CS5222 Project 2

Luca Krueger

March 26, 2020

1 Matrix Multiplication Pipeline Optimization in HLS

A. Understanding the baseline matrix multiply

The baseline code does not perform very well. The following performance estimates will be taken as reference for further optimizations, see subsection B.

Latency		Interval		Pipeline
min	max	min	max	Type
230331	230331	230332	230332	none

Table 1: Performance: baseline matrix multiplication algorithm

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	538
FIFO	-	-	-	-
Instance	0	5	384	752
Memory	16	-	0	1
Multiplexer	-	-	-	559
Register	-	-	779	-
Total	16	5	1163	1847
Available	280	220	106400	53201
Utilization (%)	5	2	1	3

Table 2: Resource utilization of the baseline code

Module	Number of instances
floating point adder	1
floating point multiplier	1

Table 3: Utilization of multipliers and adders of the baseline code

B. Pipelining in HLS

1. Optimization attempt: pipeline the most inner loop
   ⇒ speedup of ≈ 2.13 regarding the max latency compared to the baseline code

Latency		Interval		Pipeline
min	max	min	max	Type
107995	107995	107996	107996	none

Table 4: Performance: most inner loop pipelined

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	557
FIFO	-	-	-	-
Instance	0	5	384	751
Memory	16	-	0	0
Multiplexer	-	-	-	579
Register	-	-	945	64
Total	16	5	1329	1951
Available	280	220	106400	53200
Utilization (%)	5	2	1	3

Table 5: Resource utilization: most inner loop pipelined

Module	Number of instances
floating point adder	1
floating point multiplier	1

Table 6: Utilization of multipliers and adders: most inner loop pipelined

2. Optimization attempt: pipeline the first inner loop
   ⇒ speedup of ≈ 14.22 regarding the max latency compared to the non optimizeed code

Latency		Interval		Pipeline
min	max	min	max	Type
16194	16194	16195	16195	none

Table 7: Performance: first inner loop pipelined

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26003
FIFO	-	-	-	-
Instance	0	10	732	1462
Memory	16	-	0	0
Multiplexer	-	-	-	4725
Register	-	-	24650	6496
Total	16	10	25382	38686
Available	280	220	106400	53200
Utilization (%)	5	4	23	72

Table 8: Resource utilization: first inner loop pipelined

Module	Number of instances
floating point adder	2
floating point multiplier	2

Table 9: Utilization of multipliers and adders: first inner loop pipelined

The `HLS pipeline` directive for the first inner loop does not completely unroll the most inner loop. The most inner loop can only be parallelized by two units, because the input buffer is stored into two separate BRAM blocks and therefore only provides two separate access channels. The hls tool gives a warning, that there are problems with scheduling the input buffer accesses which is related to this isssue. Optimization of memory accesses will be addressed in subsection C.

3. Optimization attempt: Attempt 2 and pipelining of all input functions
   $\Rightarrow$ speedup of ≈ 16.64 regarding the max latency compared to the baseline code

Latency		Interval		Pipeline
min	max	min	max	Type
13840	13840	13841	13841	none

Table 10: Performance: first inner loop and input functions pipelined

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26037
FIFO	-	-	-	-
Instance	0	10	732	1462
Memory	16	-	0	0
Multiplexer	-	-	-	4793
Register	-	-	24611	6496
Total	16	10	25343	38788
Available	280	220	106400	53200
Utilization (%)	5	4	23	72

Table 11: Resource utilization: first inner loop and input functions pipelined

Module	Number of instances
floating point adder	2
floating point multiplier	2

Table 12: Utilization of multipliers and adders: first inner loop and input functions pipelined

C. Increasing Pipeline Parallelism by Repartitioning Memories

This optimization step introduces repartitioning of the memories `in_buf` and `weight_buf`. We group adjacent columns to a total of factor= 8 blocks. This increases parallelism by a factor 8. Therefore the `vivado_hls` tool instantiates eight times more adders and multipliers, which also increases hardware usage.

