, we find significantly better results than with the previous linear quantization scheme:

1. **Quantizing Middle Layer Parameters**: When we exclude quantization of the first and last layers, and keep activations in full-precision we obtain the accuracy results in Figure $2-6$. We are able to reduce bit precision to 8-bits on all models without increase in error. With a less than 5% increase in error, we are also able to reduce the bitwidth to 4-bits.

2. **Quantizing All Parameters**: Figure $2-7$ illustrates the error when we quantize all parameters in a network, keeping activations in full-precision. We are

able to reduce bit precision to 8-bits on all models (and 4-bits on the large FCN) without an increase in error. The large FCN and CNN appear to be the most robust models to bitwidth decrease.

3. **Quantizing All Parameters and Intermediate Activations**: Figure $2-7$ illustrates the error when we quantize all parameters in a network, and quantize activations to the same bit-width. Nearly all models diverged to 0% accuracy, with the exception of the large FCN, which experience a roughly **5-10x increase in error**. The difficulty of quantizing activations was also discussed in the previous Section 2.3.1. This observation, that networks are more sensitive quantizing intermediate activations as compared to quantizing weights is confirmed by recent literature [Zhou et al., 2016].

   The memory for storing intermediate activations is less than 10% of that required for parameters, and so it is less critical to quantize input features and activations to fulfill memory constraints. Although quantized activations would allow for less complex arithmetic units, we find that the accuracy degradations are too substantial to implement quantized activation networks. As such, in the models we deploy in our demonstration (Section 3.5), we keep activations in 32-bit full precision.

In order to achieve the error rates shown, the quantized parameters were initialized at the beginning of training to the values of the baseline models. The activation-quantized models (Figure 2-8 were initialized using the model from 2-7 and the baseline models (the former yielded only three diverging networks, while the latter caused all models, including the large FCN to diverge).

Table 2.4 show our final quantization results. All models improve their score, because of the 4-8x decrease in model size. In some models, the corresponding increase in error is less than 1%. As expected, 4-bit models have marginally worse classification error than their corresponding 8-bit models. Nearly all the models listed here fit in the maximum BRAM allocation constraints on the target economy FPGA (Xilinx Zync-7000). Our models constraining activation demonstrated unacceptably high

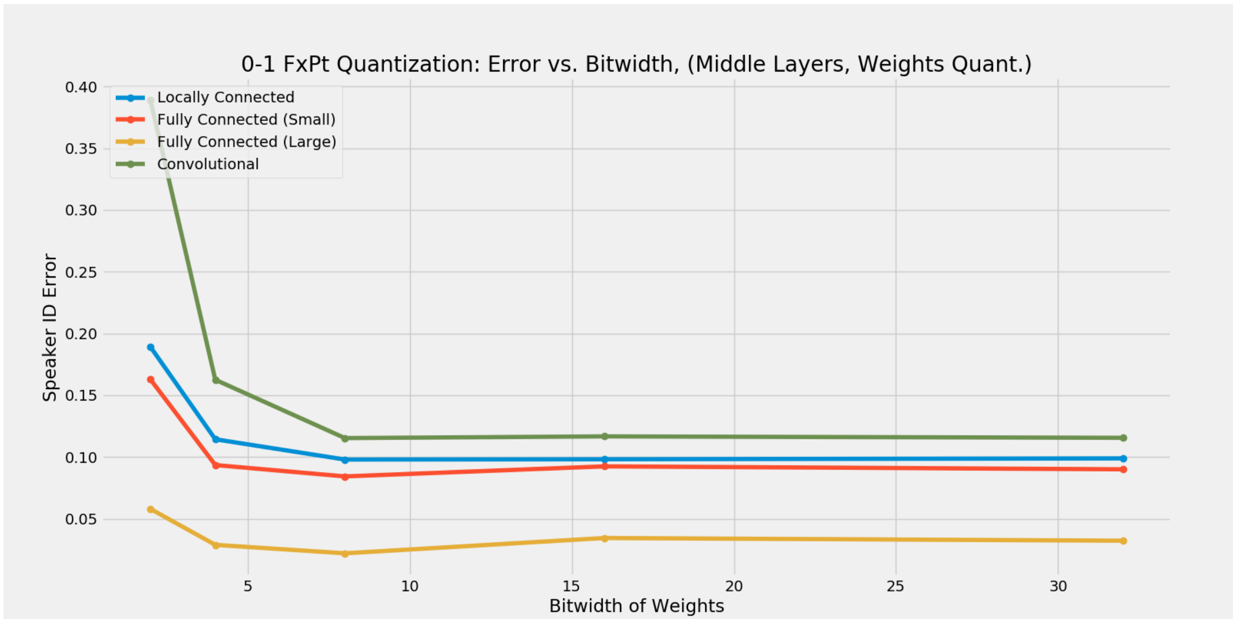


Figure 2-6: Speaker ID Error vs. Parameter Bitwidth (using the [0,1] fixed-point quantization). Parameters for the first and last layers are not quantized. Activations are 32-bit floating point (see discussion in Section 2.3.2).

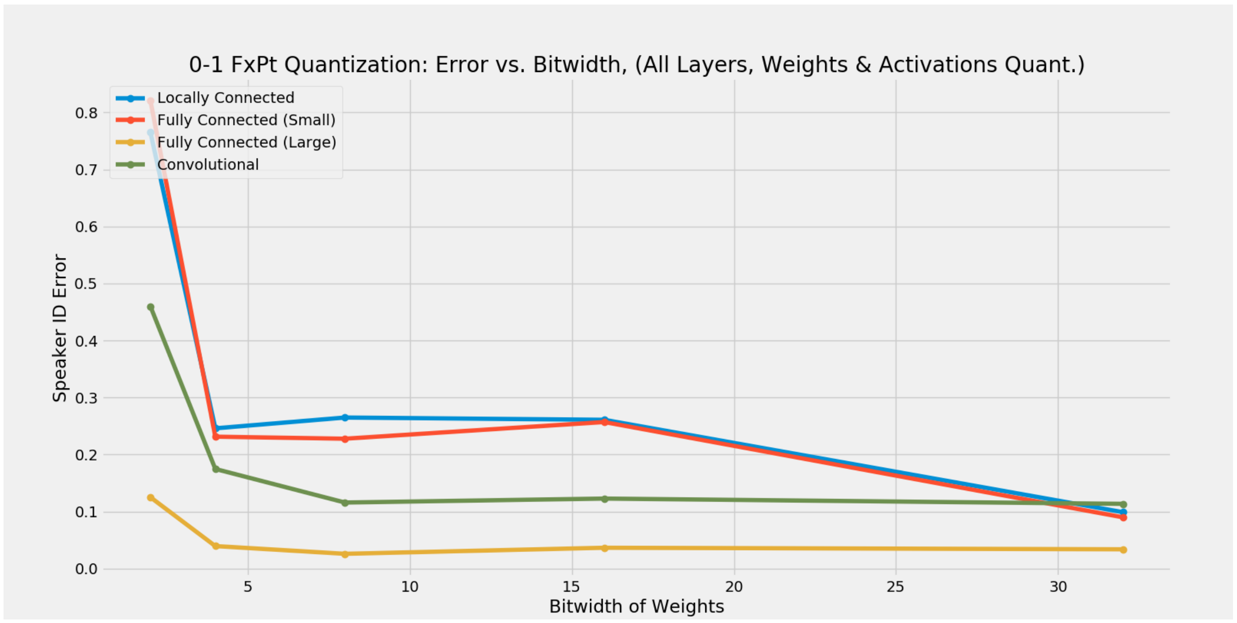


Figure 2-7: Speaker ID Error vs. Parameter Bitwidth (using the [0,1] fixed-point quantization). All parameters in the network are quantized. Activations are in this set of experiments were set to 32-bit floating point (see discussion in 2.3.2).
Section

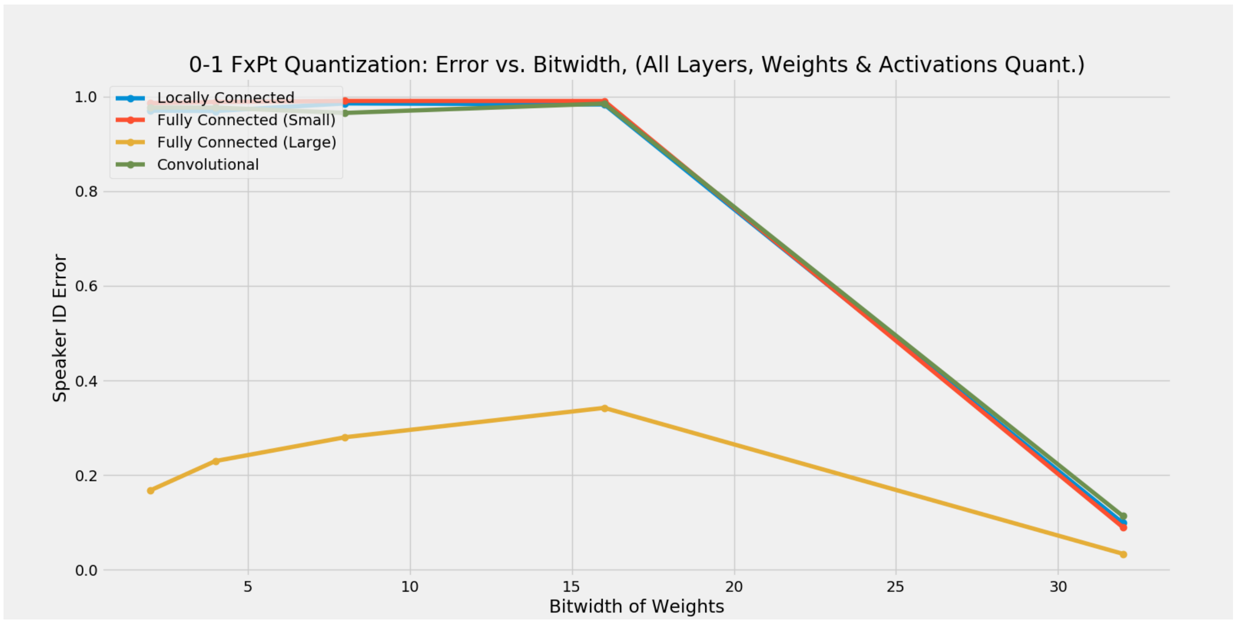


Figure 2-8: Speaker ID Error vs. Parameter Bitwidth (using the [0,1] fixed-point quantization). All parameters in the network are quantized. Activations are quantized to fixed point to match the parameter bitwidth.

Model	Error (%)	Mults	Params	Bytesize	Score	Baseline
FCN, Small (4-bit)	23.13	519K	521K	260KB	10.4962	11.0079
FCN, Small (8-bit)	22.78	519K	521K	521KB	10.7905	11.0079
FCN, Large (4-bit)	3.95	1433K	1435K	717KB	10.6096	11.3882
FCN, Large (8-bit)	2.60	1433K	1435K	1.43MB	10.7287	11.3882
CNN (4-bit)	17.48	265K	260K	130KB	9.7811	10.5411
CNN (8-bit)	11.59	265K	260K	260KB	9.9038	10.5411
LCN (4-bit)	26.49	287K	288K	144KB	10.041	10.5748
LCN (8-bit)	24.60	287K	288K	288KB	10.3098	10.5748

Table 2.4: Speaker Identification Error of [0,1]-Fixed Point Quantized Models on RSR2015. Scores on all models have show improvements compared to their respective baselines on Table 2.1.

error.

In this final section of model quantization and compression, we describe our final efforts reducing model size with ternary parameters.

2.4 Trained Ternary Quantization

We explore *trained ternary quantization* (TTQ), a method to learn weights in the set {-1,0,1} [Zhu et al., 2016]. In this work, we demonstrate that TTQ can be used in text-dependent speaker identification, with modification to the network's initialization and operational changes to decompose ternary networks into hardware-amiable binary operations. Though increasing error, ternary parameters allows for hardware optimization which we describe in the subsequent section.

In regular fully-connected and convolutional networks, every network layer l has a full-precision kernel W^l. Under TTQ, W^l is maintained, but not used during the forward pass. Rather, W^l is converted to its ternary approximation $\widetilde{W^l}$ by bucketing W^l's individual values, W^l_{ijk}, based on a layer-specific threshold Δ^l:

$$\widetilde{W}^l_{ijk} = \begin{cases} K^l_1 & \text{if } W^l_{ijk} \geq \Delta^l \\ 0 & \text{if } -\Delta^l < W^l_{ijk} < \Delta^l \\ K^l_2 & \text{if } W^l_{ijk} \leq -\Delta^l \end{cases}$$

This formulation is the original TTQ construction from Zhu et al. [2016], which uses two separate scaling constants K^l_1 and K^l_2 for the top and bottom thresholding regimes. In this work, we explored the simplification $K^l_2 = -K^l_1$, so that the full-precision scalar factors out of the weight matrix, reducing kernel multiplications to additions and subtractions, substantially decreasing circuit area and latency. The backward pass is also slightly modified; the gradient updates are applied to both the K^L scaling values, and the original W^l shadow weights. The threshold Δ^l can either be learned, but as is used in Zhu et al. [2016] and in our experiments, we approximate $\Delta^l = 0.05 \times \max|h^l|$, where h^l are the incoming activations to layer-l.

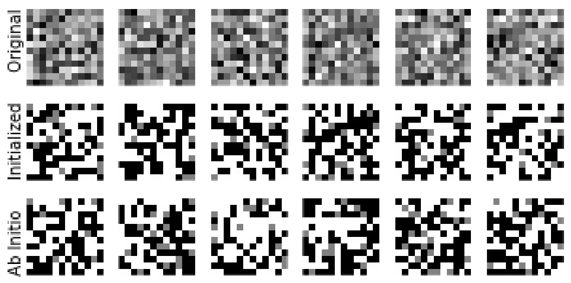


Figure 2-9: Top Row: floating-point kernels of our fully-connected SID model. Middle Row: final ternary kernels when initializing training from the floating-point model of the top row. Bottom Row: final ternary kernels when training from random initialization.

At validation and test-time, to avoid a costly max-accumulation in hardware, we use a cached average value for $\max|h^l|$ of the training set.

An example of the ternary kernels in our fully-connected SID model is shown in Figure 2-9. For the convolutional networks, pre-loading from floating point networks was required to avoid diverging training and NaNs.

To avoid full-precision floating-point operations, we can optionally constrain intermediate activations to 32 or 16-bit fixed point. After the multiply-and-accumulates in every layer, we normalize to [-1,1] (either by way of dynamic re-scaling or batch normalization) and downscale the activations to fit within 16-bits via bit-shifting and bit truncation.

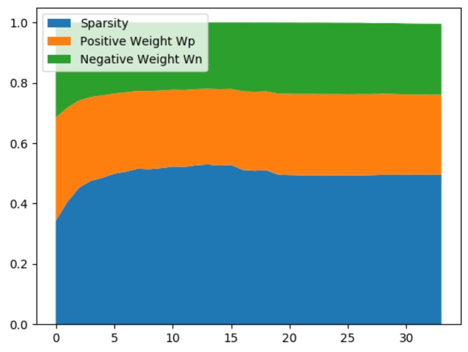
Of particular note, in this form of constraining weights, **no** multiplications are required during layer evaluation. With weights in the set {-1,0,1}, the parallelized units in hardware design needs to only support additions and subtractions across each row. The fixed point multiplication with K^l occurs lazily after these activations have been computed.

