COMP4601 CNN Project Report

Kenneth Law **z5363506**
Sicheng (Stefan) Jin **z5317861**
Ezekiel Tay **z5378748**

PROJECT OBJECTIVES

This project investigation delves into the potential benefits that FPGAs bring to accelerate Convolutional Neural Networks (CNN) inferences. CNNs have seen widespread use in various applications involving computer vision. Many of these tasks often fall under the categories of image classification and target detection/tracking.

Although CNNs have relatively cheap inference engines (compared to recent transformer-based architectures), they still require rather large amounts of computing power and resources to compute the convolution layers, making them difficult to deploy on low energy devices with adequate execution speed. We aim to prove that, by leveraging FPGA parallel processing power and high throughput, such improvements can offer a step closer to low energy devices deployment with fast inference speeds a much easier reality.

In this project, we implemented a “ToyCNN” model on a Kria KV260 board to design a solution where both a general-purpose CPU and a specialised convolution accelerator core on the FPGA work in tandem to achieve faster execution speeds to classify handwritten digits in the MNIST dataset. This report will primarily cover the design implementation phase of development and the results that follow with discussions. A timeline of project milestones will also be shown.

DESIGN IMPLEMENTATION

ToyCNN Architecture

Due to our backgrounds in working with neural networks in the past, we determined that using common open-source models such as LeNet and AlexNet would be overkill for this task and instead opted for a simpler architecture. Removing redundant over complexity and favouring simplicity. As a result, we designed ToyCNN that consists of the main features of a typical CNN architecture.

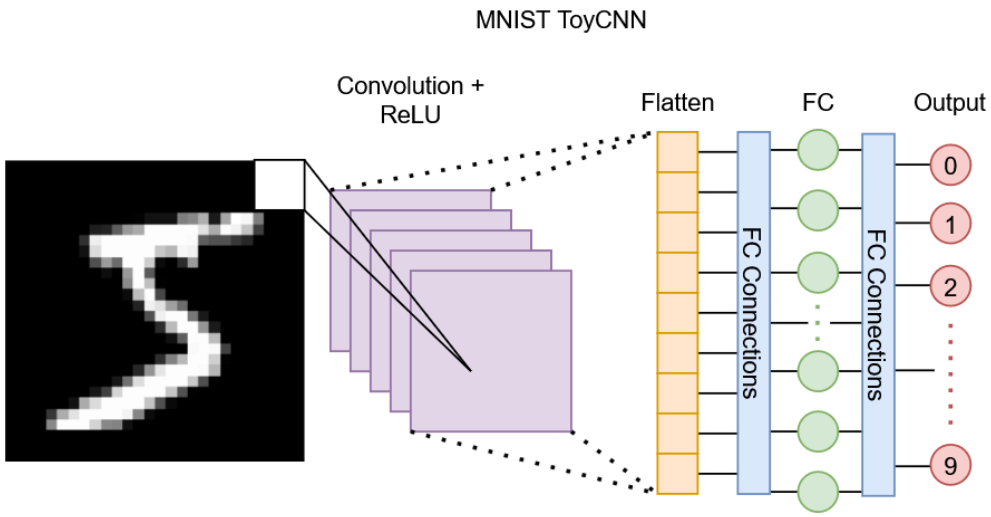

Figure 1. ToyCNN Architecture

The architectural parameters of ToyCNN are as follows

- **Convolution**
  - **Filters:** 8, 3x3 Kernels with 1 input channel
  - **Parameters:** 72 weights and 8 biases
  - **ReLU:** Linearly rectifies the output feature values
  - **Output:** An 8x26x26 feature map
- **Flatten**
  - Flattens the output feature map to a 1D array with 5,408 elements
- **Fully Connected Layer (FC)**
  - The FC layer has 10 neurons (each representing digit classes 0-9), with each neuron having connections to all elements in the flat layer.
  - **Parameters:** 5,408x10 = 54,4080 weights and 10 biases

Leading to a total of 54,170 parameters.

