MNIST CNN Accelerator on an FPGA

Aymen, Stephen, Connor, and Bradley

Introduction - Project Overview

- Project Goal: develop and test software models to interface with an FPGA-based CNN accelerator for 28x28 MNIST digit classification.
- Key Interactions (HPS and FPGA):
  - Userspace - sends 1-bit quantized MNIST pixels [784 times == 784 bits] + addresses to hardware through driver, with attached final 1 for the image load done bit..
  - Hardware - receives image-data and performs convolution, followed by pooling, densing, and output with an output computed flag (using trained model weights/biases hard-coded in initialized ROM modules).
  - Software - polling for hardware completion, retrieves 10 classification scores from FPGA/hardware registers upon a computed output.
  - Driver - functions as a bidirectional data-flow interpreter between the HPS and FPGA.

CNN Model

- Keras Structure and Independent Training/Inference
- Built a keras model from scratch, trained it to get weights/biases, translated weights and biases even after fixed point hardware translations through scaling factors
- Quantized bits from grayscale bytes to binary bits (if <127 become black, otherwise white

```
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(12, (3,3), activation="relu"),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(num_classes, activation="softmax"),
])
```

Layer	Kernel Shape	in_channels	# Filters / out_channels	Weights + Bias
Conv2D #1	(3x3)	1	12	108+12
Dense	–	2028	10	20,280+10
Total				

*To calculate number of weights = kernel shape * in_channels * filters*

Quantization Computations

Weights for models normally stored in float32

Float16 quantization reduces memory impact by 2x, still requires floating point accumulation and arithmetic

Int8 quantization reduces memory impact by 4x with int32 accumulation

Converting from real values *x* (weights, inputs, biases) requires an adjustment with a scale factor S and a zero point Z

$$x_q = round\left(\frac{x}{S} + Z\right)$$

$$x = S \times (x_q - Z)$$

Scale Factors

Layer	Type	Scale	Zero Point
Conv2D	Input	0.00392	-128
Conv2D	Weight	0.01033	0
Conv2D	Bias	0.000041	0
Dense	Input	0.198817	44
Dense	Weight	0.005921	0
Dense	Bias	0.000068	0
Output	Output	0.003906	-128

