

2025-Spring Semester
EEE3551 Intelligent System Design and Applications
Assignment 4:
INT1.58 Quantization-Aware CNN Accelerator
for the MNIST Dataset

Department: Physics
Student ID: 2020132002
Name: Hyeonbeen Jeon

June 15, 2025

Contents

1 INT8 Baseline CNN Accelerator	2
1.1 Baseline CNN Model	2
1.2 Entire System Architectures	2
1.3 Strategies to Reduce Latency and Hardware Resources	5
1.3.1 Line Buffer Algorithm of Convolution Cores(PEs)	5
1.3.2 Implementing Max Pooling Operation in a Buffer-like Manner	6
1.3.3 Batch Processing to Reduce Data Transfers between External Memory and BRAM	7
1.4 Block Diagram	8
2 INT1.58 Quantization-Aware CNN Accelerator	8
2.1 Accuracy Drop in Bit Precision Adjustment	8
2.2 Modification of Baseline CNN Model to 1.58-bit QAT Model	9
2.3 Verilog Implementation Method	11
3 Results	13
3.1 PYNQ Driver	13
3.2 INT8 Baseline Model	14
3.3 INT1.58 Quantized Model	14
4 Discussion	15
4.1 Pipelining to Break Critical Paths to Solve Timing Violation Issues	15
4.2 Future Works	16
5 Conclusion	17
6 References	17
7 Appendix: RTL Codes of INT8 Baseline Model	18

1 INT8 Baseline CNN Accelerator

1.1 Baseline CNN Model

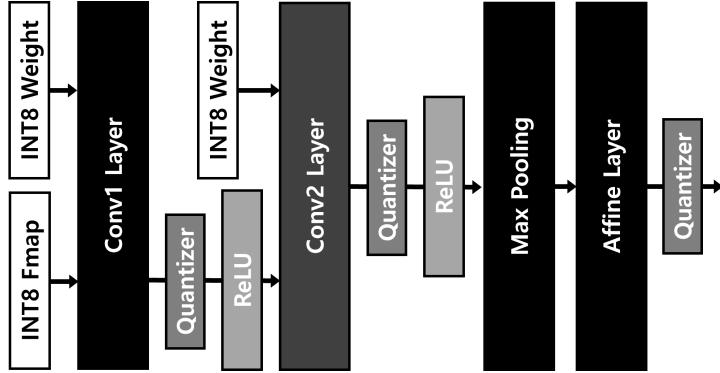


Figure 1: INT8 Quantized Baseline CNN Model

Baseline 가속기 제작을 위해 제공된 CNN 모델의 구조는 Figure 1과 같다. ReLU를 activation function으로 갖는 두 개의 convolution layer와 각각 하나의 max pooling layer(이하 pooling) 및 affine layer(이하 affine)로 구성된 간단한 구조의 CNN 모델이다.

Skeleton으로 제공된 weight와 MNIST 데이터셋은 모두 INT8로 양자화 되어 있었으며, convolution layer 1(이하 conv1)에 제공된 weight는 입력 채널 1, 출력 채널 8의 3×3 filter로 구성되어 있으며, convolution layer 2(이하 conv2)에 제공된 weight는 입력 채널 8, 출력 채널 16의 3×3 filter로 구성되어 있다. 이때, convolution 연산의 stride는 1, padding은 없으며, max pooling 연산의 stride는 2, 2×2 window에 대해 연산을 진행한다. 따라서 conv1에서는 26times26 픽셀의 feature map 8개가 출력되고, conv2에서는 24×24 픽셀의 feature map 16개가 출력되며, pooling에서는 12×12 픽셀의 feature map 16개가 출력된다. 이후 12×12 픽셀의 feature map 16개가 모두 flatten되어 2304개의 요소로 구성된 1D vector가 입력 채널 2304, 출력 채널 10의 affine에서의 weight와 연산되어, 최종적으로 10개의 정수로 구성된 logit이 출력되는 구조이다.

Conv1, conv2, affine 이후에 MAC 연산으로 인해 증가한 bit precision을 다시 INT8로 양자화하는 연산을 거친다. 모두 동일하게 MAC 연산 결과를 오른쪽으로 10-bit shift 한 후, 127을 초과하는 값은 127로, -128 미만의 값은 -128로 clipping하여 INT8로 양자화된 fmap을 얻는다.

1.2 Entire System Architectures

먼저 가능한 적은 리소스(PE 개수, BRAM)를 사용하면서도 준수한 성능과 정확도를 내는 가속기를 목표로 설계 방향성을 정하였다. 최근에 스마트폰과 같은 엣지 디바이스에서의 온디바이스 인공지능 추론이 중요해지고 있기 때문에, 엣지 디바이스의 특성을 고려하여 이러한 방향성을 결정하게 되었다.

Figure 2는 fmap과 weight의 흐름을 중심으로 나타낸 전체 가속기의 구조이다. Conv1, conv2의 모든 weight의 채널이 8 단위로 이루어지기 때문에 입력 채널이 1, 출력 채널이 1인 convolution 연산을 수행하는 PE0부터 PE7까지 총 8개의 PE를 추가해 주었다. 또한 affine에서 weight와 MAC 연산은 2304개의 weight가 총 10번이 연산되어야 하기 때문에 하나로만 수행하면 최소 23040 cycle이 필요한데, 너무 많은 latency를 동반하게 된다. 또한, FC 코어 자체는 단순한 MAC 연산만 수행하기 때문에 적은 utilization만을 요구하므로 총 10개를 만들어 별별 연산을 수행하도록 구성하였다.

각 흐름을 따라 구조를 조금 더 자세하게 살펴보면, 먼저 PS를 통해 받은 입력 이미지 데이터와 conv1, conv2에 필요한 weight 데이터를 모두 BRAM에 저장한다. 이후 8개의 PE에 전달되어 convolution 연산이 수행된 후 각 PE 당 1개의 fmap이 출력되어 총 8개의 fmap이 PL 내부에서만 이용되는 8개의 BRAM(depth = 26×26)에 저장되게 된다. 이렇게 저장된 conv1의 출력 fmap이 총 16번 반복적으로 읽혀서 conv2의 weight와 함께 다시 PE로 전달되고, 각 PE에서 convolution

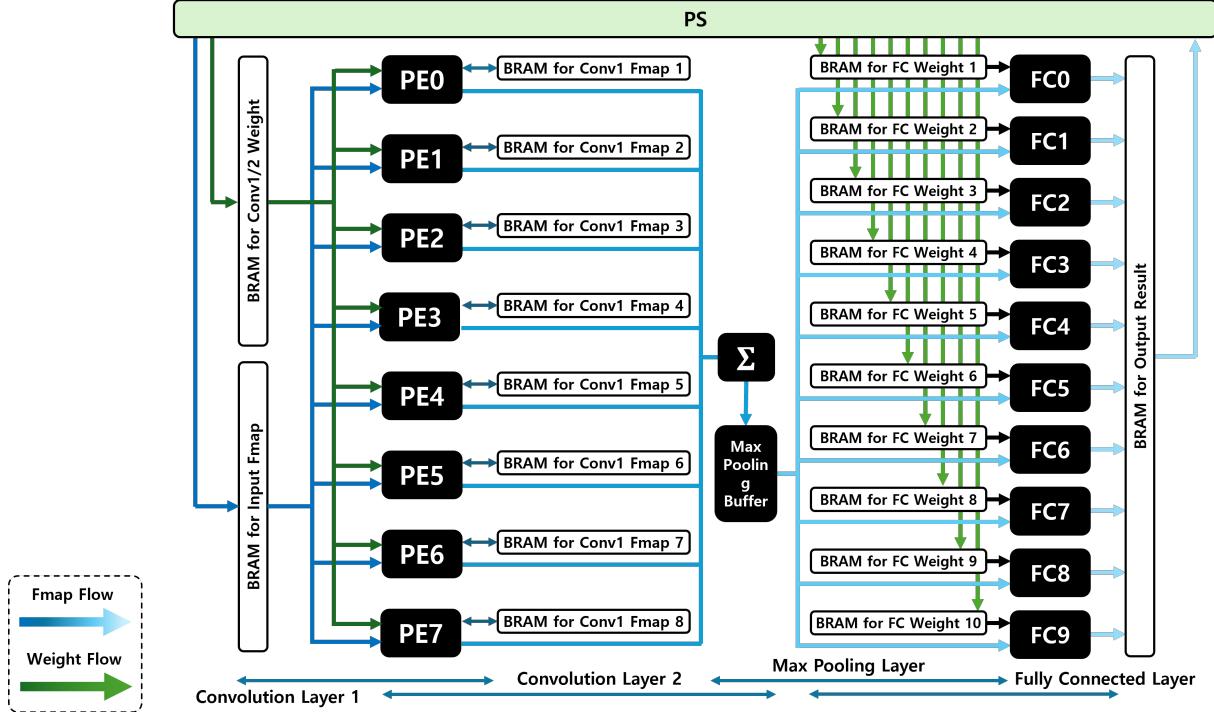


Figure 2: Data Flow Representation of CNN Accelerator

연산이 수행되어 8개의 fmap 데이터가 출력되게 된다. 다만 conv2에서는 3D convolution이 수행되기 때문에 같은 index의 데이터 8개를 합하여 하나의 fmap을 완성하며 순차적으로 총 16개의 fmap이 출력되게 된다.

Conv2에서 순차적으로 출력되는 16개의 data는 추후 자세히 다룰 max pooling buffer로 들어가게 되는데, 이는 max pooling 연산의 특성상 buffer와 유사하기 때문에 max pooling 연산을 수행하는 buffer로 구현한 것이다. buffer를 거친 conv2의 출력 fmap 데이터는 모든 코어에 동일하게 공급되며, 각 코어별로 BRAM에 10개로 나누어 저장되어 있는 affine의 weight 중 2304개의 weight와 함께 FC 코어로 전달되게 된다. 10개의 FC 코어에서 MAC 연산이 완료되면 최종 logit이 출력되게 되고, 총 10개의 INT8 데이터가 최종 출력 결과를 위한 BRAM에 저장된다. 이때 10개의 데이터는 각각 0-9까지의 classification을 의미하며 가장 큰 데이터의 index가 MNIST 손글씨 데이터의 숫자를 판단한 결과가 된다.

Figure 3은 실제 verilog 언어를 통해 각 모듈이 어떤 방식으로 implementation이 되어있는지 대략적으로 보여주는 block diagram이다. 구현한 코드는 이후 7과 2.3에서 주석과 함께 첨부할 예정이므로, 여기에서는 구체적인 동작을 중심으로 살펴보려고 한다.

먼저 top_cnn_accelerator.v라는 top 모듈에 모든 모듈과 필요한 기능들이 implementation되어 있다. 8개의 convolution PE 코어, 10개의 FC 코어, 입력 이미지를 저장할 BRAM 1개, conv1 및 conv2에 필요한 weight를 저장할 BRAM 1개, conv1의 결과를 저장하기 위한 BRAM 8개, affine에 필요한 weight를 나누어 저장할 BRAM 10개, 최종 출력 logit을 저장하기 위한 BRAM 1개가 instantiation 되어 있다. 또한, 연산을 제외한 각 BRAM에 접근할 주소를 제어하는 BRAM controller와 전체 FSM 및 reset을 제어하는 module controller, conv2의 결과가 max pooling buffer로 전달되고, buffer의 출력이 FC 코어로 전달될 때 타이밍을 맞추어주기 위한 timing controller로 구성되어 있다.

Convolution PE는 top_convolution_core.v의 모듈로 구현되어 있으며, width 8, depth 28(conv1, conv2의 연산을 모두 수행할 수 있도록 최대값인 28로 설정)의 FIFO 3개를 이용하여 conv1에서는 입력 이미지 데이터, conv2에서는 내부 BRAM에 저장된 conv1의 출력 fmap을 받아 convolution 연산을 진행하게 된다. 이전의 Assignment 3에서 FIFO 4개를 이용해 fmap을 저장하던 방식을 읽고 쓰는 것이 동시에 가능하도록 FIFO를 개조하여 4개의 FIFO를 3개로 줄여 리소스를 확보할

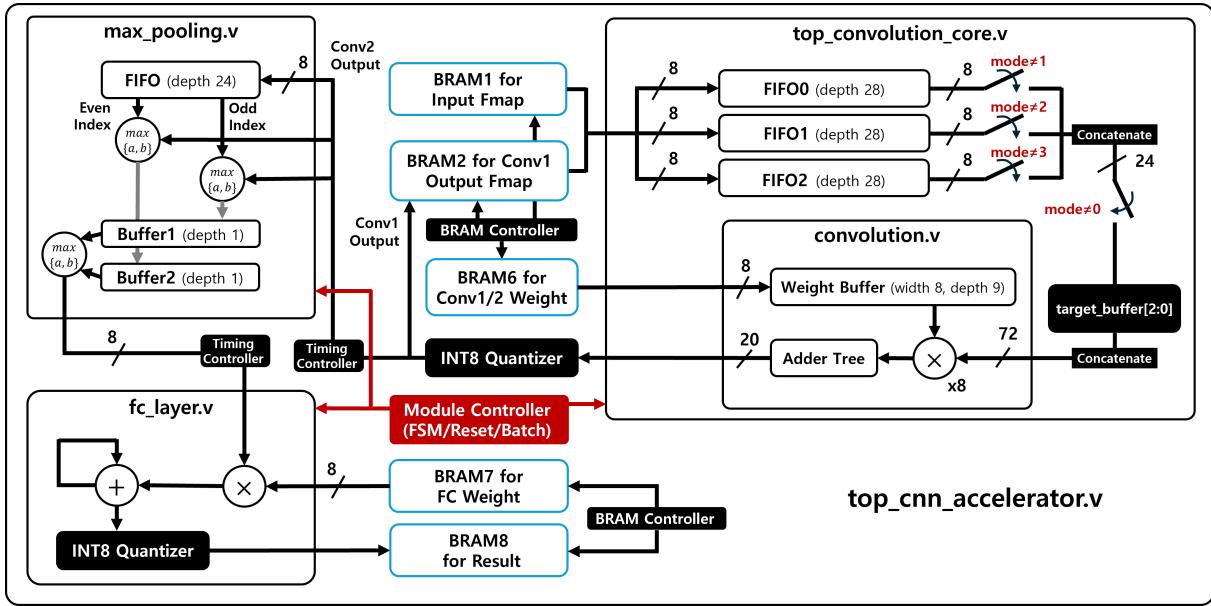


Figure 3: Block Diagram of CNN Accelerator

수 있었다. 자세한 내용은 1.3.1에서 살펴볼 것이다. FIFO에 저장된 데이터들은 각 FIFO에서 선입 선출로 하나씩 읽어들이게 되는데 따라서 한 번에 총 3개의 데이터를 flatten하여 24-bit 데이터로 만들고, 이를 target_buffer라는 이름의 register에 저장하는 과정을 3번 반복하여 총 9개의 픽셀 데이터를 buffer에 담게 된다. 9개의 픽셀 데이터(열마다 flatten된 24-bit 데이터 3개)가 모이면 이를 다시 flatten하여 72-bit 데이터로 만들고, 하나의 3×3 window에 대해 미리 BRAM으로 부터 읽어 준비해둔 weight와 convolution 연산을 수행하는 convolution.v 모듈에 입력으로 전달하여 출력된 20-bit 데이터($8 \times 2 + \lceil \log_2 9 \rceil = 20$)가 출력된다. 이렇게 얻은 convolution PE 8개의 출력 데이터가 conv1에서는 10-bit shift 및 -128-127 clipping, ReLU를 수행하는 INT8 quantizer를 거쳐 8개 각각의 데이터가 내부의 BRAM 8개에 저장되며, conv2에서는 8개의 PE에서 출력된 결과를 더하여 하나의 fmap을 만들고 이를 INT8로 양자화 하여 max pooling 연산 버퍼 모듈에 전달하게 된다.

