AAHLS Final Report - Handwritten number Generation

B07901181 王少群 R11921061 郭霖璟 R10943009 張凱茗

1. Problem statement

In this final project, we make a handwritten number generation accelerator with MNIST dataset, which have 60000 images with 0 to 9 handwritten number. Next, we train a Generative Adversarial Network (GAN) to generate the handwritten number image. In this accelerator, we only use the generator network to accelerate the image generation. The generator network will get random noise vector input, and output the generated image. Our generator model architecture is 3 layer fully-connected model, with 100 dimensional noise vector input, and 28*28 image output.

The target platform of our accelerator is PYNQ-Z2, a resource constraint FPGA device on edge. Without using the Xilinx framework – FINN, a Pytorch framework with Binary Neural Network, and compile to the HLS, we write the HLS to build our model accelerator by ourselves. Finally, we want to achieve the higher throughput than CPU and GPU. We run the Pytorch model and get 2951.24 FPS on CPU and 1333.21 FPS on GPU.

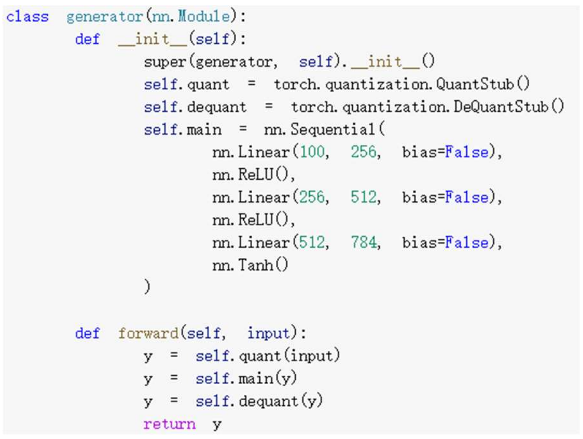
2. System overview

The system operation flow is

1. Generate random input from PS side.
2. Transfer the input from PS side to PL side.
3. Calculate each layer of fully-connected model in PL side.
4. Receive the output from PL side in PS side
5. Show the result in PS side.

3. INT8 Fully-connected model

First, we use the Pytorch framework to train the generator model. In Figure 1, it has three fully-connected layer, which have 256, 512, 784 nodes in each layer. After each layer, it follows an activation layer ReLU or Tanh.



```
class generator(nn.Module):
    def __init__(self):
        super(generator, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.dequant = torch.quantization.DeQuantStub()
        self.main = nn.Sequential(
            nn.Linear(100, 256, bias=False),
            nn.ReLU(),
            nn.Linear(256, 512, bias=False),
            nn.ReLU(),
            nn.Linear(512, 784, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        y = self.quant(input)
        y = self.main(y)
        y = self.dequant(y)
        return y
```

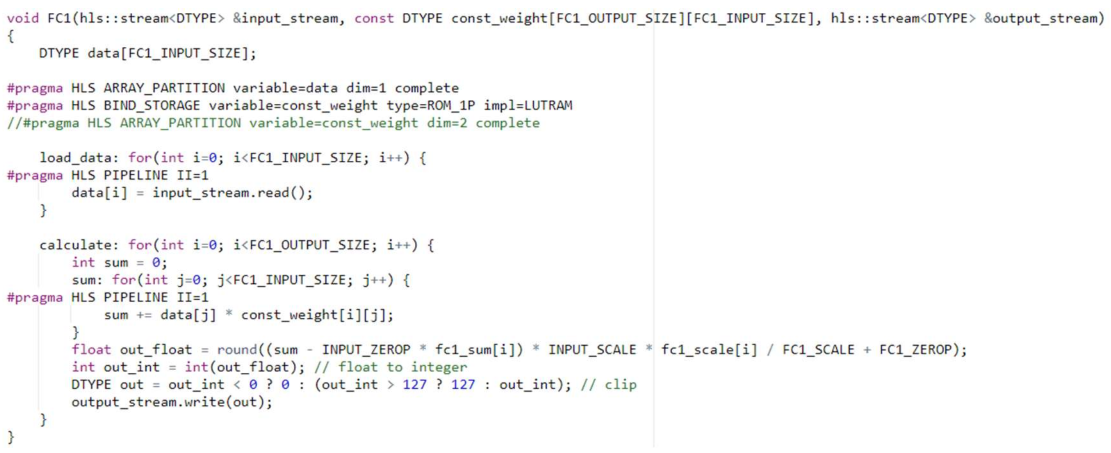
Figure 1、Generator model architecture in Pytorch.

