Report(team9)

CNN Accelerator Challenge

FPGA hardware emulation using layer-specific architecture and tiled pooling

2020180183 융합전자공학과 이정혁

2020226073 신소재공학과 송충석

2021238179 인공지능학과 김수경

1. Introduction

오늘날 하드웨어를 이용한 convolutional neural network(CNN) 가속기 플랫폼은 총 4가지로 CPU, GPU, FPGA, ASIC으로 나뉜다. 각 플랫폼마다 trade off를 가지고 있으며 Machine Learning (ML) 전용 하드웨어를 설계한다는 의미는 보통 FPGA 혹은 ASIC 설계를 의미한다. FPGA로 설계시 ASIC에 비해 성능효율이 떨어진다는 단점이 있어 성능면에서는 적합하지 않으나, ASIC은 하드웨어 제작에 긴 시간이 걸리며 한번 제작하면 설계가 고정되기 때문에 유연한 설계가 불가능하여 FPGA가 설계면에서 유용하다는 장점이 있다. 특히 ML 가속기가 상용화까지 이루어지고 있는 오늘날에 time to market은 설계 플랫폼을 정하는데 아주 중요한 요소가 되고 있는데 FPGA가 이점에서 우위를 점하고 있다고 볼 수 있다. 특히, AWS에서 제공하는 FPGA cloud 서비스는 직접 FPGA를 구매하지 않고도 cloud 비용만을 가지고도 FPGA를 설계할 수 있다는 점에서 FPGA의 장점이 더 강화된 서비스라고 볼 수 있다.

위와 더불어 FPGA를 설계하기 위해 verilog나 VHDL 같은 hardware description language(HDL)를 이용하지 않고 c++과 openCL 기반의 high level synthesis(HLS) 을 이용하여 자동으로 micro architecture 부분을 결정할 수 있기 때문에 생산성이 극대화되는 효과를 가져올 수 있다. 이러한 추세에 따라 최근 HLS를 이용한 연구가 많이 늘어나고 있으며(Figure 1.) platform 회사들도 FPGA와 HLS를 결합한 방향으로 많이 시장진입을 하고 있다.

본 보고서에서는 Xilinx VITIS tool을 이용하여 HLS기반 추론용 CNN 가속기를 FPGA로 설계할 예정이고, VGG network를 이용하여 inference까지 진행하고 tiling 기법을 이용하여 최적의 하드웨어 성능을 구현해볼 예정이다.

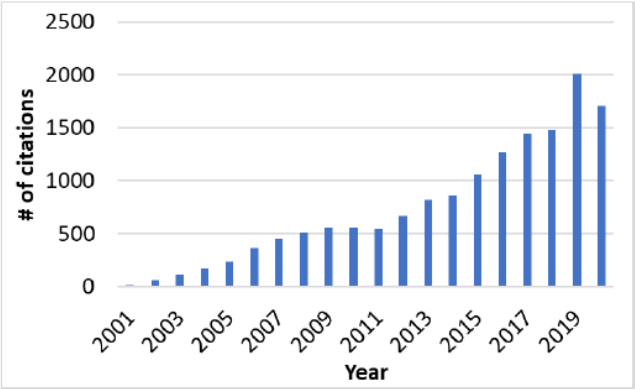


Figure 1. HLS에 관한 논문 인용 횟수

2. VGG 16 network

하드웨어 설계에 적용할 네트워크는 VGG network로 옥스포드 대학의 연구팀에 의해 개발된 모델로서 2014년 이미지넷 이미지 인식 대회에서 준우승을 한 모델이다. VGG 연구의 핵심은 네트워크의 깊이를 깊게 만드는 것이 성능에 어떤 영향이 미치는지를 확인하고자 한 것으로 역사적으로 봤을 때, 이때 이후부터 네트워크의 깊이가 깊어진 것을 확인 할 수 있다. 따라서 성능을 좌우할 수 있는 다른 변수인 kernel size를 모두 3*3 으로 통일한 것이 특징이다. 자세한 network의 구성은 Figure 2와 같다.