$$x_q = round\left(\frac{x}{S} + Z\right)$$

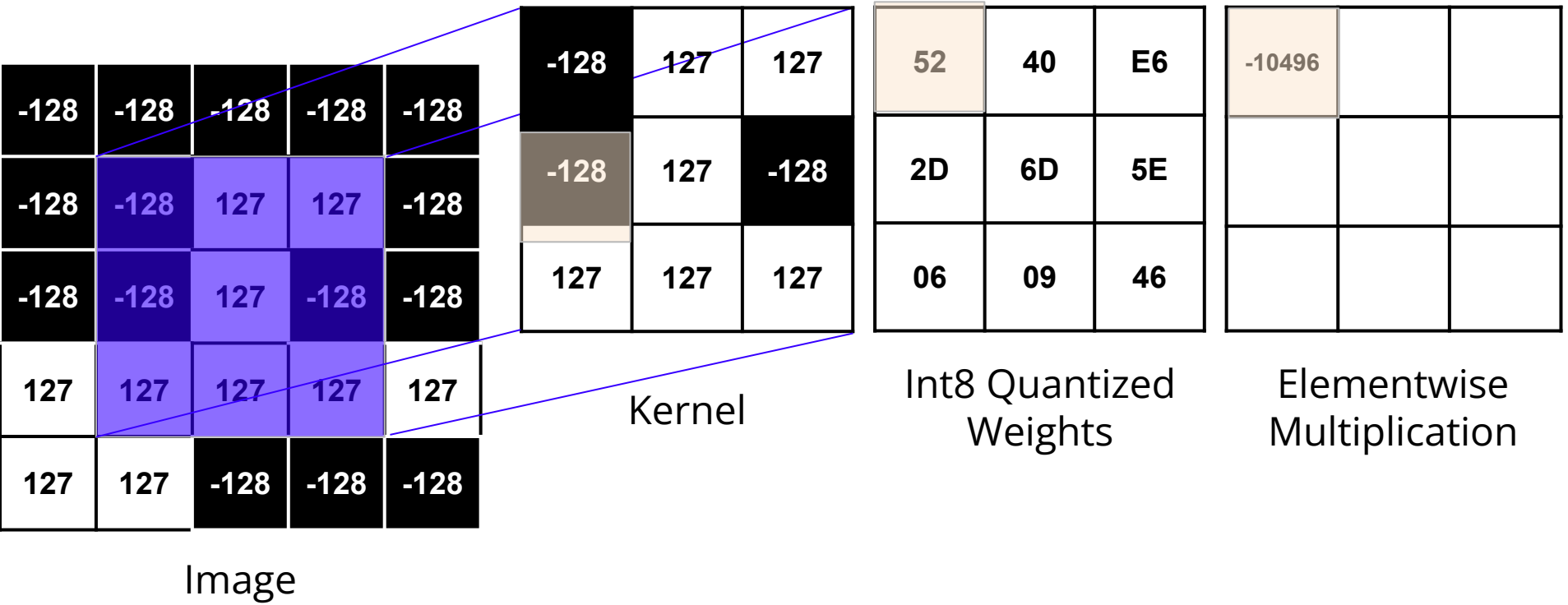
$$x = S \times (x_q - Z)$$

Scale Factors

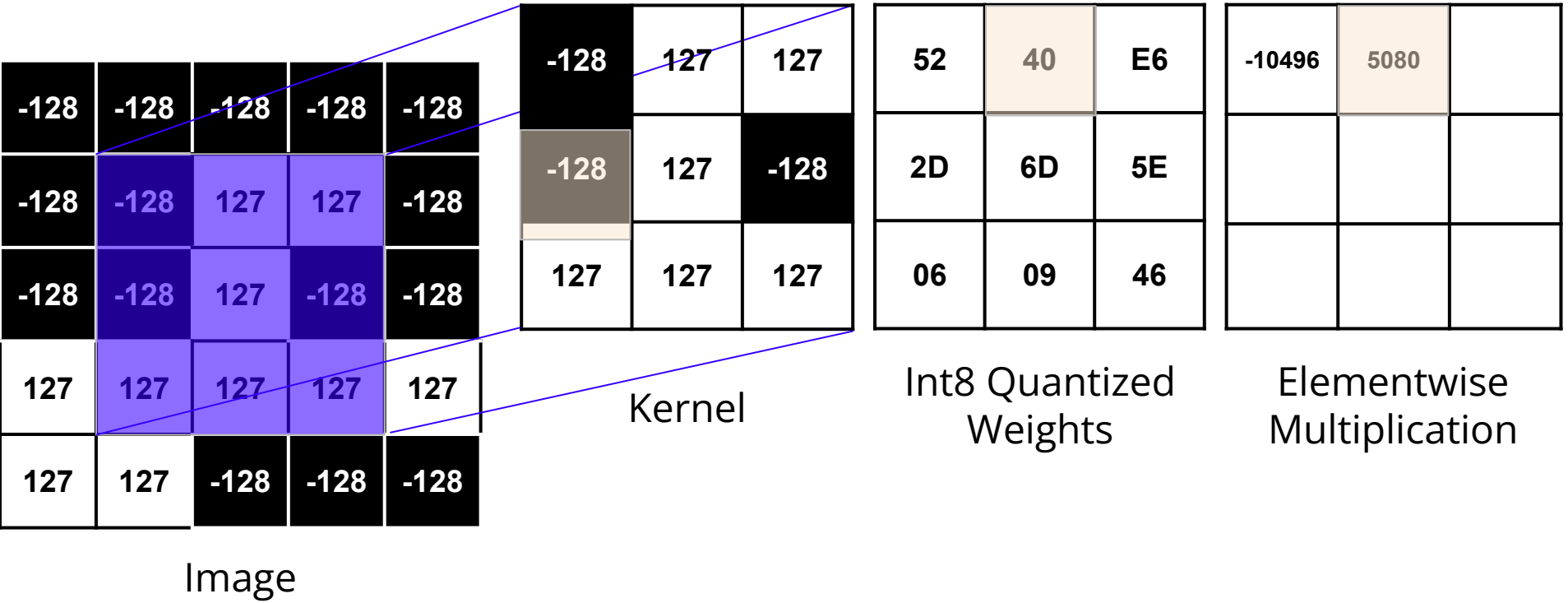
$$M = \frac{w \times x}{y}$$

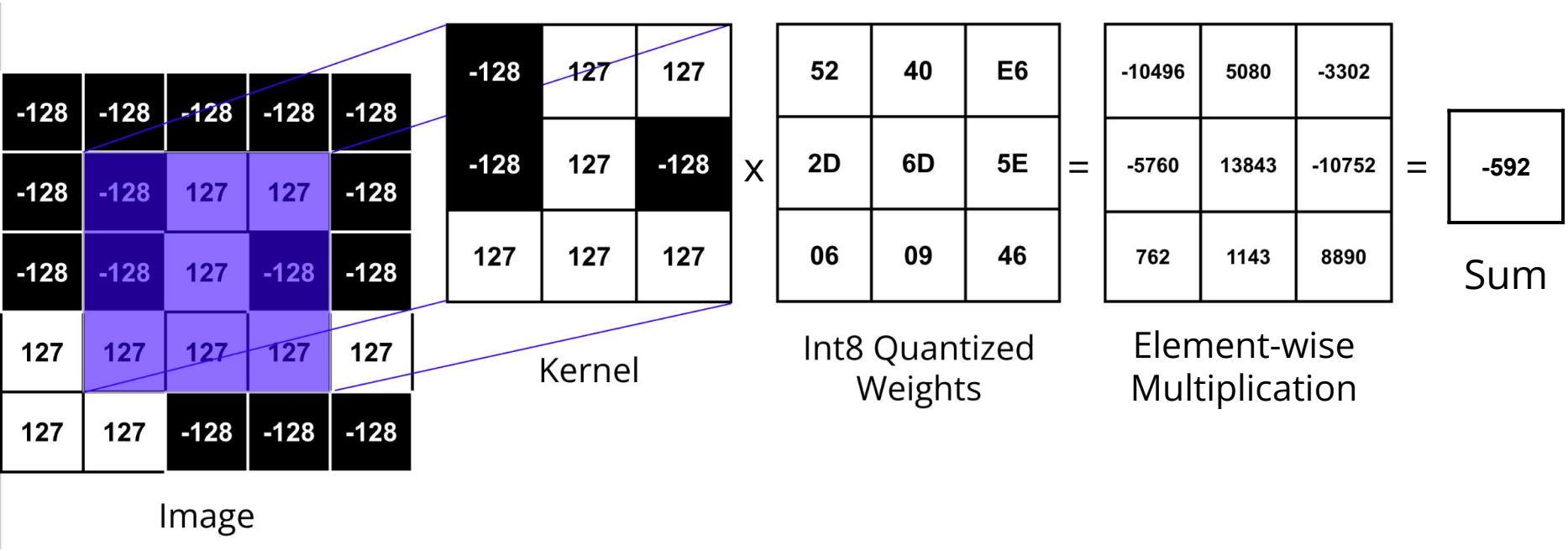
Layer	Type	Scale	Zero Point	M	Bit Shift	Multiply	% Error
Conv2D	Input	0.00392	-128	0.01036	-10	11	3.68%
Conv2D	Weight	0.01033	0				
Conv2D	Bias	0.000041	0				

Convolution Algorithm



