ECG Classifier on Resource Constrained FPGA

1st Kenta Xu
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, United States
kentaxu@gatech.edu

2nd Daniela Ramirez Sanchez
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, United States
daramirezsa@gatech.edu

3rd Zheyang Liu
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, United States
zliu891@gatech.edu

***Abstract*—Wearable ECG devices have become common everyday items which can save lives. There have been previous works that focused on making ECG signal classifiers explainable or resource-efficient but not both [1]–[3]. This paper proposes a lightweight ECG classifier on a resource-constrained FPGA which can be applied for a wearable ECG device. This low latency implementation will allow critical ECG readings to be sent out faster. Additionally, by having low resource utilization, it can go onto a small wearable device consuming less power.**

I. Introduction

Cardiovascular diseases (CVDs) continue to be one of the leading causes of death worldwide. However, these numbers can be significantly reduced through vigilant monitoring of vital signs. The recent surge in wearable devices for real-time heartbeat monitoring presents an invaluable opportunity to enhance the current state of the art in wearable electrocardiogram (ECG) technology. In this work, we address two primary objectives to advance the capabilities of wearable ECG devices and improve cardiovascular health monitoring.

Our first goal is to develop an ECG classifier implemented on a Xilinx Pynq-Z2 Field-Programmable Gate Array (FPGA) platform. Utilizing FPGA technology allows us to significantly reduce the latency of the ECG classification model, ensuring rapid detection of potentially life-threatening cardiac events. Rapid detection and response can substantially improve patient outcomes and save lives.

Secondly, we aim to limit the resource utilization of our ECG classifier. This approach not only allows the ECG classifier to go onto physically smaller boards, but also reduces the device's dynamic power consumption. A lightweight ECG classifier leads to a more patient friendly device (easier to wear) that can also run for a longer time.

To achieve these objectives, we have trained a convolutional neural network (CNN) to classify ECG signals into one of five heartbeat categories, including normal heartbeats. The model was trained using the MIT-BIH Arrhythmia Database, with appropriate preprocessing applied to the original samples. After training, the model was exported to C++ and synthesized using Vitis. Our model successfully runs on the Pynq-Z2 FPGA board hosted by Georgia Tech.

II. Problem Description

Wearable devices for continuous ECG signal monitoring offer numerous advantages, including convenience, portability, and unobtrusive integration into daily life. Despite these benefits, current wearable ECG devices face two of the primary issues are slow processing speeds and high-power consumption, which limit their practical application and effectiveness in real-time cardiac monitoring.

Slow processing speeds in wearable ECG devices can lead to delays in identifying critical ECG readings that require immediate attention. Heart problems often have a very short window for effective intervention, making timely detection and alerts crucial for preventing adverse health outcomes, as this would enable prompt response to potential cardiac events [1].

High power consumption is another critical challenge faced by current wearable ECG devices. Wearable devices typically have limited size and battery capacity, making energy efficiency a paramount concern [2]. Efficient power management is essential for continuous monitoring, as high-power consumption can significantly limit the continuous monitoring capabilities of these devices, restricting their ability to effectively detect and warn users about cardiac events.

We propose a ECG classifier on a resource-constrained FPGA which can be applied for a wearable ECG device to address these two key issues.

III. Related Work

Researchers have shown machine learning (ML) 's potential benefits in the early diagnosis of cardiovascular diseases employing ECG readings. Heart rate variability (HRV), age, and sex features have been used to diagnose abnormalities for neural network classification [3]. Other studies have employed a custom convolutional neural network (CNN) for arrhythmia classification called ArrhyNet [1]; this study used the MIT-BIH arrhythmia database to classify a five-class classification system for arrhythmias with an accuracy above 92%. In addition, some studies have included XAI for ECG classification using the grad-CAM technique [4].