ToyCNN training

The CNN was trained on the MNIST training dataset which consisted of 60,000 images of handwritten digits in 28x28 grey scale. The PyTorch library was leveraged to quickly trained ToyCNN on the GPU. Eventually achieving 99.88% accuracy, with F1-score 0.9987 on the

training set. When evaluated on the test set consisting of 10,000 images, it achieved 97.82%, with F1-score 0.9780.

```
Epoch 23/25, Accuracy: 99.86%, F1 Score: 0.9986, Loss: 0.0012
Epoch 24/25, Accuracy: 99.89%, F1 Score: 0.9989, Loss: 0.0003
Epoch 25/25, Accuracy: 99.88%, F1 Score: 0.9987, Loss: 0.0274
  model = torch.load("weights/model_weights.pth").to(device)
Test set evaluation: Accuracy = 9782/10000 97.82%, F1 Score = 0.9780
```

Figure 2. Results of ToyCNN's training and evaluation.

Inference function pseudo-code

The trained parameters were then saved into C++ header files for the inference function to be deployed on the board. With these saved parameters, the network follows the pseudo code, which follows the operations of the ToyCNN in the PyTorch implementation in Figure 3.

```
// STEP 1: Initialize Network Parameters
INPUT: image[28×28]          // Grayscale pixel values (0-1)
WEIGHTS:
  - conv_kernels[8][3×3]     // 8 feature detectors
  - conv_biases[8]           // One bias per kernel
  - fc_weights[10×5408]      // Fully connected weights
  - fc_biases[10]            // Output biases (one per digit class)

// STEP 2: Convolution Layer
FOR each kernel k in 8_kernels:
  FOR each position (i,j) in output[26×26]:
    sum = 0
    FOR each position (ki,kj) in 3×3_kernel:
      sum += image[i+ki][j+kj] × conv_kernels[k][ki][kj]
    conv_output[k][i][j] = ReLU(sum + conv_biases[k])

// STEP 3: Flatten Feature Maps
flattened_features = FLATTEN(conv_output[8×26×26])
// Result: 1D array of size 5,408 elements

// STEP 4: Fully Connected Layer
FOR each digit class i in 0-9:
  logits[i] = 0
  FOR each feature j in flattened_features:
    logits[i] += flattened_features[j] × fc_weights[i][j]
  logits[i] += fc_biases[i]

// STEP 5: Make Prediction
predicted_digit = ARGMAX(logits[0-9])

RETURN predicted_digit
```

Figure 3. Pseudo code of ToyCNN's inference function

Baseline Implementation (SW only)

The baseline implementation is the C/C++ equivalent of the ToyCNN inference function, which is run entirely on the Kria board's CPU (the code is available in main.cpp in the vitis_files).

In summary, the SW-only implementation achieved identical accuracy (as it should) and took 43 seconds to classify 10,000 images (average 4.3ms/image). With almost 56% (avg. 2.4ms/image) of the CPU runtime dedicated to the convolution layer.

```
Running Unoptimised toyCNN (Entirely PS-side)
CWD on target: /media/sd-mmcblk1p2
Succesfully opened ./mnist_test.csv.
Running 10000 images through the toyCNN, with the CONV layer computed in the PS.
Batch inference complete!
Time elapsed: 43081 ms
Accuracy: 9782/10000 correct 97.82
-------------Running Averages of each major operation-------------
Loading image into memory:              5.14%
Convolution layer on the PS:            55.98%
Flatten layer:                          3.69%
FC layer:                               35.19%
------------------------------------------------------------------
```