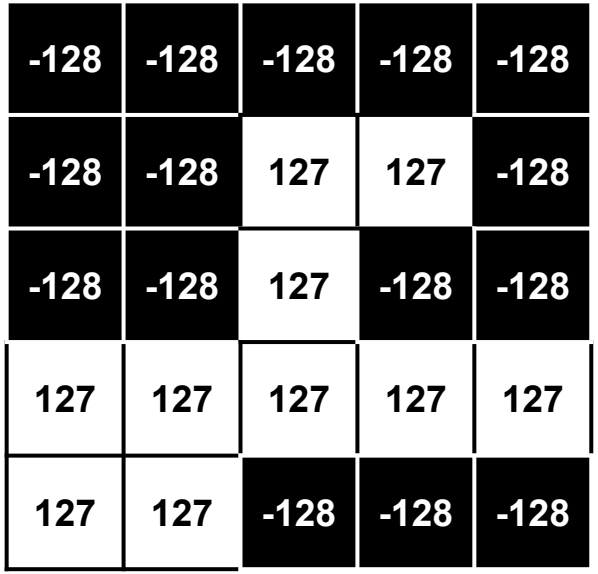
Convolution Algorithm



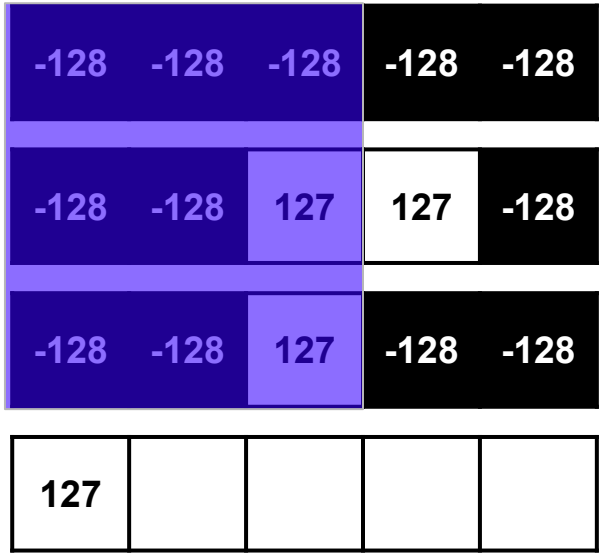

-128 -128 -128 -128 -128
-128 -128 127 127 -128
-128 -128 127 -128 -128
127 127 127 127 127
127 127 -128 -128 -128
Image
-128 127 127
-128 127 -128
127 127 127
Kernel
X
52 40 E6
2D 6D 5E
06 09 46
Int8 Quantized Weights
=
-10496 5080 -3302
-5760 13843 -10752
762 1143 8890
Element-wise Multiplication
=
-592
Sum

Parallelizing Convolutions?