On the other hand, some studies aimed to implement energy-efficient techniques for wearable devices. In [2], authors proposed a two-stage end-to-end neural network where the computation complexity and power consumption were reduced by implementing a simple MLP classifier for normal and abnormal heartbeats before feeding the signal into the neural network for further classification. Sivapalan et al. [5] proposed a low-complexity binary classification for IoT wearable devices. However, more recent studies achieved a five ECG beat classification in an FPGA with high accuracy, high computing efficiency, and low latency suitable for wearable devices [6].

IV. Methodology Overview

The development of the project involved several steps as follows: Python implementation of the CNN model, implementation of the model in C/C++, and implementation on the PYNQ-Z2 board. The following sections will discuss in detail the steps and results obtained in these three stages.

V. Python Implementation of ECG Classifier

A. Dataset for Training, Validation, and Testing ML Model

The dataset we decided to use for our model comes from the MIT-BIH Arrhythmia Database. The original database contains heartbeat recordings of 48 different patients; each recordings consists of around 30 minutes of ECG signals. However, one thing to note is that a majority of this data is made up of ”normal” heartbeats with only some ”clinically rare” phenomena. Due to our lack of understanding in ECG signals and our limited time frame, we chose to use an already pre-prossessed MIT-BIH Arrhythmia dataset for ML ECG classifiers that someone else has published. This dataset has 87554 samples and labeled each sample based on its heartbeat. Label 0 represents a normal heartbeat while labels 1-4 represent an irregular one.

We split up the dataset into a train, validation, and test set and made sure that each one has a similar distribution of classes. One problem of the data set it is class imbalanced, where the number of samples labeled as normal greatly outnumber the other four classes. As a result, to prevent our ML model from being extremely biased, we randomly chose only 5000 samples from the over represented class to use for our training set. The slightly improved distribution of the classes used to train the model can be seen in Fig. 1.

B. ECG Classifier Model

Our model was based on two existing ECG classifiers from previous works [1]. However, due to the complexity of their models, we created a simplified version of them that consists of 1D Convolutions, Max Poolings, and Linear (Dense) layers. The structure of the model is shown in Fig. 2. The model follows the order of the ArrhyNet Architecture from [1] but

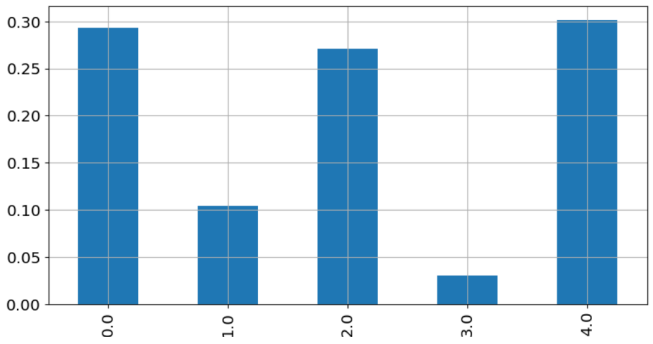

Fig. 1. Improved Dataset Class Distribution

skips the last two convolution layers. For the layer parameters, we based it off a different model architecture from Kaggle that was using our same dataset for training. One change we made to the model was replacing the AdaptiveMaxPool1D we previously had with a regular MaxPool1D; implementing Adaptive MaxPooling in C was challenging, and our advisor recommended changing it.

Model Layers	Layers Parameters
Conv1D	in channels=1, out channels=32, kernel size=5, padding=2
MaxPool1D	kernel size=5, stride=2
Conv1D	in channels=32, out channels=32, kernel size=5, padding=2
MaxPool1D	kernel size=5, stride=2
Conv1D	in channels=32, out channels=32, kernel size=5, padding=2
MaxPool1D	kernel size=5, stride=2
Conv1D	in channels=32, out channels=32, kernel size=5, padding=2
MaxPool1D	kernel size=5, stride=2
MaxPool1D	kernel size=5, stride=2
Flatten	None
Linear	in features=64, out features=32
Linear	in features=32, out features=5

