DeepGate: Hardware-Accelerated Speech Recognition System

Lindsay Davis, Estella Gong, Michael Lopez-Brau, and Cedric Orban

Department of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — This paper presents the design and implementation of DeepGate, a feedforward, deep neural network (DNN) on a low-cost field-programmable gate array (FPGA) for speech classification of a small vocabulary. We discuss our algorithmic approach as well as the limitations that are incurred from using a low-cost FPGA. These limitations are fleshed out in greater detail by an analysis of the firmware programming on the FPGA. Furthermore, we discuss the classifier as a hardware-accelerated DNN, jointly powered by the FPGA and off-board processor. Lastly, we review the printed circuit board (PCB) that we developed to house the FPGA.**

***Index Terms* — FPGA, hardware acceleration, neural network, speech recognition.**

I. INTRODUCTION

People have the remarkable ability to make vast inferences about the world with little information. A prime example of this is language: how are humans able to effectively use language, an unfathomably flexible and vague form of communication, to develop and exchange ideas? Is it possible to train machines to communicate or at least understand the vagueness of natural language? These questions and many like them have piqued the interest of many linguists, computer scientists, and engineers, leading to research thrusts in computational linguistics and natural language processing.

At the intersection of these two fields, numerous algorithms and heuristics have been developed in the search to find the one most optimal for practical speech recognition. Recent advances in hardware have caused a resurgence in the development of deep neural network (DNN) modeling, now commonly known as *deep learning*. In the past, neural network models with many hidden layers were unfeasible due to their computational complexity and their need for data. Today, many of the most common deep learning algorithms can be run on a middle-to-high end laptop. These algorithms have become very popular as of late due to their ability to dominate other machine learning algorithms on almost every benchmark. We have noticed that these algorithms also perform exceptionally well in the language domain, particularly in speech recognition.

We designed a system that mirrors current research thrusts by implementing this algorithm on a low-cost FPGA. FPGAs have the advantage of being significantly cheaper than ASIC variants for small-scale applications. They also have the ability to run a DNN faster than a CPU and with less power consumption than a GPU.

In Section II, we will begin by discussing the design of the speech classifier, including the dataset we used, data preprocessing, and the parameterization of the DNN. In Section III, we review the graphical user interface (GUI) that the users will use to interact with the system. In Section IV, we move forward to discussing the implementation of the DNN on the FPGA and the steps we took to address our algorithmic constraints. Lastly, Section V discusses the hardware design and we wrap up with a conclusion in Section VI.

II. SYSTEM OVERVIEW

The DeepGate Speech Recognition system is the combination of the implementation of a deep learning approach on the FPGA chips. The goal of DeepGate is to create an affordable, energy-efficient, low-cost, compact speech recognition system that utilizes current research. Input consists of audio data which will be received through a user interface. This allows for the user to easily interact with the system. This FPGA implementation of a neural network will contain a small vocabulary of pre-registered words that the user will be able get started with.

DeepGate consists of several parts: data acquisition, data pre-processing, recognition/decoding, and an application interface. This in turn makes up our Hardware and Software Architecture. The high level diagram is depicted as Figure 1.

A. Data Acquisition Stage

The data acquisition is the audio recording of voice files through a microphone. These will be sampled at a rate of 16,000 Hz and saved in the WAV file format. These audio files will allow us to get the user get started with the pre-registered vocabulary with includes integers from 0 to 9, as well as the cardinal directions, north, south, east, and west.

B. Data Pre-Processing Stage

The data pre-processing is accomplished through an algorithm that will be run on our processor, in this case a raspberry pi. This includes some filtering as well as possible digital signal processing. With this stage, the data is prepared for its utility in the speech recognition.

C. Recognition-Decoding Stage

The recognition/decoding stage is the implementation of the deep neural network model on the FPGA chip. We also have a PCB chip that is designed for the FPGA as well as the additional peripherals we have. This stage will run through the algorithms and identify perceived matches within the vocabulary set.

