creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

U-Net의 FPGA 구현을 위한 효율적인 resource 사용

Efficient resource utilization for an FPGA implementation of U-Net

2021 년 2 월

서울대학교 대학원

전기 정보 공학부

제 현 승

U-Net의 FPGA 구현을 위한 효율적인 resource 사용

Efficient resource utilization for an FPGA implementation of U-Net

지도 교수 이 혁 재

이 논문을 공학석사 학위논문으로 제출함
2021 년 2 월

서울대학교 대학원
전기 정보 공학부
제 현 승

제현승의 공학석사 학위논문을 인준함
2021 년 2 월

위 원 장 김 태 환 (인)

부위원장 이 혁 재 (인)

위 원 류 수 정 (인)

초록

최근 FPGA가 가지고 있는 GPU와 CPU 대비 low power consumption에 관한 장점과 구현의 flexibility로 다양한 형태의 하드웨어가 가능하기 때문에 CNN Network의 구현에 대한 많은 연구의 진행이 이루어 지고 있다. 그러나 대부분의 FPGA CNN 가속기 연구는 classification network를 대상으로 이루어져 왔다. 최근 segmentation network가 자율 주행과 의료 서비스 등 다양한 분야에 활용됨을 고려하였을 때 segmentation network의 FPGA상에서의 구동에 관한 연구 또한 필요함을 나타낸다.

FPGA 상에서의 CNN Network의 구동은 convolution layer 동작의 특징인 여러 loop의 반복을 통한 MAC 연산으로 이루어 진다. 이 때 loop가 진행되는 순서에 따라서 on-chip과 외부 메모리 간의 data 이동량과 필요한 on-chip buffer size가 달라지게 된다. 따라서 해당 하드웨어 resource의 소모를 고려하여 target으로 하고 있는 FPGA에 적합한 Loop interchange scheme의 분석과 선정이 필요하다.

본 논문에서는 기존 Loop interchange scheme 중 하나인 Row-based reuse 방식이 가지고 있는 문제점을 보안한 Adaptive Row-based reuse 방식을 제안한다. 해당 방식은 FPGA 디자인의 설계 시 설정한 input과 output buffer size를 기반으로 최대로 저장할 수 있는 feature map의 row 수를 결정한 후 해당 multiple row들에 대해서 weight block과 함께 convolution을 진행한다. 해당 방식을 통해 CNN Network의 layer 별로 달라지는 feature map size에 영향 받지 않고 on-chip buffer에 data를 최대로 저장할 수 있다. 또한 convolution 진행 시간에 비해 다음 channel에서 사용할 weight를 load 하는 시간이 hide 되지 못하여 발생하는 latency 또한 convolution을 진행하는 range를 expand 하면서

해결한다.

제안하는 Adaptive row-based reuse 방식을 검증하고자 대표 segmentation network U-Net을 구동하는 하드웨어를 설계하여 실제 동작을 구현하였다. Convolution과 Up-convolution을 동일한 process element에서 동작이 가능하도록 설계하여 DSP와 같은 resource의 낭비를 방지하였으며 weight와 data bit를 8bit로 설정하는 대신 batch normalization을 수행하는 모듈에서 weight quantization의 scale이 batch normalization scale과 folding 되어 convolution을 진행하면서 성능의 유지를 가능하게 한다.

본 연구는 width에 대한 tiling factor를 input feature map의 전체 width로 설정하는 대신 height에 대한 tiling factor를 multiple row로 설정하여 on-chip buffer의 최대 효율을 위한 tiling 진행이 가능하다는 점에서 기존 연구와의 차이점을 가지고 있다. 제안하는 방식은 Xilinx KCU 1500 board를 대상으로 구현하였으며 기존 row-based reuse 방식의 적용 시 보다 layer 별로 feature map size에 무관하게 일정한 연산 성능을 나타내고 있음을 확인하였고 U-Net network 전체에 대해서 1.2배의 성능 향상을 확인하였다. 또한 computation range의 증가로 FPGA의 외부 메모리와의 필요 bandwidth 또한 2배 가량 감소함을 확인하였다. 해당 adaptive-row reuse 방식의 적용으로 layer 별 feature map size의 변화가 있는 segmentation network의 구동에 대해서 FPGA의 resource의 효율적인 활용을 가능하게 한다.

주요어 : FPGA 디자인, Segmentation 네트워크, U-Net 네트워크, Loop Interchange 기법
학 번 : 2019-26808

목 차

제 1 장 서 론 1

1.1 연구의 배경 1

1.2 연구의 내용 3

제 2 장 관련 연구 5

2.1 CNN의 FPGA상에서의 가속기 구현 5

2.2 Semantic segmentation 알고리즘 6

2.3 Loop Interchange scheme 9

제 3 장 Loop Interchange 분석 11

3.1 Loop Interchange의 다양한 순서 11

3.2 외부 메모리 접근 횟수 12

3.3 On-chip 버퍼 설계 15

3.4 Latency 발생과 분석 17

3.5 FPGA에 따른 Hardware design 선택 19

제 4 장 Adaptive Row-based reuse 방식 제안 23

4.1 기존 Row-based reuse 방식의 문제점 23

4.2 제안 알고리즘 26

4.3 제안 알고리즘의 다른 dataflow와의 비교 30

제 5 장 제안 방식의 적용 및 가속기 구현 32

5.1 가속기 설계 디자인 32

5.2 Convolution 연산 가속 34

5.3 Batch Normalization 및 Input Quantization 36

5.4 Range-based Weight Quantization 구현 38

5.5 Up-Convolution 구현 39

5.6 Memory Arrangement 41
제 6 장 실험 결과 및 분석 43
6.1 기존 Row-based reuse 방식과의 비교 43
6.2 On-chip buffer 크기에 따른 성능 변화 45
6.3 CPU, GPU와의 비교 47
6.4 타 연구와의 비교 47
제 7 장 결론 49
참고 문헌 50
Abstract 53

표 목차

표 2.1 Application 별 대표 network의 output 뉴런 수 7
표 3.1 Loop Interchange scheme에 따른 외부 메모리 접근 횟수 . 13
표 3.2 Loop Interchange scheme 별 외부 메모리 접근을 1번으로 제한하기 위해 필요한 On-chip buffer 크기 17
표 5.1 U-Net 네트워크 layer 별 input, output 크기 33
표 6.1 본 가속기와 CPU, GPU와의 성능 비교 47
표 6.2 본 가속기와 타 연구와의 비교 48

그림 목차

그림 2.1 U-net의 네트워크 구조 8
그림 3.1 Loop Interchange의 12가지의 방식 12
그림 3.2 Scheme 별 요구 buffer 크기 21
그림 4.1 Row-based reuse 방식의 input data 진행 순서 23
그림 4.2 Width가 작은 layer에서의 convolution 진행 24
그림 4.3 Layer 별 필요 buffer 크기 25
그림 4.4 Input buffer의 row 저장 순서 28
그림 5.1 Adaptive row-based reuse 적용 CNN 가속기 구조 34
그림 5.2 Convolution과 Upconvolution 연산의 진행 과정 40
그림 5.3 Feature data의 외부 메모리 저장 방식 41
그림 6.1 기존 row-reuse 방식과의 GOPS 비교 44
그림 6.2 기존 row-reuse 방식과의 bandwidth 비교 44
그림 6.3 Buffer size에 따른 수행 시간 변화 46

제 1 장 서 론

1.1 연구의 배경

최근 인류 사회에 등장한 딥러닝은 사회 전반적으로 막대한 영향을 끼치고 있다. 딥러닝 기반 강화학습을 통하여 개발한 인공지능을 통해 인류가 개발한 바둑을 정복하였으며 [1], 인간이 고안한 언어 또한 습득하여 딥러닝을 이용한 음성합성을 통해 실제 사람과 전화 통화를 어색함 없이 수행해 낸다 [2]. 또한 이미지에 등장하는 객체를 구분하는 일은 사람의 능력을 뛰어넘어 이를 이용하고자 실제 산업에서 딥러닝을 활용하고 있다.

특히 딥러닝 네트워크 중 convolution layer를 이용한 convolutional neural network (CNN)은 computer vision 관련 다양한 문제들에 대해 효과적인 해답을 제시함으로써 해당 네트워크를 이용한 다양한 발전과 연구가 이루어 지고 있다. 이미지를 입력 받으면 해당 이미지의 객체를 구분하는 classification [3] 에서부터 이미지 내부의 객체의 위치를 탐지하는 detection [4], 이미지 내부의 객체를 분리하는 segmentation [5]까지 convolution layer를 활용한 딥러닝 네트워크를 활용하여 다양한 분야에서 뛰어난 성능을 보이고 있다.

해당 CNN의 구동은 일반적으로 CPU와 GPU에서 진행된다. 일반적인 딥러닝 네트워크는 많은 양의 연산량과 메모리 소모를 요구하기 때문에 높은 연산 성능을 가진 하드웨어를 통하여 구동시켜야 한다. 그러나 해당 높은 연산 성능을 가진 하드웨어의 특성상 전력소모량이 많아 지며 지원하는 구동 성능의 한계로 이를 해결하기 위한 다양한 노력 또한 이루어 지고 있다.

최근 field programmable gate array (FPGA)가 가지고 있는 전력효율성과 임의의 하드웨어 설계가 가능하다는 점으로 딥러닝 네트워크를 구동할 수 있는 플랫폼으로 각광받고 있다. 사용자는 Intel 사의 Altera, Xilinx 사의 Vivado 등의 FPGA 설계 툴을 이용하여 하드웨어 코딩 후 FPGA 칩이 지원하는 자원들을 활용하여 딥러닝의 연산을 수행하는 하드웨어를 설계할 수 있다. 구동하고자 하는 딥러닝 네트워크의 구조, 또는 입력하는 데이터의 크기에 맞추어 최상의 성능을 보일 수 있는 하드웨어 구조를 설계할 수 있다는 점이 딥러닝 가속기 연구에 있어서 효과적인 플랫폼이다.

CNN을 FPGA상에서 가속시키기 위한 연구 또한 활발하게 이루어지고 있다. Convolution layer의 연산은 input feature map의 가로, 세로, channel과 같은 여러 dimension의 loop가 반복되면서 MAC 연산을 수행하면서 진행한다. 따라서 해당 MAC 연산을 FPGA 상에서 구현하는 것에서 input과 parameter data를 외부 메모리에서의 load와 save를 진행하는 dataflow의 설계에 따라서 convolution layer의 구동 속도가 달라지게 된다. 해당 설계를 통해 CNN의 전체 layer를 구동하며 최종 결과를 생성할 수 있다.

그러나 CNN 가속기에 관한 연구는 classification network 위주로 이루어져 있다. Network의 output으로 객체의 클래스 별 확률 값이 나오는 classification network 구조와는 달리 전체 image의 픽셀 별로 클래스의 index가 출력되는 segmentation network는 가장 마지막의 convolution layer이 image와 같은 size의 output이 출력되어야 하기 때문에 구조가 다르다. 따라서 FPGA 상에서 segmentation network를 가속하기 위한 하드웨어 구조 및 dataflow의 연구 또한 필요하다.

1.2 연구의 내용

본 논문에서는 segmentation network의 FPGA상에서의 최적화된 구동을 위한 하드웨어 동작 구조를 제안한다. Segmentation network는 feature map size의 변화가 다양하기 때문에 다양한 size의 feature map을 convolution 시에 동일한 연산 성능을 보일 수 있는 구조가 필요하다. 또한 DSP, Block ram (BRAM)과 같은 제한되어 있는 FPGA 자원들을 효과적으로 사용하여 parameter와 feature map의 양이 큰 large size의 CNN을 구동할 수 있는 구조가 필요하다. 이를 위해 Convolution layer의 구동의 진행 요소인 loop의 진행 순서를 분석하여 각각의 순서에 따라 소모되는 FPGA의 BRAM과 외부 메모리의 접근 횟수를 비교하여 segmentation network를 효율적으로 구동할 수 있는 최적의 scheme을 찾는다. 해당 scheme 중 row-based weight reuse scheme은 feature map과 convolution을 row 방향으로 진행하고 이 후 다음 channel에 대해서 진행하는 방식으로 다른 scheme과 비교 시 input data의 BRAM 소모와 외부 메모리 접근이 상대적으로 적어 large size의 CNN을 구동할 수 있는 scheme이다. 본 논문에서는 해당 scheme을 segmentation network의 구동에 사용하면서 발생하는 latency 문제를 분석하고 이를 해결하기 위해 Adaptive-row based reuse 방식을 제안한다. 해당 방식은 FPGA가 지원하는 BRAM의 크기와 구동하고자 하는 convolution layer의 feature size에 따라서 하나의 row가 아닌 multiple row의 처리를 진행한다. 해당 방식을 통하여 convolution layer의 크기에 영향을 받지 않고 일정한 수준의 operation 속도를 보여주면서 하나의 row가 아닌 multiple row를 처리하기 때문에 외부 메모리의 접근 또한 감소한다.