Fig. 2. Model Architecture

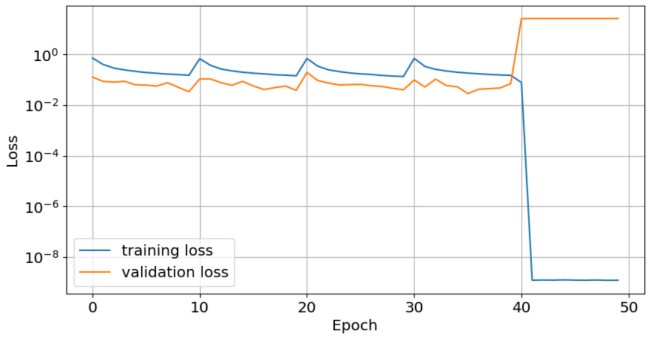

Fig. 3. Model’s Training and Validation loss

For training, we used over 5 folds (splits) using KFold of

10 epochs each and a learning rate of 1e-3. Adam optimizer was chosen for its faster convergence compared to stochastic gradient descent. We performed some hyper-parameter tuning with the number of folds and epochs. When the number of folds is too low or the number of epochs is too high, the model tends to over-fit to the training set and performs poorly on the testing set. Fig. 3 shows both the training and validation loss of our model. Our goal in creating and training this model was not to optimize its accuracy but rather have it working on an FPGA with low latency and not taking up all the resources on the board.

We ran this model on our test set to see its accuracy, precision, recall, and F1 score. Even though we did not aim to have a perfect model, Fig.4 shows the accuracy, recall and F1 score which were good enough for the objective of the project.

	precision	recall	f1-score	support
0.0	1.00	0.97	0.98	18571
1.0	0.53	0.95	0.68	307
2.0	0.91	0.94	0.92	1403
3.0	0.40	0.88	0.54	73
4.0	0.95	0.99	0.97	1538
accuracy			0.97	21892
macro avg	0.76	0.95	0.82	21892
weighted avg	0.98	0.97	0.97	21892

Fig. 4. Model's Accuracy, Precision, Recall, F-1 Score for Test Set

C. Pytorch Implementation

Using the dataset and model architecture described previously, we chose to utilize Pytorch in Python to create our ECG classifier. Leveraging the comprehensive library of pre-built functions and layers provided by Pytorch to construct a high-performance model. With the trained model, we exported its pth file to then later export its bin files. This Pytorch model allowed us to establish a solid foundation for our ECG classifier and provided us with a starting point for further optimization and hardware implementation.

VI. Golden C Implementation

A. Exporting and Reading bin files

Following the successful training of the model, our next milestone was successfully exporting binary files containing the weights, biases, and expected output for each layer of the model by using the model's pth file. These bin files of the model were necessary to be able to recreate the model in C++. We created a C++ simulation file to read and store the contents of these binary files into arrays. The arrays stored the golden outputs from each layer as well as the layer weights and biases for the 1D convolution and linear (dense) layers. A thorough comparison of the values read from the C++ simulation file with those from the Python implementation was conducted, confirming a match between the two sets of values to ensure accuracy and fidelity between the original model and its exported binary file representation, enabling accurate hardware implementation on the FPGA platform.

B. 1D Convolution Functions

Once the bin files were properly exported, we worked on the C++ implementation of our model beginning with the 1D convolution layer. We had 4 1D convolution layer functions each with a kernel size of 5, padding of 2, stride of 1, and 32 output channels. One difference between the first convolution layer and the following three was that the first layer had 1 input channel while the other three 1D convolutions had 32 input channels. Due to the MaxPool1D after each Conv1D, the dimensions of the input for each convolution got smaller starting off with a signal length of 187 and ending at 20. However, the functionality of all 4 Conv1D layers stayed the same. We followed Pytorch's documentation when implementing the layers which can be seen in Fig. 5 We multiplied the layer weights (read from the bin files) by the input array over a 5x5 kernel and added the bias. One thing to note was that we did cross correlation rather than a traditional convolution where we didn't flip the kernel. Using the bin files, the output of each convolution layer was confirmed to be correct.

