CS5222 Project 2
Custom Acceleration with FPGAs

Shen Jiamin
A0209166A
shen_jiamin@u.nus.edu

March 20, 2022

Abstract

In this project, I'm going to port the lab to **PYNQ 2.7** and **Vivado/Vitis 2020.2**. The experiment is done on ASUS RS500-E8-PS4 V2, with operating system Ubuntu 20.04.4 LTS (GNU/Linux 5.4.0-100-generic x86_64).

1 Matrix Multiplication Pipeline Optimization in HLS

1.1 Understanding the baseline matrix multiply (background)

For Vitis 2020.2, the command used should be

```
$ vitis_hls -f hls.tcl
```

The loop details report generated by HLS (as in Table 1a) shows that some pipelining has already been done automatically by Vitis HLS. After inspecting the migration guide, I added two lines in the `hls.tcl`:

```
config_compile -pipeline_loops 0
set_clock_uncertainty 12.5%
```

The performance and utilization estimates of the two profiles, together with those for the following profiles, are reported in Table 6. The new loop details is as Table 1b. It turns out that the overall performance is a little bit worse than documented. This is because every iteration in L3 loop takes 11 cycles and thus 2816 cycles in total to perform a single inner product.

The overall latency of the baseline is 228022 cycles. Since it only predicts on a batch of 8 inputs, the normalized latency is 28502.75 cycles.

1.2 Pipelining in HLS (8 marks)

The work in this section is done with auto pipelining disabled.

1.2.1 Pipelining the L3 (innermost) loop

The code is modified as Figure 1. As reported in Table 3, pipelining L3 reduces its latency from 2816 cycles to 1031 cycles. The L1 is flattened, but L2 cannot be flattened because L2 is not a perfect loop. Thus the result has 2-layer loops where the outer layer has 80 iterations, and the inner layer has 256 iterations. This design utilizes slightly more resources but no more floating-point adders or multipliers. The overall latency is 85286 cycles, which is about 2.67x speedup. Other statistics in detail can be found in Table 6.

1.2.2 Pipelining the L2 loop

The code is modified as Figure 2a. As reported in Table 4a, pipelining the loop body of L2 unrolls L3 and flattens L1. The design introduced parallelism, which uses 16 adders and 16 multipliers. The overall latency is reduced to 7341 cycles, which is about 31.1x speedup relative to the baseline. The initiation interval has increased to 128 cycles, which is pretty heavy. Other statistics in detail can be found in Table 6.

Table 1: Loop details for baseline

(a) Baseline with automatic pipelining

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	1	1	5	yes
- LOAD_W_1_LOAD_W_2	1280	1280	1	1	1	1280	yes
- LOAD_I_1_LOAD_I_2	1024	1024	1	1	1	1024	yes
- L1_L2	82800	82800	1035	-	-	80	no
+ L3	1031	1031	12	4	1	256	yes
- STORE_O_1_STORE_O_2	42	42	4	1	1	40	yes

(b) Baseline without automatic pipelining

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	1300	1300	130	-	-	10	no
+ LOAD_W_2	128	128	1	-	-	128	no
- LOAD_I_1	1040	1040	130	-	-	8	no
+ LOAD_I_2	128	128	1	-	-	128	no
- L1	225536	225536	28192	-	-	8	no
+ L2	28190	28190	2819	-	-	10	no
++ L3	2816	2816	11	-	-	256	no
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

Table 2: Loop details for baseline of array partition

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	1	1	5	yes
- LOAD_W_1_LOAD_W_2	1280	1280	1	1	1	1280	yes
- LOAD_I_1_LOAD_I_2	1024	1024	1	1	1	1024	yes
- L1_L2	11398	11398	1287	128	1	80	yes
- STORE_O_1_STORE_O_2	42	42	4	1	1	40	yes

```
@@ -78,6 +78,7 @@
            T tmp = offset_buf[j];
        L3:
            for (int k = 0; k < FEAT; k++) {
+#pragma HLS PIPELINE II = 1
                tmp += in_buf[i][k] * weight_buf[j][k];
            }
            out_buf[i][j] = tmp;
```

Figure 1: Inserting HLS directive for L3 Pipelining

An important observation is that the design can utilise 16 adders and 16 multipliers instead of being limited by the dual-port RAMs. The reason for that is Vitis HLS infers the RAM type as 1WnR, which is Multi-Ported Memory using Replication, and this type of memory has a single write port and multiple concurrent read ports.

