ECE 1718 Assignment #1

Group #1
February 19, 2023

Members:
Teng Shu shuteng 1003533651
Yih Cheng cheng442 1009206878
Yang Chen chenya26 999551783
Chen Liang liangc78 1006621443

Milestone #2

Question 1: Based on Amdahl's law, what is the maximum speedup that you can hope to achieve by optimizing the three hottest functions you found?

Based on the experiment, without modification, the overhead for forward_pass, backward_pass and update_weights were 41.46%, 27.40% and 24.80% respectively. The total overhead for these three functions was 93.66%. Based on Amdahl's law, the maximum speedup would be $S = 1/(1 - P) = 1/(1 - 0.9366) \approx 15.77$.

Question 2: What is the total data footprint of the program?

Variable Name	Type	Number of Elements	Size (Bytes)
filter_size	int	1	4
eta	double	1	8
batch_size	int	1	4
conv_w	double	5*7*7	1960
conv_b	double	5*28*28	31360
conv_layer	double	5*28*28	31360
sig_layer	double	5*28*28	31360
max_pooling	char	5*28*28	3920
max_layer	double	5*14*14	7840
dense_input	double	980	7840
dense_w	double	980*120	940800
dense_b	double	120	960
dense_sum	double	120	960
dense_sigmoid	double	120	960
dense_w2	double	120*10	9600
dense_b2	double	10	80
dense_sum2	double	10	80

dense_softmax	double	10	80
dw2	double	120*10	9600
db2	double	10	80
dw1	double	980*120	940800
db1	double	120	960
dw_max	double	5*28*28	31360
dw_conv	double	5*7*7	1960
db_conv	double	5*28*28	31360

The total data footprint is 2085296 bytes.

It was tried to replace all ‘double’ with ‘float’, and this does improve the runtime only by a little (58.783989101-57.683606176 = 1.100382925s). This could be due to the complexity of double precision number calculations being higher than the single precision number calculations, which could potentially mean more lines of assembly code. There could be more instructions that need to be executed in the original version than in the float version, which will also lead to more cache references and more cache misses.
Perf reported that the original version completed 693,500,951,323 instructions and 11,462,716,758 cache references with 4,830,198 cache misses, and the cache miss rate was about 0.042%. However, the float version completed 628,584,403,871 instructions, and 5,574,716,321 cache references with 3,808,435 cache misses, and the cache miss rate was about 0.068%. It can be observed that the original version runs more instructions, and although it had a lower cache miss rate, it still had more cache misses.

The classification accuracy troped 0.883333-0.878333 = 0.005 which was about 0.5%. This could be due to the float type being less accurate than the double type, so there might be some values being approximated during the calculation if the float cannot accurately represent it.

Question 3: What is the cache miss rate?

The cache miss rate was about 0.042% for the original version as mentioned in Question 2. These machines have caches of several megabytes in size, so the cache misses would be mostly, if not all, caused by first-time data access.

Preliminary Results:

	SISD (double)	SIMD (float)
Accuracy	88.33%	87.83%

<table>
<tr><td>Performance</td><td>Total: 57.29 sec<table><tr><td>f_pass</td><td>41.34%</td><td>23.68 sec</td></tr><tr><td>b_pass</td><td>27.96%</td><td>16.02 sec</td></tr><tr><td>u_weight</td><td>24.39%</td><td>13.97 sec</td></tr></table></td><td>Total: 32.68 sec<table><tr><td>f_pass</td><td>28.86%</td><td>9.43 sec</td></tr><tr><td>b_pass</td><td>46.26%</td><td>15.12 sec</td></tr><tr><td>u_weight</td><td>17.15%</td><td>5.60 sec</td></tr></table></td></tr>
</table>

The actual speed-up is about 1.75x, which is relatively small compared to the ideal speed-up from using SIMD instructions. This is due to extra memory allocation to use SIMD which also requires extra time to do the data copying.