A summary of our TTQ models are shown in Table 2.5. The large FCN model ternarizes all layers, but attempting to ternarize all layers on the CNN results in a network with <5% accuracy. The CNN result below represents a model with middle layers ternarized, and first and last layers in full-precision.

Of note, in ternary models, there is significant sparsity, as illustrated in Figure

Model	Error (%)	Mults	Model Size	Score	Baseline
FCN, Large (32-Bit Ends)	11.04	648K	2.88MB	11.315	11.3882
FCN, Large (Tern Ends)	12.88	6K	553KB	8.6589	11.3882
CNN (32-Bit Ends)	38.43	73K	341KB	9.9849	10.5411
CNN (Tern Ends)	>95	-	-	-	10.5411
FCN, Small (Both)	>95	-	-	-	11.0079
LCN (Both)	>95	-	-	-	10.5748

Table 2.5: Speaker identification error of ternary quantized SID Models on RSR2015. The model size is up to 90% smaller, and the scores show improvement compared to their respective baselines on Table 2.1. They are also better than our [0,1]-FxPt model scores because of the decrease in multiplications. These mixed-precision and very low precision models are better suited for FPGA implementation.



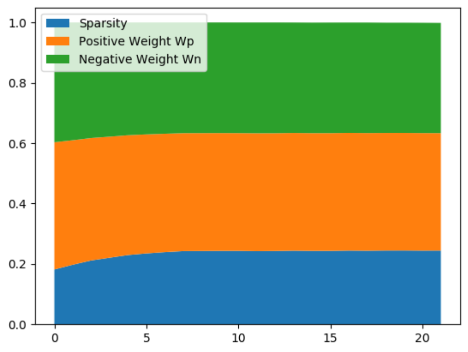


Figure 2-10: Composition of a sample ternary kernel in a SID large FCN (left) and CNN (right) as training progresses: percent of zeroes (sparsity), percent of positive weight values (K_1^l), and percent of negative weight values (K_2^l)

.

2-10.

2.5 Model Pruning and Distillation

To combine the advantages of highly sparse TTQ kernels and low-error [0,1]-quantization models, we seek to prune the latter to induce higher levels of sparsity. Blocks of zeroes enable us to reduce the number non-zero multiplications required during an evaluation of a model. Interestingly, in stark constrast to our TTQ models, our [0,1]-quantized SID models parameters have close to zero inherent sparsity.

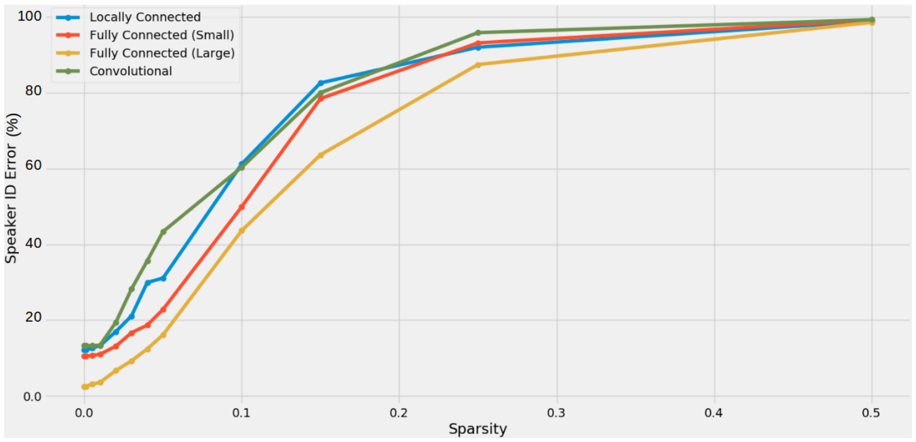


Figure 2-11: Speaker ID Error vs. Induced Sparsity into the 32-bit floating point model. Inducing 5% sparsity increase error by approximately 10%.

In particular, during pruning of our quantized models, we target a particular sparsity percentage P. Then for every weight kernel, we zero the P% of parameters that have lowest absolute magnitude. We retrain for 20 iterations, and redo the selective zeroing. This pruning process is based off the work of Han et al. [2015]. We tested pruning to the following twelve sparsity levels: 0%, 0.1%, 0.5%, 1%, 2%, 3%, 4%, 5%, 10%, 15%, 25%, and 50%.

Our results inducing sparsity are summarized in Figures 2-11, 2-12, and 2-13. In the baseline model (Fig. 2-11), we see that inducing 1% sparsity on the four models has marginal (<0.1%) effect on error. Inducing 5% sparsity or greater increases error by a roughly linear amount, leveling off at 25% sparsity to an error of about 90%. Not surprisingly, lower bit-width models are more sensitive to pruning, demonstrating higher rates of increase in error with increasing sparsity (Fig. 2-12, and 2-13). Figure A-3 in the Appendix illustrates the error of induced sparsity on 16-bit models; the trends are the same.

We suspect that we obtain lower levels of error-free pruning than Han et al. [2015] for two reasons: (1) our baseline model sizes are 10x smaller from the start than the large vision networks used in Han et al. [2015] (AlexNet/Resnet), resulting in increased initial network fragility and (2) we retrain less number of times due to GPU availability constraints.

Another possible way to achieve smaller model memory footprints is to exploit

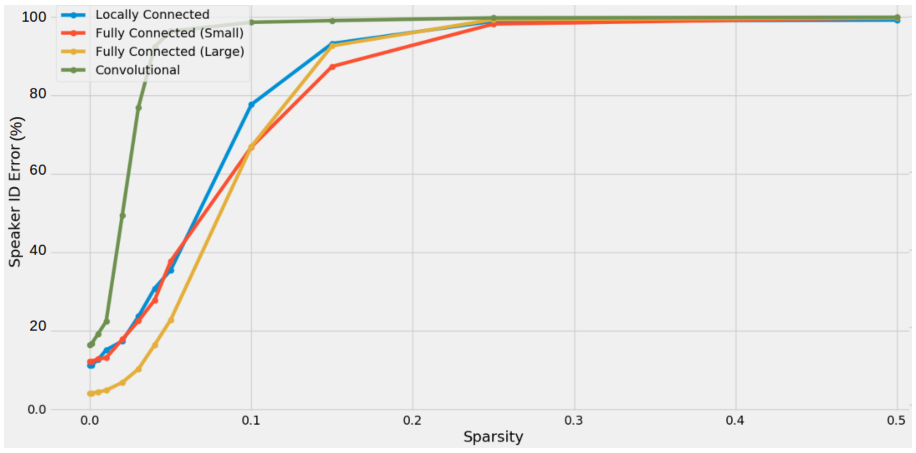


Figure 2-12: Speaker ID Error vs. Induced Sparsity into the 4-bit [0,1]-FxPt model.

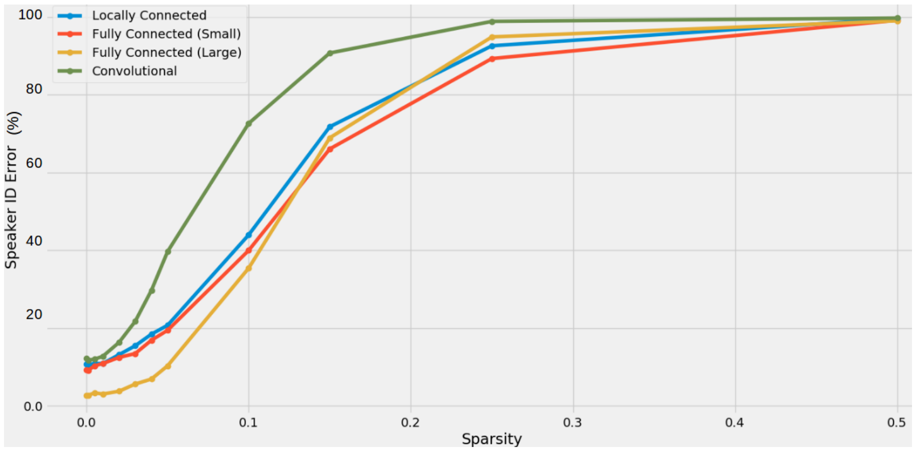


Figure 2-13: Speaker ID Error vs. Induced Sparsity into the 8-bit [0,1]-FxPt model.

sparsity, using, for example, row-compressed sparse matrix encoding. A straightforward way of doing so that has been attempted in the past would be to store a bit-vector with the locations of zero and non-zero values alongside a row-major list of the non-zero values in the matrix [Liu et al., 2013]. We find that this method would not decrease storage requirements in either our TTQ or fixed-point models.

In particular, the bit-vector representation decreases storage requirements if $\big((1-s) \times p \times b\big) + p \times \frac{1 \text{ bit}}{\text{param}} < p \times b$, where s is the sparsity level, p is the total number of parameters, and b is the bits per parameter. This condition does not hold for any of our models, because of the low sparsity levels.

We leave exploration of the other common sparse matrix representation, such as row-compressed sparse encoding, for future work. The choice of encoding is non-trivial: in particular, because of our low sparsity levels, the most common encoding schemes such as row or column-compressed sparse encoding would both increase the complexity of model evaluation hardware and actually increase the memory footprint [Wikipedia contributors, 2018] (such encoding schemes usually decrease memory requirements for sparsity levels >50%).

In constrast, we are able to reduce the number of multiplies and additions because of increased sparsity. Our hardware design (Section 3.3) for TTQ model evaluation allows us to skip zero multiplies to evaluate a partially-sparse dot product. In particular, our controller only feeds in a sequence of multiplies and additions to the matrix accelerator to execute if the operation is non-zero. This trick is common in designs like as those demonstrated by Han et al. [2016] and Chen et al. [2017].

Table 2.6 lists the summarized results of our pruning experiments. Out of the twelve different induced-sparsity levels tested (0% to 50%), we choose the sparsity level with the best score (thus balancing error, non-zero multiplications, and bytesize). For most FxPt models, the best model appears to be the one with very low sparsity level: 0.1%. The TTQ models, without any pruning, exhibit significant sparsity, and demonstrate better scores than the pruned [-1,1] fixed-point models. Though better than baseline, our pruned fixed point models are not significantly better than the original fixed models in Table 2.4.

Model	Error (%)	Sparsity (%)	NZ Mults	Bytesize	Score	Baseline
FCN, Large (4-bit)	04.13	0.1	1431.654K	717.695KB	10.6284	11.3882
FCN, Large (8-bit)	02.71	0.1	1431.654K	1.43MB	10.7457	11.3882
CNN (4-bit)	16.40	0	265.334K	130.259KB	9.7535	10.5411
CNN (8-bit)	11.69	0.1	265.068K	260.519KB	9.9073	10.5411
FCN, Large (TTQ)	12.88	22.80	4.939K	540.831KB	8.5368	11.3882
CNN (TTQ)	38.43	19.90	58.942K	139.719KB	9.5004	10.5411

Table 2.6: Speaker identification error of sparsity-induced [-1,1]-Fixed Point Quantized Models on RSR2015. 'NZ Mults' is the number of non-zero (non-skippable) multiplications in one network evaluation. For comparison, in the last two rows we list the unmodified TTQ models. More discussion in text.

2.6 Final Deployed Models and Summary

We describe the motivation for our final shortlist of models in Table 2.7. These models represent different points and our contributions along the accuracy-size-compute-sparsity surface for a text-dependent speaker ID challenge. This table lists the candidates with best score for each of the four model architecture types. Surprisingly, inducing sparsity models did not improve scores, so we did not select those models for our shortlist of final models.

This guides our approach for accelerator design in Section 3.2. Our ternary fully connected network exhibits the best score across all models. Thus, we build an architecture for ternary network evaluation (2-bit). We also demonstrate an accelerator capable of inference on 8-bit fixed-point parameters, to support the CNN, LCN, and FCN-small models that represent the next three best speaker ID models. Because of its low metric score, we did not list our most accurate model, the 4-bit FxPt FCN-Large, in the table, but the same accelerator is able to be used for this network as well.

Final Model Type	Error (%)	Mults	Bytesize	Score
FCN, Large (Tern Ends)	12.88	6K	553KB	8.6589
CNN (4-bit)	16.40	265K	130KB	9.7811
LCN (4-bit)	26.49	287K	144KB	10.041
FCN, Small (4-bit)	23.13	519K	260KB	10.4962

Table 2.7: Speaker identification error and other results across all experiments: selected from baselines, fixed-point, ternary, and pruned speaker ID models. Each of the four architecture types are listed, and we chose the best scoring within each type. Ordered by score.

Chapter 3

An FPGA-based Accelerator for Speaker Identification

In this chapter, we describe our work evaluating out trained models on a commodity FPGA. In particular, we target a low-end Xilinx FPGA briefly described in Section 3.1. We describe two different architectures for evaluation of our variable-width models: (1) in Section 3.2, we describe our Eyeriss-inspired design for 4-bit evaluation and results from post-implementation simulation. (2) in Section 3.3, we describe our architecture for evaluation of our speaker ID ternary network and corresponding post-simulation simulation in Vivado. In Section 3.4, we briefly describe our work to extract cepstral features on the Cortex A9 on the Xilinx FPGA. Finally, in Section 3.5, we describe the setup and flow of our final system used perform speaker verification with real people. A working demonstration of the system can be viewed at `https://youtu.be/n1wdkIRCnrU`.

3.1 Target: Xilinx Zynq-7000 Dev Board

We targeted our model design around the memory constraints of a commodity FPGA. In particular, in this work, we chose the Digilent Zybo Z7, which features the XC7Z020-1CLG400C Xilinx SoC [Xilinx, 2018]. This chip possesses the programmable logic resources given in Table 3.1. It has a 1GB DDR3 off-chip memory and in-built

Resource Type	Count
Look-up Tables (LUTs)	53,200
Flip-flops	106,400
DSP Slices	220
Block RAM	630 KB

Table 3.1: Available resources on the target Zybo Z7 development board (Xilinx Core XC7Z020-1CLG400C).

Cortex-A9 ARM processor alongside the programmable logic fabric. The full list of components in the SoC are give in Appendix Figure B-1. We selected this family of FPGAs because it was the lowest cost family of development boards offered by Digilent (primary distributor of consumer Xilinx/Altera FPGAs). The actual development board is pictured in the Figure 3-1. Both the programmable logic and the ARM core are clocked at 200MHz that is driven by an external crystal and PLL.

3.2 8-bit Fixed-Point DNN Accelerator

To demonstrate a simple baseline for our hardware design, we implement an Eyeriss-inspired DNN accelerator on our target FPGA. In particular, we recycle the idea of using an array of compute 'Processing Units' (PUs) to carry out all of the inner product/non-linearity computations required during inference. To begin, we target evaluation of our 8-bit 5-layer, width-256, 521KB FCNs (the first two models in Table 2.4).

The design of our accelerator is best described in Figures 3-2, 3-3, and 3-4. In particular, Figure 3-2 describes interaction of components on the Zync-7000, and information flow between the processing core (PS) and programmable logic (PL). Figure 3-3 describes our design for neural network inference on the PL of the Xilinx chip. Finally, Figure 3-4 describes the micro-architecture of the individual units the comprise the inference module.

