

4190.301A Hardware System Design

Spring 2020

Hardware System Design

PROJV0_final Report

Kim Bochang

2014-16757

1. <PROJV0_final> Introduce

project_v0 final의 목적은 우리가 project_v0 mid에서 구현했던 matrix * vector multiplier를 FPGA에 올리는 것이다.

2. Implementation

제출된 zip파일이 여러가지인데, 각각은 다음과 같다.

project_v0 : MV array가 있는 폴더. 동봉된 waveform을 뽑아내는 코드가 여기에만 들어있어서 첨부함.

proj_ip : custom ip가 구현된 폴더. 핵심.

project_v0_fpga : 실제로 bitstream을 만드는데 사용된 프로젝트.

결과로 나온 bitstream을 각각 8x8.bit, 16x16.bit, 32x32.bit, 64x64.bit으로 첨부함.

16x16.bit : lab11의 test code를 (verify.cpp) 돌리는데 사용된 bitstream. (참고용)

8x8.bit, 32x32.bit : 8x8, 32x32 matrix-vector multiplication을 위해 사용된 bitstream(참고용)

64x64.bit 은 benchmark.sh를 돌리는데 사용된 bitstream. (최종 제출물)

2.1 PE controller (project_v0\project_v0.srcs\sources_1\new\my_pe_controller.v)

혹은 (proj_ip\lab10_ip_repo\myip_1.0\src\my_pe_controller.v)

project_v0_mid에서 바뀐 점이 한가지 있어서 이에 대해서만 설명한다.

그 외의 구현은 project_v0 mid 와 같으므로, 자세한 설명은 생략하도록 하겠다.

(동봉된 project_v0_mid 보고서, lab6 보고서에 있음)

project_v0 mid에서는 matrix가 address의 처음 부분 (0~번지), vector가 address의 뒷부분 (matrix의 뒷부분)에 존재한다 가정하고 구현을 했는데, 실제 C++코드에서는 BRAM의 address의 처음 부분에(0~번지) vector를, vector 뒷부분에 matrix를 저장하므로, 이러한 부분을 바꾸어 주어야 한다.

다음과 같이 rdaddr에 전달되는 주소를 바꾸어 주어 간편하게 해결할 수 있다.

```
// LOAD part.
assign temp_rdaddr = (state == S_LOAD) ? counter[BRAM_ADDR_WIDTH-1:1] : 'd0;

assign rdaddr = (temp_rdaddr[MATRIX_NUM + VECTOR_NUM]) ? temp_rdaddr[MATRIX_NUM + VECTOR_NUM - 1:0] : (temp_rdaddr + (1 << VECTOR_NUM));
// It is needed because my module get 0-MATRIX_SIZE * VECTOR_SIZE - 1 as matrix address. (2*VECTOR_NUM = VECTOR_SIZE, 2*MATRIX_NUM = MATRIX_SIZE (number of matrix row)
// and MATRIX_SIZE * VECTOR_SIZE ~ MATRIX_SIZE * VECTOR_SIZE + VECTOR_SIZE - 1 as vector address
// so we need convert to 0 ~ VECTOR_SIZE-1 as vector address (remove MSB)
// and VECTOR_SIZE ~ MATRIX_SIZE * VECTOR_SIZE + VECTOR_SIZE - 1 as matrix address. (plus VECTOR_SIZE)
```

(project_v0\project_v0.srcs\sources_1\new\my_pe_controller.v 코드의 일부)

이 코드가 하는 역할을 예로 들어 설명하면 다음과 같다.

예를 들어, VECTOR_NUM = 2, MATRIX_NUM = 3이면, vector 크기는 $2^2 = 4$, matrix의 row 개수는 $2^3 = 8$ 에서, 총 8x4 matrix와 4x1 vector의 multiplication을 진행하게 되는데, 때문에 총 36개의 데이터가 필요하고, 주소로는 000000~100011 까지의 데이터가 필요하다. (2진수 주소)

이때 project_v0_mid에서 작성한 버전은 벡터 원소의 주소로 100000~100011을 가정하고,

matrix 원소의 주소로 000000~011111을 가정하고 구현한 버전인데,

실제로는 벡터 원소의 주소는 000000~000011이고, matrix 원소의 주소는 000100~100011에 저장이 되므로,

기존 주소(project_v0_mid에서 작성한 버전)를 다음과 같이 바꿔주면 실제 주소가 된다.

기존주소가 벡터의 주소인 경우, 즉, $addr \geq 100000$ 인 경우, 앞의 1만 제거해주면 간단히 실제 벡터의 주소로 변한다.

기존주소가 matrix의 주소인 경우, 즉, $addr < 100000$ 인 경우, 이 addr에 벡터의 크기만 더해주면 (여기서는 000100) 실제 matrix의 주소로 변하므로,

$addr \geq 100000 \rightarrow$ 앞의 1을 제거, $addr < 100000 \rightarrow$ 벡터의 크기만큼을 더함 (+ 000100)

과 같은 과정을 통해 내부 로직을 건드리지 않고 간편하게 address를 바꿀 수 있다.