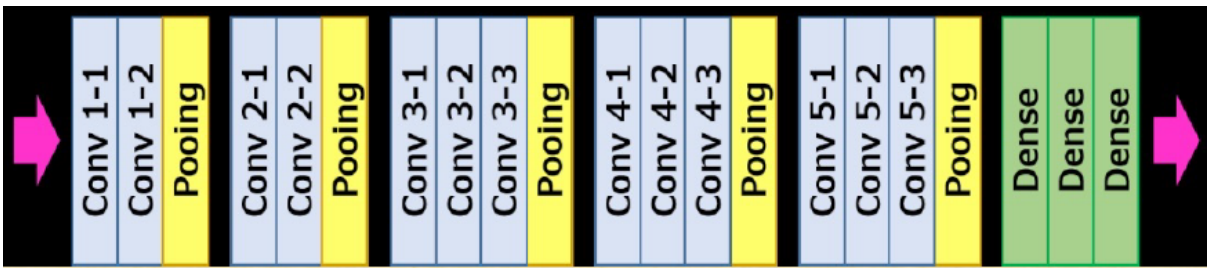


Figure 2. VGG16의 구조

Figure 2에서 Pooling layer 사이마다 있는 Convolution layer는 같은 특성을 가지고 있으며 Table1과 같다. R과 C는 output feature map의 row와 column의 개수이며, M과 N은 각각 output,

Table 1. VGG16의 5개의 대표 layer

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

input feature map의 channel 개수이다. K는 kernel size를 의미한다. 따라서 각 layer마다 최적화의 조건이 다르기 때문에 Table1 에 있는 layer 5개에 대하여 각각 최적화를 진행한 후 성능을 simulation 할 계획이다.

3. Design goals

Project의 주제는 VGG16 의 대표 layer 5개를 구현하고 연산에 소모되는 execution time을 최소화 하는 것이다. Execution time을 최소화시키 위해 디자인 목표는 총 4가지로 설정했으며, 1) 전체 CNN 구조를 구현할 수 있는 HLS 설계, 2) DRAM access 최소화, 3) FPGA resource에 맞는 architecture 결정, 4) 최적의 성능을 위한 factor 찾기 이다. 전체 CNN 구조를 구현할 수 있도록 하기 위해 오직 한 개의 FPGA bitstream만을 이용하며, convolution layer 뿐만 아니라 max pooling, ReLU activation function 까지 구현하고자 한다.

3.1. 전체 CNN 구조 구현

단일 layer의 simulation이 아닌 Neural Network를 simulation 하기 위해 convolution 연산만 FPGA에 적용하는 것이 아닌 Max pooling, ReLU activation도 구현하고자 한다. 하지만 원활한 연산을 위해 data를 임시로 저장할 buffer 설정과, dataflow, FPGA가 감당할 수 있는 최대 resource 등 여러가지 고려해야 할 점이 증가하게 된다.

3.2. DRAM access 최소화

FPGA device와 off chip memory인 DRAM은 단일 device 상에 놓여져 있지 않기 때문에 제한적인 bandwidth를 가지고 data를 전달해야 한다. 이러한 off chip DRAM의 access가 증가하면 자연히 execution time의 증가로 올 것이라 생각해 DRAM의 access를 최소화 하기 위해서 input data를 가져오는 과정과 모든 연산이 끝난 후의 최종 output data를 저장하는 과정에서만 DRAM access를 허용하고자 한다. 따라서 Multilayer 설계시 layer 사이마다 해당 layer의 연산결과를 임시로 저장할 수 있는 buffer를 두어 연산의 결과를 저장할 예정이며, 이때 하드웨어 총 메모리 크기가 buffer보다 작을 경우 추가로 DRAM access를 해야 할 수도 있다.