제안하는 방식은 대표적인 segmentation network U-Net [6]을 구동하기 위해 적용되었으며 Xilinx KCU 1500 board를 대상으로 구현하여

해당 하드웨어의 성능을 측정하였다. 그 결과 기존 row-based reuse 방식을 적용한 하드웨어와 비교 시에 layer 별로 feature map size에 일정한 연산 성능을 나타내고 있음을 확인하였고 U-Net network 전체에 대해서 1.2배의 성능 향상을 확인하였다. 또한 computation range의 증가로 FPGA의 외부 메모리와의 필요 bandwidth 또한 2배 가량 감소함을 확인하였다.

제 2 장 관련 연구

2.1 CNN의 FPGA상에서의 가속기 구현

최근 FPGA가 가지고 있는 적은 전력소모량과 설계의 자유도 때문에 CNN 가속기를 설계하여 구동하기 위한 연구가 진행되고 있다 [7], [8], [9], [10], [11], [12], [13] [14], [15], [16]. Convolution layer의 FPGA상에서의 구동은 input data와 weight data를 load 하여 MAC 연산을 진행하는 Process element (PE)에 input하여 연산을 진행하는 과정의 반복으로 이루어 진다. 그러나 FPGA 내부의 저장소에 convolution을 진행하는 모든 input과 weight data를 저장할 수 없기 때문에 외부메모리에서 일부분의 data를 BRAM에 load 하여 해당 data를 반복하여 사용하는 과정으로 이루어 진다. 따라서 FPGA가 제공하는 BRAM의 크기와 computation 성능을 고려하여 data 일부분의 tile을 load하여 PE에 input 하는 방식의 연구가 있었다 [7]. 또한 기존 convolution layer들을 처리 시에 한번에 하나의 layer를 순서대로 처리하면서 발생하는 intermediate data의 저장이 FPGA 자원의 많은 overhead를 차지 한다는 점을 발견하여 input data가 BRAM에 저장하는 순서를 변경하여 인접한 CNN layer들의 intermediate data를 caching 하여 multiple layer를 병렬적으로 처리할 수 있는 layer fused 방식 또한 제안되었다 [8]. Convolution 연산의 대부분을 차지하는 MAC 연산을 수행 시 winograd 알고리즘을 적용시켜 multiplication 수를 감소시킨 방식 또한 제안 되었다 [9]. 해당 방식으로 연산량이 감소되어 상대적으로 동일한 CNN의 처리 속도 또한 증가하게 되었다. CNN 가속기 구동 관련 연구에서 메모리에 저장하는 parameter와 input data를 최적화 하여 메모리의 overhead를 감소시키고자 하는 연구 또한 진행 되었다. convolution을 진행 시 weight의 값을 일반적인 32bit

floating point에서 1bit의 binary bit로 변환하여 동일한 성능을 보여주면서도 메모리의 저장 용량을 감소시킨 하드웨어 구조가 제안되었다 [10]. 또한 외부 메모리에서 FPGA 칩으로 data를 load 시에 data width와 load하는 burst length에 따라 effective bandwidth가 달라진다는 점을 활용하여 메모리에서 data를 load 하는 latency를 최소화 하는 dataflow를 제안한 연구 또한 있었다 [11]. 그러나 해당 연구들은 모두 ResNet [17], AlexNet [3]과 같은 classification network를 구동하기 위한 연구들이며 실험 결과 또한 해당 network 대상 처리 속도를 제시하고 있다.

2.2 Semantic segmentation 알고리즘

Image를 입력 시에 내부에 있는 객체들의 class를 찾고 해당 객체의 형상을 구분하는 segmentation network는 이미지의 객체 만을 찾는 classification, 객체의 위치를 찾는 detection network 보다 난이도가 높은 문제이다. 따라서 network의 output 또한 classification, detection 보다 크기가 커지게 된다. classification은 구분하고자 하는 class의 수만큼의 output을 출력하게 되고 detection은 class의 수와 함께 찾고자 하는 후보 객체의 위치 정보를 출력해야한다. 그러나 segmentation network는 input 하는 image의 size의 픽셀 위치마다 classification을 진행하는 문제이기 때문에 픽셀 위치마다 class수 만큼의 output을 출력해야한다. 표 2.1은 각각의 application 별로 대표적인 network의 output 뉴런 수를 나타내고 있다. Classification network는 ResNet [17], detection network의 경우 YOLO v2 [18], Segmentation은 SegNet [19]의 output 뉴런 수를 나타내고 있다. 각각 network별로 training 하는 dataset과 input 하는 image의 크기가 상이하지만 전체 output size를 class 별로 구분하는 SegNet의 output

뉴런의 수가 매우 많다는 것을 확인할 수 있다.

표 2.1 Application 별 대표 network의 output 뉴런 수

	Classification[17]	Detection[18]	Segmentation[19]
Input 크기	224 x 224	416 x 416	360 x 480
Class 수	1,000	20	11
Output 뉴런 수	1,000	21,125	1,900,800

또한 segmentation network는 일반적인 classification network와 다른 구조를 가지고 있다. Segmentation network는 Convolution layer를 통하여 image의 중요한 정보를 추출하는 encoder 영역과 다시 해당 feature를 convolution을 통해 픽셀 별 classification을 진행하는 decoder 영역으로 구성되어 있다 [6], [19], [20], [21]. Encoder 영역은 convolution을 진행하면서 기존 image size 보다 작은 feature map size를 생성하며 다시 decoder 영역이 해당 feature를 expand 하면서 기존의 image size와 동일한 크기의 segmentation map을 생성한다. 그림 2.1은 segmentation network의 대표적인 연구인 U-net의 구조를 보여주고 있다. U-net의 구조는 convolution과 max-pooling을 통하여 feature map size를 감소시키며 다시 decoder 영역에서 up-convolution과 convolution을 통해서 feature map size를 증가시킨다. 또한 encoder 영역의 feature map과 decoder 영역에서 발생하는 중간 feature map을 concatenate를 진행하여 높은 resolution의 feature map을 segmentation 과정에 사용하도록 설정되었다.

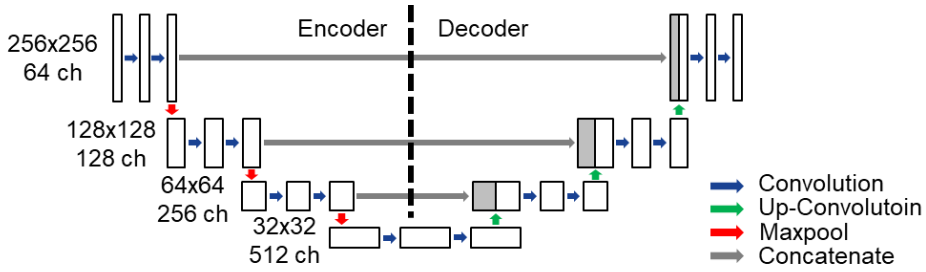

그림 2.1 U-net의 네트워크 구조

Decoder 영역에서 feature map size를 증가시키기 위해서 convolution과 다른 성격의 layer를 사용한다. U-net [6]의 경우에는 Up-convolution layer를 통하여 expand를 진행하며 SegNet [19]의 경우 Upsampling을 통하여 feature map expand를 진행한다. 따라서 FPGA를 통하여 segmentation network를 구동하기 위해서는 convolution layer 뿐만이 아닌 feature map을 expand 하는 layer의 설계 또한 필요하다.

Segmentation network를 FPGA 상에서 구동시키고자 하는 연구 또한 진행되고 있다 [22], [23], [24]. U-Net network를 구동시키기 위해 transposed convolution을 하드웨어로 구현하기 위해 효과적으로 process element에 data를 load 하기 위한 dataflow를 제안한 연구가 있다 [22]. 그러나 해당 연구는 convolution과 transposed convolution을 구동하는 process element가 각각 분리 되어 진행하여 한 쪽이 구동할 때 다른 쪽은 동작하지 않아 resource 사용 측면에서 효율적이지 않다. 또한 U-Net network를 pruning 기법을 통하여 channel 수를 감소시켜 작은 연산량으로 구동하기 때문에 해당 가속기를 통해서 large size의 network를 구동하기 어렵다. Uni-OPU [23]는 transposed convolution에 특화된 가속기로 U-Net을 포함한 transposed convolution layer를 사용하는 여러 network를 지원하고 있다. 지원하는 network는 DCGAN [25],

DiscoGAN [26]과 같은 GAN 계열의 network이며 transposed convolution도 feature map을 expand 시 발생하는 빈 공간은 0으로 채우는 zero-based transposed convolution, 또는 기존 feature map의 element로 채우는 NN-based transposed convolution을 모두 지원한다. 그러나 segmentation network 뿐만 아닌 GAN 계열의 network를 지원하면서 범용적인 dataflow를 적용하여 하드웨어를 가속하였기 때문에 segmentation network의 특징을 고려한 효율적인 하드웨어 resource 사용이 고려되지 않았다. 따라서 segmentation의 특징을 고려한 효과적인 dataflow 및 하드웨어 구조를 통해 해당 network의 성능을 향상시키는 것이 필요하다.

2.3 Loop Interchange scheme

Convolution layer를 하드웨어에서 구동하기 위해서 convolution의 연산 과정인 loop의 interchange를 통한 최적화를 진행한 연구가 있다 [13], [27], [14]. Smartshuttle [13]은 layer 별로 donate 하는 data의 종류가 다른 점에 근거하여 data가 재사용되는 정도, reusability가 달라짐을 확인하였다. 이를 통해 각각의 layer가 같은 dataflow를 갖게 되면 외부 메모리의 접근 증가하게 됨을 문제로 각각의 layer 별로 loop interchange 방식을 다르게 하는 가속기를 개발하였다. Smartshuttle [13]은 VGG16 [28]과 Alexnet [3]을 구동 시에 output channel loop를 먼저 구동하는 output feature map reuse oriented (ORO) 방식과 feature map의 width와 height loop를 먼저 구동하는 weight reuse oriented (WRO) 방식을 layer 별로 adaptive 하게 적용하였으며 이로 인해 DRAM access volume을 감소시켰다. 또한 object detection network를 FPGA에 구동하기 위해 layer 별로 다른 dataflow를 적용한 연구 [27]는 weight block이 진행하는

range에 따라서 하나의 row에 대해서 convolution을 진행하는 row-based reuse 방식과 feature map size 전체에 대해서 convolution을 진행하는 full reuse 방식을 layer 별로 적용하였다. 해당 방식은 layer 별로 input feature map과 weight의 volume이 달라짐에 따라서 외부 메모리의 접근 횟수와 on-chip buffer의 크기를 최소화 하기 위한 설계방식이다. Convolution의 loop 순서와 더불어 loop unrolling을 통한 병렬 연산의 최적화를 진행한 연구 또한 진행되었다 [14]. 해당 연구에서는 unrolling을 진행하는 loop에 따라서 적합한 Process element (PE)의 구조를 제안하며 Loop의 진행 순서와 tiling factor에 따라서 발생하는 외부메모리와 on-chip buffer의 접근 횟수를 분석하였다. 최종적으로 설계한 하드웨어 에서는 input channel의 loop를 먼저 수행하며 tiling factor의 경우 feature map의 row 전체를 on-chip buffer에 저장하는 모습을 보인다.

제 3 장 Loop Interchange 분석

3.1 Loop Interchange의 다양한 순서

본 논문은 타겟으로 하고 있는 FPGA의 제한적인 resource를 가지고 feature map size 또는 channel이 큰 network를 효과적으로 구동할 수 있음을 목표로 하고 있다. 앞 장에서 설명 한 것과 같이 FPGA 상에서 외부 메모리로부터 data를 load 하는 일은 많은 에너지 소모가 있기 때문에 해당 load의 수를 최대한 줄이는 것이 에너지 소모 측면에서 효과적이며 이를 위해서 타겟으로 하고 있는 FPGA의 on-chip buffer를 최대한 활용하는 것이 필요하다.

기본적으로 convolution을 진행하기 위해서 input feature data와 weight는 MAC 연산 (Multiply and Accumulation)을 진행하는 Process Element에 입력되어야 한다. Input feature map의 dimension은 width와 height, input channel이며 각각의 dimension이 tiling을 통한 loop를 반복하면서 이에 대응하는 weight tile과 함께 MAC 연산을 진행한다. 이와 함께 output channel의 loop를 진행 하면서 최종적으로 하나의 layer의 output feature map을 완성한다.