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid cross-correlation operator, N is a batch size, C denotes a number of channels, L is a length of signal sequence.

Fig. 5. PyTorch Conv1D Functionality

C. MaxPooling1D Functions

Originally, our approach had 4 1D max pooling layers and an adaptive max pooling layer at the end. However, FPGA typically does not have built-in hardware to perform global maximum pooling operations, FPGA resources are required to enable this operation. This will cause the adaptive max pooling layer become a bottleneck in the case of hardware resource constraints, affecting the efficiency and speed of the neural network on the FPGA, we decided to replace the adaptive max pooling layer with a regular MaxPool1D. All 5 of the MaxPool1D functions used a kernel size of 5 and stride of 2. The functionality followed that of Pytorch by taking the maximum value over each area of input based of the kernel size and stride. The functionality of these functions have been tested by cross referencing with the outputs of the bin files as well for the MaxPool1d.

D. Linear(Dense) Functions

Our C++ model had two linear functions at the end resulting in an array of five values, one for each label. We applied a

linear transformation on the input array to model PyTorch's function. The weights and bias for the two layers were read in from the bin files and the expected outputs were compared as well to confirm correctness.

E. ReLU Activation Functions

To implement the Rectified Linear (ReLU) activation function after each convolution and first dense layer, we set all values that were negative to 0. We originally had the ReLu functionality at the end of each convolutional layer but ran into problems getting the correct output. As a result, we had to move some of the ReLu activations to the beginning of the MaxPool functions. In addition, we considered to implement a softmax activation function at the output of the last fully connected layer, but once again, the accuracy of the model was beyond the scope of the project, and a ReLU function performed the task we needed.

VII. Vitis HLS Implementation

A. Synthesizing on Vitis

We utilized the tcl script and Makefile from Lab2 as a starting point to have our C implementation synthesized on Vitis. We kept the tiled_conv function as the top level function however changed the functionality of the top function to fit that of our PyTorch model. As a result, the name "tiled conv" can be misleading because we did not perform any tiled convolution but rather the tiled_conv function is an ECG classifier.

The only changes from our golden C implementation and the baseline Vitis HLS implementation is that we added the AXI interfaces using the HLS pragmas. Due to the large number of bin files we had to read in for the input, weights, and biases of the model, we took advantage of all 4 bundles each with 4 ports except one. To get the baseline latency estimate, we turned off pipelining that Vitis HLS automatically tries to do. Our baseline approach had an estimated latency of 114 ms, used 36% of BRAM, 2% of DSP, 8% of FF, and 28% of LUT.

VIII. Optimizations

Our goal in our Vitis HLS Optimizations was to get a good speedup while limiting the resource utilization as well. We did not set an exact number on the resource usage for the board but rather conducted design space exploration by looking at trade offs for the different optimizations. In the end, our best design we decided to use was one that did not optimize the dense 1 and 2 functions.

A. Design Space Exploration

When first deciding what part of the code to optimize, we started off looking at the most computationally heavy parts. Our baseline latency was 114 ms based on Vitis HLS's estimate with a resource utilization of 36% for BRAM, 2% for DSP, 8% for FF, and 28% for LUT. Most of the latency came from the 1D convolution functions which took somewhere in the order of 10 ms. Following that was the latency from the dense functions and then the latency from the 1D MaxPool functions. The MaxPool functions were not computationally intensive, taking about 1 ms each to perform. As a result, we decided not to explicitly optimize the MaxPool1D functions and focus on the convolution and dense functions.

We optimized the 4 convolution layers first because those were the slowest parts. After optimizing the Conv1D functions, we were successful in getting a Vitis latency of 4.3 ms which was a 26.5 times speedup. Another achievement was the small increase in resource utilization achieved. We only used 46% of BRAM, 23% of DSP, 23% of FF, and 52% of LUT.