C. Application Stage

Finally, we have an application that user can interact with to perform the speech recognition. This will be the graphical user interface that runs from the raspberry pi. Tis application allows the user to run training algorithms on recorded audio data. Output with regards to the speech recognition will be displayed to the user via the graphical user interface. In addition, the FPGA LEDs will also provide user feedback.

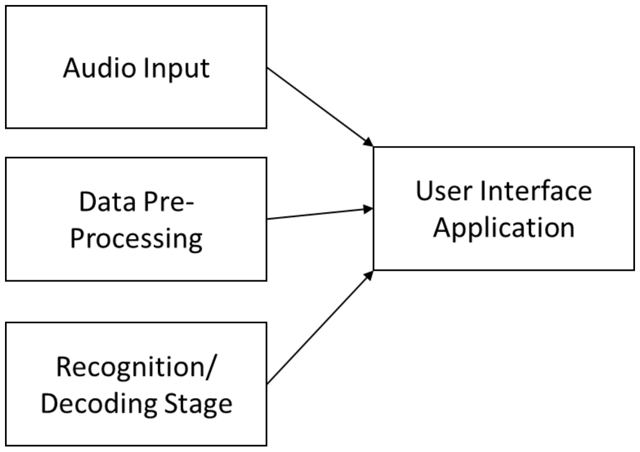

Figure 1: High – Level System Overview

II. Speech Recognition

Our speech recognition system involves a pipeline that begins with a preprocessing stage, a training stage, and a testing stage. The preprocessing stage and training stages will be performed on a Raspberry Pi 3. The results from these stages will be sent to the FPGA, where the testing stage will then be conducted. This pipeline will be discussed in greater detail later.

A. Preprocessing

The preprocessing stage is essential in order to get our inputs in the ideal format for our classifier. Just as the semantics of natural language can be finicky, so can comparing speech signals: even from the same individual. Speech signals will be initially sampled via a microphone at 16,000 Hz and stored in a WAV file. We chose our sampling rate based on the fact that 16,000 Hz captures all of the details we might want about a particular utterance, while avoiding the extra computations that would come with oversampling (i.e., having more samples per signal). The WAV format was a natural choice for us as it is an uncompressed file format for audio signals.

Our dataset currently consists of six people's speech sets. Each speech set contain five more subsets, where the each subset represents that individual saying every word in our vocabulary once. The data is collected using the sampling rate and file format mentioned above. We are always expanding the dataset to increase the generalizability of our classifier.

To account for any noise that occurs during initialization, we scan through the speech signal and look for the first sample that is greater than or equal to approximately 40% of the maximum value of the absolute value of the signal. We set this point as the starting point and take the next 6999 samples per signal. This heuristic is necessary in order to achieve a simple, automatic preprocessing routine that can grant us good classification results during testing.

Once we have trimmed our raw signals to 7000 samples, we partition each signal into a collection of frames and compute the Mel-frequency cepstrum coefficients (MFCC) for each frame. MFCCs are features widely used in speech recognition. They are extracted from audio signals, with the goal of mimicking certain parts of human speech perception. In particular, MFCC features mimic the logarithmic perception of loudness and pitch of human auditory systems through the use of the Mel scale, which relates a perceived frequency with its actual frequency.

For compatibility with our classifier, we scaled the MFCCs so that they fit between -1 and 1, divided by 2, and then shifted by half so that our feature vector lies between 0 and 1. This reduces the chances of the DNN having saturated inputs into any of the hidden or output nodes, which are detrimental for learning.

B. Speech Classifier

As mentioned previously, our speech classifier is a feedforward DNN tasked with classifying a vocabulary of 14 words. The vocabulary consists of the numbers 0-9 and the cardinal directions: east, north, south, and west. There are several other powerful choices that are often employed in speech recognition systems, such as hidden Markov models (HMM) and recurrent neural networks (RNN). These algorithms tend to be quite sophisticated in terms of implementation and require computational resources that we do not have with our hardware. Consequently, we chose to go with a feedforward DNN for our application.