한편, convolution의 진행은 여러 요소들의 dimension에 대한 loop 순서에 따라서 다양한 디자인의 가능성이 있다. 이와 같이 convolution을 진행하는 loop의 순서를 변경하는 기법을 Loop interchange라고 하며 본 논문은 해당 loop interchange의 다양한 scheme에 따라서 FPGA의 하드웨어 디자인의 resource 사용과 외부 메모리의 접근을 논의하고자 한다. 그림 3.1에서 해당 Loop interchange의 12가지의 scheme을 보여주고 있다.

```
for out_ch ; T_o ++
  for in_ch ; T_i ++
    for y ; T_y ++
      for x ; T_x ++
```

(a) Scheme 1

```
for in_ch ; T_i ++
  for out_ch ; T_o ++
    for y ; T_y ++
      for x ; T_x ++
```

(b) Scheme 2

```
for y ; T_y ++
  for out_ch ; T_o ++
    for in_ch ; T_i ++
      for x ; T_x ++
```

(c) Scheme 3

```
for y ; T_y ++
  for in_ch ; T_i ++
    for out_ch ; T_o ++
      for x ; T_x ++
```

(d) Scheme 4

```
for out_ch ; T_o ++
  for y ; T_y ++
    for in_ch ; T_i ++
      for x ; T_x ++
```

(e) Scheme 5

```
for in_ch ; T_i ++
  for y ; T_y ++
    for out_ch ; T_o ++
      for x ; T_x ++
```

(f) Scheme 6

```
for y ; T_y ++
  for x ; T_x ++
    for out_ch ; T_o ++
      for in_ch ; T_i ++
```

(g) Scheme 7

```
for y ; T_y ++
  for x ; T_x ++
    for in_ch ; T_i ++
      for out_ch ; T_o ++
```

(h) Scheme 8

```
for out_ch ; T_o ++
  for y ; T_y ++
    for x ; T_x ++
      for in_ch ; T_i ++
```

(i) Scheme 9

```
for in_ch ; T_i ++
  for y ; T_y ++
    for x ; T_x ++
      for out_ch ; T_i ++
```

(j) Scheme 10

```
for y ; T_y ++
  for out_ch ; T_o ++
    for x ; T_x ++
      for in_ch ; T_i ++
```

(k) Scheme 11

```
for y ; T_y ++
  for in_ch ; T_i ++
    for x ; T_x ++
      for out_ch ; T_o ++
```

(l) Scheme 12

그림 3.1 Loop Interchange의 12가지의 방식

논의의 편의를 위해서 convolution의 kernel내에서 수행하는 loop는 kernel의 size가 일반적으로 3 또는 5로 작은 크기이기 때문에 kernel loop에 대한 tiling은 kernel size와 동일한 크기로 설정하며 input feature map과 output feature map size는 동일한 것으로 가정, stride는 1로 설정한다. Feature map의 size는 정사각형으로 동일한 width와 height 크기를 가지고 있다고 가정한다.

3.2 외부 메모리 접근 횟수

Loop interchange의 다양한 방법으로 인하여 외부 메모리의 data load와 save 횟수가 달라지게 된다. 표 3.1은 각각의 scheme에 따라 input data, weight data의 load, output data의 메모리 저장 횟수를 제시한다.

표 3.1 Loop Interchange scheme에 따른 외부 메모리 접근 횟수

Scheme	Input Load 수	Output Save 수	weight Load 수
1	out_ch/To	in_ch/Ti	1
2	out_ch/To	in_ch/Ti	1
3	out_ch/To	in_ch/Ti	y/Ty
4	out_ch/To	in_ch/Ti	y/Ty
5	out_ch/To	in_ch/Ti	y/Ty
6	out_ch/To	in_ch/Ti	y/Ty
7	out_ch/To	1	x/Tx*y/Ty
8	1	in_ch/Ti	x/Tx*y/Ty
9	out_ch/To	1	x/Tx*y/Ty
10	1	in_ch/Ti	x/Tx*y/Ty
11	out_ch/To	1	x/Tx*y/Ty
12	1	in_ch/Ti	x/Tx*y/Ty

기본적으로 data의 load 수는 해당 data의 dimension과 무관한 loop의 위치에 따라서 결정 된다. 만약 data의 dimension과 무관한 loop A 가 loop 나열 중 가장 안쪽에 위치하여 제일 먼저 처리될 경우 해당 data가 A loop로 인해서 다시 load되지 않기 때문에 1번만 외부 메모리에서 load 되며 만약 그렇지 않을 경우에는 해당 loop A의 반복 횟수만큼 해당 data가 외부 메모리에서 반복해서 load 되어야 한다.

예를 들어 input data의 경우 해당 data는 width와 height, input channel로 구성되어 있기 때문에 output channel loop는 input data와 무관한 loop이다 따라서 output channel loop가 가장 내부에 위치한 loop interchange scheme일 경우에 input data의 load 수는 1번이며 그렇지 않을 경우에는 해당 loop의 크기에 tile로 나누어진 횟수만큼 반복해서 load 한다. weight data도 마찬가지로 input channel, output channel로 구성되어 있기 때문에 해당 loop를 제외한 width, height loop가 가장

안쪽에서 먼저 수행되어야 외부 메모리에서의 load를 1번으로 제한할 수 있으며 그렇지 않은 경우 input channel과 output channel loop의 바깥에 있는 loop의 반복 횟수만큼 load 수가 결정된다.

이를 통해서 scheme 8, 10, 12가 output channel의 loop가 가장 먼저 수행되어 외부 메모리에서 load 한 input data tile이 output channel 전체에 대해서 reuse 되기 때문에 1번만 외부 메모리에서 load한다는 것을 확인할 수 있다. 이 외의 scheme은 output channel loop의 반복 횟수인 out_{ch}/T_o 만큼 input data가 reload 된다. Scheme 1, 2의 경우는 output channel과 input channel loop가 가장 외부에 있고 weight와 무관한 width, height loop가 가장 먼저 수행되기 때문에 weight의 외부 메모리에서의 load는 1번이다. Scheme 3, 4, 5, 6은 input channel과 output channel loop의 외부에 height loop가 있기 때문에 해당 height loop의 반복 횟수인 y/T_y만큼 반복되어 load 된다.

이와 더불어 convolution 시에 하나의 output을 생성하기 위해서 모든 input channel의 multiplication 결과를 accumulation 해야하기 때문에 부분의 결과를 메모리에 저장하여 이를 반복하여 load를 하는 과정이 필요하다. 이 때 만약 output feature map dimension과 독립적인 input channel loop가 가장 먼저 수행되는 경우에 외부 메모리에서 1번만 save를 진행하면 되지만 그렇지 않은 경우에 input channel loop의 반복 횟수인 in_{ch}/T_i 만큼 load와 save를 반복하게 된다. 따라서 scheme 7, 9, 11의 경우 input channel loop가 가장 먼저 수행되기 때문에 외부 메모리의 save가 1번만 일어난다고 할 수 있다.

따라서 scheme 1과 2의 경우에는 width와 height loop를 먼저 수행하기 때문에 weight data가 전체 input feature map에 대해서 reuse 된다. 해당 방식을 weight full reuse 방식 [29], [27] 이라고 부른다.

또한 scheme 8, 10, 12의 경우에는 load한 input data tile이 전체 output

channel에 대해서 loop를 수행하기 때문에 해당하는 tile에 대응하는 모든 weight에 대해서 reuse 된다. 해당 방식은 input data full reuse 방식이라고 부른다.

output data의 경우 MAC 연산 과정에서 생성된 값이 임시로 on-chip buffer에 저장되지 않고 해당 위치의 output 생성에 필요한 모든 accumulation 과정을 한번에 수행하기 때문에 해당 방식은 output data full reuse 방식이라고 할 수 있다. 해당 과정은 output data가 1번의 외부 메모리 저장이 발생하는 Scheme 7, 9, 11이 이에 해당한다.

3.3 On-chip 버퍼 설계

CNN Network의 FPGA의 가장 이상적인 동작은 각 data의 load와 save가 1번만 발생하여 operation을 진행하는 것이다. 따라서 각 design에 따라서 FPGA상의 on-chip 버퍼의 크기를 조정하여 각 input 과 weight data의 load와 output data의 저장을 한 번으로 제한할 수 있다.

만약 해당 요소 'A' 의 dimension과 무관한 loop 'B'를 기준으로 내부에 있는 loop의 tiling factor가 해당 loop의 전체 크기와 같게 된다면, 즉 'A'의 buffer size를 전체 크기로 설정한다면 내부 loop는 더 이상 반복되지 않고 on-chip buffer에서 load 할 수 있기 때문에 'A' 요소는 외부 메모리에서 1번만 load 하게 된다. 따라서 input data의 외부 메모리 load 수를 한번으로 제한하기 위해서는 input data dimension과 무관한 output channel의 loop의 내부 loop의 tiling factor가 해당 요소의 전체 크기와 동일해야한다. weight의 경우에는 width와 height loop가 weight dimension과 무관하기 때문에 해당 loop를 기준으로 내부에 존재하는 loop의 tiling factor가 해당 요소의 전체 크기와 동일해야한다.

예를 들어 Scheme 3에서 input data를 외부 메모리에서 1번만

load하기 위해서는 output channel loop가 input data의 dimension과 무관한 loop이기 때문에 해당 loop의 내부 loop의 tiling factor가 전체 요소의 수와 동일해야한다 즉 input channel loop와 width loop의 tiling factor가 각각 input channel, width 전체가 되어야 한다. 따라서 input buffer의 size는 $x \times T_y \times in_{ch}$ 로 설정할 수 있으며 weight data를 외부 메모리에서 1번만 load 하기 위해서는 height loop가 가장 바깥에 있기 때문에 해당 loop 내부의 output channel과 input channel loop의 tiling factor와 해당 요소의 크기가 동일해야한다. 따라서 weight buffer의 크기는 $K_x \times K_y \times in_{ch} \times out_{ch}$로 설정할 수 있다.

Convolution 이후 output data에 대해서 외부 메모리를 통한 load와 accumulation이 많은 파워 소모를 불러일으키기 때문에 DMA를 통한 output data의 외부 메모리로의 저장이 1번으로 제한되어야 한다. 이를 위해 output buffer의 크기 또한 증가되어 외부 메모리에 저장하기 이전의 최종 output 을 on-chip buffer에 모두 저장할 수 있어야 한다. 따라서 output feature data의 dimension과 무관한 input channel loop를 기준으로 내부 loop의 tiling factor를 해당 요소의 전체 크기와 동일하게 설정한다.

예를 들어 Scheme 3의 경우 output channel dimension과 무관한 input channel loop의 위치로 인해서 해당 loop의 내부인 width loop의 tiling factor가 width 전체 크기가 되어야 한다. 따라서 해당 scheme의 output buffer size는 $x \times T_y \times T_o$로 설정할 수 있다.

다음 표 3.2은 각 scheme에 따라 input data와 output data, weight data의 load를 1번으로 제한하기 위해 필요한 buffer size를 나타낸다.

표 3.2 Loop Interchange scheme 별 외부 메모리 접근을 1번으로 제한하기 위해 필요한 On-chip buffer 크기

Scheme	Input buffer	Output buffer	weight buffer
1	$X \times Y \times in_{ch}$	$X \times Y \times T_o$	$K_X \times K_Y \times T_i \times T_o$
2	$X \times Y \times T_i$	$X \times Y \times out_{ch}$	$K_X \times K_Y \times T_i \times T_o$
3	$X \times T_Y \times in_{ch}$	$X \times T_y \times T_o$	$K_X \times K_Y \times in_{ch} \times out_{ch}$
4	$X \times T_Y \times T_i$	$X \times T_Y \times out_{ch}$	$K_X \times K_Y \times in_{ch} \times out_{ch}$
5	$X \times Y \times in_{ch}$	$X \times T_Y \times T_o$	$K_X \times K_Y \times in_{ch} \times T_o$
6	$X \times T_Y \times T_i$	$X \times Y \times out_{ch}$	$K_X \times K_Y \times T_i \times out_{ch}$
7	$T_X \times T_Y \times in_{ch}$	$T_X \times T_Y \times T_o$	$K_X \times K_Y \times in_{ch} \times out_{ch}$
8	$T_X \times T_Y \times T_i$	$T_X \times T_Y \times out_{ch}$	$K_X \times K_Y \times in_{ch} \times out_{ch}$
9	$X \times Y \times in_{ch}$	$T_X \times T_Y \times T_o$	$K_X \times K_Y \times in_{ch} \times T_o$
10	$T_X \times T_Y \times T_i$	$X \times Y \times out_{ch}$	$K_X \times K_Y \times T_i \times out_{ch}$
11	$X \times T_Y \times in_{ch}$	$T_X \times T_Y \times T_o$	$K_X \times K_Y \times in_{ch} \times out_{ch}$
12	$T_X \times T_Y \times T_i$	$X \times T_Y \times out_{ch}$	$K_X \times K_Y \times in_{ch} \times out_{ch}$

3.4 Latency 발생과 분석

FPGA에서 CNN 네트워크를 구동할 때 높은 처리 속도를 얻기 위해서는 convolution의 처리에 대부분을 차지하는 MAC 연산의 최대한 활용하는 것이 중요하다. 이를 위해서 FPGA에서 제공하는 resource를 활용하여 최대한 많은 multiplication을 진행해야 하며 이와 동시에 끊김 없는 진행이 필요하다.