Table 3: Loop details for L3 Pipelining

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	1300	1300	130	-	-	10	no
+ LOAD_W_2	128	128	1	-	-	128	no
- LOAD_I_1	1040	1040	130	-	-	8	no
+ LOAD_I_2	128	128	1	-	-	128	no
- L1_L2	82800	82800	1035	-	-	80	no
+ L3	1031	1031	12	4	1	256	yes
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

Table 4: Loop details for L2 pipelining

(a) L2 pipelining with 1WnR memory

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	2580	2580	258	-	-	10	no
+ LOAD_W_2	256	256	2	-	-	128	no
- LOAD_I_1	2064	2064	258	-	-	8	no
+ LOAD_I_2	256	256	2	-	-	128	no
- L1_L2	2550	2550	1287	16	1	80	yes
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

(b) L2 Pipelining with T2P memory

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	1300	1300	130	-	-	10	no
+ LOAD_W_2	128	128	1	-	-	128	no
- LOAD_I_1	1040	1040	130	-	-	8	no
+ LOAD_I_2	128	128	1	-	-	128	no
- L1_L2	11398	11398	1287	128	1	80	yes
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

```
INFO: [HLS 200-1457] Automatically inferring 1WnR RAM type for array 'weight_buf'. Use bind_storage pragma to overwrite if
↪  needed.
INFO: [HLS 200-1457] Automatically inferring 1WnR RAM type for array 'in_buf'. Use bind_storage pragma to overwrite if
↪  needed.
```

In order to prepare for the following memory partition optimization, the memory type should be forced to the dual-port (T2P) RAM as Figure 2b. The result shows that the L2 pipelining with T2P RAM uses only two adders and two multipliers. The overall latency is 13885 cycles, which is a 16.4x improvement, which has already achieved the goal for this section. As the loop details in Table 4b, the initiation interval is 128 cycles, which is slightly heavy.

```
@@ -74,6 +74,7 @@
     // Iterate over output classes
     L2:
         for (int j = 0; j < CLASSES; j++) {
+#pragma HLS PIPELINE II = 1
             // Perform the dot product
             T tmp = offset_buf[j];
         L3:
```

(a) Using 1WnR RAM

```
@@ -32,6 +32,11 @@
     T in_buf[BATCH][FEAT];
     T out_buf[BATCH][CLASSES];

+#pragma HLS bind_storage variable = offset_buf type = RAM_T2P
+#pragma HLS bind_storage variable = weight_buf type = RAM_T2P
+#pragma HLS bind_storage variable = in_buf type = RAM_T2P
+#pragma HLS bind_storage variable = out_buf type = RAM_T2P
+
     // Input and output AXI stream indices
     int is_idx = 0;
     int os_idx = 0;
@@ -74,6 +79,7 @@
     // Iterate over output classes
     L2:
         for (int j = 0; j < CLASSES; j++) {
+#pragma HLS PIPELINE II = 1
             // Perform the dot product
             T tmp = offset_buf[j];
         L3:
```

(b) Using T2P RAM

Figure 2: Inserting HLS directive for L2 Pipelining

1.2.3 Pipelining the L1 (outermost) loop

The code is modified as Figure 3a. The loop details are in Table 5a. Pipelining L1 makes both L2 and L3 completely unrolled, which makes there only one loop with 8 iterations. The unrolled loop body is heavily paralleled **with 1WnR memory used**, which make use of 160 floating-point adders and 160 floating-point multipliers. The parallelism reduces the latency for one equivalent iteration from 28192 cycles to 1291 cycles. The pipelining further reduce the latency for L1 to 1402 cycles, although there are eight iterations. The overall latency is 6193 cycles, about 36.82x speedup relative to L3 pipelining and 2.22x speedup compared to L2 pipelining. Other statistics in detail can be found in Table 6.

Although L1 pipelining with 1WnR memory achieves some speedup compared to L2 pipelining, it takes 299.1 seconds to complete the whole building process, while the time used for L2 pipelining is only 62.5 seconds. At the same time, the hardware resource usage has exceeded those available on board.

As we did in Section 1.2.2, we try using dual-port memory as well. The code is modified as Figure 3b. The result shows that both overall latency and resource usage are lower than using 1WnR. That is good, but the usage of FF and LUT still exceeds the resource budget. That might be due to the 20 adders and 20 multipliers, but I failed to figure out how the computation units are used since dual-port memory is used. As in Table 5b, the initiation interval is 128 cycles.

1.3 Increasing Pipeline Parallelism by Repartitioning Memories (8 marks)

The work in this section is based on L2 pipelining with auto pipelining enabled using T2P memory. The loop details for this baseline is in Table 2.