Figure 4. Performance of SW implementation

```
// Load image into local buffer
auto t0 = std::chrono::high_resolution_clock::now();
for (int i = 0; i < IMG_SIZE; i++) {
    // Load pixels row into buffer
    for (int j = 0; j < IMG_SIZE; j++) {
        token = strtok(NULL, ",");
        if (!token) {
            printf("Invalid row: missing pixels!\n");
            fclose(file);
        }
        float pixel_val = atof(token) / 255.0f;
        input[i][j] = pixel_val;
    }
}
auto t1 = std::chrono::high_resolution_clock::now();
// Perform PS-side convolution
float output_fm[NUM_KERNELS][OUT_SIZE][OUT_SIZE];
conv2d(input, output_fm);
auto t2 = std::chrono::high_resolution_clock::now();

// Perform remaining layers on PS
float flat[NUM_KERNELS * OUT_SIZE * OUT_SIZE];
float logits[FC_OUT];

auto t4_start = std::chrono::high_resolution_clock::now();
flatten(output_fm, flat);
auto t4_end = std::chrono::high_resolution_clock::now();

auto t5_start = std::chrono::high_resolution_clock::now();
fc(flat, logits);
auto t5_end = std::chrono::high_resolution_clock::now();

int pred = argmax(logits, 10);
printf("Pred: %d Label: %d\n", pred, label);
if (pred == label) correct++;
total++;
```

Figure 5. Code snippet of SW implementation

HW Accelerator Implementation (HW+SW)

To speed up the computations of the convolution layer, we must leverage the parallel processing power of FPGAs. Two factors were considered for optimisation. Computation and data access.

Computation optimisations

Convolution Multiply Accumulator (CMAC)

The CMAC aims to speed up the dot product operation of the convolution process. Reducing 19 operation steps down to only 5 steps. Leading to a 3.8x speedup to the dot product calculation. This is achieved by calculating the multiplications in parallel and then accumulating the results (including the kernel bias) through an adder tree. This design is similar to standard Multiply-Accumulate (MAC) units but includes another addition in the 2nd step for the kernel bias and a leftover product of the 9th multiplication. The steps are also pipelined with II=1 for ideal throughput.

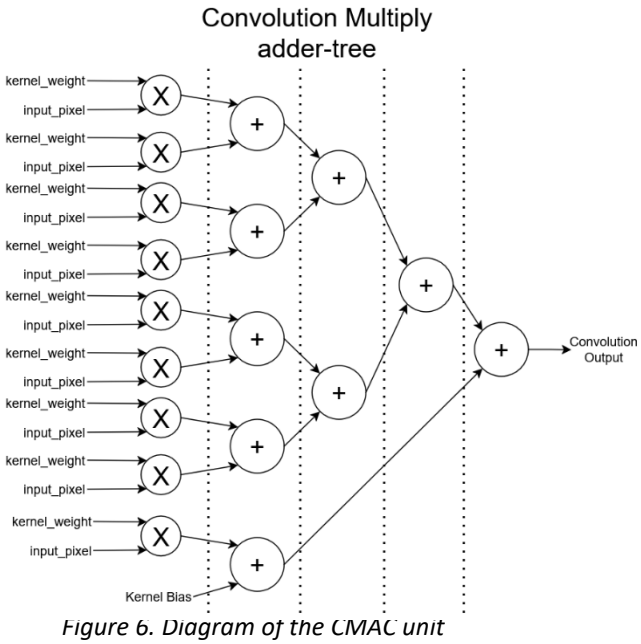

Figure 6. Diagram of the CMAC unit

The 3-line image buffer

Another optimisation is to situate the kernels on top of the image, then scrolling the image line by line under the kernels. Achieving the same effect as sliding the kernels would. This is achievable through a 3-line image buffer that exposes a 3x3 image patch, removing extra logic in the CMAC units for sliding their kernels across the image. Thus, removing needless duplication of operations.