Milestone #3

Task 1: Measure the percentage of time spent executing different segments of the function that is targeted for hardware acceleration (forward_pass())

The following table shows the time we measured for each segment of *forward_pass()* and we calculated the percentage of time consumed on these components. Based on our result, we can find that convolution operation + sigmoid activation and dense layer consumed the most computing time in f*orward_pass()* which are the bottlenecks.

	Time consumed per cycle(ms)	Percentage of time consumed
Convolution Operation + Sigmoid Activation	0.172647	59.539%
max_pooling	0.006038	2.0823%
max_layer	0.00013	0.044832%
Dense Layer	0.110209	38.00664%
Dense Layer	0.000872	0.300718%
Softmax output	7.7e-5	0.026554%

Task 2: Evaluate the impact of using different variations of fixed-point approximations on the elements in the network on the model accuracy, cost (cache), and performance per watt

A software model of forward_pass() using only fixed-point calculations was implemented to evaluate the impact of using different variations of fixed-point approximations. A number is assumed to take 32-bit, and a portion of these bits will be used to represent decimals. The accuracy of each case is shown in the table below.

Bits to represent decimals	0	2	4	6	8	10	12	14	16	18
Accuracy	0.0883333	0.806667	0.873333	0.886667	0.881667	0.883333	0.883333	0.631667	0.33	0.24

This table shows that in order to maintain some level of accuracy, 4-bit to 12-bit would be a good choice. This result makes sense since it is expected to have low accuracy when the number of bits used to represent decimals is small due to the round error, and to have low accuracy when the number of bits used to represent decimals is too big due to the higher possibility to have a number overflow.

However, measuring the cache of this model would be inaccurate since in software, it is hard to use an arbitrary number of bits to represent an integer. So for the evaluation of the cache, it will be assumed that the integer part will take 32-bit, and an extra number of bits will be added for the decimal part. This will be done in the calculation, so the result is unrelated to the accuracy evaluation.

Bits to represent decimals	0	2	4	6	8	10	12	14	16	18
Cache [Bytes]	552,780	587,329	621,878	656,426	690,975	725,524	760,073	794,621	829,170	863,719

Note that due to the accuracy being acceptable when using 4 to 6 bits for decimals from a total of 32 bits, it was chosen to implement the hardware without having decimal bits as additional bits, which means the total cache would be the same as using 0 bits to represent decimals.

Task 3: Evaluate the impact of nonlinear function approximations on the model accuracy and cost (cache)

Nonlinear function approximations were only done for exponential and sigmoid (relu is not used, and softmax is using exponential). The cost analysis was done assuming all hard-coded values are stored in a (32+6) bits register where 32-bit for the integer part, and 6-bit for the decimal part.

Approximation	Accuracy	Cost [Bytes]
No approximation	0.883333	N/A
exponential	0.868333	205*(32+6)/8 ≈ 973
sigmoid	0.901667	91*(32+6)/8 ≈ 421

Note that for the sigmoid case, the accuracy even increased compared to the accurate version. This could be due to this approximation introducing some noise to the system which somehow reduces the level of overfitting.

Task 4: Provide an estimate of the system's throughput (number of cycles per image) and memory bandwidth

Our estimated system throughput is 1,572,816 cycles per image with a 7.3ns clock cycle according to the HLS synthesis report. Assuming 10 images per second, data exchange between host and kernel will require a memory bandwidth of

(sizeof (d_type) * (28+pad_h)*(28+pad_w) + sizeof(d_type))*10 per second

where d_type is the resulting data type chosen. For example, since we are using "integer" in our C-model, therefore the data exchange will require a memory bandwidth of 5271630 bytes.

Task 5: Prepare "Golden Vectors"

The preparation of the "Golden Vectors" is done to verify the hardware implementation, by storing the file of the input and output communication between the software part that is intended to run on the host, and the software model of the hardware accelerator.

The idea is to create a set of vectors that is used as a reference to represent the expected behavior of the hardware accelerator. These golden vectors will be compared against the results of the hardware implementation to verify its accuracy and functionality.