이를 위해서 convolution을 하나의 tile에 대해서 진행하는 시간에 다음 tile을 외부 메모리에서 on-chip buffer로의 load가 모두 완료가 되어야 바로 다음 tile에 대해서 convolution을 진행할 수 있다. 즉, convolution의 operation 시간이 다음 tile에 대한 load 시간 보다 길어 해당 시간이 서로 overlap 되어야 한다. 따라서 convolution computation과 data transfer의 overlap이 되기 위한 조건은 아래 식 (3.1)와 같다.

$$\text{Conv. Operation Time} > \text{Next data tile load time} \tag{3.1}$$

Conv. operation time은 on-chip에서 MAC 연산을 진행해야하는 convolution tile volume에 한번에 동시에 처리할 수 있는 parallel factor를 나누어주어 총 진행해야하는 연산량에 비례하며, Next data tile load time은 다음 tile의 연산을 위해 외부 메모리에서 load 해야하는 tile의 크기와 FPGA에서 지원하는 data transfer bandwidth에 비례한다. 따라서 식 (3.1)은 식 (3.2), (3.3), (3.4)와 같은 방법으로 계산할 수 있다.

$$\begin{aligned}\text{Conv. Volume} \times \text{MAC op. time} + \text{latency} \\ > \text{Effective BW} \times \text{Data Volume}\end{aligned} \tag{3.2}$$

$$\text{Conv. Volume} = \frac{T_x}{P_x} \times \frac{T_y}{P_y} \times \frac{T_{kx}}{P_{kx}} \times \frac{T_{ky}}{P_{ky}} \times \frac{T_i}{P_i} \times \frac{T_o}{P_o} \tag{3.3}$$

$$\begin{aligned}\text{Data Volume} = (T_x \times T_y \times T_i) + (T_x \times T_y \times T_o) \\ + (T_{kx} \times T_{kx} \times T_i \times T_o)\end{aligned} \tag{3.4}$$

따라서 FPGA에서 지원하는 외부메모리에서 on-chip으로의 data 전송 effective bandwidth가 고정되어 있다 가정했을 때 tiling factor에 따라 최적의 convolution operation을 위한 latency 발생의 최소화를 이룰 수 있다.

3.5 FPGA에 따른 Hardware design 선택

앞서 논의한 내용을 토대로 각각의 scheme에 따른 FPGA의 on-chip buffer의 사용과 외부 메모리의 접근을 파악할 수 있다. 이 때 타겟으로 하는 FPGA 보드가 가지고 있는 on-chip buffer의 제한이 있기 때문에 구동이 가능한 Scheme의 선정이 필요하다.

Scheme 1과 2의 경우 weight를 input feature map 전체에 대해서 reuse를 하기 때문에 weight buffer의 큰 size를 요구하지 않는다. 그러나 input data와 output data의 외부 메모리 접근을 최소화를 위해서 필요한 buffer size는 input과 output feature map 전체 크기에 대해서 필요하기 때문에 feature map size가 큰 layer에 대해서 불리한 scheme이라고 할 수 있다.

Scheme 7과 8의 경우에 input channel loop와 output channel loop를 가장 먼저 수행하기 때문에 해당 scheme의 input data load 또는 output data의 load는 각각 1번이며 이에 따른 input과 output buffer의 크기는 feature map size의 tile size인 T_x, T_y만큼 필요하다. 그러나 weight의 load 수는 reuse 되지 않고 전체 feature map size에 대해서 외부 메모리에서 reload 되어야 하기 때문에 이를 1번으로 load 수를 감소하기 위해서는 weight buffer의 크기가 전체 weight data의 크기만큼 설정 되어야 하기 때문에 weight의 크기가 feature map의 크기보다 상대적으로 큰 경우에 있어서 적합하기 어려운 Scheme이라고 할 수 있다.

Scheme 3과 4의 경우에 input channel loop와 output channel loop는 width와 height loop사이에 수행되는 scheme이다. 따라서 input data와 output data의 buffer size는 layer의 width 크기만큼 설정되며 input channel loop와 output channel loop의 순서에 따라서 각각 Scheme 3는 input buffer가 input channel 전체를, Scheme 4는 output buffer가 output

channel 전체 크기를 필요로 한다.

Scheme 5와 6의 경우 input channel loop가 먼저 수행되는 Scheme 5는 input buffer를 전체 feature map 전체에 대해 필요로 하며 output channel loop가 먼저 수행되는 Scheme 6의 경우에는 output buffer를 전체 feature map 전체에 대해 필요로 한다.

Scheme 9와 10의 경우에도 channel loop의 순서에 따라서 전체 feature map의 크기만큼의 on-chip buffer size를 필요로 하지만 그 외의 buffer의 경우 하나의 tile size의 buffer size만을 필요로 한다는 것이 scheme 5, 6과의 차이점이다.

Scheme 11과 12의 경우 weight의 buffer의 크기가 전체 weight data 크기에 대해서 설정되어야 한다. scheme 11의 경우에는 input buffer의 크기를 전체 input feature map의 width와 input channel에 대해서 설정하며 output buffer는 output tile에 대해서 설정한다. scheme 12의 경우는 output buffer를 output feature map의 width와 output channel에 대해서 설정하며 input buffer는 input data tile에 대해서 설정한다.

해당 분석을 토대로 구동하고자 하는 CNN Network의 각 layer 별 feature data와 weight data의 크기를 토대로 필요로 하는 buffer size를 계산할 수 있다. 일반적으로 CNN network의 구동은 하나의 layer씩 순서대로 구동하기 때문에 전체 Network를 구동하기 위해서는 layer 별 필요로 하는 buffer size 중에서 가장 큰 size의 buffer로 설정해야 한다. 따라서 식과 같이 FPGA 가속기 설계 시 layer 별 input, output, weight buffer의 크기를 계산 후 각 buffer 별로 최대 크기의 size를 선택한다.

$$Size(\text{Buffer[i]}) = \min_{layer}(Size\{\text{Buffer}[i]\}) \quad (i = Input, Output, Weight) \tag{3.5}$$

다음 그림 3.2는 대표적인 Auto-encoder 구조의 CNN 알고리즘 U-net을 구동하기 위해 총 19개의 layer에 대해 각 scheme 별로 필요한 전체 buffer size를 나타내고 있다. 표에서 max는 각 scheme 별로 19개의 layer를 구동하기 위한 최대 buffer size를 나타낸다.

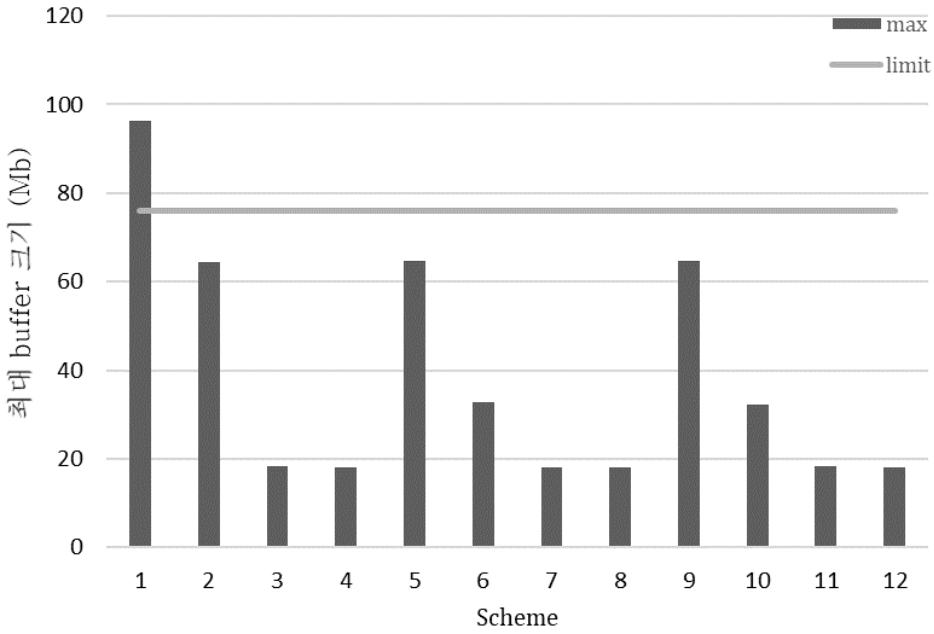

그림 3.2 Scheme 별 요구 buffer 크기

그림 3.2에서 하드웨어 설계를 위해 각 scheme 별로 max buffer size를 확인하면 scheme 1이 가장 많은 buffer size를 필요로 한다. 이와 더불어 input과 output buffer size를 feature map 전체 크기로 설정하는 scheme 1, 2, 5, 6, 9, 10이 큰 buffer size를 필요로 한다.

해당 결과를 토대로 타겟으로 하는 FPGA 보드에서 제공하는 on-chip buffer size의 조건을 만족하는 scheme을 선택하여 CNN 네트워크 구동에 적용할 수 있다. 본 논문에서 구동하고자 하는 FPGA는 Xilinx KCU 1500 보드이다. 해당 보드에서 제공하고 있는 On-chip buffer는 75.9Mb

이다. U-Net 네트워크를 해당 FPGA 보드에 구동하게 된다면 scheme 1, 2, 5, 9는 제공하고 있는 buffer size 제한을 넘어서게 되어 구동이 불가능하다. 따라서 scheme 3, 4, 7, 8, 11, 12을 사용하여 구동하는 것이 바람직하다. 본 논문에서는 그 중 row-based reuse 방식인 scheme 3을 적용하여 구동하고 row-based reuse 방식은 4장에서 논의한다.

제 4 장 Adaptive Row-based reuse 방식 제안

4.1 기존 Row-based reuse 방식의 문제점

Row-based reuse 방식은 tiling을 통한 convolution 진행 시에 weight block의 reuse 범위가 하나의 row에 제한되는 방식이다 [29], [27]. Weight block은 row 방향으로 convolution을 진행해 나가며 하나의 row에 대해서 computation이 완료되면 새로운 weight block이 동일한 row 위치의 다음 T_i channel의 data에 대해서 computation을 진행한다.

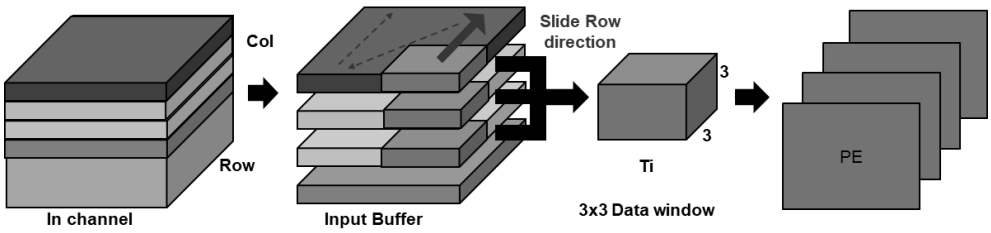

그림 4.1 Row-based reuse 방식의 input data 진행 순서

그림 4.1은 row-based reuse 방식을 수행 시에 이를 구현하는 하드웨어에서의 input data의 data flow를 보여주고 있다. Input feature map은 필요한 row에 대해서만 input buffer에 저장된 후 weight block과 함께 load 되어 MAC 연산을 위한 Process Element(PE)에 입력된다. 해당 row에 대해서 convolution을 진행하면서 이와 동시에 weight buffer는 convolution을 진행하기 위한 새로운 T_i channel의 weight block을 load한다.

해당 방식은 앞서 분석한 loop interchange에 따르면 convolution 시에 row loop를 먼저 처리한 후 다른 input channel의 loop를 처리하는

Scheme 3에 해당하며 input buffer에 저장하는 Tile의 크기는 T_x의 경우 input feature map의 width, T_y는 convolution을 진행하며 하나의 row를 생산하기 때문에 1이다.

해당 방식은 input buffer의 크기를 feature map 전체 크기가 아닌 하나의 row의 크기에 맞추어 설정할 수 있어 다른 방식과 비교했을 때 FPGA 상에서의 buffer를 많이 차지하지 않는다는 점에서 합리적인 방식이다.

그러나 해당 방식의 사용에 있어서 Tiling factor인 T_x를 고정적인 값이 아니라 feature map의 width로 선정하면서 convolution 진행에 있어서 문제점을 야기할 수 있다.

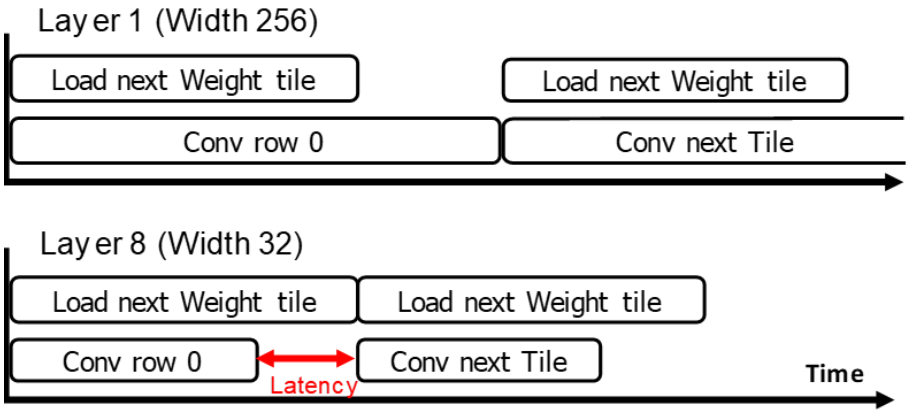

그림 4.2 Width가 작은 layer에서의 convolution 진행

다음 그림 4.2에서 U-Net 네트워크가 row-based reuse 방식을 통하여 진행하면서 대표적인 layer 1번과 8번의 초기 진행을 보이고 있다. 두 layer의 차이점은 convolution을 진행할 input feature map의 size가 각각 256과 32로 길이 측면에서 8배 차이가 나는 점이다. 따라서 row-based reuse 방식을 적용하게 되면 Input buffer에 저장하는 data의 양이 달라지며 on-chip에서 해당 row의 convolution을 수행하는데 걸리는 시간

또한 달라진다. 그러나 새로운 weight tile을 load 하면서 소요되는 시간은 일정하다. 따라서 만약 feature map size가 작은 layer을 처리하게 되면 convolution을 완료 하였음에도 불구하고 weight load 시간으로 인해서 다음 tile에 대한 convolution을 시작하지 못하게 된다.

