CNN Accelerator Challenge

Team 3: 권혁성, 이웅규, 김장현, 주진원

1. 프로젝트 목표

이 프로젝트의 목표는 VGG16 model의 5 layer를 가장 빠르게 가속하는 accelerator를 설계하는 것이다. 기존에 제공된 tiled Conv2D, load input/kernel to stream, store result from Conv2D out stream 코드를 baseline으로 사용했다. 우리는 단순히 conv2d 연산만을 구연하는 것을 넘어, 이전 layer의 output 이 다음 layer의 input으로 들어가는 온전한 형태의 CNN accelerator를 구현하는 것을 목표로 프로젝트를 진행하였다. 따라서 기준으로 제시된 layer configuration을 아래와 같이 변경하여 output channel 과 다음 input channel이 이어지도록 하였다.

layer	1	2	3	4	5
R	224	112	56	28	14
C	224	112	56	28	14
M	64	128	256	512	512
N	64	128	256	512	512
K	3	3	3	3	3

→

layer	1	2	3	4	5
R	224	112	56	28	14
C	224	112	56	28	14
M	128	256	512	512	512
N	64	128	256	512	512
K	3	3	3	3	3

표 1. Layer configuration 변경

2. Accelerator 설계

Complete CNN accelerator의 overview는 [그림 1]과 같다. 5개의 Conv2D layer가 존재하고, Conv2D layer 사이에 Pooling layer와 Zero Padding layer를 추가하여 기능적으로도 완벽한 CNN accelerator를 계획하였다. 이러한 구조의 accelerator를 구현하기 위해 3가지 구체적인 목표를 설정했다.

i) tiled convolution시 설정해야 하는 tiling factor중 optimal한 값 탐색

ii) feature dimension 축소를 위한 2x2 Pooling과 Zero Padding를 설계

iii) 각 node 사이에서 data stream을 최대한 유지하여 memory access를 지양하는 설계

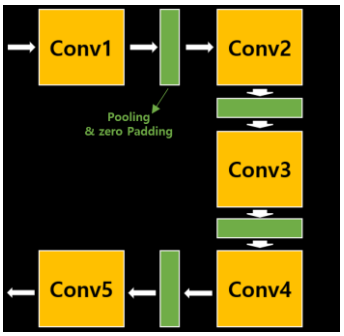


그림 1. Complete CNN Accelerator overview

3. Accelerator 구현

(1) Optimal tiling factor 탐색

optimal tiling factor를 찾기 위해 tiling factor(Tr, Tc, Tm, Tn)를 조정해가면서 Vitis HLS를 통해 simulation을 수행했고, CTC 및 Attainable Performance(GOP/s)를 측정했다. 각 layer 마다 configuration이 다르기 때문에 최적의 tiling factor역시 layer마다 달라지게 된다. 하지만 layer 와 layer 사이에서 stream을 유지하고자 하는 우리의 설계 목표를 고려하였을 때, layer 마다 tiling factor가 달라지게 되면 stream을 control 하기 힘들게 된다. 따라서 우리는 5개의 layer에 대해 공통적인 tiling factor를 사용하기로 하였다. 그 결과 Tr, Tc는 14보다 작게, Tm, Tn은 64보다 작게 설정해야 하는 조건이 생겼다.

layer 1

R	C	M	N	K	K	Tr	Tc	Tm	Tn	CTC (OP/Byte)	Measured Cycles	BRAMs	FFs	LUTs	DSPs	GOP/second	ReqBW (GB/s)
224	224	128	64	3	3	14	14	64	64	110	1145033	128	190108	271183	4096	1615	14.71
224	224	128	64	3	3	14	14	64	32	110	2187465	96	103444	140358	2048	846	7.70
224	224	128	64	3	3	14	14	32	64	88	2289865	96	95645	137361	2048	808	9.19
224	224	128	64	3	3	14	14	32	32	88	4374729	64	51990	71368	1024	423	4.81
224	224	128	64	3	3	14	7	64	64	69	1170535	64	191048	273315	4096	1580	22.90
224	224	128	64	3	3	7	14	64	64	69	1170535	64	191043	273263	4096	1580	22.90
224	224	128	64	3	3	14	14	16	16	63	17084617	32	5908	19737	256	108	1.72

layer 2

R	C	M	N	K	K	Tr	Tc	Tm	Tn	CTC (OP/Byte)	Measured Cycles	BRAMs	FFs	LUTs	DSPs	GOP/second	ReqBW (GB/s)
112	112	256	128	3	3	14	14	64	64	121	1093833	128	190132	271206	4096	1691	13.93
112	112	256	128	3	3	14	14	64	32	121	2135752	96	103444	140358	2048	866	7.13
112	112	256	128	3	3	14	14	32	64	95	2187465	96	95670	137383	2048	846	8.88
112	112	256	128	3	3	14	14	32	32	95	4271305	64	51990	71367	1024	433	4.55
112	112	256	128	3	3	14	7	64	64	73	1118822	64	191073	273337	4096	1653	22.52
112	112	256	128	3	3	7	14	64	64	73	1118823	64	191068	273285	4096	1653	22.52
112	112	256	128	3	3	7	7	64	64	41	1155126	0	192010	274378	4096	1601	38.66
112	112	256	128	3	3	14	14	16	16	66	16877769	32	5908	19737	256	110	1.65