1.3.1 Understanding the `dim` parameter

Instead of diving into the factor, which is of interest in this project, we try to choose an appropriate `dim`. The `dim` option is used to specify which dimension is partitioned. Figure 4 is a figure coming from *Vitis HLS User Guide (ug1399)*, which clearly

Table 5: Loop details for L1 pipelining

(a) L1 pipelining with 1WnR memory

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	1300	1300	130	-	-	10	no
+ LOAD_W_2	128	128	1	-	-	128	no
- LOAD_I_1	2064	2064	258	-	-	8	no
+ LOAD_I_2	256	256	2	-	-	128	no
- L1	1402	1402	1291	16	1	8	yes
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

(b) L1 pipelining with T2P memory

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	-	-	5	no
- LOAD_W_1	1300	1300	130	-	-	10	no
+ LOAD_W_2	128	128	1	-	-	128	no
- LOAD_I_1	1040	1040	130	-	-	8	no
+ LOAD_I_2	128	128	1	-	-	128	no
- L1	2186	2186	1291	128	1	8	yes
- STORE_O_1	136	136	17	-	-	8	no
+ STORE_O_2	15	15	3	-	-	5	no

shows the idea of `dim`. In our application, the loop body of L2 is pipelined, which reads 256 elements from `in_buf` and 256 elements from `weight_buf`. In order to improve parallelism, it is obvious that we should distribute the one vector of 256 elements to several blocks of memory. That is, we need to set `dim` to 2.

I have also experimented on this, where the code is modified as Figure 5. The HLS summary (Table 6) shows setting `dim` to 1 introduces more resource usage but the latency is worse. On the contrary, setting `dim` to 2 leads to introducing more floating-point adders and multipliers and achieves a lower latency. The result also supports our conclusion above.

1.3.2 Trying different `factors`

As shown in Table 6, adding the `factor` to 32 makes the utilization of DSP and LUT exceeds the resource budget.

When `factor`=8, the latency is reduced to 4911, which means 46.4x improvement, where 16 floating-point adders and 16 floating-point multipliers are used. When `factor`=16, the latency is reduced to 4279, which means 53.3x improvement, where 32 floating-point adders and 32 floating-point multipliers are used. That is exactly 16 times of that when array partition is not used. From Table 7 and Table 8, it can be seen that the initiation interval is 16 cyles when `factor`=8 and 8 cycles when `factor`=16.

1.4 Amortizing Iteration Latency with Batching (8 marks)

The result is reported in Table 6 as well as Figure 6. The usage of BRAM_18K exceeds 100% when the batch size comes to 512. When the batch size is 256, the BRAM_18K usage is 60%, which achieves the goal of 55%. The corresponding latency is 57103 cycles.

In order to evaluate its efficiency, an additional baseline with the batch size 256 and without any optimization is conducted. The result shows the baseline took 29020446 cycles, which is 508x longer than the optimized one.

It can also be evaluated by calculating the average latency per input. In baseline[1], the batch size is 8 and the latency per

[1]Partition (`dim`=2, `factor`=16)

Table 6: Performance and utilization estimates for `mmult_float`

	Profile	Latency (cycles)		Latency (ms)		Interval (cycles)		Pipeline Type
		min	max	min	max	min	max	
1.1	Baseline (AutoPipe)	85160	85160	1.236	1.236	85161	85161	none
1.1	Baseline (NoPipe)	228022	228022	2.280	2.280	228023	228023	none
1.2.1	L3 Pipelining	85286	85286	1.238	1.238	85287	85287	none
1.2.2	L2 Pipelining (1WnR)	7341	7341	0.073	0.073	7342	7342	none
1.2.2	L2 Pipelining (T2P)	13885	13885	0.139	0.139	13886	13886	none
1.2.3	L1 Pipelining (1WnR)	6193	6193	0.062	0.062	6194	6194	none
1.2.3	L1 Pipelining (T2P)	5953	5953	0.060	0.060	5954	5954	none
1.3	Baseline (L2, AutoPipe, T2P)	13759	13759	0.138	0.138	13760	13760	none
1.3.1	Partition (`dim=1, factor=2`)	13772	13772	0.138	0.138	13773	13773	none
1.3.1	Partition (`dim=2, factor=2`)	8703	8703	0.087	0.087	8704	8704	none
1.3.2	Partition (`dim=2, factor=4`)	6175	6175	0.062	0.062	6176	6176	none
1.3.2	Partition (`dim=2, factor=8`)	4911	4911	0.049	0.049	4912	4912	none
1.3.2	Partition (`dim=2, factor=16`)	4279	4279	0.043	0.043	4280	4280	none
1.3.2	Partition (`dim=2, factor=32`)	3963	3963	0.040	0.040	3964	3964	none
1.4	Amortizing (`batch=16`)	5983	5983	0.060	0.060	5984	5984	none
1.4	Amortizing (`batch=32`)	9391	9391	0.094	0.094	9392	9392	none
1.4	Amortizing (`batch=64`)	16207	16207	0.162	0.162	16208	16208	none
1.4	Amortizing (`batch=128`)	29839	29839	0.298	0.298	29840	29840	none
1.4	Amortizing (`batch=256`)	57103	57103	0.571	0.571	57104	57104	none
1.4	Amortizing (`batch=512`)	111631	111631	1.116	1.116	111632	111632	none
1.4	Baseline (NoPipe, `batch=256`)	29020446	29020446	290.000	290.000	29020447	29020447	none
1.5	Tiling (`batch=2048, tile=128`)	458106	458106	4.581	4.581	458107	458107	none
1.6	Hardware Compilation	949242	949242	9.492	9.492	949243	949243	none