Name	Value	93,477,000 ps	93,477,002 ps	93,477,004 ps	93,477,006 ps	93,477,008 ps	93,477,010 ps	93,477,012
aresetn	1							
start	0							
> din[31:0]	40000000				40000000			
done	0							
> rdaddr[14:0]	0010				0010			
> wrdata_u_d[51:0]	44bb00004				44bb0000455d000045548000454c000045438000453b00004528000452a00004521800045190000451080004508000044ff000044ee000044dd000044cc0000			
> wrdata[0:15][31:0]	1632.0,1768.0,1904.0,2040.0,2176.0,2312.0,2448.0,2584.0,2720.0,2856.0,2992.0,3128.0,3264.0,3400.0,3536.0,1496.0							
> [0][31:0]	44cc0000				44cc0000			
> [1][31:0]	44dd0000				44dd0000			
> [2][31:0]	44ee0000				44ee0000			
> [3][31:0]	44ff0000				44ff0000			
> [4][31:0]	45080000				45080000			
> [5][31:0]	45108000				45108000			
> [6][31:0]	45190000				45190000			
> [7][31:0]	45218000				45218000			
> [8][31:0]	452a0000				452a0000			
> [9][31:0]	45328000				45328000			
> [10][31:0]	453b0000				453b0000			
> [11][31:0]	45438000				45438000			
> [12][31:0]	454c0000				454c0000			
> [13][31:0]	45548000				45548000			

project_v0 내부의 tb_my_pe_controller.v 를 실행한 결과는 위와 같이 바뀌었는데,

이를 통해 우리가 원하는 대로 연산이 잘 일어남을 알 수 있다.

(전의 결과에서, vector와 matrix의 1번째 row가 같았 으므로, 위와 같이 주소를 바꿔주면 matrix의 i+1 번째 row와 1 번째 row를 각각 i번째 row와 16번째 row로 사용한 결과가 나오고, 위와 같이 의도한 값이 나옴을 알 수 있다.)

2.2 custom module-outside

(proj_ipWlab10_ip_repoWmyip_1.0WhdlWmyip_v1_0_S00.v)

lab10에서 제공한 shifter가 들어있는 custom ip를 그대로 사용하였다.

변화된 점은, 후에 사용할 parameter만 추가해준 것을 빼면 없다.

```
6      // Users to add parameters here
7      parameter integer C_MOO_BRAM_ADDR_WIDTH = 32,
8      parameter integer C_MOO_BRAM_DATA_WIDTH = 32,
9      parameter integer C_MOO_BRAM_WE_WIDTH = 4,
10     parameter integer C_MOO_MATRIX_NUM = 4, // 2**4 = 16
11     parameter integer C_MOO_VECTOR_NUM = 4, // 2**4 = 16
12     parameter integer C_MOO_G_BUF_SIZE = 6, // 64 entry
13     parameter integer C_MOO_L_RAM_SIZE = 6, // 64 entry
14     // User parameters ends
15     // Do not modify the parameters beyond this line

59     myip_v1_0_S00_AXI # (
60         .BRAM_ADDR_WIDTH(C_MOO_BRAM_ADDR_WIDTH),
61         .BRAM_DATA_WIDTH(C_MOO_BRAM_DATA_WIDTH),
62         .BRAM_WE_WIDTH(C_MOO_BRAM_WE_WIDTH),
63         .MATRIX_NUM(C_MOO_MATRIX_NUM),
64         .VECTOR_NUM(C_MOO_VECTOR_NUM),
65         .G_BUF_SIZE(C_MOO_G_BUF_SIZE),
66         .L_RAM_SIZE(C_MOO_L_RAM_SIZE),
67         .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
68         .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
69     ) myip_v1_0_S00_AXI_inst (
```

각 paramter가 무엇인지는 아래 서술한다.

2.3 custom module-inside

(proj_ipWlab10_ip_repoWmyip_1.0WhdlWmyip_v1_0_S00_AXI.v)

lab10에서 제공한 shifter가 들어있는 custom ip를 약간 바꿔서 사용하였다.

```
5 : // Users to add parameters here
6 : parameter integer BRAM_ADDR_WIDTH = 32,
7 : parameter integer BRAM_DATA_WIDTH = 32,
8 : parameter integer BRAM_WE_WIDTH = 4,
9 : parameter integer MATRIX_NUM = 4, // 2**4 = 16
10 : parameter integer VECTOR_NUM = 4, // 2**4 = 16
11 : parameter integer G_BUF_SIZE = 6, // 64 entry
12 : parameter integer L_RAM_SIZE = 6, // 64 entry
13 : // User parameters ends
```

먼저, 원래 있던 parameter에서 다음과 같은 4개의 parameter를 추가하였다.

MATRIX_NUM : $2^{\text{MATRIX_NUM}}$ = matrix row의 개수.

VECTOR_NUM : $2^{\text{VECTOR_NUM}}$ = vector의 크기 (내부에 들어가는 원소의 개수 = matrix col 개수)

G_BUF_SIZE : pe controller에 들어갈 global buffer의 size. ($2^{\text{G_BUF_SIZE}}$)

L_LAM_SIZE : pe 내부의 local buffer의 size. ($2^{\text{L_LAM_SIZE}}$)

위 parameter는 내부 MY_PE_controller에 사용된다.