3.3. FPGA architecture 선정

FPGA architecture는 크게 두가지로 고려해볼 수 있는데, 첫번째는 cross-layer architecture로 모든 layer를 연산할 수 있는 최적의 CNN 연산 장치 1개를 FPGA에 구현하는 것이고(Figure 3.(좌)), 두번째는 layer-specific architecture로 각각의 layer마다 최적화를 시켜 FPGA 안에 최적화된 CNN 연산 장치 5개를 구현하는 것이다(Figure 3.(우)). Cross-layer architecture를 사용할 경우 DSP와 memory를 각 layer마다 배분할 필요가 없어 최대한 많이 설정할 수 있겠지만 input layer의 configuration에 따라 최대한 활용하지 못할 수도 있다. Layer-specific architecture를 사용할 경우에는 각 layer마다 최적화를 한 것이기 때문에 더 높은 성능을 낼 것이라 예상되지만 각각 DSP

및 memory를 잘 분배해야 한다. 본 보고서에서는 tiling 기법과 제한된 buswidth를 적용하기 때문에 layer-specific architecture가 더 적합할 것이라 판단, 최종선정하였다.

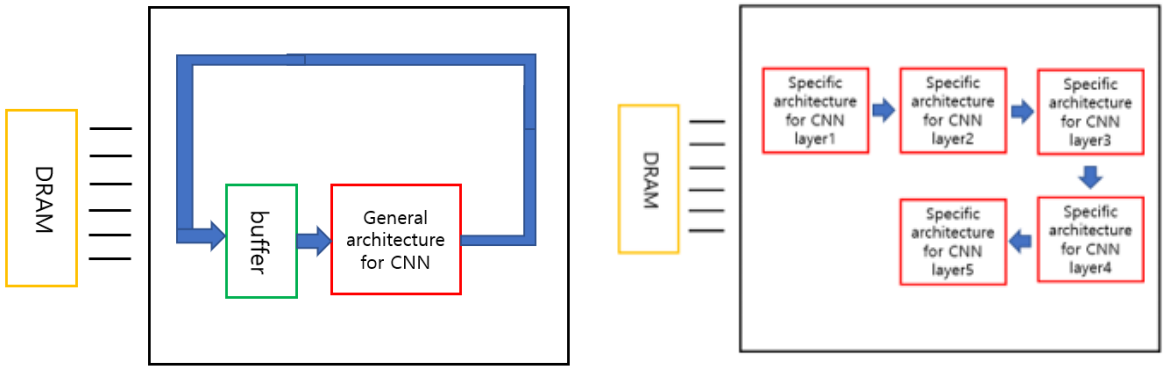


Figure 3. cross-layer architecture(좌), layer-specific architecture(우)

3.4. 각 layer 마다 최적화된 factor 찾기

Layer-specific architecture의 장점은 각 layer를 최적화시킬 수 있다는 점이기 때문에 각 layer 마다 factor를 선정해주어야 한다. 본 CNN 구조는 tiling 기법을 사용하기 때문에 최적의 tiling factor를 찾는 과정을 진행하였고 자세한 것은 chapter 4에서 소개한다.

4. Tiling factor 최적화 (Tr, Tc, Tm, Tn)

한번에 모든 영역을 convolution 연산을 할 경우 data reuse 측면에서 효율성이 떨어지게 된다. 그에따라 memory access하는 횟수가 증가해 성능이 감소하게 되는데 이를 해결해주는 방법이 tiling으로 특정 작은 영역을 지정해(tile) data reuse 특성을 높이는 알고리즘이다. 본 보고서에서는 각각의 tiling factor를 input feature map size의 tiling size를 Tr (row), Tc (column) 이라 하고, channel의 관점으로 input feature map channel을 Tn, output feature map channel을 Tm 이라 정했다.

4.1. Tm, Tn 최적화

Tm, Tn 은 unrolling factor로 tiling 시 parallel한 연산을 할 수 있는 parameter이다. 따라서 동시에 연산을 해야하기 때문에 동시에 FPGA의 resource 중 DSP와 직접적인 연관이 있으며

필요한 DSP 개수 = Tm * Tn ----------(1)

가 된다. 본 프로젝트에서 사용한 FPGA resource의 DSP 개수는 6840개이므로 각 layer specific architecture마다 1024개를 분배하였다. 더불어 BUSWIDTH가 32로 제한되어있기 때문에 Tm 혹은 Tn 값은 32보다 키워도 한번에 parallel하게 연산되지 못할 것이라 예상하여 Tm, Tn 의 최대개수도 32로 제한하였다. 따라서 Tm 과 Tn 은 모든 layer가 동일하게 32개로 선정하였다.

