EECE 4632: Spring 2022
Hardware-Software Codesign for FPGA-Based Systems
Final Project Report

Oscar Kellner

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 [2]. This project intended to design a hardware implementation of part of a convolution neural network, specifically the convolution algorithm used within a convolutional layer (known here as a "convolution hardware accelerator").

The ultimate goal of this project, outside the scope of this class, 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. Because this was a joint project with another class (EECE 4534: Microprocessor-Based Design) that emphasized interfacing with embedded systems, the Darknet framework was used 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 this project, the hardware had to be designed such that the organization of the input and output data was convenient for the Darknet framework to interpret. Sample data generated from the framework was also used for testing. For this class, the goal of this project was to synthesize the hardware, validate its functionality, optimize the design, and compare the performance to a similar software-based implementation. In regards to hardware-software codesign, the software loads the inputs from an image or data file and into the hardware, which performs the image convolution. The software then takes the outputs and stores them into feature maps.

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" map that is passed back into the accelerator if it is processing multiple channels for a given filter. 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. 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 [1].

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.

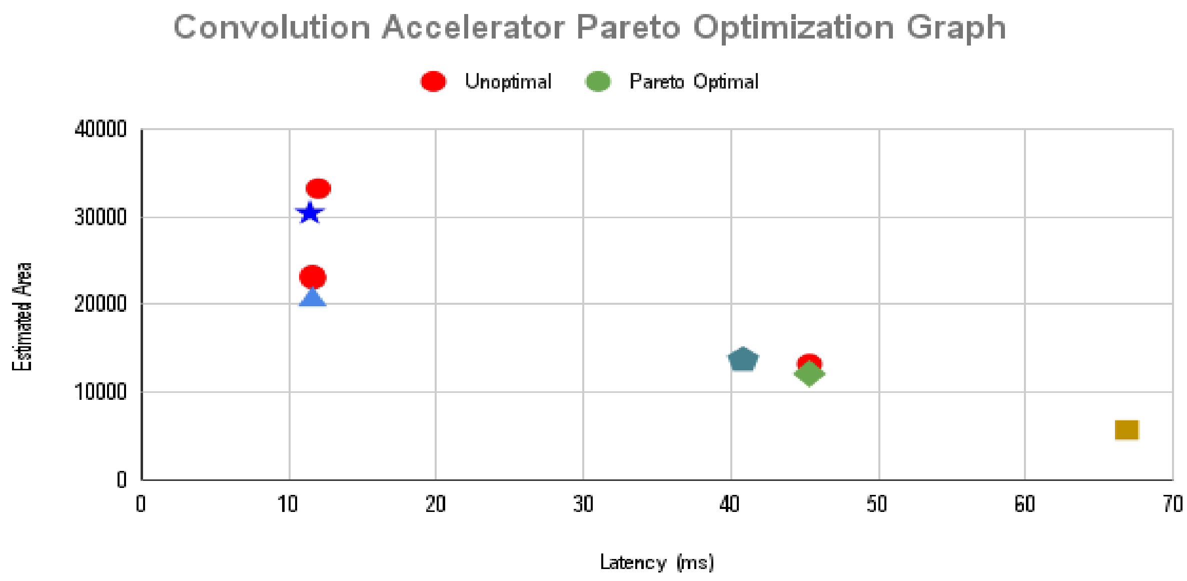
Debugging the code used within High Level Synthesis was challenging as it needed to conform to the way data was structured in the Darknet framework, as it was important to ensure integration back into the Darknet framework was as simple as possible later in the project. The approach for debugging was modifying the original 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 IPython 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 [4]. 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.

For a more visual demonstration of the accelerator, an additional test script was made to recreate the software implementation that was originally made as a "proof of concept", where a few predefined kernels were used to convolve over a sample 128x128 RGB image depicting a flower. This script borrows resources from an educational article referenced at the end of this document [3]. This script also served as a temporary method of comparing the performance between the convolution hardware accelerator and the software implementation of the Conv2D algorithm written in Python before the hardware interfacing was written in C.

The C program works very similarly to the first IPython script in that it simulates the first convolution layer of Darknet's tiny.cfg network using the same data files as described above. Code was written to operate with the AXI-Stream and DMA interfaces and run the accelerator as before [5]. The software implementation that it is compared with is virtually the same code used in the High Level Synthesis 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). Data for each design iteration's utilization and latency is included in Appendix A. Iteration 9 was chosen the partial unrolling of the conv_x for loop in iteration 10 consumed too much area for the marginal improvement in latency. A Pareto Optimization chart for the different designs and their comparison between latency and area is shown below. The exact data for each point can be referenced from Appendix A.



