EECE 4534: Spring 2022
Microprocessor-Based Design
Final Project Report

Oscar Kellner

Abstract

Image convolution is a common process used in convolutional neural networks for machine learning applications, and many solutions exist for accelerating this process through use of dedicated hardware. This project intends to synthesize a 'hardware convolution accelerator' through High Level Synthesis for use within an FPGA's embedded system in order to increase the performance of an existing machine learning framework's convolution. The Darknet framework was used as it was open-source and written in C, potentially making it ideal for use within an embedded system. While the project was unsuccessful in reintegrating the accelerator back into the framework due to unexplained segmentation faults that occurred on the PYNQ's PetaLinux, a test bench program written in C revealed that using the accelerator to process the single convolution layer was just under 2 times faster than using a purely software-based implementation. Had the integration with Darknet been successful, it's likely that the accelerator would have allowed for greater performance in comparison to running Darknet purely on software, but not by much.

I. Introduction

Convolution neural networks (CNNs) are a type of neural network that excel in processing multidimensional data (e.g. images) and are a crucial component in computer vision applications. A CNN is composed of multiple layers, including convolutional layers, max pooling layers, and fully-connected layers. Within each layer, more complex and larger features of the input data are distinguished until at the end of the network where the network would make its best prediction as to what object it is observing [3]. This project intended to design a hardware implementation of specifically the convolution algorithm used within a convolutional layer (known here as a "convolution hardware accelerator"). The hardware was developed through High Level Synthesis (HLS), an emerging technology that allows for a function written in a high-level language (C in this case) to be converted into a register transfer level (RTL) intellectual property (IP) core that can be used within a larger RTL project for FPGA-based systems. In order to interface with the generated hardware, the PYNQ FPGA contains built-in libraries that allow for contiguous memory allocation as well as loading the bitstream and hardware handoff files generated by Vivado.

II. Related

The final hardware design for this project takes heavy inspiration from Basile Hoorick's implementation of non-separable 2D convolutions in HLS, which makes use of a sliding window and line buffers in order to make the best use of AXI-Stream interfaces for efficient data transfer [2]. To further improve data transfer, two DMA controllers were used to handle the two input and one output streams for the accelerator, which were implemented with assistance from documentation created by Sivasankar Palaniappan [6]. Finally, a blog post from Lauri Võsandi provided the basis for userspace driver code that was developed to interface with the accelerator and DMA controllers in C [5]. Special thanks extends to Aly Sultan for additional guidance towards the development of this project.

III. Design Approach

The ultimate goal of this project was to be able to interface with an existing machine learning framework so that the accelerator would run if the parameters of the current convolution layer fit within the constraints of the hardware, or run a software function if it didn't. The Darknet framework [4] was chosen as it was both open-source and written in C, potentially making it the most convenient choice for integration within an embedded system. In the context of hardware, the hardware had to be designed such that the organization of the input and output data was convenient for the Darknet framework to interpret as to reduce any potential latency that could be caused by needing to reorganize data.

The code used in High Level Synthesis had been overhauled and rewritten many times, finally settling for a more simpler design that will perform a single convolution for one channel and one kernel at a time, accumulating the results to a "partial sum" feature map that is streamed back into the accelerator if it is processing multiple channels for a given filter, as visualized in the figure below. Note that the partial sum is handed back to the software rather than directly back into the hardware, causing some overhead in data transfer.