max_pooling.v 모듈은 max pooling 연산과 동시에 FC 코어로 들어가기 전 버퍼 역할을 수행하는 모듈이다. 먼저 width 8, depth 24의 FIFO 하나를 통해 conv2의 24×24 의 결과 fmap 상에서 홀수번 째 행을 받아 저장한다. 이후 짹수 번째 행을 받을 때에는 따로 FIFO를 선언하는 것이 아니라 짹수 번째 행의 첫 데이터를 FIFO의 처음 들어온 데이터와 최댓값 연산을 통해 depth 1의 버퍼에 저장하고, 두 번째 데이터 역시 FIFO의 두 번째로 들어온 데이터와 최댓값 연산을 통해 또 다른 버퍼에 저장한 뒤, 세 번째 데이터가 들어올 시점에 depth 1의 두 버퍼에 담긴 데이터에 최댓값 연산을 수행하여 max pooling 연산이 수행된 데이터를 출력하는 과정을 반복한다. 즉, 미리 저장된 i 번째 행과 순차적으로 들어오고 있는 $i+1$ 번째 행의 데이터를 비교하여 홀수 번째 열(index 기준으로는 0 포함 짹수)일 경우에는 buffer 1에 저장하고, 짹수 번째 열(index 기준 홀수)일 때에는 buffer 2에 저장한 뒤 다음 홀수 번째 열 데이터가 들어올 때 buffer 1과 buffer 2에 저장된 데이터 중 큰 값을 내보내는 방식으로 PE와 FC 코어 사이의 buffer 역할을 수행함과 동시에 max pooling 연산을 수행하게 된다.

다음으로 max pooling까지 마친 픽셀 데이터가 FC 코어 10개에 모두 동일하게 들어가 내부에 미리 저장된 weight와 MAC 연산이 수행된다. 12×12 크기의 fmap 16개가 모두 완료되어 총 $12 \times 12 \times 16 = 2304$ 개의 데이터에 대해 weight와 행렬 곱 연산이 완료되면 코어당 하나의 logit 데이터가 출력되고, 총 10개의 FC 코어에서 10개의 logit이 출력되어 이중 최댓값을 갖는 데이터의 index가 바로 입력 이미지의 손글씨를 인식한 숫자가 된다.

1.3 Strategies to Reduce Latency and Hardware Resources

앞서 설명한 전반적인 시스템의 구조를 바탕으로 제한된 자원 내에서 입력 이미지의 하나의 CNN 수행 시간을 줄이는 데에 핵심이 된 두 가지 아이디어와, FPGA와 PYNQ Driver 사이에 데이터 교환에 걸리는 시간을 최소화하기 위한 하나의 아이디어에 대해 자세하게 설명하고자 한다.

1.3.1 Line Buffer Algorithm of Convolution Cores(PEs)

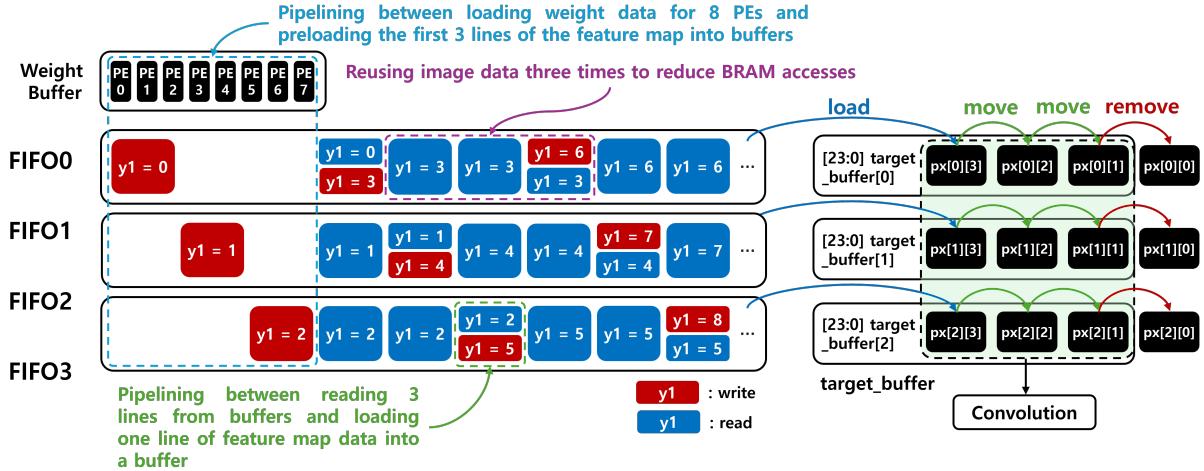


Figure 4: Line Buffer Algorithm for Convolution Cores

Figure 4는 convolution 연산에 걸리는 cycle을 최소화하기 위해 고안한 line buffer algorithm을 그림으로 나타낸 것이다. 총 3가지의 algorithm을 통해 입력 fmap과 3×3 filter의 2D convolution 연산을 최대한 적은 cycle로 마칠 수 있었다.

먼저 fmap의 재활용이다. convolution은 stride가 1이기 때문에 데이터가 중복되는 지점이 매우 많이 존재한다. 먼저 행을 기준으로 보더라도 3×3 filter의 경우에 대해 출력 fmap의 첫 행의 연산과 두 번째 행의 연산에서 입력 fmap의 2행이 그대로 재사용되며, 열을 기준으로 보아도 3×3 window 영역에 대해 다음 영역과 비교하여 열 두 개(6개 데이터)가 중복이 된다. 따라서 열의 중복을 해소하기 위해 출력한 데이터를 바로 지우는 것이 아니라 그림의 오른쪽의 모습처럼 target_buffer에 3개를 저장해두고, 다음 데이터가 FIFO로부터 나오면 가장 오래된 데이터를 지우고 새로운 데이터를 받아오는 방식으로 convolution 연산을 위한 window를 구성하여 fmap을 재사용할 수 있었다. 이보다 더욱 중요한 것으로 행의 관점에서의 재사용이다. 첫 행과 두 행만을 제외하면 모든 행이 총 3번의 재활용이 될 수 있기 때문에, 그림에서도 보는 것처럼 FIFO에 데이터를 한번 불러오게 되면 불러온 데이터를 바로 버리는 것이 아니라 총 3번의 읽기를 반복하였다. 그 결과로 fmap 데이터를 다시 읽어들이지 않고 한 핵심당 한 번만 BRAM에 접근하여 데이터를 가져오기 때문에 많은 cycle을 아낄 수 있었다.

두 번째로 fmap 데이터를 BRAM에서 읽어 line buffer에 불러오는 write stage와, convolution 연산 수행을 위해 buffer에 저장된 fmap 데이터를 읽는 read stage를 pipelining 해주었다. 이전에는 FIFO 4개를 사용하여 이를 구현하였지만, 이번에는 읽고 쓰기가 동시에 가능한 FIFO 3개를 이용하여 fmap 재활용을 위해 3번의 반복 읽기를 하는 와중에도 마지막 3번째 재활용을 수행하는 동시에 데이터를 처음 write한 시점이 가장 오래된 FIFO 순서대로 새로운 입력 fmap의 행을 write하여 read와 write stage를 pipelining 하였다. 이를 통해 convolution 연산을 위해 buffer로부터 fmap 데이터를 불러올 때, fmap 데이터를 buffer에 쓰는 과정을 기다릴 필요가 없으며, BRAM으로부터 입력 fmap을 받아오는 과정에 latency 없이 한 cycle 당 한 핵심씩 쉬지 않고 연속해서 buffer로 읽어들이는 것이 가능하여 cycle을 최대한 줄였다.

세 번째는 weight 데이터를 loading하는 데에 걸리는 latency를 hiding하였다. Convolution 연산을 수행하기 위해서는 fmap을 불러오는 것에서의 cycle을 최소화하는 것 역시 중요하지만, weight

데이터를 불러오는 것도 같이 고려해주어야 한다. 단순하게 생각하면 weight를 먼저 불러오고 fmap을 불러와 convolution을 진행하면 쉽겠지만, weight를 불러오는 것으로 인해 convolution 연산이 시작되는 시점에 latency가 발생하게 된다. 따라서 fmap을 buffer로 write할 때 fmap의 첫 3개의 행을 모두 쓸 때까지는 convolution이 되지 않는데 이를 활용하여 문제를 해결하였다. 첫 3개의 행을 buffer에 쓰기 위해서 conv1의 경우 $28 \times 3 = 84$ cycle이 소요되며, conv2의 경우 $26 \times 3 = 78$ cycle이 소요된다. 그러나 8개의 모든 PE에 대해 8개의 3×3 filter의 데이터 72개를 불러오기에는 충분한 cycle이기 때문에 convolution 연산이 시작되기 전 fmap의 첫 3개의 행을 불러올 때 weight를 같이 불러와 weight로딩에 걸리는 모든 latency를 hiding하여 0으로 만들 수 있었다.

이를 통해 conv1에서 입력 이미지 784 픽셀을 filtering 하는 데에 걸리는 cycle은 822 cycle, conv2에서 입력 이미지 676 픽셀을 filtering 하는 데에 걸리는 cycle은 712 cycle로 각각 한 픽셀 당 1.0485, 1.053 cycle이 소요되었다. 결론적으로 입력 fmap의 1 픽셀 당 약 1.05 cycle로 거의 1 픽셀 당 1 cycle 만에 convolution 연산을 수행할 수 있다.

1.3.2 Implementing Max Pooling Operation in a Buffer-like Manner

먼저 기존에 계획했던 architecture를 설명하고 문제점을 파악해보자. 기존에는 conv2의 결과를 하나의 BRAM에 저장하고, 이를 읽어 max pooling 연산을 코어 하나로 수행한 뒤 다시 하나의 BRAM에 저장한 후 다시 이를 읽어 FC 연산을 코어 하나로 순차적으로 진행하는 방법을 생각했었다. 이렇게 되면 conv2의 결과를 저장하기 위한 BRAM 총 9216-bytes가 필요하며 max pooling 결과 저장하기 위한 BRAM이 2304-bytes로 총 11520-byte의 BRAM 공간이 필요하다. 또한 conv2가 끝날 때까지 기다린 후 max pooling을 진행하며, affine만 해도 최소 23040 cycle이 걸리기 때문에 매우 많은 cycle을 요구하게 된다. 즉, core 수는 적게 사용하지만 이를 위해 필요한 BRAM 공간과 latency가 매우 크다. 따라서 performance를 위해 core를 더 많이 사용하고 BRAM을 잘게 쪼개어 max pooling과 affine layer의 연산을 병렬로 처리하고자 하였다. 그러나 이 방법 또한 max pooling을 위한 코어 16개와, affine을 위한 코어 10개가 필요하고 BRAM 2개를 26개로 쪼개야 하기 때문에 utilization과 복잡성 측면에서 문제가 있었다. 즉, core 수와 BRAM control logic이 매우 많은 대신 latency를 줄이는 방안이었다.

그러나 한정된 자원 하에서 효율을 내는 것이 목표이므로 두 가지 해결책 모두 좋은 방안이 아니었고, 새로운 구조를 찾아내었다. 바로 **max pooling 레이어를 독립된 코어가 아니라 conv2과 affine 사이의 buffer 역할로 사용하여**(이를 max pooling buffer라고 편의상 부르겠다) max pooling과 affine layer에 걸리는 latency를 대폭 줄이면서도 자원 이용을 최소화 하는 방안이었다. Max pooling의 연산 자체의 특성상 4개의 데이터를 받아 하나만 출력하는 buffer의 성격을 갖고 있기 때문에 conv2의 결과와 max pooling의 결과를 따로 저장하여 여러개의 core를 사용하는 것이 아니라 conv2에서 데이터가 나오면 바로 max pooling buffer에 전달되고, max pooling buffer 내부에 FIFO에 저장되는 과정에서 max pooling 연산을 같이 수행한 뒤 FIFO에서 읽어 affine을 위해 FC 코어로 전달하는 방식으로 구현하였다. 이를 위해 필요한 것은 max pooling 버퍼 하나와 conv2에서 max pooling buffer로 데이터가 전달되는 타이밍 및 max pooling buffer에서 FC 코어로 데이터가 전달되는 타이밍을 맞추기 위한 timing controller logic만 간단하게 구현하면 되었다.

결론적으로 **affine에 필요한 core만 10개로 증가시키고 사이에 BRAM을 전혀 사용하지 않으며, conv2 - max pooling - affine 단의 연산을 모두 병렬로 수행하여 latency를 더욱 최소한으로 줄일 수 있었고, PPA(Performance, Power, Area) 측면에서 하나로 치우치는 것이 아니라 3가지 관점에서 모두 효율적으로 설계할 수 있었다.**

Figure 5는 max pooling buffer를 사용하여 최종적으로 모든 CNN 연산에 대해 입력 이미지의 숫자를 추론해 내기 까지의 총 latency를 나타낸 것이다. 다음 conv2가 수행되기 전에 이전 conv2 데이터가 max pooling, affine layer의 연산이 모두 완료되는 것을 확인 할 수 있으며, 총 12327 cycle($123.27\mu s$)로 소요된 것을 확인할 수 있었다. 단순히 연산 시간만을 고려하면 10000장의 데이터에 대해 대략 1.23초 만에 손글씨 추론을 완료할 수 있다.

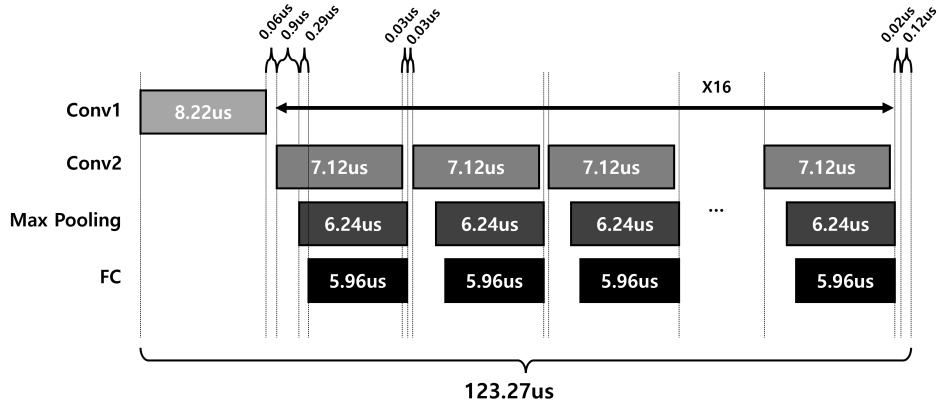


Figure 5: Total Cycles of CNN Operations including Max Pooling Buffer

1.3.3 Batch Processing to Reduce Data Transfers between External Memory and BRAM

마지막으로 PYNQ driver에서 FPGA 입력 이미지 데이터를 보내고, FPGA로부터 PYNQ driver에 출력 logit을 보내는 과정에서 발생하는 병목에 의한 효과를 줄이기 위한 batch processing이다.