The network has 516 nodes at the input layer (to resemble the size of our feature vector), 100 nodes at the first hidden layer, 50 nodes at the second hidden layer, and 14 nodes at the output layer (to resemble the size of our vocabulary). For training, the network uses the sigmoid function as the activation function and gradient descent with weights initialized from sampling a normal distribution with parameters dependent on the number of hidden layers in the network. The network does not currently have a regularization term.

In desktop and laptop processors, floating-point precision is rarely a concern. Because we plan to implement this network in a low-cost FPGA, we are limited by the amount of block memory and, hence, are restricted in the amount of precision we can utilize. Our nodes will be unsigned and encoded using 8 bits and our weights will be signed and encoded using 3 to 5 bits. The sigmoid function will also have to be approximated by using a combinational approximation, which we will discuss in greater detail in Section IV. Figure 1 demonstrates the tight fit that the combinational approximation has with the standard sigmoid function. Empirical tests show that the error induced by the approximation is minimal: significantly less than 1%. Our learning algorithm (i.e., gradient descent) is unaffected since the training will be off-loaded to the Raspberry Pi 3.

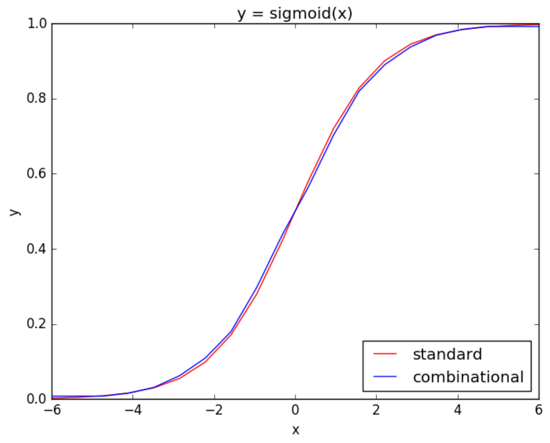

Figure 1: Comparison between a sigmoid function with 16-bit precision with its combinational approximation.

C. Classification

With an understanding of the preprocessing steps and the speech classifier, we can now take a bird's eye view of the classification process. In particular, we will split up our discussion into a training phase and a testing phase, according to where each phase is processed.

During the training phase, we perform all of our computations on the Raspberry Pi 3. We first preprocess all of the raw signals in our dataset. This includes trimming, framing, and computing the MFCC feature vector for every signal. We then instantiate an instance of the DNN to initialize the weights, set the learning rate, and set the number of epochs to iterate through. Iterating through the training set multiple time via epoch iteration is useful for "generating" additional training data for the network to learn from. Though this can cause the network to overfit the training data, we have tested that 100 epochs is generally sufficient to obtain the classification results we want (greater than 80%) without overfitting.

During the testing phase, we move our computations over to the FPGA. First, we approximate the weights generated on the Raspberry Pi 3 to values that fall within the precision available with 3 to 5 bits. These weights are then exported to a text file, converted to an appropriate format for the FPGA to read, and then sent to the FPGA via serial peripheral interface (SPI). At this point, the system switches to testing mode. Any speech signals that are recorded from the mic are converted to an appropriate format for the FPGA on the Raspberry Pi 3 and then sent over for processing. The FPGA will then communicate back to the Raspberry Pi 3 with the classification results.

III. Graphical User Interface

The graphical user interface is how the user can easily interact with our system. Much of the functions that we need for our speech recognition system involve different programs that can be run on the command line. By using a graphical user interface, we have a central application that encompasses all the functionality of the desired programs.

This application will allow for audio input and features buttons that have various functionalities. These include audio processing and training. Other widgets will be used to output text and results to the user.

Each of the functionalities on the graphical user interface application demonstrate successful interfacing with the algorithms and hardware. The user interface allows the users to connect to the hardware via the SPI bus. This in turn, allows data to be sent and received to the raspberry pi.