layer 3

R	C	M	N	K	K	Tr	Tc	Tm	Tn	CTC (OP/Byte)	Measured Cycles	BRAMs	FFs	LUTs	DSPs	GOP/second	ReqBW (GB/s)
56	56	512	256	3	3	14	14	64	64	128	1067977	128	190132	271206	4096	1732	13.52
56	56	512	256	3	3	14	14	64	32	128	2109897	96	103444	140358	2048	877	6.84
56	56	512	256	3	3	14	14	32	64	99	2135753	96	95670	137383	2048	866	8.72
56	56	512	256	3	3	14	14	32	32	99	4219593	64	51990	71367	1024	438	4.41
56	56	512	256	3	3	14	7	64	64	76	1092455	64	191073	273337	4096	1693	22.33
56	56	512	256	3	3	7	14	64	64	76	1092455	64	191068	273285	4096	1693	22.33
56	56	512	256	3	3	7	7	64	64	42	1127478	0	192010	274377	4096	1641	38.90
56	56	512	256	3	3	14	14	16	16	68	16774345	32	5908	19736	256	110	1.61

layer 4

R	C	M	N	K	K	Tr	Tc	Tm	Tn	CTC (OP/Byte)	Measured Cycles	BRAMs	FFs	LUTs	DSPs	GOP/second	ReqBW (GB/s)
28	28	512	512	3	3	14	14	64	64	132	527625	128	190130	271204	4096	1753	13.30
28	28	512	512	3	3	14	14	64	32	132	1048584	96	103442	140356	2048	882	6.69
28	28	512	512	3	3	14	14	32	64	101	1055049	96	95668	137382	2048	877	8.64
28	28	512	512	3	3	14	14	32	32	101	2096969	64	51988	71366	1024	441	4.35
28	28	512	512	3	3	14	7	64	64	77	539686	64	191071	273336	4096	1714	22.23
28	28	512	512	3	3	7	14	64	64	77	539687	64	191066	273284	4096	1714	22.23
28	28	512	512	3	3	7	7	64	64	43	556854	0	192008	274376	4096	1661	39.02
28	28	512	512	3	3	14	14	16	16	69	8361417	32	5906	19735	256	111	1.59

layer 5

R	C	M	N	K	K	Tr	Tc	Tm	Tn	CTC (OP/Byte)	Measured Cycles	BRAMs	FFs	LUTs	DSPs	GOP/second	ReqBW (GB/s)
14	14	512	512	3	3	14	14	64	64	132	132057	128	190128	271198	4096	1751	13.28
14	14	512	512	3	3	14	14	64	32	132	262297	96	103440	140350	2048	881	6.69
14	14	512	512	3	3	14	14	32	64	101	263913	96	95664	137374	2048	876	8.63
14	14	512	512	3	3	14	14	32	32	101	524393	64	51984	71358	1024	441	4.34
14	14	512	512	3	3	14	7	64	64	77	134999	64	191067	273333	4096	1713	22.22
14	14	512	512	3	3	7	14	64	64	77	134999	64	191062	273281	4096	1713	22.22
14	14	512	512	3	3	7	7	64	64	43	139254	0	192004	274373	4096	1660	39.01
14	14	512	512	3	3	14	14	16	16	69	2090505	32	5900	19726	256	111	1.59

표 1. Layer별 CTC 및 Attainable Performance(GOP/second) 측정

이러한 실험 결과를 바탕으로 layer별 roof analysis를 진행했다. simulation에사용되는 VU9P FPGA의 Peak INT8 Performance가 21300GOP/s로 설정되어 있고, 이 프로젝트 내에서 short 자료형(2Byte)이 연산에 사용되므로 Computational roof는 21300의 절반인 10650GOP/s이라 가정했다. 또한 AXI interface의 channel수가 4이고, clock frequency는 250MHz, data bitwidth는 512bit이므로 ideal ReqBW는 4 * 250MHz * 512bit = 64 GB/s로 가정했다.

각 roofline analysis 에서 tiling factor가 클수록 좋은 성능을 얻을 수 있다는 결과를 얻었다.