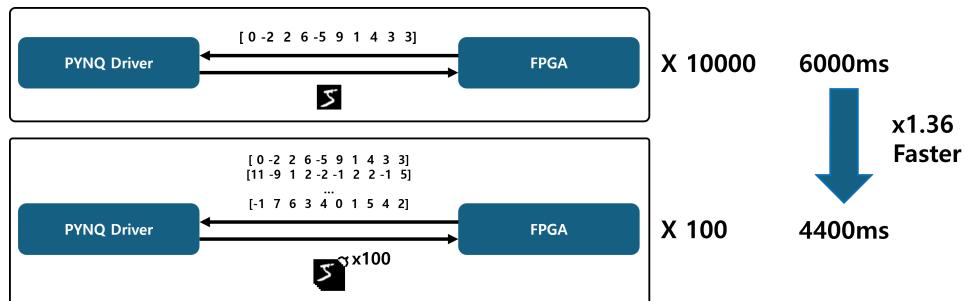


Figure 6: Latency Reduction by Batch Processing

앞서 다룬 설계를 통해 이미지 한장을 따로 처리하는 CNN 가속기를 실제로 검증해 보았을 때, 거의 6초에 달하는 시간이 걸리는 것을 확인할 수 있었다. 단순히 연산에 걸리는 latency가 1.2초에 external memory에서 BRAM으로 데이터를 전송하는 시간을 고려해도 과하게 오래 걸리는 것을 확인할 수 있었는데, 그 원인으로 PYNQ driver 상에서 for문을 이용하여 반복해서 입력 이미지 데이터를 넣어주는 작업을 10000번 반복하면서 발생한 latency라고 판단하였다. 따라서 이를 해소하기 위해 Figure 6와 같이 한번에 100장을 보드에 보내어 100장에 대한 결과 100개를 한번에 받는 batch processing을 구현하였다. FSM에 1장의 이미지가 끝난 state 이후, 완료된 이미지 개수를 세어 100장에 대한 작업 완료되었을 때의 완료 state를 추가하여 구현하였으며, BRAM 접근 주소의 경우에도 입력 이미지가 담긴 BRAM의 주소에 한 장의 이미지가 완료될 때마다 784의 offset을 계속 더해가는 방식으로 구현할 수 있었다.

결론적으로 그림에 나타난 것처럼 for문이 10000번 수행되는 것에 비해 for문이 100번 수행하는 것으로 줄였을 때 4.4초 가량 걸리며 1.36배의 성능 향상을 이룰 수 있었다.

이때, batch size를 더 늘리면 시간이 더 줄거라고 예상하였으나 batch size를 500까지 증가시켰을 때 latency가 거의 동일하여 유의미한 감소를 보이지 않았다. 한 번에 보내는 양이 많아지면서 이로 인해 외부 메모리와 데이터를 주고 받는 시간이 크게 증가하였기 때문으로 생각되며, BRAM 하나에 500장의 데이터를 넣는 것이 아니라 더 조개었다면 유의미한 향상을 보였으리라 기대할 수 있다. 이에 관해서는 4.2에서 더 자세하게 언급할 예정이다.

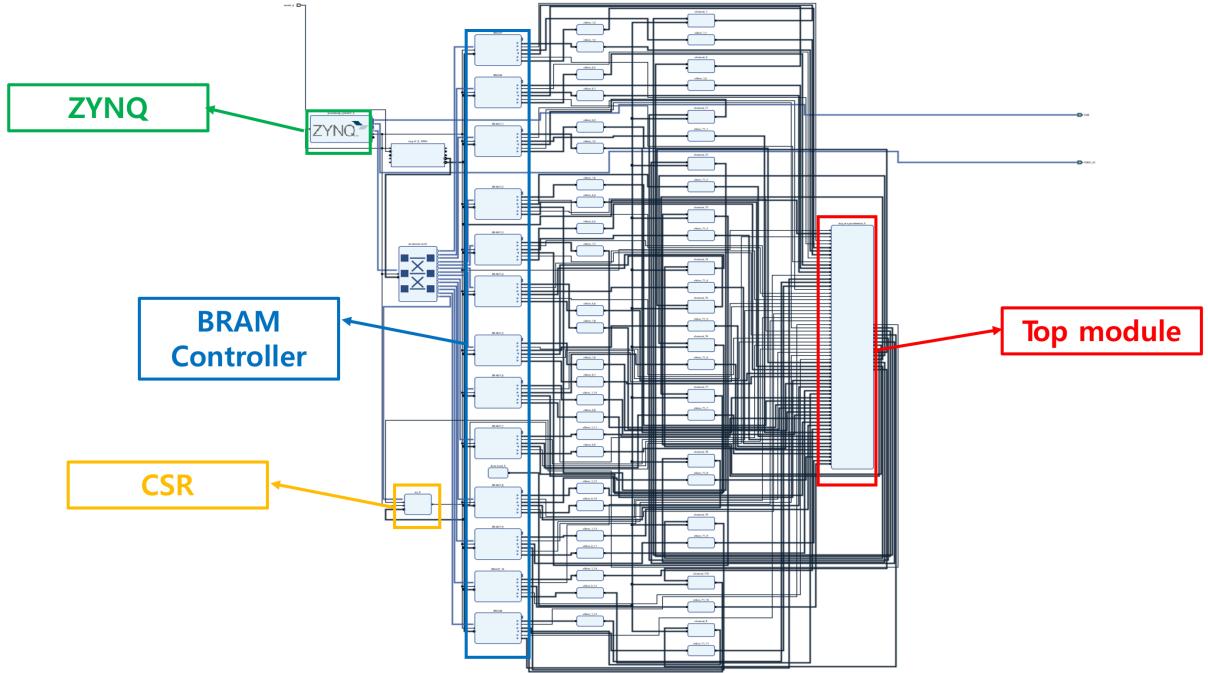


Figure 7: Block Design for CNN Accelerator

1.4 Block Diagram

Figure 7은 설계한 PL을 IP로 packaging하여 block design 결과이다. 초록 색 상자 부분에 ZYNQ PS를 확인할 수 있으며 PS Reset에 외부 switch로 동작하는 active low reset 신호가 들어가고 있는 것을 확인할 수 있다. 또한 받은 이미지 및 weight 데이터를 PL로 전달하기 위해 BRAM Controller 13개가 사용된 것을 확인할 수 있으며(입력 이미지 BRAM 1개, conv1/2 weight BRAM 1개, affine weight BRAM 10개, 출력 데이터 저장용 BRAM 1개), BRAM controller와 PL 사이에 concat, slice, constant block을 적절히 설계하여 BRAM controller와 PL 내부의 BRAM의 입출력 포트 사이에 bit width 차이를 조정해주고 있는 모습이다.

또한 CSR(control/status register)는 이전 assignment와 동일하게 PYNQ driver로부터 start 신호를 받으면 debouncing하여 PL로 전달하는 역할과, PL에서의 done 신호를 CSR에 반영하여 PYNQ driver 상에서 PL에서 이미지 100장의 연산이 완료되었음을 알 수 있도록 한다.

2 INT1.58 Quantization-Aware CNN Accelerator

2.1 Accuracy Drop in Bit Precision Adjustment

처음에는 weight와 fmap의 bit-precision을 LSB를 자르는 방식으로 낮추어가며 정확도를 분석해 보았다. Pytorch를 사용하여 주어진 .npy에 저장된 weight 및 입력 이미지 데이터를 그대로 사용하여 동일한 CNN 모델을 사용하였다. 다만, weight 및 fmap의 하위 N 를 자르는 방식으로 사용하였으며 각각의 N 에 대해 layer 별로 동일하게 10-bit shift 하던 것을 파라미터로 사용하여 accuracy를 최대로 만드는 각 layer 별 shift하는 bit 수를 구하였다. shift bit 수는 0-15까지로 설정했기 때문에 총 $7 \times 15 \times 15 \times 15 = 23675$ 번의 추론을 수행하였다.

이때 accuracy 측정은 INT8과의 비교가 아닌 주어진 정답 label과의 비교를 수행하였으며, 원래 Baseline CNN 모델의 accuracy는 93.49%였다.

Table 1는 각 N 에 대해 최대 accuracy를 구하고, 그 때의 parameter를 정리한 결과이다. 결과를 보면 오히려 weight와 입력 이미지의 bit를 7-bit로 줄이면 accuracy가 기존보다 더 증가하는 것을 볼 수 있으며, 기존에 주어진 CNN 모델의 accuracy가 그리 높지는 않고 학습 자체도 loose하게 진행되었다는 것을 알 수 있다. 또한, weight와 입력 이미지의 bit를 4-bit로 줄이는 순간 84%까지

Bit Precision N (Bits)	Conv1 Shift(bits)	Conv2 Shift(bits)	Affine Shift(bits)	Accuracy(%)
1	15	15	15	10.01
2	7	13	15	33.8
3	8	13	14	67.71
4	8	13	13	84.27
5	8	12	12	90.42
6	9	11	11	93.41
7	9	10	10	93.99

Table 1: Accuracy with Bit Precision Adjustment

떨어지기 때문에 QAT(Quantization-Aware Training) 기법을 적용하여 accuracy를 증가시키고자 하였다.

2.2 Modification of Baseline CNN Model to 1.58-bit QAT Model

QAT를 이용하여 INT1.58 양자화에 성공하였으며, QAT 위해 구성한 train 및 inference 모델 architecture, verilog implementation을 위한 간소화에 대해 설명하고자 한다. 이때, ternary라는 표현 대신 INT1.58이라고 사용한 이유가 있는데, 단순히 ternary라고 언급을 하게되면 INT8과의 직접 대조가 어려운 것이 첫 번째이다. 가장 주된 사유로는 데이터 종류가 3개이면 어차피 INT2로 표현해야 하는데 ternary의 장점이 잘 드러나지 않는다. Ternary(INT1.58) 양자화는 단순히 width가 2-bit 인 것을 넘어 INT2와는 다르게 곱셈 연산에 대해 closed set이라는 것이 가장 중요하다. INT2의 경우에는 -2로 인해 곱셈 연산에 대해 width가 무한히 증가할 수 있지만, ternary의 경우에는 곱셈을 수행해도 -1, 0, 1로 다시 ternary가 된다. 따라서 곱셈 연산이 많은 convolution과 행렬곱 연산에서 큰 이점을 가져갈 수 있기 때문에 INT2와 구분하여 $\log_2 3 = 1.58$ -bit로 표현하여 강조하였다.

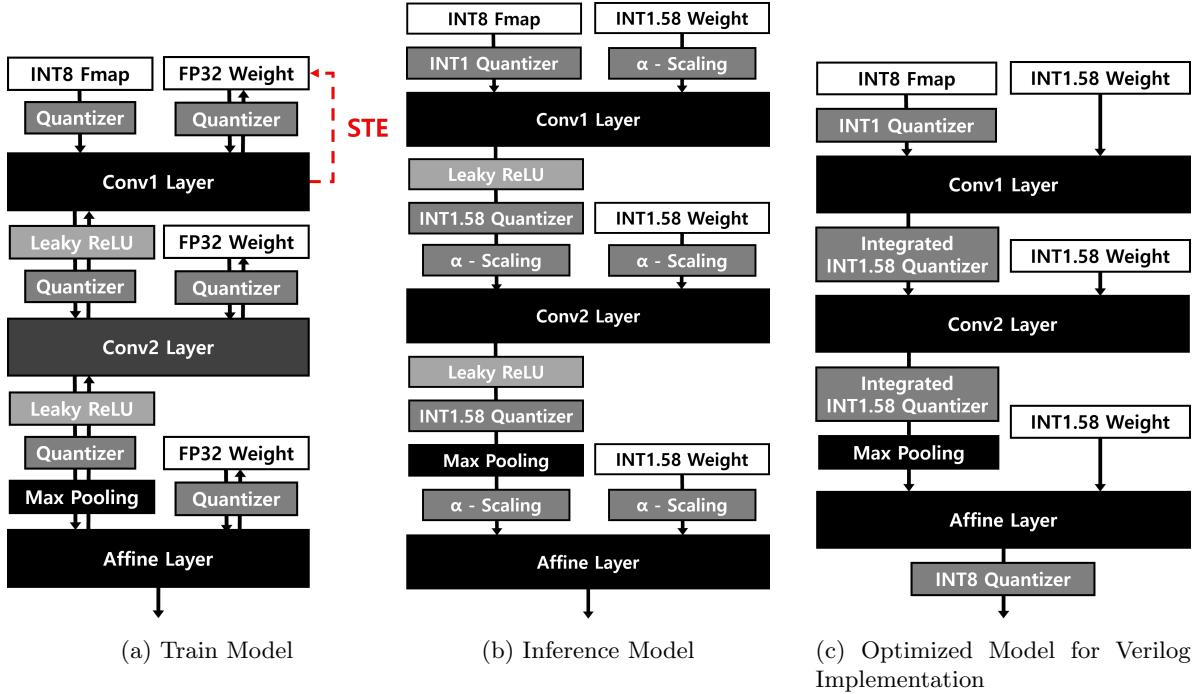


Figure 8: Model Architectures for INT1.58 Quantization

Figure 8은 기존의 Baseline CNN 모델을 ternary quantization을 수행하기 위한 모델의 architecture를 나타낸 것이다. 그림의 train model을 보면 fake quantization 방식을 이용해서 가중치를

학습시켰다. Fake quantization 방식은 먼저 INT8 fmap 및 FP32 weight 데이터를 INT8 형태로 양자화를 수행하고 이를 다시 α 라는 FP32 값을 곱하여 dequantization 하는 방식이다. 이때, FP32 혹은 INT8로 동작하지만 데이터만 INT8 형태로 잠깐 바꿔주어 사용하기 때문에 fake quantization 이라고 불린다. 그러나 역전파의 경우에는 fake quantization을 적용하면 미분이 불가능하기 때문에 STE(straight-through estimate) 방식을 이용해 quantizer를 거치지 않고 by-passing 되도록 해주었다. 이때, Ternary 모델의 경우에는 ReLU가 음수 데이터를 모두 제거하기 때문에 정확도가 떨어질 것이라 판단하여 음수 데이터 정보를 살려주기 위해 Leaky ReLU를 사용하였다. 또한, 입력 이미지에서는 글씨가 있는 부분을 확실하게 강조하기 위해 0인 데이터는 0, 0이 아닌 데이터는 1로 만드는 INT1 quantizaion을 적용하였다.