We chose to run the application on the raspberry pi with a Debian Jesse operating system. This Linux based operating system allows for development to be done conveniently from a Linux Virtual Machine running Ubuntu.

The graphical user interface is developed using Qt Creator and PyQt. This gave us access to libraries that would assist in the design of our graphical user interface. The advantage of using Python is that it allows us to easily cross compile.

Our vision for the system is to be able to record audio data of the user speaking any of the integers from 0 – 9 or cardinal directions, north south, east, or west. This will be achieved through buttons on the user interface. The user will press record to start recording and stop to end and save the recording. In addition, the user interface also allows for the opening of a WAV audio file. This file will then still be able to undergo the same processing.
After pre-processing the data, the user can click on a button that will trigger the deep learning algorithm to run on the data on the FPGA chip. This feedback would then be sent back over the SPI bus and communicated through a display text box on the user interface. The FPGA will also be able to physically indicate a successful connection through the LEDs on the board.

IV. Hardware Architecture

The design of the FPGA-based artificial neural network core is based on the work presented in [1C]. We expand on the previous work by allowing the synthetization of register-transfer level feed-forward neural networks of arbitrary layer and node counts. These parameters can be changed solely by modifying values in a global include file, though they must be determined before compilation. In addition to implementing arbitrarily sized networks, latency and logic resource utilization can be controlled by specifying how parallelized the forward-pass computation needs to be. Table [1] below depicts the processing time in a 516-100-50-14 feedforward network for various parallelization configurations. Each tile has a loop value associated with it that specifies how many nodes are calculated per processing unit. As seen in Table [x], the smaller the loop value, the shorter the processing time. This decrease comes at the cost of resource utilization (more processing units are needed).

Loop Parameters	**Latency**
loop (1, 10, 10, 2)	645 us
loop (1, 5, 5, 2)	335.9 us
loop (1, 1, 1, 1)	83.7 us

Table 1: Latency for Various Design Configurations

Along the same line, weight storage can be split between on-chip and off-chip RAM in any proportion to decrease latency (more on-chip weights) or reduce on-chip memory usage (more off-chip weights). Furthermore, we implement a combinational approximation of the sigmoid activation function with a higher precision than the function used in [1C], allowing us to achieve a greater accuracy when computing a forward pass through the network. Lastly, we determine the precision and range of the network weights that will best suit our needs, in addition to expanding the storage location of these weights to include an external SDRAM.

A master control module wraps the ANN core and allows for its use in a larger system by distributing weights from the SDRAM to the appropriate tiles and implementing an SPI interface to facilitate communication with an external entity. In addition to these major responsibilities, the control unit acts as a mediator between an SPI master and the off-chip SDRAM, allowing the forwarding of data from one to the other.

A. Processing Unit

The ANN's processing unit forms the design's basic execution unit. A processing unit is capable of taking an 8-bit input value and a 3 to 5-bit weight, multiplying the two, and adding the result to a sum stored in a register once every clock cycle. It uses fixed-point arithmetic with 8 fractional bits to stay consistent with the output of sig_368p.

It is designed to have as small of a logic resource utilization footprint as possible to allow for the inclusion of a large number in a single design. To this end, it is parameterized so that the RTL synthesis tool calculates optimal internal register widths based on the size of the neural network and the range of the discretized weights. Furthermore, to avoid synthesizing multipliers, multiplication is performed using shifts and additions, which turn out to be less costly when compared with the equivalent multiplier logic (compared using look-up-table utilization metrics).

A processing unit is used to calculate the output of the ANN's nodes. It can be reset to a default value, the node's bias value at any time, allowing it to be used to compute the output of an indefinite number of nodes in the same layer.

B. Combinational Approximation the Sigmoid Activation Function

Our network makes use of a sigmoid activation function implemented using an AND-OR array, a characteristic which lends itself well to reprogrammable logic devices that have a plethora of LUTs. Originally, we used the sig_337p module introduced in [2C] but found that, especially in larger networks, it was leading to very noticeable discrepancies between the FPGA's forward-pass outputs and the ideal outputs as computed by a software model of the ANN. To remedy this, we designed a higher precision approximation function following a similar procedure to the one used in determining sig_337p's AND-OR array.