Our design is a simplification of the Eyeriss accelerator. The following changes were appropriate given our target model:

- We reduce the buffer sizes across the design because of our target model/MFCC

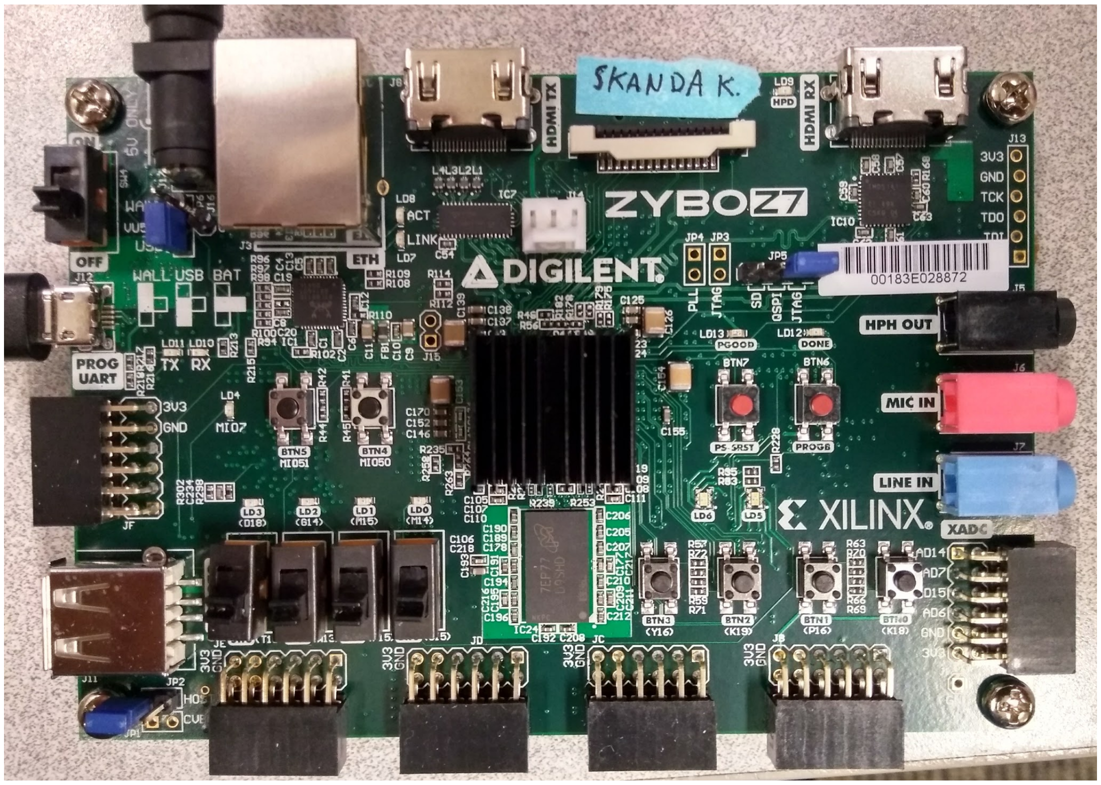


Figure 3-1: Zync-7000 development board used in our speaker verification demonstration. Uses a Xilinx XC7Z020-1CLG400C programmable logic core (center of board, with the black heat sink on top of it).

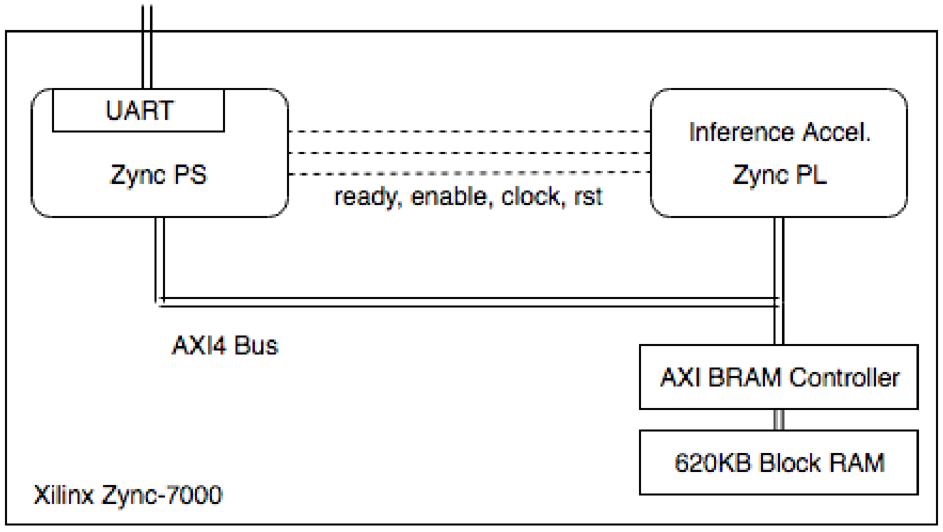


Figure 3-2: Macro-architecture of our speaker identification system on the Xilinx Zync-7020. The two key components available on the PS (ARM Processing System) for UART communication and feature pre-processing and PL (Programmable Logic), for model evaluation.

inputs are much smaller than the vision networks/input images targeted by Eyeriss

- We replace the 2D PU array with a small linear PU array that is more suitable for parallel evaluation of FCN row dot products (description in Section 3.2.2).
- A linear PU array means less complicated data routing patterns (e.g. avoiding the necessity to tile and collect data across the spatial array), allowing for smaller and less complex network-on-chip modules
- Fewer buffers in total by removing the need to buffer off-chip memory read/writes (which is not used in the design), and because we halve inter-PU connections by using a linear PU array.

Note that these complexity and area advantages are possible because this design is specifically optimized to evaluate our top scoring network, the speaker ID FCN networks. As such, our design requires some input manipulation to also evaluate convolutional and locally-connected networks (discussed in Section 3.2.2).

Figure 3-2 describes the macro-architecture of the design. In particular, the Processing Core (Zync PS) interfaces with the outside world via UART (in-built

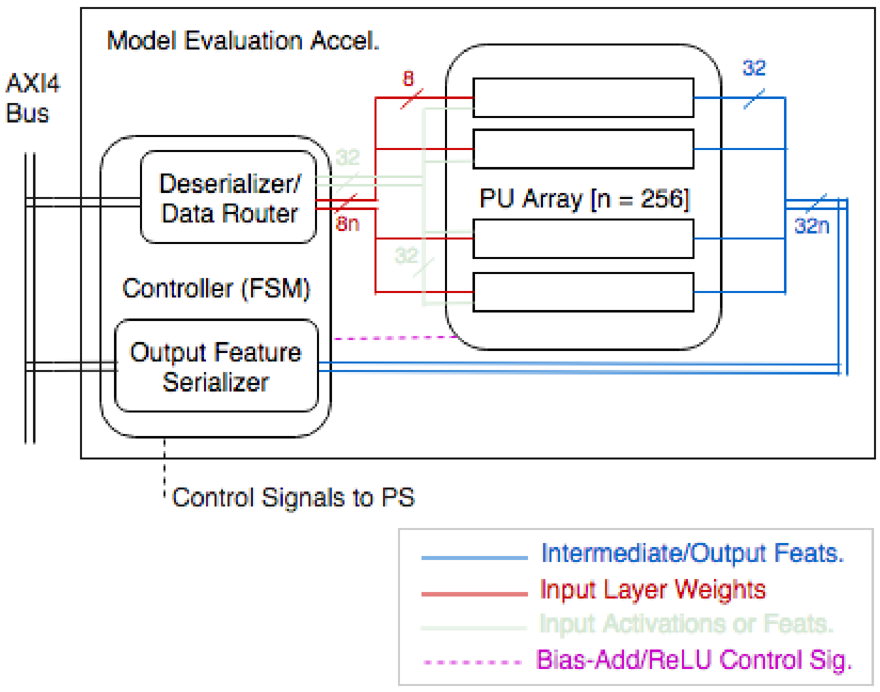

Model Evaluation Accel.
AXI4 Bus
Deserializer/ Data Router
Controller (FSM)
Output Feature Serializer
PU Array [n = 256]
8
32
8n
32
32
32n
Control Signals to PS
Intermediate/Output Feats.
Input Layer Weights
Input Activations or Feats.
Bias-Add/ReLU Control Sig.

Figure 3-3: Architecture of our core model evaluation accelerator. There are three main data streams: the input feature (blue) and weight stream (red), and the corresponding output accumulation stream (green). Each network layer is evaluated sequentially. The routing unit acts as (1) a controller, orchestrating communication based on control inputs (2) network routers for the PUs (distributing and collecting data from the array) and (3) the interface to the BRAM.

FT-232R IC for USB/UART communications). The feature extractor runs on the core processing the initial data samples received over UART. The extracted features are sent to BRAM via Xilinx's AXI4 bus, where they are stored in a 20K segment (occupying about half the segment), as shown in Figure 3-5. Model evaluation occurs using our programmable logic low-precision accelerator, and results are stored in Block RAM. The PS accumulates these results, and returns back a verification decision over UART.

Figure 3-3, the inference accelerator and focus of this design, centers around a linear array of Processing Units (PUs) that sequentially compute the dot product summation. The correct segments of Block RAM are read using logic in a controller module which feeds the layer weight and layer input data simulatenously into the PU array. Parallel outputs from the PU array are received on the same cycle and, after serialization, fed back into BRAM, overwriting the old block of interme-

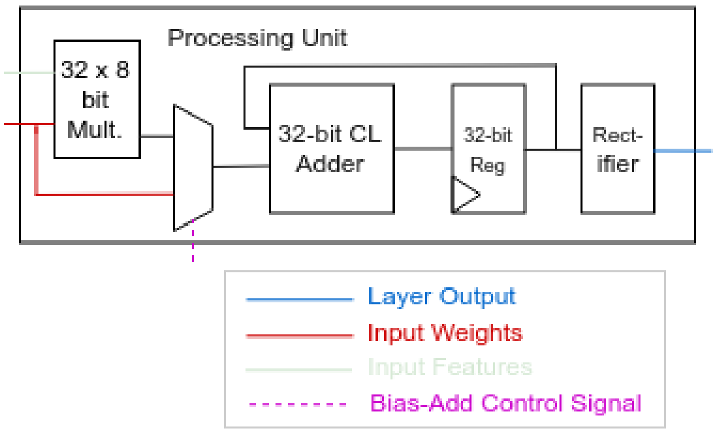


Figure 3-4: Micro-architecture of each individual processing unit. Each PU computes the dot-product $w \cdot x$ by accumulating the terms in the summation in a full-precision register. After the dot product has been computed, the bias-add line is pulled high and the bias is added to the accumulation register, and the output is read.

diate activations/layer inputs. The controller is a simple 3-state finite state machine (sleep/intialize/run) that responds to control signals from the PS. 3-element FIFOs separate each part of the flow, just like in the original Eyeriss design (e.g. the data router and input lines to elements of the PU array).

In the first revision of our design, we use n PUs. Every cycle, each PU is fed an an 8-bit weight and 32-bit input activation to add the accumulation. On the last cycle, the "Bias-Add" signal is toggled high and the 32-bit bias is fed in and added to the accumulation. The output of the array is the set of n 32-bit accumulation registers. The reason for 32-bit activations in this design is hashed out in Section 2.3.2.

For models with bigger or smaller layer output sizes, we can scale the size of the linear array. In this work we tried $n = 256$ and $n = 512$, to fit our small and large FCNs.

Figure 3-4 describes the core computation block for the multiply-accumulate. We add a rectification block at the end (comparator and mux) to perform the non-linear ReLU activation. Additionally, to support models with quantized activations (like models in Figure 2-8), we added a small quantization block after the rectification that performed a bit-shift and truncate to bring the activations within range. The

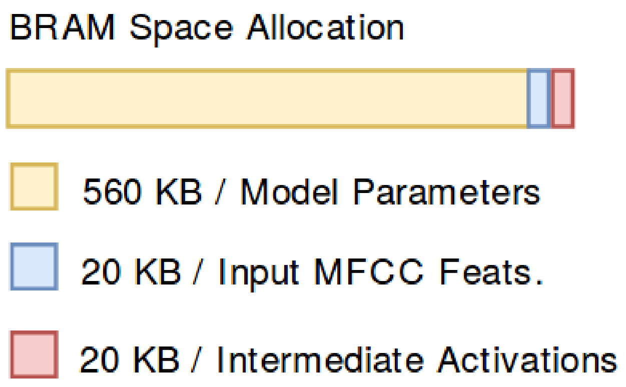


Figure 3-5: Utilization of available BRAM on the Xilinx development board.

adder is 32-bit fixed-point, with an implicit 4-bit/28-bit integer/decimal fixed point representation. The adder design is the default chosen by Bluespec compiler/Xilinx synthesis/mapping tools. The multiplier can afford to be 32-bit by 8-bit, but for simplicity, in our initial design, we use the default 32-bit multiplier provided by Xilinx synthesis tools.

3.2.1 Memory Management on Xilinx Zync-7000

Xilinx FPGAs offer I/O to peripherals and memory through their *Advanced eXtensible Interface* (AXI) protocol. Xilinx provides the corresponding IP for communication using AXI [Xilinx, 2018]. We use AXI4 for data communication between the Zync PS core, the programmable logic, and Block RAM.

In particular, on the 7000-series Xilinx cores, each block is 36 Kbit. We set each block to use simple-dual port communication, resulting in 72-bit wide read/write accesses. Each memory block is 512 by 72 bits. In total, our design all available 140 blocks, the capacity of our particular core type, the ZC-7020.

The allocation of Block RAM available for our design is given in Figure 3-5. Unsurprisingly, majority of space is used for model storage. Our layout of values within these allocations is rather straightfoward: we store each layer's weights as a block, storing the 4/8-bit values continguously in a row-major fashion. To read out a 512-wide layer weight matrix requires us $(512 \times 512 + 512) \div \frac{72}{8} \approx 29200$ reads.

At 200 MHz, this translated to a 291 μS per-layer memory load latency. For our 5-layer, width-512 FCN network, we measured model parameter read time + output feature write time to be 1.455 ms. These lower bound memory read/write overheads were benchmarked using the simple Vivado design outlined in Figure B-2. This was the design we used in November 2017 to test basic functionality of BRAM on the development board. The design has connected: the Zync Processing Core, BRAM Controller, all 140 Block RAM units, and a simple controller written in Vivado HLS used to execute read and writes from BRAM.

3.2.2 Computation Flow on the Accelerator

At a high level, it is useful to quickly understand how computation of each neural network layer maps to the hardware. At its core, the accelerator repeatedly evaluates the matrix-vector formula non-lin$(W_l \cdot x + b_l)$.

Figure 3-6 describes the evaluation of matrix-vector multiplication $W_l \cdot x$. Every cycle, two inputs are fed in: a column of weight matrix W (light blue highlight on the left) and a value of input feature vector x (beige highlight in the center). Each PU is passed a different value along the weight matrix column, but the same x_i.

The PUs have accumulation registers, which are represented by the right-most dark blue vectors in the figure. Each output cell corresponds to a different PU. To evaluate a 512×512 matrix multiply, it takes 512 cycles of feeding in each column of input.