Next, we need to quantize our model because we can't store the full Float32 weight on FPGA, the resource on FPGA is limited. Therefore, we quantized the weight into INT8 to save the resource usage. The weight become a quarter with the Float32 to INT8 quantization, and the formula to transfer real value into quantized value is in below

$$real\_value = scale(quant\_value - zero\_point)$$

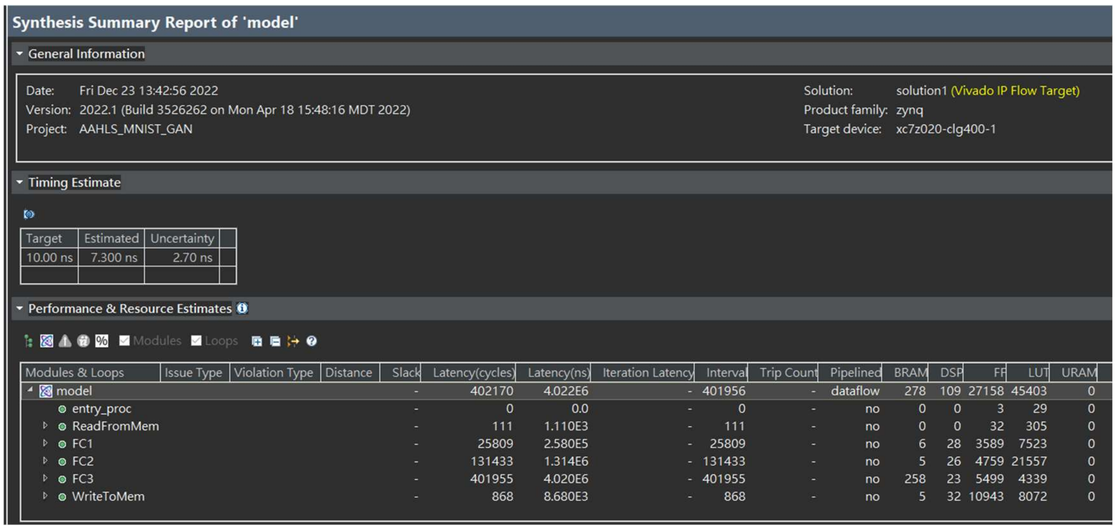
Next, we accumulate the activation in 32-bit integer and quantize the activation into 8-bit integer by the input and output scales and zero points.

In the baseline architecture, we try to make the accelerator outputs the correct value same with the golden file, and the first fully-connected layer HLS code is in Figure 2. After writing the three layers of the model, we synthesis with the clock in 10 ns, and the synthesis summary report is in Figure 3.



```
void FC1(hls::stream<DTYPE> &input_stream, const DTYPE const_weight[FC1_OUTPUT_SIZE][FC1_INPUT_SIZE], hls::stream<DTYPE> &output_stream)
{
    DTYPE data[FC1_INPUT_SIZE];

#pragma HLS ARRAY_PARTITION variable=data dim=1 complete
#pragma HLS BIND_STORAGE variable=const_weight type=ROM_1P impl=LUTRAM
//#pragma HLS ARRAY_PARTITION variable=const_weight dim=2 complete

    load_data: for(int i=0; i<FC1_INPUT_SIZE; i++) {
#pragma HLS PIPELINE II=1
        data[i] = input_stream.read();
    }

    calculate: for(int i=0; i<FC1_OUTPUT_SIZE; i++) {
        int sum = 0;
        sum: for(int j=0; j<FC1_INPUT_SIZE; j++) {
#pragma HLS PIPELINE II=1
            sum += data[j] * const_weight[i][j];
        }
        float out_float = round((sum - INPUT_ZEROP * fc1_sum[i]) * INPUT_SCALE * fc1_scale[i] / FC1_SCALE + FC1_ZEROP);
        int out_int = int(out_float); // float to integer
        DTYPE out = out_int < 0 ? 0 : (out_int > 127 ? 127 : out_int); // clip
        output_stream.write(out);
    }
}
```

Figure 2、FC1 HLS code.



Synthesis Summary Report of 'model'

General Information

Date:	Fri Dec 23 13:42:56 2022	Solution:	solution1 (Vivado IP Flow Target)
Version:	2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)	Product family:	zynq
Project:	AAHLS_MNIST_GAN	Target device:	xc7z020-clg400-1

Timing Estimate

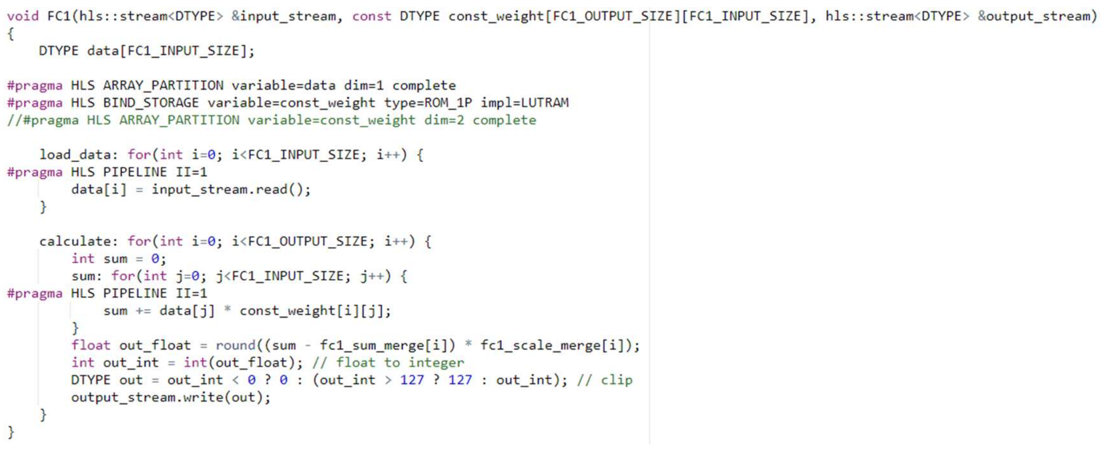
Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
model				-	402170	4.022E6	-	401956	-	dataflow	278	109	27158	45403	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	29	0
ReadFromMem				-	111	1.110E3	-	111	-	no	0	0	32	305	0
FC1				-	25809	2.580E5	-	25809	-	no	6	28	3589	7523	0
FC2				-	131433	1.314E6	-	131433	-	no	5	26	4759	21557	0
FC3				-	401955	4.020E6	-	401955	-	no	258	23	5499	4339	0
WriteToMem				-	868	8.680E3	-	868	-	no	5	32	10943	8072	0

Figure 3、Baseline architecture synthesis summary report.