We first mapped signed fixed-precision inputs (3 integer bits, 6 fractional bits) to their corresponding sigmoid function fixed-precision outputs (8 fractional bits) using a C++ program [3C]. This created 8 truth tables with 512 entries. Each truth table was then passed as input to a MATLAB implementation of the Quinne-McClusky logic minimization algorithm [3C]. Following a lengthy computation, the minimized Boolean equations mapping all 512 possible fixed-point inputs to the 8 fractional output bits were determined. A Verilog code generation program written in C++ was then used to convert the MATLAB output to usable HDL, thus creating the sig_368p sigmoid approximation module [3C].

Like sig_337p, sig_368p only takes positive input values, therefore the 2's complement of negative values must be taken before being passed through the AND-OR array. Additionally, the output value of the logic array must be subtracted from 1 if the input value is negative.

Performing a comparison using Altera's Quartus synthesis tool, sig_368p uses 115 LUTs whereas sig_337p makes use of 19 LUTs. While this difference is significant, it is an almost insignificant amount when compared to the total number of LUT resources available on most FPGAs.

Finally, we compared the forward-pass outputs of the network using both sigmoid approximation modules. In a dummy 516-100-50-14 network using randomly generated weights and 100 instances of randomly generated input data, sig_368p had an average discrepancy from the ideal output layer values of 0.003. In the same scenario, sig_337p had an average discrepancy of 0.05.

C. Tiles

A tile in the hardware-based ANN core is composed of several processing units, one sig_368p module, a block RAM, and a basic control unit state machine. Moreover, a tile is roughly equivalent to a neural network layer, the only major difference being that a tile's constituent processing units can be used to calculate the output of multiple nodes each.

The block RAM in each tile contains the activated data of the processing units in that tile. This block RAM also feeds directly into the inputs of the following layer, thus forming a buffer between layers of the network. Using this buffer allows multiple tiles to be pipelined; a tile simply has to write all of its outputs to the RAM before it can begin accepting inputs from the preceding RAM, rather than waiting for the following tile to complete processing.

In an ideal situation, this pipelining can halve the latency of a forward-pass through the network (latency is equal to processing time of one tile rather than two). In practice, however, since not all layers are the same size, latency through the full network is equal to the processing time of the largest tile in the ANN.

A simple FSM instructs the processing units to add-multiply, reset their registers, and transfer their data through sig_368p to the block RAM. This FSM sits idle unless there is input data available either from the input layer of the ANN or the block RAM of the preceding tile.

D. Weights

The characteristics of our network weights affect almost every aspect of our design, including logic resource utilization, memory usage, and latency. Early on we saw that several people had experienced success using small, low-precision weights. This influenced us to experiment with weights of size 3 to 5 bits and in the range of [-3, 3] as well as [-1.5, 1.5]. In all cases, we avoided instantiating multipliers in the processing units and performed multiplication using shifts and adds. For our purposes (speech recognition), we saw that 5-bit weights in the range of [-1.5, 1.5] gave the best classification accuracy as most weights in the trained network fell in this range. Table [2] below depicts logic resource utilization vs. weight width on a Spartan-6 LX9 FPGA.

Our overall system had an external SDRAM chip to store and read weights from, so running out of overall memory was a non-issue. However, running out of on-chip memory was a serious issue that affected the processing time of the ANN core, as its speed was bottlenecked by the SDRAM's read latency.

