NN Synthesis Project

Yu-Cheng Wu
B08901027

Guan-Wei Wu
B08901019

I. Introduction

In this project, we aim to synthesize a deep neural network (DNN) model which does MNIST classification with direct logic implementation and convert the circuit to AIG format, which the number of AND gates is required to be less than 500k. We first trained a model using quantization-aware-learning and the model architecture is introduced in SECTION II. After training the model, we then synthesized it and generated a RTL circuit (verilog file) by transforming the weights of each layer into a Booth matrix and finding partial sums which can be shared by different outputs. The synthesis algorithm we used was proposed in [1] and is presented in details in SECTION III. Our model performs 96.23 percent accuracy on MNIST and the number of AND gates of its AIG version is 448129. In addition to the main project, we synthesized our model for FPGA with Vivado and finished the implementation step. The results are performed in SECTION IV.

II. NN Model Design and Performance

A. Model with Top-1 Accuracy

For MNIST dataset, our model with top-1 accuracy composed of a convolutional layer and three fully-connected layers. For the convolutional layer, its input and output channel were set to 1 and 3 with 3*3 kernels size and the padding and the stride were set to 0 and 2. We used relu6 as the activation function which clipped the weighted results to an upper bound of 6 and lower bound of 0 as we made a 4-bit quantization for both activation function and weights of this model. Besides, we applied batch normalization in the convolutional layer and dropout method, which the dropout rate was set to 0.2, in the first fully-connected layer to boost our training efficiency and avoid overfitting. Furthermore, learning rate scheduling was applied to the model by initially setting the learning rate to 0.002 and exponentially reducing it by 5 percents for each epoch. We preprocessed the dataset as well by normalizing pixels' value and augmented data by small-angle random rotation. The model performs 96.23 percent accuracy on ABC and the number of AND gates is 448129, which meets the requirement of problem description.

B. Compact Model with Decent Performance

To optimize the usage of resources, a 3-bit quantized neural network was taken into our consideration. Although it also consisted of a convolutional layer and three fully-connected layers, which was same as the previous model, we designed relu3, which was defined as $max(0, min(x, 3))$, as the activation function and correspondingly set the quantization for activation function and weights to 3 bits. We also made an attempt to apply the quantization method to weights only and changed different sizes of layers to see the difference. Consequently, the 3-bit model with bigger FC1 layer (output features = 80) displayed great performances on MNIST, which accuracy (95.1 percent) is only one percent lower than the 4-bit model while its circuit size is only three fourths of the 4-bit model.

C. Model Trained with Knowledge Distillation

To further improve the performance, we applied knowledge distillation technique to our model training process. We first trained a so-called teacher model which is much larger than our targeted student model. In our design, the architecture of the teacher model was nearly identical to that of the student model, except for the bigger output channel (8) of the convolutional layer and the bigger output feature map of the first fully-connected layer. We then trained the student models along with the teacher model with quantization of 3 bits and 4 bits. We also trained the 3-bit models with different alpha scheduling, which represented the ratio of ability that the student model learns from the distribution of the teacher model. One way of scheduling was to decrease the alpha from 0.9 to 0.1 during the training process, another way was to keep training at an alpha of 0.9 without changing. Due to the experimental results shown in TABLE IV, knowledge distillation improved the performances for both student models on the testing accuracy, especially for that of 3-bit model. What's more, for the 3-bit student model, we found that fixing the value of alpha at 0.9 resulted in better testing accuracy of AIG circuits. However, for the 4-bit model, knowledge distillation technique's effect was unsignificant and the number of AND gates surpassed the limit of 500k as well. Nonetheless, the experiment of 3-bit model still show that knowledge distillation truly enhances performance while we trained more restricted models.

TABLE I
MODEL ARCHITECTURE.

model	conv	fc	Qbit (a, w)
(1)	1, 3, 3, 0, 2	3*13*13, 32, 32, 10	(4, 4)
(2)	1, 3, 3, 0, 2	3*13*13, 32, 32, 10	(4, 3)
(3)	1, 3, 3, 0, 2	3*13*13, 32, 32, 10	(3, 3)
(4)	1, 3, 3, 0, 2	3*13*13, 80, 32, 10	(3, 3)
(5)	1, 5, 3, 0, 2	5*13*13, 32, 32, 10	(3, 3)

TABLE II
ACCURACY AND CIRCUIT SIZE.