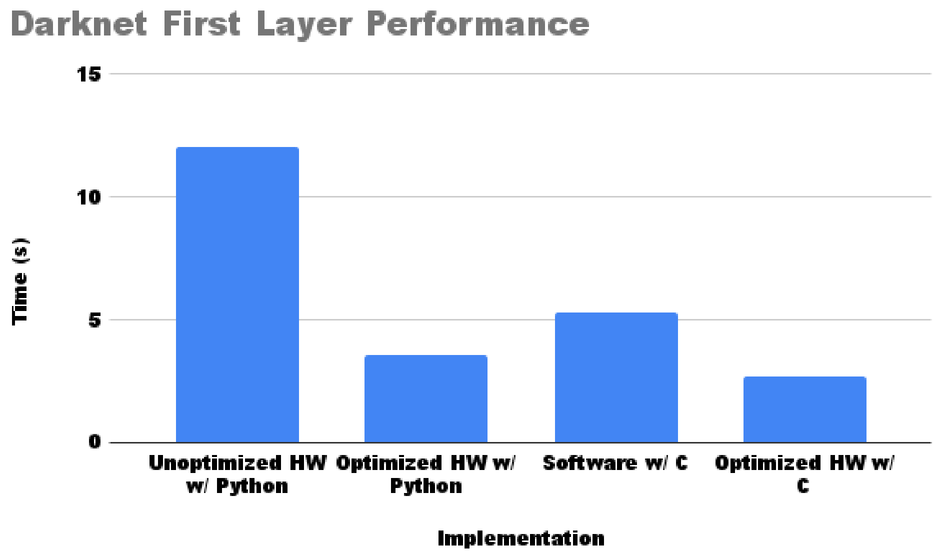
LEGEND (SHAPE: ITERATION)

STAR: 10	TRIANGLE: 9	PENTAGON: 4	DIAMOND: 3	SQUARE: 1

After optimization, the aforementioned scripts were run to compare the performance of the software, unoptimized hardware, and optimized hardware, where applicable. For the ‘flower’ script, the Python code takes about 44.55 seconds to complete the convolution. The unoptimized hardware takes about 1.336 seconds, presenting a speedup of about 33 times faster than pure Python. The optimized hardware takes 0.294 seconds to run, 4.5 times faster than the unoptimized accelerator and 151.4 times faster than pure Python. Interestingly, the feature maps generated by the hardware appear to be “brighter” than those generated by Python. It is unknown whether that is a programming bug or an inconsistency with the hardware. The generated output maps (along with the original image) are shown in Appendix B.

The next test script was the “Darknet 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, 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 is negligible.

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:



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, I believe that had I been able to get Darknet fully operational on the PYNQ, it would have been worth emphasizing compatibility with Darknet in the design of the accelerator. Based on the time spent filling the buffers in the "Darknet Python" test, moving and shifting data had proven to be a major source of latency. It's likely that needing to reorganize data could have caused too much overhead and negated the advantage of using a hardware accelerator rather than just designing the accelerator around the framework. That being said, I had wasted a substantial amount of time 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," which is unfriendly for High Level Synthesis as it requires a large amount of area and does not easily allow for use of streaming (if at all). Ironically, the final code that I ended up with 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.

Subsequent designs prior to the current implementation were slight improvements but 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. 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 more control. All of these mistakes had consumed a lot of time that could have been spent exploring optimizations for the design, and many of these mistakes could have been avoided by closely studying the Darknet framework and spending more time planning out the hardware rather than hastily implementing half-working designs. A visual representation of these designs can be found within Appendix C.

If more time was allowed for the project, a potential next step 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.

To run the aforementioned IPython test scripts, create a directory in the PYNQ's file system to hold the bitstream and hardware handoff files of the hardware design:

`'home/xilinx/pynq/overlays/fcl_accel/'`

Add the 'fcl_accel.bit' and 'fcl_accel.hwh' files within this subdirectory. You can run the test scripts (Jupyter Notebook files 'dma_flower.ipynb' and 'dma_darknet.ipynb') by executing each cell in the order that they appear. You may need to create the subdirectory 'flower_outputs/' in order to be able to save the output feature maps generated by 'dma_flower.ipynb'. Additionally, you will need the flower sample image 'flower128.jpg' and data files 'inputs.dat', 'weights.dat', and 'exp_output.dat' within the same directory as the Jupyter Notebooks for everything to work properly.