Convolution Algorithm: Sliding Windows



Image



Convolution Verilog

```
part_mult0 <= $signed({{24{top_line[0][7]}}, top_line[0]}) * $signed({{24{weight[0][7]}}, weight[0]});
part_mult1 <= $signed({{24{top_line[1][7]}}, top_line[1]}) * $signed({{24{weight[1][7]}}, weight[1]});
part_mult2 <= $signed({{24{top_line[2][7]}}, top_line[2]}) * $signed({{24{weight[2][7]}}, weight[2]});

part_mult3 <= $signed({{24{mid_line[0][7]}}, mid_line[0]}) * $signed({{24{weight[3][7]}}, weight[3]});
part_mult4 <= $signed({{24{mid_line[1][7]}}, mid_line[1]}) * $signed({{24{weight[4][7]}}, weight[4]});
part_mult5 <= $signed({{24{mid_line[2][7]}}, mid_line[2]}) * $signed({{24{weight[5][7]}}, weight[5]});

part_mult6 <= $signed({{24{bot_line[0][7]}}, bot_line[0]}) * $signed({{24{weight[6][7]}}, weight[6]});
part_mult7 <= $signed({{24{bot_line[1][7]}}, bot_line[1]}) * $signed({{24{weight[7][7]}}, weight[7]});
part_mult8 <= $signed({{24{bot_line[2][7]}}, bot_line[2]}) * $signed({{24{weight[8][7]}}, weight[8]});

partial1_add1 <= part_mult0 + part_mult1;
partial1_add2 <= part_mult2 + part_mult3;
partial1_add3 <= part_mult4 + part_mult5;
partial1_add4 <= part_mult6 + part_mult7;

partial2_add1 <= partial1_add1 + partial1_add2;
partial2_add2 <= partial1_add3 + partial1_add4;

partial3_add1 <= partial2_add1 + partial2_add2;

acc <= partial3_add1 + part_mult8 + $signed({{24{bias[7]}}, bias});
```

```
Inputs: -128, -128, -128, 127, 127, 127, 127, 127, 127
Multiplications: -10496, -8192, 3328, 5715, 13843, -12032, 762, 1143, 8890
  First stage of adder tree: -18688, 9043, -25984, 1905
Second stage of adder tree: -21120, -26374
Third Stage of adder tree: -49024
  Final acc value: -128
```

Dense Verilog

```
always_ff @(posedge clock) begin
    if (reset) begin
        mult <= 0;
        sum <= 0;

    end
    else if (count == 2029) begin
        sum <= mult;
        mult <= ($signed({{8{data_in[7]}}, data_in})) * $signed({{8{weight[7]}}, weight});
    end
    else if (enable) begin
        mult <= ($signed({{8{data_in[7]}}, data_in})) * $signed({{8{weight[7]}}, weight});
        sum <=  sum + mult;
    end
    else begin
        mult <= 0;
    end
end
```

Dense Computation

Pulling 12 Convolutional Filters into 10 Possible Values

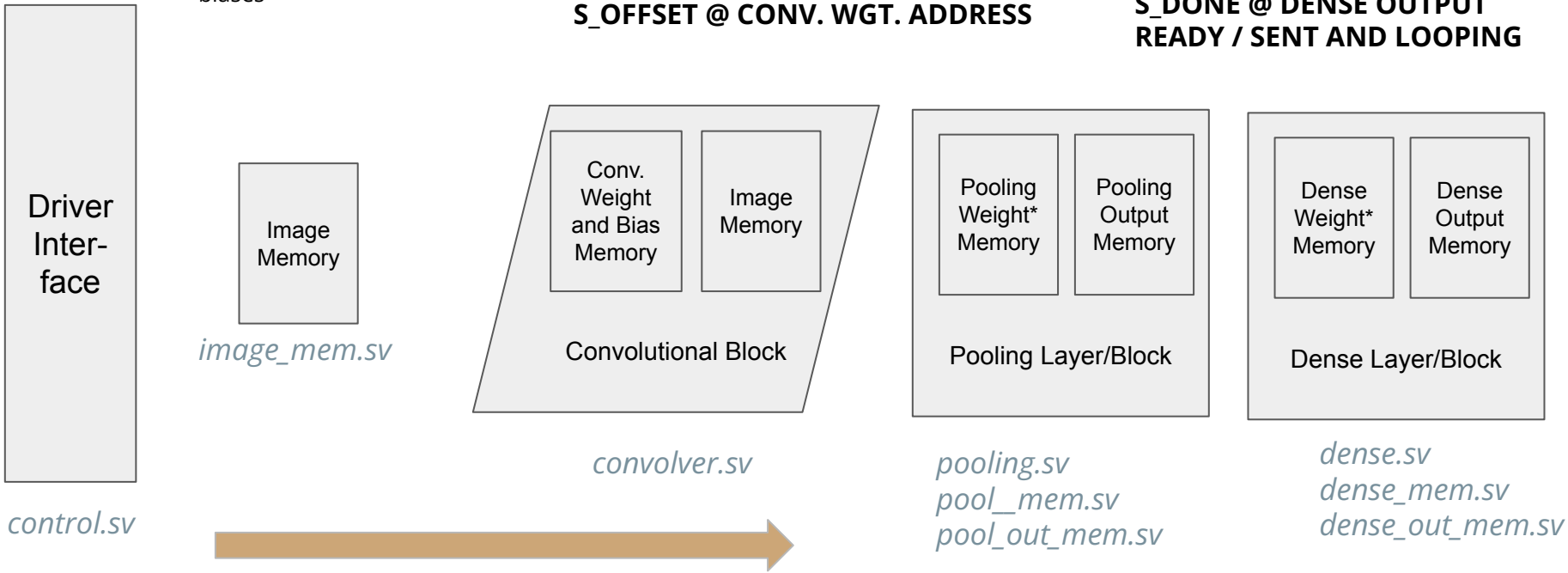
Hardware Overview (control.sv)