	Profile	Utilization Summary					Instance	
		BRAM_18K	DSP	FF	LUT	URAM	fadd	fmul
1.1	Baseline (AutoPipe)	13 (4%)	5 (2%)	1050 (~0%)	2000 (3%)	0 (0%)	1	1
1.1	Baseline (NoPipe)	14 (5%)	5 (2%)	817 (~0%)	1635 (3%)	0 (0%)	1	1
1.2.1	L3 Pipelining	14 (5%)	5 (2%)	921 (~0%)	1713 (3%)	0 (0%)	1	1
1.2.2	L2 Pipelining (1WnR)	182 (65%)	80 (36%)	38357 (36%)	34359 (64%)	0 (0%)	16	16
1.2.2	L2 Pipelining (T2P)	16 (5%)	10 (4%)	24710 (23%)	22359 (42%)	0 (0%)	2	2
1.2.3	L1 Pipelining (1WnR)	70 (25%)	800 (363%)	415044 (390%)	243128 (457%)	0 (0%)	160	160
1.2.3	L1 Pipelining (T2P)	16 (5%)	100 (45%)	312992 (294%)	120185 (225%)	0 (0%)	20	20
1.3	Baseline (L2, AutoPipe, T2P)	16 (5%)	10 (4%)	24776 (23%)	22615 (42%)	0 (0%)	2	2
1.3.1	Partition (`dim=1, factor=2`)	16 (5%)	10 (4%)	32491 (30%)	45788 (86%)	0 (0%)	2	2
1.3.1	Partition (`dim=2, factor=2`)	16 (5%)	20 (9%)	27306 (25%)	19326 (36%)	0 (0%)	4	4
1.3.2	Partition (`dim=2, factor=4`)	20 (7%)	40 (18%)	31022 (29%)	20786 (39%)	0 (0%)	8	8
1.3.2	Partition (`dim=2, factor=8`)	36 (12%)	80 (36%)	37606 (35%)	26782 (50%)	0 (0%)	16	16
1.3.2	Partition (`dim=2, factor=16`)	68 (24%)	160 (72%)	52606 (49%)	40402 (75%)	0 (0%)	32	32
1.3.2	Partition (`dim=2, factor=32`)	132 (47%)	320 (145%)	72201 (67%)	65294 (122%)	0 (0%)	64	64
1.4	Amortizing (`batch=16`)	68 (24%)	160 (72%)	52634 (49%)	40446 (76%)	0 (0%)	32	32
1.4	Amortizing (`batch=32`)	68 (24%)	160 (72%)	52675 (49%)	40501 (76%)	0 (0%)	32	32
1.4	Amortizing (`batch=64`)	68 (24%)	160 (72%)	52708 (49%)	40547 (76%)	0 (0%)	32	32
1.4	Amortizing (`batch=128`)	102 (36%)	160 (72%)	52741 (49%)	40597 (76%)	0 (0%)	32	32
1.4	Amortizing (`batch=256`)	170 (60%)	160 (72%)	52774 (49%)	40646 (76%)	0 (0%)	32	32
1.4	Amortizing (`batch=512`)	306 (109%)	160 (72%)	52807 (49%)	40699 (76%)	0 (0%)	32	32
1.4	Baseline (NoPipe, `batch=256`)	552 (197%)	5 (2%)	940 (~0%)	1724 (3%)	0 (0%)	1	1
1.5	Tiling (`batch=2048, tile=128`)	102 (36%)	160 (72%)	52864 (49%)	40805 (76%)	0 (0%)	32	32
1.6	Hardware Compilation	78 (27%)	40 (18%)	31458 (29%)	21570 (40%)	0 (0%)	8	8

```
@@ -71,6 +71,7 @@
 // Iterate over batch elements
 L1:
     for (int i = 0; i < BATCH; i++) {
+#pragma HLS PIPELINE II = 1
     // Iterate over output classes
     L2:
         for (int j = 0; j < CLASSES; j++) {
```

