Convolution Hardware Accelerator

Oscar Kellner

Goals

This is a ‘joint’ project between this class and Microprocessor-Based Design (“MBD”)

Overall goal is to create a hardware accelerator for the convolution algorithm within a convolutional neural network, and embed it within an existing machine learning framework.

On MBD-side, the framework must be modified in order to interface with the hardware.

Darknet was chosen as it was written in C and is open-source, potentially making it ideal for use within an embedded system.

As a result, many of the design decisions for the accelerator were centered around making it convenient for Darknet to read and write to the accelerator’s streaming interfaces.

HW drives convolution, SW converts image into data and loads to/from buffers

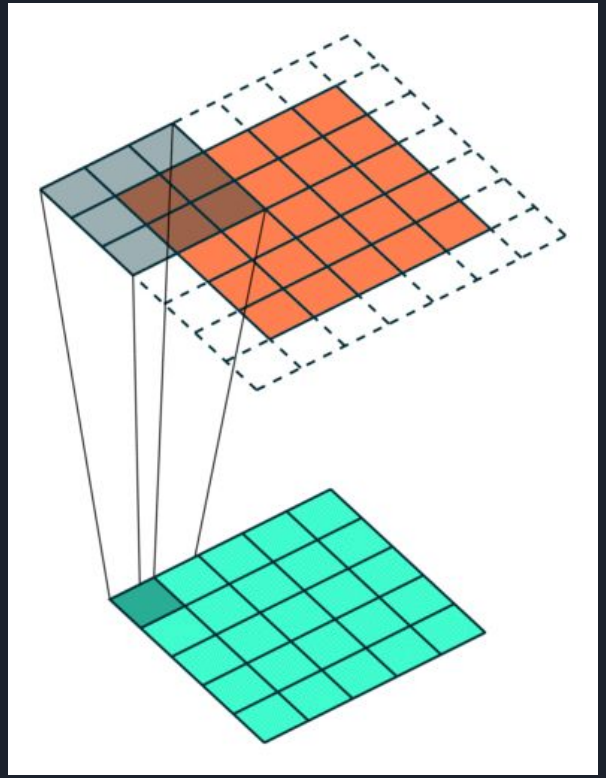
Convolution (“Conv2D” algorithm)

Weights from a filter are fit within a “window” (often 3x3) and strided across image, multiplying and accumulating values of weights and associated pixels and outputted to a “feature map”.