FSM - [S_LOAD; S_COMPUTE; S_OFFSET; S_PAUSE; S_DENSE; S_DONE]

* = includes weights and biases

S_LOAD @ IDLE/MAGE LOAD
S_COMPUTE @ CONV. COMPUTATION
S_OFFSET @ CONV. WGT. ADDRESS

S_PAUSE @ EXTRA CYCLE
S_DENSE @ DENSE LAYER
S_DONE @ DENSE OUTPUT
READY / SENT AND LOOPING



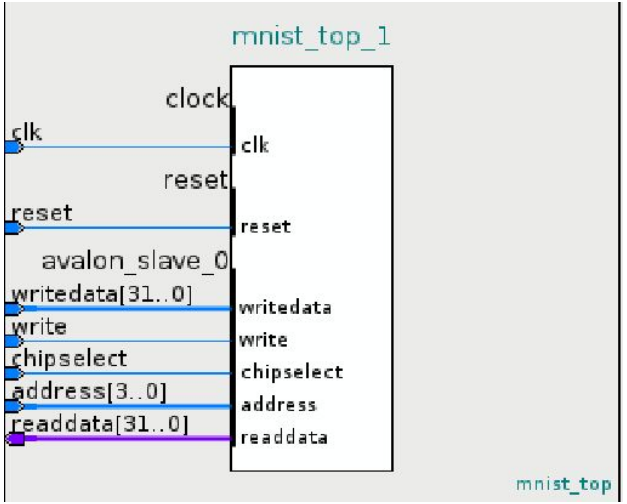
Revolving Line-Buffer Microarchitecture

Convolutional Block *convolver.sv*	MEM array 1	CALCULATIONS
	MEM array 2	
	MEM array 3	
	MEM array 4	LOADING ROW 4
	NEXT STATE	
	MEM array 2	CALCULATIONS
	MEM array 3	
	MEM array 4	
	MEM array 1	LOADING ROW 5

```
module Line_Buff#(
    parameter WORD = 8,
    parameter HOR = 28,

    parameter ADDR_W = $clog2(HOR)
)(
    input logic clock,
    input logic reset,
    input logic data_valid_in,
    input logic signed [WORD-1:0] pixel_data_in,
    input logic [ADDR_W-1:0] hcount_in,

    output logic signed [WORD-1:0] line_buff_out [0:2],
    output logic [ADDR_W-1:0] hcount_out,
    output logic signed [WORD-1:0] mem0_out [0:HOR-1],
    output logic signed [WORD-1:0] mem1_out [0:HOR-1],
    output logic signed [WORD-1:0] mem2_out [0:HOR-1],
    output logic signed [WORD-1:0] mem3_out [0:HOR-1]


);
```

Hardware Interface - Top Level Module

Connections	Name	Description
	clk_0	Clock Source
	clk_in	Clock Input
	clk_in_reset	Reset Input
	clk	Clock Output
	clk_reset	Reset Output
	hps_0	Arria V/Cyclone V Hard Proce...
	h2f_user1_clock	Clock Output
	memory	Conduit
	hps_io	Conduit
	h2f_reset	Reset Output
	h2f_axi_clock	Clock Input
	h2f_axi_master	AXI Master
	f2h_axi_clock	Clock Input
	f2h_axi_slave	AXI Slave
	h2f_lw_axi_clock	Clock Input
	h2f_lw_axi_master	AXI Master
	mnist_top_0	MNIST Top
	clock	Clock Input
	reset	Reset Input
	avalon_slave_0	Avalon Memory Mapped Slave



