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

공학박사 학위논문

PIM을 지원하는 GPU system 에서의 CNN 연산 가속

Acceleration of CNN Computation on a PIM-enabled GPU system

2022 년 8 월

서울대학교 대학원

전기정보공학부

최 정 우

PIM을 지원하는 GPU system 에서의 CNN 연산 가속

Acceleration of CNN Computation on a PIM-enabled GPU system

지도 교수 이 혁 재

이 논문을 공학박사 학위논문으로 제출함

2022 년 8 월

서울대학교 대학원

전기정보공학부

최 정 우

최정우의 공학박사 학위논문을 인준함

2022 년 8 월

위 원 장 김 장 우 (인)

부위원장 이 혁 재 (인)

위 원 심 재 웅 (인)

위 원 이 채 은 (인)

위 원 손 교 민 (인)

초 록

최근, convolutional neural network (CNN)은 image processing 및 computer vision 등에서 널리 사용되고 있다. CNN은 연산 집약적인 convolutional layer와 메모리 집약적인 fully connected layer, batch normalization layer 및 activation layer 등 다양한 layer로 구성된다. 일반적으로 CNN을 처리하기 위해 GPU가 널리 사용되지만, CNN은 연산 집약적인 동시에 메모리 집약적이기에 성능이 제한된다. 또한, 고화질의 image 및 video application의 사용은 GPU와 메모리 간의 data 이동에 의한 부담을 증가시킨다. Processing-in-memory는 메모리에 연산기를 탑재하여 data 이동에 의한 부담을 줄일 수 있어, host GPU와 PIM을 함께 사용하는 system은 CNN을 처리하기에 적합하다.

먼저 convolutional layer의 연산량을 감소시키기 위해, 근사 연산을 수행할 수 있다. 그러나 기존의 근사 연산은 host GPU로 data를 load 한 후 data 간 유사도를 파악하기에, GPU와 DRAM 간의 data 이동량을 줄이지는 못한다. 이는 메모리 intensity를 증가시켜 메모리 bottleneck을 유발한다. 게다가, 근사 연산으로 인해 warp 간 load imbalance 또한 발생하게 되어 성능이 저하된다. 이를 해결하기 위해, 본 논문에서는 data 간 근사 비교를 PIM에서 수행하는 방법을 제안한다. 제안하는 방법은 PIM에서 data간 유사도를 파악한 후, 대표 data와 유사도 정보만을 GPU로 전송한다. GPU는 대표 data에 대해서만 연산을 수행하고, 유사도 정보에 기반하여 해당 결과를 재사용하여 최종 결과를 생성한다. 이때, 메모리에서의 data 비교로 인한 latency 증가를 방지하기 위해 DRAM의 bank 단과 TSV 단을 모두 활용하는 2-level PIM 구조를 제안한다. 또한, 대표 data를 적당한 address에 재배치한 후 GPU로 전송하여 GPU에서의 별도 작업 없이 load balancing을 가능하게 한다.

다음으로, batch normalization 등 non-convolutional layer의 높은 메모리

사용량으로 인한 메모리 bottleneck 문제를 해결하기 위해 PIM에서 non-convolutional layer를 수행할 수 있다. 기존 연구에서는 PIM으로 non-convolutional layer를 가속하였지만, 단순히 GPU와 PIM이 순차적으로 동작하는 상황을 가정하여 성능 향상에 한계가 있었다. 제안하는 방법은 non-convolutional layer가 ouptut feature map의 channel 단위로 수행된다는 점에 착안하여 host와 PIM을 pipeline적으로 수행함으로써 CNN 학습을 가속한다. PIM은 host에서 convolution 연산이 끝난 output feature map의 channel에 대해 non-convolution 연산을 수행한다. 역전파 과정에서 발생하는 weight update와 feature map gradient 계산에서의 convolution과 non-convolution 간 job 균형을 위해, 적절하게 non-convolution job을 분배하여 성능을 향상시킨다. 이에 더해, host와 PIM이 동시에 memory에 access하는 상황에서 전체 수행 시간을 최소화하기 위해 bank 소유권 기반의 host와 PIM 간 memory scheduling 알고리즘을 제안한다.

마지막으로, image processing application 처리를 위해 logic die에 탑재 가능한 PIM GPU 구조를 제안한다. GPU 기반의 PIM은 CUDA 기반의 application을 수행할 수 있어 딥러닝 및 image application의 처리에 적합하지만, GPU의 큰 용량의 on-chip SRAM은 logic die에 충분한 수의 computing unit의 탑재를 어렵게 한다. 본 논문에서는 PIM에 적합한 최적의 lightweight GPU 구조와 함께 이를 활용하기 위한 최적화 기법을 제안한다. Image processing application의 메모리 접근 패턴과 data locality가 보존되도록 각 computing unit에 data를 할당하고, 예측 가능한 data의 할당을 기반으로 prefetcher를 탑재하여 lightweight한 구조임에도 충분한 수의 computing unit을 탑재하여 높은 성능을 확보한다.