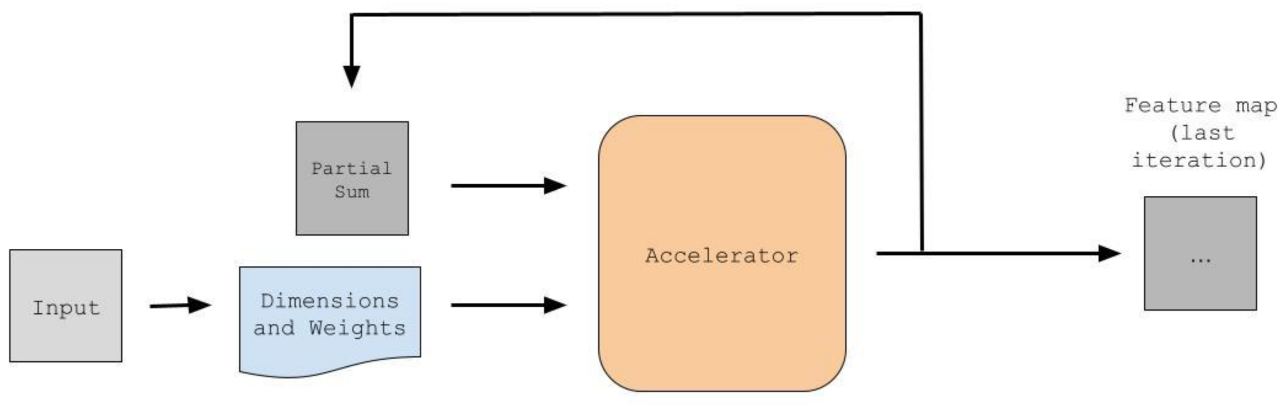

Figure 1: Visualization of final design

The typical "Conv2D" algorithm is used in this implementation, in that the value of a given pixel for an output map is determined by multiplying the input pixel and its surrounding neighbors within the kernel window by its weights stored within the window and accumulated together [1]. In this implementation, an output value is streamed out as soon as it is calculated. The accelerator uses AXI-Stream and DMA interfaces for a greater efficiency in handling inputs and outputs. In order to conform to the nature of streaming in that data is only read once in a sequential order, a line buffer was implemented to retain input values that may appear again within the window [2]. Occasionally, there may be a convolution layer that has a kernel size of 1x1. When this occurs, there is no need for a line buffer as only one value is being multiplied and stored to the output map at a time in sequential order. Thus, a simple nested for loop was sufficient for a convolution involving a 1x1 kernel. It's worth noting that kernels with even dimensions (e.g. 2x2) are not compatible with this accelerator.

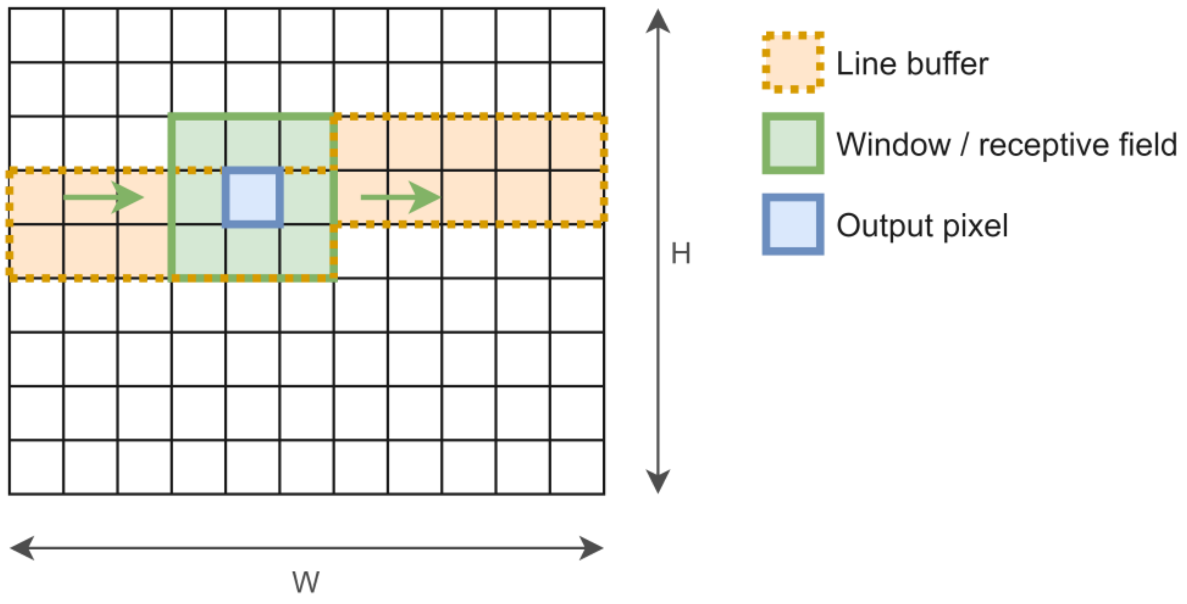

Figure 2: Description of line buffer and sliding window [2]

Designs prior to the current implementation were very limited. The first proposed design attempted to keep the input data as local as possible by attempting to store it all at once, however this design exhausted the entire area of the FPGA with the amount of memory that was needed. Another design contained 4 input streams, 3 of which were reserved for input of individual channels. This allowed for the hardware to fully process a convolution for an entire filter at a time, but restricted to only processing convolution layers with 3 input channels.