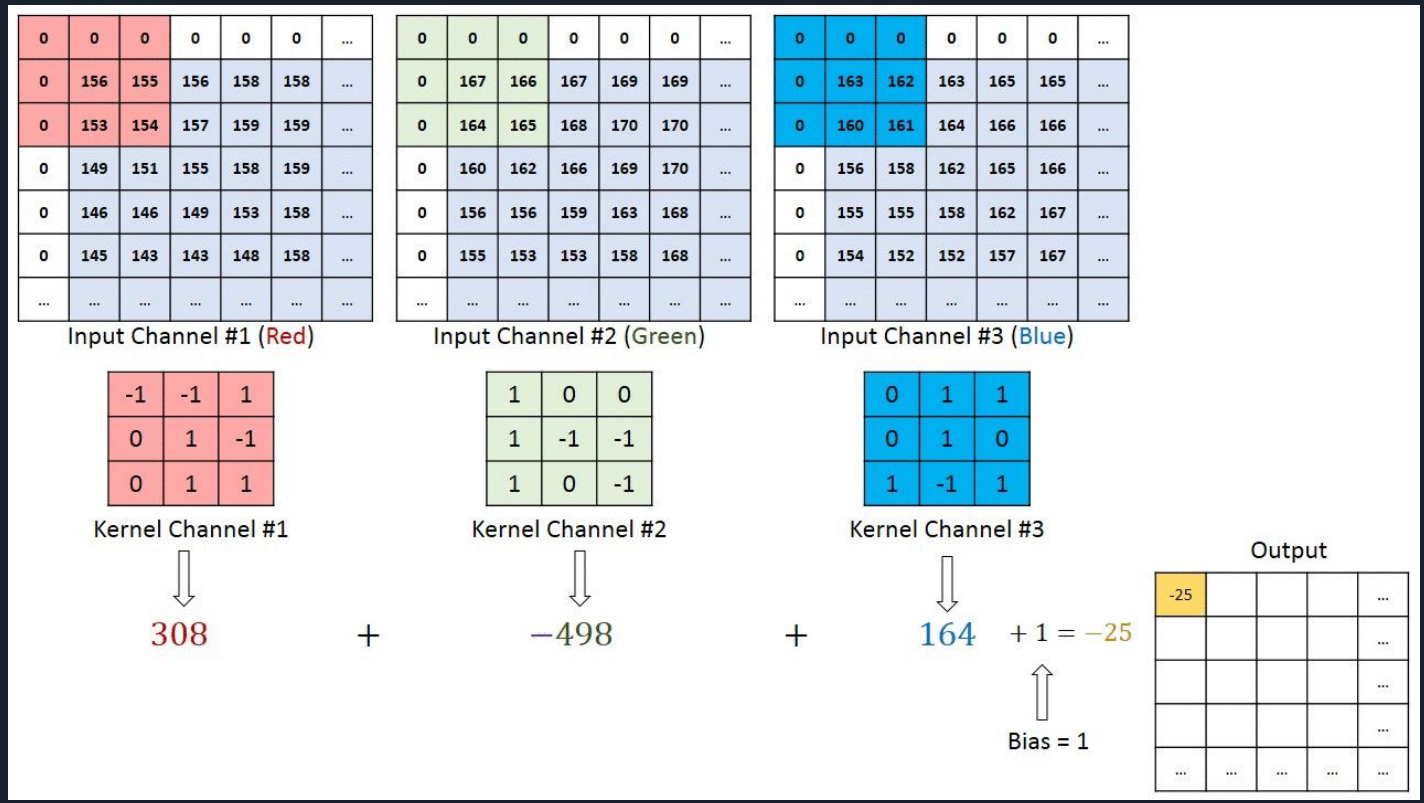
Input image in orange, output in green. Window is shown as shaded area.

Outlined region is beyond the dimensions of the image and are padded with zeros to keep the output map the same size as the input.

(No bias in this case.)