According to the high usage of DSP and LUT, we merge the scaling factor and zero points into a single floating point value, in order to reduce the number of floating points multiplication and division, the first fully-connected layer HLS code is in Figure 4.



```
void FC1(hls::stream<DTYPE> &input_stream, const DTYPE const_weight[FC1_OUTPUT_SIZE][FC1_INPUT_SIZE], hls::stream<DTYPE> &output_stream)
{
    DTYPE data[FC1_INPUT_SIZE];

#pragma HLS ARRAY_PARTITION variable=data dim=1 complete
#pragma HLS BIND_STORAGE variable=const_weight type=ROM_1P impl=LUTRAM
//#pragma HLS ARRAY_PARTITION variable=const_weight dim=2 complete

    load_data: for(int i=0; i<FC1_INPUT_SIZE; i++) {
#pragma HLS PIPELINE II=1
        data[i] = input_stream.read();
    }

    calculate: for(int i=0; i<FC1_OUTPUT_SIZE; i++) {
        int sum = 0;
        sum: for(int j=0; j<FC1_INPUT_SIZE; j++) {
#pragma HLS PIPELINE II=1
            sum += data[j] * const_weight[i][j];
        }
        float out_float = round((sum - fc1_sum_merge[i]) * fc1_scale_merge[i]);
        int out_int = int(out_float); // float to integer
        DTYPE out = out_int < 0 ? 0 : (out_int > 127 ? 127 : out_int); // clip
        output_stream.write(out);
    }
}
```

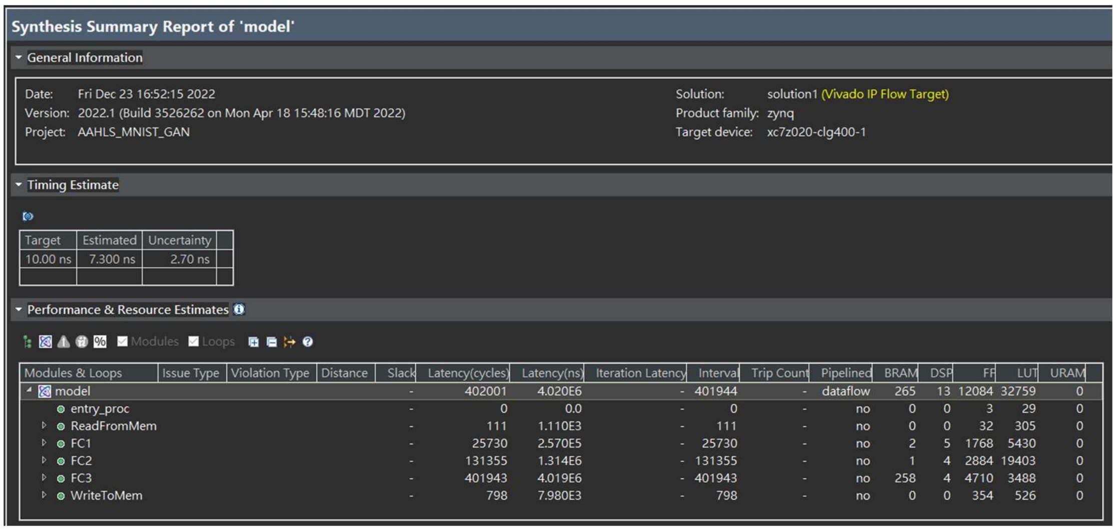
Figure 4、FC1 HLS code.

The other optimization is changing the Tanh activation in the last fully-connected layer into HardTanh. This optimization will reduce the resource usage with a little error from the original output. The Tanh and HardTanh equation is below.

$$Tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$HardTanh(x) = \begin{cases} 1, & if\ x > 1 \\ -1, & if\ x < -1 \\ x, & otherwise \end{cases}$$

Finally, the synthesis summary report is in Figure 5. We can see the DSP and LUT usage reduced, but the latency didn't reduce because most of the latency in the layer is accumulation, so we need to do other optimization.