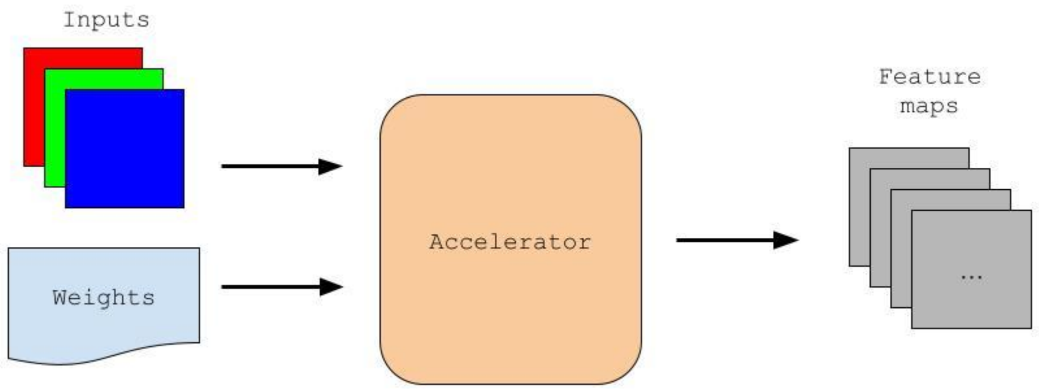

Figure 3: Visualization of first design

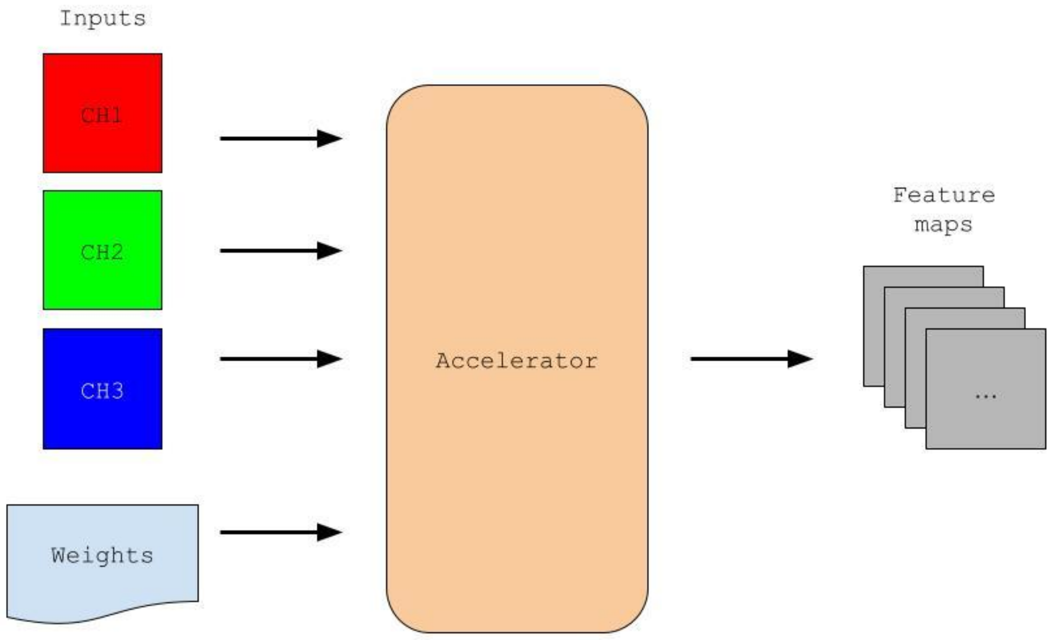

Figure 4: Visualization of 3-channel design

Debugging the code used within HLS was challenging as the function needed to conform to the way data was structured in the Darknet framework, as it was important to ensure integration back into the Darknet framework was as simple as possible later in the project. The approach for debugging was modifying a copy of the Darknet framework so that a different function (which would be used for hardware synthesis) would be called if the current convolution layer's parameters were sufficient for the accelerator. The new function would process the data in isolation, and return the values back into the framework. Keeping the function isolated from the rest of the framework (i.e. no libraries or functions shared with the source code of the framework) would allow the code to easily be transferred to Vivado HLS for synthesis.

After synthesizing the hardware (with no HLS optimizations), a Jupyter Notebook was created to interact with the hardware through use of Python and the PYNQ's built-in libraries. A basic test script was first written with help of the PYNQ documentation to ensure that data could be written to and read from the AXI-Stream and DMA interfaces [6]. Once the interfaces were confirmed to be working and that the outputs looked sensible, another script was created to emulate the first convolution layer of the Darknet framework's example configuration found within the tiny.cfg file. This was accomplished by creating 3 data files each containing a stream of float32 values for the layer's inputs, weights, and expected outputs, which were obtained by storing the values from a test run with an unmodified copy of the Darknet framework. This layer consists of a 3-channel input map of 224x224 values with 16 filters using 3x3 kernels, resulting in 16 feature maps each with a size of 224x224. The script would read the inputs and weights, run the accelerator, and compare the generated outputs with the outputs from the data file and count the number of incorrect values. After careful debugging, the script was able to report no discrepancies between the generated output and the output from the data file, confirming that the hardware design worked as intended.