4.2 Tr, Tc 최적화

Tr, Tc 를 최적화하기 위해 각 layer마다 Tr, Tc에 따른 필요 memory(ReqMem)와 연산량(OP/cy)을 측정하였다. Layer1의 input data의 크기가 가장 크기 때문에 필요 memory도 많을 거라고 판단하여 layer1에 대하여 먼저 조사하였다.

우선 변수를 줄이기 위해 Tr 과 Tc가 같다고 놓고 측정해본 결과 Tr, Tc가 작아질수록 요구 memory 크기는 작아지지만, 연산량도 줄어드는 것을 확인 할 수 있었다(Figure 4(좌)). 더불어 FPGA resource의 memory는 75Mbit를 가지고 있으므로 모든 layer architecture에 memory를 분배하기 위하여 한 개의 layer에는 최대 10Mbit 이상 사용하지 않다는 제한을 두어 layer1의 Tr, Tc는 56으로 선정하였다.

Tr, Tc의 값을 56으로 기준으로 <Tr, Tc>=<14,224>, <28,112>, <56,56>, <112,28>, <224,14> 의 조합으로 다시 한번 memory와 연산량을 조사하였을 때 필요 memory는 가장 작으면서 연산량을 가장 큰 sweet spot을 찾을 수 있었다(Figure 4(우)). 최종적으로 layer1의 <Tr, Tc, Tm, Tn> 의 값은 <56, 56, 32, 32>로 선정하였다.

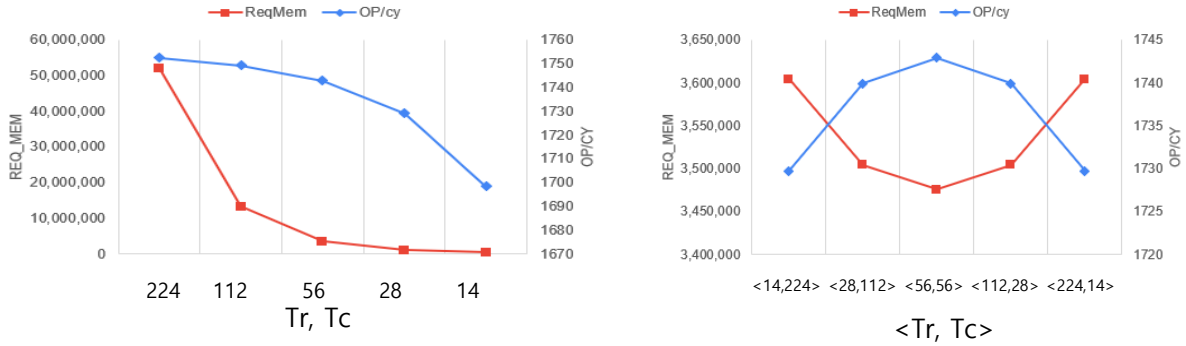


Figure 4. Tr, Tc가 같다고 가정했을시의 필요 memory크기와 연산량(좌), Tr, Tc 조합에 따른 memory 크기와 연산량(우)

위와 마찬가지 방식으로 layer2~layer5 까지 조사한 결과를 Table 2에 표현하였다. 타일링이 크기는 input의 크기를 넘지 못하기 때문에 layer4, layer5는 input size의 크기와 동일하게 두었다. Tiling의 크기를 비교적 크게 잡았음에도 불구하고 연산에 필요한 tiling memory 의 부족함은 없었다.