Synthesis Summary Report of 'model'

General Information

Date:	Fri Dec 23 16:52:15 2022	Solution:	solution1 (Vivado IP Flow Target)
Version:	2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)	Product family:	zynq
Project:	AAHLS_MNIST_GAN	Target device:	xc7z020-clg400-1

Timing Estimate

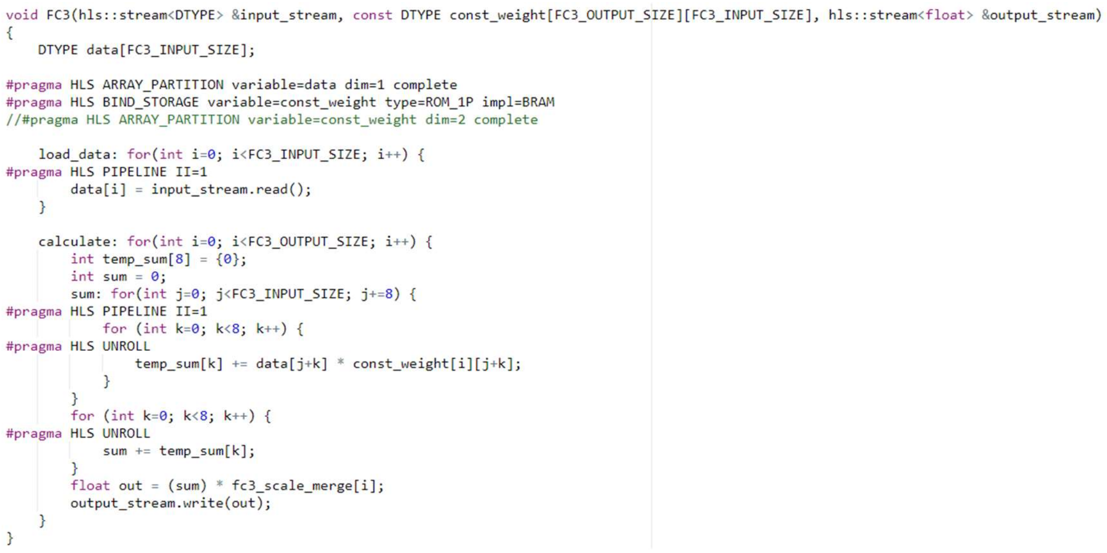
Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
model				-	402001	4.020E6	-	401944	-	dataflow	265	13	12084	32759	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	29	0
ReadFromMem				-	111	1.110E3	-	111	-	no	0	0	32	305	0
FC1				-	25730	2.570E5	-	25730	-	no	2	5	1768	5430	0
FC2				-	131355	1.314E6	-	131355	-	no	1	4	2884	19403	0
FC3				-	401943	4.019E6	-	401943	-	no	258	4	4710	3488	0
WriteToMem				-	798	7.980E3	-	798	-	no	0	0	354	526	0

Figure 5、Kernel optimization 1 architecture synthesis summary report.

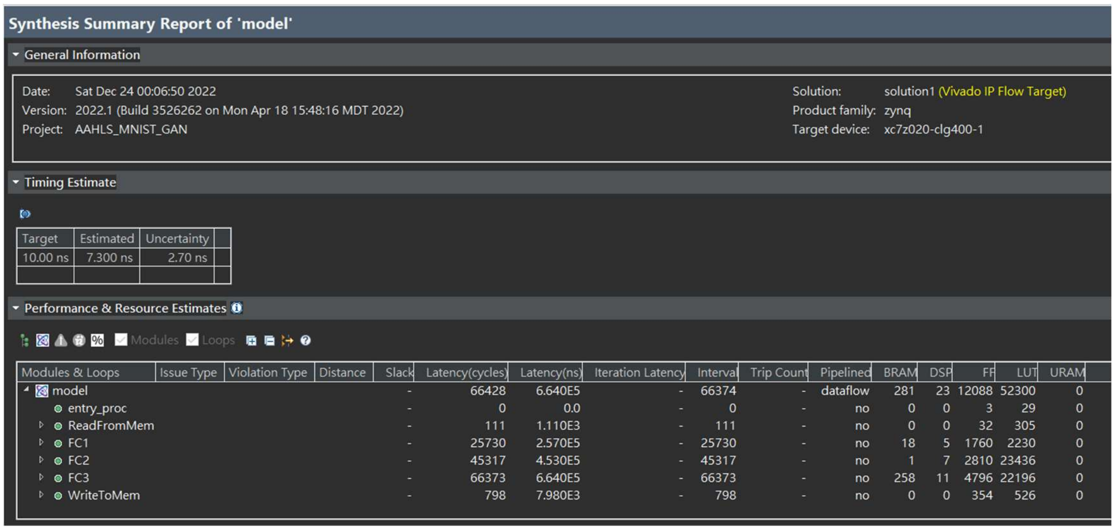
To reduce the kernel latency, we need more PE to calculate the multiplication between inputs and weights at the same time. Therefore, we unroll the inner for loop to calculate the value and accumulation for multiple PEs. As the result in Figure 6, the HLS code of last fully-connected layer have 8 PE for calculation.



```
void FC3(hls::stream<DTYPE> &input_stream, const DTYPE const_weight[FC3_OUTPUT_SIZE][FC3_INPUT_SIZE], hls::stream<float> &output_stream)
{
    DTYPE data[FC3_INPUT_SIZE];

#pragma HLS ARRAY_PARTITION variable=data dim=1 complete
#pragma HLS BIND_STORAGE variable=const_weight type=ROM_1P impl=BRAM
//#pragma HLS ARRAY_PARTITION variable=const_weight dim=2 complete

    load_data: for(int i=0; i<FC3_INPUT_SIZE; i++) {
#pragma HLS PIPELINE II=1
        data[i] = input_stream.read();
    }

    calculate: for(int i=0; i<FC3_OUTPUT_SIZE; i++) {
        int temp_sum[8] = {0};
        int sum = 0;
        sum: for(int j=0; j<FC3_INPUT_SIZE; j+=8) {
#pragma HLS PIPELINE II=1
            for (int k=0; k<8; k++) {
#pragma HLS UNROLL
                temp_sum[k] += data[j+k] * const_weight[i][j+k];
            }
        }
        for (int k=0; k<8; k++) {
#pragma HLS UNROLL
            sum += temp_sum[k];
        }
        float out = (sum) * fc3_scale_merge[i];
        output_stream.write(out);
    }
}
```

Figure 6、FC3 HLS code.