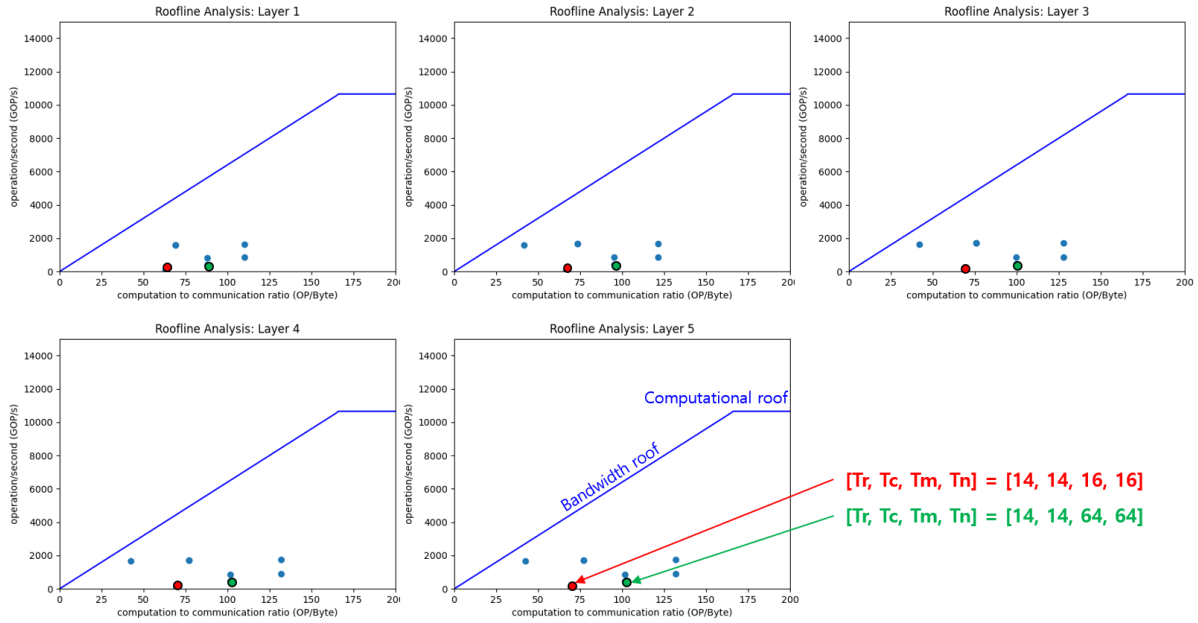


그림 2. Layer별 Roof Analysis

위 roof analysis를 분석해보면, 먼저 각 layer에서 수행하는 task가 Conv2d로 동일하기 때문에 그래프의 경향성은 layer간 차이가 크지 않다. 또한 모든 layer에 동일한 tiling factor를 적용하다 보니 tiling factor 설정에 제약이 생겼고, 이로 인해 FPGA의 resource를 최대로 활용하지 못해 낮은 bandwidth와 CTC를 얻게 되었다.

또한 전체 5개의 layer에 대해서도 roofline analysis를 수행했다. 전체 operation수와 전체 Access 수는 각 layer의 operation수의 총합과 각 layer의 Access수의 총합으로 계산했고, 각 layer들이 병렬적으로 처리되기 때문에 Measured Cycles는 각 layer의 Measured Cycles중 최대값으로 설정했다. Tr, Tc, Tm 값이 feature size가 작은 convolution의 영향을 받아 board의 resource를 최대로 활용하지 못하였다.

Total OPs	Measured Cycles	#Access	CTC (OP/Byte)	GOP/second	ReqBW (GB/s)
26820476928	17084617	201795584	66.45	392.47	5.91

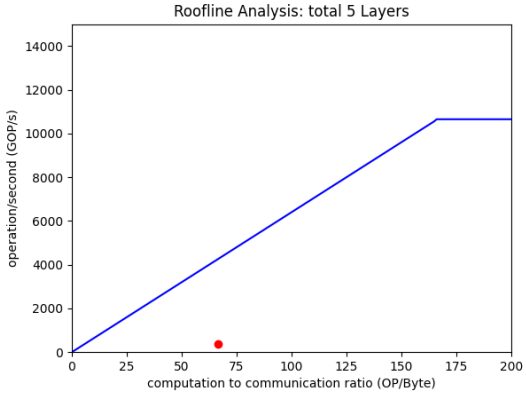


그림 3. Total 5 Layer의 Roof Analysis (14, 14, 16, 16)

(2) Max / average pooling 설계

3x3 convolution을 stride = 1, padding = 1 로 수행하게 되면 input feature와 output feature의 size는 동일하게 유지된다. 때문에 다음 conv2d layer로 들어가기 전에 downsize를 수행해 주어야한다. 따라서 pooling은 우리가 목표로 하는 complete CNN accelerator에 필수적인 구성요소이다.

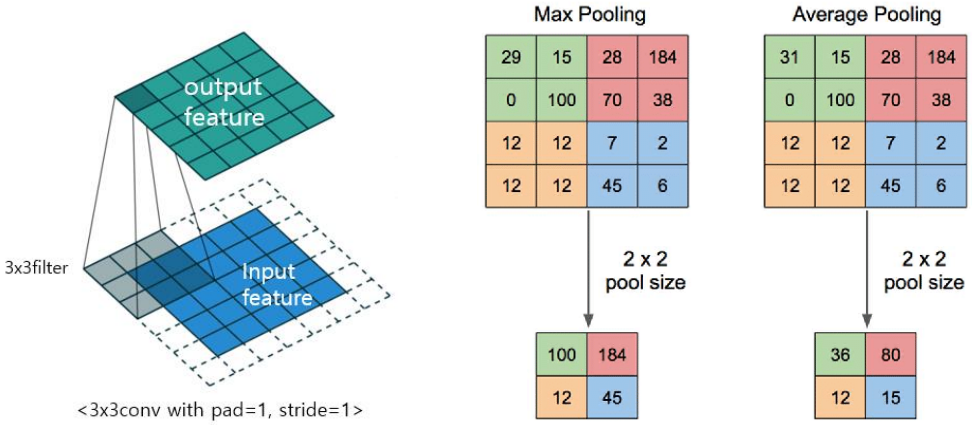


그림 4. 3x3 convoultion example and 2x2 pooling example

FPGA위에서 stream방식을 통해서 data가 전달되고 있기 때문에 이를 고려하여 pooling을 수행해야 한다. 우리는 Kernel 내부에서 stream을 고려하여 pooling을 처리하는 과정을 분석해 보았다.