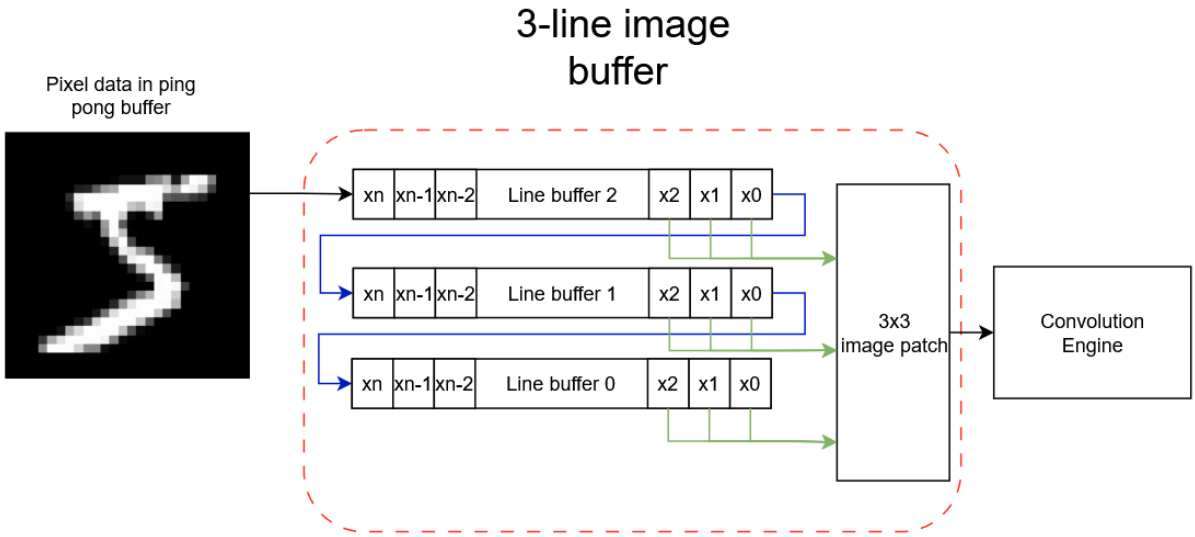

Figure 7. Diagram of 3-line image buffer

This buffer is also pipelined with II=1, to sufficiently supply the fully pipelined CMAC units.

Data Access Optimisations

While the 3-line buffer solution above also falls under this category, other optimisations were done to reduce latency caused by data reads/writes.

Shared Memory Access

The AXI-Lite interface was used in this project, since it was deemed adequate for our design. To minimise latency caused by PS-PL data transfer, the input image data array was partitioned completely to open access to all elements to be written to in shared memory by the PS. This led to each row of the image being stored in allocated 128B blocks. Totalling to ~3.6KB of shared memory.

Similarly, the same was done for the output feature map. With each feature map occupying 4KB blocks, totalling 32KB.

Hence, allowing the PS software to leverage bulk transfers via memcpy.

```
0x0080 ~
0x00ff : Memory 'input_image_0' (28 * 32b)
         Word n : bit [31:0] - input_image_0[n]
0x0100 ~
0x017f : Memory 'input_image_1' (28 * 32b)
         Word n : bit [31:0] - input_image_1[n]
0x0180 ~
0x01ff : Memory 'input_image_2' (28 * 32b)
         Word n : bit [31:0] - input_image_2[n]
0x0200 ~
0x027f : Memory 'input_image_3' (28 * 32b)
         Word n : bit [31:0] - input_image_3[n]
```

Figure 8. Snippet of register mappings of the input image array.

```
0x1000 ~
0x1fff : Memory 'output_feature_map_0' (676 * 32b)
         Word n : bit [31:0] - output_feature_map_0[n]
0x2000 ~
0x2fff : Memory 'output_feature_map_1' (676 * 32b)
         Word n : bit [31:0] - output_feature_map_1[n]
```

Figure 9. Snippet of register mappings of output FM

Parallel data read

Switching over to the local side of the FPGA design, the convolution layer weights and biases are stored locally on the chip and are fully partitioned to allow all the CMAC units to read the parameters in parallel for computation.

```
    // Partition weights to be accessible in parallel
#pragma HLS ARRAY_PARTITION variable=conv_W type=complete dim=1
#pragma HLS ARRAY_PARTITION variable=conv_W type=complete dim=2
#pragma HLS ARRAY_PARTITION variable=conv_W type=complete dim=3
#pragma HLS ARRAY_PARTITION variable=conv_B type=complete
```