주요어 : processing-in-memory, convolutional neural networks, approximate computing, GPU, image processing
학 번 : 2016-20982

목 차

제 1 장 서 론 1
1.1 연구의 배경 1
1.2 연구의 내용 3
1.3 논문 구성 4

제 2 장 연구의 배경 지식 5
2.1 High Bandwidth Memory 5
2.2 Processing-In-Memory 6
2.3 GPU의 구조 및 동작 모델 7

제 3 장 PIM을 활용한 근사적 데이터 비교 및 근사 연산을 통한 Convolution 가속 9
3.1 관련 연구 10
3.1.1 CNN에서의 Approximate Computing 10
3.1.2 Processing In Memory를 활용한 CNN 가속 11
3.2 Motivation 13
3.2.1 GPU에서 Convolution 연산 시의 Approximation 기회 13
3.2.2 Approxiamte Convolution 연산에서 발생하는 문제점 14
3.3 제안하는 ADC-PIM Design 18
3.3.1 Overview 18
3.3.2 Data 간 유사도 비교 방법 19
3.3.3 ADC-PIM 아키텍처 21
3.3.4 Load Balancing을 위한 Data Reorganization 27
3.4 GPU에서의 Approximate Convolution 31
3.4.1 Instruction Skip을 통한 Approximate Convolution 31
3.4.2 Approximate Convolution을 위한 구조적 지원 32
3.5 실험 결과 및 분석 36
3.5.1 실험 환경 구성 36
3.5.2 제안하는 각 방법의 영향 분석 38
3.5.3 기존 연구와의 성능 비교 41
3.5.4 에너지 소모량 비교 44
3.5.5 Design Overhead 분석 44
3.5.6 정확도에 미치는 영향 46
3.6 본 장의 결론 47

제 4 장 Convolutional layer와 non-Convolutional Layer의 Pipeline 실행을 통한 CNN 학습 가속 48
4.1 관련 연구 48
4.1.1 Non-CONV Lasyer의 Memory Bottleneck 완화 48
4.1.2 Host와 PIM 간 Memory Scheduling 49
4.2 Motivation 51
4.2.1 CONV와 non-CONV의 동시 수행 시 성능 향상 기회 51
4.2.2 PIM 우선도에 따른 host 및 PIM request의 처리 효율성 변화 52
4.3 제안하는 host-PIM Memory Scheduling 알고리즘 53
4.3.1 host-PIM System Overview 53
4.3.2 PIM Duration Based Memory Scheduling 53
4.3.3 최적 PD_{TH} 값의 계산 방법 56
4.4 제안하는 CNN 학습 동작 Flow 62
4.4.1 CNN 학습 순전파 과정 62
4.4.2 CNN 학습 역전파 과정 63
4.5 실험 결과 및 분석 67
4.5.1 실험 환경 구성 67
4.5.2 Layer 당 수행 시간 변화 68
4.5.3 역전파 과정에서의 non-CONV job 배분 효과 70
4.5.4 전체 Network Level에서의 수행 시간 변화 72
4.5.5 제안하는 최적 PD_{TH} 추정 방법의 정확도 및 선택 알고리즘의 수렴 속도 74
4.6 본 장의 결론 75

제 5 장 Image processing의 데이터 접근 패턴을 활용한 PIM에 적합한 lightweight GPU 구조 76
5.1 관련 연구 77
5.1.1 Processing In Memory 77
5.1.2 GPU에서의 CTA Scheduling 78
5.1.3 GPU에서의 Prefetching 78
5.2 Motivation 79
5.2.1 PIM GPU system에서 Image Processing 알고리즘 처리 시 기존 GPU 구조의 비효율성 79
5.3 제안하는 GPU 기반 PIM System 82
5.3.1 Overview 82
5.3.2 Access Pattern을 고려한 CTA 할당 83

5.3.3 PIM GPU 구조 90
5.4 실험 결과 및 분석 94
5.4.1 실험 환경 구성 94
5.4.2 In-Depth Analysis 95
5.4.3 기존 연구와의 성능 비교 98
5.4.4 Cache Miss Rate 및 Memory Traffic 102
5.4.5 에너지 소모량 비교 103
5.4.6 PIM의 면적 및 전력 소모량 분석 105
5.5 본 장의 결론 107