Table 2. Layer configuration에 따른 선정된 최적화 tiling factor

	R	C	M	N	K	Tr	Tc	Tm	Tn	OP/cy	MEM (bit)
Layer1	224	224	64	64	3	56	56	32	32	1734	3,475,456
Layer2	112	112	128	128	3	56	56	32	32	1785	3,475,456
Layer3	56	56	256	256	3	56	56	32	32	1807	3,475,456
Layer4	28	28	512	512	3	28*	28*	32	32	1803	1,009,664
Layer5	14	14	512	512	3	14*	14*	32	32	1770	378,880

5. Memory efficiency approach

Layer-specific architecture 방식의 핵심은 모든 layer를 하나의 FPGA에 다 구현할 수 있는지의 여부이다. 따라서 buffer의 크기를 사전에 조사하여야 하는데, 온전한 Neural Network를 구현하기 위해 pooling을 진행하게 되면 추가적인 buffer가 필요하게 된다. Dataflow 를 적용하지 않은 상태로 pooling layer를 위한 buffer까지 설정할 경우 Table 3과 같이 총 120Mbit 정도의 memory가 필요하게 되는데 FPGA의 resource는 약 75Mbit의 용량밖에 없으므로 구현이 불가능하다. 이 문제를 해결하기 위해 Tiled pooling method 를 소개하고자 한다.

Table 3. 각 layer마다 필요한 buffer 용량

Layer	Layer1	Pooling	Layer2	Pooling	Layer3	Pooling	Layer4	Pooling	Layer5	Pooling
Size(bit)	51M	13M	26M	6M	13M	3M	6M	1.6M	1.6M	-

제공된 base code에 의하면 stream 형식으로 dataflow를 구성하여 FIFO가 되게 구성이 되어있다. 따라서 연산을 위한 memory는 tiling 값을 저장하기 위한 buffer만 필요하지만, pooling을 하게 된다면 tiling 결과값을 모두 저장하여 pooling을 해주어야 하기 때문에 이 부분에서 overhead가 발생한다.

본 project 에서 소개하는 Tiled pooling method 방식(코드 Figure 5)은 tiling을 통해 얻은 tile에다가 pooling을 직접 하는 것이다. Base code의 tout에 tiling convolution 연산을 임시로 저장하는데 이 tout에 pooling을 직접하여 이 결과값을 그대로 buffer에 저장한다면 buffer를 두개 사용할 필요가(convolution 결과용, pooling 결과용) 없어진다. 따라서 tile을 임시로 저장할 buffer와 pooling 연산을 저장할 buffer로 필요 memory 크기가 줄어들게 되며 그 크기는 Table 4에 정리하였다. 필요 memory는 총 37Mbit가 되어 기존에 비해 3.2배정도 memory 효율을 늘렸고 더불어 단일 FPGA에 5개의 layer specific architecture 를 구현할 수 있게 되었다.

```
          // Main computation
          ki: for (int ki = 0; ki < K; ki++) {
            kj: for (int kj = 0; kj < K; kj++) {
              tr: for (int tr = 0; tr < Tr; tr++) {
                tc: for (int tc = 0; tc < Tc; tc++) {
#pragma HLS pipeline II = 1
                  tm: for (int tm = 0; tm < Tm; tm++) {
#pragma HLS unroll
                    tn: for (int tn = 0; tn < Tn; tn++) {
#pragma HLS unroll
                      L: tout[tr][tc][tm] += tker[ki][kj][tm][tn] * tinp[tr+ki][tc+kj][tn];

          }}}}}}
          //tiled_pooling
          p_tm: for (int pm = 0; pm < Tm; pm++) {
            p_tr: for (int pr = 0; pr < Tr/2; pr++) {
              p_tc: for (int pc = 0; pc < Tc/2; pc++) {
                  int relu = Max(tout[2*pr][2*pc][pm], tout[2*pr][2*pc+1][pm], tout[2*pr+1][2*pc][pm], tout[2*pr+1][2*pc+1][pm]);
                P: maxx[pr][pc][pm] = relu > 0 ? relu : 0;
        }}}
```

Figure 5. tiled pooling method를 적용한 코드. ReLU도 추가하였다.