Figure 10. HLS code partitioning the convolution parameters

Parallel output writers

Since the output feature map consists of over 5,408 values to be written to memory, 8 writers work in parallel to write each kernel's feature map to memory. Leaving it to the AXI modules to write the values to shared memory as efficiently as possible. If fully achieved, this effectively reduces latency from writing the values serially by 8x.

```
    // Once the feature map finally computed,
    // upload local feature map buffer to the external memory feature map (via AXI)
    writeFM_loop: for (int k = 0; k < NUM_KERNELS; k++) {
#pragma HLS UNROLL // 8 writers
        for (int i = 0; i < OUT_SIZE; i++) {
            for (int j = 0; j < OUT_SIZE; j++) {
#pragma HLS PIPELINE II=1
                output_feature_map[k][i][j] = local_fm[k][i][j];
```

Figure 11. HLS code for writing the output FM to shared memory.

Overall FPGA Design

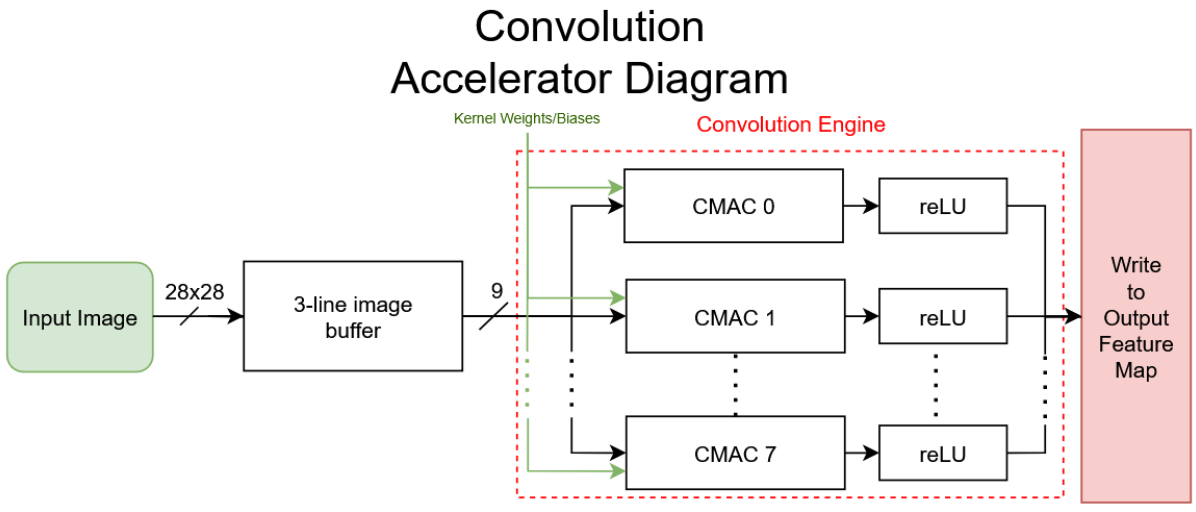

Figure 12. Convolution Accelerator Design

With all these optimisations integrated with each other, we get the FPGA design above to accelerate the convolution process.

Synthesis

After synthesis, the following performance and resource estimates were generated. With the target clock period of 10ns. Estimating that the whole process takes 15.86us to run.