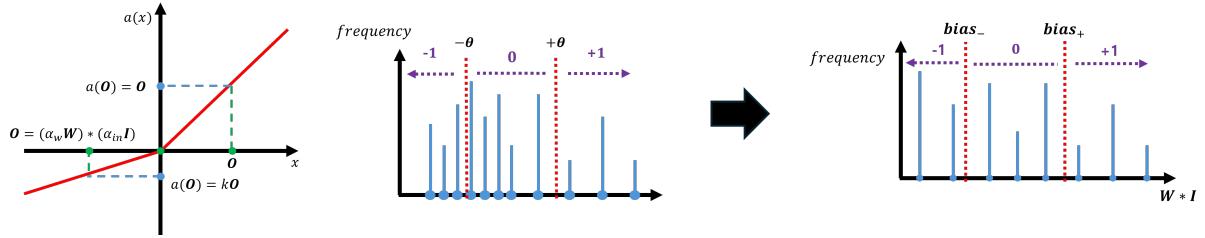


Figure 9: Flow of INT1.58 Quantization

Inference 모델 과정에 대해서도 자세하게 살펴보자. 입력 fmap을 \mathbf{I} , filter를 \mathbf{W} 의 행렬로 나타내면 출력 fmap \mathbf{O} 는 둘의 convolution인

$$\mathbf{O}_{\text{conv}} = \mathbf{I} * \mathbf{W} \quad (1)$$

이다.

이때 입력 fmap 및 weight에 각각 α_{in} , α_w 이라는 scaling을 부여하면 출력 fmap은

$$\mathbf{O}_s = (\alpha_{in} \mathbf{I}) * (\alpha_w \mathbf{W}) \quad (2)$$

Conv1, 2 layer의 activation으로 다음과 같이 음수 영역에서 $k > 0$ 의 slope를 갖는 leaky ReLU를 사용하면

$$\text{Leaky ReLU}(x) = \begin{cases} x & , x \geq 0 \\ kx & , x < 0 \end{cases} \quad (3)$$

$$\mathbf{O}_{ij} = \begin{cases} [(\alpha_{in} \mathbf{I}) * (\alpha_w \mathbf{W})]_{ij} & , (\mathbf{O}_s)_{ij} \geq 0 \\ k[(\alpha_{in} \mathbf{I}) * (\alpha_w \mathbf{W})]_{ij} & , (\mathbf{O}_s)_{ij} < 0 \end{cases} \quad (4)$$

Ternary 양자화를 위한 threshold 값을 θ 라고 하면 ternary 양자화된 최종 출력 fmap은 다음과 같은 함수로 결정된다.

$$(\mathbf{O}_{\text{INT1.58}})_{ij} = \begin{cases} 1 & , (\mathbf{O})_{ij} > \theta \\ 0 & , -\theta \leq (\mathbf{O})_{ij} \leq \theta \\ -1 & , (\mathbf{O})_{ij} < -\theta \end{cases} \quad (5)$$

그러나 위 식은 verilog implementation 시에 구조가 복잡해질 가능성이 높으므로 모든 상수 값을 모아 다음과 같이 단순화 하였다. 이를 적용한 구조는 8의 (c)에 해당하며, 원래의 CNN 모델과 동일한 수의 계층을 갖도록 만들 수 있었다.

$$(\mathbf{O}_{\text{INT1.58}})_{ij} = \begin{cases} 1 & , (\mathbf{I} * \mathbf{W})_{ij} > \frac{\theta}{\alpha_{in} \cdot \alpha_w} \\ 0 & , -\frac{\theta}{k \cdot \alpha_{in} \cdot \alpha_w} \leq (\mathbf{I} * \mathbf{W})_{ij} \leq \frac{\theta}{\alpha_{in} \cdot \alpha_w} \\ -1 & , (\mathbf{I} * \mathbf{W})_{ij} < -\frac{\theta}{k \cdot \alpha_{in} \cdot \alpha_w} \end{cases} \quad (6)$$

따라서 $bias_- = -\frac{\theta}{k \cdot \alpha_{in} \cdot \alpha_w}$ 값과 $bias_+ = \frac{\theta}{\alpha_{in} \cdot \alpha_w}$ 값을 QAT를 통해 구하여 verilog 상에서 레지스터로 저장해두면, α -scaling + Leaky ReLU + INT1.58 Quantizer를 하나로 통합하여 구현이 가능하다. 이때, $bias$ 값은 verilog implementation 시에 구현의 편의성 및 고정 소수점 width를 줄이고자 최대한 간단한 값을 hyperparameter로 설정해주었다. 설정한 값은 아래와 같다.

$$\text{Conv1: } bias_+ = 0.25 \quad bias_- = -0.5$$

$$\text{Conv2: } bias_+ = 0.625 \quad bias_- = -1, 25$$

$$\begin{bmatrix} -1 & 0 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ 1 & -1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 1 & 1 \\ -1 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & -1 & 1 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & 0 \end{bmatrix}$$

(a) Trained Weight for Conv1 Layer

Accuracy: 95.38%

[Inference 결과]

logits : [[-19 -34 6 ... -17 -13 -3]
[95 -59 21 ... -21 21 22]
[-46 17 -24 ... 6 2 22]
...
[3 8 -15 ... -46 2 -10]
[-45 -15 -34 ... 31 23 78]
[-23 2 -2 ... 58 6 51]]

(b) Accuracy

Figure 10: INT1.58 QAT Result

Figure 10는 선정한 hyperparameter를 통해 학습을 진행한 결과 얻은 conv1의 weight 예시이다. 모두 -1, 0, 1의 값을 갖고있는 것을 확인할 수 있으며, 정답률 또한 95.38%로 1.58-bit 모델임에도 원래의 8-bit 모델보다 더 높은 정답률을 얻어낼 수 있었다.

2.3 Verilog Implementation Method

Verilog implementation을 위해 간소화된 inference 모델 구조를 사용하게 되면 가장 큰 이점이 baseline 모델에서 바꿔줄 부분이 bit width만을 제외하면 Quantization 부분만 바꾸어주면 된다는 장점이 있다. 기존 모델에도 INT8 Quantization(10-bit shift right, -128-127 clipping)을 ReLu와 통합하여 구현했었기 때문에, 이 부분만 Integrated INT1.58 Quantizer로만 바꿔어주면 된다.

이때, bias와의 비교를 통해 ternary 데이터로 바꿔주기 위해서는 bias가 소수 데이터이기 때문에 표현도 어렵고 직접 비교가 불가능하다. 이때 부동소수점은 연산이 여려우므로 고정소수점 Q-format을 통해 단순히 convolution PE로부터 나온 데이터를 shift left 연산만으로 직접 비교가 가능하도록 해주었다.

Listing 1: Integrated INT1.58 Quantizer

```

1 // Quantizer
2 localparam Q_SHFT1 = 2, Q_SHFT2 = 3;
3 reg signed [5+Q_SHFT1:0] dout_conv1_Q [7:0];
4 reg signed [1:0] dout_conv1 [7:0];
5
6 reg signed [5+Q_SHFT2:0] dout_conv2_Q;
7 reg signed [1:0] dout_conv2;
8
9 // Quantization (Ternary) + Leaky Relu
10
11 wire signed [7:0] bias1_pos = {{4{1'b0}},4'b0001}; // 0.25 (00.01)
12 wire signed [7:0] bias1_neg = {{4{1'b1}},4'b1110}; // -0.50 (11.10)
13 wire signed [11:0] bias2_pos = {{7{1'b0}},5'b00101}; // 5/8 (00.101)
14 wire signed [11:0] bias2_neg = {{7{1'b1}},5'b10100}; // -5/4 (10.110)
15
16 wire signed [1:0] ter_pos = 2'b01, ter_neg = 2'b11;

```

```

17
18 // Q format
19 always @(posedge clk) begin
20   if(!resetn) begin
21     dout_conv1_Q[0] <= 0; dout_conv1_Q[1] <= 0; dout_conv1_Q[2] <= 0;
22     dout_conv1_Q[3] <= 0;
23     dout_conv1_Q[4] <= 0; dout_conv1_Q[5] <= 0; dout_conv1_Q[6] <= 0;
24     dout_conv1_Q[7] <= 0;
25     dout_conv2_Q <= 0;
26   end else begin
27     if(state == CONV1) begin // Q_shift
28       dout_conv1_Q[0] <= PE_dout[0] <<< Q_SHFT1;
29       (중략)
30       dout_conv1_Q[7] <= PE_dout[7] <<< Q_SHFT1;
31     end else if(state == CONV2) begin // sum & Q_shift
32       dout_conv2_Q <= (PE_dout[0] + PE_dout[1] + PE_dout[2] + PE_dout
33 [3] + PE_dout[4] + PE_dout[5] + PE_dout[6] + PE_dout[7]) <<< Q_SHFT2;
34     end
35   end
36   dout_conv1[0] <= 0; dout_conv1[1] <= 0; dout_conv1[2] <= 0;
37   dout_conv1[3] <= 0;
38   dout_conv1[4] <= 0; dout_conv1[5] <= 0; dout_conv1[6] <= 0;
39   dout_conv1[7] <= 0;
40   dout_conv2 <= 0;
41   end else begin
42     if(state == CONV1) begin // saturation + relu
43       dout_conv1[0] <= (dout_conv1_Q[0] > bias1_pos) ? ter_pos : (
44       dout_conv1_Q[0] < bias1_neg) ? ter_neg : 0;
45       (중략)
46       dout_conv1[7] <= (dout_conv1_Q[7] > bias1_pos) ? ter_pos : (
47       dout_conv1_Q[7] < bias1_neg) ? ter_neg : 0;
48
49     end else if(state == CONV2) begin // saturated + relu
50       dout_conv2 <= (dout_conv2_Q > bias2_pos) ? ter_pos : (
51       dout_conv2_Q < bias2_neg) ? ter_neg : 0;
52     end else begin
53       dout_conv1[0] <= 0; dout_conv1[1] <= 0; dout_conv1[2] <= 0;
54       dout_conv1[3] <= 0;
55       dout_conv1[4] <= 0; dout_conv1[5] <= 0; dout_conv1[6] <= 0;
56       dout_conv1[7] <= 0;
57       dout_conv2 <= 0;
58     end
59   end
60 end
61
62 wire signed [1:0] s1_dout_ternary;
63 assign s1_dout_ternary = (s1_dout > 0) ? 2'b01 : 2'b00;

```

Listing 1는 baseline 모델의 quantization을 위한 always 구문 2개를 변형하여 INT1.58로 바꾸어 준것이다. INT8에서는 conv1의 경우 10-bit shift right, conv2에서는 8개의 데이터를 합한 후 10-bit shift right 수행하던 첫 번째 always 구문을 소수의 크기 비교를 위해 Q format으로 shift left 해주는 always 구문으로 변형해주었다. Bias 값들이 conv1에서는 고정소수점 Q2.2로 표현되며, conv2에서는 Q2.3으로 표현되기 때문에 conv1에서는 2-bit만큼, conv2에서는 데이터 8개를 합한 후 3-bit만큼 shift left를 수행하였다. 두 번째 always 구문의 경우에는 INT8 baseline 모델에서 -128에서 127 clipping 및 ReLu를 적용하던 것을 bias와 Q-format으로 대소 비교하는 연산을 수행해주는 것으로 대체함으로써 간단하게 구현할 수 있었다.

Listing 2: Reduction of Bit Width

```

1 // define kernels (-2^2)
2 reg signed [1:0] gx [8:0];
3
4 // kernel sum
5 reg signed [2:0] sum1[4:0]; // -2 ~ 2 : int2
6 reg signed [3:0] sum2[2:0];
7 reg signed [4:0] sum3[1:0];
8 reg signed [5:0] sum; // -9 ~ 9 : int6
9 reg signed [1:0] gx_mul [8:0]; // ternary
10 reg [3:0] index = 0;

```

Listing 2는 convolution.v 모듈의 일부로 INT1.58 양자화의 효과로 register에서 나타난 bit width의 감소 효과를 보여주는 예시이다. 원래 최종 sum 결과가 20-bit였던 것에 비교하여 6-bit로 큰 감소효과를 보인 것을 확인할 수 있다. 이 외에도 BRAM 입출력 포트나 모듈 입출력 포트 등에서도 큰 감소 효과가 발생하였으며 BRAM 자체의 width도 2로 줄면서 utilization 측면에서 많은 감소 효과를 기대할 수 있다.

3 Results

3.1 PYNQ Driver

Listing 3: PYNQ Driver

```

1 input_fmap = np.load("input.npy").flatten()
2 conv1_weight = np.load("layer1_0_weight.npy")
3 conv2_weight = np.load("layer2_0_weight.npy")
4 fc_weight = np.load("fc1_weight.npy")
5 conv12_weight = np.concatenate((conv1_weight.flatten(), conv2_weight.flatten()))
6 answer = np.load("label.npy")
7
8 class PynqTestDriver:
9     def __init__(self, bitfile_path):
10         self.hw = Overlay(bitfile_path)
11         self.csr = self.hw.csr_0.mmio.array
12         self.ifmap = self.hw.BRAM1.mmio.array
13         self.conv12_w = self.hw.BRAM6.mmio.array
14         self.fc_w1 = self.hw.BRAM7_1.mmio.array
15         self.fc_w2 = self.hw.BRAM7_2.mmio.array
16         self.fc_w3 = self.hw.BRAM7_3.mmio.array
17         self.fc_w4 = self.hw.BRAM7_4.mmio.array
18         self.fc_w5 = self.hw.BRAM7_5.mmio.array
19         self.fc_w6 = self.hw.BRAM7_6.mmio.array
20         self.fc_w7 = self.hw.BRAM7_7.mmio.array
21         self.fc_w8 = self.hw.BRAM7_8.mmio.array
22         self.fc_w9 = self.hw.BRAM7_9.mmio.array
23         self.fc_w10 = self.hw.BRAM7_10.mmio.array
24         self.omem = self.hw.BRAM8.mmio.array
25
26     def start(self, input_fmap, conv1_weight, conv2_weight, fc_weight):
27         result = np.empty((100000), dtype=np.int8)
28         start_time = time.time()
29         self.conv12_w[0:1224] = conv12_weight
30         self.fc_w1[0:2304] = fc_weight[0]
31         self.fc_w2[0:2304] = fc_weight[1]
32         self.fc_w3[0:2304] = fc_weight[2]
33         self.fc_w4[0:2304] = fc_weight[3]
34         self.fc_w5[0:2304] = fc_weight[4]
35         self.fc_w6[0:2304] = fc_weight[5]
36         self.fc_w7[0:2304] = fc_weight[6]

```

```

37     self.fc_w8[0:2304] = fc_weight[7]
38     self.fc_w9[0:2304] = fc_weight[8]
39     self.fc_w10[0:2304] = fc_weight[9]
40     for i in range(100):
41         self.ifmap[0:78400] = input_fmap[0+78400*i:78400*(i+1)]
42         self.csr[1] = 1
43         while (self.csr[0] == 0):
44             pass
45         result[1000*i:1000*(i+1)] = np.array(self.omem[0:1000])
46     end_time = time.time()
47     runtime = end_time - start_time
48     print(f"Runtime: {runtime*1000:.3f}ms")
49     return result.reshape((10000, 10))

```

Listing 3과 같이 .npy 데이터를 불러와 weight 데이터를 BRAM의 형태에 맞게 순서는 바꾸지 않고 flatten 및 slicing만 하여 PL에 보내고, 한번에 100개의 이미지를 PL로 보내어 완료 신호를 받으면 다음 이미지를 보내는 방식의 PYNQ driver를 구성하여 전체 시간 및 출력 결과를 받아볼 수 있도록 하였다.

