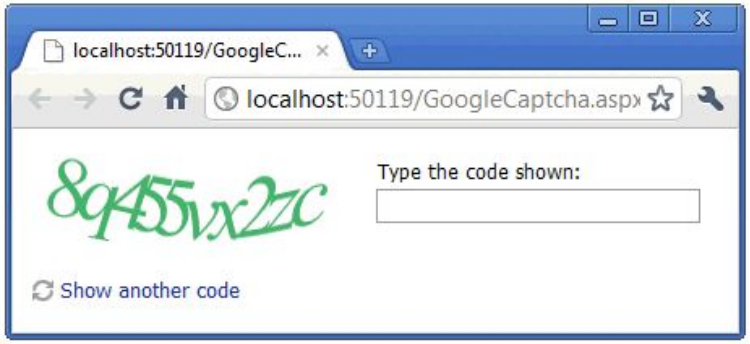
The CAPTCHA Beater

Mauris, Nikhil, Randy & Srivatsan

Project Overview

- We focused on text recognition for CAPTCHAS
- Implemented a neural network on an FPGA to tackle this issue
- Potential applications:
  - Malicious captcha farms
  - Captcha solving assistant to individuals that are visually impaired
  - Converting handwriting to text
  - Finding vulnerabilities in captcha generators
  - Captcha avoidance for automated testing purposes

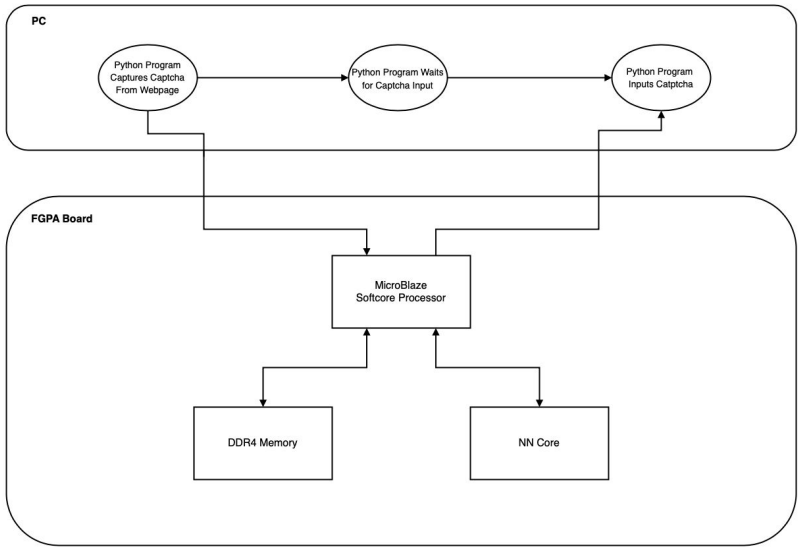


Project Goals

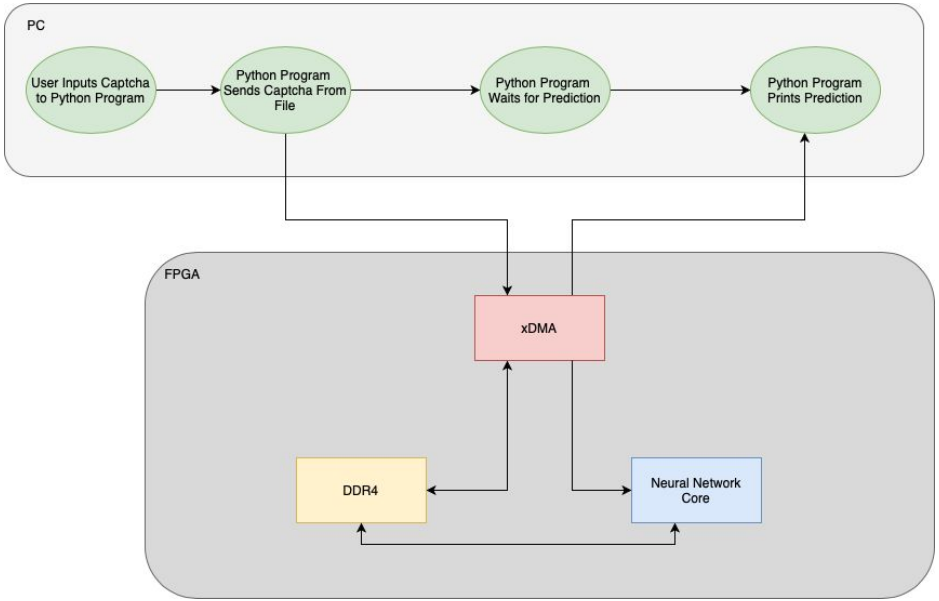
- Achieve a 80% accuracy for predicting text based CAPTCHAs
- Achieve a < 20s prediction time; accelerate predictions using hardware.
- Finish by end of May
- Design an application for the user to interface with
    - Application able to be run from their own PC