model	best valid	test accu.	AIG accu.	# of AND gates
(1)	**95.99%**	**96.21%**	**95.73%**	**516492**
(2)	96.22%	96.21%	86.93%	309698
(3)	93.86%	93.61%	84.80%	245716
(4)	**94.92%**	**95.42%**	**95.10%**	**323941**
(5)	95.68%	95.32%	93.77%	325484

TABLE III
MODEL ARCHITECTURE WITH KNOWLEDGE DISTILLATION.

model	conv	fc	Qbit (a, w)
(T1)	1, 8, 3, 0, 2	8*13*13, 512, 128, 10	(4, 4)
(S1)	1, 3, 3, 0, 2	3*13*13, 32, 32, 10	(4, 4)
(T2)	1, 8, 3, 0, 2	8*13*13, 512, 128, 10	(3, 3)
(S2)	1, 3, 3, 0, 2	3*13*13, 32, 32, 10	(3, 3)

III. Synthesis Flow

A. Expand Convolutional Layers

To represent a convolutional layer by a single matrix, we first squeezed all input feature maps into a one-dimensional vector, which length is equal to (input feature map size * number of input channels). As we can represent the output feature maps in the same way, we then constructed a matrix which size is equal to (output vector length * input vector length) and put all the weights in right positions as shown in Fig.1.

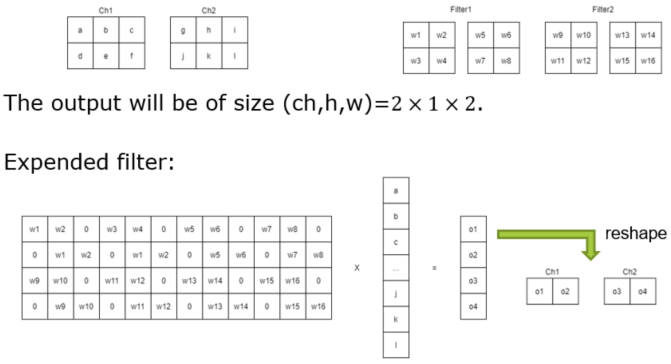

Fig. 1. Example of expanding a convolutional layer.

B. Booth Encoding

To reduce the number of terms we needed to sum up in each layer, we used the Booth Encoding method which was also applied in [1]. When there are consecutive 1's existing in the original Boolean matrix, we can express the same value from less non-zero number if we introduce -1 and represent the value in ternary way. For example, 0111 can be represented by 100(-1) and the number of terms needed to be summed up decreases (from 3 terms to 2 terms) as we only used add and shift operation in our circuit.

TABLE IV
ACCURACY AND CIRCUIT SIZE OF MODEL USING KNOWLEDGE DISTILLATION.

model	alpha	best valid	test accu.	AIG accu. # of AND gates
(T1)	–	98.56%	98.36%	– –
(S1)	–	95.99%	96.21%	95.73% 516492
(T1)+(S1)	0.9	96.24%	96.12%	94.79% 526876
(T2)	–	98.00%	97.91%	– –
(S2)	–	93.86%	93.61%	84.80% 245716
(T2)+(S2)	0.9	95.15%	95.06%	88.11 302894
	0.9-0.1	94.08%	94.49%	– –

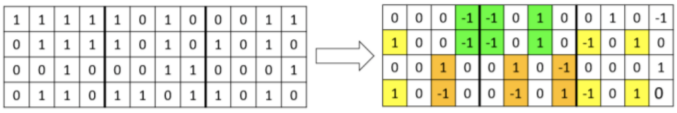

Fig. 2. Example of Booth Encoding.[1]

C. Find Sharing

The Find-Sharing algorithm is aimed to find partial sums which can be shared by different outputs in the same layer and reduce interconnect cost of our circuit. The algorithm we applied was proposed in [1].

Before finding sharing, the interconnect cost of a weight matrix defined in [1] is equal to the product of weight matrix size and 2b, which b is the number of quantization bits we used in our model. This formula comes from the reason that we need a wire to connect every weight bit and input bit and a product of two b-bit values needs 2b bits to represent. A sharing in matrix W is defined as a tuple (R,C) in [1] which R refers to a set of row indices and C refers to another set of column indices. For any two elements r1, r2 in set R, W[r1][C] = W[r2][C] or -W[r2][C] and both W[r1][C] and W[r2][C] are non-zero. Before sharing, the interconnect cost of W[R][C] is 2b*abs(R)*abs(C). After sharing, the cost is equal to the sum of 2b*abs(R) (sub-adder's inputs) and 2b*abs(C) (sub-adder's outputs), as a result, the gain of this sharing is defined as 2b*(abs(R)abs(C)-abs(R)-abs(C)).