The C testbench program works very similarly to the Python script in that it simulates the first convolution layer of Darknet's tiny.cfg network using the same data files as described above. A userspace driver was written to operate with the AXI-Stream and DMA interfaces and run the accelerator [5]. The software implementation that it is compared with is virtually the same code used in HLS for the most recent design.

Once the hardware was verified to be functioning properly, options for performance optimization were explored. The approach taken for exploration was to add a single optimization per iteration to test if the optimization had improved the performance (i.e. reduced the latency of the design), and keep it for subsequent optimizations if it did. The type of optimizations applied generally involved heavy tradeoffs between higher area and lower latency, such as loop unrolling, loop pipelining, and array partitioning. Use of arbitrary precision integers for loop control variables had only reduced the area of the design, and use of fixed-point arithmetic for the multiply-accumulate operations had increased the design's area contrary to what was expected (but still resulted in a significant speedup). The collective optimizations applied to the final design reduced the latency by almost a sixth in comparison to the unoptimized version.

IV. Experimental Results

After optimization, the aforementioned programs were run to compare the performance of the software, unoptimized hardware, and optimized hardware, where applicable. The first test program was emulating the Darknet convolution layer in Python, which compared the latency and accuracy of the unoptimized and optimized hardware designs. In this test, the accelerator is also run multiple times as many channels and filters needed to be processed, presenting a significant latency from loading data into the buffers. The unoptimized design spent 12.06 seconds to fully complete the layer, while the optimized design took only 3.56 seconds, presenting a 3.4 times speedup with the optimized design. When not accounting for the ~1.73 seconds it takes to fill the input buffers in between each use, the optimized design performed 5.6 times faster than the unoptimized hardware. Some accuracy is sacrificed with the optimized accelerator as it uses fixed-point arithmetic, though the average error as a result of this was negligible (on the order of 10^{-7}).

Finally, the C test program was used to compare the software and optimized hardware implementations of the convolution algorithm, using the same inputs and outputs of the previous script. The software takes around 5.321 seconds to run, while the hardware takes 2.685 seconds - almost half the time of the software. For the optimized design, the C program is about 1.32 times faster than the equivalent Python script. A graph comparing the performance of the Darknet-based tests is shown below:

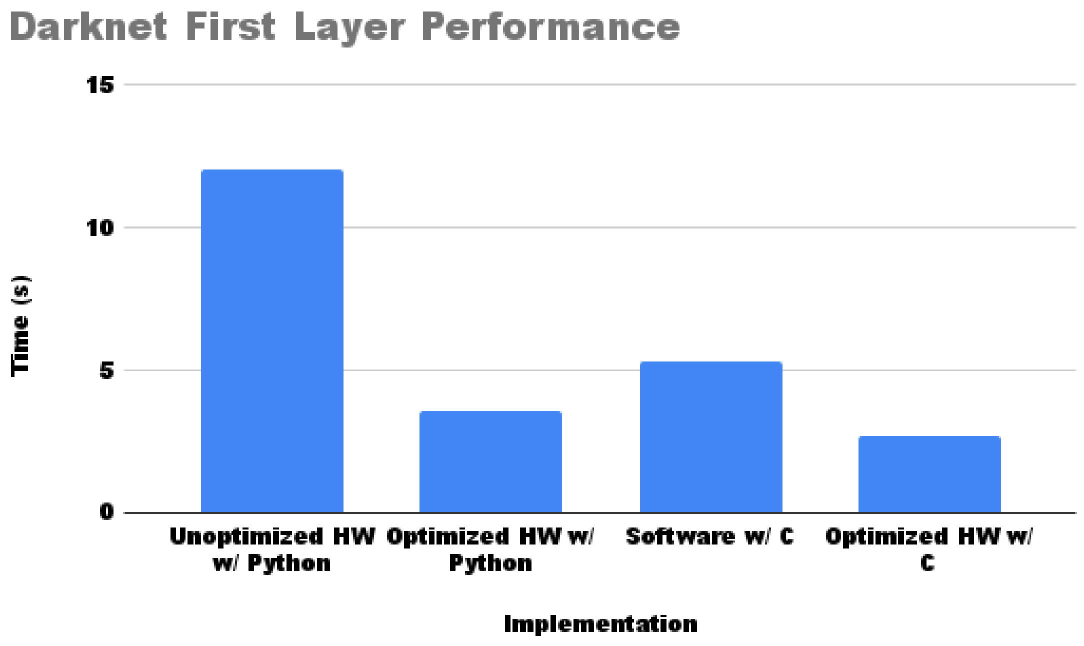

Figure 5: Performance comparison of each test

V. Reflection

Unfortunately, there wasn't enough time to integrate the Darknet framework with the project as there were several compatibility issues between the FPGA and how Darknet parsed the input data, causing segmentation faults that weren't obvious to resolve. Despite this, if it were possible to get Darknet fully operational on the PYNQ, it could be argued that emphasizing compatibility with Darknet in the design of the accelerator was ultimately worth it. Based on the time spent filling the buffers in the Python test program, moving and shifting data had proven to be a major source of latency. It's likely that needing to reorganize data could have caused a significant amount of overhead and negated the advantage of using a hardware accelerator rather than just designing the accelerator around the framework. That being said, a substantial amount of time had been wasted by hastily attempting to implement designs that tried to change as little as possible from the original Darknet source code. The problem with trying to do this is that Darknet's algorithm for convolution uses an algorithm known as "im2col" [1], which is unfriendly for HLS as it requires a large amount of area and does not easily allow for use of streaming (if at all). Ironically, the final code has almost no relation to Darknet aside from the source of the input data used for testing and how the inputs and outputs were organized.

There were also occasionally some design errors when attempting to implement the AXI-Stream and DMA interfaces, such as forgetting to include a TLAST signal in order to indicate that the data had finished streaming. An additional AXI-Lite signal was added to determine the length of data being sent to the DMA for additional control to ensure that this problem would not occur. Other questionable design choices included the use of control statements within or outside the nested for loop that performed the convolution, which had limited the ability for Vivado HLS to pipeline the loops. All of these mistakes had consumed a lot of time that could have been spent exploring optimizations for the design, and could have been avoided by first closely studying the Darknet framework and spending more time planning out the hardware rather than hastily implementing half-working designs.

VI. Conclusion

This project aimed to develop a hardware-based accelerator to speed up image convolution, and integrate the hardware into an existing machine learning framework (Darknet). Multiple designs were considered, finally settling on a design that processes one channel at a time for greater versatility. The performance of the hardware is compared to an equivalent software implementation within a test bench program, revealing a speedup of almost 2 times faster than software - a result that is favorable but underwhelming. Integration with the Darknet framework was unsuccessful due to compatibility errors and time constraints.

If more time was allowed for the project, a potential next step after finding a solution to the compatibility issue with Darknet would have been attempting to redesign the algorithm so that sections of the input would be processed as chunks in parallel to increase performance. In other words, rather than convolving through an entire 224x224 image at once, four 56x56 chunks would be processed in parallel and written to the same output buffer to potentially increase throughput. There may also be more efficient algorithms for image convolution that were undiscovered throughout the course of this project that may provide a greater performance while still being flexible for different convolution layer parameters, which could have been revealed with additional research.

Bibliography

[1] A. Shenoy, “How Are Convolutions Actually Performed Under the Hood?,” *Towards Data Science*, 13-Dec-2019. [Online]. Available: https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf. [Accessed: 05-May-2022].

[2] B. V. Hoorick, “A tutorial on non-separable 2D convolutions in Vivado HLS,” *Basile Van Hoorick*, 18-Mar-2019. [Online]. Available: https://basile.be/2019/03/18/a-tutorial-on-non-separable-2d-convolutions-in-vivado-hls/. [Accessed: 04-May-2022].

[3] “Convolutional Neural Networks,” *IBM*, 20-Oct-2020. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks. [Accessed: 04-May-2022].

[4] J. Redmon, “Darknet: Open Source Neural Networks in C,” 2016. [Online]. Available: https://pjreddie.com/darknet/. [Accessed: 05-May-2022].

[5] L. Võsandi, “AXI Direct Memory Access,” *Lauri's blog*, 12-Dec-2014. [Online]. Available: https://lauri.xn--vsandi-pxa.com/hdl/zynq/xilinx-dma.html. [Accessed: 04-May-2022].

[6] S. Palaniappan, “Lab: Axistream Multiple DMAs (axis),” *Read the Docs*, 2019. [Online]. Available: https://pp4fpgas.readthedocs.io/en/latest/axidma2.html. [Accessed: 04-May-2022].