Subsequently, we optimized the dense functions one by one to see the resource cost for the additional speedup we can achieve. For the first dense layer, the function originally took 0.23 ms. After pipelining, we got a latency of 0.045ms which was a 5 times speedup. However, the resource utilization increased as seen in Fig. 6 by about 20% for each resource unit (BRAM, DSP, FF, LUT). Because the latency for the dense function only takes up a small percentage of the overall latency, you can see there is basically no overall speedup even though the dense function itself had a 5x speedup. The overall latency only went down by about 0.24 ms from 4.3 ms to 4.06 ms. As a result, we determined that the trade off we got of a negligible speedup for 20% increase in resource utilization is not worth it.

Similarly, we optimized the second dense function along with the first dense function to see how fast our ECG classifier can run if we optimized everything. But as you can see in Fig 6 optimizing the second dense layer resulted in an even smaller speedup (second dense function had a smaller baseline latency than the first one) for a similar increase in resource utilization which made it even less worth it. One other issue we ran into with optimizing the second dense function was getting negative timing slack sometimes based on the optimizations we did. As a result, we were further limited in the optimizations we could do for this function.

Resource	Latency	BRAM	DSP	FF	LUT
Baseline	114 ms	36%	2%	8%	28%
Optimized_NoDense	4.3 ms	46%	23%	23%	52%
Optimized_Dense1	4.06 ms	66%	47%	36%	76%
Optimized_Dense1&2	4.04 ms	66%	84%	47%	103%

Fig. 6. Design Space Exploration

Our final design optimized the 4 convolution 1D functions as well as the flatten function while turning pipelining

off for both dense functions. We allowed the compiler to automatically optimize the MaxPool1D and ReLu functions because that wasn't our focus. The compiler optimizations for the MaxPool functions achieved almost a 10 times speedup but the latency was low to begin with (about 1 ms) so the overall speedup was negligible. The Optimized_NoDense implementation in Fig. 6 was the one we chose to run on the Pyqn-Z2 board for our optimized on-board run.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|    639|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |        0|    5|    7230|  12460|    -|
|Memory           |      103|    -|      32|      3|    0|
|Multiplexer      |        -|    -|       -|   2251|    -|
|Register         |        -|    -|    2045|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |      103|    5|    9307|  15353|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       36|    2|       8|     28|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Fig. 7. Vitis estimated baseline resource utilization

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|     12|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |        0|   52|   19873|  23870|    -|
|Memory           |      129|    -|      32|      3|    0|
|Multiplexer      |        -|    -|       -|   4045|    -|
|Register         |        -|    -|    5122|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |      129|   52|   25027|  27930|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       46|   23|      23|     52|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Fig. 8. Vitis estimated optimized resource utilization

B. Conv1D Optimizations

Originally, the latency for the 1D convolution functions were about 4 ms for the first layer and 10-60 ms for the three following convolution functions (32 input channels rather than 1). After the optimizations, we were able to reduce the latency for each convolution to 0.07 ms for the first layer and 0.5-2 ms for the three following convolutions. In order to achieve this speedup, we experimented with a combination of loop reordering, pipelining, and array partitioning. We placed the kernel loop as the innermost loop and pipelined the loop before it to automatically unroll the kernel loop as well. Additionally, we partitioned the corresponding arrays along the corresponding dimension that indexed with the kernel loop.

C. Flatten Optimizations

The flatten operation was simple and took 0.001 ms to begin with. However, because it was easy to optimize, we pipelined the outermost loop, partitioning the arrays along the corresponding dimension which were indexed. As a result, we were able to complete the entire flatten operation in one clock cycle.

D. Reducing Communication Latency

When reading in data such as the layer weights from DRAM (through the AXI interface), we chose not to store all the arrays into BRAM to reduce the communication latency and BRAM utilization. We only loaded the arrays we partitioned into BRAM. As a result, loading everything from DRAM to BRAM took less than 0.1 ms. Our final communication overhead ratio (ratio of load/store latency and the computation latency) was about 1 to 55. The load store latency was 0.076 ms while the computation latency was 4.2 ms.