또한 row-based reuse를 적용한 하드웨어 설계 시에 input buffer와 output buffer, weight buffer의 크기를 설정해주어야 하는데 이는 구동하고자 하는 CNN 알고리즘의 모든 layer를 동작할 수 있는 크기로 설정해야한다. 따라서 각 layer를 구동하는데 필요한 buffer 크기 중 가장 큰 size를 buffer size로 설정해야 한다. 이와 같은 경우에 있어서 가장 큰 size를 가지는 layer 이외의 layer를 구동하는 경우 buffer에 저장하는 data의 양이 작아 해당 buffer를 충분히 utilize하지 않게 되는 문제가 발생한다.

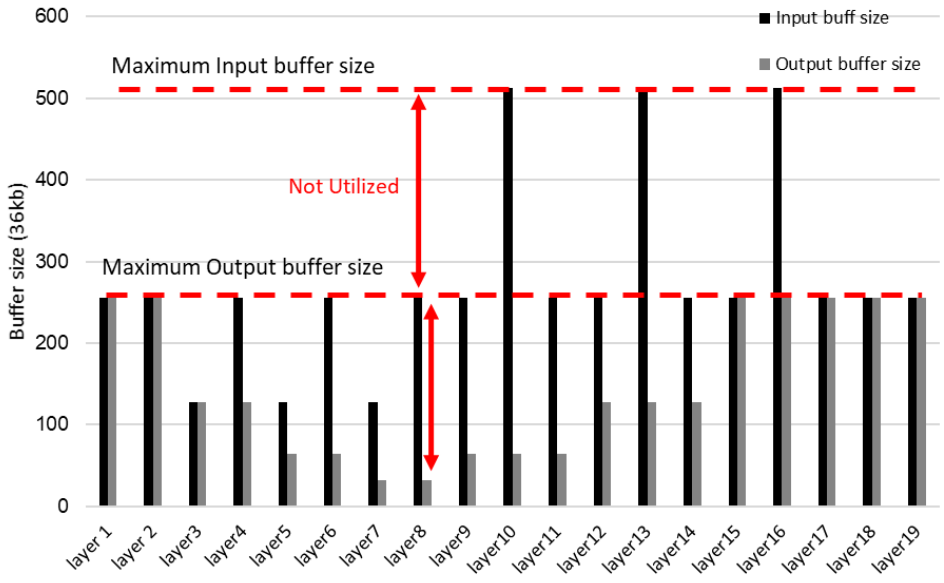

그림 4.3 Layer 별 필요 buffer 크기

다음 그림 4.3은 U-Net 알고리즘을 구동하는데 있어서 각 layer 별

필요한 input, output buffer size를 나타내고 있다. 여기서 input buffer는 해당 input feature map의 row와 해당 row의 전체 channel size를 저장하기 위해 설정된 크기이며 output buffer는 output feature map의 하나의 row만을 저장하기 위해 설정 된 크기이다. U-Net 알고리즘은 feature map size가 감소하다 다시 증가하는 Auto-Encoder 형식의 구조를 가지고 있기 때문에 필요한 output buffer size 또한 감소하다 다시 증가하는 형태를 보이고 있다. 그림 4.3에서 보여주는 결과는 앞서 말한 것과 같이 buffer size를 layer 중 가장 큰 buffer size를 토대로 설정하기 때문에 나머지 layer를 구동 시에는 설정한 buffer를 충분히 utilize 하지 않게 된다. 이는 하드웨어 resource의 낭비와 함께 design의 발전 가능성을 보여준다.

4.2 Adaptive row-based reuse 방식 제안

앞서 논의한 row-based reuse 방식의 문제점을 해결하기 위해서 adaptive row-based reuse 방식을 제안한다. 본 논문의 목적은 feature map size가 다양하게 변화하는 Auto-encoder 형태의 CNN Network를 resource의 낭비 없이 효과적으로 구동시키기 위함이며 기존 row-based reuse 방식에서 보여주는 문제점은 feature map size가 작은 layer를 구동 시에 weight의 load 시간이 convolution computation 시간에 hide가 되지 못하여 convolution이 원활하게 진행하지 못하고 latency가 길어진다는 점이다. 이를 해결하기 위해 2가지의 방법을 생각 할 수 있는데 하나는 weight의 load 시간을 줄여 computation 시간에 hide 될 수 있도록 하는 것이며 다른 하나는 computation 시간을 늘려 고정적인 weight load 시간을 hide 하는 방법이다. 이 때 weight의 load 시간을 감소시키는 방법은 weight의 tile size는 tiling factor 선정으로 변화 하지 않음을

가정하고 이를 가져오기 위한 FPGA board의 Bandwidth 또한 변화 시킬 수 없기 때문에 해당 방법의 논의는 추후에 진행하기로 한다.

Weight load 시간을 hide 하기 위한 두 번째 방법으로 convolution 시간을 늘리는 것을 생각할 수 있는데 해당 문제가 feature map size가 작은 input의 하나의 row를 computation 하는데 발생하는 문제이기 때문에 computation을 처리하는 row를 하나가 아닌 여러 개의 row를 처리할 수 있도록 진행한다.

제안하는 방식은 하드웨어의 input과 output buffer 크기에 맞추어 하나의 weight tile로 convolution을 처리할 수 있는 최대 multiple row의 수를 구하여 해당 수만큼 row 방향으로의 convolution을 진행한다.

$$Multi\ Row = \min\left(\frac{Input\ buffer\ size}{In\ width \times \lceil in_{ch}/T_i \rceil}, \frac{Output\ buffer\ size}{Out\ width}\right) \qquad (4.1)$$

Convolution을 진행하는 multiple row의 수는 식 (4.1)과 같이 input buffer와 output buffer에 각각 최대로 저장할 수 있는 row의 수 중 최소값을 토대로 결정한다. Input buffer에는 전체 channel의 row가 저장되기 때문에 Input buffer size에 width와 input channel size의 곱셈 값을 나누어준 값으로 multiple row를 결정하며 output buffer에는 T_i channel 만큼의 row가 저장 되기 때문에 output buffer size에 width를 나누어준 값으로 multiple row를 결정한다. 이 후 두 값 중 더 작은 값을 최종 multiple row로 결정하여 해당 layer에 대해 연산을 진행한다.

결국 on-chip buffer에 저장하는 tiling factor T_x를 구동하고자 하는 layer의 width의 크기로 선정하면서 on-chip buffer의 크기를 기준으로 Ty이 값이 layer에 따라서 가변적으로 변화하게 된다.

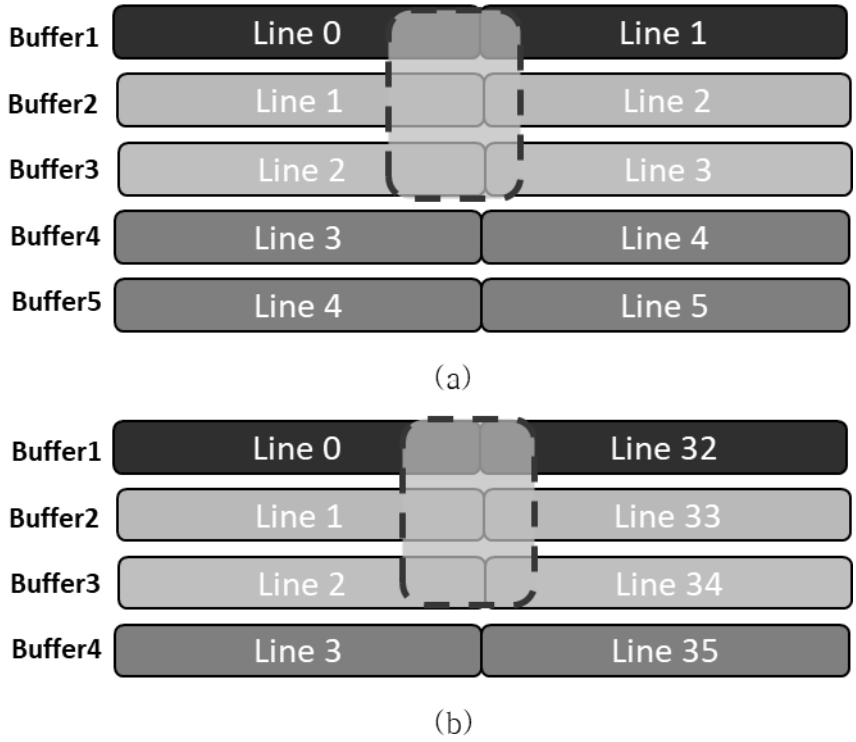

그림 4.4 (a) 연속적인 row를 buffer에 저장 시 필요 buffer, (b) 일정한 간격을 갖는 row를 buffer에 저장 시 필요 buffer

On-chip buffer에 저장하는 Row의 순서는 feature map의 height를 대상으로 일정한 간격을 두고 load 하여 computation을 진행한다. 그림 4.4에서 input buffer에 저장하는 row의 순서를 연속적으로 저장하는 경우와 간격을 두고 저장하는 경우를 비교한다. 그림 (a)는 buffer 1에 row 0와 row 1을 저장하며 buffer 2에는 row 1과 row 2를, buffer 3에는 row 2와 row 3을 저장하여 on-chip에서 2개의 row를 computation 하기 위한 data window를 생성하는데 다음과 같은 방식으로 진행하게 되면 다음 convolution을 진행하기 위한 row 3과 row 4를 추가적인 buffer에 저장해 놓아야 하기 때문에 2개의 buffer가 더 필요하게 된다. 그러나 그림 (b)에서는 buffer 1에 row 0와 row 32를 저장하며 다음 convolution의 진행을 위한 row 3과 row 35의 저장을 위해서 추가적인 buffer가 1개만

필요하게 된다. 이는 multiple row에 대해서 convolution을 진행하는데 circular buffer상에서 overlap 되는 row가 적을수록 새로운 row를 load하기 위한 추가적인 buffer가 증가함에 따름이다. 따라서 row의 간격을 두고 convolution을 처리하는 것이 buffer 사용에 있어서 경제적이라고 할 수 있다. 따라서 하나의 buffer에 저장되는 row는 일정한 간격을 가진 row들이 저장되도록 한다. 해당 buffer에 저장하는 row들의 Load interval은 식 4.2와 같이 feature map의 height 값에 multiple row factor를 나눈 값으로 feature map의 전체 row들 중 multiple row factor 만큼 분할하여 buffer에 저장하는 의미로 해석할 수 있다.

$$Load\ \ Interval = \left\lfloor \frac{In\ \ height}{Multi\ \ Row} \right\rfloor \tag{4.2}$$

이를 바탕으로 Input buffer에 저장되는 multiple row의 모습은 그림 4.5와 같다. 예를 들어 Input feature map size가 256이고 multiple row factor가 on-chip buffer size와 input channel에 따라 4로 결정되었을 때 하나의 buffer에 4개의 row가 저장된다. 이 때 하나의 buffer에 저장되는 row들의 Load interval은 feature map height 256을 multiple row 4로 나눈 64가 되며 buffer에 64의 간격을 갖는 row들을 저장한다. 따라서 그림 4.5의 buffer 1에는 row 0와 row 64, row 128, row 192가 저장되며 이후로 다음 tile의 동일한 위치의 row가 input buffer에 저장된다. Convolution을 진행하는 동안에는 kernel size가 3일 경우에 3개의 buffer에서 input data를 load 하여 data window를 제작하며 그 외의 buffer에서 다음 row를 외부 메모리에서 load 하여 저장하여 다음 row의 convolution을 준비한다.