내부 전체적인 로직은 다음과 같다.

크게 FSM과 카운터로 구성되며, FSM은 STATE들로 이루어져있다.

S_IDLE : 계산 요청이 들어오기 전의 상태.

S_WORK : 계산 요청을 받아 matrix-vector multiplication이 진행되고 있는 상태

S_WRITE : matrix-vector multiplication이 끝난 다음, 결과를 BRAM에 쓰는 상태

카운터는 write state에서, 결과를 BRAM에 저장하는데 쓰이게 된다.

모두 lab10에서 제공된 shifter의 로직을 참고해서 구현하였다.

FSM에 해당하는 코드는 다음과 같다.

```

425 //FSM: IDLE -> WORK -> WRITE -> IDLE
426 reg [1:0] state;
427 localparam S_IDLE = 2'd0;
428 localparam S_WORK = 2'd1;
429 localparam S_DONE = 2'd2; // don't used.
430 localparam S_WRITE = 2'd3;
431
432 wire magic_code = (slv_reg0 == 32'h5555);
433
434 always @( posedge S_AXI_ACLK )
435 begin
436     if ( S_AXI_ARESETN == 1'b0 )
437         state <= S_IDLE;
438     else
439         case (state)
440             S_IDLE: state <= (magic_code)? S_WORK : S_IDLE;
441             S_WORK: state <= (done)? S_WRITE : S_WORK;
442             S_WRITE: state <= (run_complete)? S_IDLE: S_WRITE;
443             default : state <= S_IDLE;
444         endcase
445     end
446

```

먼저, c++ 코드에서 계산 요청을 보내면 magic_code가 1이된다.

따라서, magic_code를 받아 S_WORK상태로 들어가고, my_pe_controller의 작동이 모두 끝나면 done이 1이 되므로, 이를 받아 WRITE state로 들어간다.

WRITE state에서는 my_pe_controller의 결과를 BRAM에 저장하며, 저장이 끝나면 다시 IDLE state로 돌아가게 된다.

참고로, 여기서 run_complete가 1이되면, C++ 코드에서 slv_reg0의 입력에 해당되는 address에 0을 저장하게 되어.

아래와 같이 loop 조건으로 걸어놨던 부분이 풀리게 된다.

```

const float *__attribute__((optimize("O0"))) FPGA::blockMV()
{
    num_block_call_ += 1;

    // fpga version
    *output_ = 0x5555;
    while (*output_ == 0x5555)
        ;

    return data_;
}

```

<hsd_lab07\src\fpga_api.cpp>

```
// Add user logic here
genvar j;
wire pe_clk;
clk_wiz_1 u_clk(.clk_out1(pe_clk), .clk_in1(S_AXI_ACLK)); // make clock stable
clk_wiz_2 u_clk_180 (.clk_out1(BRAM_CLK), .clk_in1(S_AXI_ACLK)); // make clock stable, and negate clk
assign BRAM_EN = 1'b1;
assign BRAM_RST = 1'b0;
wire bram_write;
assign BRAM_WE = (bram_write)? 4'hF : 4'h0;
wire [31:0] bram_rd_addr;
wire [31:0] bram_wr_addr;

// for my custom pe
wire [(2**MATRIX_NUM)*32 - 1:0] wrdata_unpacked;
wire [31:0] wrdata[0:2**MATRIX_NUM - 1];
wire start;
wire [BRAM_DATA_WIDTH-1:0] din = BRAM_RDDATA;
wire [BRAM_ADDR_WIDTH-3:0] rdaddr;
wire done;
wire aresetn;
reg[31:0] wrdata_bram;
```

처음의 이 부분은 필요한 와이어와 레지스터를 선언하는 부분이다.

윗부분은 lab10의 shifter와 용법이 매우 유사하고,

아래부분은 my_pe_controller에 필요한 부분들을 지정한것이다.

여기서, 사용한 clk_wiz_1, 2의 역할은, S_AXI_CLOCK을 받아서, 안정화된 클럭을 pe_clk와 BRAM_CLK에 제공하는 것이다.

이때, clk_wiz_1는 50MHZ의 clk을 받아 안정화된 50MHZ의 클럭을 내보내기만 하지만,

clk_wiz_2는 50MHZ의 clk을 받아 안정화 시킨뒤, 50MHZ의 ~clk의 값을 내보낸다.

Output Clock	Port Name	Output Freq (MHz)	Phase (degrees)	Duty Cycle (%)
		Requested	Actual	Requested
<input checked="" type="checkbox"/> clk_out1	clk_out1	50	180	50.000
<input type="checkbox"/> clk_out2	clk_out2	100.000	N/A	50.000
<input type="checkbox"/> clk_out3	clk_out3	100.000	N/A	50.000
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	50.000
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	50.000
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	50.000
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	50.000

위와 같이, phase를 180으로 주어 설정하였다.