(a) Using 1WnR RAM

```
@@ -32,6 +32,11 @@
     T in_buf[BATCH][FEAT];
     T out_buf[BATCH][CLASSES];

+#pragma HLS bind_storage variable = offset_buf type = RAM_T2P
+#pragma HLS bind_storage variable = weight_buf type = RAM_T2P
+#pragma HLS bind_storage variable = in_buf type = RAM_T2P
+#pragma HLS bind_storage variable = out_buf type = RAM_T2P
+
     // Input and output AXI stream indices
     int is_idx = 0;
     int os_idx = 0;
@@ -71,6 +76,7 @@
 // Iterate over batch elements
 L1:
     for (int i = 0; i < BATCH; i++) {
+#pragma HLS PIPELINE II = 1
     // Iterate over output classes
     L2:
         for (int j = 0; j < CLASSES; j++) {
```

(b) Using T2P RAM

Figure 3: Inserting HLS directive for L1 Pipelining

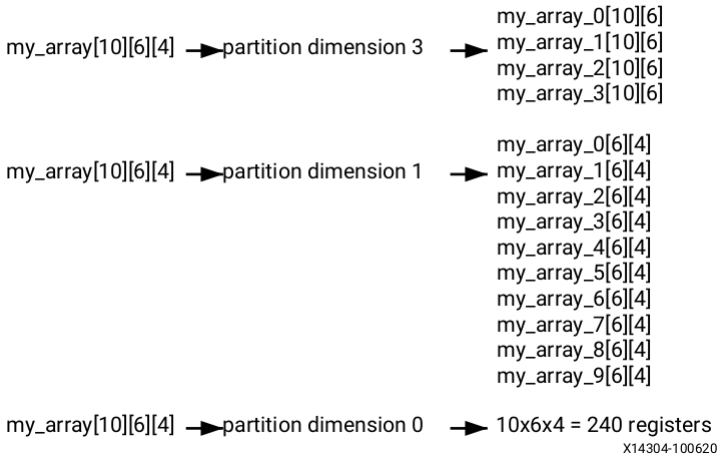

Figure 4: Partitioning Array Dimensions

input is 534.9 cycles. This design's latency per input is 223.1 cycles, a 2.4x improvement.

1.5 Extending Batch Size with Tiling (8 marks)

After tiling, the latency comes to 458106 cycles for 2048 batches, i.e., the latency per input is 223.7 cycles, which is almost the same as that in Section 1.4. Without performance loss, the BRAM usage is reduced from 60% to 36%.

```
@@ -37,6 +37,9 @@
 #pragma HLS bind_storage variable = in_buf type = RAM_T2P
 #pragma HLS bind_storage variable = out_buf type = RAM_T2P

+#pragma HLS array_partition variable = weight_buf block factor = 2 dim = 1
+#pragma HLS array_partition variable = in_buf block factor = 2 dim = 1
+
     // Input and output AXI stream indices
     int is_idx = 0;
     int os_idx = 0;
```

(a) Setting array partition with `dim=1`

```
@@ -37,6 +37,9 @@
 #pragma HLS bind_storage variable = in_buf type = RAM_T2P
 #pragma HLS bind_storage variable = out_buf type = RAM_T2P

+#pragma HLS array_partition variable = weight_buf block factor = 2 dim = 2
+#pragma HLS array_partition variable = in_buf block factor = 2 dim = 2
+
     // Input and output AXI stream indices
     int is_idx = 0;
     int os_idx = 0;
```

(b) Setting array partition with `dim=2`

Figure 5: Comparing different setting of `dim`

Table 7: Loop details for partition with `dim=2` `factor=8`

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	1	1	5	yes
- LOAD_W_1_LOAD_W_2	1280	1280	1	1	1	1280	yes
- LOAD_I_1_LOAD_I_2	1024	1024	1	1	1	1024	yes
- L1_L2	2550	2550	1287	16	1	80	yes
- STORE_O_1_STORE_O_2	42	42	4	1	1	40	yes

Table 8: Loop details for partition with `dim=2` `factor=16`

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- LOAD_OFF_1	5	5	1	1	1	5	yes
- LOAD_W_1_LOAD_W_2	1280	1280	1	1	1	1280	yes
- LOAD_I_1_LOAD_I_2	1024	1024	1	1	1	1024	yes
- L1_L2	1918	1918	1287	8	1	80	yes
- STORE_O_1_STORE_O_2	42	42	4	1	1	40	yes

1.6 Hardware compilation and FPGA testing on the PYNQ (8 marks)

In this section, we met some problems.