Table 4. Tiled pooling method를 적용했을 때 필요한 buffer 용량

Layer	Layer1	Pooling	Layer2	Pooling	Layer3	Pooling	Layer4	Pooling	Layer5	Pooling
Size(bit)	3.9M	13M	3.9M	6M	3.9M	3M	1.1M	1.6M	0.4M	-

6. Result

6.1. 5-layer configuration

주어진 5개의 layer를 이어 붙이기 위하여 3개의 convolution layer를 추가해 각 layer마다의 channel 의 크기를 맞춰주었다(Figure 6). Figure 6에서의 Maxpooling은 따로 layer를 둔 것이 아니라 실제로는 Layer1, 2, 3, 4, 5 안에서 tiled pooling method 를 이용하여 같이 진행되었으며 모든 layer에서 padding은 추가되었다. Sw_emu 결과 Time은 90.0424s가 나왔고, GFLOPS 는 0.118119가 나왔다. 연산량은 총 8개의 convolution layer의 연산량을 모두 합친 것으로 계산하였다.

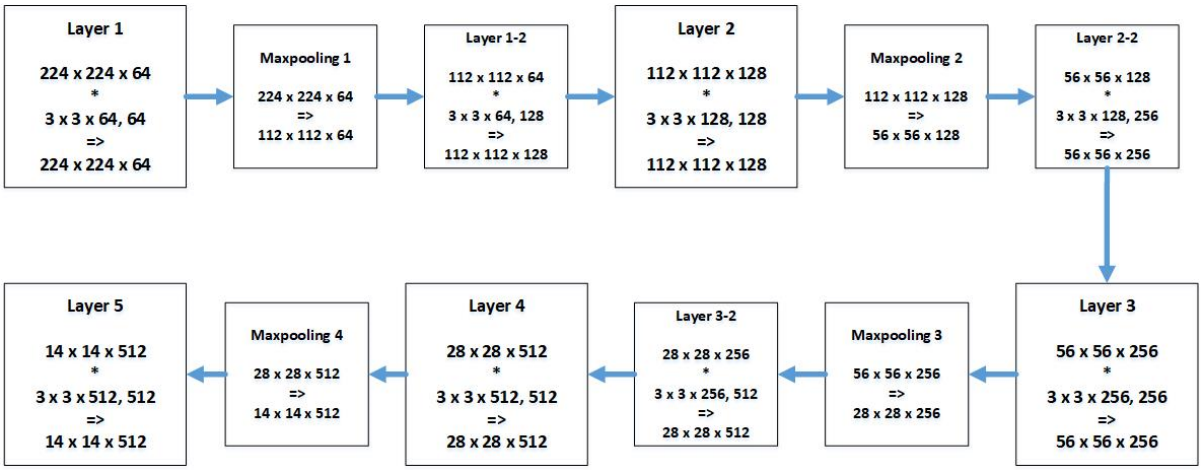


Figure 6. 기본 layer 5개에 layer1-2, layer2-2, layer3-2를 추가함.

Hw_emu은 simulation에 실패하였는데 "cannot read data in multiple processes" 오류가 확인되었다. 이는 여러 layer의 kernel 값을 streaming 형식으로 동시에 계속 받아오는데, 여기서 충돌이 일어난 것으로 예상된다.

6.2. Single layer configuration

5-layer 측정에 실패하여 single layer 로 측정을 다시 진행하였다. Single layer로 측정 시에는 Table 1의 configuration을 그대로 따랐으며 layer마다 한개씩 총 5개의 결과가 나왔다. 이 때, convolution 연산만 진행한 것이 아닌 tiled pooling method 방식으로 max pooling까지 함께 진행하였다.