Convolution Output은 다음과 같이 Tr x Tc x Tm 의 크기를 가진다. Tr, Tc, Tm 은 각각 row tile, column tile, output channel tile의 크기를 의미한다. 하나의 Stream은 Tm 만큼의 [1x1xTm]의 data를 의미하며, [그림3]과 같이 매 cycle 마다 Tr, Tc 를 한 칸씩 옮겨가며 Stream이 흐르게 된다.

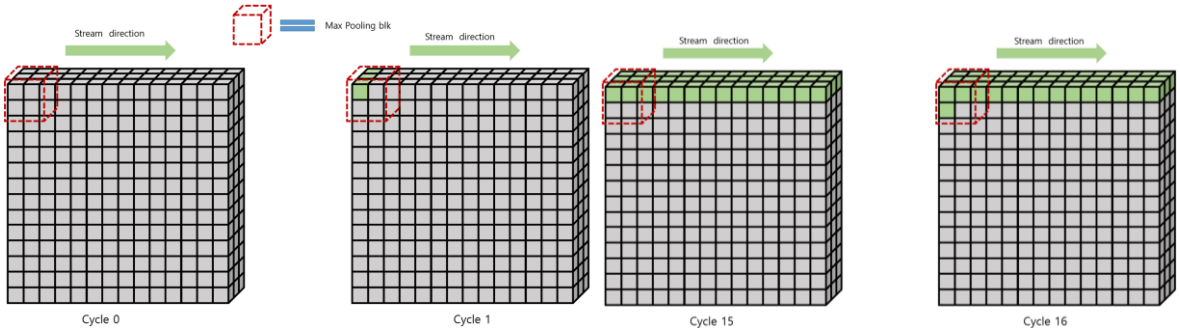


그림 5. Stream data flow in tile of feature

이상적으로는 pooling을 2x2xTm 즉 4개의 stream 마다 처리할 수 있지만, row 단위로 stream이 전달되고 있기 때문에, 첫번째 pooling을 위해서 Tc+2 cycle을 불가피하게 기다리게 된다. 물론 이 cycle 동안 전달받은 data 역시 기억하고 있어야 한다. 가장 쉬운 구현 방법은 Tr x Tc x Tm 만큼의 stream 을 모두 저장한 뒤에 pooling을 수행하는 것이다. 하지만 우리는 이 방법이 runtime과 resource 측면에서 비 효율적이라고 생각하였고, hls::vector를 활용한 memory access를 지양하는 설계를 고안하였다.

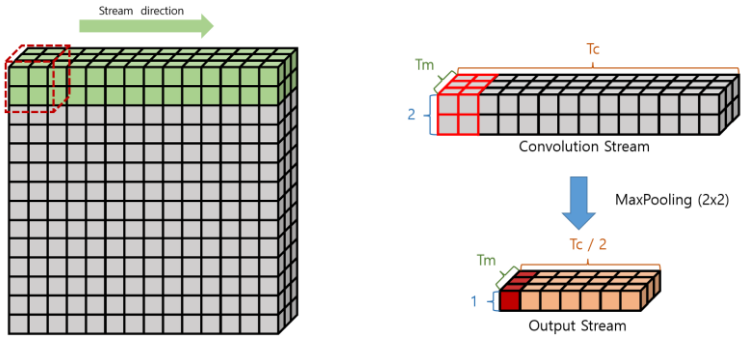
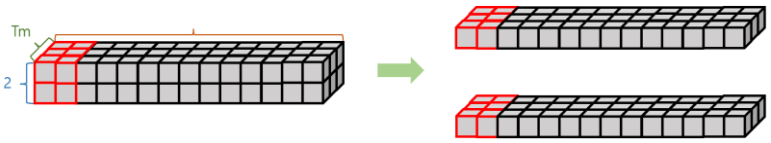


그림 6. Pooling with temporary hls::vector

[그림4]과 같이 2 x Tc x Tm의 사이즈를 가진 Convolution Stream을 Pooling하여 1 x Tm x Tc/2 의 결과를 얻을 수 있다. 이 때, 2 x Tc x Tm 크기의 vector에 stream들을 잠시 저장해두고, 가득 차면 이 vector를 이용하여 pooling을 하는 방식을 고안하였다. 이 방법을 구현하여 sw_emu는 통과하였지만, Hardware make 과정에서 hls::vector의 size 에 대한 문제를 확인하였다.