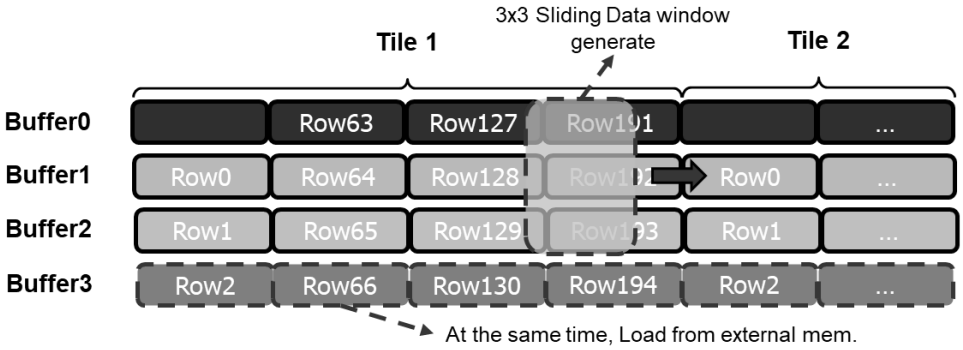

그림 4.5 Adaptive row-based reuse 방식 적용 시 Input buffer

4.3 제안 방식의 다른 dataflow와의 비교

본 논문에서 제시한 Adaptive row-based reuse scheme은 하드웨어 설계 시 설정한 on-chip buffer의 size에 맞추어 CNN의 layer 별로 feature size에 저장할 수 있는 최대 data를 저장하는 방식이다. 이 때 buffer에 저장하는 input data tile은 T_x의 경우 feature data의 전체 row 길이이며 T_y의 경우 layer 별로 adaptive하게 변환하는 multiple row factor이다. 해당 dataflow는 타 연구에서 제시한 dataflow와 비교를 할 수 있는데 Smartshuttle [13]에서 제시한 dataflow는 CNN layer 별로 input data와 weight data에 따라서 loop interchange scheme을 다르게 적용하여 외부 메모리 접근 양을 감소시킨 방식이다. 그러나 해당 방식은 외부 메모리의 접근을 감소시키기 위해 layer별로 다른 dataflow를 적용하였지만 on-chip buffer의 제한된 크기에 대한 고려와 해당 buffer를 최대로 활용하기 위한 dataflow의 적용은 부족하다. 또한 convolution을 진행하는 tiling factor의 경우 T_x를 feature map의 width보다 작은 값으로 설정하여 T_x를 feature map width로 설정한 본 연구의 dataflow보다 loop를 한번 더 반복해야한다.

Row-based reuse scheme과 full reuse scheme을 layer 별로

adaptive하게 적용한 dataflow의 경우 [27], weight와 feature data의 크기에 따라서 두 dataflow를 선택하여 convolution을 진행한다. 그러나 row-based reuse scheme과 full reuse scheme의 tiling factor T_y가 각각 1 또는 전체 feature height 두 factor로만 결정되기 때문에 본 연구보다 layer 별 adaptive한 dataflow가 제한적이다. 본 연구는 on-chip buffer와 layer의 feature map size에 따라서 T_y 값이 결정되기 때문에 주어진 buffer를 가지고 모든 layer에 걸쳐 최대의 성능을 보일 수 있다.

제 5 장 제안 방식의 적용 및 가속기 구현

이번 장에서는 본 논문에서 제안한 dataflow 방식인 Adaptive row-based reuse 방식의 FPGA 상에서의 구현에 관하여 고려해야하는 사항과 함께 설계 과정에 대한 내용을 설명한다.

5.1 가속기 설계 디자인

CNN 네트워크를 가속기에서 구동하기 위해서는 CNN 네트워크 각 layer 마다의 input, output feature map 크기와 weight의 크기를 고려한 on-chip buffer size 선정이 선행되어야 한다. 본 논문에서 구동하고자 하는 네트워크는 U-Net 이기 때문에 해당 네트워크의 layer 별 크기를 확인하여 해당 layer를 돌리기 위한 필요 data의 size를 확인한다.

다음 표 5.1은 Semantic segmentation의 대표적인 네트워크 U-Net의 layer 별 input과 output feature map의 size와 channel을 나타내고 있다. U-Net은 앞서 언급한 것과 같이 feature map size가 Encoder 단에서 감소하였다 다시 Decoder 단에서 증가하는 형태를 보이고 있다. 또한 feature map 크기를 증가시키기 위해서 Up-convolution을 활용하여 증가시키며 Encoder 단에서의 feature map과 Decoder 단에서의 feature map을 concatenate를 하는 모습을 보이고 있다.

표 5.1 U-Net 네트워크 layer 별 input, output 크기

Layer	Type	Input		Output	
		W, H	C	W, H	C
1	Conv	256	3	256	64
2	Conv	256	64	256	64
3	Conv	256	64	128	128
4	Conv	128	128	128	128
5	Conv	128	128	64	256
6	Conv	64	256	64	256
7	Conv	64	256	32	512
8	Conv	32	512	32	512
9	Up	32	512	64	256
10	Conv	64	512	64	256
11	Conv	64	256	64	256
12	Up	64	256	128	128
13	Conv	128	256	128	128
14	Conv	128	128	128	128
15	Up	128	128	256	64
16	Conv	256	128	256	64
17	Conv	256	64	256	64
18	Conv	256	64	256	2
19	Conv	256	2	256	1

Row-based reuse 방식에서 Input buffer의 크기는 input feature map의 width와 전체 input channel을 tiling factor로 나눈 값의 곱셈을 통하여 계산하며 output buffer의 크기는 output feature map의 width를 통하여 설정한다. 따라서 U-Net을 구동하기 위한 최소 buffer의 크기는 input buffer는 512, output buffer는 256이다.

이와 동시에 FPGA 하드웨어에서 on-chip에 저장할 data의 양을 결정하는 tiling factor를 설정해야한다. FPGA에서 제공하는 AXI Bus의 data width는 512, data bit를 8bit로 설정하였을 시에 data가 64의 data가 하나의 line으로 표현할 수 있어 data transfer 과정에서의 bit 수 낭비를 줄일 수 있다. 또한 layer-by-layer 방식으로 하나의 layer씩 처리하는 과정 속에서 외부 메모리에 output data를 저장하는 형식 그대로 다음

layer에서 사용해야하기 때문에 input과 output tiling factor를 동일하게 64로 적용한다.

On-chip에서 buffer에 저장되어 있는 data를 load 하여 동시에 computation을 진행하는 parallel factor를 설정해야한다. U-Net을 구동하기 위한 layer 중 convolution layer는 kernel size 3x3을 사용하는데 반해 Up-convolution layer는 kernel size 2x2를 사용한다. 따라서 두 종류의 layer를 지원하기 위해서는 kernel size에 가변적으로 동작할 수 있도록 설정한다. 따라서 on-chip에서 동작하는 parallel factor P_x와 P_y를 각각 1로 설정한다.

5.2 Convolution 연산 가속

Convolution의 연산을 위한 가속기 구조는 다음 그림 5.1과 같다. on-chip buffer에 저장되어 있는 input과 이에 대응하는 weight data가 MAC 연산을 위해 Process Element에 입력되며 Process element에서 multiply를 진행하면서 partial sum buffer에 중간 결과를 저장한다.

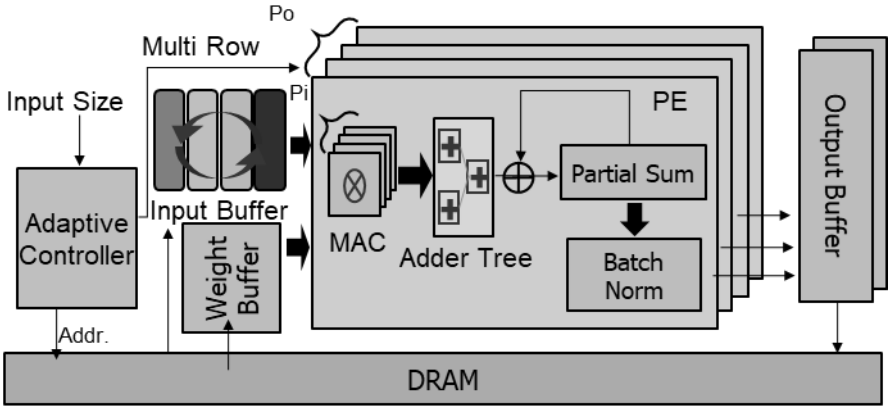

그림 5.1 Adaptive row-based reuse 적용 CNN 가속기 구조

Adaptive row-based reuse 방식에서 Convolution의 연산은 weight의 kernel 크기가 3x3 이기 때문에 computation을 위해서는 row를 저장하는 input buffer가 3개 이상이 필요하다. 따라서 3개의 input buffer를 통하여 convolution 진행에서 필요한 input data를 load 하며 이와 동시에 다음 row의 input data를 저장하기 위해 buffer 하나를 추가하여 외부 메모리에서 load하는 시간을 convolution operation 시간에 hide 할 수 있도록 한다. 하나의 row에 대한 convolution을 완료 하면 기존의 2개의 buffer와 새로 input data를 저장한 buffer를 통하여 다음 row에서의 convolution을 진행하며 나머지 하나의 buffer를 이용하여 새로운 row data를 input buffer에 저장한다.

Process element에서의 MAC 연산은 input buffer과 weight buffer에서 3x3 kernel size의 convolution을 위해 각 위치 별로 순서대로 총 9번의 multiply를 진행한다. 이와 동시에 register를 이용하여 해당 결과를 accumulation을 진행하며 이후 최종 9번의 multiply를 완료하게 되면 adder tree를 통해 T_i channel의 output을 하나의 결과값으로 summation을 진행한다.

해당 MAC 연산을 반복하면서 모든 input channel에 대한 convolution이 완료된 최종 output은 batch normalization 모듈을 거친 후 output buffer에 저장되어 DMA를 통한 외부 메모리로의 전송을 기다리게 된다. output buffer 또한 외부 메모리로의 전송과 on-chip에서의 convolution 연산 시간의 hide를 위하여 2개의 output buffer를 두어 ping-pong 형식으로 외부 메모리로의 전송과 on-chip에서의 결과물 저장을 순환한다.

5.3 Batch Normalization 및 Input Quantization

CNN 네트워크를 구동하기 위해 convolution layer와 함께 필요한 추가 layer는 batch normalization과 activation layer이다. Batch normalization layer [30]는 기본적으로 convolution 이후의 output을 토대로 평균과 표준편차를 계산한 다음 학습과정에서 구한 alpha와 beta 값을 곱셈과 덧셈을 통하여 output data를 학습된 분포로 변형하는 역할을 한다.

$$Scale = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}, \quad Bias = \beta - \frac{\gamma \times \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{5.1}$$

그러나 하드웨어 상에서 convolution output의 평균과 분산을 구하는 연산은 많은 resource의 사용을 필요로 한다. 따라서 본 논문에서는 평균과 분산의 연산 과정을 training 시에 저장한 training data의 이동 평균과 이동 분산을 이용하며 연산의 용이함을 위해서 해당 값과 batch normalization의 alpha와 beta 값을 결합하여 식 (5.1)과 같이 scale과 bias 값으로 변환한다. 이를 통하여 batch normalize의 연산은 output channel dimension에 대해서 한 번의 multiply와 한 번의 addition으로 대체될 수 있다.

FPGA 상에서 convolution 가속기를 설계 시 구동하고자 하는 CNN 네트워크의 합리적인 성능을 위해서 data bit의 width 또한 고려해야한다. 본 연구에서 진행한 CNN 가속기는 U-Net을 구동하기 위함인데 소프트웨어 상에서의 시뮬레이션 결과 data와 weight를 8 bit fixed-point로 표현하더라도 충분한 성능을 얻을 수 있음을 확인하였다. 따라서 weight와 data는 8 bit length에 fractional bit는 각각 4bit, 8bit 이며, batch normalization에서 사용하는 scale과 bias는 모두 16 bit length에 8 bit

fractional bit로 연산하도록 설계하였다.

본 논문에서 구현한 CNN 가속기의 Convolution 연산 과정은 아래 식 (5.2)와 같다.

$$\text{output} \times 2^4 = \left[\begin{matrix} \{(I \times 2^4) * (W \times 2^8)\} \times \frac{(Scale \times 2^8)}{2^{12}} + \\ (Bias \times 2^8) \end{matrix} \right] \tag{5.2}$$

Convolution 연산을 마친 후 최종 output data를 4 bit factional bit로 설정하기 위한 quantization 과정은 batch norm 모듈에서 bit shift를 통하여 구현할 수 있다. Convolution을 위하여 weight와 input data의 multiplication을 진행하게 되면 각각 4bit와 8bit의 fractional bit를 가지고 있기 때문에 최종 12bit의 fractional bit를 가지게 된다. 해당 multiplication 값은 이후 batch normalization의 scale 값을 곱해주면 scale의 fractional bit가 8bit이기 때문에 최종 20 bit의 fractional bit를 가진다. 이후 bias의 덧셈을 진행하기 위해서는 더하고자 하는 data의 fractional bit length가 동일해야하기 때문에 bit shift를 통하여 부분 output의 fractional bit를 8bit로 맞추어 주어 bias의 덧셈을 진행한다. 마지막 batch normalization까지 완료된 최종 output은 다시 4bit fractional output을 위해 bit shift를 통하여 4 fractional bit로 맞추어 주며 해당 과정에서 overflow를 확인하여 8bit의 범위 내에 data를 맞추어 주는 과정을 진행한다.

5.4 Range-based Weight Quantization 구현

Weight의 quantization을 통해 weight의 8 bit fixed point 값을 얻을 수 있다. 기본적으로 weight의 fractional bit를 8bit로 설정하여 CNN 가속기 상에서의 inference를 진행할 수 있지만 구동하고자 하는 네트워크 또는 해상도에 따라서 좀 더 정교한 weight의 값을 통한 inference를 진행할 필요가 있다.