The first problem that I met is when exporting the IP, which has been sovled in: Vivado fails to export IPs with the error message "Bad lexical cast: source type value could not be interpreted as target".

The second problem occurs when validating the system design, which reports a critical warning as Figure 7. As this signla shall be provided by Vitis HLS when implementing the AXI Stream interface, I changed the signature of the top function from C array to `hls::stream`. That fixes the problem.

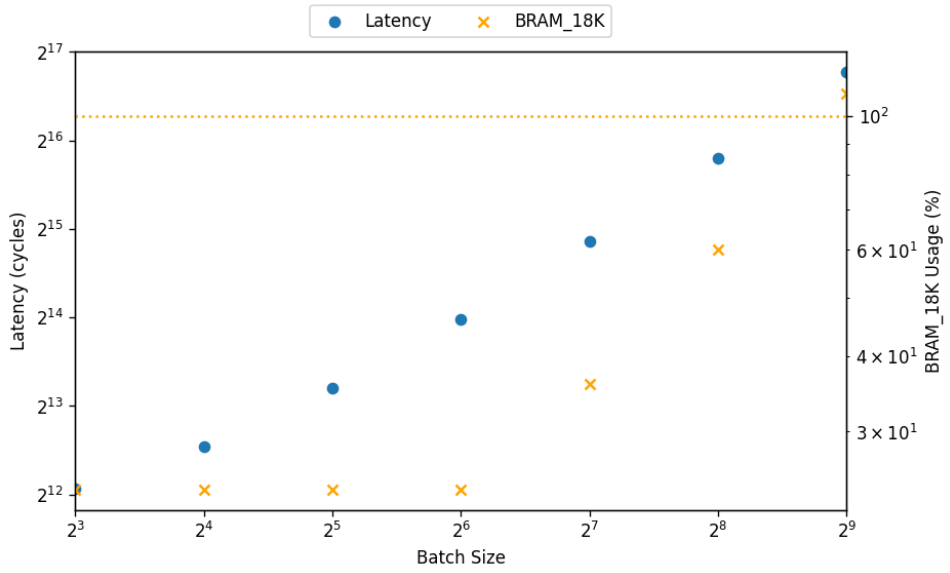

Figure 6: Latency and BRAM_18K usage

```
[xilinx.com:ip:axi_dma:7.1-9] /axi_dma_1
############################################################
Interface connected to S_AXIS_S2MM does not have TLAST port
############################################################
```

Figure 7: Critical Warning from validating the design

After building, move the bitstream files to the board.

```
$ scp `find -name "*.bit"` xilinx@<pynq-ip-addr>:~/classifier.bit
$ scp `find -name "*.hwh"` xilinx@<pynq-ip-addr>:~/classifier.hwh
```

According to the different versions of PYNQ, I have to modify the notebook as well. The new version is attached in the submission.

The measured misclassification rate is 13.04% for both FPGA and CPU, which indicates that our implementation is correct. The running time on FPGA is around 16.16ms, while 78.04ms on CPU. That is, speedup on FPGA is 4.83x, a little bit smaller than required. This may due to the optimization between different versions of python and numpy.

2 Fixed-Point Optimizations

2.1 Dataset Preparation

I did some some work to adapt `mnist.py` to Python 3. In order to get an appropriate scale, I did a search and got Figure 8, which shows the validation accuracy meets the requirement when the scale is around 2^{16}. The fixed-point validation accuracy reported by `mnist.py` is as Figure 9, where the validation error with fixed-point number is 18.98%.