Example Module Interfaces - dense.sv and pooling.sv

Note: Memory modules are standard 1-port/2-port ROMs.

```
module dense(
    input logic clock,
    input logic reset,
    input logic signed [7:0] data_in,
    input logic signed [7:0] weight,
    input logic enable,
    input logic signed [7:0] dense_bias,

    output logic signed [31:0] output_data,
    output logic dense_valid,
    output logic [10:0] count_dbg,
    output logic [3:0] times_out
);
```

```
module pooling#(
    parameter WORD = 32,
    parameter HOR = 26,

    parameter ADDR_W = $clog2(HOR),
    parameter Q = 3     // number of fractional bits
) (
    input logic clock,
    input logic reset,
    input logic data_valid_in,
    input logic signed [WORD-1:0] data_in,
    input logic [ADDR_W-1:0] hcount_in,

    output logic signed [7:0] pooled_out,
    output logic signed [WORD-1:0] mem0_out [0:HOR-1],
    output logic signed [WORD-1:0] mem1_out [0:HOR-1],
    output logic signed [WORD-1:0] mem2_out [0:HOR-1],
    output logic signed [WORD-1:0] mem3_out [0:HOR-1],
    output logic signed [WORD-1:0] top_line_out [0:1],
    output logic signed [WORD-1:0] mid_line_out [0:1]
);
```

HW/SW Interfacing

Register Map

Register # (Offset / Addr)	Function	Code Definitions
0	Constructed Image Word	`#define IMAGE(x) (x)`
1	Dense Classification 0	`#define PROB_0(x) ((x)+4)`
2	Dense Classification 1	`#define PROB_1(x) ((x)+8)`
3	Dense Classification 2	`#define PROB_2(x) ((x)+12)`
4	Dense Classification 3	`#define PROB_3(x) ((x)+16)`
5	Dense Classification 4	`#define PROB_4(x) ((x)+20)`
…….	*Dense Classifications 5-7*	
10	Dense Classification 8	`#define PROB_8(x) ((x)+36)`
11	Dense Classification 9	`#define PROB_9(x) ((x)+40)`
……	[EMPTY]	
15	Dense Done Bit	`DENSE_DONE_F(x) ((x)+60)`

user.h defines:
ACCEL_IOC_MAGIC ‘k’ and accelerator_arg_t (union struct for all ioctl data)

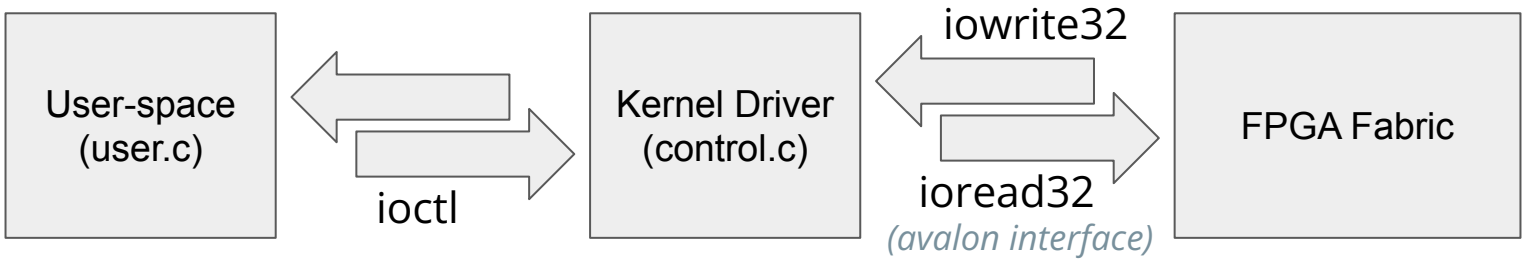
Offset == Avalon Slave Addr == Register #

System Architecture Overview (Final Implementation)

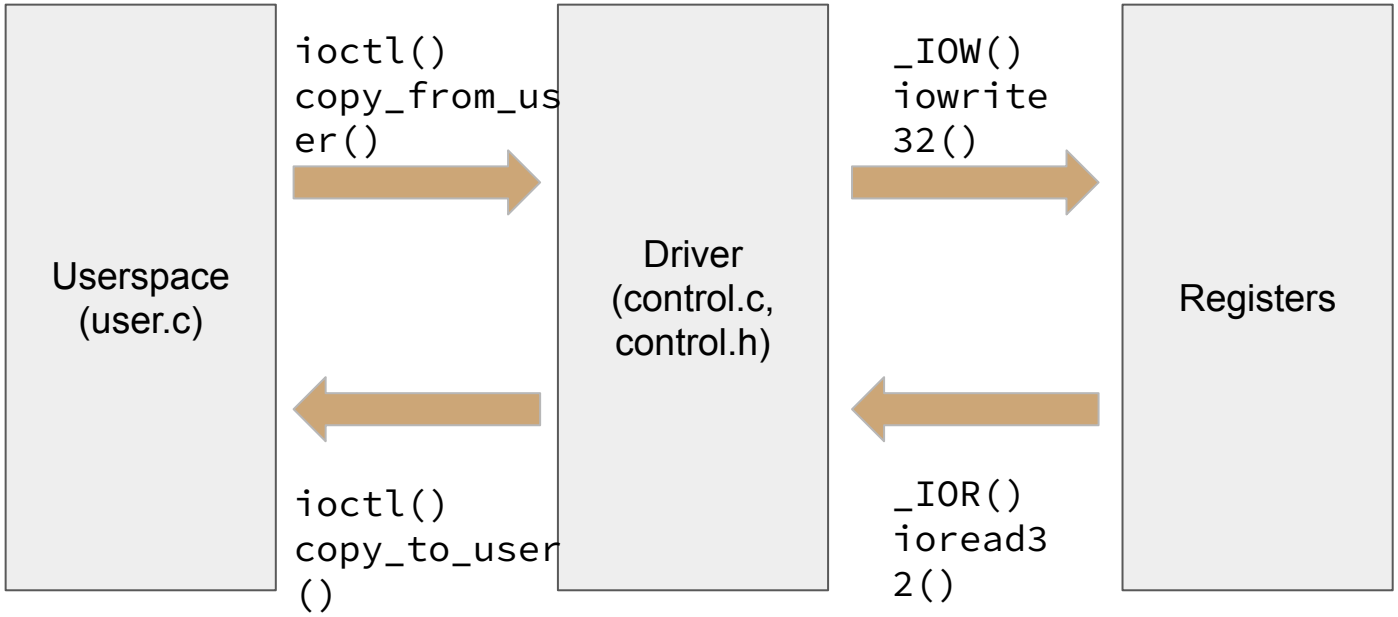
Userspace App (user.c) - manages MNIST image processing (file read), packs pixel + address + done into 32-bit words, and communicates to hardware via ioctls.