The input part of the golden vector is generated from the C model weights, and their corresponding control values. Due to the amount of weights that need to be sent to the accelerator, the input part of the golden vectors is too large to be put into the report, but the file can be found in the hardware source code along with the testbench.

The followings are the expected output vectors assuming the design needs A clock cycle to finish the calculation:

Cycle	Control	Done (from design)	Bus (from design)
A	0	1	0
A+1	0	1	0
A+2	0	1	0
A+3	0	1	0
A+4	0	1	0
A+5	0	1	0
A+6	0	1	0
A+7	0	1	64
A+8	0	1	0
A+9	0	1	0

Task 6: Description of Hardware C-Model

The hardware C-model is designed in host-kernel mode. The host is responsible for scattering the weights and bias into device memory, along with controlling the inputs to the kernel. When the “done” signal is triggered from the device, the host then gathers the results from the device storage. The kernel is responsible for running the full network on the device. The pre-trained weights for the convolution and dense layers are stored in device memory prior to the inference process. During the calculations of each layer, these weights and bias are read and respective calculations are made.

Clock cycles required	Time required [s]
400720	10ns*400720 = 0.0040072 sec

Resources	Resource_Count
BRAM	359
DSP	349

FF	49814
LUT	69695

Description of HDL model with performance and cost estimates

Description of HDL model

The HDL model was made by a big Finite State Machine (FSM) which controls data loading and initiates each calculation model based on the control signal it received from the host.

The first step is to load all necessary data from the host to the HDL model. This is done by the top FSM and a list of modules named data_loader. Top FSM will check the control signal from the host, and assert the corresponding load enable signal for the data_loader. The data_loader will read the input bus and put the value into the correct register.

Most data was stored in registers for speed and simplicity, except for dense_w since it is an array of 980*120 and will consume lots of resources if put in registers. It was decided to have registers that hold an array of 980 elements in dense_w to do the calculation, and values of dense_w were stored in a block memory and would need to do a data load for every 980 values of it.

Modules of convolution, max pooling, dense layer1, dense layer2 and softmax were implemented independently and communicated by synchronous signals controlled by the top FSM. If the module is sequential, a start signal will be initiated by the top FSM, and a done signal will be sent back to the top FSM, triggering the next calculation. If the module is combinational, then the calculated value will be used directly by the top FSM.

The convolution model is designed the same way as suggested by the paper [1]. There were 49 multiplications calculated at the same time. Then all these values are added, then the sum will be sent to a sigmoid module to compute the sig_layer value.