i. HLS Vector Issue

HLS Vector의 최대 크기는 4096bit이다. 때문에 2 x Tc x Tm 의 사이즈가 4096bit 이상이 되도록 선언 할 수 없었다. short type은 1개의 값을 16bit으로 표현하므로, 2 x Tc x Tm x 16 < 4096 을 만족해야 한다. 즉 Tm x Tc < 128 을 만족해야 한다. 앞서 optimal tiling factor의 탐색에서 Tm 과 Tc 값이 클수록 좋다는 것을 확인하였기 때문에, 위와 같은 제약은 성능에 방해요소로 작용하게 된다. 우리는 이를 해결하기 위해서 Tm x Tc 값을 고려하여 능동적으로 pooling_temp_vector를 설정하는 방식을 채택하였다. [그림5]의 예시를 보면, Tm = 16, Tc = 14일때, Tm x Tc x <short> = 3584bit 의 vector를 두개 선언하여 이 문제를 해결하였다.




```
static void aver_pooling( int l, hls::stream<hls::vector<short, Tm>> & in_stream, hls::stream<hls::vector<short, Tm>> & out_stream){

        //static short tinp[Tr*Tc*Tm];
        //static short t_tmp[(Tr/2)][(Tc/2)][Tn];
        //static short tout[(Tr/2+K-1)*(Tc/2+K-2)*(Tn)];
        //static short tmp_tile[2][2][Tn];

        hls::vector<short, Tm> temp_inp;
        hls::vector<short, Tm> temp_out;
        hls::vector<short, Tm*Tc> tmp_pool;
        hls::vector<short, Tm*Tc> tmp_pool_2;

        r_loop: for(int row = 0; row < R[l]; row+=Tr) {
                c_loop: for(int col = 0; col < C[l]; col+=Tc) {
                        m_loop: for(int cho = 0; cho < M[l]; cho+=Tm) {

                                init_tinp_r: for (int tr = 0; tr < Tr; tr+=2) {

                                        // init tmp_pool to 0 //
                                        for (int tc = 0; tc<Tc; tc++){
                                                for (int tm = 0; tm<Tm; tm++){
                                                        tmp_pool[tc*Tm + tm] = 0;
                                                        tmp_pool_2[tc*Tm + tm] = 0;
                                                }
                                        }
```

그림 7. Split temporary pooling vector example

이후 선언해준 tmp_pool vector를 이용해서 max / average pooling을 수행하였다.

ii. pooling result streaming issue

Tr x Tc x Tm 크기의 tile을 pooling 한 결과는 Tr/2 x Tc/2 x Tm 의 크기를 가지게 된다. 우리는 모든 layer 에서의 Tr, Tc 를 통일하였기 때문에 pooling에서 stream.write 하는대로 stream.read 하게 되면 [그림] 과 같은 문제가 발생한다.

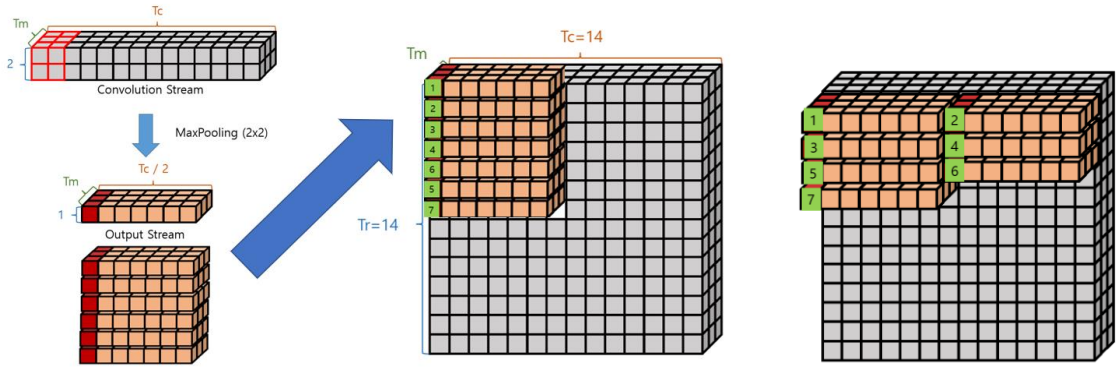


그림 8. Pooling result Stream.write vs tiled conv2d stream.read

우리는 이 문제를 다음 zero-pad layer의 stream.read 순서를 바꾸어 해결하고자 하였다.

iii. Zero padding the pooling output

Pooling 의 output을 다음 conv layer에 넣기 전에 zero padding을 수행해야 주어야 하며, Base_line 으로 제공된 Conv2d kernel code를 그대로 사용하려면, [Tc+2][Tr+2][Tm] size의 tile을 stream해주어야 conv의 functionality를 유지 할 수 있다. 즉 tile의 경계에 존재하는 data들은 인접한 tile의 convolution을 위해 여러번 Tiled conv에 사용되게 된다. 필요하게 된다. [그림7]

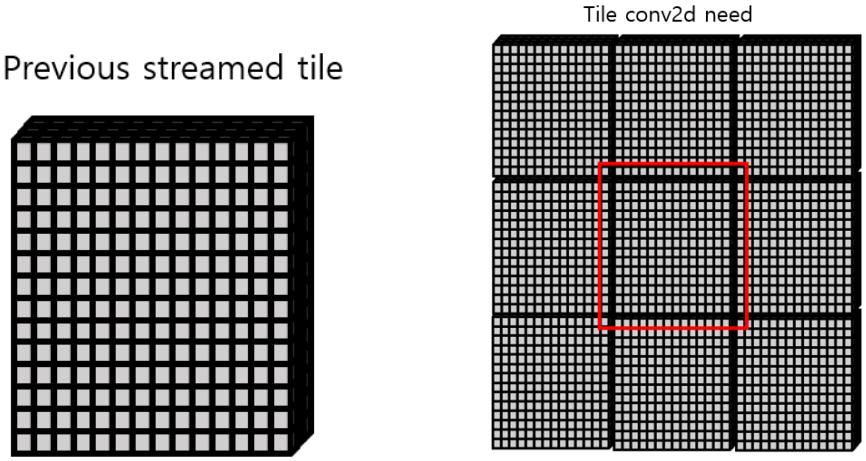


그림 9. Stream tile size difference between layer

Base line code 에서는 host 로부터의 Input load 에서 이를 고려하여 tile stream을 생성하여 convolution으로 전달하였다. 그러나 우리의 pooling result는 이미 stream 형태로 계산되어 나오

고 있기 때문에, 지나간 tile의 data를 가져와야 하는 문제에 대한 해결책이 필요하였다. 우리는 이 부분에 대해서 많은 시도를 하였지만 최종적으로 FPGA 위에서 complete CNN accelerator를 구현하는 것에 실패하였다.