위와 같이, BRAM에 negate된 clk을 공급하는 이유는, BRAM은 write와 read에 1사이클 ,2사이클이 걸리는데, 이 말을 풀어 써보면 address가 입력된 상태일 때, write요청을 받았을때 다음 clock edge에서 즉시 write가 되고, address가 입력된 상태일 때, read 요청을 받았을 때 다다음 clock edge에서 read된 결과를 출력하게 된다.

그런데 my_pe_controller에서 외부로 데이터를 요청하기 위해 보내는 address는 기존 clk의 posegde에 작동하는 counter의 값에 의존해서 만들어지므로, BRAM에 같은 clk를 물려 주게 되면 clk의 posedge 타이밍 때문에 posedge 전의 counter value가 사용될 것인지, 혹은 posedge 후의 counter value가 사용될 것인지 애매해지게 되고, 내부 회로 delay를 감안하면 문제가 생길 여지가 생긴다.

따라서, BRAM과 my_pe_controller의 작동 타이밍을 다르게 만들어주면 이러한 문제가 사라지므로, 의도한 대로 확실하게 작동을 보장하기 위해 BRAM과 my_pe_controller가 사용하는 clock을 negate해주는 것이다.

```
447 //for counter
448 reg [31:0] counter;
449 reg [31:0] counter_delay1;
450 wire counter_reset = (S_AXI_ARESETN == 1'b0) || (state == S_IDLE);
451 wire counter_enable = (state == S_WRITE);
452
453 always@(posedge S_AXI_ACLK)
454 begin
455     counter_delay1 <= counter;
456 end
457
458 always @( posedge S_AXI_ACLK )
459 begin
460     if ( counter_reset )
461         counter <= 2**MATRIX_NUM - 2'd1;
462     else
463     begin
464         if (counter_enable)
465             counter <= counter - 2'd1;
466         else
467             counter <= counter;
468     end
469 end
```

이는 카운터 부분이다. down-counter를 사용하였고, counter가 reset되어야 할 때 counter를 벡터의 크기 -1만큼으로 리셋해주고, 이 카운터값을 계산 결과를 저장할 bram의 address로 사용하게 된다.


```

472 //for address part
473 assign bram_rd_addr = rdaddr << 2; // 4byte (floating point) shift needed.
474 assign bram_wr_addr = counter_delay1 << 2; // 4byte shift needed. it used delayed value because
475 assign bram_write = (state == S_WRITE)? 1'b1 : 1'b0;
476 assign BRAM_ADDR = (bram_write) ? bram_wr_addr : bram_rd_addr;
477
478 assign BRAM_WDATA = wrdata_bram;
479 assign start = (state == S_WORK);
480 assign run_complete = (counter_delay1 == 0);
481 assign aresetn = (S_AXI_ARESETN && ~(state == S_IDLE)); // if S_AXI_ARESETN is 0 or current sta
482

```

우리의 my_pe_controller에서, 데이터를 요청하는 address는 rdaddr인데, 이 rdaddr은 데이터의 크기를 고려하지 않은 address이므로, 4byte float data를 받아오려면 shift 연산을 통해 실제 address로 만들어줄 필요가 있다.

또한, aresetn (pe_controller에 들어가는 reset 신호)를 위와 같이 주어,

외부에서 리셋 신호가 들어올 때나, IDLE 상태일 때 내부 pe controller를 초기화 하도록 하였다.

```

483 // wrdata_bram is data should write to bram.
484 always @( posedge S_AXI_ACLK )
485 begin
486     if (S_AXI_ARESETN == 1'b0 || state == S_IDLE)
487         wrdata_bram <= 32'd0;
488     else
489         begin
490             if (bram_write)
491                 wrdata_bram <= wrdata[counter];
492             else
493                 wrdata_bram <= wrdata_bram;
494         end
495     end
496
497     for(j=0; j<2**MATRIX_NUM; j=j+1) begin
498         assign wrdata[j] = wrdata_unpacked[j*32 +: 32] ;
499     end
500 //packing

```

연산이 끝난 후, 데이터를 BRAM에 저장하기 위해서는 각 data를 해당하는 address에 저장해줄 필요가 있으므로, 위 코드를 통해 계산 결과를 해당 bram의 address에 저장한다.