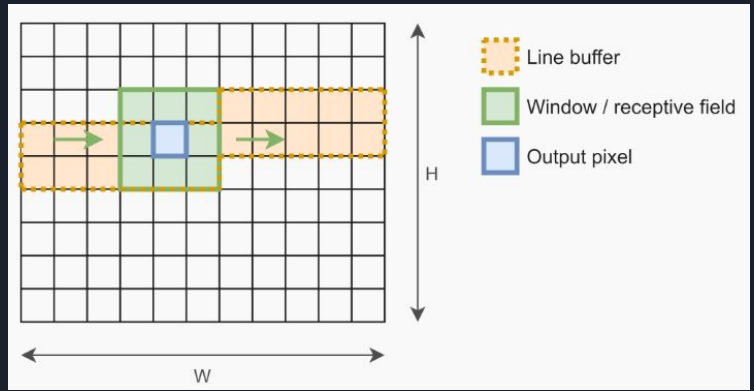


Convolution with multiple channels

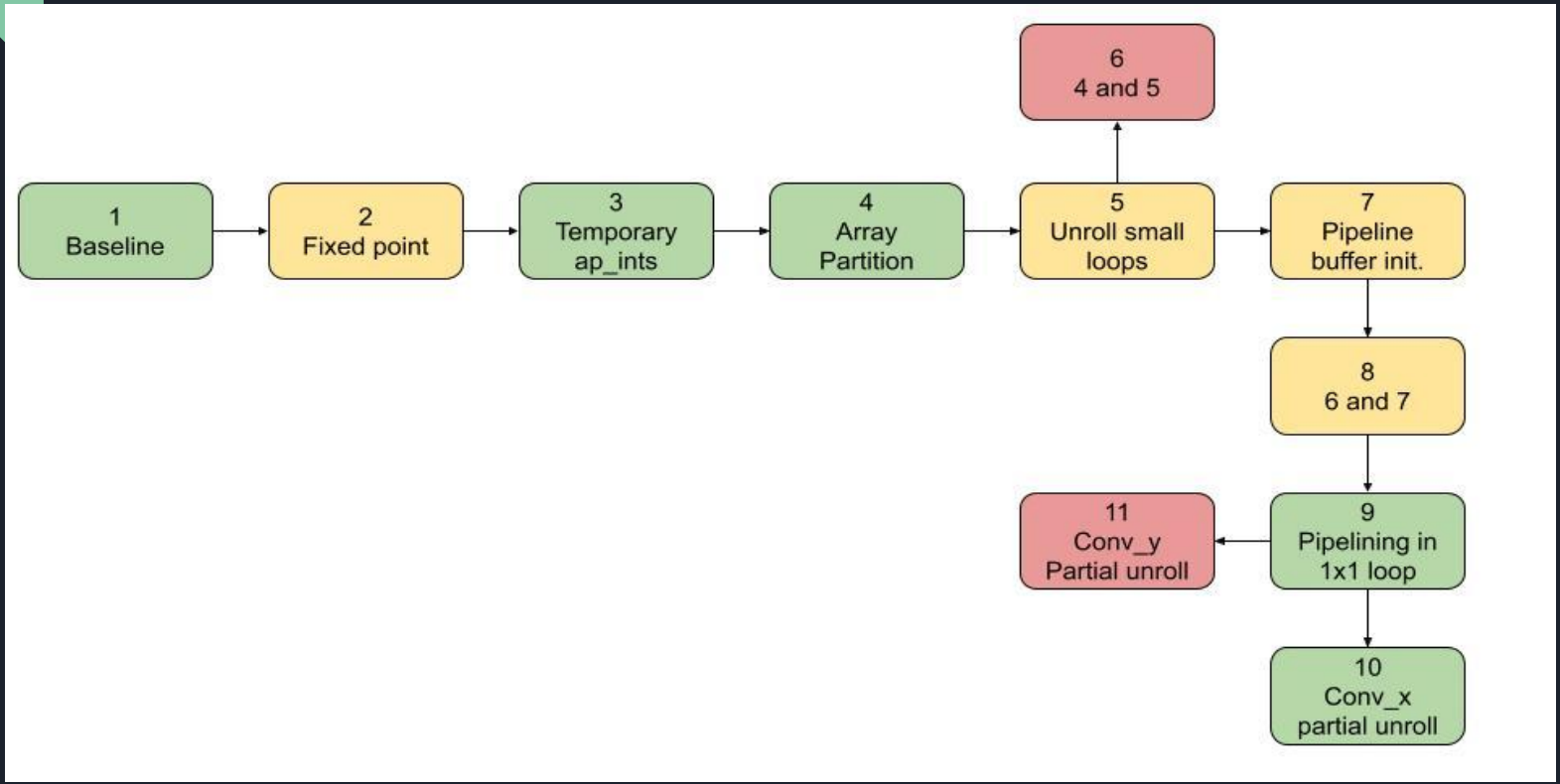


Design Evolution

1. Tried to “copy-paste” Darknet’s code into HLS
   a. Dumb, Darknet uses im2col algorithm which is unfriendly for HLS
2. Conv2D with window and line-buffer, but for 3 channels only
   a. Not very portable, only supports first layer
   b. 3 rows for line buffer are used for simplicity
3. Per channel convolution, with “partial sum” input
   a. Result of multiply accumulate is added to partial sum input before being outputted
   b. Allows for any number of channels, but more overhead for streaming data in between them