IX. On-board Implementation

A. Pynq-Z2 Board

Due to the limited options and time we had, our team decided to run our ECG classifier onto the Xilinx Pynq-Z2 board provided by Georgia Tech. After some discussion, we concluded it would take some time to find and configure a new physically smaller board than the Pynq-Z2 board which we were more familiar with. The board has 4 Advanced eXtensible Interface (AXI) bundles that can be used to interface with the board. Additionally, it has 280 BRAM, 220 DSP, 106400 FF, and 53200 LUT units that can be used.

B. Juypter Notebook

The required bitstream for the board was generated by Vivado, which also provided the reports with the resource utilization. We used the bit and harwdware handoff (hwh) files along with the Jupyter notebook to run our implementation on-board. We employed existing code previously provided to convert the floating point values to fixed point values. The Jupyter notebook read the bin files with the weights and biases, and followed the same logic used in Lab 3 to read the

files, load the buffers, run the implementation and calculate the latency on-board.

C. Latency and Resource Utilization

The original unoptimized estimated latency by Vitis was 114 ms, but the on-board latency is a bit slower, taking 183.29 ms. This result is shown in Fig. 11. This is due to Vitis being unable to perfectly predict how long our ECG classifier will take on-board. It makes sense because Vitis does not take into consideration different real-world factors and entropy which will cause variations in how long each on-board run will take. Vitis simply looks at some factors such as the clock period, number of loops, operations, and pragmas to come up with its generic estimate, but does not consider the whole scope of factors. As seen in Lab 3, the on-board implementation always has a higher latency than the one predicted. Similarly, the on-board latency for the optimized code is slower than the latency that Vitis estimated. Vitis estimated the optimized latency to be 4.3 ms, but the on-board latency was 7.63 ms, as seen in Fig. 12.

Looking at the resource utilization, the Vitis and Vivado estimates had some differences. Fig. 7 and Fig. 8 show the estimated resource utilization made by Vitis, and Fig. 9 and Fig. 10 show the Vivado report for the resource utilization.

The on-board resource utilization for the baseline implementation was 13.31% for LUT, 10.70% for FF, 38.57% for BRAM, and 2.27% for DSP. The estimate and actual values were only a few percent off for the most part. Similarly, this can be see for the on-board resource utilization for the optimized implementation. It used 23.80% of LUT, 20.85% of FF, 42.50% of BRAM, and 31.36% of the DSPs. Because the resource utilization is a lot closer to the estimates, the design space exploration done is Vitis HLS was probably a good representation of what each different optimization implementation's resource utilization would look like. Optimizing the dense 1 and dense 2 functions would probably cause the on-board resource utilization to also increase by a great amount for a small speed up. We predict the on-board latency would also take longer than what Vitis HLS estimates.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	7083	53200	13.31
LUTRAM	420	17400	2.41
FF	11387	106400	10.70
BRAM	54	140	38.57
DSP	5	220	2.27
BUFG	1	32	3.13

Fig. 9. On-board Baseline Resource Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	12660	53200	23.80
LUTRAM	640	17400	3.68
FF	22181	106400	20.85
BRAM	59.50	140	42.50
DSP	69	220	31.36
BUFG	1	32	3.13

Fig. 10. On-board Optimized Resource Utilization

```
start_time = time.time()

dut.register_map.CTRL.AP_START = 1
dut.register_map.CTRL[4] = 1
while not dut.register_map.CTRL.AP_DONE: pass

end_time = time.time()
duration = end_time - start_time
print(f'Kernel completed in {duration * 1000:.2f}ms')
```

```
Kernel completed in 183.29ms
```

Fig. 11. On-board Baseline Latency

X. Conclusion

With the lightweight ECG classifier we have designed, we were able to have a functional model with 98% accuracy working on a Pynq-Z2 board. Compared to the baseline latency, we achieved a 24 times speedup on-board, taking only about 8ms to be able to make a prediction. We are confident this fast prediction time for a patient's heartbeat signal achieves the goal of real-time monitoring of a person's condition and can save someone's life in a critical state. Additionally, we were able to greatly limit the resource utilization on our board while maintaining this speedup. We only used 42.5% of BRAM, 23.8% of LUT, 31.4% of DSP, and 20.9% of FF. As a result, the dynamic power consumption would be less than another ECG classifier implementation that uses the entire Pynq-Z2's resources. Furthermore, our Vitis HLS implementation of the ECG model has the potential to go onto a smaller FPGA board which would have less resources. This achieves our goal of applying our ECG classifier onto a wearable device that would be small, lightweight, and energy efficient.