To handle FCN networks of smaller size than the PU array, we under-utilize the hardware and do not feed any inputs to the number of extra PUs that we have. This is useful for evaluation of our locally-connected networks, where the first layer is comprised of many smaller fully-connected subsections. To evaluate FCN networks of large size (for example, evaluation of a 512-wide network on a $n = 256$ PU array), we break the multiply into $n \times n$ matrix multiplies, and save the partial sums in the accumulation registers.

To evaluate the CNN matrix, we convert the convolution operation $(W \circledast x)$ into a matrix multiply $(W \cdot x)$, using the step of converting the kernel into the corresponding

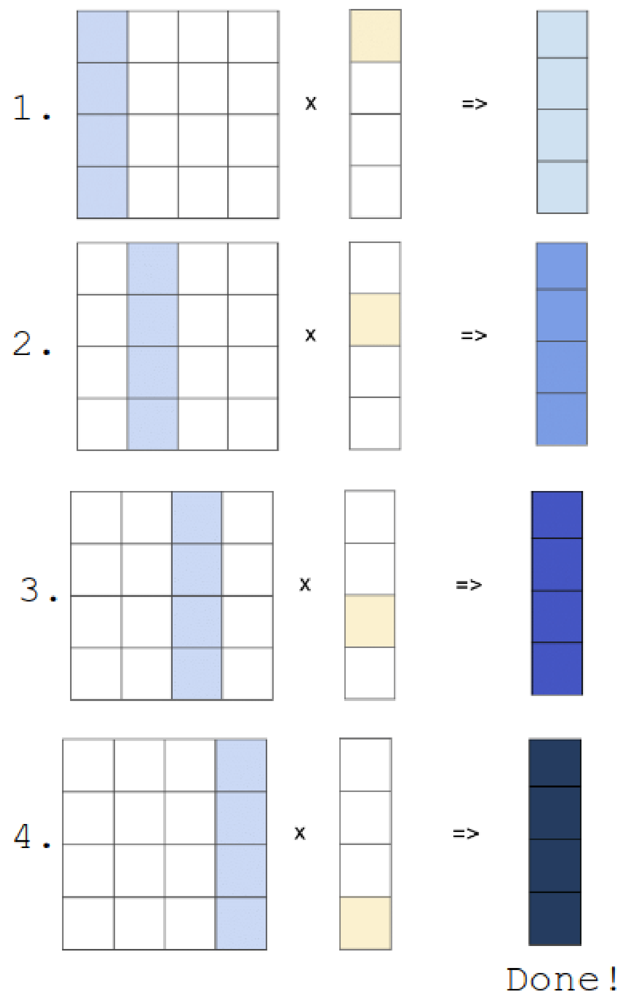


Figure 3-6: High-level procedure of how the accelerator computes layer computation $W \times x$. W is the light blue matrix to the left, x is the center beige vector, and the output is given by the rightmost vector. Every 32 cycles, another column of the weight matrix is processed and corresponding terms in the dot product added to the accumulation output registers.

$$x \circledast W = \underbrace{\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix}}_{x} \circledast \underbrace{\begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix}}_{W} \tag{3.1}$$

$$= W_T \cdot x_T = \underbrace{\begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix}}_{W_T} \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}}_{x_T} \tag{3.2}$$

Figure 3-7: Example of a 3 by 3 Toeplitz transformation to support convolutional networks on our design. The transformation allows us to reduce convolution to a matrix-vector multiply.

Toeplitz matrix. This is described in Example 3-7. By converting the convolution operation to matrix multiplication, we can use the same design to evaluate convolutional networks. In our convolutional networks, input x, is folded to be a matrix in the first layer, and unfolded for the following fully-connected layers. This conversion can either be done on the fly (which results in no memory storage overhead, but at the cost of increasing latency and controller complexity), or as a pre-processing step, which would increase memory overhead. We chose the former approach to implement. We are still in the process of changing our original kernel to support on the fly Toeplitz transformation, and this is an avenue for future work.

3.2.3 Evaluation of the Design

We obtain post-synthesis results to characterize our design for $n = 256$ and $n = 512$, as well as for 8 and 32-bit parameter width values. Among the things that changed between the 8-bit and 32-bit design variants: port widths, buffer widths, and arithmetic unit input widths, and the deserializer loading logic of 8 vs. 32-bit values from BRAM.

In particular, in this section we look at a few of the reports provided by Vivado: FPGA resource utilization, post-routing power consumption estimates, and slack w.r.t. a 100 MHz target clock. We use a vector-based power estimation: we create a simulation activity file (SAIF) in Vivado XSIM, that captures switching in the design, and then import this into the open post-routing implementation in Vivado Designer to obtain power estimates.

Resource Utilization

Table 3.2 summarizes the FPGA resource utilization, totalled across all registers, memory cells, muxes, and other programmable logic components in the design. We divide our usage into the four main categories of available resource types: look-up tables, flip-flops, block RAM cells, and DSP slices.

	n=256, 8-bit	n=256, 32-bit	n=512, 8-bit	n=512, 32-bit	Total Available
FF	52,172	71,262	86,224	104,524	106,400
LUT	29,832	33,996	47,021	50,901	53,200
DSP48E	200	200	220	220	220
BRAM	140	140	140	140	140

Table 3.2: Resource utilization for four designs of the fixed-point accelerator: varying the size of the linear PU array ($n = 256$ and $n = 512$), and varying the bitwidth of the accelerator parameters and multipliers.

Unsurprisingly, reducing the data line widths, buffer sizes, and arithmetic units to 8-bit reduces resource utilization on the FPGA (in terms of FF/LUT/DSPs). All designs instantiate a fixed 140 BRAM blocks (all 630KB), and are able to fit our 8-bit small FCN or 4-bit large FCN. Utilization of BRAM does not decrease in order to always accomodate the largest model size possible. In our $n = 512$ designs, we maxed out DSP slice resources on the FPGA.

Latency and Slack

Our designs were synthesized targeting a timing constraint of 10 ns clock period (100 MHz). Here we report the timing slack in the design (given in nanoseconds), and the latency of one model evaluation obtained during simulation (given in clock cycles).

	n=256, 8-bit	n=256, 32-bit	n=512, 8-bit	n=512, 32-bit
Slack [uncertainty] (ns)	8.18 [1.12]	8.31 [1.09]	8.23 [1.21]	8.61 [1.35]
Latency (cycles)	2,231,834	2,612,050	2,232,864	2,613,080
Throughput (eval/s)	44.806	38.284	44.785	38.269

Table 3.3: Timing characteristics of variants of the fixed-point accelerator design. We used a target 100MHz clock constraint. Latency is given in cycles, throughput in model evaluations per second.

Running 50 model evaluations in simulation, we also obtain an estimated evaluations per second throughput of our designs.

One particularly tricky aspect when comparing these designs is model sizing: 32-bit designs are only able to fit in the available BRAM only models with fewer parameters, meaning there are less overall operations to compute compared to low-precision parameter designs. Instead, to cross-compare timing for lower-precision designs with high-precision designs, we keep the model the same across evaluations: a 5-layer, 90-wide 'small FCN', that is 522KB for 32-bit parameters and 130KB for 8-bit parameters. This model is able to fit in BRAM for all parameters widths.

Table 3.3 summarizes the results of the fixed-parameter-count model evaluation of our designs. All designs were able to meet the timing constraint. The 32-bit designs had significantly higher cycle latency; we suspect this is because of the larger amount of data serialized and deserialized from memory. The throughput followed the same trends as the latency, because of no differences in pipelining between the designs. We suspect that the differences between the $n = 512$ and $n = 256$ designs for same bitwidth is because of choices in resource allocation/arithmetic module types by the Vivado synthesis/PaR compiler.

If we had a model with kernels of size between 256 and 512, we would see the cycle count between the $n = 256$ and $n = 512$ significantly differ (by a factor of approximately four), because it would require four complete 256×256 kernel evaluations (which can be done in one pass in the $n = 256$ hardware), to evaluate a 512×512 kernel. If we instead performed the timing evaluations with a fixed memory footprint for the model (and different parameter count), we would have encountered a roughly $2\times$ difference in number of multiply-and-accumulates, and roughly $2\times$ difference in cycle count latency between the 8-bit parameter and 32-bit parameter designs.

This is because for a fixed model bytesize, there would be 4 $\times$ as many 8-bit parameters, instead of 32-bit parameters. A square matrix with 4-times as many parameters, has 2x the number of multiplies and adds in each row's dot-product during the matrix-vector multiply.

Power Consumption

Power estimates were conducted by importing simulation results (SAIF file, a summary of switching activity) into the Xilinx Power Estimator tool.

For the four variants of the design, we obtain the estimates given in Figures 3-8a, 3-8b, 3-8c, and 3-8d.

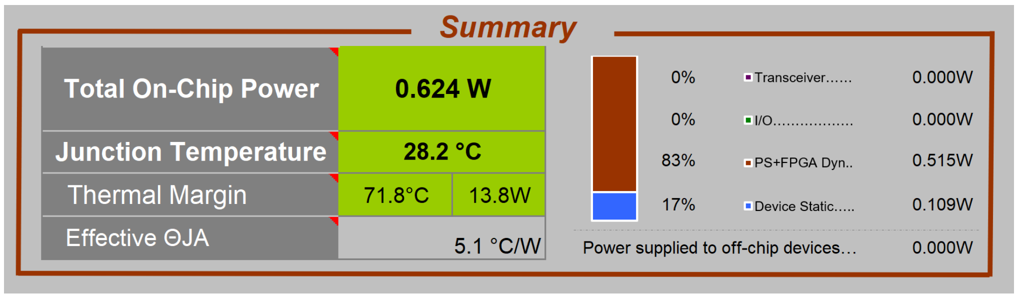
For comparison, we benchmarked a sample design (with nearly less than half the FF/LUT usage of our accelerator). This design performed periodic read/writes to a DDR3 DRAM module, and gave us an estimate of the power consumption if we did use model compression or quantization (and if we required off-chip memory). As expected, the power consumption estimates more than doubles, as shown in the summary report in Figure 3-9.

The breakdown of power consumption for the $n = 256$, 8-bit parameter design is shown in Figure 3-10. The logic resources and clock draw the most power, followed by the BRAM.

3.3 An Accelerator for Ternary Networks

We explored a design for evaluating our fully-connected ternary networks. In particular, this design exploits the unique fact that in ternary networks, there is only multiplication with one number per layer. The overall information flow between the programmable logic and processing core and BRAM is the same as in Figure 3-3.

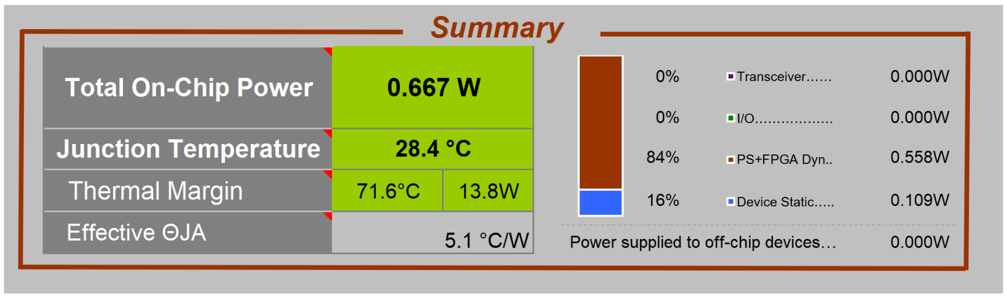
Figures 3-11 and 3-12 outline our ternary network accelerator design. The key difference is in the PU design: we delay the multiplication to the serialization block, multiplying the layer accumulations by the same layer weight constant W_p and W_n,


Summary

Total On-Chip Power	0.624 W	
Junction Temperature	28.2 °C	
Thermal Margin	71.8°C	13.8W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
0%	I/O.................	0.000W
83%	PS+FPGA Dyn..	0.515W
17%	Device Static.....	0.109W
	Power supplied to off-chip devices...	0.000W

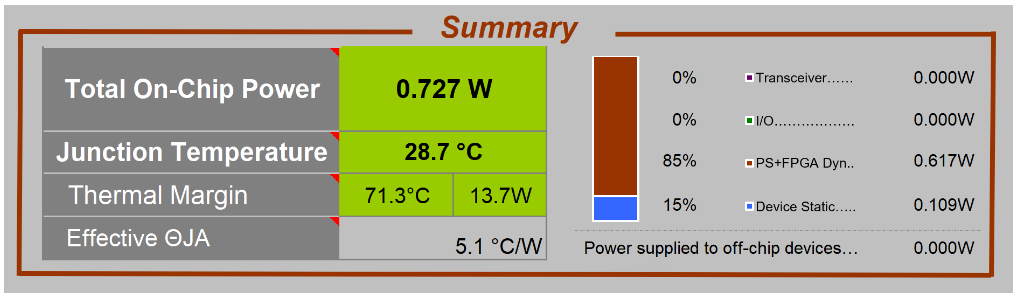
(a) Summary report for power consumption of the $n = 256$, 8-bit parameter design. Notably, by removing the need for off-chip memory accesses to DRAM we excise the energy costs associated with that high-frequency memory I/O and power supplied to off-chip devices, compared to results shown in Figure 3-9.


Summary

Total On-Chip Power	0.667 W	
Junction Temperature	28.4 °C	
Thermal Margin	71.6°C	13.8W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
0%	I/O.................	0.000W
84%	PS+FPGA Dyn..	0.558W
16%	Device Static.....	0.109W
	Power supplied to off-chip devices...	0.000W

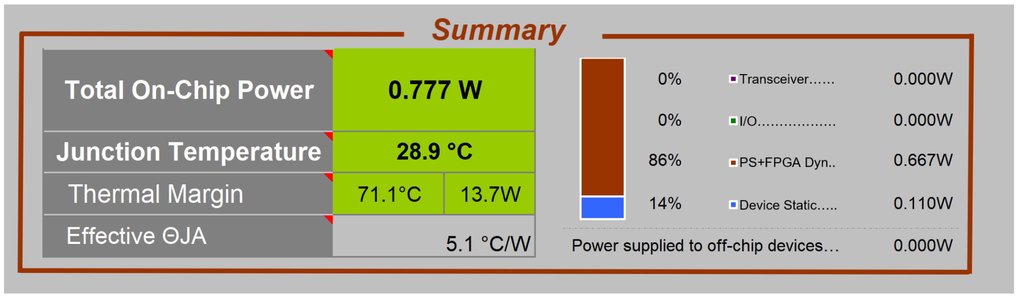
(b) Summary report for the $n = 256$, 32-bit parameter design. The power consumption is 7% higher compared to the 8-bit design because of the corresponding larger design.


Summary

Total On-Chip Power	0.727 W	
Junction Temperature	28.7 °C	
Thermal Margin	71.3°C	13.7W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
0%	I/O.................	0.000W
85%	PS+FPGA Dyn..	0.617W
15%	Device Static.....	0.109W
	Power supplied to off-chip devices...	0.000W

(c) Summary report for the $n = 512$, 8-bit parameter design. The power consumption is 9-16% higher compared to either of the $n = 256$ designs because of the $2 \times$ PU logic.