Layer1, 2, 3 는 180분까지 hw_emu을 진행하였으나, 시간제한이 걸려있는지 180분까지만 연산을 해서 멈춰버렸고, layer 4, 5는 180분 이전에 hw_emu이 끝났기 때문에 온전히 emulation time과

GOPS값을 계산할 수 있었다. Operation 수는

$$OPs = R \times C \times M \times N \times K \times K \times 2 \text{----------(2)}$$

으로 계산하였고, pooling은 연산량에 포함되지 않았다. Layer 1,2,3의 경우에는 전부 simulation이 되지 않았기 때문에 GOPS를 계산하지 않았지만, 180분까지의 진행상황을 Table 5에 표시해두었다. Input, ker 의 read 용량을, output의 write 용량을 적었다. Layer 1,2,3의 경우 타일의 크기가 같기 때문에 같은 시간동안 read 하는 data의 크기는 같다는 것을 알 수 있었고, output은 input의 size가 다르기 때문에 write 의 크기도 함께 작아진다는 것을 알 수 있었다.

Layer 4, 5의 경우에는 식 (2) 를 이용하여 GOPS 를 계산한 결과 각각 196.4, 206.9 GOPS 를 얻었고, 이 둘의 performance 차이가 많이 나지 않는 것으로 보아 Tm, Tn 을 BUSWIDTH만큼 최대로 끌어올려 streaming이 layer 크기에 영향을 받지 않고 제대로 진행되고 있다는 것을 알 수 있었다.

Table 5. Single layer configuration 실험 결과

Layer(w/ pooling)	Sw_emu		Hw_emu		
	EM_time(s)	GOPS	EM_time(ms)	GOPS	IN/KER/OUT (KB)
Layer1	14.6863	0.251893	18.0276	-	13456, 1152, 1568
Layer2	15.2288	0.24292	17.7838	-	13456, 1152, 784
Layer3	16.2478	0.227685	17.6613	-	13456, 1152, 392
Layer4	15.3818	0.240504	17.8469	206.9	-
Layer5	15.1582	0.244051	4.70707	196.4	-

7. Discussion

7.1. streaming kernel

5-layer configuration 에서 hw_emu simulation을 실패한 이유는 총 8개의 convolution layer에 들어가는 weight값을 streaming 형식으로 동시에 받게 되는데 이때 DRAM과 FPGA의 인터페이스에서 충돌이 일어난 것으로 보인다. 따라서 이러한 충돌을 없애기 위해 kernel 용 buffer를 생성해서 미리 FPGA에 저장한 후 연산을 하면 이러한 문제가 해결될 것이라 예상하고 있다.

7.2 padding

5- layer configuration에서 layer 간 연산을 할 때, streaming 형식으로 data padding을 진행하는 것을 구현하지 못하였다. 따라서 padding을 하기 위해 buffer에 streaming data를 모두 넣고 padding을 진행하여 다음 layer에 다시 streaming 형태로 data를 넣어주는 형식으로 진행하였는데, 이는 결국엔 추가 buffer를 설계해야 하고 또한 dataflow 설계가 제대로 FIFO 형태로 이루어지지 않는다는 의미이다. 이를 확인해보기 위해 단일 layer에서 output을 배열의 형태로 변경하여 emulation time의 변화가 있는지를 살펴보았다.

Simulation 단축을 위해 Table 1의 layer 5로 진행하였고 확인결과 배열로 모두 받은 뒤에 DRAM에 저장을 해야하기 때문에 연산이 모두 끝난 뒤 DRAM에 write 과정을 추가로 거치는 것을 확인할 수 있었으며(streaming 방식으로는 RD와 WR를 독립적으로 동시에 진행함) 이는 0.2ms 정도의 emulation time 증가로 나타났다(Appendix 9.2 참고). 연산량은 186.1GOPS 로 약 5퍼센트 정도의 성능 감소가 있었으며 더 큰 layer로 진행할 경우 성능저하는 더 커질 것으로 예상된다.