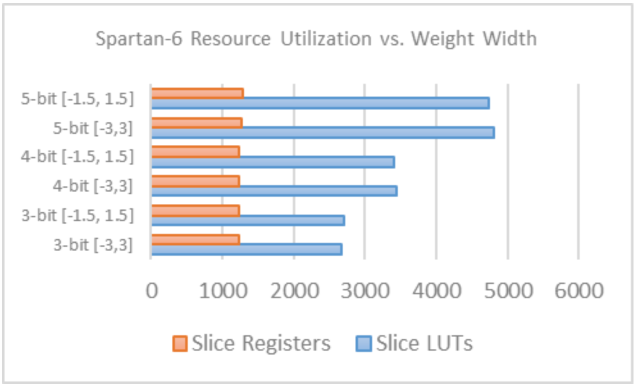

Table 2: Resource Utilization on the Spartan-6 FPGA with increasing precision of the weights.

E. Master Control Unit

The master control unit is not a part of the ANN core, but rather exists to support the core and allow it to interface with the rest of our speech recognition system. Its main duties are distributing weights to the tile modules by interfacing with the SDRAM memory controller and allowing an external entity access to the ANN through a serial peripheral interface. Additionally, it performs miscellaneous functions like dealing with externally generated reset conditions and controlling LEDs present on our custom circuit board.

III. Physical Hardware

A. Raspberry Pi

Since the processing of analog to digital for audio files is such a large computation that requires large amounts of processing power it was decided to use a Raspberry Pi 3. The processor of the Raspberry Pi 3 utilizes a 1.2 GHz 64-bit quad-core ARMv8 CPU which will be able to handle the conversions in a timely manner. The Raspberry Pi 3 also handles the inputs from the GUI and processes information depending on what the user requests via GUI input options.

B. Application Board

The application board contains various components of the project all in one. It has a power section, the FPGA deep learning computation section, and the applications section. All of these different sections work together using the digital audio from the Raspberry Pi 3 to identify the word spoken using the FPGA's programming and output the response using the LED application section.

The main components that are involved in this board are the power components, the FPGA, SDRAM, LEDS and connectors. The flow of data is from the Raspberry Pi 3 to the FPGA, back and forth from the SDRAM to FPGA and then finally from the FPGA to a particular LED. There are 14 LEDS, an SDRAM, Raspberry Pi 3 SPI pins and power connected to the FPGA.

For the power components, a barrel jack connector was chosen for the board to supply power. The wall outlet power supply of 5V was chosen to supply power to the board through the barrel connector. To get the two desired voltage levels for this board two voltage regulators were chosen. The two regulators would regulate the voltage from 5V to 3.3V and then to 1.2V as well. To keep the voltage steady and make up for any drops in voltage levels or noise decoupling capacitors were placed on the input and output voltages for each voltage regulator. This reduced the effect the change in voltage would have on the rest of the digital circuit. For our applications this is very important because the FPGA and SDRAM are dealing with very precise digital signals and steady voltages are pertinent.

The Xilinx Inc. XC6SLX9-3TQG144C FPGA was chosen based mostly on cost and assembly constraints. It has 144 pins in a quad-flat package with 102 IO pins which is great for connecting all of the components we need. With this FPGA at less than 20 dollars with meant IO pins and logic elements, it will be great for the application of the project. There is a 3.3V main power supply to the FPGA and also a second voltage level 1.2V.

The SDRAM is another important element to the application board design. For the design the IC SDRAM 256MBIT 143MHZ 54TSOP was chosen. This component stores information for the deep learning computations. It is needed because the FPGA can only store so many calculation values, with the SDRAM's help many more calculations can be saved for future calculations. All of the calculations will be explained further in other sections. The SDRAM we chose is also very cost effective and can store 256MB of data.

The LEDS chosen are 1206 SMD green LEDs with a voltage drop of 2V across them. They are connected to the FPGA with current limiting resistor banks in between them. These current limiting resistors are important in pulling the current down to a desired level according to KCL and KVL since the LED requires 20mA and 2V to turn on. These LEDs are labeled 0-9 and North, South, East and West. Once the FPGA has computed which word was said by the user it will output one result and light up the corresponding LED.

II. Application

The graphical user interface is how the user can easily interact with our system. Much of the functions that we need for our speech recognition system involve different programs that can be run on the command line. By using a graphical user interface, we have a central application that encompasses all the functionality of the desired programs.