기본적인 weight quantization 방법은 range-based quantization으로 구동하고자 하는 layer의 weight distribution의 range를 토대로 하나의 bit당 대응하는 step size를 계산하여 각 floating point 값이 fixed point 값에 대응하도록 설정하게 된다. 그러나 제안한 가속기에서의 weight는 8bit fractional point를 기본으로 가정하기 때문에 하나의 bit당 대응하는 step size가 고정되어 있다. 식 (5.3)은 range-based quantization의 1 bit에 해당하는 step size를 나타내며 식 (5.4)은 4 bit fractional bit를 통한 quantization시 1 bit에 해당하는 step size를 나타낸다.

$$Step_{range-based} = \frac{2 \times \max(abs(weight))}{2^n - 1} \tag{5.3}$$

$$Step_{4frac-bit} = \frac{1}{2^4} \tag{5.4}$$

4 bit fractional bit를 통한 quantization시 quantization을 진행 하는 weight의 distribution 또한 고정되어 있기 때문에 layer별 weight의 distribution을 잘 대응하지 못한다.

따라서 제안하는 하드웨어 구성에서 weight value의 range-based quantization을 적용하기 위해 weight와 batch normalization의

parameter를 quantization을 진행 시에 weight는 range-based quantization을 진행한 후 해당 quantization scale은 batch normalization scale에 folding 하여 진행하는 방식을 적용한다. 다음 식 (5.5)은 weight의 range-based quantization을 적용한 값을 이용하여 convolution을 진행하는 모습이다.

$$\text{output} \times 2^4 = \left[\{(I \times 2^4) * (W \times S_Q)\} \times \frac{\left(S_B \times \frac{2^9}{S_Q} \times 2^8\right)}{2^{12}} + (Bias \times 2^8) \right] \quad (5.5)$$

Weight data는 range-based quantization을 진행하며 batch normalization에서 고정된 bit shift 값이 영향을 받지 않기 위하여 batch normalization의 scale 값과 quantization의 scale 값을 folding 하여 해당 값을 batch norm 모듈에서 multiplication을 진행한다.

해당 방식을 통하여 input feature map의 convolution은 weight의 range에 fit 된 quantization 값을 이용하여 convolution을 진행하게 된다.

5.5 Up-Convolution 구현

U-Net 알고리즘에서 사용하는 up-convolution은 하나의 input data가 2x2 weight와 multiply를 진행하여 2x2 output을 생성하며 이 후 input channel에 대해서 summation을 진행하는 convolution이다. 본 논문에서 제안하는 가속기는 추가적인 resource의 사용없이 동일한 process element가 up-convolution을 지원한다.

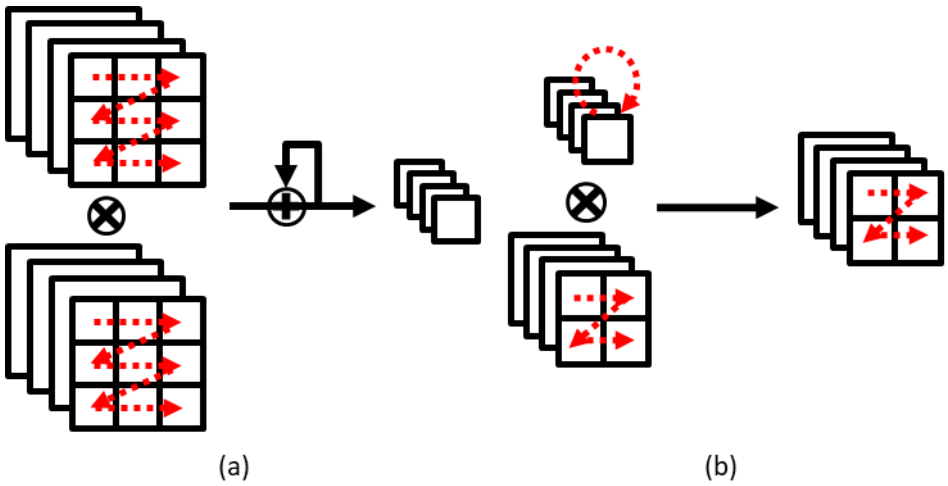

그림 5.2 (a) Convolution 연산의 진행 과정, (b) Up-convolution 연산의 진행 과정

그림 5.2은 Up-convolution의 연산 진행에 관하여 convolution과의 차이점을 보여주고 있다. Convolution은 3x3의 input data가 각기 다른 weight 값에 대응하여 multiply를 진행하지만 Up-convolution은 동일한 data가 반복하여 2x2의 weight 값에 대응하여 multiply를 진행한다. 또한 convolution은 3x3xInput channel의 data가 하나의 output을 생성하지만 up-convolution은 1x1xInput channel의 data가 2x2의 output을 생성하기 때문에 필요한 buffer size의 크기는 input 보다 output buffer가 더 크다.

하나의 weight kernel를 이용하여 2개의 row를 생성 해야하기 때문에 row를 저장하는 output buffer가 2개 필요하다. 따라서 해당 가속기는 convolution의 연산 시에는 ping-pong 형식으로 동작하는 output buffer를 모두 사용하여 on-chip의 output data를 2개의 output buffer에 저장하며 이후 외부 메모리에 2개의 output buffer에 저장되어 있는 output을 전송한다.

5.6 Memory Arrangement

본 절에서는 convolution을 진행하기 위하여 input data와 weight data를 외부 메모리에 저장하기 위한 address mapping과 Adaptive row-based reuse 방식에서의 메모리 접근에 관하여 논의한다.

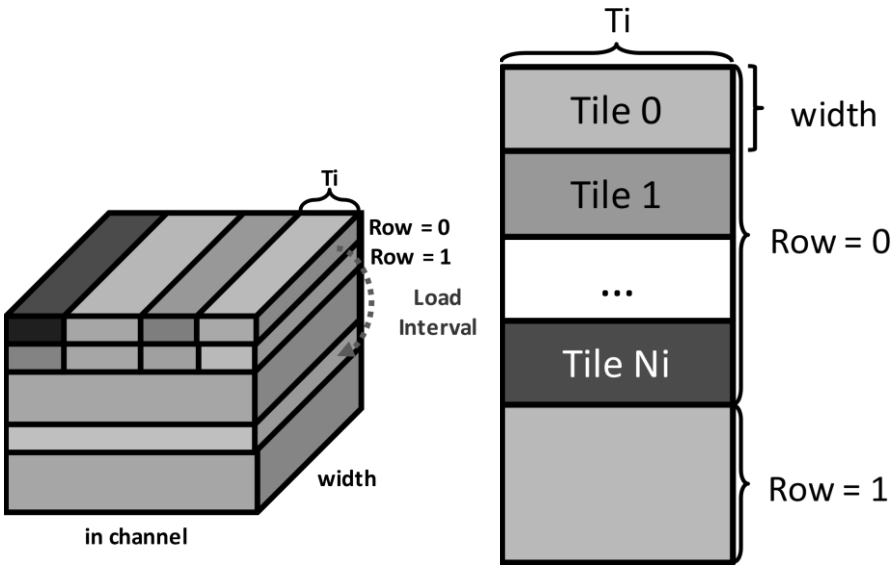

그림 5.3 Feature data의 외부 메모리 저장 방식

그림 5.3에서 하나의 feature data가 외부 메모리에서 저장되어 있는 구성을 보여주고 있다. 기본적으로 외부 메모리에서 512bit의 data width를 가지고 있는 T_i channel의 input data가 row에 대해서 저장하며 이 후 같은 row에 다음 T_i channel의 input data가 저장되는 순서를 가지고 있다. 해당 row에 대해 저장이 완료되면 다음 row가 저장되는 순서를 가지고 있다.

Adaptive row-based reuse 방식은 convolution 시에 multiple row를 동시에 저장하기 때문에 외부 메모리에서의 multiple row의 접근이

필요하다. 앞서 언급한 것과 같이 multiple row는 load interval만큼 떨어져 있는 row이기 때문에 해당 row를 load하기 위한 address는 식 (5.6)과 같다.

$$Address\ \ Space = width\ \ \times Input\ \ tiles\ \ \times Load\ \ Interval \tag{5.6}$$

따라서 input data는 T_i channel의 하나의 row에 대해서 load를 완료한 후 address apace 만큼 떨어진 row를 load 하여 input buffer에 저장한다.

외부 메모리에서 저장해야하는 data는 input data와 output data, 그리고 weight data이다. on-chip에서 convolution을 처리할 때 외부 메모리의 input data가 저장되어 있는 위치에서 data를 load 해야하며 이와 동시에 output data가 저장되어 있는 위치에서 output data를 save 해야한다. 따라서 본 가속기는 2개의 메모리 address를 통해 input data load와 output data save를 진행하며 해당 layer가 완료되면 다음 layer에서는 ping-pong 형식으로 2개의 address의 역할을 교환하여 해당 과정을 반복한다.

U-Net 네트워크를 구동하기 위해서는 몇몇 layer에서 convolution을 처리하기 위해 2개의 feature map을 concatenate하는 과정이 필요하다. 이를 위해서는 decoder 단에서 concatenate를 진행할 feature map을 보존해야한다. 그러나 ping-pong 형식으로 address를 mapping 하는 경우 보존해야하는 feature data가 layer를 진행하면서 새로운 data를 save하기 때문에 지워지게 된다. 따라서 concatenate에 사용되는 feature data는 새로운 메모리 address를 할당하여 저장한다.

제 6 장 실험 결과 및 분석

본 장에서는 앞서 논의한 adaptive row-reuse 방식의 실제 FPGA의 구현 결과와 이에 대한 분석을 진행한다. 본 가속기는 System Verilog을 통해 구현되었으며 Xilinx KCU 1500 보드에서 150Mhz에서 구동하도록 설계 되었다. U-Net 네트워크의 구동을 위하여 Tensorflow를 통한 model training을 진행하였으며 해당 weight와 parameter 값은 앞서 논의한 내용대로 각각 8 bit와 16 bit로 quantization을 진행하여 FPGA 상에서 사용된다.

6.1 기존 Row-based reuse 방식과의 비교

먼저 본 논문에서 제안한 adaptive row-reuse 방식과 기존 row-based reuse 방식과의 비교를 진행하였다. 기존 row-based reuse 방식은 on-chip buffer에 feature map 하나의 row를 load하여 convolution을 진행하는 방식이다.

그림 6.1에서 기존 row-reuse 방식과 adaptive row reuse 방식의 GOPS를 비교하였다. 실험 결과 convolution layer의 경우에 대부분의 layer가 adaptive-row reuse 방식을 적용함으로써 더 높은 연산 성능을 가짐을 확인할 수 있다. 또한 up-convolution layer의 경우에도 adaptive row-reuse 방식이 기존의 row-reuse 방식보다 2배가량 높은 GOPS를 보이고 있다.

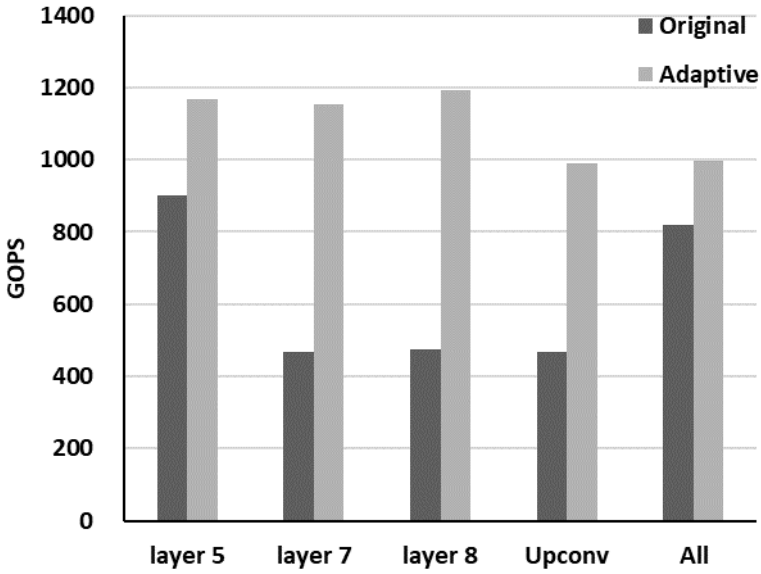

그림 6.1 기존 row-reuse 방식과의 GOPS 비교

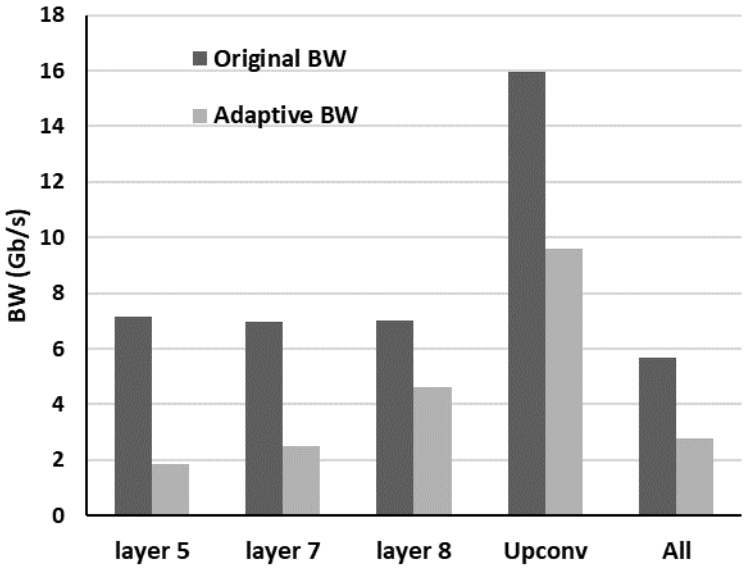

그림 6.2 기존 row-reuse 방식과의 bandwidth 비교

외부메모리와 on-chip 간의 data의 transfer 양을 나타내는 bandwidth의 측면에서도 adaptive row-reuse 방식이 기존보다 더 높은 결과를 보이고 있다. 기존 row-reuse 방식은 weight tile이 하나의 row에 대해서 reuse 되면서 convolution을 진행하지만 adaptive-row reuse 방식은 multiple row에 대해서 weight tile이 reuse 되기 때문에 weight block을 외부 메모리에서 load 하는 횟수가 더 작아진다. 그림 6.2에서 해당 결과를 보여주고 있는데 전체 필요 bandwidth양이 5.7 Gb/s에서 2.7 Gb/s로 2배가량 감소하였다. 또한 convolution layer 뿐만 아니라 up-convolution layer에서도 data의 bandwidth가 감소함을 확인할 수 있다.