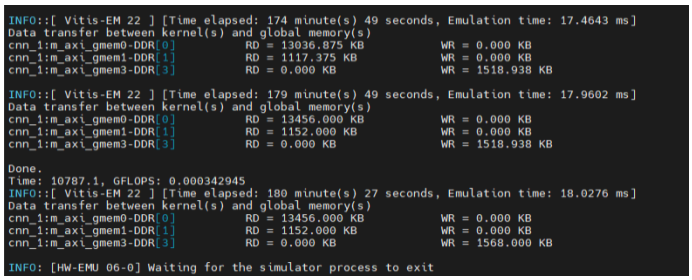
8. Concluding remarks

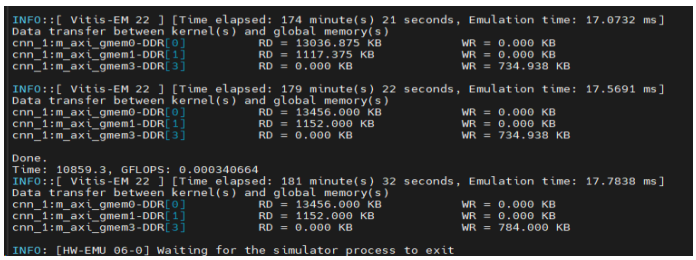
본 project를 통해 소개한 specific layer architecture와 tiled pooling method는 추론용 CNN 가속기의 성능을 높이는 novelty라고 생각한다. Specific layer architecture를 이용하여 모든 layer마다 최적화를 진행 할 수 있었으며 tiled pooling 방식으로 하나의 FPGA에 모든 convolution layer architecture를 설계할 수 있었다. 더불어 Tiled pooling 방식과 dataflow (FIFO) 를 이용해 memory 효율성을 높혀 tiling size를 매우 키울 수 있었다.

9. Appendix

9.1. Single layer configuration hw_emu result capture

Single layer configuration에서 hw_emu 로 얻은 결과의 마지막 부분을 캡쳐하였다. Layer 1~3은 180분까지만 진행되었고, layer4, 5는 정상적으로 완료되었다. 위에서부터 순서대로 layer 1~layer 5의 결과





```
INFO::[ Vitis-EM 22 ] [Time elapsed: 169 minute(s) 32 seconds, Emulation time: 16.7294 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 12826.625 KB           WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 1099.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 342.938 KB

INFO::[ Vitis-EM 22 ] [Time elapsed: 174 minute(s) 32 seconds, Emulation time: 17.2185 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 13247.125 KB           WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 1135.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 342.938 KB

Done.
Time: 10698.2, GFLOPS: 0.000345795
INFO::[ Vitis-EM 22 ] [Time elapsed: 178 minute(s) 49 seconds, Emulation time: 17.6613 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 13456.000 KB           WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 1152.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 392.000 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
```

```
Done.
Time: 9244.87, GFLOPS: 0.000400155
INFO::[ Vitis-EM 22 ] [Time elapsed: 154 minute(s) 17 seconds, Emulation time: 17.8469 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 14400.000 KB           WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4608.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 196.000 KB
```

```
INFO::[ Vitis-EM 22 ] [Time elapsed: 35 minute(s) 33 seconds, Emulation time: 4.23789 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 3697.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4159.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 42.812 KB

Done.
Time: 2359.3, GFLOPS: 0.000391999
INFO::[ Vitis-EM 22 ] [Time elapsed: 39 minute(s) 31 seconds, Emulation time: 4.70707 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 4096.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4608.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 49.000 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
```

9.2. 7.2의 실험 결과

Emulation time이 4.3ms까지 WR를 하지 않음. 그 이후 WR만 진행.

```
INFO::[ Vitis-EM 22 ] [Time elapsed: 44 minute(s) 45 seconds, Emulation time: 4.38195 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 3825.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4303.375 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 0.000 KB

INFO::[ Vitis-EM 22 ] [Time elapsed: 49 minute(s) 45 seconds, Emulation time: 4.93146 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 4096.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4608.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 135.527 KB

Done.
Time: 2979.12, GFLOPS: 0.000310442
INFO::[ Vitis-EM 22 ] [Time elapsed: 50 minute(s) 10 seconds, Emulation time: 4.97964 ms]
Data transfer between kernel(s) and global memory(s)
cnn_1:m_axi_gmem0-DDR[0]          RD = 4096.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem1-DDR[1]          RD = 4608.000 KB            WR = 0.000 KB
cnn_1:m_axi_gmem3-DDR[3]          RD = 0.000 KB               WR = 162.000 KB
```