For running the test program written in C, copy the 'pynq' directory and its contents to the PYNQ's PetaLinux file system using 'scp', and run 'make' within the directory. Once the test program has compiled, it must be run with superuser privileges in order to be able to access the system's memory ("/dev/mem"). You can run the program by entering 'sudo ./hwfcl_test' into the command line. The program will initialize the hardware interfaces, then run the accelerator for a simple 1x1 kernel convolution, again with the Darknet layer, and finally the software implementation for the Darknet layer. The timings for each run are shown and the last two are compared.

In addition to these test scripts, the code used to synthesize the current design using High Level Synthesis has also been included (named 'fcl.ccp' and 'fcl.h'). All necessary files are included in a ZIP archive alongside this report.

Bibliography

[1] 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].

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

[3] J. Ideami, “Convolutional layer hacking with Python and Numpy,” *Towards Data Science*, 15-Mar-2021. [Online]. Available: https://towardsdatascience.com/convolutional-layer-hacking-with-python-and-numpy-e5f64812ca0c. [Accessed: 04-May-2022].

[4] 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].

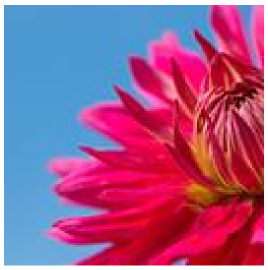
[5] L. Võsandi, “AXI Direct Memory Access,” *Lauri's blog*, 12-Dec-2014. [Online]. Available: https://lauri.xn--vsandi-pxa.com/hdl/zynq/xilinx-dma.html. [Accessed: 04-May-2022].

Appendix A: Design Optimization Data

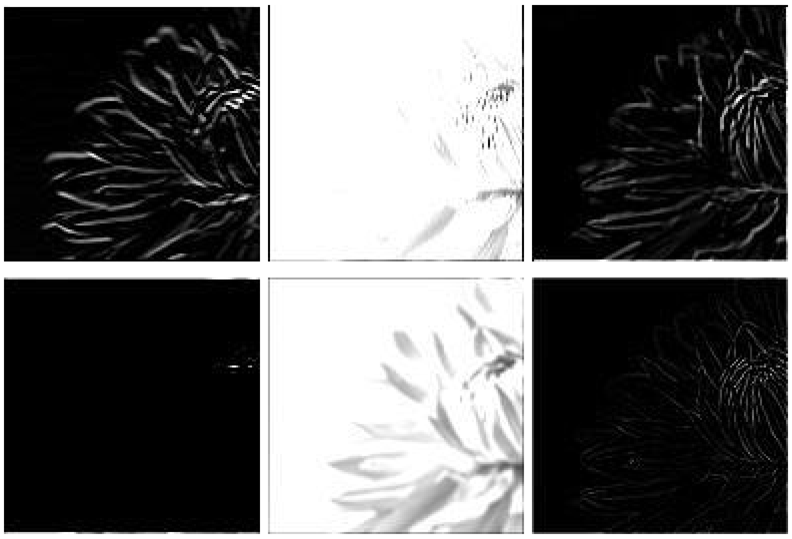
Design	Latency (cy)	Latency (ms)	BRAMs	DSPs	FF	LUT	Est Area	Desc
1	6684443	66.844	2	10	1637	4531	5731	Baseline
2	4531415	45.314	2	14	4484	11651	13251	Temp storage of floats are converted to 32-bit fixed point
3	4531415	45.314	2	10	4359	10945	12145	2 + use of ap_ints for loop variables
4	4081632	40.816	3	8	7213	12614	13714	3 + Complete partitioning of ker, win, and dim=1 of buf
5	1165306	11.653	2	41	10394	18585	22885	3 + All small loops (< KSIZE) are fully unrolled
6	1164183	11.642	6	40	10076	18540	23140	4 + 5
7	1162606	11.626	2	41	10727	18753	23053	5 + Attempts to pipeline buffer initialization
8	1161483	11.615	6	40	10411	18723	23323	6 + 7
9	1161483	11.615	6	10	10973	18953	20553	8 + Pipeline in ks = 1 loop
10	1143339	11.433	6	28	15503	26994	30394	9 + partial unroll of conv_x with factor 3
11	1200424	12.004	6	28	16808	29781	33181	9 + partial unroll of conv_y with factor 3