The Find-Sharing algorithm first constructs an undirected weighted complete graph G which every vertex in G represents a row in weight matrix. For any two row r1, r2, we can find two set C1, C2 which W[r1][C1] = W[r2][C2] and W[r1][C1] = -W[r2][C2]. Then, we define sharing S-same as (set(r1,r2),C1) and sharing S-diff as (set(r1,r2),C2) and the weight of edge(r1,r2) is equal to gain(S-same)+gain(S-diff). After building the graph, we used the function MaxWeight-

Matching in python package NetworkX to acquire a set of edges with maximum weighted sum under the constraint that edges in the set can't share any vertex. As every edge represents a row pair, we calculated the gain of S-same and S-diff of each pair and collected the sharing which gain is bigger than 0. We refined the weight matrix by replacing column indices in sharing with 0 and repeated the algorithm until no more sharing had been found. The algorithm flow is presented in Fig.6.

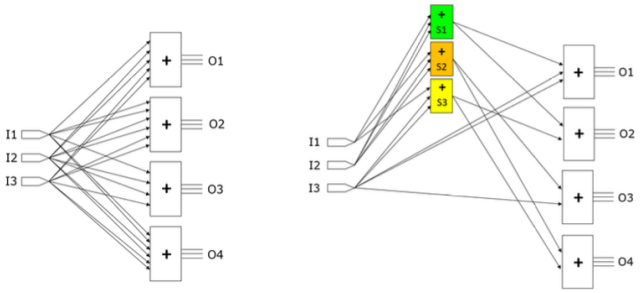

Fig. 3. Example of circuit before and after sharing[1].

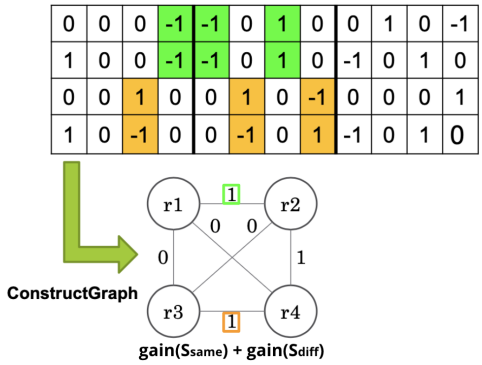

Fig. 4. Example of constructing graph.

Fig. 5. Example of MaxWeightMatching.

D. Generate Circuits

After dividing all terms into a sharing set and a non-sharing set, we first connected wires in the sharing set to acquire several partial sums, then we connected each output with partial sums it used and other products which came from the non-sharing set. We also applied relu6 function to each output in non-output layers and added additional circuit which generated one-hot vectors in output layer. We generated our

Algorithm 1 FindSharing

Input: An $m \times nb$ Booth matrix W_B
Output: A list of shared terms

$C := \{0, \ldots, nb-1\}$;
$unSharedCols$:= an empty hash table;
for $r = 0$ to $m-1$ **do**
    $unSharedCols$[r] = C;
$result$:= [];
$sharings$:= *FindMaxPairing*(W_B,$unSharedCols$);
while *size*($sharings$)> 0 **do**
    append elements in $sharings$ to $result$;
    $unSharedCols$:= *refine*($unSharedCols$,$sharings$);
    $sharings$:= *FindMaxPairing*(W_B,$unSharedCols$);
return $result$;

Algorithm 2 FindMaxPairing

Input: A Booth matrix W_B, a hash table D that maps row indices to a set consists of column indices
Output: A list of shared terms

$G(V, E)$:= *ConstructCompleteGraph*(W_B,D);
$pairs$:= *MaxWeightMatching*(V,E);
$result$:= [];
foreach $pair$ in $pairs$ **do**
    S^{same}, S^{diff} := *PairRows*(vertex 1 of $pair$, vertex 2 of $pair$);
    if $gain(S^{same}) > 0$ **then**
        append S^{same} to $result$;
    if $gain(S^{diff}) > 0$ **then**
        append S^{diff} to $result$;
return $result$;

Fig. 6. Algorithm proposed in [1].

RTL circuit layer by layer and put them under top module "model", then we converted it to AIG with Yosys.