이때, counter의 번지에 해당하는 계산결과는 wrdata_bram이라는 레지스터를 거쳐 bram에 저장되고, 이렇게 레지스터에 들어가는 동안 1사이클 딜레이가 생기므로, BRAM의 address도 counter를 바로 이용하는 것이 아니라, counter를 한 사이클 딜레이를 주어서 활용할 필요가 있다. 이에 대한 코드는 카운터 부분에 있다.

```

504 | my_pe_controller #(G_BUF_SIZE(G_BUF_SIZE), L_RAM_SIZE(L_RAM_SIZE), VECTOR_NUM(VECTOR_NUM), MATRIX_NUM(MATRIX_NUM), BRAM_ADDR_WII
505 |     .start(start),
506 |     .aclk(pe_clk),
507 |     .aresetn(aresetn),
508 |     .rddata(din),
509 |     .rdaddr(rdaddr),
510 |     .wdata_unpacked(wdata_unpacked),
511 |     .done(done)
512 | );
513 | // my code end

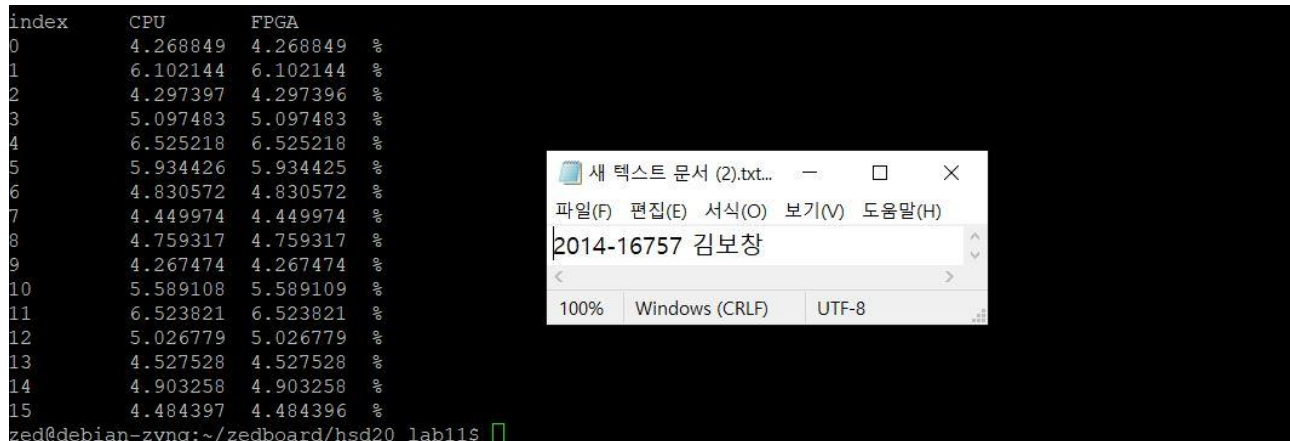
```

마지막으로, pe_controller 모듈에 대한 선언이다.

3. Result & Discussion

동봉된 benchmark.mp4 파일을 실행하면, zedboard 전원을 키는것 ~ benchmark.sh를 실행하기까지의 결과가 나와있다. 이때 사용한 bitstream file은 64x64.bit 파일이다. (64 x 64 multiplier)

아래의 결과는 lab11에서 배포한 test code에 대한 결과이다. 이때 사용한 bitstream file은 16x16.bit이다.



index	CPU	FPGA	
0	4.268849	4.268849	응
1	6.102144	6.102144	응
2	4.297397	4.297396	응
3	5.097483	5.097483	응
4	6.525218	6.525218	응
5	5.934426	5.934425	응
6	4.830572	4.830572	응
7	4.449974	4.449974	응
8	4.759317	4.759317	응
9	4.267474	4.267474	응
10	5.589108	5.589109	응
11	6.523821	6.523821	응
12	5.026779	5.026779	응
13	4.527528	4.527528	응
14	4.903258	4.903258	응
15	4.484397	4.484396	응

zed@debian-zyng:~/zedboard/hsd20 lab11\$

잘 작동하는 것을 볼 수 있다.

이제, matrix-vector multiplier 크기에 따른 benchmark에 대한 결과를 보자.

benchmark.sh의 내용을 m_size, v_size를 바꿔서 사용하였다.

즉, mlp, cnn에 대해 각가 m_size와 v_size를 바꿔가면서, 100개의 데이터에 대해 cpu, fpga 옵션을 주어 계산한 결과를 촬영하였다.

스크린샷에 내용이 모두 나와있으므로 개별 결과에 대한 설명은 생략한다.


3.1 8 x 8 (m_size = 8, v_size = 8, 8x8.bit)

```
zed@debian-zynq:~/hsd20_lab07$ bash ./benchmark8.sh
[*] Arguments: Namespace(m_size=8, network='mlp', num_test_images=100, run_type='cpu', v_size=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 37500,
 'm_size': 8,
 'total_image': 100,
 'total_time': 7.240315198898315,
 'v_size': 8}

=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=8, network='mlp', num_test_images=100, run_type='fpga', v_size=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 37500,
 'm_size': 8,
 'total_image': 100,
 'total_time': 63.283823013305664,
 'v_size': 8}

=> Accuracy should be 0.97
```




```
[*] Arguments: Namespace(m_size=8, network='cnn', num_test_images=100, run_type='cpu', v_size=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 3388,
 'm_size': 8,
 'total_image': 100,
 'total_time': 0.8130879402160645,
 'v_size': 8}

=> Accuracy should be 1.0

[*] Arguments: Namespace(m_size=8, network='cnn', num_test_images=100, run_type='fpga', v_size=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 3388,
 'm_size': 8,
 'total_image': 100,
 'total_time': 5.323554992675781,
 'v_size': 8}

=> Accuracy should be 1.0
zed@debian-zynq:~/hsd20_lab07$
```