Optimizations

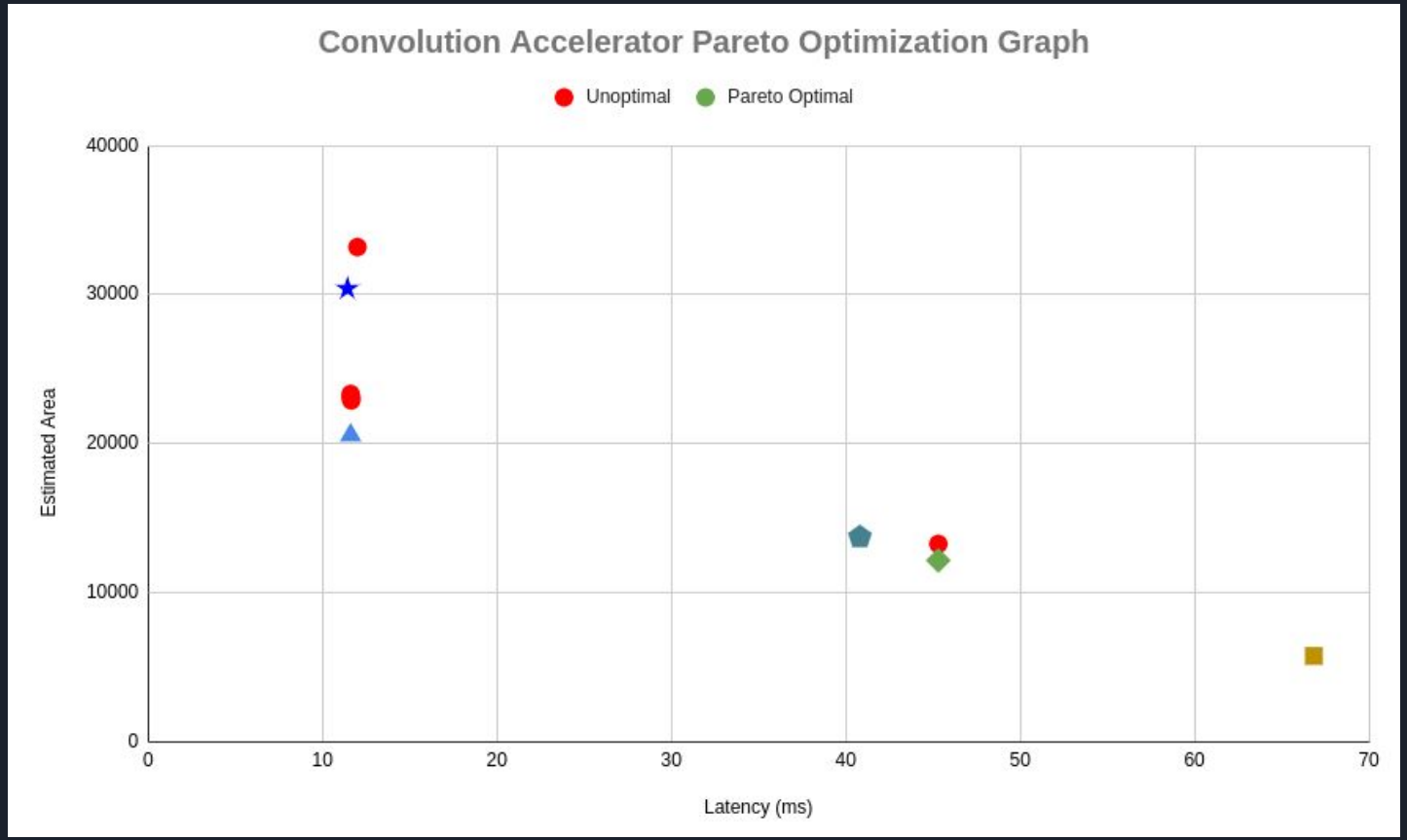


Data

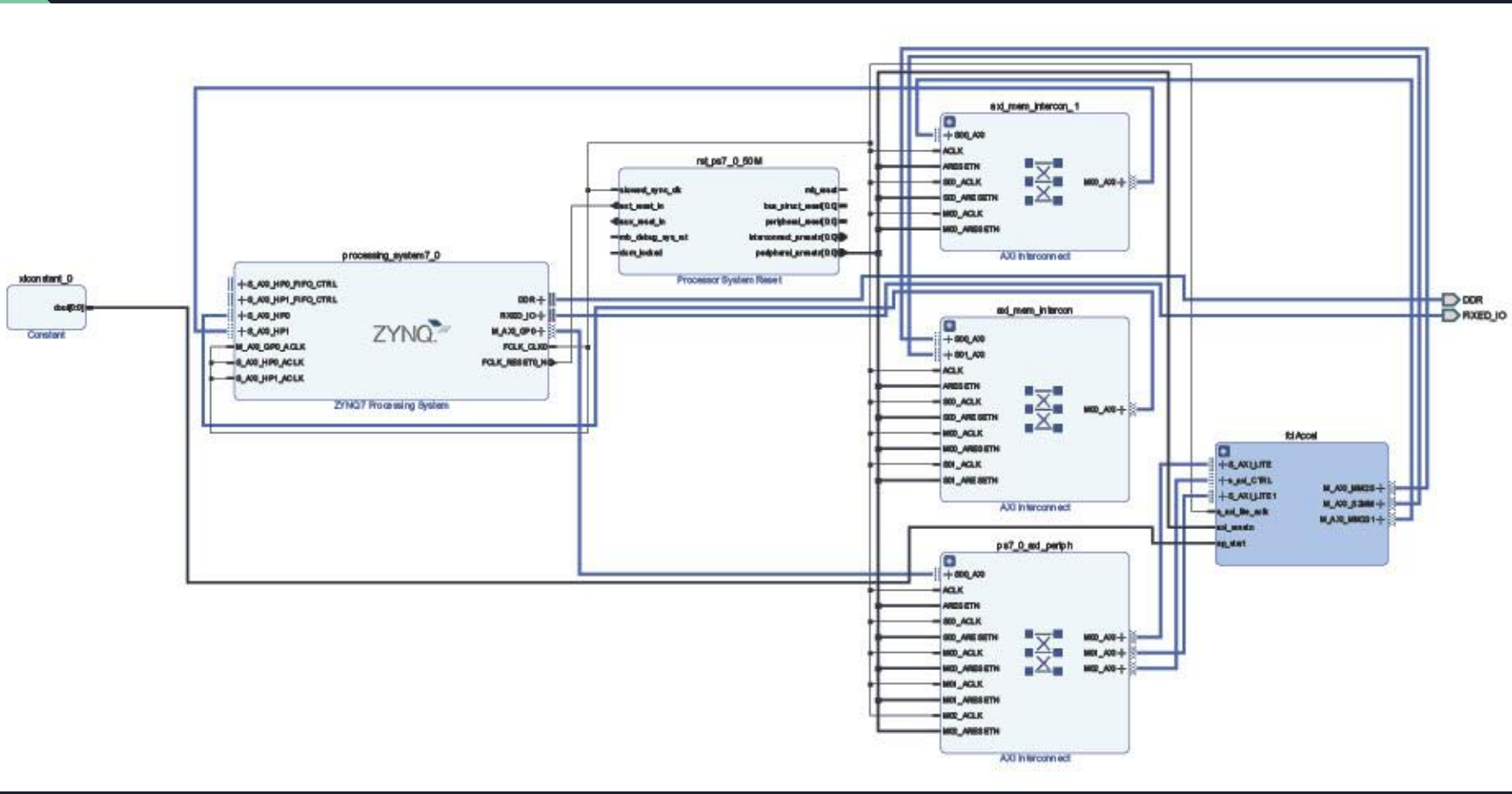
Design	Latency (cy)	Latency (ms)	BRAMs	DSPs	FF	LUT	Est Area	Desc
1	6684443	66.844	2	10	1637	4531	5731	Baseline
2	4531415	45.314	2	14	4484	11651	13251	Temp storage of floats are converted to 32-bit fixed point
3	4531415	45.314	2	10	4359	10945	12145	2 + use of ap_ints for loop variables
4	4081632	40.816	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

Ultimately, design 9 was used as the cost in area from design 10 was too much for the small decrease in latency.

Pareto Optimal Graph



Block Diagram



Testing

Performance test:

compared software and hardware implementations by convoluting predefined kernels across 3-channel image

Accuracy test:

runs hardware with data files ripped from test run from original Darknet framework and compares to expected output

Performance Comparison

Software implementation used a similar algorithm, but not exact.