3.2 INT8 Baseline Model

```

[6]: result = hw.start(input_fmap, conv1_weight, conv2_weight, fc_weight)
      Runtime: 4363.858ms
[8]: print("Comparizon with output.npy")
      cnt = 0;
      for i in range(10000):
          if(int(np.array_equal(result[i], output[i])) == 1):
              cnt += 1;
      print(f"Accuracy: {cnt/100:.2f}%")
      print("-----")
      print("Comparizon with label.npy")
      cnt = 0;
      for i in range(10000):
          if(np.argmax(result[i]) == answer[i]):
              cnt += 1;
      print(f"Accuracy: {cnt/100:.2f}%")
[9]: print(result)
      Comparizon with output.npy
      Accuracy: 100.00%
      -----
      Comparizon with label.npy
      Accuracy: 93.49%
[[ 0 -2  2 ...  4  3  3]
 [11 -9  1 ...  2 -1  5]
 [-7 -4 -1 ...  2  1 -3]
 ...
 [-1  2  2 ... -4  1 -2]
 [-5 -1  0 ...  4  3  8]
 [ 4  0  0 ...  9  3  5]]

```

Figure 11: FPGA Result of INT8 Baseline Model

Figure 11은 bitstream 생성 후 PYNQ driver를 이용하여 MNIST 데이터셋을 추론한 결과이다. 스켈레톤으로 제공된 output.npy의 logit 데이터와의 비교에서는 100% 일치로 CNN 가속기 설계가 과제 명세서의 요구대로 잘 설계된 것을 검증해주고 있으며, label.npy에 저장된 정답 레이블과도 비교해 보았을 때 제공된 weight 데이터로 추론한 실제 정답률은 93.49%에 달하는 것도 확인할 수 있었다. 최종적으로 10000장 처리에 걸린 시간은 약 4.364초 정도인 것을 확인할 수 있다.

Figure 12는 Implementation 결과 나타난 Timing, Power, Utilization report이다. Timing violation도 모두 문제 없으며 power는 1.679W 정도, utilization은 LUT가 convolution PE 및 BRAM controller로 인해 조금 많이 사용된 것을 제외하면 모두 충분히 적은 자원만으로 효율이 높은 결과를 얻었음을 확인할 수 있다.

3.3 INT1.58 Quantized Model

Figure 13은 INT1.58 quantization aware 모델의 FPGA 결과이다. Latency 측면에서는 거의 변한 것이 없지만, accuracy가 95.36%로 baseline 모델의 93.49%보다 더 높은 정확도를 보이고 있다.

Figure 14는 Implementation 결과 나타난 Timing, Power, Utilization report이다. Timing 측면에서 조금 더 여유가 생겼으며, Power 및 Utilization 측면에서 많은 이점을 보이고 있다. 특히 utilization에서 LUT 이용률이 46%에서 32%로 크게 줄었으며 PL 영역에서 동적으로 사용하는 power(Clocks, Signals, Logic, BRAM)이 0.278W에서 0.187W로 32%의 향상을 보이고 있다. 이를 통해 Quantization을 사용하여 한정된 자원 내에서 준수한 성능이라는 최종 목표에 훨씬 가까워졌음을 알 수 있었다.

Timing Report

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.274 ns	Worst Hold Slack (WHS): 0.019 ns	Worst Pulse Width Slack (WPWS): 3.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 78690	Total Number of Endpoints: 78690	Total Number of Endpoints: 26689

All user specified timing constraints are met.

Power Report

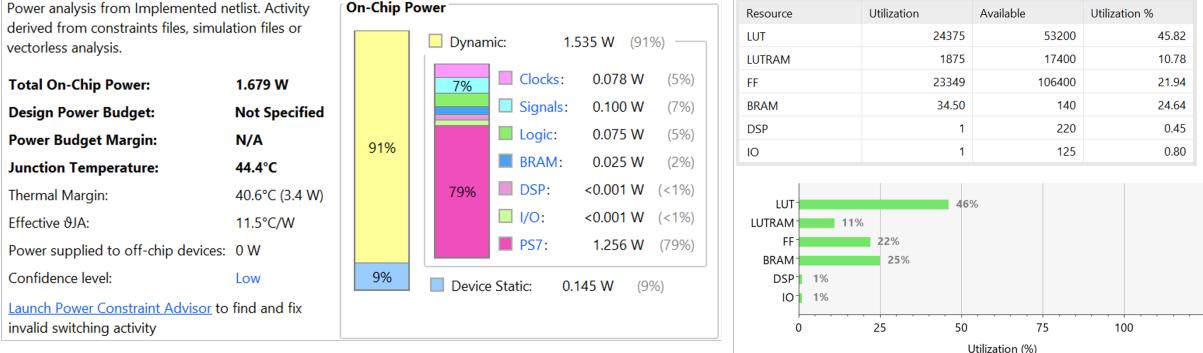


Figure 12: Timing, Power, Utilization Report of INT8 Baseline Model

```
[4]: hw = PynqTestDriver('npu_batch100_ternary.bit')
[5]: result = hw.start(input_fmap, conv1_weight, conv2_weight, fc_weight)
      Runtime: 4366.651ms
[6]: cnt = 0;
      for i in range(10000):
          if(np.argmax(result[i]) == answer[i]):
              cnt += 1;
      print(f"Accuracy: {cnt/100:.2f}%")
[7]: print(result)
      Accuracy: 95.36%
      [[-19 -34 6 ... -17 -13 -3]
       [ 95 -59 21 ... -21 21 22]
       [-47 15 -22 ... 5 2 23]
       ...
       [ 3 8 -15 ... -46 2 -10]
       [-47 -15 -32 ... 30 22 79]
       [-23 2 -2 ... 58 6 51]]
```

Figure 13: FPGA Result of INT1.58 Quantization-Aware Model

4 Discussion

4.1 Pipelining to Break Critical Paths to Solve Timing Violation Issues

처음 baseline model을 수행할 때, 다음과 같이 timing violation 문제가 크게 발생했었다. 다양한 모듈에서 critical path가 많이 발생하여 문제가 됨을 알 수 있었다. 이를 끊어주기 위해 모듈 구조를 갈아엎거나 수정을 많이 진행해 주었는데 정리하면 다음과 같다.

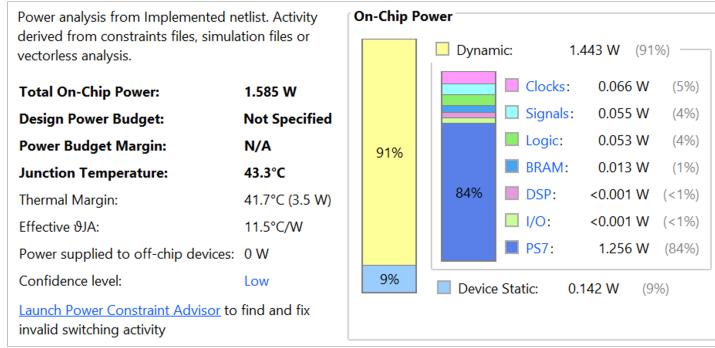
1. Convolution Core: convolution 연산을 수행하기 위해 9개의 데이터를 합하는 과정에서 carry4가 많이 발생하며 critical path가 발생하는 것을 확인할 수 있었다. 이를 해소하기 위해 모든 덧셈 과정을 adder tree로 구현하여 timing violation을 제거하였다.
2. Max Pooling Buffer: 처음에 설계되어 있던 코드는 버퍼 3개를 이용하여 최댓값 연산을 combinational 회로로 3번의 비교를 한번에 수행하는 과정에서 critical path가 발생하였다고 판단하였다. 따라서 buffer 2개만 사용하여 buffer에 데이터를 저장할 때부터 각각 1번씩 2번의 비교 연산을 수행하여 buffer에 저장하고, 그 다음 사이클에 buffer에 저장된 2개의 데이터를 비교하는 방식으로 바꾸어 combinational로 1번에 수행되던 3개의 비교 구문을 3 cycle로 쪼갠 뒤 연산 결과 출력을 한 사이클 delay를 가해 해결하였다.
3. FC Core: MAC 연산이 단순히 할당 구문으로 $sum = sum + weight * fmap$ 과 같은 형식으로 수행되고 있었는데, 곱하기 연산과 더하기 연산의 cycle을 둘로 쪼개어 critical path를 끊어줄 수 있었다.

Timing Report

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.291 ns	Worst Hold Slack (WHS): 0.028 ns	Worst Pulse Width Slack (WPWS): 3.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 54584	Total Number of Endpoints: 54584	Total Number of Endpoints: 18394

All user specified timing constraints are met.

Power Report



Utilization Report

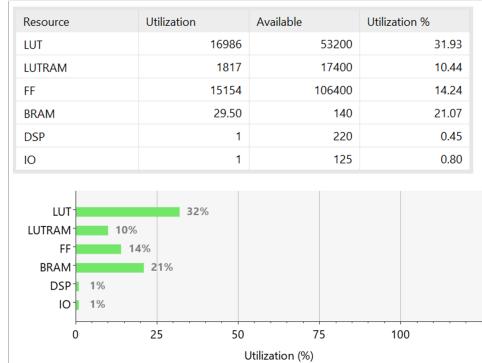


Figure 14: Timing, Power, Utilization Report of INT1.58 Quantization-Aware Model

WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM
-1.603	-182.841	0.017	0.000	0.000	1.773	0	25335	22999	15.5

Figure 15: Timing Violation before Breaking Critical Paths

- Batch Processing: 100개 중 i번째 이미지를 처리한다고 i번째 이미지의 BRAM 접근 주소는 원래 주소에 비해 $784*(i-1)$ 이 더해져야 한다. 그러나 이를 식 그대로 구현해 놓으면 784라는 큰 수와 곱해져야 하기 때문에 DSP로 합성이 되고 있었고 이로 인해 critical path가 발생하였다. 따라서 이를 하나의 이미지가 처리될 때마다 784를 미리 더해두어서 batch processing의 주소 연산에 필요한 시간을 전체 processing 시간에 고르게 분배함으로써 timing violation을 제거할 수 있었다.

위의 과정을 통해 설계한 가속기 전반에 있던 critical path를 모두 제거하는 것에 성공했고, WNS를 양수로 만들어 TNS를 0으로 만들 수 있었다.

4.2 Future Works

지금까지 구현한 CNN 가속기 설계에 이어서 더욱 자원 효율적인 CNN 가속기 설계를 위해 추후 연구로 개선해볼 점을 정리하면 다음과 같다.

먼저 batch size에 관한 논의이다. 앞서 batch processing에 대해 설명할 때 100개와 500개의 batch size에 대해 테스트를 해보았고, 소요되는 latency가 거의 차이가 나지 않았다고 언급했었다. 그러나 100개(BRAM 사용률 25% 내외)와 500개(BRAM 사용률 80%)라는 숫자는 차이가 많이 난다. 따라서 250개와 같이 100개와 500개 사이에 파이썬 코드 상에서 반복되는 for문의 수와 한 번에 전송하는 이미지 데이터의 양 사이에서 적절한 균형을 찾아 local minimum이 되는 batch size를 구하면 latency를 더 극적으로 줄이는 것이 가능할 것이라 예상된다.

다음으로 DSP 이용과 관련된 논의이다. Utilization report를 살펴보면, DSP 이용률이 1%로 매우 낮은 것을 확인할 수 있다. 사용된 곳을 살펴보니 곱셈 연산도 아닌 BRAM 접근 주소를 연산하는

과정에서 발생하는 곱셈에 사용된 것을 확인할 수 있었다. 첨부한 RTL 코드를 확인하면 `use_dsp`를 통해 강제로 DSP를 사용하도록 코드를 구성해 보았지만, 여전히 DSP로 합성이 이루어지지 않았음을 알 수 있다. DSP로 합성이 이루어지지 않는 가장 큰 원인으로는 곱셈에서 수행되는 register의 bit width가 DSP를 쓸만큼 크지 않기 때문임을 확인할 수 있다. DSP 사용을 유도하려면 15-bit에서 20-bit 내외의 width를 가진 두 signed register의 곱셈을 구성하면 자연스럽게 DSP가 사용되는 것을 알 수 있으나, 여기서는 timing violation을 해결하고자 adder tree를 구성하는 과정에서 72-bit로 flatten된 데이터를 실제 INT8 픽셀 데이터 9개로 잘라서 각각 곱셈을 수행하고 있기 때문에 INT8 모델에서는 8-bit 데이터 2개의 곱셈이고 INT1.58 모델에서는 심지어 2-bit 데이터 두 개의 곱셈을 수행하기 때문에 오히려 각각에 DSP를 이용하는 것이 비효율적인 상황이다. 이를 개선하기 위해, 여러 개의 convolution window를 하나로 합쳐 한 번에 곱셈 연산을 수행하는 방식으로 개선하면 병렬로 곱셈 연산이 가능하면서도 bit width가 증가하기 때문에 DSP 사용을 유도하고 효율적으로 사용하여 LUT 이용률이나 전력 측면에서 이점을 가져갈 수 있을 것이다.

마지막으로 Quantization 이후에 대한 논의이다. INT8 baseline 모델을 INT1.58의 ternary 양자화에 성공하면서 utilization 및 power 측면에서 상당한 이득을 얻을 수 있었는데, INT1.58 모델의 적은 utilization을 이용하면 성능 향상을 위해 더 많은 것을 구현할 수 있을 것이다.

먼저 BRAM의 batch size를 크게 늘릴 수 있을 것이다. 지금 모델은 INT8의 입력이미지를 가공하면 안된다는 조건을 위해 입력 이미지 100장을 저장하는 BRAM의 width를 모두 8로 설정한 상태이다. 이를 입력 포트로 데이터가 들어가는 시점에서 INT1 양자화를 수행하여 BRAM에 저장한다면 width를 2로 설정하더라도 BRAM 이용률이 11%까지 떨어지는 것을 확인할 수 있었다. 따라서 batch 사이즈를 1000장까지도 늘려볼 수 있으리라 기대되며, INT1 양자화는 0이면 0이고 안0이 아니면 1로 수행하는 조건문 하나만 필요하기 때문에 PYNQ driver 상에서 INT1 양자화를 미리 진행하면 한번에 많은 수의 입력 이미지를 전송하는 과정에서 발생하는 latency가 감소하여 batch processing의 효과가 극대화 될 것이다.