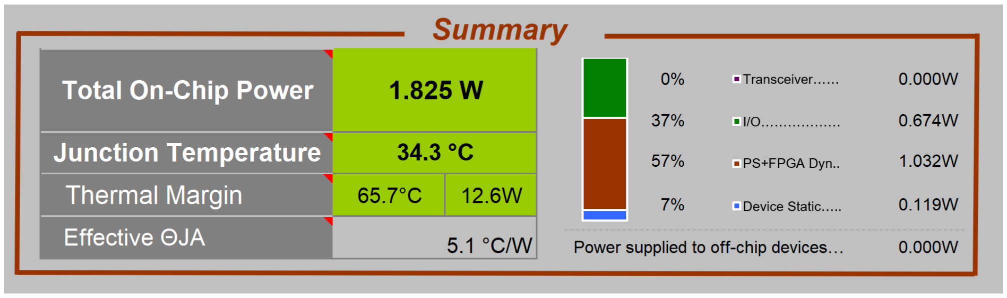

Summary

Total On-Chip Power	0.777 W	
Junction Temperature	28.9 °C	
Thermal Margin	71.1°C	13.7W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
0%	I/O.................	0.000W
86%	PS+FPGA Dyn..	0.667W
14%	Device Static.....	0.110W
	Power supplied to off-chip devices...	0.000W

(d) Summary report for the $n = 512$, 32-bit parameter design. This design has the highest estimated power consumption of all designs, by up to 24%

as outlined in Section 2.4. This allows us to save having to run the multiply every cycle in each PU, and correspondingly removes a multiplier from each of the 256/512


Summary

Total On-Chip Power	1.825 W	
Junction Temperature	34.3 °C	
Thermal Margin	65.7°C	12.6W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
37%	I/O................	0.674W
57%	PS+FPGA Dyn..	1.032W
7%	Device Static.....	0.119W
Power supplied to off-chip devices…		0.000W

Figure 3-9: Summary report for a sample design with similar FF/LUT usage that performs reads and writes to DDR3 DRAM.

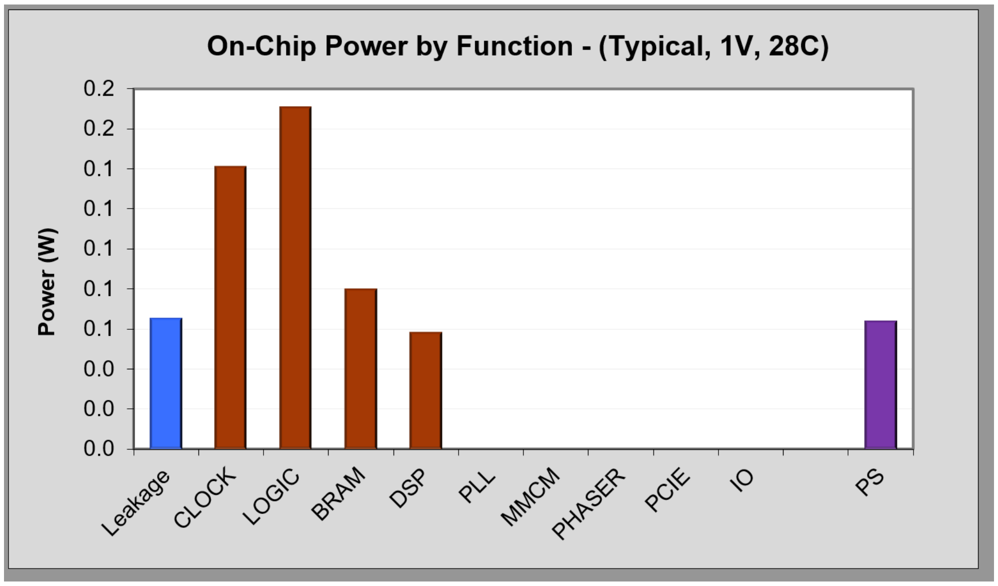


Figure 3-10: Summary of power consumption breakdown of the $n = 256$, 8-bit parameter design. This distribution is the same for other design variants, as shown in Figures B-4, B-5, and B-3.

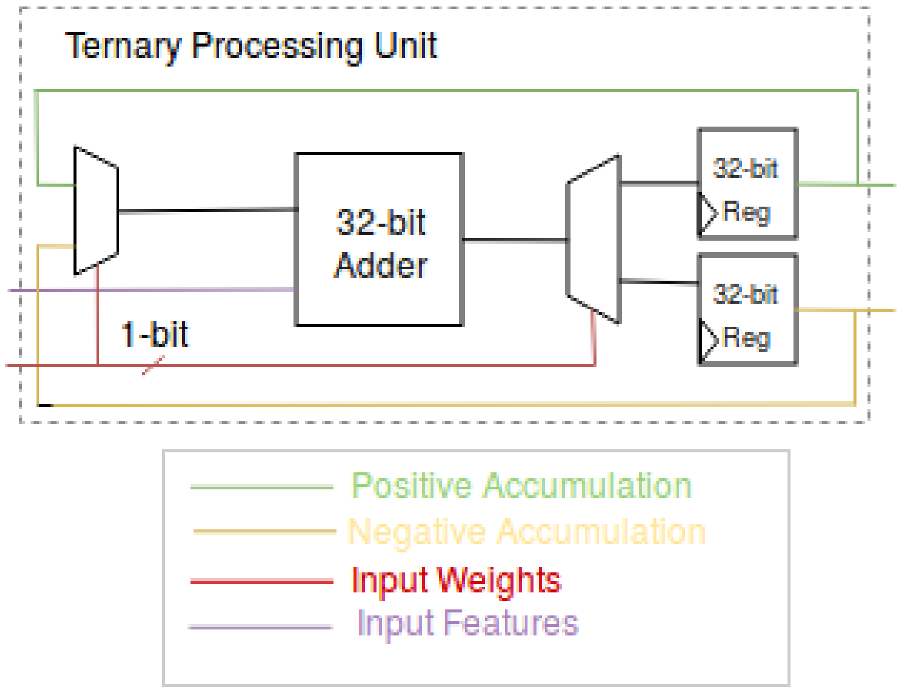


Figure 3-11: Modified PU design for the ternary accelerator. Forward evaluation does not require multiplication with a unique number on every input, so we delay the multiplication with layer constant until the serialization when we write back into BRAM.

PU blocks. Adding the bias b is added as an extra input b/W_p.

An aspect of this design that were briefly explored, but still warrants future work, is adding parallel multipliers in the serializer (having only one multiplier working sequentially reduces area/power, but increases latency). Moving the multiplication to the serializer allows us to tune where we are in this trade-off curve, by increasing or decreasing the number of in-parallel multiplies.

We briefly explored skipping zero multiplies, which offer us 20-30% latency improvements given the sparsity of ternary models. This was implemented in the deserializer, which checked if the corresponding loaded weight from BRAM was zero, if so, did not feed the input feature and weight into PU array. Testing this flow, and verifying improvements currently a work in progress.

We only benchmark the $n = 512$ design because corresponding network size of width 512 is the only network size that achieves reasonable identification accuracies.

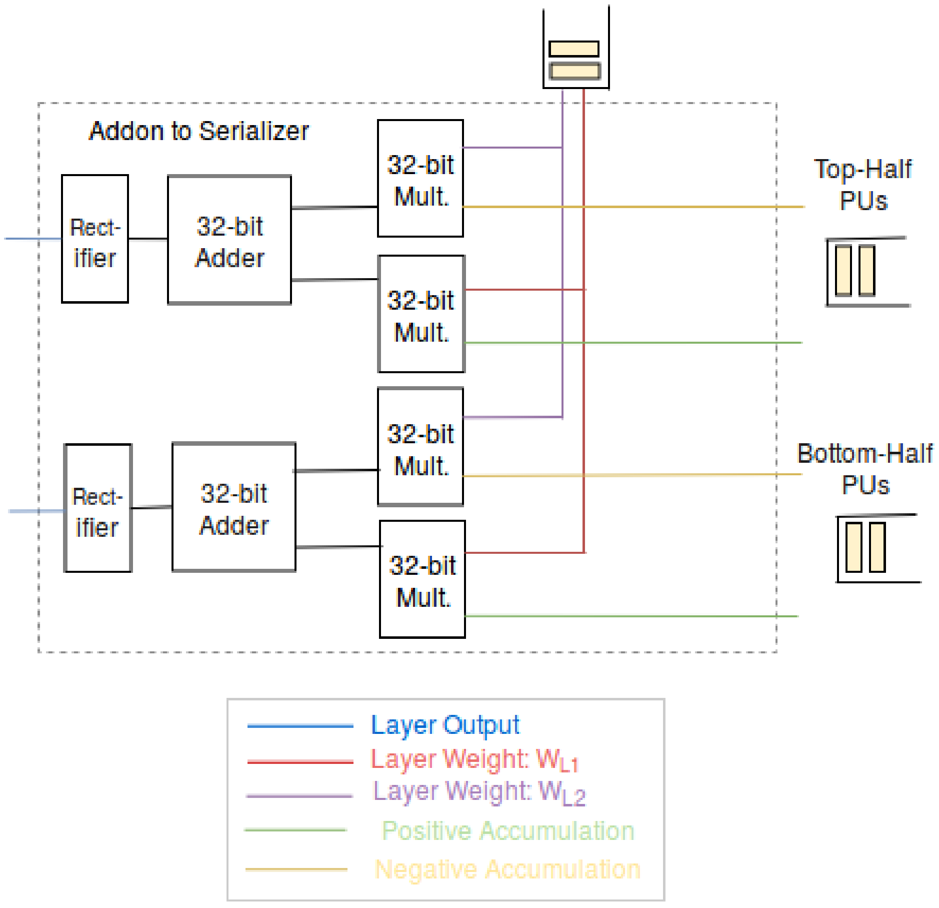


Figure 3-12: Addition to the serializer for the ternary accelerator.

3.3.1 Evaluation of the Design

The same Vivado synthesis and simulation tools as were used in the previous section. The following is a summary of resource, timing, and power estimates for this design.

Resource Utilization

The breakdown of resource utilization is given below in Table 3.4.

	n=512, Ternary	Total Available
FF	70,712	106,400
LUT	41,619	53,200
DSP48E	200	220
BRAM	140	140

Table 3.4: Resource utilization for ternary accelerator.

Latency and Slack

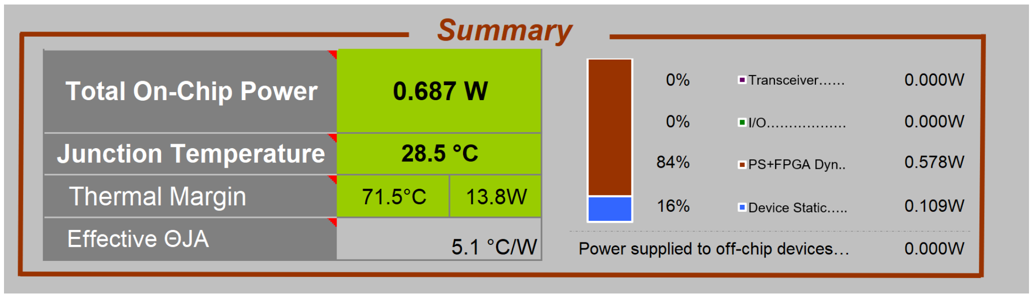
Adherence to 100 MHz target timing constraints and the average latency for one model evaluation is given in Table 3.5.

	n=512, Ternary
Slack [uncertainty] (ns)	9.07 [0.92]
Latency (cycles)	2,206,090
Throughput (eval/s)	45.329

Table 3.5: Timing characteristics of variants of the ternary accelerator design. We used a target 100MHz clock constraint. Latency is given in cycles, throughput in model evaluations per second. Slack is much tighter than in our previous designs.

Power Consumption

Summary reports of estimated power consumption for our ternary models from the Xilinx Power Estimator tool (post-implementation) are provided below in Figures 3-13 and 3-14.



Summary

Total On-Chip Power	0.687 W	
Junction Temperature	28.5 °C	
Thermal Margin	71.5°C	13.8W
Effective ΘJA	5.1 °C/W	

0%	Transceiver......	0.000W
0%	I/O.................	0.000W
84%	PS+FPGA Dyn..	0.578W
16%	Device Static.....	0.109W
	Power supplied to off-chip devices…	0.000W

Figure 3-13: Summary report for the ternary design, demonstrating a 6% reduction in power consumption from the $n = 512$ 8-bit designs previously demonstrated.

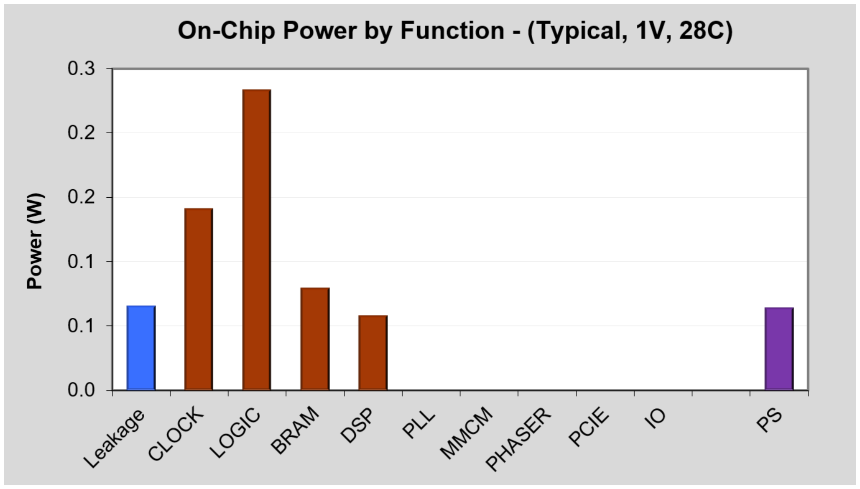


Figure 3-14: Breakdown of power expenditure for the ternary design. This design follows roughly the same distribution of power consumption for different resource types as the original fixed point design.

3.4 Feature Extraction

We perform feature extraction from audio samples on the ARM core provided by our Xilinx Zybo development board. This feature extraction was done on the ARM core (as opposed to custom Verilog) for three key reasons: (1) feature extraction represents roughly 10% of the total critical path computations in a single speaker evaluation, and on the 200 MHz ARM core, extraction would not be the bottleneck in the system throughput (2) a C-based feature extractor can be developed and tested more quickly, which allowed the author to complete a working demonstration and submit his thesis in a timely fashion and (3) a firmware implementation is able to easily bootstrap off the existing C++ source in audio toolkit Kaldi.

As part of future work, we desire to fold this audio sample pre-processing into the main accelerator design, developing a Verilog module based on the work of Price et al. [2017] for MFCC extraction. We expect this would offer significant power savings, eliminating the need to clock, power, and run the ARM core on the Zybo.

Our implementation runs extraction of Mel-frequency cepstral coefficients from 25 ms windows of 8KHz audio samples (the steps outlined in Section 1.3.1). We buffer 400 incoming samples at time from the host in order to compute the windowing and cepstral calculations.