- No window or line buffer as it wasn't necessary

```
def conv(data):
    #print("w_out:", w_out, "h_out:", h_out, "c_out:", c_out, "c_in:", c_in)
    output_map = np.zeros((1, c_out, w_out, h_out))
    for oi in range(w_out):
        for oj in range(h_out):
            for co in range(c_out):
                total = 0
                for ci in range(c_in):
                    kt = 0
                    for ki in range(k):
                        for kj in range(k):
                            weight = w[co, ci, ki, kj]
                            y = ki+oi*stride
                            x = kj+oj*stride
                            val = data[0, ci, y, x]
                            kt += weight * val
                    total += kt
                output_map[0, co, oi, oj] = total
    return output_map
```

Performance Comparison

Software timing: about 44.55 seconds

Unoptimized hardware:

```
Hardware execution time: 1.335648775100708 seconds.
0.05732011795043945 seconds were spent filling the buffers.
```

```
Hardware was 33.35468930763138 times faster than software.
```

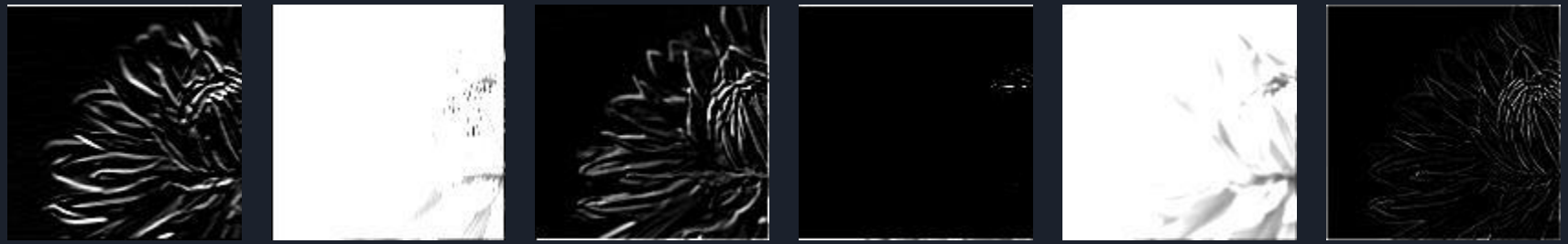
Optimized hardware:

```
Hardware execution time: 0.2942657470703125 seconds.
0.055039405822753906 seconds were spent filling the buffers.
```

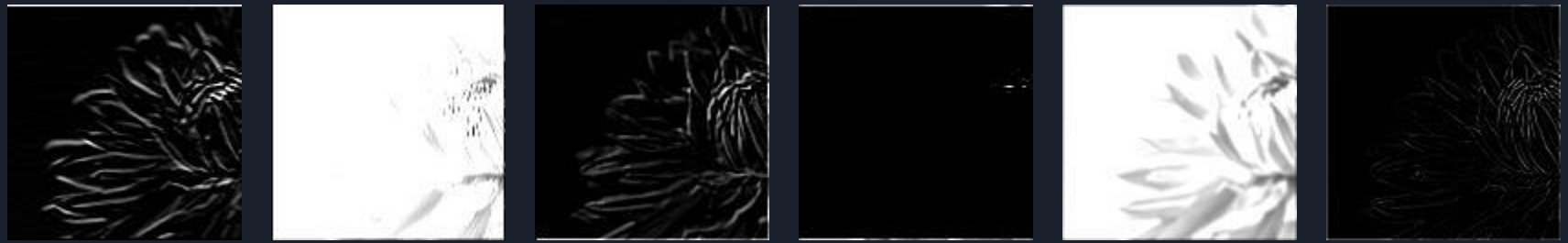
```
Hardware was 151.39427663987556 times faster than software.
```