Appendix B: Flower Input Image and Feature Maps

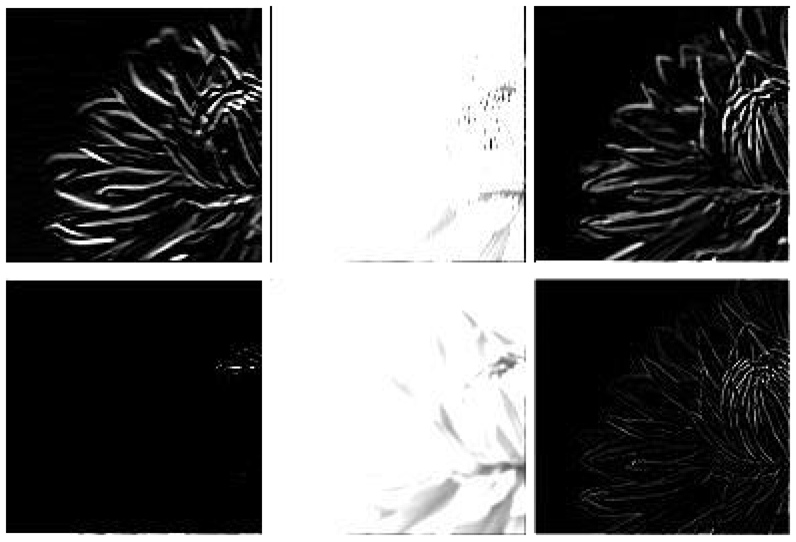
Original image:



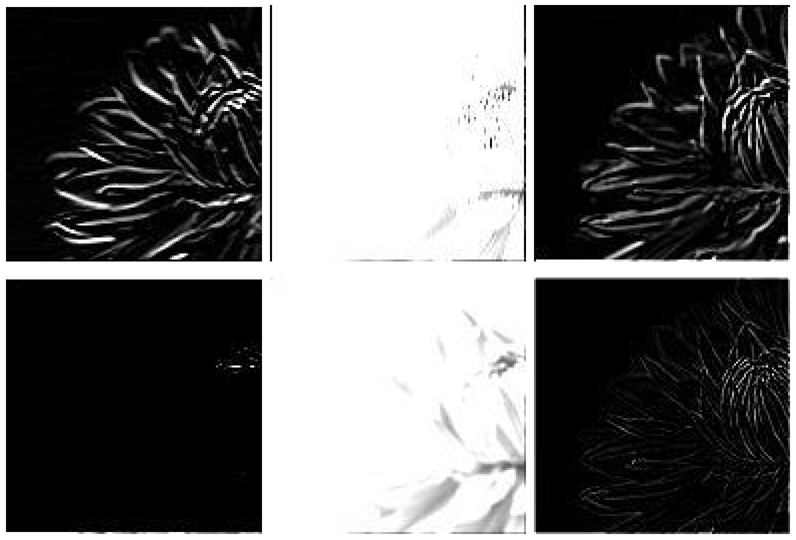
Software feature maps:



Unoptimized hardware feature maps:

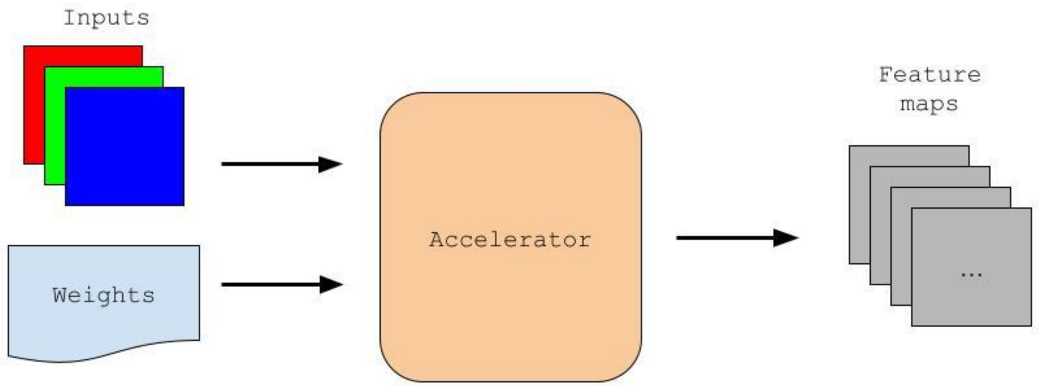


Optimized hardware feature maps:

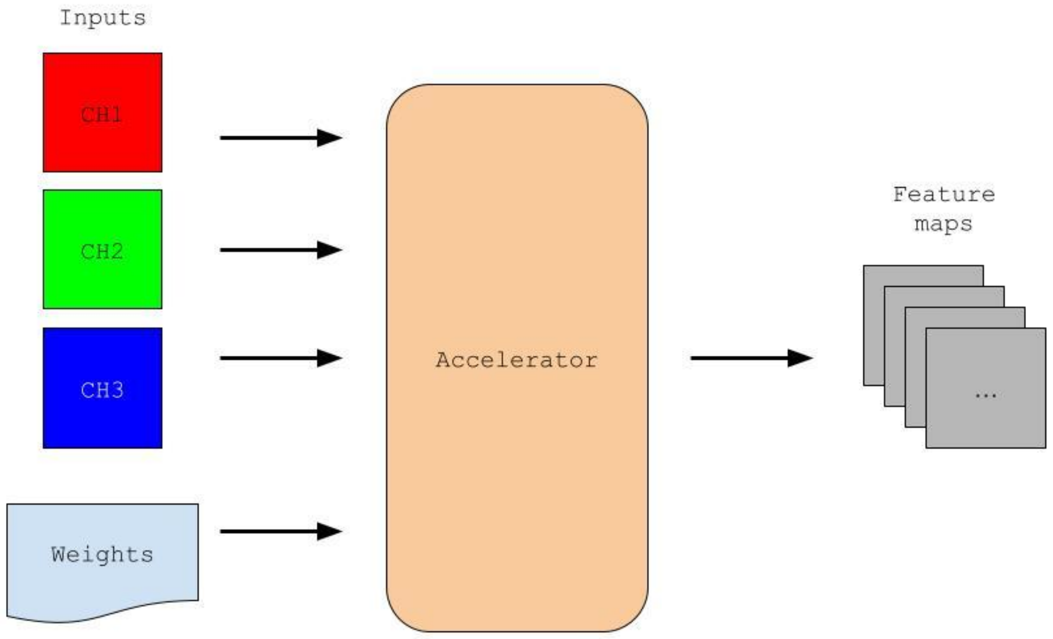


Appendix C: Design Visualizations

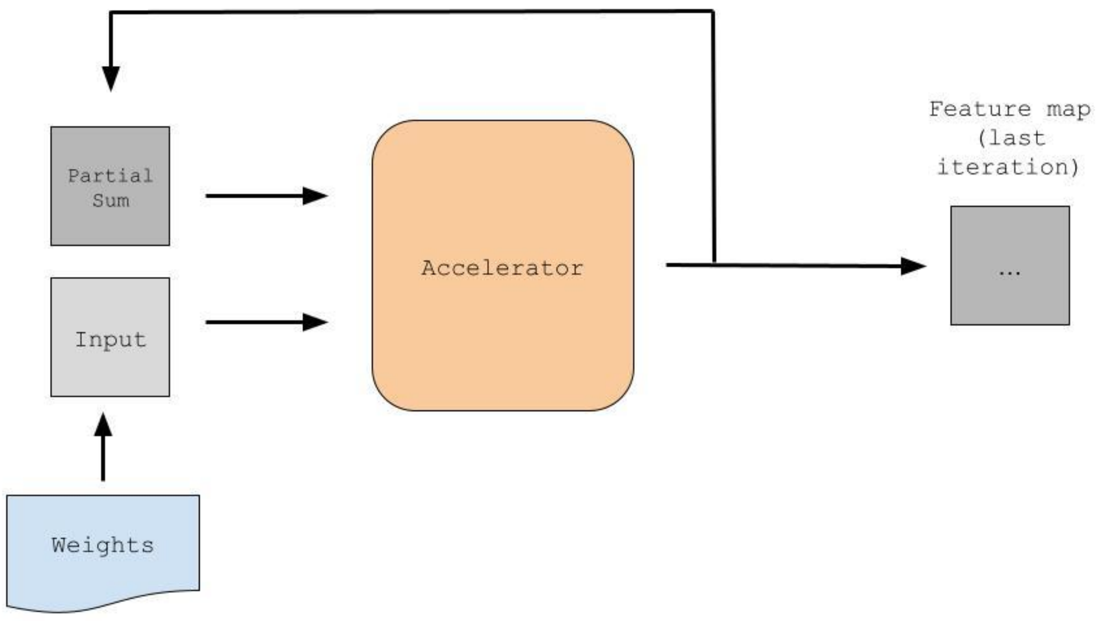
Initial design:



3-channel design:



Final design:


Partial Sum
Input
Weights
Accelerator
Feature map (last iteration)
...