System Architecture

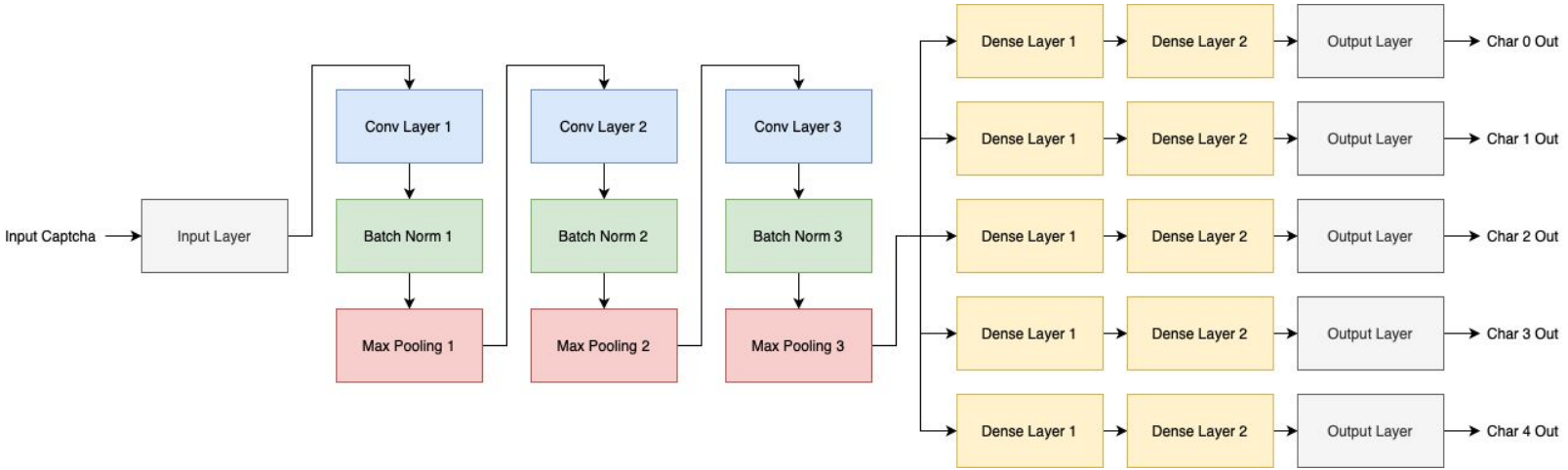
Proposed:



Implemented:



Neural Net Architecture



Project Organization

- Roles for all members were clearly defined
  - No two people working on exactly the same thing
- Github was used as version control for software
- Google Drive used for all miscellaneous items
  - Used Google Sheets to track resource usage
  - Used Google Slides to create presentations
- Overleaf (online LaTeX document generator) used for report
- Discord used for meetings/communication within group
  - Different channels for discussing different aspects of the project
- Weekly meetings initially, daily meetings at the end
- Slack and email were used for coordinating meetings/seeking assistance from instructors

Software Verification and Testing

- Python verification of model using Kaggle and Google Colab
  - Tested model across thousands of images to generate expected predictions
- C testbenches for testing software NN model
  - Manual inspection of each layer's outputs/error to verify functionality of individual layers
  - Full test-bench to test entire model
  - Verified model accuracy over 1000 test images
- CSim and CoSim testing of all layers individually
  - Verified functionality of all layers in HLS by performing RTL Co-Simulation with C testbench
  - Created table of expected performance and resource usage for different clock periods

Hardware Verification and Testing

- Testing of each NN layer was performed through Vivado
  - Loaded preset images using temporary module which would transmit them to the NN (controlled using VIOs)
    - Avoided issues with PCIe
  - Assembled layers successively by generating bitstream after each layer and ensuring hardware output matched software output
  - Output matching was done via ILA debug module and Vivado simulator
- PCIe testing was done using software and Vivado simulator
  - Ensured drivers were working through software
  - Read and Write tests to memory were performed using a combination of DMA code from HW#2 and PCIe test files
  - Read and Write to specified offsets from hardware NN to PCIe and verified results through ILA

Unique Design Aspects

- Restructured Dense layers to allow for immediate processing of input stream data. This reduced resources requirements to cache the input stream and also reduced latency.
- Modified max pool layer to only store two full rows of input pixels at any given time. Used a buffer to store the required pixels, made a calculation and then loaded in the next set of pixels. The entire process was parallelized, so buffer loading was done while a calculation was being performed. (not currently implemented)

HLS Components

- Neural Network will be implemented in HLS
  - Input Layer, Convolution Layer, Batch normalization layer, Max pool layer, Dense layer, Output layer
- What else is needed that is not HLS?
  - PCIe to AXI based DDR memory module
  - AXI broadcast module
  - AXI Interconnect modules to communicate between Neural Network cores and memory
- Summary
  - HLS was used to implement the processing cores of the design
  - HLS was not used to interface with external hardware and also not used for PC to memory communication

Overall Impressions of HLS

- Easier flow as compared to writing HDL
  - Ability to change synthesis very easily using couple pragmas
  - Ability to target multiple clock periods
  - Built-in tools provide optimized RTL synthesis for specified constraints
  - Higher level of abstraction means less lines of code need to be written
- Allows us to skip an entire step in design process
  - Would have needed to create a C-model anyways to understand the different layers and their function
  - This C code was directly converted to hardware using HLS
  - Verification was also simplified as Co-Sim provided synthesis testing; no Verilog testbenches required
- Design is a “black box” - less control over IP