$\Rightarrow$ speedup of ≈ 46.14 regarding the max latency compared to the baseline code

Latency		Interval		Pipeline
min	max	min	max	Type
4992	4992	4993	4993	none

Table 13: Performance: optimizations as in B. 3 and memory repartitioning

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3309
FIFO	-	-	-	-
Instance	0	80	5604	11416
Memory	36	-	0	0
Multiplexer	-	-	-	6324
Register	-	-	32355	14129
Total	36	80	37959	35178
Available	280	220	106400	53200
Utilization (%)	12	36	35	66

Table 14: Resource utilization: optimizations as in B. 3 and memory repartitioning

Module	Number of instances
floating point adder	16
floating point multiplier	16

Table 15: Utilization of multipliers and adders: optimizations as in B. 3 and memory repartitioning

As we can see, the hardware usage does not exeed any limits in this configuration. The next step for memory optimization would be a repartitioning into 16 groups. This allows higher parallelism, but also increases the amount of multipliers and adders by factor 4. In this configuration, the usage of flipsflops would exeed the maximum by 5%. This overhead might be solved by more detailed optimizations during hardware implementation, but here it does not meet our requirements. Intermediate partition steps between $8 - 16$ result in a significant increase of latency, most likely caused by the misaligned memory groupings, which causes additional control hardware.

D. Amortizing Iteration Latency with Batching

The optimization step exploits that initialization overhead does not increase proportionally with increasing batch size. The batch size is now 256, the largest possible value, as power to two, for given hardware resources.

$\Rightarrow$ speedup of ≈ 92.84 regarding the max latency compared to the baseline code and normalized to batch size

Latency		Interval		Pipeline
min	max	min	max	Type
79392	79392	79393	79393	none

Table 16: Performance: batch size increased to 256 and optimizations as in C.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3704
FIFO	-	-	-	-
Instance	0	80	5604	11416
Memory	154	-	0	0
Multiplexer	-	-	-	6324
Register	-	-	32591	14129
Total	154	80	38195	35573
Available	280	220	106400	53200
Utilization (%)	55	36	35	66

Table 17: Resource utilization: batch size increased to 256 and optimizations as in C.

E. Extending Batch Size with Tiling

Now the batch size increased to 2048, but the batch is separated into tiles of size 128. The usage of `BRAM_18K` Blocks is now lower, whereas the normalized speedup increased a lot.

⇒ speedup of ≈ 11507.26 regarding the max latency compared to the baseline code and normalized to batch size

Latency		Interval		Pipeline
min	max	min	max	Type
655889	655889	655890	655890	none

Table 18: Performance: tiling and optimizations as in C.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3632
FIFO	-	-	-	-
Instance	0	80	5604	11416
Memory	86	-	0	0
Multiplexer	-	-	-	6319
Register	-	-	32542	14129
Total	86	80	38146	35496
Available	280	220	106400	53200
Utilization (%)	30	36	35	66

Table 19: Resource utilization:tiling and optimizations as in C.

F. Hardware compilation and FPGA testing on the PYNQ

Uploading and execution of the Jupyther notebook unfortunately gives the following error:

```
        TimeoutError                                 Traceback (most recent call last)
<ipython-input-3-28b2e4095493> in <module>()
      7 start_t = time()
      8 dma1.transfer((CLASSES+CLASSES*FEAT+BATCH*FEAT)*4, direction=0)
----> 9 dma2.wait()
     10 fpga_time = time()-start_t
     11

/opt/python3.6/lib/python3.6/site-packages/pynq/drivers/dma.py in wait(self, wait_timeout)
    439             with timeout(seconds = wait_timeout, error_message = Error):
    440                 while True:
--> 441                     if libdma.XAxiDma_Busy(self.DMAengine,self.direction) == 0:
    442                         break
    443

/opt/python3.6/lib/python3.6/site-packages/pynq/drivers/dma.py in handle_timeout(self, signum, frame)
    173
    174     def handle_timeout(self, signum, frame):
--> 175         raise TimeoutError(self.error_message)
    176
    177     def __enter__(self):

TimeoutError: DMA wait timed out.
```

UPDATE: This problem is now fixed. The FPGA was programmed in order to receive the offset values and the full weight matrix in each iteration, whereas this is only required once in the beginning. The implementation on the FPGA gives a speedup of 6.74. The validation error 14.79% is as good as on the CPU.

2 Fixed-Point Optimizations

(1) After setting the `SCALE` factor in `mnist.py` to 100000, a fixed-point validation accuracy of 19.76% can be achieved.

```
Misclassifications (float) = 13.77%
Misclassifications (fixed) = 19.76%
```

(2) Design latencies:

Latency		Interval		Pipeline
min	max	min	max	Type
444135	444135	444136	444136	none

Table 20: Latencies: fixed point

(3) Overall design utilization:

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	129	-	-
Expression	-	-	0	3993
FIFO	-	-	-	-
Instance	0	0	8672	9184
Memory	4	-	8192	2560
Multiplexer	-	-	-	8252
Register	-	-	6580	96
Total	4	129	23444	24085
Available	280	220	106400	53200
Utilization (%)	1	58	22	45

Table 21: Resource utilization: fixed point

(4) The measured system speedup over the CPU fixed point computation is 74.92.