Modules & Loops	Issue Type	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)
conv_axilite		1586	1.586e4	-	1587	-	no	11	28	27	31
init_weights		75	750.000	-	75	-	no	~0	0	~0	~0
init_weights_Pipeline_VITIS_LOOP_12_1_VITIS_LOOP_13_2_VITIS_LOOP_14_3		74	740.000	-	74	-	no	~0	0	~0	~0
VITIS_LOOP_12_1_VITIS_LOOP_13_2_VITIS_LOOP_14_3		72	720.000	2	1	72	yes	-	-	-	-
init_weights_Pipeline_VITIS_LOOP_17_4		10	100.000	-	10	-	no	0	0	~0	~0
VITIS_LOOP_17_4		8	80.000	2	1	8	yes	-	-	-	-
conv_axilite_Pipeline_triline_buffer_eachPatch_loop		813	8.130e3	-	813	-	no	0	28	23	25
triline_buffer_eachPatch_loop		811	8.110e3	29	1	784	yes	-	-	-	-
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
cmac_unit_stream		23	230.000	-	1	-	yes	0	3	2	3
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_4		681	6.810e3	-	681	-	no	0	~0	~0	~0
VITIS_LOOP_147_3_VITIS_LOOP_148_4		679	6.790e3	5	1	676	yes	-	-	-	-
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_41		691	6.910e3	-	691	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_42		691	6.910e3	-	691	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_43		692	6.920e3	-	692	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_44		692	6.920e3	-	692	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_45		692	6.920e3	-	692	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_46		693	6.930e3	-	693	-	no	0	0	~0	~0
conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_47		693	6.930e3	-	693	-	no	0	0	~0	~0

Figure 13. Performance & Resource Estimates of the accelerator core.

To clarify, the ***init_weights*** only runs once at power-on to initialise the kernel parameters registers. The ***triline_buffer_eachPatch_loop*** consists of the 3-line buffer and the 8 CMAC units. The ***conv_axilite_Pipeline_VITIS_LOOP_147_3_VITIS_LOOP_148_4**** are the 8 output feature map writers.

PS Software

Compared to the baseline SW implementation, the key changes involve replacing the convolution function call with moving data to and from shared memory space and setting signals to control the HW.

The ***memcpy()*** function is used to efficiently bulk transfer data to and from shared memory space and local variables.

```
// Load image into local buffer then send to shared MM
auto t0 = std::chrono::high_resolution_clock::now();
for (int i = 0; i < IMG_SIZE; i++) {
    // Load pixels row into buffer
    for (int j = 0; j < IMG_SIZE; j++) {
        token = strtok(NULL, ",");
        if (!token) {
            printf("Invalid row: missing pixels!\n");
            fclose(file);
            return -1;
        }
        float pixel_val = atof(token) / 255.0f;
        input[i][j] = pixel_val;
    }
    // Then mem copy to the shared MM
    uint32_t *rowPtr = (uint32_t*)(axiBasePtr + INPUT_IMAGE_OFFSET_BASE + INPUT_IMAGE_BLOCK_SIZE * i);
    memcpy(rowPtr, input[i], sizeof(pixel_t) * IMG_SIZE);
}
auto t1 = std::chrono::high_resolution_clock::now();
// Start IP
*(uint32_t*)(axiBasePtr+USER_IP_ADDR_OFFSET_CTRL) = 1 << AP_START_BIT;
while(!(*(uint32_t*)(axiBasePtr+USER_IP_ADDR_OFFSET_CTRL) & 0b110)) {
}
auto t2 = std::chrono::high_resolution_clock::now();
// Perform remaining layers on PS
float flat[NUM_KERNELS * OUT_SIZE * OUT_SIZE];
float logits[FC_OUT];

// Flatten layer
auto t4_start = std::chrono::high_resolution_clock::now();
// flatten(output_fm, flat);
for (int k = 0; k < NUM_KERNELS; k++) {
    uint32_t *kernelPtr = (uint32_t*)(axiBasePtr + OUTPUT_FM_OFFSET_BASE + OUTPUT_FM_BLOCK_SIZE*k);
    memcpy(&flat[k*OUT_SIZE*OUT_SIZE], kernelPtr, sizeof(float) * OUT_SIZE * OUT_SIZE);
}
auto t4_end = std::chrono::high_resolution_clock::now();
```

Table 1. Code snippet of PS-side software. Full code available in main.cpp in vitis_files.

PERFORMANCE RESULTS

SW-ONLY VS HW+SW

The following results were tabulated based on a test run on the Kria KV260 board via Vitis.