다음으로 입력 이미지의 분산 추론 시스템의 구현이다. LUT 이용률이 31%로 감소했기 때문에, 전체 시스템을 최대 3개까지도 instantiation하는 것이 가능할 정도로 LUT가 감소했다. 따라서 더 상위의 top 모듈을 만들고 그곳에 전체 CNN 가속기를 멀티 코어처럼 2-3개를 instantiation하여 여러 장의 이미지를 병렬로 추론하는 가속기 설계도 가능해질 것이다. 마지막으로 ternary 양자화도 충분히 많은 리소스를 절약했지만, binary 혹은 XNOR 양자화에도 도전하여 더 제한된 자원에서 구동되는 가속기 설계에도 도전하고 싶다.

5 Conclusion

이번 가속기 설계 프로젝트를 하면서 팀 내에서 다양한 부분의 설계에 참여하면서 많은 것을 배우고 경험할 수 있었던 것 같다. 나에게 할당되어 담당한 부분으로는 convolution core 및 conv1, conv2의 설계와 convolution core, max pooling buffer와 FC core에서 기존의 알고리즘 개선을 통해 critical path를 끊어주어 timing violation issue를 제거하는 부분을 담당하였다. 또한, 팀원과 함께 참여한 부분으로는 FPGA 디버깅 작업, batch processing 설계, INT1.58 quantization aware 모델의 verilog implementation 또한 참여하여 같이 설계하였다. 이렇게 다양한 주제에 대해 팀원과 함께 고민하는 과정에서 critical path나 PPA 최적화와 같이 디지털 회로 설계의 관점에서 많은 고민을 하고 경험을 통해 성장할 수 있었던 계기가 되었던 것 같다. 또한 quantization과 관련된 주제는 인공지능 분야와 관련이 깊기 때문에 software - hardware co-design 측면에서의 중요성을 알게 되었고, hardware-aware AI 모델 최적화 및 경량화 분야에 관심을 갖게 되는 계기가 되었던 것 같다.

6 References

- [1] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, *Ternary Weight Networks*, arXiv preprint arXiv:1605.04711, 2016.

- [2] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, *Learning to Quantize Deep Networks by Optimizing Quantization Intervals with Task Loss*, arXiv preprint arXiv:1808.05779, 2018.
- [3] M. Courbariaux, Y. Bengio, and J.-P. David, *BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations*, in Advances in Neural Information Processing Systems (NeurIPS), vol. 28, pp. 3123–3131, 2015.
- [4] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*, arXiv preprint arXiv:1603.05279, 2016.

7 Appendix: RTL Codes of INT8 Baseline Model

INT1.58 Quantization-aware 모델의 경우에는 INT8 baseline model을 기반으로 quantization 부분을 제외하면 bit width만을 조정한 것이 전부이며, quantization과 관련된 RTL 코드는 2.3에서 모두 다루었기 때문에 appendix로써 INT8 baseline 모델에 대한 RTL 코드만 첨부하고 마무리하고자 한다. 또한, top_cnn_accelerator 모듈의 경우에는 BRAM과 연산 core가 숫자만 바꿔면서 반복되서 이용되기 때문에 BRAM의 입출력 signal과 같이 이름의 숫자만 바뀌면서 반복되는 경우에는 분량을 줄이고자 ‘(중략)’과 같이 녹색의 글씨로 표기하여 대체하였다. 온전히 전체 RTL 코드는 제출시 파일로 제출할 예정이기 때문에 여기에서는 반복되는 부분은 조금 줄이어 첨부했다.

Listing 4: top_cnn_accelerator.v

```

1  `timescale 1ns / 1ps
2
3 module top_cnn_accelerator(
4   input wire clk,
5   input wire resetn_d,
6   input wire start,
7   output wire done,
8   input wire s1_clka,
9   input wire s1_ena,
10  input wire [1:0] s1_wea,
11  input wire [16:0] s1_addr,
12  input wire [7:0] s1_dina,
13  output wire [7:0] s1_douta,
14
15  input wire s6_clka,
16  input wire s6_ena,
17  input wire [1:0] s6_wea,
18  input wire [10:0] s6_addr,
19  input wire [7:0] s6_dina,
20  output wire [7:0] s6_douta,
21
22  input wire s8_ena,
23  input wire [3:0] s8_wea,
24  input wire [9:0] s8_addr,
25  input wire [7:0] s8_dina,
26  output wire [7:0] s8_douta,
27
28  input wire s7_clka_1, (중략) ,s7_clka_10,s8_clka,
29  input wire s7_ena_1, (중략) ,s7_ena_10,
30  input wire [1:0] s7_wea_1, (중략) ,s7_wea_10,
31  input wire [11:0] s7_addr_1, (중략) ,s7_addr_10,
32  input wire [7:0] s7_dina_1, (중략) ,s7_dina_10,
33  output wire [7:0] s7_douta_1, (중략) ,s7_douta_10
34 );
35 // async to sync reset
36 reg resetn_s;

```

```

37     always @(posedge clk) begin
38         resetn_s <= resetn_d;
39     end
40
41
42     wire conv_done[7:0]; // done signals for each PEs
43     reg resetn_pe[7:0]; // negative reset signals for each PEs
44     assign done_led = done;
45
46     // Input Fmap BRAM
47     wire s1_en, s1_we;
48     wire [1:0] s1_web;
49     wire [16:0] s1_addr;
50     wire [7:0] s1_din, s1_dout;
51     assign s1_we = 1'b0;
52     assign s1_web = {2{s1_we}};
53
54     // Conv1 Result Fmap BRAM
55     wire s2_1_ena, (중략), s2_8_ena; // Port A
56     wire s2_1_wea, (중략), s2_8_wea;
57     wire [9:0] s2_1_addr_a, (중략), s2_8_addr_a;
58     wire [7:0] s2_1_dina, (중략), s2_8_dina;
59     wire [7:0] s2_1_douta, (중략), s2_8_douta;
60     assign {s2_1_wea, (중략), s2_8_wea} = 8'b00000000;
61
62     wire s2_1_enb, (중략), s2_8_enb; // Port B
63     wire s2_1_web, (중략), s2_8_web;
64     wire [9:0] s2_1_addr_b, (중략), s2_8_addr_b;
65     wire [7:0] s2_1_dinb, (중략), s2_8_dinb;
66     wire [7:0] s2_1_doutb, (중략), s2_8_doutb;
67     assign s2_1_web = s2_1_enb;
68     // (중략)
69     assign s2_8_web = s2_8_enb;
70
71     // Conv1, Conv2 Weight BRAM
72     wire s6_en, s6_we;
73     wire [10:0] s6_addr;
74     wire [7:0] s6_din, s6_dout;
75     reg [8:0] conv_w_channel;
76     reg [4:0] conv_w_index;
77     wire [7:0] w_data[7:0];
78     reg w_req[7:0];
79
80     wire resetn;
81     assign resetn = (resetn_d == 1'b0 || state == DONE || state == DONE1) ? 1'b0
82     : 1'b1;
83
84     assign s6_en = 1'b1;
85     assign s6_we = 1'b0;
86     assign s6_web = {2{s6_we}};
87     assign s6_addr = conv_w_channel * 9 + conv_w_index;
88
89     assign w_data[0] = (w_req[0]) ? s6_dout : 0;
90     // (중략)
91     assign w_data[7] = (w_req[7]) ? s6_dout : 0;
92
93     // FSM
94     localparam [2:0] IDLE = 3'b000, CONV1 = 3'b001, CONV1_DONE = 3'b010, CONV2 =
95     3'b011, CONV2_DONE = 3'b100, POOL = 3'b101, DONE = 3'b110, DONE1=3'b111;
96     assign conv1 = (state == CONV1);
97     assign conv1done = (state == CONV1_DONE);
98     assign conv2 = (state == CONV2);

```

```

97     assign conv2done = (state == CONV2_DONE);
98     assign pool = (state == POOL);
99     assign done1 = (state == DONE1);
100    assign o_valid = o_v_1;
101
102    reg [2:0] state;
103    reg [4:0] conv2_counter;
104
105    always @(posedge clk) begin
106        if(!resetn_s) state <= IDLE;
107        else begin
108            case(state)
109                IDLE: state <= (start) ? CONV1 : IDLE;
110                CONV1 : state <= (conv_done[0] & conv_done[1] & conv_done[2] &
conv_done[3] & conv_done[4] & conv_done[5] & conv_done[6] & conv_done[7]) ?
CONV1_DONE : CONV1;
111                CONV1_DONE : state <= (!conv_done[0] & !conv_done[1] & !
conv_done[2] & !conv_done[3] & !conv_done[4] & !conv_done[5] & !conv_done[6]
& !conv_done[7]) ? CONV2 : CONV1_DONE;
112                CONV2 : state <= (conv_done[0] & conv_done[1] & conv_done[2] &
conv_done[3] & conv_done[4] & conv_done[5] & conv_done[6] & conv_done[7]) ?
CONV2_DONE : CONV2;
113                CONV2_DONE : state <= (conv2_counter >= 16) ? POOL : (!conv_done
[0] & !conv_done[1] & !conv_done[2] & !conv_done[3] & !conv_done[4] & !
conv_done[5] & !conv_done[6] & !conv_done[7]) ? CONV2 : CONV2_DONE;
114                POOL : state <= (x==10) ? DONE1 : POOL;
115                DONE1 : state <= (batch==99) ? DONE : CONV1;
116                DONE : state <= (start)? CONV1: DONE;
117                default: state <= IDLE;
118            endcase
119        end
120    end
121
122    reg [6:0] batch;
123    reg [16:0] addr_offset;
124    reg [9:0] output_offset;
125    always @(posedge clk) begin
126        if(!resetn_s) begin batch <=0; addr_offset <= 0; output_offset = 0; end
127        else begin
128            if( state==DONE1)begin
129                if(batch < 7'd100) begin
130                    batch <= batch+1;
131                    addr_offset <= addr_offset + 784;
132                    output_offset <= output_offset + 10;
133                end else if(batch ==7'd100) begin
134                    batch <= batch;
135                    addr_offset <= addr_offset ;
136                    output_offset <= output_offset + 10;
137                end
138            end else if(state == DONE) begin batch <=0; addr_offset <= 0;
output_offset = 0; end
139        end
140    end
141
142    assign done = (state==DONE);
143    reg done_neg; wire done_d; //debouncing done
144    always @(posedge clk) begin
145        if(!resetn_s) done_neg <= 1;
146        else done_neg <= ~done;
147    end
148
149    assign done_d = done_neg & done;

```

```

150
151     reg start_flag;
152
153     always @(posedge clk) begin
154         if(!resetn_s) start_flag <=0;
155         else begin
156             if(start) start_flag<= 1;
157             else begin
158                 if(done_d) start_flag <= 0;
159                 else start_flag <= start_flag;
160             end
161         end
162     end
163 end
164
165
166 // resetn PEs & start PEs & conv2 iteration 16_times
167 always @(posedge clk) begin
168     if(!resetn) begin {resetn_pe[0], resetn_pe[1], resetn_pe[2], resetn_pe
169     [3], resetn_pe[4], resetn_pe[5], resetn_pe[6], resetn_pe[7]} <= 8'b11111111;
170     end
171     else begin
172         if(state == CONV1_DONE) begin
173             {resetn_pe[0], resetn_pe[1], resetn_pe[2], resetn_pe[3],
174             resetn_pe[4], resetn_pe[5], resetn_pe[6], resetn_pe[7]} <= 8'b00000000;
175         end else if(state == CONV2) begin
176             {resetn_pe[0], resetn_pe[1], resetn_pe[2], resetn_pe[3],
177             resetn_pe[4], resetn_pe[5], resetn_pe[6], resetn_pe[7]} <= 8'b11111111;
178         end else if(state == CONV2_DONE) begin
179             {resetn_pe[0], resetn_pe[1], resetn_pe[2], resetn_pe[3],
180             resetn_pe[4], resetn_pe[5], resetn_pe[6], resetn_pe[7]} <= 8'b00000000;
181         end else begin
182             {resetn_pe[0], resetn_pe[1], resetn_pe[2], resetn_pe[3],
183             resetn_pe[4], resetn_pe[5], resetn_pe[6], resetn_pe[7]} <= 8'b11111111;
184         end
185     end
186
187     reg conv2_flag;
188     always @(posedge clk) begin
189         if(state == CONV2) begin
190             if(y2 == input_depth - 2 && x2 == 0) begin
191                 conv2_counter <= (conv2_counter < 16 && conv2_flag) ?
192                 conv2_counter + 1 : conv2_counter;
193                 conv2_flag <= ~conv2_flag;
194             end
195             else conv2_counter <= conv2_counter;
196         end else if(state == CONV2_DONE || state == POOL) conv2_counter <=
197         conv2_counter;
198         else begin conv2_flag <= 0; conv2_counter <= 0; end
199     end
200
201 // Conv1, 2 Weight BRAM Addressing
202 always @(posedge clk) begin
203     if(!resetn) begin
204         {w_req[0], w_req[1], w_req[2], w_req[3], w_req[4], w_req[5], w_req
205         [6], w_req[7]} <= 8'b00000000;
206         conv_w_channel <= 0; conv_w_index <= 0;
207     end else begin
208         if(state == CONV1) begin
209             if(conv_w_index < 8) begin
210                 conv_w_index <= conv_w_index + 1;

```

```

204         if(conv_w_index == 0) begin
205             w_req[conv_w_channel] <= 1;
206             if(conv_w_channel > 0) w_req[conv_w_channel - 1] <= 0;
207         end else begin
208             w_req[conv_w_channel] <= w_req[conv_w_channel];
209         end
210     end else begin
211         conv_w_channel <= (conv_w_channel < 8) ? conv_w_channel + 1
212             : conv_w_channel;
213             conv_w_index <= (conv_w_channel < 8) ? 0 : conv_w_index;
214         end
215     end else if(state == CONV1_DONE) begin
216         {w_req[0], w_req[1], w_req[2], w_req[3], w_req[4], w_req[5],
217         w_req[6], w_req[7]} <= 8'b00000000;
218         conv_w_index <= 0;
219     end else if(state == CONV2) begin
220         if(conv_w_index < 8) begin
221             conv_w_index <= conv_w_index + 1;
222             if(conv_w_index == 0) begin
223                 w_req[conv_w_channel - 8*(conv2_counter+1)] <= 1;
224                 if(conv_w_channel - 8*(conv2_counter+1) > 0) w_req[
225                 conv_w_channel - 8*(conv2_counter+1) - 1] <= 0;
226             end else begin
227                 w_req[conv_w_channel - 8*(conv2_counter + 1)] <= w_req[
228                 conv_w_channel - 8*(conv2_counter+1)];
229             end
230         end else begin
231             conv_w_channel <= (conv_w_channel < 8 + 8*(conv2_counter+1)
232             ) ? conv_w_channel + 1 : conv_w_channel;
233             conv_w_index <= (conv_w_channel < 8 + 8*(conv2_counter+1) )
234             ? 0 : conv_w_index;
235         end
236     end else if(state == CONV2_DONE) begin
237         {w_req[0], w_req[1], w_req[2], w_req[3], w_req[4], w_req[5],
238         w_req[6], w_req[7]} <= 8'b00000000;
239         conv_w_index <= 0;
240     end else begin
241         conv_w_index <= 0;
242         conv_w_channel <= 0;
243     end
244 end
245
246 // Conv1, 2 data depth selection
247 wire [4:0] input_depth;
248 assign input_depth = (state == CONV1) ? 5'd28 :
249 (state == CONV2) ? 5'd26 : 5'd10;
250
251 // Conv1, 2 BRAM data selection
252 wire signed [7:0] PE_din[7:0];
253 wire signed [19:0] PE_dout[7:0];
254 wire PE_enin[7:0];
255 wire PE_enout[7:0];
256
257 // Quantizer
258 reg signed [9:0] dout_conv1_clipped [7:0];
259 reg signed [7:0] dout_conv1 [7:0];
260 reg signed [13:0] dout_conv2_clipped;
261 reg signed [7:0] dout_conv2;
262
263 always @(posedge clk) begin
264     if(!resetn) begin