Kernel Driver (control.c via control.h) - characterizes device/device tree (dev/control) and translates ioctls from user.c into Avalon R/W on FPGA registers using copy_from/to_user.

FPGA Fabric - defines register map for data input, status, and results from memory-mapped registers accessible to the HPS.



Userspace <-> Kernel <-> FPGA Fabric



Software Interface - Userspace ⟶ Driver (Through Header Handshake)

1

```c
int load_image_data(uint32_t *image_data_word) {
  control_arg_t vla;
  vla.image_data = *image_data_word;
  if (ioctl(control_fd, CONTROL_WRITE_IMAGE, &vla)) {
    perror("ioctl(CONTROL_WRITE_IMAGE) failed");
    return -1;
  }
  return 0;
}
```

image load process

2

```c
/* ioctls and their arguments */
#define CONTROL_WRITE_IMAGE          _IOW(CONTROL_MAGIC, 1, control_arg_t)
#define CONTROL_READ_DENSE_DONE      _IOR(CONTROL_MAGIC, 2, control_arg_t)
```

3

```c
switch (cmd) {
case CONTROL_WRITE_IMAGE:
    if (copy_from_user(&vla, (control_arg_t *) arg,
            sizeof(control_arg_t)))
        return -EACCES;
    write_image_data(&vla.image_data);
    break;
```

4

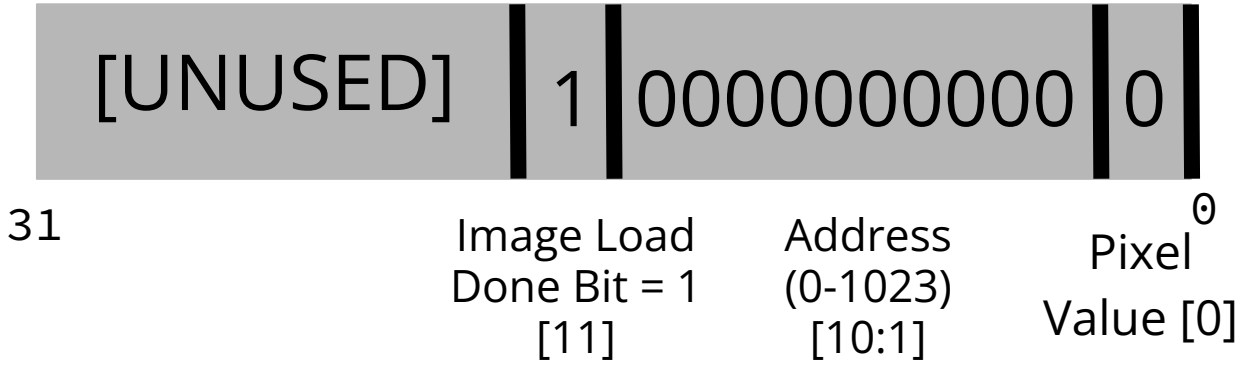
```c
static void write_image_data(uint32_t *image_data)
{
    iowrite32(*image_data, IMAGE(dev.virtbase));
    dev.image_data = *image_data;
}
```

Communication 1: Image 32-Bit Word (HW <- SW)

IOW ⇒ 4-BIT ADDRESS + 32-BIT DATA
IOR ⇒ 4-BIT ADDRESS

```
#define CONTROL_WRITE_IMAGE
 _IOW(CONTROL_MAGIC, 1, control_arg_t)
```