6.2 On-chip buffer 크기에 따른 성능 변화

Adaptive row-reuse 방식을 적용했을 때 on-chip에서 convolution을 진행하는 multiple row의 수는 buffer size의 크기에 따라 결정된다. 따라서 buffer size의 크기에 따른 multiple row의 수와 이에 따른 성능 변화를 실험하였다.

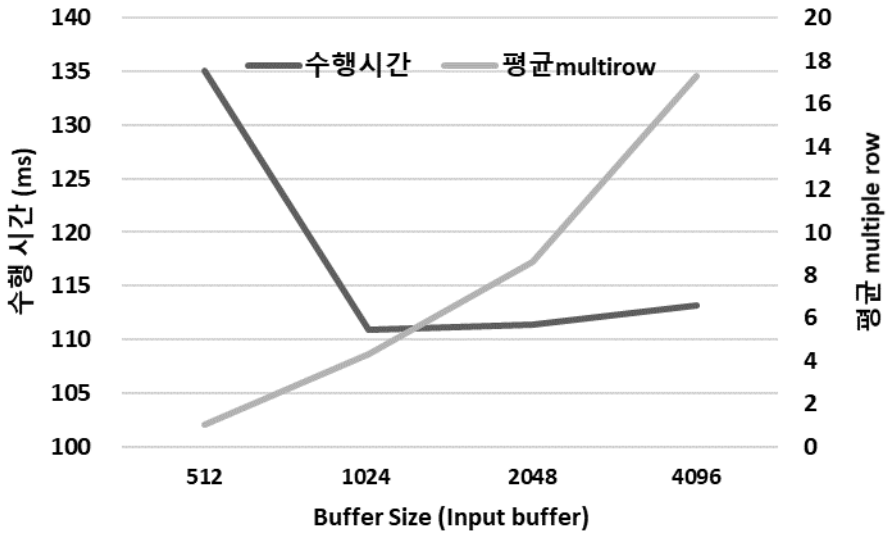

그림 6 .3 Buffer size에 따른 수행 시간 변화

그림 6.3에서 FPGA의 on-chip buffer size가 증가할수록 on-chip에 저장할 수 있는 row가 증가하기 때문에 가속기가 평균적으로 수행하는 multiple row 또한 상승한다. 그러나 한 장의 이미지를 처리하는 수행 시간은 buffer size가 증가하더라도 비례하여 감소하지 않는 모습을 보인다.

이는 buffer size의 증가와 함께 on-chip에 저장해야하는 input feature map의 row 또한 증가하기 때문에 convolution을 시작하기 위해 외부 메모리에서 load 해야하는 data의 양이 증가하게 된다. 이는 convolution의 multi row의 연산으로 얻게 되는 처리속도의 이득보다 해당 input data를 on-chip buffer에 저장하기 위한 latency가 더 크게 되어 전체 처리 속도에 영향을 끼치게 된다.

6.3 CPU, GPU와의 비교

본 논문에서 구현한 FPGA CNN 가속기의 결과를 같은 U-Net network를 구동하였을 시에 CPU와 GPU 플랫폼에서의 결과와 비교하였다. CPU와 GPU는 CNN 네트워크를 구동하는 주요 플랫폼으로 tensorflow와 같은 상용화된 딥러닝 도구의 발전으로 높은 유용성을 가지고 있다.

표 6.1은 본 가속기와 CPU와 GPU의 성능을 비교한 결과이다. 본 가속기가 CPU의 수행 속도 면에서 매우 높은 성능을 보이고 있으며 더 적은 power를 소비함으로써 에너지의 효율이 높다. GPU의 경우 본 가속기보다 높은 주파수와 병렬처리 계산의 특화로 수행 시간의 차이가 발생하지만 동작 전력 측면에서 본 가속기가 더 좋은 결과를 보이고 있다.

표 6.1 본 가속기와 CPU, GPU와의 성능 비교

Resource	CPU	GPU	FPGA
Device	Intel Xeon	Titan XP	KCU 1500
Frequency	3.5Ghz	1.5Ghz	150Mhz
Execution Time (ms)	1046.5	7.41	73.94
Precision	32-bit Float	32-bit Float	8-bit Fixed
Power (W)	193.25	238	33.28

6.4 타 연구와의 비교

본 논문에서 제안한 adaptive row-reuse 방식을 기존에 segmentation network를 FPGA에 구현한 타 연구와 비교를 진행하였다.

표 6.2 본 가속기와 타 연구와의 비교

	TRETS [22]	MDPI [31]	Ours
FPGA	ZC7Z045	Arria-10 GX1150	XCKU115
Clock (Mhz)	200	202.08	150
Precision	16-bit fixed	8-bit dynamic	8-bit fixed
Network	U-Net	SegNet	U-Net
CNN Size (GOP)	5.9	61.37	73.55
Image Size	512x512	486x360	256x256
BRAM	364	1703	796.5
DSPs	900	1515	4367
GOPS	125 (CONV) 29 (DECONV)	432.8	994.74
GOPS/DSP	0.14 (CONV) 0.033(DECONV)	0.285	0.180

타 연구 중 U-Net 네트워크와 동일한 segmentation 문제를 해결하는 Auto-encoder architecture인 SegNet을 구동한 연구를 비교하였다. 실험 결과 [22]에서 구동한 U-Net은 pruning을 통한 channel 수를 감소시킨 네트워크로 총 연산량이 적다. 그러나 본 가속기가 더 높은 GOPS를 보이고 있으며 하나의 DSP 당 연산능력을 나타내는 GOPS/DSP 지표에서도 더 뛰어난 성능을 보이고 있다. SegNet을 구동하는 [31]과 비교 시 [31]은 SegNet network가 up-convolution이 아닌 더 작은 resource를 필요로 하는 up-sampling layer를 적용하고 있다는 점, OpenCL을 통한 MAC 연산 최적화를 진행한 점으로 인하여 본 가속기 대비 더 높은 동작주파수와 GOPS/DSP를 보이고 있다. 그러나 본 가속기에서 목표로 하고 있는 on-chip buffer의 효율적인 사용으로 인하여 [31]보다 연산량 대비 더 적은 on-chip buffer를 사용하고 있다.

제 7 장 결론

본 연구는 segmentation network를 FPGA상에서 구동하기 위한 하드웨어의 설계 방식을 제안하였으며 실제 하드웨어 구현을 통해 해당 하드웨어의 성능 향상을 증명하였다. FPGA상에서 CNN 가속기의 설계를 위해서 Convolution layer의 Loop의 반복 순서에 따른 BRAM 사용량과 외부 메모리 접근 횟수를 분석하여 구동하고자 하는 network에 따른 최적의 Loop interchange Scheme을 선정하였다. 본 가속기는 하드웨어의 BRAM 크기와 feature map의 row의 길이에 따라 최대 multiple row를 BRAM에 저장하여 convolution을 진행하는 Adaptive Row-based reuse scheme을 제안하여 하드웨어의 resource를 효과적으로 사용할 수 있다. 또한 Row-based reuse scheme에서 row의 길이가 짧을 때 weight block을 외부 메모리에서 load 하는 시간과 convolution을 진행하는 시간이 overlap되지 않아 latency가 발생하는 문제를 convolution을 진행하는 row를 증가시켜 해결한다. 해당 방식은 Xilinx KCU1500 board에 구현되어 U-Net을 구동하며 실험 결과 Row-based reuse 방식과 비교 시 2배 증가한 GOPS를 보여주었다. 또한 CPU와 GPU대비 높은 전력 효율성을 보여주었다.

참고 문헌

[1] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017

[2] Y. Wang *et al.*, "Tacotron: Towards end-To-end speech synthesis," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, Mar. 2017, vol. 2017-Augus, pp. 4006–4010.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, Jun. 2017.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[5] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[6] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, 2015, vol. 9351, pp. 234–241.

[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2015, pp. 161–170.

[8] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, Dec. 2016, vol. 2016-Decem.

[9] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 101–108.

[10] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[11] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2072–2085, 2018.

[12] S. Han *et al.*, "EIE: efficient inference engine on compressed deep neural network," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 243–254, 2016.

[13] J. Li *et al.*, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, Apr. 2018, vol. 2018-Janua, pp. 343–348.

[14] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 26, no. 7, pp. 1354–1367, 2018.

[15] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[16] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[18] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[19] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, 2017.

[20] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-net: Fully convolutional neural networks for volumetric medical image segmentation," in *2016 fourth international conference on 3D vision (3DV)*, 2016, pp. 565–571.

[21] N. Ibtehaz and M. S. Rahman, "MultiResUNet: Rethinking the U-Net architecture for multimodal biomedical image segmentation," *Neural*

Networks, vol. 121, pp. 74–87, 2020.

[22] S. Liu, H. Fan, X. Niu, H. Ng, Y. Chu, and W. Luk, "Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, pp. 1–22, 2018.

[23] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks," *IEEE Trans. Very Large Scale Integr. Syst.*, 2020.

[24] D. Xu, K. Tu, Y. Wang, C. Liu, B. He, and H. Li, "Fcn-engine: Accelerating deconvolutional layers in classic cnn processors," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–6.

[25] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[26] T. Kim, M. Cha, H. Kim, J. K. Lee, and J. Kim, "Learning to discover cross-domain relations with generative adversarial networks," in *34th International Conference on Machine Learning, ICML 2017*, 2017, vol. 4, pp. 2941–2949.

[27] D. T. Nguyen, H. Kim, and H.-J. Lee, "Layer-specific Optimization for Mixed Data Flow with Mixed Precision in FPGA Design for CNN-based Object Detectors," *IEEE Trans. Circuits Syst. Video Technol.*, 2020.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015, [Online]. Available: http://www.robots.ox.ac.uk/.

[29] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 27, no. 8, pp. 1861–1873, 2019.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *32nd International Conference on Machine Learning, ICML 2015*, 2015, vol. 1, pp. 448–456.

[31] H. Huang *et al.*, "EDSSA: An Encoder-Decoder Semantic Segmentation Networks Accelerator on OpenCL-Based FPGA Platform," *Sensors*, vol. 20, no. 14, p. 3969, 2020.

Abstract

Efficient Resource Utilization for an FPGA Implementation of U-Net

Hyeonseung Je
Electrical and Computer Engineering
The Graduate School
Seoul National University

There is an increasing need and research to implement the Convolutional Neural network with an FPGA thanks to its design flexibility to design any form of hardware and low power consumption over GPU and CPU. However, most FPGA CNN accelerator studies are targeted at classification networks. Therefore, considering the recent use of segmentation networks in various areas such as autonomous driving and medical services, it is also necessary to research the operation of the segmentation network on the FPGA.

Convolution on FPGA is composed of multiple loops, and the output is calculated through MAC operation with repetition of multiple loops. The order in which loops are performed determines the required on-chip buffer size and the amount of data movement between on-chip and external memory. Therefore, it is necessary to select the suitable loop interchange scheme considering the consumption of the hardware resource in target FPGA.

In this paper, the Adaptive row-based reuse scheme is proposed that resolves the problem from original row-based reuse scheme, one of

the existing loop interchange schemes. The proposed scheme determines the maximum number of rows of feature map that can be stored in on-chip buffer based on the input and output buffer size. This method allows maximum data to be stored in on-chip buffer without being affected by the feature map size which is different by layers. Also, the latency caused by load time of weights used in next tile is solved by expanding the range of the convolution.

This paper also implemented the hardware design on FPGA to deploy u-net, the representative segmentation network, by applying the proposed adaptive row-based reuse scheme. Convolution and Up-convolution were designed to be operated in the same process element to prevent waste of resources in FPGA. It maintains the segmentation performance even with the 8 bits of weight and data by folding the weight quantization scale and batch normalization scale in batch normalization module.

In this study, tiling factor is set for width of the input feature map and multiple row so that enable to efficiently use of the on-chip buffer making difference from the existing research. The proposed design is implemented with a Xilinx KCU 1500 board, and the design achieves robust computational performance regardless of the input feature map size, and 1.2x performance increase comparing original row-based reuse scheme. Also, with the expansion of computation range, the require bandwidth between DRAM and on-chip is double decreased.

Keywords : FPGA, Segmentation network, U-Net, Loop Interchange
Student Number : 2019-26808