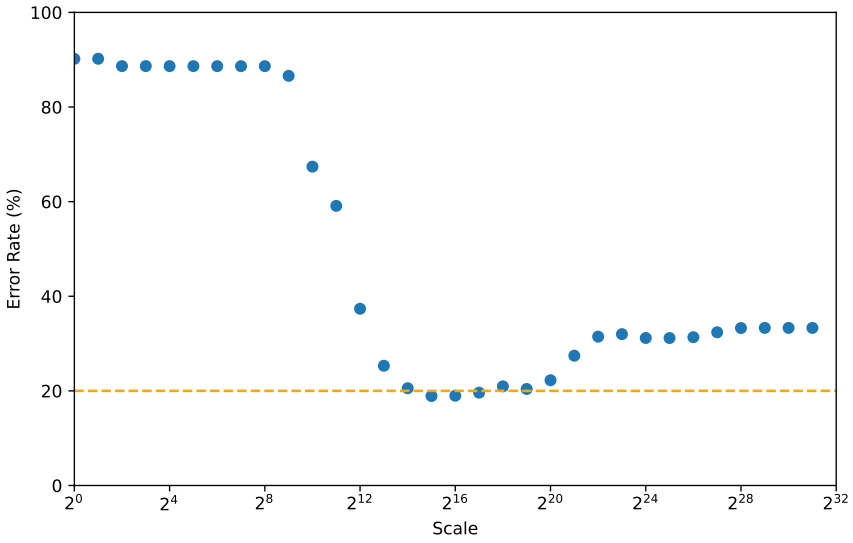

Figure 8: Misclassifications Rate with fixed point numbers under different scales

```
$ python3 mnist.py
Min/Max of coefficient values [-0.49227501415123354, 0.3931926418998524]
Min/Max of intersect values [-0.12369156945342877, 0.2528140743366973]
Misclassifications (float) = 14.05%
Misclassifications (fixed) = 18.98%
```

Figure 9: The output of `mnist.py`

2.2 Hardware Design

2.2.1 Timing

The overall latency, as reported in Table 9, is 386378 cycles for 8192 batches, which means 47.17 cycles for each input on average. It is a 604.25x speedup over our baseline without auto pipelining. The pipelined loop has an initiation interval of 1 cycle achieved, as reported in Table 10.

Table 9: Performance Estimates

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
386378	386378	3.864 ms	3.864 ms	386379	386379	none

Table 10: Loop detail of the pipelined loop

		L1_L2
Latency (cycles)	min	1287
Latency (cycles)	max	1287
Iteration Latency		9
Initiation Interval	achieved	1
Initiation Interval	target	1
Trip Count		1280
Trip Count		1280
Pipelined		yes

Table 11: Loop Latency of load, compute and store

Loop Name	Latency (cycles) min	max
LOAD_INPUT_VITIS_LOOP_73_4	4096	4096
L1_L2	1287	1287
STORE_OUTPUT_VITIS_LOOP_106_6	642	642

2.2.2 Device Utilization

The overall device utilization is reported as Table 12. We have 128 multipliers implemented by LUT and 129 multiply-accumulate operator implemented on DSP.

Table 12: Utilization Estimates Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	129	-	-	-
Expression	-	-	0	3967	-
FIFO	-	-	-	-	-
Instance	0	0	36	5247	-
Memory	260	-	4160	517	-
Multiplexer	-	-	-	7513	-
Register	-	-	5633	128	-
Total	260	129	9829	17372	0
Available	280	220	106400	53200	0
Utilization (%)	92	58	9	32	0

2.3 Evaluation

The measured system speedup over the fixed-point CPU implementation is 34.94x. The measured misclassification rate on the 8k MNIST test sample is 19.96% both on FPGA and on CPU.

2.4 Reflection

The width of TDATA signal in AXI Stream interface is 64bit. Each input data has 256 bytes, i.e., 32 words, and a tile of 128 inputs has 4096 words. Our design, as reported in Table 11, took exactly 4096 cycles to load the 4096 words, but the following computation took only another 1287 cycles. Thus I believe the design is memory-bandwidth limited. That is, if we can load more data at a single cycle, we can achieve a higher overall throughput.

3 Open-ended Design Optimization

3.1 Hardware Compatible

3.1.1 Reducing the Input Dimension

The size of input is determined by the dimension of inputs (originally 16×16) and the depth of each pixel (8-bit in the previous design). A search is done by trying different pair of input dimension and depth and calculating the prediction accuracy using linear regression model, which is actually a lower bound we can achieve. The result is as Figure 10.

It is shown that

When the input size is shrunk to 8×8, the misclassifications rate is 15.96%, which is slightly worse than our target (over 85% in accuracy). However, this is a desirable configuration, which means we can fit an input in precisely 8 bytes. Thus I believe it is worth sacrificing some accuracy here and getting it back using a more aggressive model.