우리는 최초의 목표달성에 실패하였지만, 포기하지 않고 실패 경험을 토대로 목표를 재 설정하여 [그림8]과 같이 설계 구조를 변경하였다.

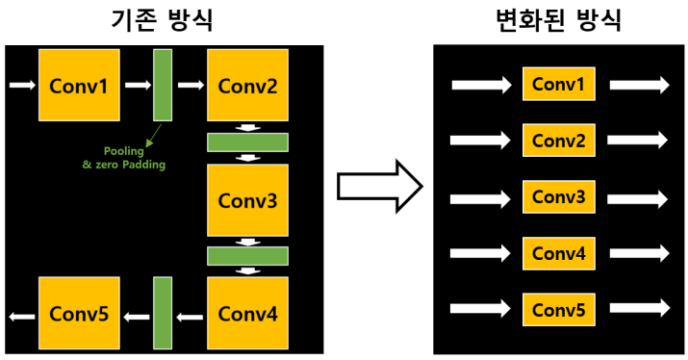


그림 10. Accelerator 설계 변경

4. Accelerator 설계 변경 (Parallel implementation)

구현 실패의 원인을 분석해본 결과 convolution 이후에 tile stream을 유지하면서 pooling과 padding을 올바르게 구현하는 것이 어려웠다는 결론을 내렸다. 이 문제를 해결하기 위해서 우리는 padding과 pooling을 host에서 처리하기로 하고, FPGA 위에서는 tiled conv2d만을 수행하는 것으로 계획을 변경하였다. 하지만 5개의 layer를 거쳐서 의미를 가지는 CNN output feature를 얻는 기능성을 포기할 수 없었고, 마침내 두가지 조건을 모두 만족시킬 수 있는[그림9]과 같은 parallel architecture를 고안하였다.

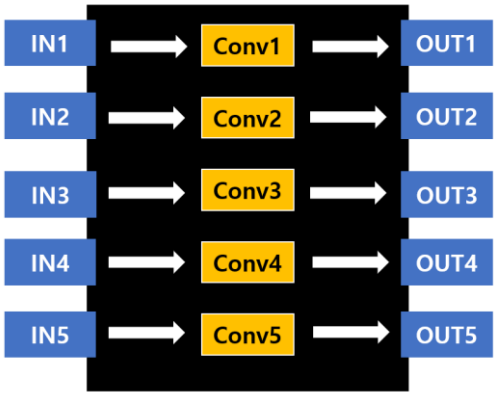


그림 11. Parallel CNN Accelerator overview

[그림9]과 같이 각 Conv2d node에 서로 다른 input을 넣고 병렬적으로 output을 출력하는 방식으로 accelerator의 설계를 변경했다. 이러한 구조적 변경은 이전 설계에 비해 몇 가지 차별점을 가진다.

i. 여러 iteration을 수행하여 complete CNN을 수행 가능하며, throughput이 증가한다.

변경된 Accelerator의 동작 방식은 [그림10]과 같다. 먼저 iteration 시작시 Conv1 node로 input을 받는다. 하나의 iteration이 끝나면 Conv1 node를 거친 input은 host에서 Max Pooling 및 Zero Padding을 수행한다. 이후 다음 iteration을 시작하면서 Conv2 node로 입력된다. 위 과정을 pipeline처럼 병렬적으로 반복하게 된다. 1iteration에서는 complete CNN result를 얻을 수 없지만, 5iteration 이후로는 결과를 얻을 수 있다.

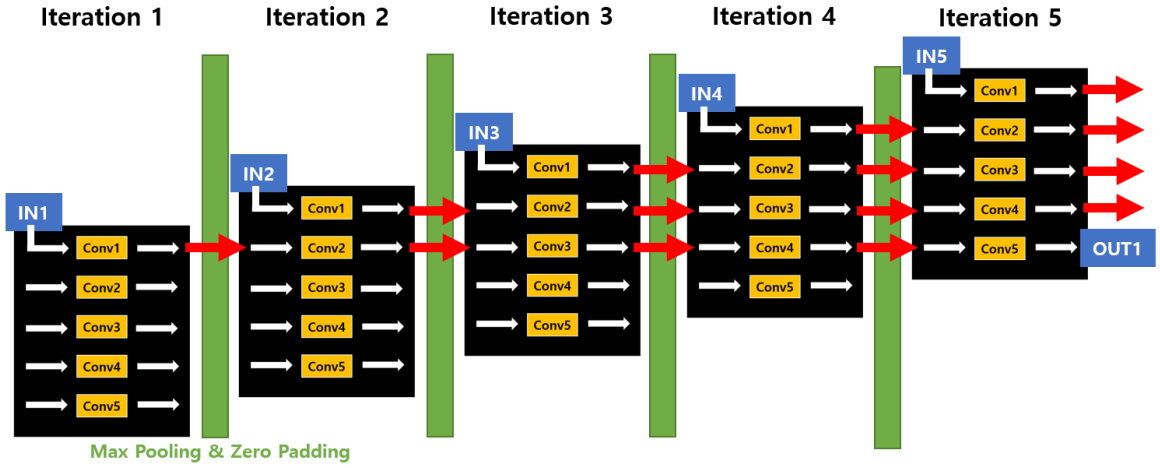


그림 12. 변경된 Accelerator의 Parallel Implementation

ii. Conv2D node별로 다른 tiling factor를 설정할 수 있고, 한정된 DSP수를 효율적인 배분함으로써 성능을 최적화할 수 있다.