Unfortunately, porting log-Mel feature extraction to the Xilinx Zync ARM core was not as straightforward as expected. During training of the network, we used the Kaldi framework which is written in C++. Re-using sections of Kaldi, as we discovered, was not possible for a few reasons:

- Kaldi depends extensively on the C++ Standard Template Library (STL), which requires dynamic memory allocation [Povey et al., 2011]. Dynamic memory allocation requires a heap, and memory allocator, which is usually bundled into the host kernel. Running on the Xilinx ARM core, on bare metal, means that we have no dynamic memory allocator, and certainly no kernel [Povey and Contributors, 2018].

- Kaldi has significant tooling and infrastructure code within the project (>30K lines of code in total), from which extricating the core log-Mel functionality is messy.

We ended up writing our own feature extractor in pure C. This implementation (`https://github.com/skoppula/speaker-id-thesis/feature-extraction/extractor.c`) has no external library dependencies, and requires no dynamic memory allocation. We based our implementation on prior bare-metal C implementations of spectra calculation [Hills, 2013] and DCT calculation [Sawruk, 2010]. Our feature extracton implementation was compiled using the 7.2.1 'arm-none-eabi-gcc' compiler and successfully run on our Xilinx ARM core.

We noticed slight differences in the Mel-frequency cepstral coefficients output by the Kaldi implementation and our system implementation. In particular, we noticed what appears to be random deviations from the golden Kaldi standard with the average deviation magnitude of 0.75 (picture in Fig 3-15). It appeared that our MFCC calculation was considerably more noisy (though maintaining the same average trends), possibly because of numerical precision issues. These deviations caused a 2% increase in speaker ID error for our large ternary FCN model. We were unable to isolate the reason for the difference, but to accommodate the slight difference, we retrained our ternary large-FCN model on features extracted using our implementation. This resulted in a model with identical accuracy to the original feature extractor/model combination.

3.5 Speaker Verification System: Implementation and Demo Setup

In our final section, we describe the complete speaker verification system, and the steps behind the final demonstration available for viewing at `https://youtu.be/n1wdkIRCnrU`. Figure 3-16 shows the system setup (host, FPGA, microphone, and display) and the previously linked video summarizes the system behavior and showcases an example of speaker verification with the author.

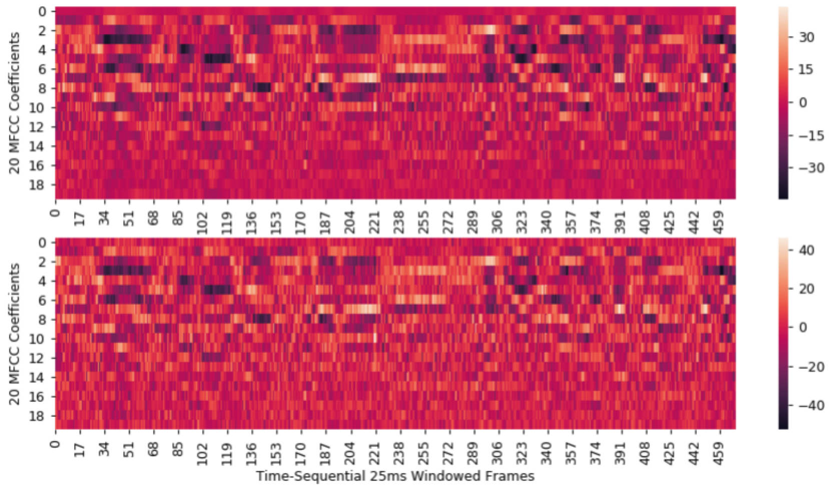
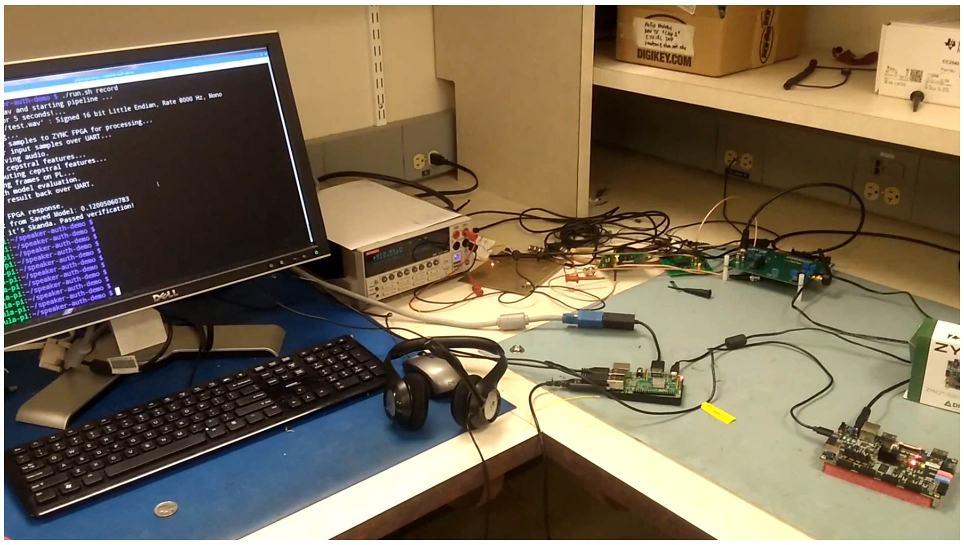


Figure 3-15: Subtle feature differences in the Kaldi standard implementation (top) and our bare metal implementation (bottom) on a sample utterance segment. Retraining our models using our extractor features resulted in no loss in accuracy, and allowed us to run feature extraction on the Xilinx development board.



```
r-auth-demo $ ./run.sh record
av and starting pipeline ...
r 5 seconds!...
/test.wav' : Signed 16 bit Little Endian, Rate 8000 Hz, Mono
g...
 samples to ZYNC FPGA for processing...
 input samples over UART...
ving audio.
 cepstral features...
uting cepstral features...
ng frames on PL...
h model evaluation.
 result back over UART.

 FPGA response.
 from Saved Model: 0.12005060783
 it's Skanda. Passed verification!
:~/speaker-auth-demo $
i:~/speaker-auth-demo $
pi:~/speaker-auth-demo $
pi:~/speaker-auth-demo $
-pi:~/speaker-auth-demo $
-pi:~/speaker-auth-demo $
a-pi:~/speaker-auth-demo $
a-pi:~/speaker-auth-demo $
la-pi:~/speaker-auth-demo $
la-pi:~/speaker-auth-demo $
ula-pi:~/speaker-auth-demo $
```

Figure 3-16: The speaker verification demonstration system. A Zync FPGA (rightmost board) communicates over serial to a host (Raspberry Pi, center) which displays the results and receives audio input.

On start-up, the system has a basic board and BRAM initialization:

1. Configure the accelerator design onto the FPGA fabric over USB using Vivado.
2. Load into BRAM a hex-file encoding all the layer-wise parameters of our model (Vivado does this simultaneous to flashing of the main design)
3. Program the ARM core on the development board with the C-based feature extractor over JTAG.
4. Switch the UART/USB cable from the machine running Vivado to the system host (Raspberry Pi), to begin speaker evaluation using the host-connected microphone. We now have the loop: FPGA to Raspberry Pi to Monitor and Microphone as shown in Fig 3-1.

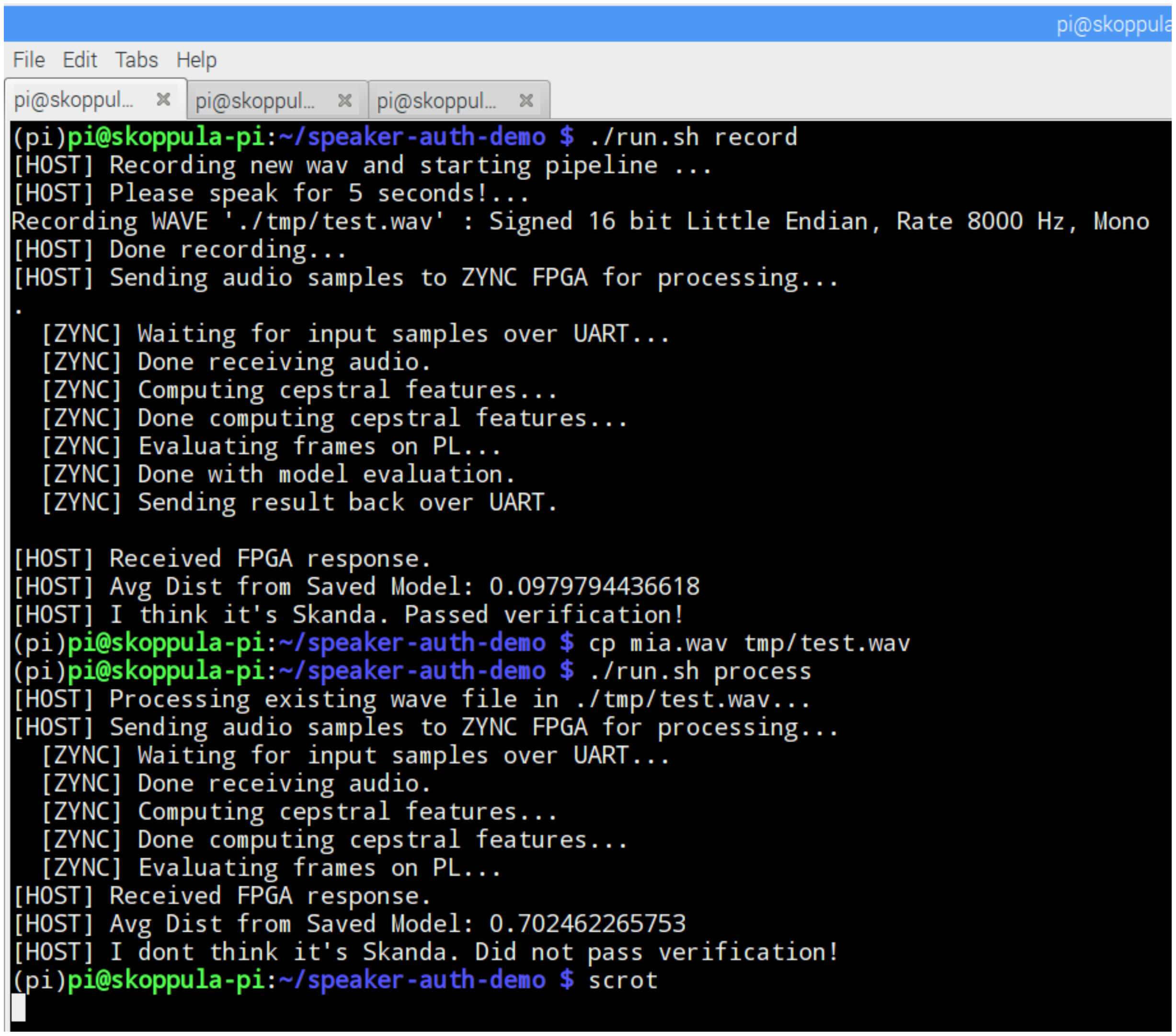
After initialization, the system begins its basic run-time evaluation control loop:

- If the FPGA receives the 'Overwrite Model' UART control signal, the system begins accepting bytes over UART to overwrite the stored BRAM model parameters. This feature is only to debug, and adapt the system for new speakers, and can be disabled. Overwriting the model takes roughly 20-40 seconds.
- If the FPGA receives the 'Evaluate' UART control signal, the FPGA begins accepting bytes over UART and saving the incoming MFCC frames into BRAM. When 20 MFCC frames have been received (400 inputs/1.6 KB in total), evaluation is triggered and returns back over UART the model response.

  Model evaluation is triggered 40 more times (on overlapping MFCC frame collections in the utterance) before the average speaker identification decision is returned. The total latency for classification, from the time the first audio samples are received at the Zync Core to the final speaker ID decision is 1.8 seconds. Broken down into roughly 0.2 seconds for feature extraction, and 40ms per model evaluation (this includes time required for communication of incoming features over UART)

3.5.1 Terminal Interface to Host and FPGA

The interface to the system is a simple terminal application that runs on the host and sends serial/UART messages to the Xilinx development board. The application uses the ‘arecord‘ Linux tool to pipe audio samples from a USB microphone to its own memory, and standard Linux tooling (‘cat‘, ‘echo‘, and ‘</>‘) to communicate over serial. Figure 3-17 illustrates the interface in use, and an example of positive verification and negative verification.



```
(pi)pi@skoppula-pi:~/speaker-auth-demo $ ./run.sh record
[HOST] Recording new wav and starting pipeline ...
[HOST] Please speak for 5 seconds!...
Recording WAVE './tmp/test.wav' : Signed 16 bit Little Endian, Rate 8000 Hz, Mono
[HOST] Done recording...
[HOST] Sending audio samples to ZYNC FPGA for processing...
.
  [ZYNC] Waiting for input samples over UART...
  [ZYNC] Done receiving audio.
  [ZYNC] Computing cepstral features...
  [ZYNC] Done computing cepstral features...
  [ZYNC] Evaluating frames on PL...
  [ZYNC] Done with model evaluation.
  [ZYNC] Sending result back over UART.

[HOST] Received FPGA response.
[HOST] Avg Dist from Saved Model: 0.0979794436618
[HOST] I think it's Skanda. Passed verification!
(pi)pi@skoppula-pi:~/speaker-auth-demo $ cp mia.wav tmp/test.wav
(pi)pi@skoppula-pi:~/speaker-auth-demo $ ./run.sh process
[HOST] Processing existing wave file in ./tmp/test.wav...
[HOST] Sending audio samples to ZYNC FPGA for processing...
  [ZYNC] Waiting for input samples over UART...
  [ZYNC] Done receiving audio.
  [ZYNC] Computing cepstral features...
  [ZYNC] Done computing cepstral features...
  [ZYNC] Evaluating frames on PL...
[HOST] Received FPGA response.
[HOST] Avg Dist from Saved Model: 0.702462265753
[HOST] I dont think it's Skanda. Did not pass verification!
(pi)pi@skoppula-pi:~/speaker-auth-demo $ scrot
```

Figure 3-17: Interface to the speaker verification demonstration system. This front-end application is used to initiate evaluation, record utterances, and view progress of the system (both the host and FPGA serial print-outs).

Chapter 4

Conclusions

4.1 Summary

In this thesis, we demonstrated an approach for text-independent speaker identification useful for evaluation on commodity FPGAs and other resource-constrained hardware.

The first chapter of this work explored techniques that reduced the bytesize of existing speaker ID models by >85%, tolerating a ±3% accuracy degradation. In particular, we demonstrated three methods of compression of our SID models via dropping into low-precision: manual linear quantization (Section 2.3.1), 0-1 Fixed Point Re-Training (Section 2.3.2), and Trained Ternary Quantization (Section 2.4). We introduced a constructed metric that balances the competing demands in a resource-constrained setting: (1) memory, (2) operations/evaluation, and (3) accuracy. With respect to this metric, we find (1) manual linear quantization not particularly effective compared to baseline, (2) significant improvements to the metric with 0-1 Fixed-Point Re-Training, and (3) the best results with Trained Ternary Quantization. We also explore model pruning, which we find to have moderate effectiveness in further reducing the model size. To the best of our knowledge, this is the first set of speaker identification model sized to fit in the on-chip memory of commodity FPGAs.

In the second chapter of this work, we build an RTL design for evaluation of our speaker ID models. In particular, we design, implement, and benchmark architectures for a low-precision fixed point neural network accelerator (Section 3.1). We present another design that is specific for ternary network evaluation, that improves on the fixed-point accelerator in power and area (Section 3.3). Compared to a baseline network full-precision accelerator with the same timing constraints, our low-precision, sparsity-cognizant design decreases LUT/FF resource utilization by 27% and power consumption by 12% in simulation.