```

```

259         dout_conv1_clipped[0] <= 0; dout_conv1_clipped[1] <= 0;
260         dout_conv1_clipped[2] <= 0; dout_conv1_clipped[3] <= 0;
261         dout_conv1_clipped[4] <= 0; dout_conv1_clipped[5] <= 0;
262         dout_conv1_clipped[6] <= 0; dout_conv1_clipped[7] <= 0;
263         dout_conv2_clipped <= 0;
264     end else begin
265         if(state == CONV1) begin // clipping
266             dout_conv1_clipped[0] <= PE_dout[0] >>> 10;
267             (중략)
268         end else if(state == CONV2) begin // sum & clipping
269             dout_conv2_clipped <= (PE_dout[0] + PE_dout[1] + PE_dout[2] +
270             PE_dout[3] + PE_dout[4] + PE_dout[5] + PE_dout[6] + PE_dout[7]) >>> 10;
271         end
272     end
273     always @(posedge clk) begin
274         if(!resetn) begin
275             dout_conv1[0] <= 0; dout_conv1[1] <= 0; dout_conv1[2] <= 0;
276             dout_conv1[3] <= 0;
277             dout_conv1[4] <= 0; dout_conv1[5] <= 0; dout_conv1[6] <= 0;
278             dout_conv1[7] <= 0;
279             dout_conv2 <= 0;
280         end else begin
281             if(state == CONV1) begin // saturation + relu
282                 dout_conv1[0] <= (dout_conv1_clipped[0] > 8'sd127) ? 8'sd127 : (
283                 dout_conv1_clipped[0] < 0) ? 0 : dout_conv1_clipped[0];
284                 (중략)
285                 dout_conv1[7] <= (dout_conv1_clipped[7] > 8'sd127) ? 8'sd127 : (
286                 dout_conv1_clipped[7] < 0) ? 0 : dout_conv1_clipped[7];
287             end else if(state == CONV2) begin // saturated + relu
288                 dout_conv2 <= (dout_conv2_clipped > 8'sd127) ? 8'sd127 : (
289                 dout_conv2_clipped < 8'sb0) ? 0 : dout_conv2_clipped;
290             end else begin
291                 dout_conv1[0] <= 0; dout_conv1[1] <= 0; dout_conv1[2] <= 0;
292                 dout_conv1[3] <= 0;
293                 dout_conv1[4] <= 0; dout_conv1[5] <= 0; dout_conv1[6] <= 0;
294                 dout_conv1[7] <= 0;
295                 dout_conv2 <= 0;
296             end
297         end
298     end
299 end
300
301 assign PE_din[0] = (state == CONV1) ? s1_dout : (state == CONV2) ?
302 s2_1_dout : 0;
303 // (중략)
304 assign PE_din[7] = (state == CONV1) ? s1_dout : (state == CONV2) ?
305 s2_8_dout : 0;
306
307 assign s2_1_dinb = (state == CONV1) ? dout_conv1[0] : 0;
308 // (중략)
309 assign s2_8_dinb = (state == CONV1) ? dout_conv1[7] : 0;
310
311 reg [4:0] x1, y1, x2, y2;
312 reg [9:0] ifmap_rd_offset;
313 // Conv1, 2 BRAM Read Addressing
314 always @(posedge clk) begin
315     if(!resetn) begin
316         x1 <= 0; y1 <= 0; ifmap_rd_offset = 0;
317     end else begin
318         if (PE_enin[0] || PE_enout[0]) begin

```

```

309         x1 <= (x1 < input_depth - 1) ? x1 + 1 : 0;
310         y1 <= (y1 < input_depth - 1 && x1 == input_depth - 1) ? y1 + 1 :
311             y1;
312             ifmap_rd_offset <= (y1 < input_depth - 1 && x1 == input_depth - 1) ? ifmap_rd_offset + input_depth : ifmap_rd_offset;
313             end else begin x1 <= 0; y1 <= 0; ifmap_rd_offset <= 0; end
314         end
315     end
316
317 // Conv1, 2 BRAM Write Addressing
318 always @(posedge clk) begin
319     if(!resetn) begin x2 <= 0; y2 <= 0; end
320     else begin
321         if(PE_enout[0] ) begin
322             if(y1 == 3 && x1 >= 11) x2 <= (x2 < input_depth - 3 && y2 < input_depth - 2) ? x2 + 1 : 0;
323             else if (y1 > 3 && (x1 >= 11 || x1 <= 8)) x2 <= (x2 < input_depth - 3 && y2 < input_depth - 2) ? x2 + 1 : 0;
324             else x2 <= x2;
325             y2 <= (x2 == input_depth - 3 && y2 < input_depth - 2) ? y2 + 1 : y2;
326         end else begin x2 <= 0; y2 <= 0; end
327     end
328 end
329
330 wire [9:0] read_addr, write_addr;
331 wire [16:0] read_addr_1;
332 wire pooling_ready;
333 assign pooling_ready = (y1 > 3) ? (x1 >= 11 || x1 <= 8) ? 1 : 0 : (y1 == 3) ? (x1 >= 11) ? 1 : 0 : 0;
334 assign read_addr = x1 + ifmap_rd_offset;
335 assign read_addr_1= read_addr + addr_offset;
336 assign write_addr = x2 + (input_depth-2)*y2;
337
338 // write address
339 assign s2_1_addrb = (state == CONV1) ? write_addr : 0;
340 // (중략)
341 assign s2_8_addrb = (state == CONV1) ? write_addr : 0;
342
343 // read address
344 assign s1_addr = (state == CONV1) ? read_addr_1 : 0;
345 assign s2_1_addra = (state == CONV2) ? read_addr : 0;
346 // (중략)
347 assign s2_8_addra = (state == CONV2) ? read_addr : 0;
348
349 // input bram enable
350 assign s1_en = (state == CONV1) ? PE_enin[0] : 0;
351 assign s2_1_ena = (state == CONV2) ? PE_enin[0] : 0;
352 // (중략)
353 assign s2_8_ena = (state == CONV2) ? PE_enin[7] : 0;
354
355 // output bram enable
356 assign s2_1_enb = (state == CONV1) ? PE_enout[0] : 0;
357 // (중략)
358 assign s2_8_enb = (state == CONV1) ? PE_enout[7] : 0;
359
360 top_convolution_core#(.DATA_WIDTH(8))
PEO (.clk(clk), .input_depth(input_depth), .resetn(resetn_pe[0] & resetn), .
start(start_flag), .d_in(PE_din[0]), .w_in(w_data[0]), .w_req(w_req[0]), .
conv_out(PE_dout[0]), .x1(x1), .y1(y1), .s1_en(PE_enin[0]), .s2_en(PE_enout[0]), .done(conv_done[0]));
361

```

```

362     // CSR로부터 받는 시작, 완료 signals
363
364     BRAM2_1 fmap1_1 (
365         .clka(clk), .ena(s2_1_ena), .wea(s2_1_wea), .addr(a(s2_1_addr)), .dina(
366             s2_1_dina), .douta(s2_1_douta), // A Port
367             .clk(b(clk), .enb(s2_1_enb), .web(s2_1_web), .addrb(s2_1_addrb), .dinb(
368                 s2_1_dinb), .doutb(s2_1_doutb) // B Port
369     );
370
371     // (중략)
372
373     BRAM2_8 fmap1_8 (
374         .clka(clk), .ena(s2_8_ena), .wea(s2_8_wea), .addr(a(s2_8_addr)), .dina(
375             s2_8_dina), .douta(s2_8_douta), // A Port
376             .clk(b(clk), .enb(s2_8_enb), .web(s2_8_web), .addrb(s2_8_addrb), .dinb(
377                 s2_8_dinb), .doutb(s2_8_doutb) // B Port
378     );
379
380     BRAM6 conv_weight (
381         .clka(s6_clka), .ena(s6_ena), .wea(s6_wea), .addr(a(s6_addr)), .dina(
382             s6_dina), .douta(s6_douta), // A Port
383             .clk(b(clk), .enb(s6_en), .web(s6_we), .addrb(s6_addr), .dinb(s6_din), .doutb(
384                 s6_dout) // B Port
385     );
386
387
388
389     reg [11:0] counter_x; //0~2303
390     reg flag;
391     always @(posedge clk) begin
392         if(!resetn)begin counter_x<=0; flag=0;end
393         else begin
394             if (state==CONV2 && o_valid_1==1) begin
395                 if(counter_x < 12'd2303) counter_x <= counter_x +1;
396                 else if (counter_x == 12'd2303 && flag==0) flag=1;
397                 else if (counter_x == 12'd2303 && flag==1) begin
398                     counter_x <= 12'd2304;
399                     flag=0;
400                 end
401             end
402         end
403     end
404
405     wire s8_en;
406     wire s8_we;
407     wire [3:0] s8_web;
408     reg [7:0] s8_din;
409     wire [9:0] s8_addr;
410     assign s8_en=(o_v_1==1 && x<=9);
411     assign s8_we=(o_v_1==1 && x<=9);
412     assign s8_web={4{s8_we}};
413
414     reg [3:0] x;
415     assign s8_addr=x + output_offset ;
416

```

```

417     always @ (posedge clk) begin
418         if (!resetn) x <= 4'd0;
419         else begin
420             if (o_v_1==1) x <= (x < 10) ? x + 1 : 10;
421             else x <= 0;
422         end
423     end
424
425     wire [7:0] o_data_1,o_data_2,o_data_3,o_data_4,o_data_5,o_data_6,o_data_7,
426     o_data_8,o_data_9,o_data_10;
427
428     always @(*) begin
429         case (x)
430             4'd0: s8_din = o_data_1;
431             // (중략)
432             4'd9: s8_din = o_data_10;
433             default: s8_din = 1'b0;
434         endcase
435     end
436
437     // Weight for FC Layer BRAM Signals
438     wire s7_en_1, (중략), s7_en_10;
439     wire s7_we_1, (중략), s7_we_10;
440     wire [1:0] s7_web_1, (중략), s7_web_10;
441     wire [14:0] s7_addr_1, (중략), s7_addr_10;
442     wire [7:0] s7_din_1, (중략), s7_din_10;
443     wire [7:0] s7_dout_1, (중략), s7_dout_10;
444
445     assign {s7_en_1, (중략), s7_en_10}={10{((state==CONV2))}};
446     assign {s7_we_1, (중략), s7_we_10}=10'b00_0000_0000;
447
448     assign s7_web_1 = {2{s7_we_1}};
449     // (중략)
450     assign s7_web_10 = {2{s7_we_10}};
451
452     assign s7_addr_1=(counter_x <= 12'd2303) ? counter_x: 12'd2303;
453     // (중략)
454     assign s7_addr_10=(counter_x <= 12'd2303) ? counter_x: 12'd2303;
455
456     wire o_v_1,o_v_2,o_v_3,o_v_4,o_v_5,o_v_6,o_v_7,o_v_8,o_v_9,o_v_10;
457
458     // FC Layer Core Instantiation
459     fc_layer fc_1(
460         .clk(clk), .resetn(resetn), .i_data(o_data_0), .i_weight(s7_dout_1),
461         .i_valid(o_valid_1), .o_data(o_data_1), .o_data_real(o_v_1)
462     );
463
464     // (중략)
465
466     fc_layer fc_10 (
467         .clk(clk), .resetn(resetn), .i_data(o_data_0), .i_weight(s7_dout_10),
468         .i_valid(o_valid_1), .o_data(o_data_10), .o_data_real(o_v_10)
469     );
470
471     // Weight for FC Layer BRAM Instantiation
472     BRAM7_1 FC_layer_weight_1 (
473         .clka(s7_clka_1), .ena(s7_ena_1), .wea(s7_wea_1), .addr(a7_addr_1),
474         .dina(s7_dina_1), .douta(s7_douta_1), // Port A
475         .clkb(clk), .enb(s7_en_1), .web(s7_web_1), .addrb(s7_addr_1), .dinb(
476         s7_din_1), .doutb(s7_dout_1) // Port B
477     );

```

```

474     // (중략)
475
476     BRAM7_10 FC_layer_weight_10 (
477         .clka(s7_clka_10), .ena(s7_ena_10), .wea(s7_wea_10), .addr(a7_addr_10)
478         , .dina(s7_dina_10), .douta(s7_douta_10), // Port A
479         .clkb(clk), .enb(s7_en_10), .web(s7_web_10), .addrb(s7_addr_10), .dinb(
480         s7_din_10), .doutb(s7_dout_10) // Port B
481     );
482
483     // BRAM for Storing Final Result
484     BRAM8 FC_layer_result(
485         .clka(s8_clka), .ena(s8_ena), .wea(s8_wea), .addr(a8_addr), .dina(
486         s8_dina), .douta(s8_douta), // Port A
487         .clkb(clk), .enb(s8_en), .web(s8_web), .addrb(s8_addr), .dinb(s8_din), .
488         doutb(s8_dout) // Port B
489     );
490
491 endmodule

```

Listing 5: top_convolution_core.v

```

1  `timescale 1ns / 1ps
2
3  module top_convolution_core#
4      parameter DATA_WIDTH = 8
5  )(
6      input wire [4:0] input_depth,
7      input wire clk,
8      input wire resetn,
9      input wire start,
10     input wire [DATA_WIDTH - 1:0] d_in,
11     input wire [DATA_WIDTH - 1:0] w_in, // weight
12     input wire w_req, // weight requirement
13     input wire [4:0] x1,
14     input wire [4:0] y1,
15     output wire signed [DATA_WIDTH+11:0] conv_out,
16     output wire s1_en,
17     output wire s2_en,
18     output wire done
19 );
20     localparam IDLE = 3'b000, LOAD = 3'b001, CONV1 = 3'b010, CONV2 = 3'b011,
21     DONE = 3'b100;
22     reg [2:0] state, next_state;
23
24     assign s1_en = (state == LOAD || state == CONV1) ? 1:0;
25     assign s2_en = (state == CONV1 || state == CONV2) ? 1:0;
26
27     reg [4:0] wrptr; reg [1:0] buffer_wrmux;
28     reg [4:0] rdptr; reg [2:0] buffer_rdmux;
29
30     reg [7:0] fmap_buffer0 [27:0];
31     reg [7:0] fmap_buffer1 [27:0];
32     reg [7:0] fmap_buffer2 [27:0];
33     reg [23:0] buffer_out;
34
35     reg [4:0] line_cnt;
36     reg [2:0] target_counter; // counter for loading 3 columns
37     reg [23:0] target_buffer [2:0];
38     wire [71:0] pixel_target;
39     integer i;
40
41     assign done = (state == DONE) ? 1 : 0;
42     assign done_led = done;