A. Potential Future Work

One limitation we had with our work was the FPGA board we had available. A future work could be to implement the ECG classifier onto a different FPGA board such as a Lattice FPGA which are known to be optimized for medical devices. Compared to the Pynq-Z2 board we used, a Lattice FPGA

```
start_time = time.time()

dut.register_map.CTRL.AP_START = 1
dut.register_map.CTRL[4] = 1
while not dut.register_map.CTRL.AP_DONE: pass

end_time = time.time()
duration = end_time - start_time
print(f'Kernel completed in {duration * 1000:.2f}ms')
```

```
Kernel completed in 7.63ms
```

Fig. 12. On-board Optimized Latency

would take much less power to run and be much smaller in size.

Another area of improvement we could focus on in the future would be the ECG classifier model we developed. Because a majority of the project focused on implementing the model in C, optimizing the code in Vitis HLS, and running it on an FPGA, our model has room for improvement. This would allow for more accurate heartbeat readings which is important in reducing the false positive rate and more importantly the false negative rate.

One last potential future work would be to implement an Explainable AI (XAI) algorithm on top of the model to help medical professionals diagnose and understand a patient's heartbeat. Our model's prediction is like a blackbox where medical professionals are unable to understand what made a model output a specific predicition label. This will thus improve the usability of our model in the health field.

XI. Acknowledgements

Work done in this paper was thank to support from our advisor Ashwin Bhat and our Teaching Assistant Akshay Kamath. Special thanks to Ashwin Bhat for proposing the idea and providing advice through our progression. Special thanks to Akshay Kamath for answering all of our questions and helping resolve our issues.

References

[1] S. S. Aphale, E. John and T. Banerjee, "ArrhyNet: A High Accuracy Arrhythmia Classification Convolutional Neural Network," 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Lansing, MI, USA, 2021, pp. 453-457, doi: 10.1109/MWSCAS47672.2021.9531841.

[2] N. Wang, J. Zhou, G. Dai, J. Huang and Y. Xie, "Energy-Efficient Intelligent ECG Monitoring for Wearable Devices," in IEEE Transactions on Biomedical Circuits and Systems, vol. 13, no. 5, pp. 1112-1121, Oct. 2019, doi: 10.1109/TBCAS.2019.2930215.

[3] X. Ye, Y. Huang and Q. Lu, "Explainable Prediction of Cardiac Arrhythmia Using Machine Learning," 2021 14th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), Shanghai, China, 2021, pp. 1-5, doi: 10.1109/CISP-BMEI53629.2021.9624213.

[4] G. M., V. Ravi, S. V, G. E.A and S. K.P, "Explainable Deep Learning-Based Approach for Multilabel Classification of Electrocardiogram," in IEEE Transactions on Engineering Management, doi: 10.1109/TEM.2021.3104751.

[5] G. Sivapalan, K. K. Nundy, S. Dev, B. Cardiff and D. John, "ANNet: A Lightweight Neural Network for ECG Anomaly Detection in IoT Edge Sensors," in IEEE Transactions on Biomedical Circuits and Systems, vol. 16, no. 1, pp. 24-35, Feb. 2022, doi: 10.1109/TBCAS.2021.3137646.

[6] J. Lu, D. Liu, X. Cheng, L. Wei, A. Hu and X. Zou, "An Efficient Unstructured Sparse Convolutional Neural Network Accelerator for Wearable ECG Classification Device," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 11, pp. 4572-4582, Nov. 2022, doi: 10.1109/TCSI.2022.3194636.