3.2 16 x 16 (m_size = 16, v_size = 16, 16x16.bit)

```
[sudo] password for zed:
[*] Arguments: Namespace(m_size=16, network='mlp', num_test_images=100, run_type='cpu', v_size=16)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 9375,
 'm_size': 16,
 'total_image': 100,
 'total_time': 6.5124780000000015,
 'v_size': 16}

=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=16, network='mlp', num_test_images=100, run_type='fpga', v_size=16)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
random: nonblocking pool is initialized
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 9375,
 'm_size': 16,
 'total_image': 100,
 'total_time': 52.228493,
 'v_size': 16}

=> Accuracy should be 0.97
```




```
[*] Arguments: Namespace(m_size=16, network='cnn', num_test_images=100, run_type='cpu', v_size=16)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 1186,
 'm_size': 16,
 'total_image': 100,
 'total_time': 0.8311080000000004,
 'v_size': 16}

=> Accuracy should be 1.0

[*] Arguments: Namespace(m_size=16, network='cnn', num_test_images=100, run_type='fpga', v_size=16)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 1186,
 'm_size': 16,
 'total_image': 100,
 'total_time': 5.4424549999999995,
 'v_size': 16}

=> Accuracy should be 1.0
```



3.3 32 x 32 (m_size = 32, v_size = 32, 32x32.bit)

```
[*] Arguments: Namespace(m_size=32, network='mlp', num_test_images=100, run_type='cpu', v_size=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 2432,
 'm_size': 32,
 'total_image': 100,
 'total_time': 6.358187000000001,
 'v_size': 32}

=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=32, network='mlp', num_test_images=100, run_type='fpga', v_size=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
random: nonblocking pool is initialized
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 2432,
 'm_size': 32,
 'total_image': 100,
 'total_time': 48.576017,
 'v_size': 32}

=> Accuracy should be 0.97
```

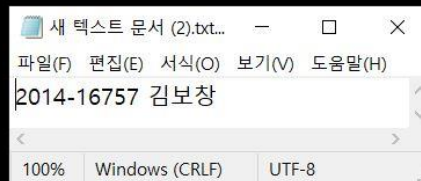


```
[*] Arguments: Namespace(m_size=32, network='cnn', num_test_images=100, run_type='cpu', v_size=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 804,
 'm_size': 32,
 'total_image': 100,
 'total_time': 1.51949899999999962,
 'v_size': 32}

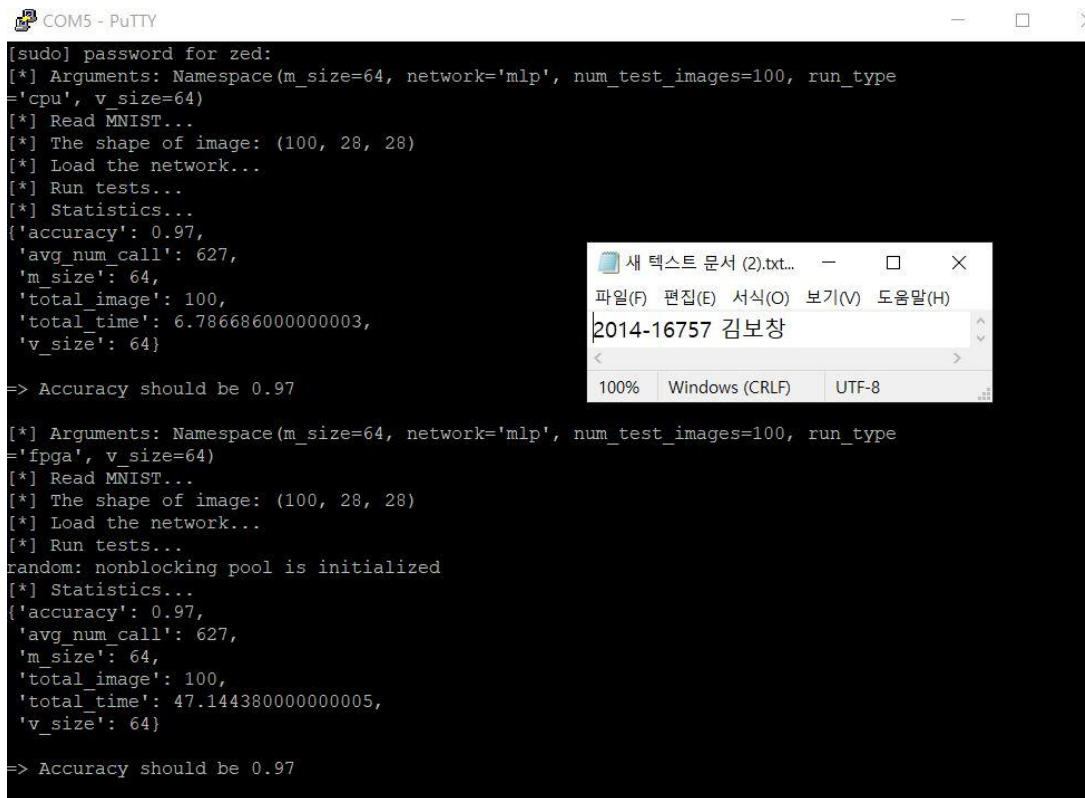
=> Accuracy should be 1.0

[*] Arguments: Namespace(m_size=32, network='cnn', num_test_images=100, run_type='fpga', v_size=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 804,
 'm_size': 32,
 'total_image': 100,
 'total_time': 10.4525,
 'v_size': 32}

=> Accuracy should be 1.0
```