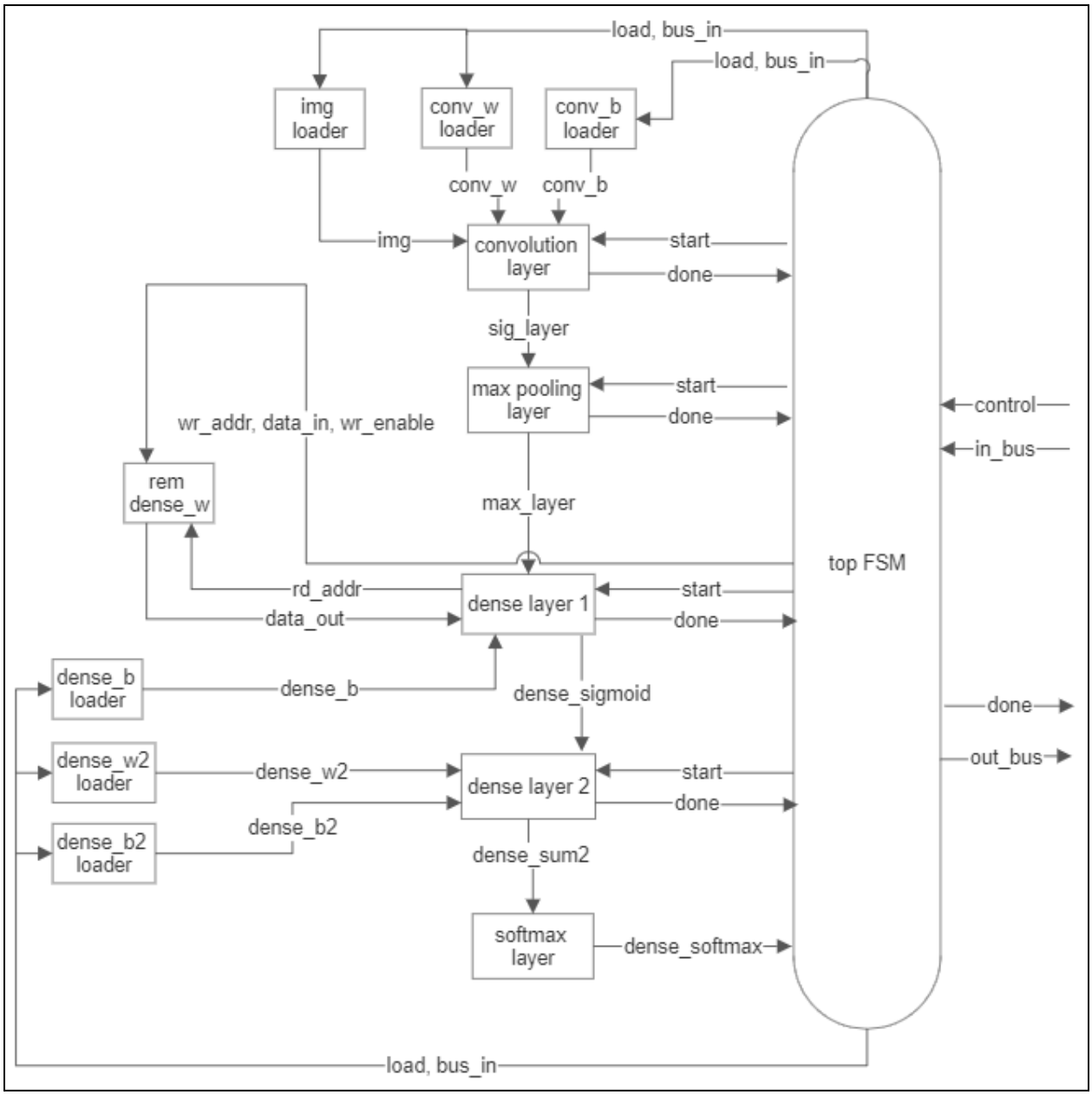
In the max pooling layer, multiple comparators were used to find the cur_max from sig_layer. This is combinational so all the rest is just going through all elements for max_layer and storing the cur_max.

Dense layer1 and dense layer2 were mostly the same, except dense layer1 gets the dense_w from a block memory. The sum was computed by 14 fixed point multipliers and summed using combinational logic. Each output element in dense layer1 needs 70 cycles to compute the sum plus one cycle to compute the sigmoid once the dense_w is loaded.

Softmax module is purely combinational, so no control signals are needed. This module is implemented by 10 exp modules and 10 fixed point division modules.

In the end, the top FSM will do data scatter which asserts the top-level done signal, and send each dense_softmax value out through the output bus.

All of the fixed point multiplication, fixed point division, exponential and softmax were done by desiccated combinational modules. Note that the estimation of exponential and softmax calculations were done by piecewise approximation, which will need lots of MUXs in hardware and also will have quite a big propagation delay.



Performance and cost estimates

By simulation, the implemented HDL model can finish the first round of forward pass calculation in 255619 clock cycles. A detailed breakdown of time consumption for each state is shown in the table below. The timing analysis suggested that the maximum clock frequency is 51.279422 MHz, so the time consumed for the first round would be 0.0049848261190s.

For the following rounds, since the weighting of the neural network will not change, there is no need to resend weights which means the time consumption would be approximately 0.0025843505240s.

