EECE 4632: Spring 2022
Hardware-Software Codesign for FPGA-Based Systems
Midpoint Progress Report
Oscar Kellner

Within the past month, a lot of research has been conducted, but a large diversion has changed the scope of the project from the perspective of the Microprocessor-Based Design (MBD) class. A better understanding of what needs to be completed before the end of the semester has been established, as well as ideas for how to approach the project.

Initially, the strategy I had to ensure that the hardware I synthesized would be as easy to reintegrate into the original framework (Darknet) was to modify the original code as little as possible, and verify that these changes still worked if I had reintegrated them into a modified version of darknet. While this did produce synthesizable code after careful tweaking, the hardware was extremely unoptimized - without any directives, the latency of the design was a grueling 3.306 seconds for only the first convolutional layer. In comparison, the original Darknet program could complete a prediction (processing multiple convolution and pooling layers) at around 0.13 seconds give or take a couple hundredths of a second on my laptop.

Despite the horrible performance, it was worth attempting to optimize the design through use of directives to see how much faster it could get. However, due to the abundance of independent nested for-loops within the design, attempting to unroll or pipeline any of the loops except for the innermost ones would result in the synthesis failing due to an extremely large code size. The same issue arises from attempting to perform a complete partitioning of the input arrays - the most that could be done was a block partition with a factor of 224, which would ultimately slow the design regardless. Because of these limitations, not much analysis can be done on various optimizations, however the latency and area for these designs were still recorded:

Design	Latency (seconds)	Est. Area	BRAM	DSP	FF	LUT
1	3.306s	2235	0	5	991	1735
2	7.099s	8884	0	9	4310	7984
3	1.837s	3476	0	8	1554	2676
4	1.993s	11593	0	12	8084	10393

Brief description of the designs:

1. Base implementation. No directives added.
2. Block partitioning with factor=224 on array parameters ‘input’ and ‘output’
3. Pipelining of innermost loops within GEMM and im2col functions
4. Combination of directives within designs 2 and 3

On top of the poor performance, the design also had some other ‘artifacts’ of a more software-based implementation that was unfriendly for hardware. One example of this was the ‘workspace’ input array that was in the original forward convolutional layer algorithm, which was apparently a large space of memory reserved for storing the result of the im2col function so that it could immediately be used within the subsequent gemm function. Attempting to synthesize this would very quickly exhaust all of the FPGA’s block RAMs just to temporarily store data that would immediately be used within the next function as opposed to just streamlining the data.

It was extremely apparent that this design wasn’t going to be a good choice for this project, so it was decided to scrap the original code from the framework and rewrite the forward convolutional layer algorithm from scratch. This rewriting is still in process - the challenge is in deciphering how the input data, output data, and weights are stored within Darknet so that the algorithm can be adjusted to ensure correctness as the output is forwarded to the next layer within the network. Additionally, if we know that the data can be written and read from in a continuous manner, it would be easier to implement an AXI-Stream and DMA interface to the design. Even if this wasn’t the case, it may be possible to have a helper function (in software) to organize the input and output data as such before being handed off between the Darknet framework and the hardware accelerator.

This new design will be somewhat reminiscent of the software implementation described back in the first project update from about a month ago. The plan is to modify it so that the convolution is processed as a “sliding window,” as described in this link below:
https://basile.be/2019/03/18/a-tutorial-on-non-separable-2d-convolutions-in-vivado-hls/

From my past research into this topic, my original intention was to interface the design using a kernel module or user space application within the ZedBoard FPGA used in MBD. This had originally generated an expectation that the project would be able to be used as a modular component within the class project currently being worked on within MBD, hence the insistence of the first design focusing on ensuring that minimal changes are made between the HLS code and modifications to the Darknet framework to ensure compatibility. Recently, the scope of the project had changed so that the new objective is to “hijack” the PYNQ library in order to interface with the synthesized IP in C, specifically the header file found within this folder within the PYNQ public repository:

https://github.com/Xilinx/PYNQ/tree/master/sdbuild/packages/libsds/libcma

In other words, this project is now being moved to work with the PYNQ boards used within this class, which should make the integration and debugging process easier. I am not entirely sure how I will be able to interface the IP with the original Darknet framework after the revelation given to me from a helpful graduate student, though that is outside the scope of this project for this class.

Because the GitHub for this project is likely restricted to students and faculty of the MBD class, I will upload a ZIP archive of the contents within my GitHub at the current moment. The source files fcl_old.c and fcl_old.h contain the original design taken from Darknet's forward_convolutional_layer function from convolutional_layer.c before it was scrapped. The fcl.c and fcl.h source files hold the current incomplete code for the new design for the convolution algorithm, and currently is not compatible with the framework (in that it does not reproduce the same results as the framework's original function). Within the 'hls' folder contains two more source files, hwfcl.c and hwfcl.h, which are slightly modified versions of fcl_old.c and fcl_old.h that should be synthesizable in Vivado HLS.

Within the root folder, I have also included a 'darknet' folder that contains 2 instances of the Darknet framework - a 'modified' and 'original' stored within their respective folders. The 'modified' Darknet, as the name implies, contains the modified source code so that it will run the hw_fcl function that I had written if the parameters of the convolutional layer match that of which is defined within fcl_old.h. To compile them (in a Linux environment), you can change to either of these folders (the root directory for the Darknet instance) and run "make." To test them, run a "classify" command with the 'tiny' configuration as shown below:

./darknet classify cfg/tiny.cfg tiny.weights data/dog.jpg

For both the 'modified' and 'original' Darknets, you should see the same predictions verifying that the hw_fcl function works:

14.51%: malamute
 6.09%: Newfoundland
 5.59%: dogsled
 4.55%: standard schnauzer
 4.05%: Eskimo dog

Below is a brief outline as to what the next steps of this project are in regards to my current position:

For this class' project:

1. Finish rewriting the hw_fcl function in fcl.c and fcl.h so that it is more "hardware-friendly" while still reproducing the same results when substituted back into the modified Darknet framework.

2. Extract data on input weights, input image, and output feature maps from the original Darknet framework so that they can be used for test benching and debugging.

3. Modify the fcl.c and fcl.h source files so that they are synthesizable in Vivado HLS.

4. Integrate AXI-Stream interfaces for all of the function parameters.

5. Add DMA interface(s) to the IP in an RTL project in Vivado. Synthesize and gather the hardware handoff and bit stream files.

6. Verify their functionality in a Jupyter Notebook using the data from step 2.

7. If time allows it, delve deeper into design space exploration and create a detailed Pareto optimization graph using multiple different optimizations to find the designs with the least area, least latency, and the best tradeoff between the two.

8. Make a performance comparison with the software implementation.

It's important that I finish these steps first, and quickly, as the project for MBD will rely on me having a working hardware design so that I could interface with it using a user space application in C that will communicate with the PYNQ board's kernel drivers and Darknet as opposed to using just Python.