우리는 optimal tiling factor를 탐색하는 과정에서 tiling factor가 클수록 성능이 좋다는 점, DSP의 수가 동일할 경우 Tn보다 Tm을 키울 때 더 높은 CTC와 Attainable Performance (GOP/s)를 얻을 수 있다는 점을 확인하였다. 이를 토대로 최대 DSP의 수 6840를 넘지 않는 이상적인 tiling factor를 [표2]와 같이 설정하였다. 5개의 Conv2D layer 중 첫번째 layer에서 가장 많은 cycle을 필요로 하므로 이를 고려하여 첫번째 layer에 더 큰 tiling factor를 주었다.

layer	Tm	Tn	DSP수	전체 DSP수
1	64	32	2048	6144
2	32	32	1024	
3	32	32	1024	
4	32	32	1024	
5	32	32	1024	

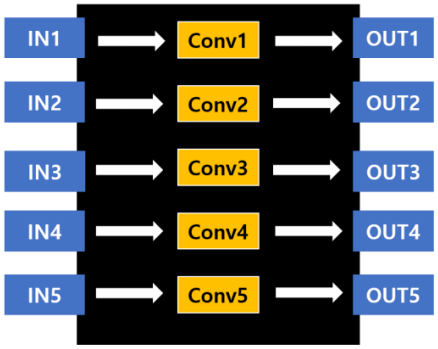
표 2. Layer별 Tm, Tn 설정 및 DSP수

5. 프로젝트 결과 및 분석

(1) 실험 1: Parallel Convolution + Maximum tile size

- 목표: Tm, Tn size를 F1 instance의 DSP를 최대한 활용하도록 설정

- F1 instance 6800 DSP 중 6144 사용



Tm	Tn	DSPs
64	32	**2048**
32	32	**1024**
32	32	**1024**
32	32	**1024**
32	32	**1024**

그림 13. Parallel CNN Accelerator & channel tile configuration

- SW Emulation 결과: Functionality check

```
Running FPGA CNN...
Done.
Time: 63.9953, GFLOPS: 0.24568
Conv1 verification
Conv1 Error count : 0
Conv2 verification
Conv2 Error count : 0
```

```
Conv3 verification
Conv3 Error count : 0
Conv4 verification
Conv4 Error count : 0
Conv5 verification
Conv5 Error count : 0
```

- HW make 결과: Fail ('Cannot allocate memory' error)

```
===>The following messages were generated while processing /home/centos/src/project_data/aws
l/prj/prj.runs/impl_1 :
ERROR: [VPL 17-179] Fork failed: Cannot allocate memory
ERROR: [VPL 17-179] Fork failed: Cannot allocate memory
ERROR: [VPL 60-773] In '/home/centos/src/project_data/aws-fpga/Vitis/examples/xilinx_2021.1
oblem implementing dynamic region, impl_1: Design Initialization ERROR, please look at the
u9p-f1_shell-v04261818_201920_2/link/vivado/vpl/prj/prj.runs/impl_1/runme.log' for more inf
WARNING: [VPL 60-732] Link warning: No monitor points found for BD automation.
ERROR: [VPL 60-704] Integration error, problem implementing dynamic region, impl_1: Design
es/xilinx_2021.1/cnn_2/_x.hw.xilinx_aws-vu9p-f1_shell-v04261818_201920_2/link/vivado/vpl/pr
ERROR: [VPL 60-1328] Vpl run 'vpl' failed
ERROR: [VPL 60-806] Failed to finish platform linker
INFO: [v++ 60-1442] [21:36:19] Run run_link: Step vpl: Failed
Time (s): cpu = 00:03:27 ; elapsed = 02:55:18 . Memory (MB): peak = 2033.902 ; gain = 0.000
ERROR: [v++ 60-661] v++ link run 'run_link' failed
ERROR: [v++ 60-626] Kernel link failed to complete
ERROR: [v++ 60-703] Failed to finish linking
INFO: [v++ 60-1653] Closing dispatch client.
make: *** [build_dir.hw.xilinx_aws-vu9p-f1_shell-v04261818_201920_2/cnn.xclbin] Error 1
```

- 원인 분석: DSP를 최대한 사용하기 위해 가능한 가장 크게 Tm, Tn 값을 정하였는데, 이로 인해 필요한 memory size가 커져 allocation error가 발생한 것으로 보임.