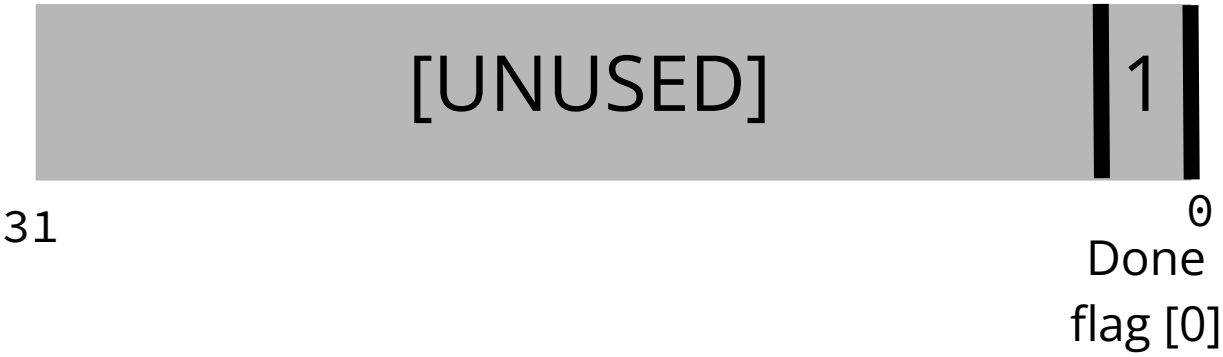
IOW Address: offset 0



Comm. 2: Dense_Done Signal (HW -› SW)

```
#define CONTROL_READ_DENSE_DONE        _IOR(CONTROL_MAGIC, 2,
control_arg_t)
```

IOR Address: offset 15



Comm. 3: Output Trigger (HW -› SW)*

* Technically reading data from HW -> SW, but the purpose of the call is to send reg addr from SW -> HW (readdata is not stored)

```
#define CONTROL_TRIGGER_DENSE_ADDR     _IOR(CONTROL_MAGIC, 3,
control_arg_t)
```

IOR Address: offset 1-10

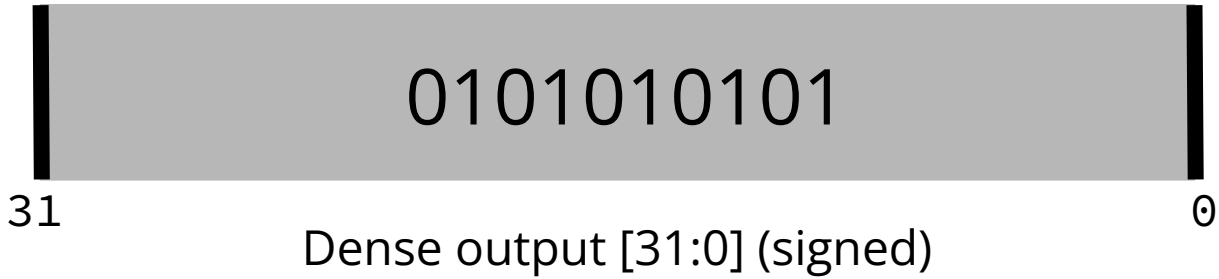
[Register contents (UNUSED)]

31 0

Comm. 4: Read Dense Outputs (HW -› SW)

```
#define CONTROL_READ_DENSE_DATA        _IOR(CONTROL_MAGIC, 4,
control_arg_t)
```

IOR Address: 1-10 offset 1-10



Dense output [31:0] (signed)

Final Results - Testing and Functional Verification

Test: Sent a 28x28 (784 bits) MNIST image of a 7.

Verified:

- Image data words are read from image file, constructed into 32-bit word, and written to register 0.
- Kernel/driver software and header interface translate this word for hardware via ioctl ⇐⇒ ioread32/iowrite32
- Dense_done signal polled and detected.
- Values read back from dense output registers.
- Validity of hardware mathematical computations, via paper-calculations.

```
root@de1-soc:~/mnist-accelerator/sw# ./user
Control Userspace program started
Dense output 0: -67
Dense output 1: -77
Dense output 2: -102
Dense output 3: -79
Dense output 4: -143
Dense output 5: 24
Dense output 6: -143
Dense output 7: -114
Dense output 8: -205
Dense output 9: 0
Control Userspace program terminating
```

Final Conclusions and Considerations

Successful Demonstrations:

- Bit-packed and processed image data loading.
- Successful translation of data across hardware and software through kernel/header software.
- Status polling for hardware completion flags and FSM for state.
- Retrieval of final 32-bit classification probabilities for each possible digit.
- Total Operational Time: Ranges from 1.01-1.07 milliseconds @ 28% resource utilization

Future Considerations:

- **Control as many software-(in)dependent variables as possible** - having access to CNN model's specific scaling factors and values for fine tuning due to hardware scaling factor as bits are sent
- **Expand/improve system architecture as a whole** - take advantage of efficient and revolving line buffer microarchitecture in convolutional and pooling blocks.

Thank you!