The synthesis summary report is in Figure 7, we can see latency of second and third layer reduce very much, and the bottleneck layer is third fully-connected layer. Each input vector needs almost 66000 cycles to output an image, like the third fully-connected layer.



Synthesis Summary Report of 'model'

General Information

Date: Sat Dec 24 00:06:50 2022
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)
Project: AAHLS_MNIST_GAN

Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

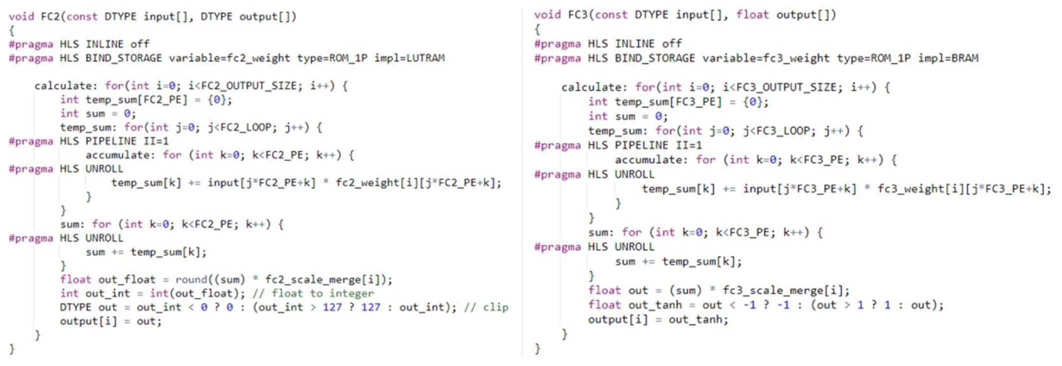
Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
model				-	66428	6.640E5	-	66374	-	dataflow	281	23	12088	52300	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	29	0
ReadFromMem				-	111	1.110E3	-	111	-	no	0	0	32	305	0
FC1				-	25730	2.570E5	-	25730	-	no	18	5	1760	2230	0
FC2				-	45317	4.530E5	-	45317	-	no	1	7	2810	23436	0
FC3				-	66373	6.640E5	-	66373	-	no	258	11	4796	22196	0
WriteToMem				-	798	7.980E3	-	798	-	no	0	0	354	526	0

Figure 7、Kernel optimization 2 architecture synthesis summary report.

Finally, we modified the streaming interface between each kernel into the ping-pong buffer. The HLS code of second and third fully-connected layer is in Figure 8.

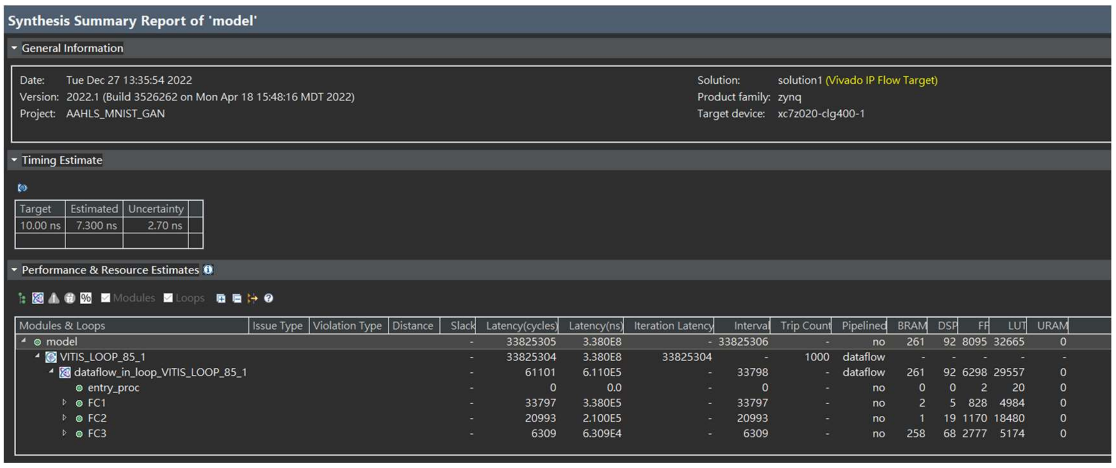


```
void FC2(const DTYPE input[], DTYPE output[])
{
#pragma HLS INLINE off
#pragma HLS BIND_STORAGE variable=fc2_weight type=ROM_1P impl=LUTRAM

    calculate: for(int i=0; i<FC2_OUTPUT_SIZE; i++) {
        int temp_sum[FC2_PE] = {0};
        int sum = 0;
        temp_sum: for(int j=0; j<FC2_LOOP; j++) {
#pragma HLS PIPELINE II=1
            accumulate: for (int k=0; k<FC2_PE; k++) {
#pragma HLS UNROLL
                temp_sum[k] += input[j*FC2_PE+k] * fc2_weight[i][j*FC2_PE+k];
            }
        }
        sum: for (int k=0; k<FC2_PE; k++) {
#pragma HLS UNROLL
            sum += temp_sum[k];
        }
        float out_float = round((sum) * fc2_scale_merge[i]);
        int out_int = int(out_float); // float to integer
        DTYPE out = out_int < 0 ? 0 : (out_int > 127 ? 127 : out_int); // clip
        output[i] = out;
    }
}
```

```
void FC3(const DTYPE input[], float output[])
{
#pragma HLS INLINE off
#pragma HLS BIND_STORAGE variable=fc3_weight type=ROM_1P impl=BRAM

    calculate: for(int i=0; i<FC3_OUTPUT_SIZE; i++) {
        int temp_sum[FC3_PE] = {0};
        int sum = 0;
        temp_sum: for(int j=0; j<FC3_LOOP; j++) {
#pragma HLS PIPELINE II=1
            accumulate: for (int k=0; k<FC3_PE; k++) {
#pragma HLS UNROLL
                temp_sum[k] += input[j*FC3_PE+k] * fc3_weight[i][j*FC3_PE+k];
            }
        }
        sum: for (int k=0; k<FC3_PE; k++) {
#pragma HLS UNROLL
            sum += temp_sum[k];
        }
        float out = (sum) * fc3_scale_merge[i];
        float out_tanh = out < -1 ? -1 : (out > 1 ? 1 : out);
        output[i] = out_tanh;
    }
}
```