```

```

42     assign pixel_target = (line_cnt == 1 && target_counter == 3 || (line_cnt >=
43         2 && (target_counter == 0 || target_counter == 3))) ? {target_buffer[2],
44         target_buffer[1], target_buffer[0]} : 72'b0;
45     convolution conv_core (.clk(clk), .resetn(resetn), .i_data(pixel_target), .
46     w_data(w_in), .w_req(w_req), .o_data(conv_out));
47
48
49 // FSM
50 always @(posedge clk) begin
51     if(!resetn) state <= IDLE;
52     else state <= next_state;
53 end
54
55 always@(*) begin
56     case(state)
57         IDLE : next_state = (start) ? LOAD : IDLE;
58         LOAD : next_state = (y1 == 2 && x1 == input_depth - 1) ? CONV1 :
59             LOAD;
60             CONV1 : next_state = (y1 >= input_depth - 1 && x1 == input_depth -
61             1) ? CONV2 : CONV1;
62             CONV2: next_state = (line_cnt >= input_depth - 1 && x1 >= 9) ? DONE
63             : CONV2;
64             DONE : next_state = DONE;
65             default : next_state = IDLE;
66         endcase
67     end
68
69 // Write
70 always @(posedge clk) begin
71     if(!resetn) begin
72         for (i = 0; i < 28; i = i + 1) fmap_buffer0[i] <= 8'd0;
73         for (i = 0; i < 28; i = i + 1) fmap_buffer1[i] <= 8'd0;
74         for (i = 0; i < 28; i = i + 1) fmap_buffer2[i] <= 8'd0;
75     end else begin
76         if(state == LOAD) begin
77             if(buffer_wrmux == 0) fmap_buffer0[wptr] <= d_in;
78             else if(buffer_wrmux == 1) fmap_buffer1[wptr] <= d_in;
79             else if(buffer_wrmux == 2) fmap_buffer2[wptr] <= d_in;
80         end else if(state == CONV1) begin
81             if(buffer_wrmux % 3 == 0) fmap_buffer0[wptr] <= d_in;
82             else if(buffer_wrmux % 3 == 1) fmap_buffer1[wptr] <= d_in;
83             else if(buffer_wrmux % 3 == 2) fmap_buffer2[wptr] <= d_in;
84         end
85     end
86 end
87
88 // Read
89 always @(posedge clk) begin
90     if(!resetn) begin
91     end else begin
92         if(state == CONV1 || state == CONV2) begin
93             if(buffer_rdmux == 0) buffer_out <= {fmap_buffer2[rdptr],
94                 fmap_buffer1[rdptr], fmap_buffer0[rdptr]};
95             else if(buffer_rdmux == 1) buffer_out <= {fmap_buffer0[rdptr],
96                 fmap_buffer2[rdptr], fmap_buffer1[rdptr]};
97             else if(buffer_rdmux == 2) buffer_out <= {fmap_buffer1[rdptr],
98                 fmap_buffer0[rdptr], fmap_buffer2[rdptr]};
99             else buffer_out <= buffer_out;
100         end else buffer_out <= 24'b0;
101     end
102 end
103
104 // Write Pointer

```

```

95     always @(posedge clk) begin
96         if(!resetn) begin
97             wrptr <= 0; buffer_wrmux <= 0;
98         end else begin
99             if(state == IDLE) begin
100                 wrptr <= 0; buffer_wrmux <= 0;
101             end else if (state == LOAD || state == CONV1 || state == CONV2)
102             begin
103                 wrptr <= x1;
104                 buffer_wrmux <= y1 % 3;
105             end else if (state == DONE) begin
106                 wrptr <= 0; buffer_wrmux <= 0;
107             end else begin
108                 wrptr <= wrptr; buffer_wrmux <= buffer_wrmux;
109             end
110         end
111
112         // Read Pointer
113         always @(posedge clk) begin
114             if(!resetn) begin
115                 rdptra <= 0; line_cnt <= 0; buffer_rdmux <= 0;
116             end else begin
117                 if(state == IDLE) begin rdptra <= 0; line_cnt <= 0; buffer_rdmux <=
118 0; end
119                 else if (state == CONV1 || state == CONV2) begin
120                     if(line_cnt < input_depth - 1) begin
121                         rdptra <= (rdptr < input_depth - 1) ? rdptra + 1 : 0;
122                         if(buffer_rdmux < 2) begin
123                             if(line_cnt < 1) buffer_rdmux <= (rdptr == 0) ? 0 : (
124 rdptra == input_depth - 1) ? buffer_rdmux + 1 : buffer_rdmux;
125                             else buffer_rdmux <= (rdptr == input_depth - 1) ?
126 buffer_rdmux + 1 : buffer_rdmux;
127                             end else buffer_rdmux <= (rdptr == input_depth - 1) ? 0 :
128 buffer_rdmux;
129                             line_cnt <= (line_cnt < input_depth && rdptra == 0) ?
130 line_cnt + 1 : line_cnt;
131                         end else begin
132                             rdptra <= rdptra; line_cnt <= line_cnt; buffer_rdmux <=
133 buffer_rdmux;
134                         end
135                     end else begin rdptra <= 0; buffer_rdmux <= 0; line_cnt <= 0; end
136                 end
137             end
138
139             // target buffer
140             always @(posedge clk) begin
141                 if(!resetn) begin
142                     target_counter <= 0;
143                     target_buffer[0] <= 0;
144                     target_buffer[1] <= 0;
145                     target_buffer[2] <= 0;
146                 end else begin
147                     if(y1 >= 3 && (state == CONV1 || state == CONV2)) begin
148                         if(target_counter < 3) begin
149                             target_buffer[target_counter] <= buffer_out;
150                             target_counter <= (rdptr == 0) ? 0 : target_counter + 1;
151                         end else begin
152                             if(rdptra == 0) target_counter <= 0;
153                             else target_counter <= target_counter;
154                             target_buffer[0] <= target_buffer[1];
155                             target_buffer[1] <= target_buffer[2];

```

```

150          target_buffer[2] <= buffer_out;
151      end
152  end else begin
153      target_counter <= 0;
154      target_buffer[0] <= 0;
155      target_buffer[1] <= 0;
156      target_buffer[2] <= 0;
157  end
158 end
159 end
160 endmodule

```

Listing 6: convolution.v

```

1  `timescale 1ns / 1ps
2
3 module convolution(
4     input wire clk,
5     input wire resetn,
6     input wire [71:0] i_data,
7     input wire [7:0] w_data,
8     input wire w_req,
9     output wire signed [19:0] o_data
10 );
11
12 // define kernels (-2^2)
13 reg signed [7:0] gx [8:0];
14
15 // kernel sum
16 reg signed [16:0] sum1 [4:0];
17 reg signed [17:0] sum2 [2:0];
18 reg signed [18:0] sum3 [1:0];
19 reg signed [19:0] sum;
20 reg signed [15:0] gx_mul [8:0];
21 reg [3:0] index;
22
23 always @ (posedge clk) begin
24     if (!resetn) begin
25         index <= 0;
26     end else begin
27         if (w_req) begin
28             gx[index] <= w_data;
29             index <= (index < 9) ? index + 1 : index;
30         end else index <= 0;
31     end
32 end
33
34 // pipelining 1 : multiplication
35 always @ (posedge clk) begin
36     (* use_dsp = "yes" *) gx_mul[0] <= (i_data[7:0] == 0) ? 0 : gx[0] *
37 $signed(i_data[7:0]);
38     (* use_dsp = "yes" *) gx_mul[1] <= (i_data[15:8] == 0) ? 0 : gx[3] *
39 $signed(i_data[15:8]);
40     (* use_dsp = "yes" *) gx_mul[2] <= (i_data[23:16] == 0) ? 0 : gx[6] *
41 $signed(i_data[23:16]);
42     (* use_dsp = "yes" *) gx_mul[3] <= (i_data[31:24] == 0) ? 0 : gx[1] *
43 $signed(i_data[31:24]);
44     (* use_dsp = "yes" *) gx_mul[4] <= (i_data[39:32] == 0) ? 0 : gx[4] *
45 $signed(i_data[39:32]);
46     (* use_dsp = "yes" *) gx_mul[5] <= (i_data[47:40] == 0) ? 0 : gx[7] *
47 $signed(i_data[47:40]);
48     (* use_dsp = "yes" *) gx_mul[6] <= (i_data[55:48] == 0) ? 0 : gx[2] *
49 $signed(i_data[55:48]);

```

```

43      (* use_dsp = "yes" *)gx_mul[7] <= (i_data[63:56] == 0) ? 0 : gx[5] *
$signed(i_data[63:56]);
44      (* use_dsp = "yes" *)gx_mul[8] <= (i_data[71:64] == 0) ? 0 : gx[8] *
$signed(i_data[71:64]);
45  end
46
47 // pipelining : adder tree
48 always @(posedge clk) begin
49   if(!resetn) begin
50     sum1[0] <= 0; sum1[1] <= 0; sum1[2] <= 0; sum1[3] <= 0;
51   end else begin
52     sum1[0] <= gx_mul[0] + gx_mul[1];
53     sum1[1] <= gx_mul[2] + gx_mul[3];
54     sum1[2] <= gx_mul[4] + gx_mul[5];
55     sum1[3] <= gx_mul[6] + gx_mul[7];
56     sum1[4] <= gx_mul[8] + 0;
57   end
58 end
59 always @(posedge clk) begin
60   if(!resetn) begin
61     sum2[0] <= 0; sum2[1] <= 0;
62   end else begin
63     sum2[0] <= sum1[0] + sum1[1];
64     sum2[1] <= sum1[2] + sum1[3];
65     sum2[2] <= sum1[4] + 0;
66   end
67 end
68
69 always @(posedge clk) begin
70   if(!resetn) begin
71     sum3[0] <= 0; sum3[1] <= 0;
72   end else begin
73
74     sum3[0] <= sum2[0] + sum2[2];
75     sum3[1] <= sum2[1] + 0;
76   end
77 end
78
79 always @(posedge clk) begin
80   if(!resetn) begin
81     sum <= 0;
82   end else begin
83     sum <= sum3[0] + sum3[1];
84   end
85 end
86
87 assign o_data = sum;
88 endmodule

```

Listing 7: max_pooling.v

```

1 `timescale 1ns / 1ps
2
3 module max_pooling(
4   input wire clk,
5   input wire resetn,
6   input wire [7:0] i_d,
7   input wire i_v,
8   output reg [7:0] o_d,
9   output reg o_v
10 );
11   reg [4:0] cnt;
12   reg x_1;

```

```

13     reg [7:0] buffer1[0:23];
14     reg [7:0] buffer2, buffer3;
15     reg o_v_temp;
16
17
18     always @(posedge clk) begin
19         if(!resetn) begin x_1<=0; buffer2<=0; buffer3<=0; cnt<=0; end
20         else begin
21             if(i_v==1 && x_1 == 0)begin // 1 line read
22                 buffer1[cnt]<=i_d;
23                 cnt <= (cnt<5'd23) ? cnt+1 : 0;
24                 x_1 <= (cnt<5'd23) ? 0 : 1'b1;
25             end else if (i_v==1 && x_1 == 1) begin // next line read & compare
26                 if (cnt[0]==0)begin
27                     buffer2 <= (i_d >= buffer1[cnt]) ? i_d : buffer1[cnt];
28                     cnt<=cnt+1;
29                 end else if (cnt[0]==1)begin
30                     buffer3 <= (i_d >= buffer1[cnt]) ? i_d : buffer1[cnt];
31                     cnt <= (cnt<5'd23) ? cnt+1:0;
32                     x_1 <= (cnt<5'd23) ? 1 : 0;
33                 end
34             end
35         end
36     end
37
38     always @(posedge clk) begin
39         if(!resetn) o_v_temp <= 0;
40         else o_v_temp <= (x_1==1 && cnt[0]==1); // delay 1 cycle
41     end
42     always @(posedge clk) begin
43         if(!resetn) o_v <= 0;
44         else o_v <= o_v_temp; // delay 2 cycle
45     end
46
47     always @(posedge clk) begin
48         if(!resetn) o_d <= 0;
49         else begin
50             if(cnt[0] == 0) o_d <= (buffer2 > buffer3) ? buffer2 : buffer3;
51             else o_d <= o_d;
52         end
53     end
54 endmodule

```

Listing 8: fc_layer.v

```

1 module fc_layer (
2     input wire clk,
3     input wire resetn,
4     input wire [7:0] i_data,
5     input wire [7:0] i_weight,
6     input wire i_valid,
7     output wire [7:0] o_data,
8     output reg o_data_real
9 );
10
11     reg o_valid;
12     reg [11:0] counter, counter_d;
13     reg signed [27:0] sum;
14     reg signed [15:0] mul;
15
16     always @(posedge clk) begin
17         if (!resetn) mul <= 0;
18         else begin

```

```

19         if (i_valid) (* use_dsp = "yes" *) mul <= (i_data == 0) ? 0 :
20             $signed(i_data) * $signed(i_weight);
21     end
22 end
23
24 always @(posedge clk) begin
25     if(!resetn) counter <= 0;
26     else begin
27         if(i_valid) begin
28             if(counter==0) counter <= counter +1;
29             else if(counter < 12'd2304 && counter > 0) counter <= counter +
30                 1;
31             end else if(counter == 12'd2304) counter <= 0;
32         end
33     end
34
35     always @(posedge clk) begin
36         if(!resetn) counter_d <= 0;
37         else counter_d <= counter; // counter delayed
38     end
39
40     always @(posedge clk) begin
41         if(!resetn) begin o_valid<=0; sum <=0; end
42         else begin
43             if(i_valid) begin
44                 if(counter_d == 0) begin
45                     sum <= 0;
46                     o_valid <=0;
47                 end else if(counter_d < 12'd2304 && counter_d > 0) sum <= sum +
48                     mul;
49                 end
50             end
51         end
52     end
53
54     wire signed [17:0] sum_slash;
55     assign sum_slash = sum[27:10];
56
57     reg signed [7:0] o_data_temp;
58
59     always @(posedge clk) begin
60         if(!resetn) begin
61             o_data_real<=0;
62             o_data_temp<=0;
63         end else begin
64             if (o_valid) begin
65                 o_data_real <=1;
66                 if (sum_slash > 127) o_data_temp <= 8'b0111_1111;
67                 else if (sum_slash < -128) o_data_temp <= 8'b1000_0000;
68                 else o_data_temp <= sum_slash[7:0];
69             end
70             else begin o_data_real <= 0;end
71         end
72     end
73
74     assign o_data= o_data_temp;
75 endmodule

```