Exercise 04 - Midterm Report
Design and Optimization of a CNN Accelerator

Sarah Rerecich
402774

April 3, 2025

1 Introduction

This report documents the design and optimization process for a hardware accelerator targeting a CNN-based dog/cat image classifier. The goal is to apply techniques such as coefficient and row caching, and computation parallelization to improve the initial HLS implementation.

2 Step-by-Step Implementation and Results

2.1 Step 1: Basic Accelerator Integration

The basic accelerator mimics the functionality of the provided software implementation. When run on the Pynq board, the time for one image classification is approximately 88 seconds. Including the bias and ReLU operations in the accelerator improved the performance slightly, reducing the time for one classification to about 86 seconds.

2.2 Step 2: Caching Filter Coefficients

By locally allocating a cache for the filter coefficients within the hardware accelerator itself, the number of memory accesses performed by the accelerator to the external DDR DRAM is reduced by a factor of about two.

The implementation of the caching is simple: one time at the beginning of the filter, an array is initialized with the values of the filter coefficients. It uses the exact same syntax to calculate the coefficients as the initial implementation. Then, as before, the filter values are assigned in the inner loop based on the now pre-loaded coefficients.

2.3 Step 3: Caching Input Rows

The next optimization technique allows Vitis to infer a burst transfer. Similar to the caching of filter coefficients, the hardware accelerator is modified to "pre-read" all of the pixel values from a row. This removes the redundant additional memory accesses to adjacent pixels as the convolution window slides across the row.

This step introduced a rather complex bug due to memory allocation and resource usage limitations that the testbench was unable to detect. The original implementation passed the testbench

in Vitis, but consistently produced incorrect results on hardware. It was with careful deliberation with TA Rubén that the problem came to light: my initial implementation used more BRAM than the Pynq board had available, which resulted in seemingly random - though consistent - behaviour regarding where data was being written. The solution was two-fold: initialize the row buffer to be two dimensional instead of three, and define the size as the maximum product of number of channels and width size *that the convolver actually uses*. The initial implementation is summarized below for reference:

```
// the x and y loops are essentially the position of the "window" moving across and down (x and y)
// so first along the height (rows - y), then across the width (columns - x)
for (ap_uint<32> y = 0; y < (inputHeight - 2); ++y) {
    TFXP row_buffer[3][MAX_CHANNELS][MAX_WIDTH]; // initialize buffer for three rows

    // then, now caching the rows (y), we move along each row and store accordingly
    for (ap_uint<32> cy = 0; cy < convHeight; ++cy) {
        ap_uint<32> iy = y + cy;
        // ...which we do for each channel
        for (ap_uint<32> iChannel = 0; iChannel < numChannels; ++iChannel) {
            // ...and each component of the row, which of course moves in the x direction
            for (ap_uint<32> x = 0; x < inputWidth; ++x) {
                row_buffer[cy][iChannel][x] = *(input + iChannel * inputWidth * inputHeight + iy * inputWidth + x);
            }
        }
    }
}
```

Listing 1: Memory-Exhaustive Row Caching Implementation

```
// so first along the height (rows - y), then across the width (columns - x)
for (uint32_t y = 0; y < (inputHeight - 2); ++y) {
    TFXP row_buffer[3][4064]; // initialize buffer for three rows
    #pragma HLS ARRAY_PARTITION variable=row_buffer complete dim=1

    // then, now caching the rows (y), we move along each row and store accordingly
    for (uint32_t cy = 0; cy < convHeight; ++cy) {
        uint32_t iy = y + cy;
        // ...which we do for each channel
        for (uint32_t iChannel = 0; iChannel < numChannels; ++iChannel) {
            // ...and each component of the row, which of course moves in the x direction
            for (uint32_t x = 0; x < inputWidth; ++x) {
                #pragma HLS PIPELINE II=1
                row_buffer[cy][iChannel * inputWidth + x] = *(input + iChannel * inputWidth * inputHeight + iy * inputWidth + x);
            }
        }
    }
}
```

Listing 2: Final Row-Caching Implementation

2.4 Step 4: Parallelizing Output Filters

This step involves the computation of multiple output filters at the same time. Increasing the number of hardware-compatible storage caches allows the output filters to be accessed simultaneously instead of sequentially, thus reducing the speed of the overall task by a factor of about n, where n is the number of parallel filters.

3 Performance and Resource Usage

Design Variant	**Time (s)**	**LUTs**	**FFs**	**BRAMs**	**DSPs**	**Cost (C_R)**
Software Only	1.610	–	–	–	–	–
Initial Hardware	87.958	3935	6729	1	64	0.17946
Add Biases and ReLU	86.413	7936	12795	12	64	0.21411
Caching Coefficients	52.200	18696	21317	12	59	0.33500
Caching Rows	29.297	8442	13107	27	49	0.28664
Parallel Output Filters	19.890	20257	14352	48	64	0.40755

Table 1: Design comparison with averaged execution time and resource cost (C_R) based on FPGA capacity. Pareto-optimal designs are highlighted.

Cost Calculation Formula:

$$CR = \frac{25}{140} \cdot \#BRAM + \frac{25}{220} \cdot \#DSP + \frac{25}{53200} \cdot \#LUT + \frac{25}{106400} \cdot \#FF$$

4 Continuations

Upon introduction of the row caching, the first layer of the convolution increased to approximately 15 seconds, where it was approximately 3 seconds in the previous step. Despite this, the overall time still improved due to the time reductions in the other convolutional layers. I believe that a further modified version of the row caching would bring the first layer convolution time down to under 2 seconds, resulting in the overall computation for one image taking about 6 seconds.