Figure 8、FC2 and FC3 HLS code.

The synthesis summary report is in Figure 9, and the Co-simulation timeline report is in Figure 10. We can see when the first fully-connected layer finish first task, it can start the second task. Therefore, it can increase the throughput of the generator model.



Synthesis Summary Report of 'model'

General Information

Date:	Tue Dec 27 13:35:54 2022	Solution:	solution1 (Vivado IP Flow Target)
Version:	2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)	Product family:	zynq
Project:	AAHLS_MNIST_GAN	Target device:	xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
model				-	33825305	3.380E8	-	33825306	-	no	261	92	8095	32665	0
VITIS_LOOP_85_1				-	33825304	3.380E8	33825304	-	1000	dataflow	-	-	-	-	-
dataflow_in_loop_VITIS_LOOP_85_1				-	61101	6.110E5	-	33798	-	dataflow	261	92	6298	29557	0
entry_proc				-	0	0.0	-	0	-	no	0	0	2	20	0
FC1				-	33797	3.380E5	-	33797	-	no	2	5	828	4984	0
FC2				-	20993	2.100E5	-	20993	-	no	1	19	1170	18480	0
FC3				-	6309	6.309E4	-	6309	-	no	258	68	2777	5174	0

Figure 9、Kernel optimization 3 architecture synthesis summary report.

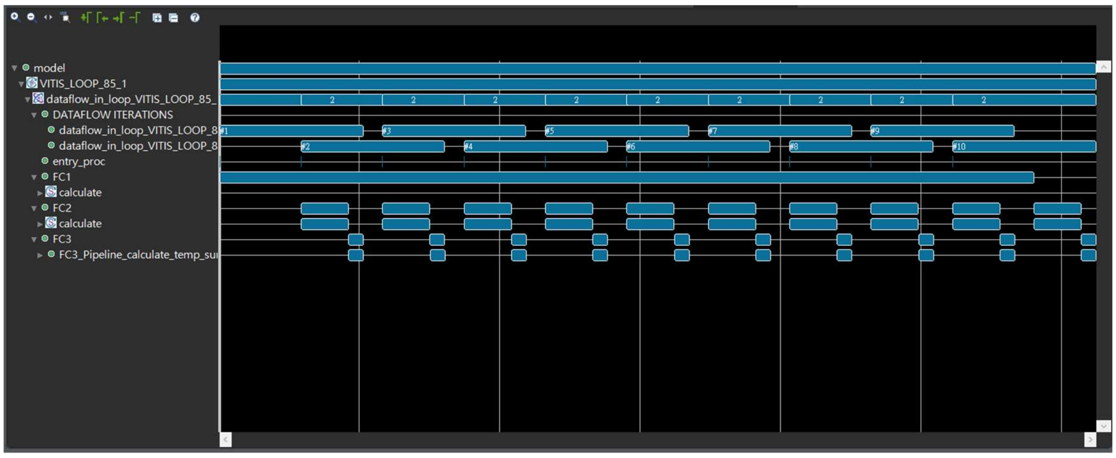

model
VITIS_LOOP_85_1
dataflow_in_loop_VITIS_LOOP_85_
DATAFLOW ITERATIONS
dataflow_in_loop_VITIS_LOOP_8
dataflow_in_loop_VITIS_LOOP_8
entry_proc
FC1
calculate
FC2
calculate
FC3
FC3_Pipeline_calculate_temp_su

Figure 10、Kernel optimization 3 architecture co-simulation report.