Output Feature Map Comparison

Unoptimized hardware

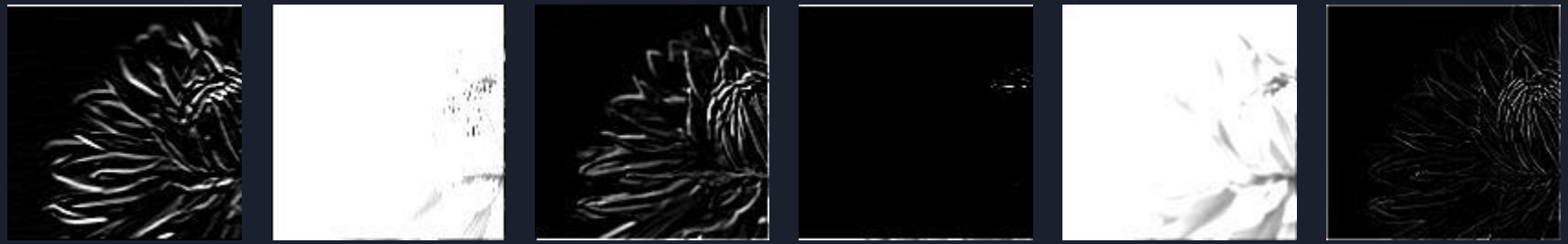


Software

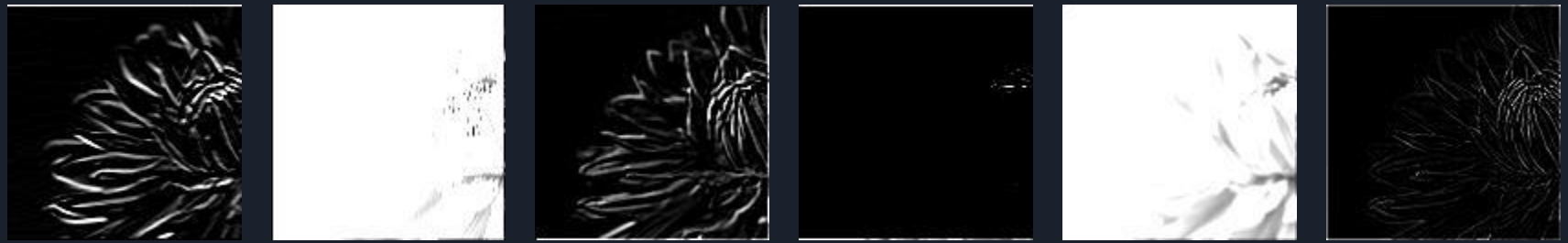


Output Feature Map Comparison

Unoptimized hardware



Optimized Hardware



Accuracy Testing

Data files created by extracting floating point values from inputs, weights, and outputs from the first layer of the tiny.cfg configuration when classifying dog.jpg.

The input files are read from and written to buffers that are sent through AXI-Stream interface.

The output generated by the hardware is compared value-by-value with the output data file, checking for discrepancies and calculating the average error (if any) for each filter.

Accuracy Testing

Unoptimized hardware:

```
Discrepancies in filter 0 : 0
Discrepancies in filter 1 : 0
Discrepancies in filter 2 : 0
Discrepancies in filter 3 : 0
Discrepancies in filter 4 : 0
Discrepancies in filter 5 : 0
Discrepancies in filter 6 : 0
Discrepancies in filter 7 : 0
Discrepancies in filter 8 : 0
Discrepancies in filter 9 : 0
Discrepancies in filter 10 : 0
Discrepancies in filter 11 : 0
Discrepancies in filter 12 : 0
Discrepancies in filter 13 : 0
Discrepancies in filter 14 : 0
Discrepancies in filter 15 : 0
Success. Output data matches data from exp_output.dat
```