We are excited by the possible applications of this work to the growing number of speech systems run on today's consumer devices and data centers.

The code, experiment logs, and a demonstration video for this thesis is available at `https://skoppula.github.io/thesis.html`.

4.2 Future Directions

Building on the work of this thesis, there are a number of research directions that would be exciting to explore in the future. The breadth of work in this thesis has yielded many opportunities that we identify for possible areas for work:

- Folding of the feature extraction and UART communication into the programmable logic/RTL designs. This would remove the need for the Cortex A9 on the Zybo development board, and by turning off that core, we could achieve pretty drastic power reductions.

- Finish demonstration of evaluation of convolutional models on our accelerator designs (in particular, on the fly Toeplitz transformation). This is work that was ongoing at the time of thesis completion, but was abandoned to complete other experiments.

- State-of-art speaker identification models as of 2018 (that are extremely large), are tending towards recurrent networks, and it would be interesting to explore applying our quantization approaches to these newer models. It is possible that new quantization approaches might need to be developed to successfully quantize the gates inside the recurrent kernels.

- Adding data compression/de-compression modules to the RTL design. This is a trick used in Eyeriss to reduce off-chip data bandwidth, which we could employ to reduce on-chip memory usage.

- Comparison of our quantization techniques with recent techniques for binarizing neural networks, and a comparison of performance benchmarks with accelerators for binary neural accelerator (e.g. YodaNN, TrueNorth) from Andri et al. [2016]. These designs are ASICs, so a fair comparison would require either implementing our design as an ASIC, or re-implementing their designs on our FPGA.

  We chose ternary networks in this work (as compared to binary networks), because at the time of our experiments, ternary networks had comparable accuracy to full-precision networks on computer vision tasks, but state-of-art binary networks showed a 5-10% accuracy degradation. This likely has changed in the past year, with new techniques for binarization.

Appendix A

Additional Experimental Results for Chapter 2

Tables A.1, A.2, and A.3 describe performance on the baseline network architectures on the RSR and SRE datasets with various configurations of batch normalization/no batch normalization, and regularization/no regularization.

Figures A-1 and A-2 describe the accuracy changes and absolute magnitude weight differences introduced after quantizing each of the layers in our CNN SID model.

Table A.3 describes the bias explosion when using manual fixed point question in Section 2.3.1.

Finally, Figure A-3 describes speaker identification error as we induce higher levels of sparsity by pruning the model.

Model	Error (%)	Mults	Parameters	P-Score
Fully Connected, Small	81.93	517K	519K	11.343
Fully Connected, Large	79.77	1394K	1396K	12.191
Convolutional	61.22	263K	258K	10.621
Locally Connected	61.57	274K	286K	10.685
Maxout, Small	74.78	1813K	1818K	12.391
Maxout, Large	92.20	2127K	2130K	12.621
Depth-Seperable Conv., Small	94.86	1233K	1221K	12.155
Depth-Seperable Conv., Large	32.20	360K	332K	10.586

Table A.1: Speaker Identification Error of Baseline Models on RSR2015, using no batch norm and no regularization. While we were unable to find prior work that has tried trimming both batch norm and regularization from the basic SID models, for reference and to provide a comparison point: with batch normalization and regularization, a fully-connected large network from Bhattacharya et al. [2016] is able to achieve 2.8% error. This matches the results from Table 2.1.

Model	Error (%)	Mults	Parameters	P-Score
Fully Connected, Small	10.85	519K	520K	10.466
Fully Connected, Large	03.93	1399K	1399K	10.886
Convolutional	14.53	266K	260K	9.991
Locally Connected	13.10	276K	287K	10.017
Maxout, Small	39.93	1821K	1822K	12.121
Maxout, Large	22.92	2134K	2133K	12.017
Depth-Seperable Conv., Large	14.36	1245K	1227K	11.335
Depth-Seperable Conv., Small	28.48	368K	335K	10.532

Table A.2: Speaker Identification Error of Baseline Models on RSR2015, using batch norm and no regularization. This roughly matches the referenced performance in prior work (Tashev and Mirsamadi [2016] for the large FCN), and our own results with batch normalization and regularized models (Table 2.1)

Model	Error (%)	Mults	Parameters	P-Score
Fully Connected, Small	26.43	519K	520K	10.929
Fully Connected, Large	23.07	1399K	1399K	11.710
Convolutional	27.00	266K	260K	10.413
Locally Connected	26.35	276K	287K	10.321
Maxout, Small	46.97	1821K	1822K	12.213
Maxout, Large	39.85	2134K	2133K	12.295
Depth-Seperable Conv., Large	43.86	1245K	1227K	11.853
Depth-Seperable Conv., Small	97.05	368K	335K	10.532

Table A.3: Speaker Identification Error of Baseline Models on SRE10, using no batch norm and no regularization. For reference, state of art on the same testing split of SRE10 is 7.6% equal error rate, using a much larger network (>20MB) Greenberg et al. [2011]. Note that this number is the SRE10 benchmark on an open-set, not closed-set, speaker ID task – so is not directly comparable, even though the number of speakers in the task remain the same.

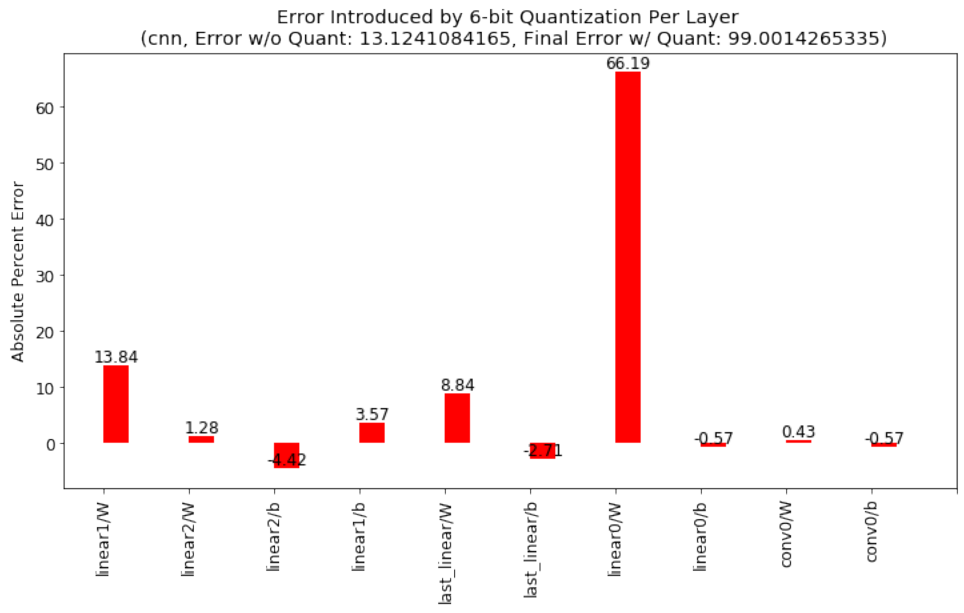


Figure A-1: Increase in error after quantizing each layer in the convolutional SID network (30-bit min-max linear quantization).

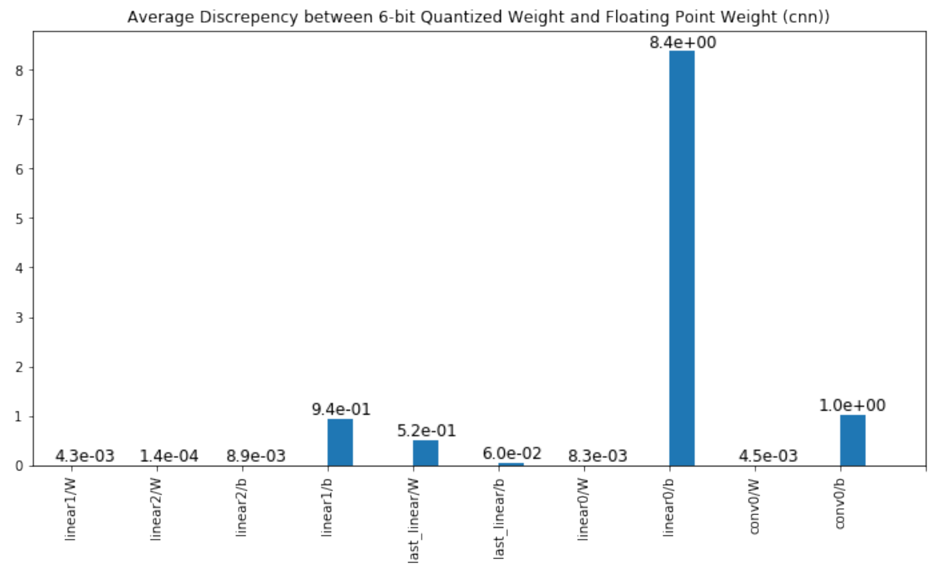


Figure A-2: Maximum magnitude of discrepency between the quantized and non-quantized parameters of each layer in the convolutional SID network (30-bit min-max linear quantization).

Layer	Max W	W Scale	Max Act.	Act. Scale	Max Bias	Post-Scale Max Bias
linear0	0.07580	3.814e-06	38.8238	0.0019531	62.347168	8368202349.18437
linear1	0.3336	1.525e-05	21.67454	0.0009765	56.698997	3805248814.53354
linear2	0.3490	1.525e-05	42.9360	0.0019531	48.980305	1643527350.61185
linear3	0.01165	4.768e-07	0.500426	0.000015271	1.4277624	196073261258.909

Table A.4: Factoring out the scale terms to perform arithmetic in 30-bit integers. The small weight scaling factors ('W Scale') and activation scaling factors ('Act. Scale') multiply to create an extremely small inverse scaling factor for the bias, causing extremely large biases, which are susceptible to overflow

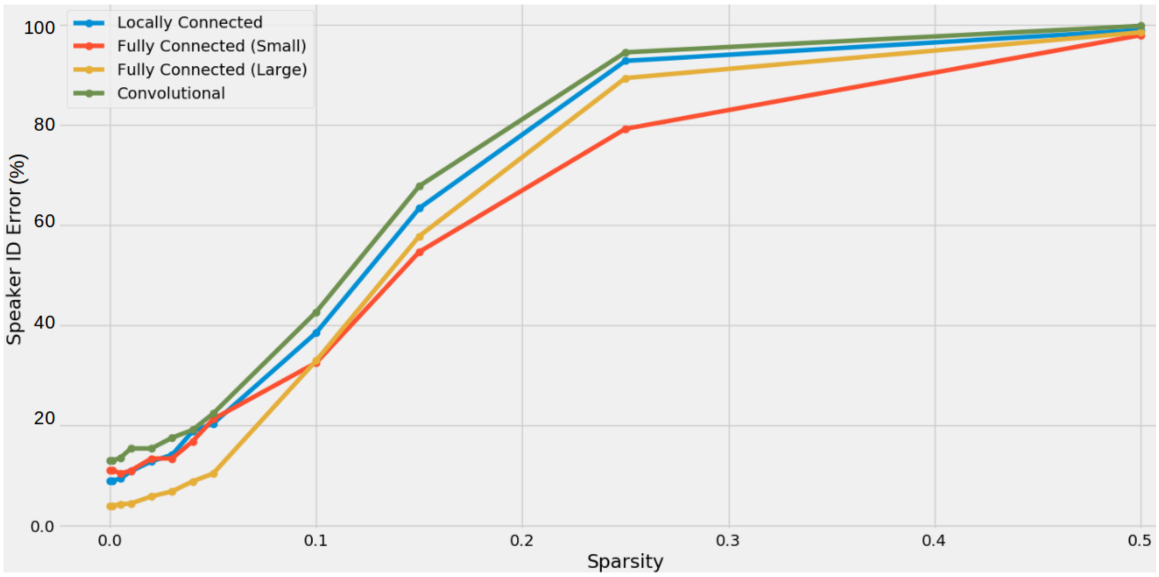


Figure A-3: Speaker ID Error vs. Induced Sparsity into the 16-bit [0,1]-FxPt model.

Appendix B

Additional Experimental Results for Chapter 3

Figure B-1 shows the SoC elements available on the Digilent Zybo Z7 development board used in this work.

Figures B-3, B-4, and B-5 describe power reports for the variants of the fixed-point accelerator design.

Finally, Figure B-2 describes our sample Vivado Block Design used to test functionality of BRAM on the Zybo Z7 and acquaintance ourselves with the AXI4 BRAM interface.

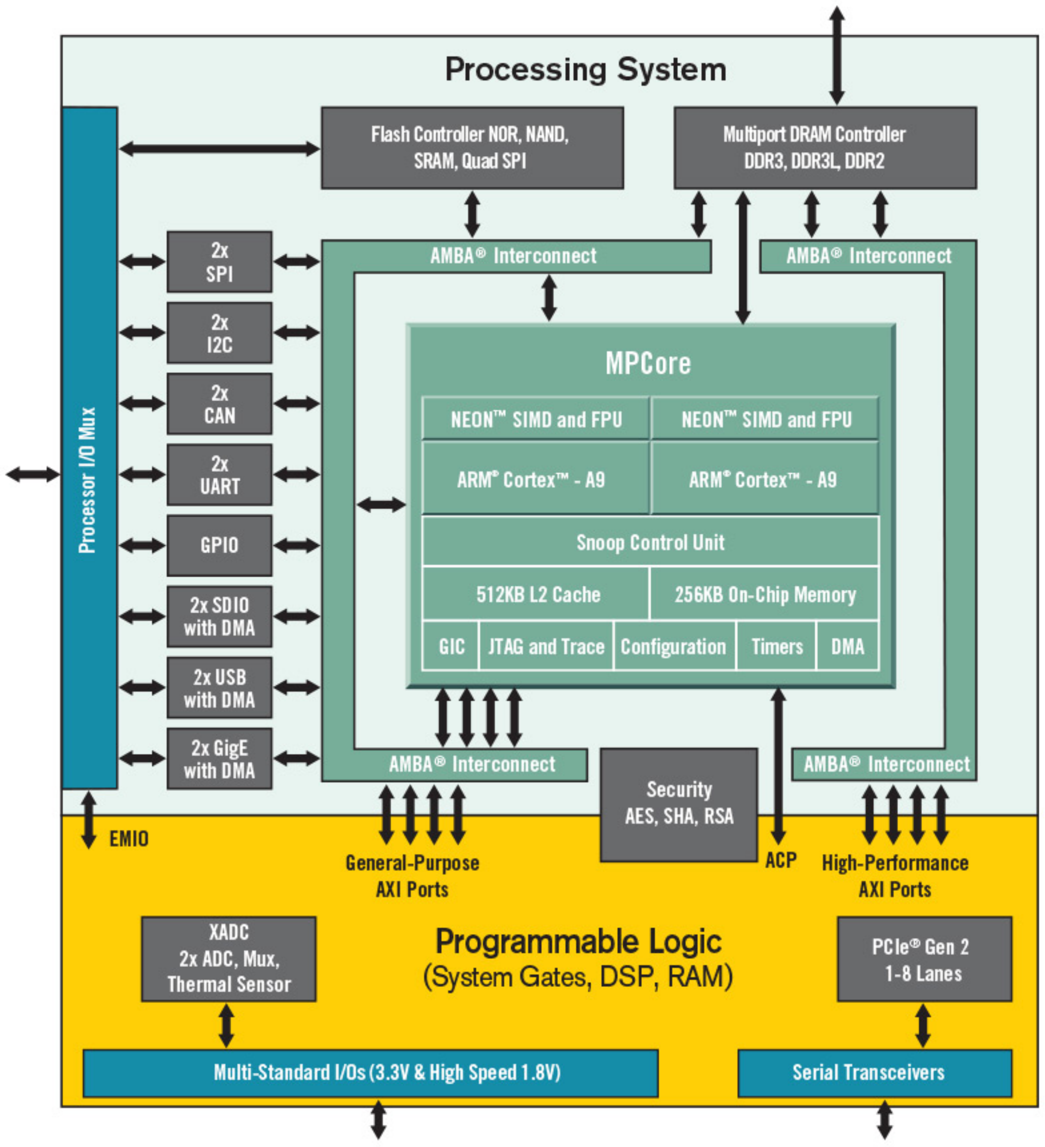


Figure B-1: Listing of the peripherals on the Xilinx XC7Z020-1CLG400C programmable logic chip. Diagram is from Xilinx Series-7 datasheets Xilinx [2018]