State	Clock cycles required	Time required [s]
Load img	1120	0.0000218411200
Load conv_w	245	0.0000047777450
Load conv_b	3920	0.0000764439200
Load dense_w	117600	0.0022933176000
Load dense_b	120	0.0000023401200
Load dense_w2	1200	0.0000234012000
Load dense_b2	10	0.0000001950100
Convolution	3923	0.0000765024230
Max pooling	983	0.0000191694830
Dense layer1 Load dense_w to registers calculation	 117840 8524	 0.0022979978400 0.0001662265240
Dense layer2	123	0.0000023986230
Data scatter	11	0.0000002145110
Total	255619	0.0049848261190
Total without loading weights	132524	0.0025843505240
Total with only load img and calculation (theoretical)	13684	0.0002668516840

The design turns out can be implemented sufficiently in “Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit“, which costs $1678.

Explanation of verification methodology (comparison with golden vectors)

The Golden vectors were generated from the fixed point C model. It contains all input values from the host with their control values which will be used by the HDL model to distinguish what data is on the bus. Expected output values were also generated. The HDL testbench will read these golden vectors, set the control signal and the input bus to their desired values, then initiate the calculation for the HDL model. In the end, the testbench will compare the value computed by HDL and from the golden vector and will flag if there are any mismatches. This will also show any incorrectness of our design. The test case is passed when the expected output from the golden vectors matches the actual output from HDL.

Throughput, area, power and performance analysis and comparison of the HDL vs SIMD method. Compare in terms of throughput and accuracy

HDL method

Total Area	423876 cells
Total On-Chip Power	1.172 W
Maximum Clock Frequency	Critical path delay is 19.501 ns, so maximum frequency is 51.279422 MHz
Throughput (the following rounds)	387 rounds/s
Expected Accuracy	0.886667

SIMD method

Throughput	10668 rounds/s
Expected Accuracy	0.8783

It is observed that the HDL version is actually slower than the SIMD method. This is likely due to some potential design issues in the HDL method. From the table above about clock cycles needed for each state, it was realized that data loading is taking about half of the total time. The “load dense_w to registers” is also the main bottleneck on the following rounds (weights will be loaded in the first round so no need to load them again in the following rounds), which takes about 90% of the time. Maybe this data loading can be optimized using multi-port ram, or using multiple rams for dense_w so fewer clock cycles will be needed for the same amount of data.

Another factor might be the critical path is too long which forces the design to use a slower clock. The 50MHz clock is too slow compared to the CPU clock that the SIMD runs on, so this can greatly impact the result. One way to optimize it is to break the critical path with registers.

This will add extra delay to the combinational circuit but can help to reduce the propagation delay.

Accuracy-wise, the SIMD method has a slightly lower accuracy. This may be due to SIMD being used also for training to get the maximum speed up but the trained weights could be less accurate. It was also observed that the estimated HDL method accuracy is even higher than the original model that uses double precision. This could be due to the less accurate HDL method introducing some noise which reduces overfitting. However, this could be data set dependent, so no way to tell if this “higher accuracy” is better or not. Overall, HDL and SIMD methods are having comparable accuracy, and the SIMD method ends up faster than our HDL method.

Note that the estimated on-chip power is quite low for the HDL model, so maybe it is suitable to use it in power-critical situations with the cost of speed.

Contribution of each team member

Teng Shu - Hardware implementation, integration and verification, report, and presentation
Yih Cheng - Implement software model, assist with the hardware design, report, and presentation
Yang Chen - Hardware implementation, report, and presentation
Chen Liang - Implement software model, report, and presentation

References

[1] T. H. Vu, R. Murakami, Y. Okuyama and A. Ben Abdallah, “Efficient Optimization and Hardware Acceleration of CNNs towards the Design of a Scalable Neuro inspired Architecture in Hardware,” 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), Shanghai, 2018, pp. 326-332, doi: 10.1109/BigComp.2018.00055.