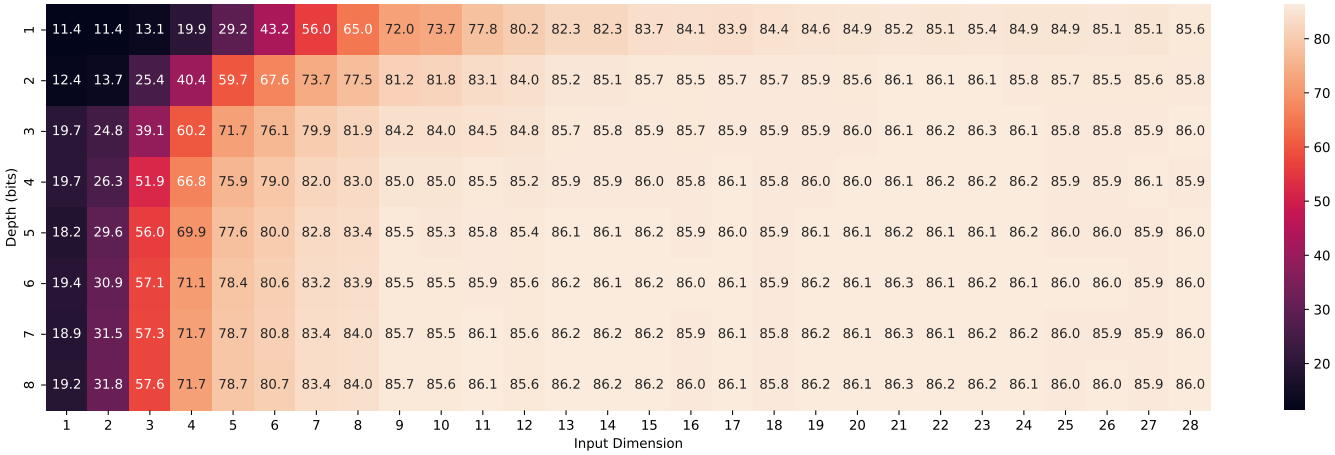

Figure 10: Accuracy of prediction using floating point numbers under different input size

3.1.2 Improve Prediction Accuracy

The accuracy is improve by using a more complex model, which includes a ReLU activated input layer (64×16), which is unbiased, and a biased output layer (16×10). Using ReLU with an unbiased input layer, we can refrain from floating point calculation while keep the linear property.

$$\mathbf{O} = \mathrm{ReLU}(\mathbf{I} \cdot \mathbf{W_1}) \cdot \mathbf{W_2} + \mathbf{b}$$

To get a better training result, we add a Softmax layer after the model and use train towards minimized cross entropy loss. This layer can be removed when predicting on FPGA, which will not affect the final result.

3.1.3 Evaluation

The design latency is 200699 cycles.

It took 4.54 ms to generate the prediction while the CPU used 169.64 ms, which means the FPGA solution achieves a 37.36x speedup against the CPU solution. The prediction accuracy is 88.18% on both FPGA and CPU.

3.2 Enhancing a Single AXI Transfer

In order to accomodate more data in a single AXI transfer[2], we need to increase the `TDATA` width.

The width in the default design is 64 bits, which took 32 cycles to transfer a single input of 256 bytes. By widening the `TDATA` width to 256 bits, it takes only 8 cycles to transfer the 256 bytes (Table 13).

The overall latency of our design is 159646 cycles. Its utilization is as Table 14.

[2] Stream terms

Table 13: LOAD_INPUT latency under different TDATA width

TDATA width (bits)	Latency (cycles)	Iteration Latency	Initiation Interval (achieved)	Trip Count	Pipelined
64	4096	32	32	128	yes
128	2048	16	16	128	yes
256	1024	8	8	128	yes

Table 14: Utilization Report with 256-bit AXI Stream

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	192	-	-	-
Expression	-	-	0	4703	-
FIFO	-	-	-	-	-
Instance	0	0	36	1352	-
Memory	202	-	32	3	-
Multiplexer	-	-	-	3577	-
Register	-	-	6552	128	-
Total	202	192	6620	9763	0
Available	280	220	106400	53200	0
Utilization (%)	72	87	6	18	0

Since we modified the AXI data width, the data width of DMA IP needs to be fixed manually as Figure 11.

The evaluation shows that FPGA took 3.85 ms to produce the result, while CPU took 169.33 ms as well. That is a 44x speedup which is better but not that better as we expected. It is because the maximum AXI data width of Zynq 7000 processor is 64 bit, which limited the bandwidth.

The accuracy is still 88.18%.

Figure 11: Adjust AXI data width by re-customizing AXI-DMA IP