- 해결을 위해 tile size를 축소시켜 재시도

(2) 실험 2: Parallel Convolution + Small tile size

- Tm, Tn 값을 전부 16으로 낮춰서 다시 실험 진행

- F1 instance 6800 DSP 중 1280(256 * 5)개 사용

Tm	Tn	DSPs
16	16	**256**
16	16	**256**
16	16	**256**
16	16	**256**
16	16	**256**

```
Found Platform
Platform Name: Xilinx
INFO: Reading cnn.awsxclbin
Loading: 'cnn.awsxclbin'
Trying to program device[0]: xilinx_aws-vu9p-f1_shell-v04261818_201920_2
Device[0]: program successful!
Running FPGA CNN
Done.
Time: 0.0912319, GFLOPS: 172.334
Conv1 verification
Conv1 Error count : 0
Conv2 verification
Conv2 Error count : 0
Conv3 verification
Conv3 Error count : 0
Conv4 verification
Conv4 Error count : 0
Conv5 verification
Conv5 Error count : 0
PASSED!
```

그림 14. channel tile configuration & F1 measure result

- Tile size 변경 후 make hw (create bitstream) 성공 (memory allocation error X)

- F1 instance에서 time evaluation 결과: **0.0912319 sec / 172.33 GFLOPS**

- 성능 비교: VGG conv1 + pseudo pooling (2*2 window에서 왼쪽 위 값만 뽑음) 만 구현하여 F1 instance에서 실험했을 때, 0.0635655sec, 116.396 GFLOPS의 결과가 측정되었다. 이를 바탕으로 새로 설계한 HW와 convolution 1회당 시간을 비교해보면 0.063 sec >> 0.018 sec (0.091 sec / 5) 이고, GFLOPS 역시 172.334 >116.396 이므로 parallel CNN 방식이 더 빠르게 동작함을 확인할 수 있다.

```
Initialize input data...
Initialize output data...
Done.
5 layer start...
*******Layer 1 start*******
Conv2d_sw...
MaxPool2d...
Done.
Time: 1.59497, GFLOPS: 4.63879
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.awsxclbin
Loading: 'vadd.awsxclbin'
Trying to program device[0]: xilinx_aws-vu9p-f1_sh
Device[0]: program successful!
Running FPGA Conv...
Done
Time: 0.0635655, GFLOPS: 116.396
```

그림 15. Single convolution with pseudo pooling result

- 분석: 동일한 16*16 (Tm*Tn) tile size 일 경우, single convolution만 kernel에 구현했을 때는 한 번에 256 DSP만 사용하여 연산을 해야 한다. 따라서 나머지 DSP는 연산에 참여하지 못하게 되는데, 우리가 구현한 대로 5개 convolution을 병렬적으로 수행하게 되면 256*5 만큼의 DSP를 같은 시간에 사용할 수 있다.

하지만 5개 convolution에 대해 input read와 output write을 해야 하므로, bandwidth의 제한으로 인해 memory read/write에 더 많은 cycle이 소모될 것이다. 따라서 DSP 측면에서만 보면 5배 빠른 속도가 예상되지만, memory access로 인해 그보다는 적은 비율의 성능 향상이 있는 것으로 분석된다.

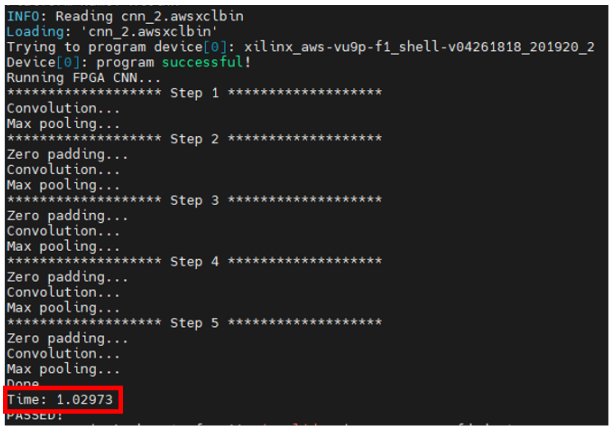
(3) 실험 3: Parallel Convolution X 5 steps

- 5회에 걸쳐 convolution을 반복적으로 수행하며, 이전 iteration의 output이 다음 iteration에서 input으로 사용되도록 host code 수정

- Iteration 사이에 host에서 max pooling + zero padding 수행

- Tm, Tn = 16, 16으로 유지

- F1 instance time evaluation: **1.02973 sec**



- 분석: Iteration 없이 convolution만 1회 수행하였을 때 0.09 sec가 측정되었다. 이를 바탕으로, 5회 반복 시 convolution에만 소요되는 시간을 계산해보면 0.09 sec * 5 = 0.45 sec임을 알 수 있다. 전체 시간이 약 1.03 sec로 측정되었으므로 나머지 시간인 0.58 sec가 host의 zero padding과 max pooling에 소요된 것으로 분석된다.

6. 결론

- VITIS HLS 기반 convolution 구현 code를 활용하여, VGG network의 5개 convolution에 대해 병렬적으로 연산 가능한 hardware를 구현하였다. 5개 convolution이 FPGA의 DSP를 나눠서 사용 가능하도록 설계하였으며, tile size가 같다는 가정 하에 single convolution만 구현했을 때에 비해 3배 이상의 속도 향상을 보였다. 이를 바탕으로 host code에 max pooling과 zero padding을 포함하여 한 layer의 output이 다음 layer의 input으로 이어지는 network 구조를 구현하는 것에 성공하였다.

7. GitHub Repository

- HLS_CNN: https://github.com/LeeWoongkyu/HLS_CNN