(5) The measured classification on the 8k MNIST test sample is 20.96%.

```
FPGA accuracy: 20.69% validation error
CPU accuracy:  20.69% validation error
FPGA has a 74.92x speedup
```

(6) Instantiation of multipliers

Module	Number of instances
DSP48	129
HW multiplier modules	127
Total	256

Table 22: Utilization of multipliers: fixed point

(7) The pipelined inner-loop of the matrix multiplication achieves an initiation interval of a single cycle, as shown in table 23.

	Latency		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined
- LOAD_OFF_1	5	5	2	1	1	5	yes
- LOAD_W_1	350	350	35	-	-	10	no
+ LOAD_W_2	32	32	2	1	1	32	yes
- LT	443776	443776	3467	-	-	128	no
+ LOAD_I_1	2240	2240	35	-	-	64	no
++ LOAD_I_2	32	32	2	1	1	32	yes
+ L1_L2	646	646	8	1	1	640	yes
+ STORE_O_1	576	576	9	-	-	64	no
++ STORE_O_2	6	6	3	1	1	5	yes

Table 23: Snapshot of the loop latency table

(8) Regarding the loop latency table (table 23), the loading the tiled set of input values requires $\approx 3.5x$ more cycles than the actual matrix multiplication. Hence we can say that this design is bandwidth limited by the bandwidth of the AXI port.

3 Open-ended design optimization

This is the last part of this project which covers an open-ended optimization of MNIST dataset classification on an FPGA. For better comparison, the `Batch size` ($\approx 8k$) of the processed subset is the same as for previous design optimiztations in part 1 and part 2.

Learning First, the learning algorithm will be further optimized by introducing the *cross-entropy* as new error function and the *SELU*(Self-normalizing Linear Unit and *softmax* functions as chained transfer function. The goal is to increase classifiction accuracy to at least 85% without increasing the overall hardware usage. Consequently, the network cannot be extended by another layer of neurons. The output has to be one hot encoded, such that the number of neurons can not be changed either.

Initially the given learning algorithm uses a quadratic error function to calculate the cost after each prediction. But for this particular classification problem, we have two more contraints on the prediction outcome which are ignored, by the current error function:

- the prediction outcome should follow the one hot encoding scheme.
- the prediction outcome should be a valid probability distribution.

Therefore the *cross-entropy* will be used as new error function in combination with the *softmax* function as output transfer function. The *SELU* function and helps to keep the weigths show a self-normalizing behavior. The two distribution parameters λ and α are determined in order to gurarantee this behavior for normalized input values. Therefore the inputs now must be scaled to $[0, 1]$. To improve accuracy and to better fit the weights, the training algorithm is repeated `ITERATION` times with random permutations of the training set. The file `mnist.py` had to be slightly changed to integrate the new learning algorithm. Furthermore extra functionality to plot weight representation and error rate over the training period was added for debugging purposes. The optimized learning algorithm can be found in the file `CrossSLP.py`.

Prediction acceleration Since softmax and *SELU* only scale the denditric potential of the networks neurons, the following equation still holds:

$$\text{argmax}(x) = \text{argmax}(\text{softmax}(\text{SELU}(x)))$$

As we see from the equation above, the transferfunctions are only required for learning, not for prediction. In previous implementations in part 1 and 2, we just computed the argmax(x) of the denditric potentials, which gave us the correct prediction output. This means even after changing the learning algorithm and its transfer functions, the hardware does not necessarily have to change, because the network structure remains the same.

Nevertheless the harware is also optimized in part 3. In the previous implementations, the output of the accellerator transmitted back to the cpu was

always the whole output vector of the network. We still had to compute the argmax(x) on the cpu. First this produces unnecessary transmission overhead, which can be avoided, second the computation of the *argmax* function can also be offloaded. So the harware design was changed accordingly and now the output is only one byte long indicating the predicted number between 0 and 9. Because the output buffer size is reduced by 90% more memory can be used for the input buffer, such that the `TILING` size can be increased to 128 samples. This also give a small latency speedup.

```
FPGA accuracy: 8.15\% validation error
CPU accuracy:  8.06\% validation error
FPGA time:  0.004762924000033308
CPU time:  0.2810866919999171
FPGA has a 59.02x speedup
(* real speedup is slightly higher, because of argmax offloading)
```

The simulation, predicts a lower latency of what can be performed on the FPGA.