4. PYNQ-Z2 result

Finally, we put the RTL design generated from Vitis HLS into Vivado, and the block design is in Figure 11. The utilization in post-implementation is in Figure 12. The LUT usage is 58%, FF usage is 10%, BRAM usage is 94%, DSP usage is 42%. The FPGA device usage is in Figure 13.

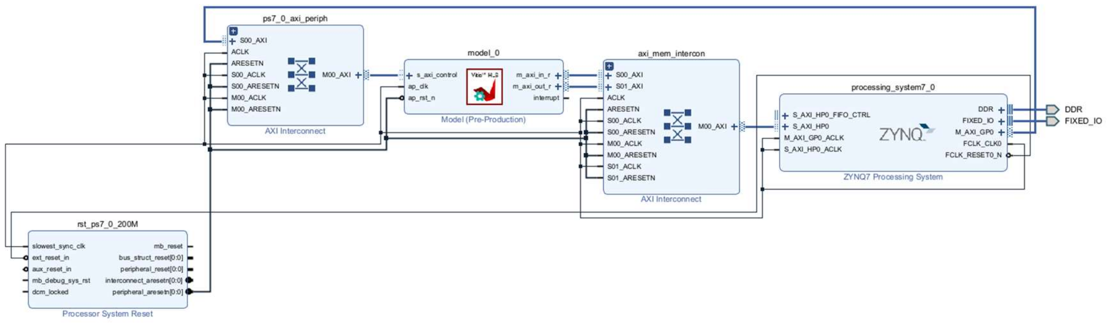


Figure 11、Block design of generator model.

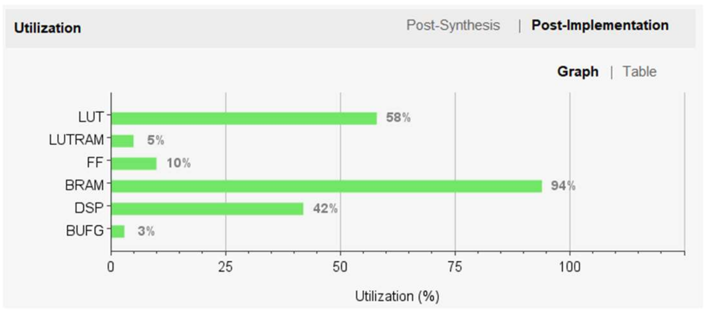


Figure 12、Utilization.

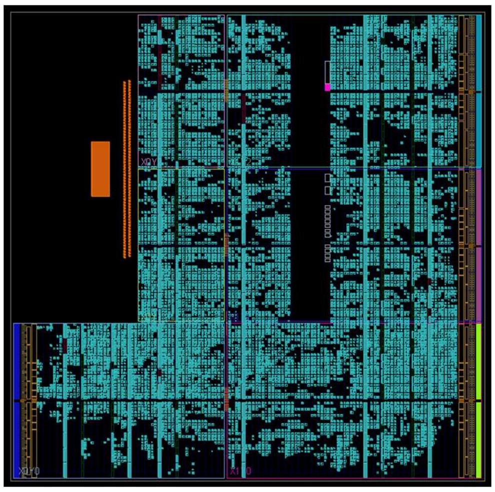
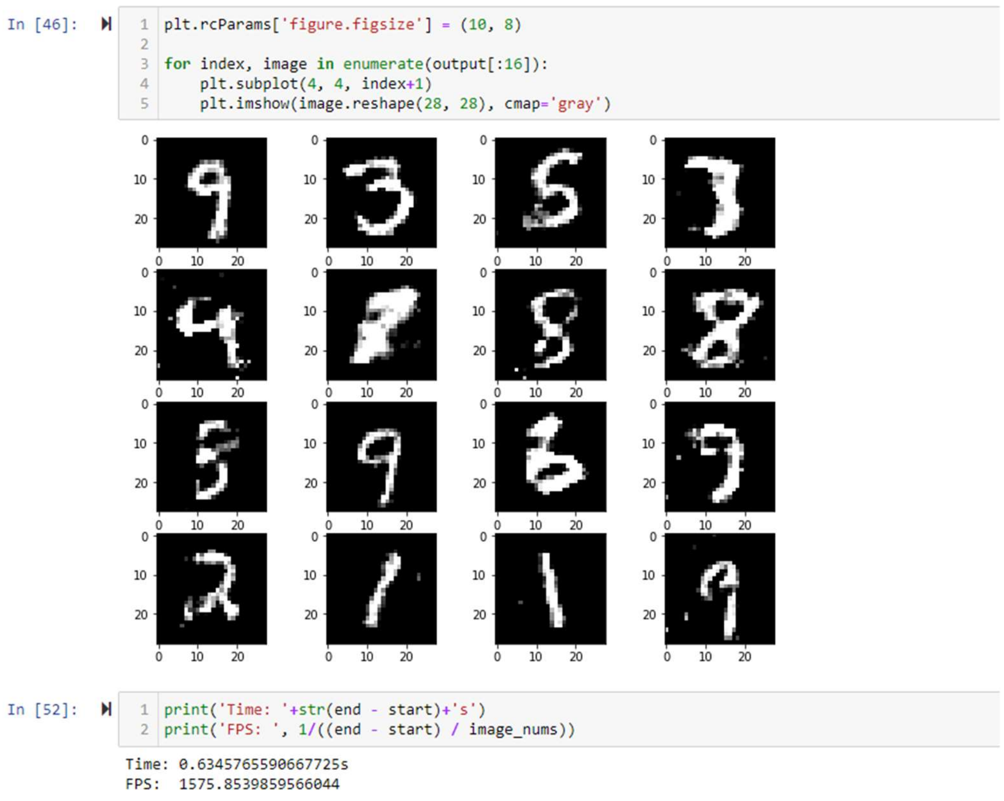


Figure 13、FPGA device usage.

