CS5222 Project 2
Custom Acceleration with FPGAs

Shen Jiamin
A0209166A
shen_jiamin@u.nus.edu

March 6, 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 report generated by HLS (as in Table 1) 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 3. The new loop details is as Table 2. 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.

1.2 Pipelining in HLS (8 marks)

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

Table 1: Loop details for 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

Table 2: Loop details for 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 3: 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	13759	13759	0.138	0.138	13760	13760	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

	Profile	Utilization Summary					Instance	
		BRAM	DSP	FF	LUT	URAM	fadd	fmul
	Available	280	220	106400	53200	0		
1.1	Baseline (AutoPipe)	13	5	1050	2000	0	1	1
1.1	Baseline (NoPipe)	14	5	817	1635	0	1	1
1.2.1	L3 Pipelining	14	5	921	1713	0	1	1
1.2.2	L2 Pipelining	13	10	24840	22620	0	2	2
1.2.3	L1 Pipelining (1WnR)	70	800	415044	243128	0	160	160
1.2.3	L1 Pipelining (T2P)	16	100	312992	120185	0	20	20

1. "{L1, L2, L3} Pipelining" are based on Baseline (NoPipe).
2. "L2/Partition" are based on L2 Pipelining.

1.2.1 Pipelining the L3 (innermost) loop

The code is modified as Figure 1. As reported in Table 4, 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 3.

```
@@ -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

Table 4: 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

1.2.2 Pipelining the L2 loop

The code is modified as Figure 2. As reported in Table 5, pipelining the loop body of L2 unrolls L3 and flattens L1. The design introduced some parallelism, which uses two adders and two multipliers. The overall latency is reduced to 13759 cycles, which is about 16.57x speedup relative to the baseline, which has already achieved the goal for this section. The initiation interval has increased to 128 cycles, which is pretty heavy. Other statistics in detail can be found in Table 3.

```
@@ -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:
```

Figure 2: Inserting HLS directive for L2 Pipelining

Table 5: Loop details for L2 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	11398	11398	1287	128	1	80	yes
- STORE_O_1_STORE_O_2	42	42	4	1	1	40	yes

1.2.3 Pipelining the L1 (outermost) loop

The code is modified as Figure 3. The loop details are in Table 6. 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 3.

```
@@ -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++) {
```

Figure 3: Inserting HLS directive for L1 Pipelining

Table 6: Loop details for 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

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 for L2 pipelining is only 62.5 seconds. At the same time, the hardware resource usage has exceeded those available on board.[1]

An important observation is that the design can utilise 160 adders and 160 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 short for Multi-Ported Memory using Replication, and this type of memory has a single write port and multiple concurrent read ports.

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

[1]DSP (363%), FF (390%) , LUT (457%)

In order to prepare for the following memory partition optimization, the memory type should be forced to the dual-port (T2P) RAM as Figure 4. 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 7, the initiation interval is 128 cycles.

```
@@ -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++) {
```

Figure 4: Setting memory type to true dual-port RAM with L1 Pipelining.

Table 7: Loop details for 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

2 Part 2: Fixed-Point Optimizations (30 marks)

1. the fixed-point validation accuracy reported by mnist.py after you've tweaked the SCALE factor.
2. the design latency in cycles
3. the overall device utilization (as Total per Resource).
4. your measured system speedup over the fixed-point CPU implementation
5. your measured classification accuracy on the 8k MNIST test sample
6. how many multipliers are instantiated in your desing?

7. report the initiation interval of the matrix multiplication loop that you pipelined
8. given the number of multipliers in your design and input throughput via the AXI port, is the design bandwidth- or compute-limited?

3 Part 3: Open-ended design optimization (30 marks)

Vitis High-Level Synthesis User Guide