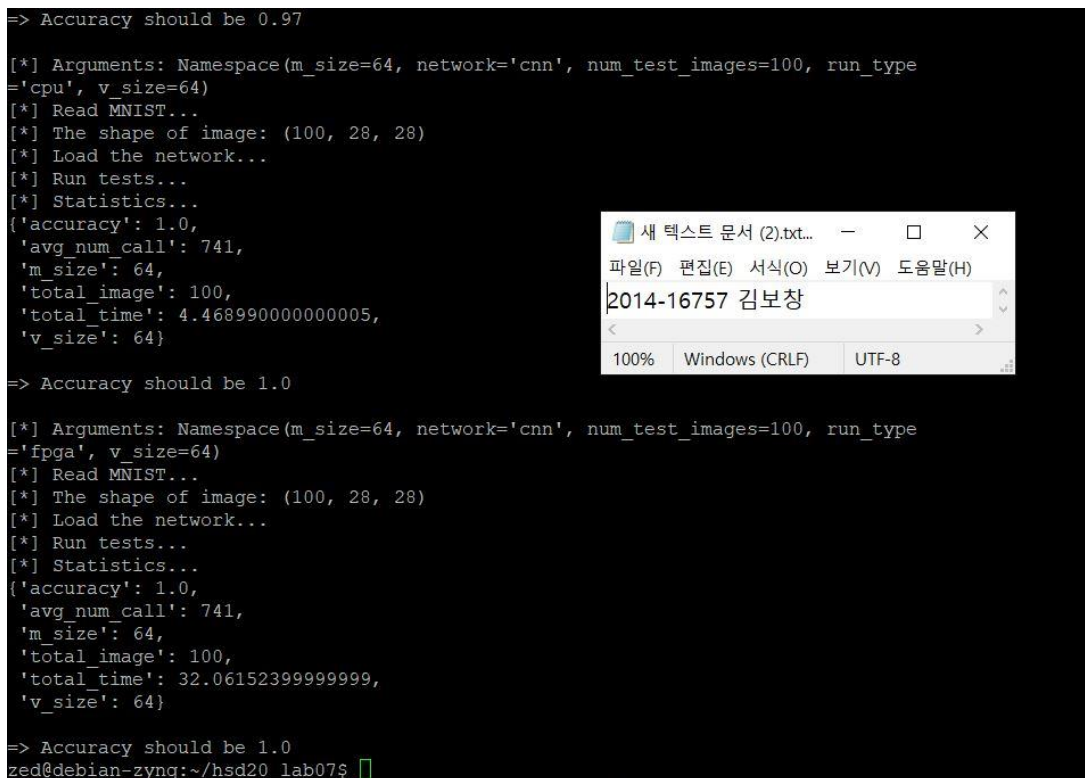
3.4 64 x 64 (m_size = 64, v_size = 64, 64x64.bit, 원래 benchmark.sh의 실행결과)



```
[sudo] password for zed:
[*] Arguments: Namespace(m_size=64, network='mlp', num_test_images=100, run_type
='cpu', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 100,
 'total_time': 6.7866860000000003,
 'v_size': 64}
=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=64, network='mlp', num_test_images=100, run_type
='fpga', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
random: nonblocking pool is initialized
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 100,
 'total_time': 47.1443800000000005,
 'v_size': 64}
=> Accuracy should be 0.97
```

먼저 mlp test다. cpu로 실행할때는 6.78초, fpga로 실행할때는 47.144초가 걸림을 알 수 있다.



```
=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=64, network='cnn', num_test_images=100, run_type
='cpu', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 100,
 'total_time': 4.4689900000000005,
 'v_size': 64}
=> Accuracy should be 1.0

[*] Arguments: Namespace(m_size=64, network='cnn', num_test_images=100, run_type
='fpga', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 100,
 'total_time': 32.061523999999999,
 'v_size': 64}
=> Accuracy should be 1.0
zed@debian-zyng:~/hsd20_lab07$
```

cnn test의 결과이다. cpu로 실행할때는 4.47초, fpga로 실행할때는 32.06초가 걸림을 알 수 있다.

결과적으로, 우리가 FPGA에 올린 matrix vector multiplier가 잘 작동함을 확인할 수 있었다.

(accuracy가 모두 정확히 일치한다)

MV multiplier의 크기를 바꿔가면서 각각에 대해 테스트를 해봤는데,

cnn의 경우 lab7의 보고서에도 서술했듯이, 필요없는 0을 곱하는 연산이 matrix 크기가 커질수록 많이 일어나 전체적인 실행시간이 늘어나게 되므로, cpu에서나 fpga에서나 matrix 크기가 커질수록 실행시간이 늘어나는 것을 볼 수 있다.