The result on the PYNQ-Z2 shows in Figure 14. Import the overlay and bitstream generated from Vivado, record the start time and end time of the accelerator generating 1000 images, the throughput is 1575.85 FPS.



```
In [46]:
plt.rcParams['figure.figsize'] = (10, 8)

for index, image in enumerate(output[:16]):
    plt.subplot(4, 4, index+1)
    plt.imshow(image.reshape(28, 28), cmap='gray')
```

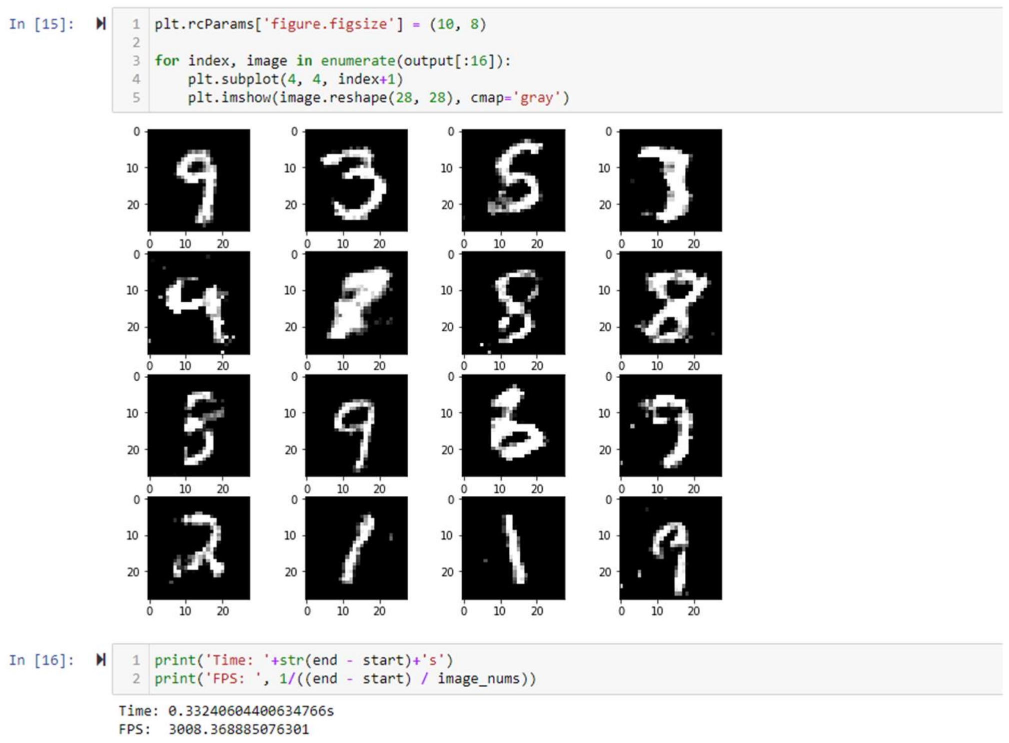
```
In [52]:
print('Time: '+str(end - start)+'s')
print('FPS: ', 1/((end - start) / image_nums))
```

```
Time: 0.6345765590667725s
FPS:  1575.8539859566044
```

Figure 14、Accelerating result on PYNQ-Z2.

Next, we change the ZYNQ7 Processing System PL clock from 100 MHz to 200 MHz, and get the result on PYNQ-Z2 in Figure 15. We get the throughput in 3008.36

FPS, which is higher than 2951.24 FPS on CPU and 1333.21 FPS on GPU. It achieves the higher throughput compare to CPU and GPU on our accelerator.



```
In [15]:
plt.rcParams['figure.figsize'] = (10, 8)

for index, image in enumerate(output[:16]):
    plt.subplot(4, 4, index+1)
    plt.imshow(image.reshape(28, 28), cmap='gray')
```

```
In [16]:
print('Time: '+str(end - start)+'s')
print('FPS: ', 1/((end - start) / image_nums))
```

```
Time: 0.33240604400634766s
FPS:  3008.368885076301
```

Figure 15、Accelerating result on PYNQ-Z2.

5. Conclusion

In this final project, we learn how to build a neural network model accelerator on FPGA without using FINN. Otherwise, we learn how to use streaming interface connection between multiple kernels. The experience we learned improve our HLS skills. In the future, we can try to implement more neural network architecture, like convolution, and we can try much lower-bit weight and activation accelerator, like 4-bit, 2-bit or binary neural network.

6. GitHub Link

https://github.com/alankuo04/AAHLS_Final