Optimized hardware:

```
Discrepancies in filter 0 : 49381
Average difference: 2.464716545474773e-07
Discrepancies in filter 1 : 48960
Average difference: 2.1215896327596377e-07
Discrepancies in filter 2 : 49250
Average difference: 2.1407937602645854e-07
Discrepancies in filter 3 : 49446
Average difference: 2.491971273241159e-07
Discrepancies in filter 4 : 48796
Average difference: 1.6525763539542882e-07
Discrepancies in filter 5 : 47526
Average difference: 1.444851904207647e-07
Discrepancies in filter 6 : 47108
Average difference: 2.1247245370392747e-07
Discrepancies in filter 7 : 47289
Average difference: 2.3281328807394872e-07
Discrepancies in filter 8 : 49353
Average difference: 1.7537599666088235e-07
Discrepancies in filter 9 : 49519
Average difference: 2.3269470509772634e-07
Discrepancies in filter 10 : 47645
Average difference: 1.0916160517123454e-07
Discrepancies in filter 11 : 45455
Average difference: 1.4108407043044341e-07
Discrepancies in filter 12 : 46821
Average difference: 1.6239494071308534e-07
Discrepancies in filter 13 : 45597
Average difference: 1.5567974840117292e-07
Discrepancies in filter 14 : 46479
Average difference: 1.752641023233837e-07
Discrepancies in filter 15 : 47199
Average difference: 1.7675597367227277e-07
Failure. There were 765824 mismatched values compared to exp_output.dat
```

Accuracy Testing

Unoptimized hardware will replicate output perfectly but optimized doesn't?

Use of fixed-point values in optimized hardware may result in a slight compromise in accuracy.

But what about performance?

Unoptimized hardware:

```
Hardware execution time:  12.059700012207031
1.7432472705841064 seconds were spent filling the buffers.
```

Optimized hardware:

```
Hardware execution time:  3.5553882122039795
1.7212183475494385 seconds were spent filling the buffers.
```

About 3.4 times speedup after optimization, 5.6 times when discounting loading overhead

Retrospection

- Too much time wasted trying to hastily throw stuff together instead of spending time researching first
  - First confirmed working design was created about 2 weeks ago
  - Wasn't too aware of difference between existing algorithms before start of this project
- Too focused in compatibility
  - Darknet's algorithm was designed for software in mind, so the only compatibility that mattered was how inputs and outputs were organized
  - Most of the time was spent rewriting code and validating against original Darknet output, time that could have been spent getting a "baseline" working to avoid future issues in implementation
  - Didn't implement fixed-point until yesterday
- Optimized design still hilariously slow
  - Takes hardware 3.55 seconds to complete only first layer, Lenovo T580 laptop can finish inferencing entirely with only CPU in 0.12 seconds
  - Large latency can likely be attributed to latency filling buffers, latency from streaming data in between channels, and possibly from other areas of unoptimized design

Future Work

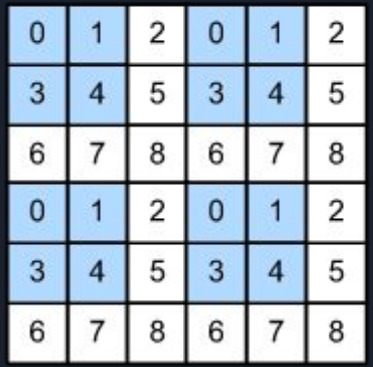
- HLS-side basically done
  - May explore some other minor options in HLS optimization
  - Explore discrepancy in software and hardware feature maps

- MBD-side still requires integration
  - Use PYNQ static library to help allocate memory for DMA
    - Originally going to use udmabuf kernel driver but can't (cross-)compile
  - Write user-level driver to be called within Darknet to interface hardware
  - Run hardware accelerator if constraints match, Darknet's im2col if not
  - Compare performance between with and without accelerator

If I had more time...

Explore more “radical” methods to parallelize convolution

- Split input image into equal parts and convolute over them
- Research better methods of convolution for hardware



Keep data more local to reduce streaming and loading overhead

- Redesign a modified version of “design 2” (per filter instead of per channel)
- “Loopback” output data back into input data for partial sum

Expand capabilities of accelerator to allow variation in other parameters (stride, etc)

References

https://medium.datadriveninvestor.com/convolutional-neural-networks-3b241a5da51e

https://m-alcu.github.io/blog/2018/01/13/neural-layers/

https://towardsdatascience.com/convolutional-layer-hacking-with-python-and-numpy-e5f64812ca0c

https://pjreddie.com/darknet/

https://basile.be/2019/03/18/a-tutorial-on-non-separable-2d-convolutions-in-vivado-hls/