따라서, 이와 같은 이유로 cnn의 결과는 우리가 구현한 PE Array-controller에 의한 성능 향상을 잘 반영하지 못하므로, mlp의 결과에 따라서 성능을 분석할 것이다.

mlp연산의 경우, fpga 옵션을 주어 실행한 경우에서, 각각

8x8 : 63.28초

16x16 : 52.23초

32x32 : 48.57초

64x64 : 47.14초

의 성능을 내어, 더 큰 matrix-vector multiplication을 처리할 수 있는 회로일수록 계산 속도가 빠른 것을 알 수 있다.

이는, $m \times n$ matrix와 $n \times 1$ vector의 multiplication을 할 수 있는 PE-array의 경우, 내부에 m개의 연산 유닛,

즉 m개의 PE를 가지고 있기 때문에, 크기가 커질수록 전체적으로 계산속도가 빨라지는 것을 관찰할 수 있는것이다.

여기서는 m이 2배가 될수록 computation power가 2배씩 증가하므로, 계산속도의 감소를 관찰할 수 있다.

하지만, computation power의 증가만큼 실행시간의 감소량은 그렇게 크지 않은데,

전체 실행시간 = 데이터 I/O 시간 + 연산시간 에서, computation power의 증가는 연산시간만 줄여주기 때문에 그렇다. 즉, 데이터 I/O시간에 꽤 많은 시간이 소모된다고 추측할 수 있다.

또한, 결과를 보면 FPGA를 이용하여 matrix-vector multiplication을 하는 속도가 cpu를 이용한것보다 많이 느린 것을 확인할 수 있는데, 이는 다음과 같은 원인 때문이라고 생각된다.

첫번째는, IO에 너무 많은 시간이 걸리기 때문이다. matrix-vector multiplication을 하기 위해서는 matrix와 vector를 BRAM으로 옮기고, BRAM에서 다시 matrix와 vector를 FPGA의 로컬 버퍼와 글로벌 버퍼에 저장하는 과정이 필요하다.

우리는 이러한 과정을 Matrix x vector 연산 결과가 모두 계산될때까지 반복해야 하는데, cpu에서 돌아가는 코드의 경우에는 캐시를 이용하여 이러한 반복적인 데이터 전달과정에서 걸리는 시간을 많이 줄일 수 있다.

하지만, FPGA의 경우에는 방금 전에 연산에 사용된 데이터라 하더라도, 반복적으로 BRAM을 거쳐 FPGA의 버퍼로 옮겨주는 과정이 필요하기 때문에, 이 때문에 많은 시간이 걸리게 된다.

두번째로는, 연산 속도가 차이 나기 때문이다.

cpu를 사용할경우, zedboard의 cpu는 약 766mhz의 클럭으로 작동하는데, 우리의 FPGA에서 사용되는 회로의 기본 clk은 100mhz로, 클럭 자체도 차이 날 뿐더러, cpu에는 파이프 라이닝 등의 기술이 적용되어 있어 많은 양의 연산을 짧은 시간에 처리할 수 있게 되어있다.

물론, FPGA에서 연산 유닛인 PE의 개수가 꽤 많긴 하지만, CPU보다 월등히 빠른 속도가 나오지 않는 것은 이러한 원인이 있기 때문이라고 생각한다.

여기서 특히 문제가 된다고 생각하는 것이 IO에 걸리는 시간이 너무 길다는 것으로, FPGA에서 local buffer와 global buffer에 데이터를 옮기는 시간이 전체 실행 시간의 대부분을 차지하는데, 캐싱을 이용하거나, 강의시간에 배운 quantization과 같은 방법을 통해 옮겨야 할 데이터의 크기 자체를 줄이면, 꽤 큰 속도향상을 기대할 수 있을 것이다. 이러한 부분이 개선된다면 FPGA의 연산속도를 꽤 많이 끌어올릴 수 있을 것이다.

4. Conclusion

FPGA로 코드를 올리는 과정에서 구현할 것도 꽤 있었고, 무엇보다 지금까지의 실습과 달리, 베릴로그 상에서 waveform을 출력해보면서 디버깅을 하기가 힘들다는 점 때문에 구현에 꽤 애를 먹었다.

일단 보드에 올려보고 돌아가면 성공이고, 돌아가지 않으면 처음부터 무엇이 문제인지 계속 찾아야 했는데, 이러한 과정때문에 문제가 발생 했었을 때 무엇이 문제인지 몰라서 많이 힘들었다.

거기에 실수로 [31:0]의 wire를 선언해야 하는데, 1bit짜리 wire를 선언한 적이 있었는데, 이러한 wire에 길이가 다른 wire의 대입 연산을 진행할 때 아무런 경고없이 generate되는 경우도 있어서, 버그를 알아채기도 어려웠다. 그럼에도 불구하고, 직접 보드에서 돌아가는 회로를 구현해 보면서, 많은 것을 배울 수 있었던 보람찬 프로젝트 였던 것 같다.