This application will allow for audio input and features buttons that have various functionalities. These include audio processing and training. Other widgets will be used to output text and results to the user.

Each of the functionalities on the graphical user interface application demonstrate successful interfacing with the algorithms and hardware. The user interface allows the users to connect to the hardware via the SPI bus. This in turn, allows data to be sent and received to the raspberry pi.

We chose to run the application on the raspberry pi with a Debian Jesse operating system. This Linux based operating system allows for development to be done conveniently from a Linux Virtual Machine running Ubuntu.

The graphical user interface is developed using Qt Creator and PyQt. This gave us access to libraries that would assist in the design of our graphical user interface. The advantage of using Python is that it allows us to easily cross compile.

Our vision for the system is to be able to record audio data of the user speaking any of the integers from 0 – 9 or cardinal directions, north south, east, or west. This will be achieved through buttons on the user interface. The user will press record to start recording and stop to end and save the recording. In addition, the user interface also allows for the opening of a WAV audio file. This file will then still be able to undergo the same processing.

After pre-processing the data, the user can click on a button that will trigger the deep learning algorithm to run on the data on the FPGA chip. This feedback would then be sent back over the SPI bus and communicated through a display text box on the user interface. The FPGA will also be able to physically indicate a successful connection through the LEDs on the board.

V. Conclusion

Our project DeepGate was inspired by the capabilities and research possibilities of neural networks and FPGAs. Through research, design, development, and testing, the team was able to create a prototype for a speech recognition application that combines cutting-edge technologies.

This was accomplished in part by leveraging team member's experience and interests in FPGA and neural network research. Our goals were determined with the consideration of budget, time, and skill limitations. Overall, DeepGate achieved its objective as a usable prototype and a valuable learning experience for professional engineering work in both research and industry.

Acknowledgement

The authors wish to acknowledge the assistance and support of our review committee. Their time, mentorship, as well as consideration is greatly appreciated. We would also like to thank our advisors, Michael F. Young, an adjunct professor at UCF and entrepreneur, and Dr. Lei Wei for their advice and encouragement throughout the development process. Thanks so much to our sponsor SoarTech for their funding, support, and advice.

References

[1C] Park, J, & Sung, W. (2016). FPGA based implementation of deep neural networks using on-chip memory only. 2016

[2C] Tommiska, M. (2003). Efficient digital implementation of the sigmoid function for reprogrammable logic. IEE Proceedings - Computers and Digital Techniques, 150(6), 403. doi:10.1049/ip-cdt:20030965

[3C] Orban, Cedric. "Orbancedric/DeepGate." *GitHub*. N.p., n.d. Web. 06 Apr. 2017.

Biography

Lindsay Davis will be graduating from the University of Central Florida with a Bachelor's of Science in Electrical Engineering and a minor in Film. During her time at UCF, Lindsay held an intern position with Northrop Grumman focusing on ultrasonic sensors. She also led a robotics team for the 2016 NASA Student Launch Competition.

Estella Gong will be graduating from the University of Central Florida with a Bachelor's of Science in Computer Engineering. During her time at UCF, Estella interned with Lockheed Martin, SAIC, and State Farm for systems engineering, IT, and tech innovation research positions, respectively. After graduation, Estella will be entering Texas Instruments' Technical Sales Engineering and Product Marketing Engineering Rotation Program.

Michael Lopez-Brau will be graduating from the University of Central Florida with a Bachelor's of Science in Electrical Engineering with minors in Computer Science and Mathematics. During his time at UCF, Michael worked as a research assistant in biology, computer science, engineering, and psychology. After graduation, Michael will enroll in a PhD program in Psychology to bridge the gap between artificial and natural intelligence.

Cedric Orban will be graduating from the University of Central Florida with a Bachelor's of Science in Electrical Engineering. During his time at UCF, Cedric worked with FPGAs. After graduation, Cedric will be entering Stanford University's Electrical Engineering Master's Program.