Metric	SW-Only	HW+SW
Time Elapsed (ms)	43,081 (~43.1 secs)	23,647 (~23.6 secs)
Avg. Time per Image	4.31ms	2.36ms
Loading image into memory	5.14%	9.25%
Convolution Layer	55.98%	**0.72%**
Avg. Time in Conv Layer	2.41ms/image	**17.02us/image**
Flatten Layer	3.69%	25.82%
FC Layer	35.19%	64.21%

Table 2.Performance Comparison of SW-only vs HW+SW.

Results Discussion

From the results alone, it is evident that the FPGA accelerator core significantly cuts down the compute time of the convolution layer, to the point that loading images from disk (SD card in this case) and the remaining layers now take up the majority of CPU time. For the HW+SW optimised design, it averages 2.36ms/image, with only 17.02us on average for the FPGA design to compute the feature maps, closely meeting the synthesis estimates (15.86us). Suggesting a little over **140x** speed up of the convolution operation. Resulting in overall CNN speedup factor of **~82%**. This is primarily due to **Amdahl's law** limiting the speedup effects of the accelerator core asymptotically. When extending this solution to deeper models with multiple convolution layers, where SW-only implementations would result in over 90% of runtime for convolution, this acceleration would improve those networks by much higher factors.

Another limiting factor was in fact the flatten layer on the SW-side. Which primarily involves moving data in shared memory over into a local 1D array to setup for the FC layer.

PROJECT TIMELINE

Week 1-5: Initial investigation, building the toyCNN model and experimenting with it

Week 6: preparing for the initial presentation and seminar

Week 7: Running the initial network (without accelerating) on the board, with PS+PL code

Week 8: Implementing the aforementioned optimisation strategies and testing

Week 9: Further testing to ensure the code and configuration were correct

Week 10: Working on final presentation and this report

FUTURE WORKS & CONCLUSION

Potential Future works

Over the course of this investigation, there were some optimisation considerations that were originally planned to be looked into but never saw the light of day due to time constraints or post-results evaluation.

More advanced AXI interfaces: Due to time constraints and IMMENSE difficulty with Xilinx software, the investigation wasn't able to delve into AXI-Stream/AXI-DMA burst transfer versions of the accelerator, which could potentially further lower the latency from data accesses.

Fixed point: Another consideration was the change from floats to fixed points. This was later discarded after viewing the results. Primarily due to the main bottlenecks now involving high latency from memory access patterns on the SW instead of HW latency. Suggesting that, future works should focus on either integrating the FC into the HW as well or add kernel parameter configurability to be able to use the same convolution engine for multiple convolution layers.

Moving the FC layer into HW: While this WAS also considered after obtaining the first results, time constraints prohibited our investigation to pursue this pathway. A potential pain point with this pathway would come from the loss of architectural flexibility if one were to change their network architecture. Ideally, an implementation could have the core operate with 3 modes available (Conv only, FC only, Conv->FC). With these 3 modes alone, one could design networks that use the accelerator core to significantly accelerate convolution layers, FC layers and layers where a convolution layer feeds into an FC layer. However, configuration of inputs, outputs, layer parameters make this a very ambitious endeavour.

Deeper layer models: While we did see significant performance improvements, the small scale of our model meant the speedup was limited by Amdahl's law. Investigating this solution on models with multiple convolution layers could see more of the HW's potential.

Conclude

In summary, the project investigated the implementation of a task-specific CNN model we dubbed, "ToyCNN" to classify handwritten digits from the MNIST dataset with a small model, to emulate an environment where this network would be deployed on a low-energy device. After applying the knowledge learned in the HLS domain, the resulting implementation made several key improvements to enhance latency and throughput. While we did meet difficulties working with the Xilinx platform (due to using old versions and mismatching versions), as well as running out of time, we made significant progress. Hence, demonstrating the parallel processing power of FPGAs to accelerate convolution operations as well as opening paths for future works to branch from.