Fig. 7. Circuit of sharing and weighted sums.

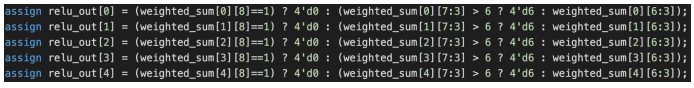

Fig. 8. Circuit of relu6.

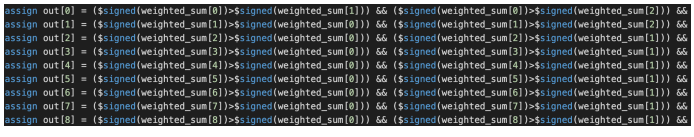

Fig. 9. Circuit of generating one-hot vectors.

IV. FPGA Implementation

A. Simulation

We used Vivado 2021.1 version to realize the FPGA implementation. We create a new RTL project with our system verilog files added, along with the constraint file named "Nexys-Video-Master.xdc" and the "Nexys-Video" default board. After that, we created a new IP with the memory type of single port ROM and the width and depth of port A were set to 3136 and 2000 according to input data. Finally, we loaded the "1000-mnist-bramData.coe" as the memory initialization file and generated the output product. After adding the simulation sources and running, we obtained the result of the correct count indicating the accuracy of the system on the testing set as shown in Fig.10.

B. Synthesis and Implementation

For the synthesis, we set the strategy to "Flow-AlternateRoutability" and directly run the synthesis. After synthesis was finished, we could open the synthesis design and saw the detailed hardware summary such as the usage of LUTs and FFs. The timimg report including slack information and the power report were both available. On the other hand, for the implementation, we set the strategy to "Congestion-SpreadLogic-medium". After implementation is finished, we could also obtain the final report for hardware usage, timing information, and power distribution. All results for our model are shown in Fig.11, 12, and 13. As we synthesized with clock period equal to 50ns and designed a pipeline system which each layer represented a stage, latency of our design equals to 80ns (20ns * 4) and the throughput equals to 50M pictures per second (1 / 20ns).

```
========
index:       9999 true label:  6

pred:0001000000
correct
========
correct count:       9616
$finish called at time : 1400105 ns : File "C:/Users/User/Desktop/files/sim/tb_unpacked.sv" Line 59
run: Time (s): cpu = 00:00:29 ; elapsed = 00:00:39 . Memory (MB): peak = 1413.230 ; gain = 21.754
```

Fig. 10. Accuracy of Vivado simulation.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1 (active)	constrs_1	synth_design Complete!								101675	11814
impl_1	constrs_1	route_design Complete!	0.124	0.000	0.060	0.000	0.000	1.764	0	80284	11815
Out-of-Context Module Runs											

BRAM	URAM	DSP	Start	Elapsed	Run Strategy
0.0	**0**	**0**	**6/21/22, 10:01 AM**	**00:17:04**	**Flow_AlternateRoutability (Vivado Synthesis 2021)**
174.5	0	0	6/21/22, 10:25 AM	01:08:51	Congestion_SpreadLogic_medium (Vivado Implementation 2021)

Fig. 11. Hardware information.

References

[1] Y.-S. Huang, J.-H. R. Jiang, and A. Mishchenko, "Quantized Neural Network Synthesis for Direct Logic Circuit Implementation," in Proceedings of International Workshop on Logic & Synthesis (IWLS), 2021.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.124 ns	Worst Hold Slack (WHS):	0.060 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	25915	Total Number of Endpoints:	25915	Total Number of Endpoints:	12166

All user specified timing constraints are met.

Fig. 12. Timing report.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.764 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**30.8°C**
Thermal Margin:	54.2°C (16.0 W)
Effective ϑJA:	3.3°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

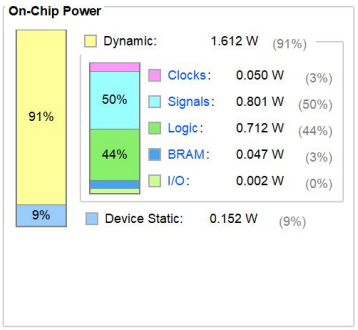

Fig. 13. Power report.

V. Job Division

B08901019:
Training 3-bit and 4-bit models, implement Find-Sharing algorithm, generate circuit, implement models with Vivado.

B08901027:
Training 4-bit models, implement Find-Sharing algorithm, generate circuit, generate AIG and test it on ABC.