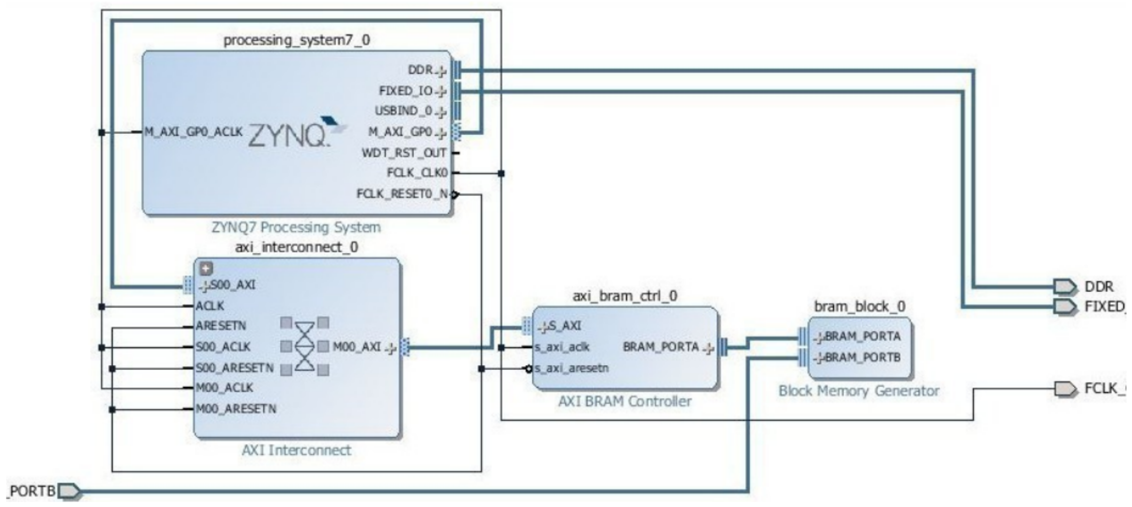


Figure B-2: Simple design created in Vivado to develop familiarity with Xilinx's AXI communication protocol/BRAM usage, test functionality of BRAM on our development board, and make preliminary measurements on the time required to read and write model parameters/outputs to and from BRAM.

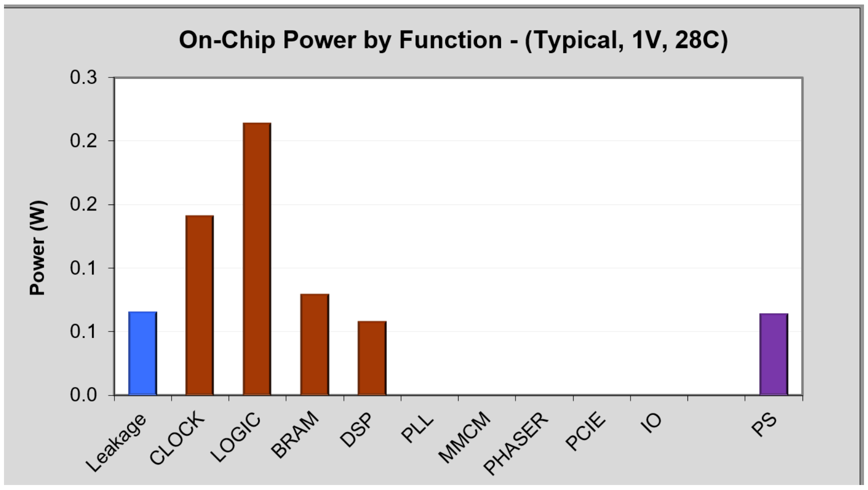


Figure B-3: Breakdown report for the $n = 256$, 32-bit parameter design. It follows the same distribution as the other design variants.

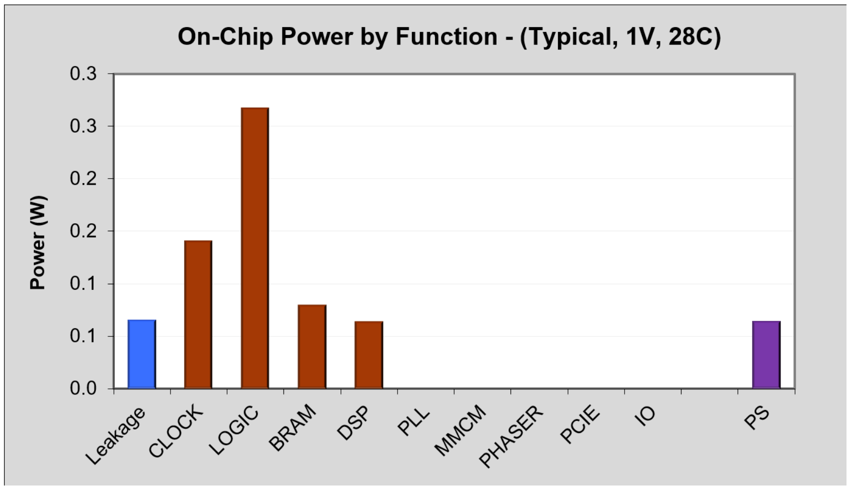


Figure B-4: Breakdown report for the $n = 512$, 8-bit parameter design. It follows the same distribution as the other design variants.

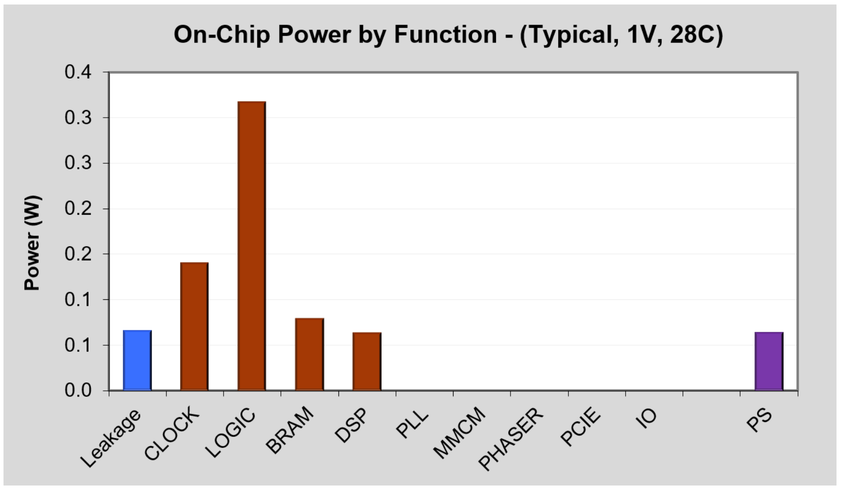


Figure B-5: Summary report for the $n = 512$, 32-bit parameter design. This design has the highest estimated power consumption of all designs, by up to 24%

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 236–241. IEEE, 2016.

BBC. Amazon Echos activated by TV comment. *BBC News*, Jan 2017. URL `http://www.bbc.com/news/technology-38553643`.

Gautam Bhattacharya, Jahangir Alam, Themos Stafylakis, and Patrick Kenny. Deep Neural Network based Text-Dependent Speaker Recognition: Preliminary Results. *Odyssey 2016*, 2016.

William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 4960–4964. IEEE, 2016.

Yu-hsin Chen, Ignacio Lopez-Moreno, Tara N Sainath, Mirkó Visontai, Raziel Alvarez, and Carolina Parada. Locally-connected and convolutional neural networks for small footprint speaker recognition. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

Phaklen Ehkan, Timothy Allen, and Steven F. Quigley. FPGA Implementation for GMM-Based Speaker Identification. *International Journal of Reconfigurable Computing*, 2011:1âĂŞ8, 2011. doi: 10.1155/2011/420369.

Ekapol Chuangsuwanich and James Glass. Robust Voice Activity Detector for Real World Applications Using Harmonicity and Modulation frequency, 2011.

Brian Feldman. NPR Segment on Amazon Echo Accidentally Activates Robot Army of Speakers, url=http://nymag.com/selectall/2016/03/npr-segment-on-amazon-echo-messes-with-devices.html, Mar 2016.

Craig S Greenberg, Alvin F Martin, Linda Brandschain, Joseph P Campbell, Christopher Cieri, George R Doddington, and John J Godfrey. Human Assisted Speaker Recognition In NIST SRE10. In *Odyssey*, page 32, 2010.

Craig S Greenberg, Alvin F Martin, Bradford N Barr, and George R Doddington. Report on performance results in the nist 2010 speaker recognition evaluation. In *Twelfth Annual Conference of the International Speech Communication Association*, 2011.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

Iowa Hills. Fft algorithm and spectral analysis windows. `http://www.iowahills.com/FFTCode.html`, 2013.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

Phak Kan, Tim Allen, and Steven Quigley. A GMM-based speaker identification system on FPGA. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 358–363, 2010.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Anthony Larcher, Kong Aik Lee, Bin Ma, and Haizhou Li. Text-dependent speaker verification: Classifiers, databases and RSR2015. *Speech Communication*, 60:56–77, 2014.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

Yun Lei, Nicolas Scheffer, Luciana Ferrer, and Mitchell McLaren. A novel scheme for speaker recognition using a phonetically-aware deep neural network. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 1695–1699. IEEE, 2014.

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

Leo Liu et al. *Acoustic models for speech recognition using Deep Neural Networks based on approximate math.* PhD thesis, Massachusetts Institute of Technology, 2015.

Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 273–282. ACM, 2013.

Liang Lu, Michelle Guo, and Steve Renals. Knowledge distillation for small-footprint highway networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 4820–4824. IEEE, 2017.

Jinhwan Park and Wonyong Sung. FPGA based implementation of deep neural networks using on-chip memory only. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 1011–1015. IEEE, 2016.

Dan Povey. Kaldi MFCC generation. `https://github.com/kaldi-asr/kaldi/blob/master/egs/sre10/v2/conf/mfcc.conf`, 2017.

Dan Povey and Kaldi Contributors. Kaldi requirements. `http://kaldi-asr.org/doc/dependencies.html`, 2018.

Daniel Povey, Stephen M Chu, and Balakrishnan Varadarajan. Universal background model based speech recognition. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 4561–4564. IEEE, 2008.

Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The Kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*. IEEE Signal Processing Society, 2011.

Kishore Prahallad. Topic: Spectrogram, Cepstrum and Mel-Frequency Analysis, 2011. URL `http://www.speech.cs.cmu.edu/15-492/slides/03_mfcc.pdf`.

Michael Price, James Glass, and Anantha P Chandrakasan. A 6 mw, 5,000-word real-time speech recognizer using wfst models. *IEEE Journal of Solid-State Circuits*, 50 (1):102–112, 2015.

Michael Price, Anantha Chandrakasan, and James R Glass. Memory-efficient modeling and search techniques for hardware asr decoders. In *Interspeech*, pages 1893–1897, 2016a.

Michael Price, James Glass, and Anantha P. Chandrakasan. A scalable speech recognizer with deep-neural-network acoustic models and voice-activated power gating. *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017. doi: 10.1109/isscc.2017.7870352.

Michael Price et al. *Energy-scalable speech recognition circuits*. PhD thesis, Massachusetts Institute of Technology, 2016b.

Shawn Price. Radio broadcast hacks listeners' Amazon Echo devices, Mar 2016. URL `http://www.upi.com/Odd_News/2016/03/16/Radio-broadcast-hacks-listeners-Amazon-Echo-devices/8711458115390/`.

Rafael Ramos-Lara, Mariano López-García, Enrique Cantó-Navarro, and Luís Puente-Rodriguez. SVM speaker verification system based on a low-cost FPGA. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 582–586. IEEE, 2009.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

Douglas Reynolds. Universal background models. *Encyclopedia of biometrics*, pages 1547–1550, 2015.

Fred Richardson, Douglas Reynolds, and Najim Dehak. Deep neural network approaches to speaker and language recognition. *IEEE Signal Processing Letters*, 22 (10):1671–1675, 2015.

Gourav Sarkar and Goutam Saha. Real time implementation of speaker identification system with frame picking algorithm. *Procedia Computer Science*, 2:173–180, 2010.

Jeremy Sawruk. libmfcc - c library for computing mel-frequency cepstral coefficients. `https://github.com/jsawruk/libmfcc`, 2010.

Stephen Shum. Low-dimensional speech representation based on Factor Analysis and its applications, 2011.

David Snyder, Pegah Ghahremani, Daniel Povey, Daniel Garcia-Romero, Yishay Carmiel, and Sanjeev Khudanpur. Deep neural network-based speaker embeddings for end-to-end speaker verification. *Spoken Language Technology Workshop (SLT), 2016 IEEE*, 2016.

Ivan Tashev and Seyedmahdad Mirsamadi. DNN-based Causal Voice Activity Detector. In *Information Theory and Applications Workshop*, 2016.

Amirsina Torfi, Nasser M Nasrabadi, and Jeremy Dawson. Text-Independent Speaker Verification Using 3D Convolutional Neural Networks. *arXiv preprint arXiv:1705.09422*, 2017.

Ehsan Variani, Xin Lei, Erik McDermott, Ignacio Lopez Moreno, and Javier Gonzalez-Dominguez. Deep neural networks for small footprint text-dependent speaker verification. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 4052–4056. IEEE, 2014.

Wikipedia contributors. Sparse matrix — Wikipedia, the free encyclopedia, 2018. URL `https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=836271982`. [Online; accessed 23-April-2018].

Yuxin Wu et al. Tensorpack. `https://github.com/tensorpack/`, 2016.

Aaron Xichen. Quantization in pytorch. `https://github.com/aaron-xichen/pytorch-playground/blob/master/utee/quant.py`, 2017.

Xilinx. 7 series fpgas memory resources, 2018. URL `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.

Shi-Xiong Zhang, Zhuo Chen, Yong Zhao, Jinyu Li, and Yifan Gong. End-to-End Attention based Text-Dependent Speaker Verification. *arXiv preprint arXiv:1701.00562*, 2017.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.