제 6 장 결론 108

참고문헌 110

Abstract 118

표 목차

[표 3.1] ADC-PIM 동작을 위해 추가된 DRAM 명령어 26
[표 3.2] GPU 및 DRAM 시뮬레이션 변수 36
[표 3.3] 실험에 사용된 convolution layer configuration 37
[표 3.4] 실험된 configuration 37
[표 3.5] 추가된 구조의 latency, area 및 에너지 소모량 45
[표 3.6] Approximate computing으로 인한 CNN 정확도 변화 46
[표 4.1] GPU 및 DRAM 시뮬레이션 변수 67
[표 5.1] Application의 access pattern에 따른 CTA 할당 방법 84
[표 5.2] 실험에 사용된 PIM GPU configuration 94
[표 5.3] 실험에 사용된 benchmark 목록 96
[표 5.4] 실험에 사용된 GPU configuration 96
[표 5.5] SM 구성 요소들의 면적 및 전력 소모량 105

그림 목차

[그림 2.1] GPU와 interposer를 통해 연결된 HBM 구조 5
[그림 2.2] GPU의 구조 8
[그림 3.1] Approximate convolution operation의 예시 (a) 기본 convolution 연산 (b) OFM B_1에 대한 convolution 연산 (c) OFM C_1에 대한 convolution 연산 (d) OFM C_1에 대한 approximate convolution 연산 (e) row-wise convolution 연산 13
[그림 3.2] Memory 대역폭, skip level에 따른 cycle 및 IPC 변화.....15
[그림 3.3] Approximate computing으로 인해 발생하는 상황에 대한 (a) Cycle-Ops 그래프 (b) Roofline 그래프 17
[그림 3.4] ADC-PIM 시스템 overview.....18
[그림 3.5] Random projection 기반 hasing의 예시 19
[그림 3.6] Data 비교 범위와 hash vector 수에 따른 classification 정확도 및 유사하다고 판단된 data group의 비율 20
[그림 3.7] Hash calculation unit (HCU)와 bank data merge unit (BDMU)으로 구성된 ADC-PIM 구조 24
[그림 3.8] TSV data merge unit (TDMU)의 구조.....25
[그림 3.9] DRAM address mapping (a) 기존 (b) 변경된 mapping 26
[그림 3.10] Start tag, metadata, 대표 data, index data, end tag로 구성된 최종 비교 결과 형식 27
[그림 3.11] 그림 3.11 Data reorganization에 따른 warp 간 load 분배의 예시 (a) IFM data (b) weight data (c) 각 warp가 기본적으로 수행하는 convolution (d) approximate computing이 적용되지 않은 경우 각 warp가 원래 수행하는 연산 (e) 이상적으로 warp들에 job이 분배된 경우 (f) 대표 data가 주소 2와 3에 할당된 경우 (g) 대표 data가 주소 2와 5에 할당된 경우.....30
[그림 3.12] Convolution을 위한 instruction skip의 예시.....32
[그림 3.13] 연산 skip을 지원하기 위해 변경된 GPU의 pipeline 구조...33
[그림 3.14] 추가된 table들의 구조 (a) CRT (b) RWRT (c) AWRT.....35
[그림 3.15] Baseline, A1P-LB, A2P-LB의 memory access latency.....39
[그림 3.16] AHG-LB, A1P-LB, A2P, A2P-LB의 Baseline 대비 속도 향상 정도 40
[그림 3.17] AHG-LB, A1P-LB, A2P, A2P-LB의 Baseline 대비 IPC 40
[그림 3.18] Convolution에 사용되는 instruction 및 전체 instruction 중 skip된 instruction의 비율 42
[그림 3.19] ADC-PIM의 Baseline 대비 memory traffic.....42

[그림 3.20] NNCAM 대비 ADC-PIM의 속도 향상 정도.....43
[그림 3.21] NNCAM 대비 ADC-PIM의 IPC 향상 정도.....43
[그림 3.22] Convolution에 사용되는 instruction 및 전체 instruction 중 NNCAM과 ADC-PIM에 의해 skip된 instruction의 비율.....43
[그림 3.23] Baseline (B)와 ADC-PIM (P)의 에너지 소모량.....44
[그림 4.1] CONV와 non-CONV의 수행 위치 및 방법에 따른 수행시간 변화 (a) CONV, non-CONV 모두 host에서 수행하는 경우 (b) CONV는 host에서, non-CONV는 PIM에서 serial하게 수행하는 경우 (c) CONV는 host에서, non-CONV는 PIM에서 parallel하게 수행하는 경우.....51
[그림 4.2] PIM의 bank 점유 시간에 따른 host request의 BLP.....52
[그림 4.3] Host-PIM system overview 54
[그림 4.4] Cycle에 따른 host의 DRAM 접근 빈도 (bank group 단위).....54
[그림 4.5] $\boldsymbol{PD_{TH}}$ 값에 따른 host와 PIM의 수행 시간. $\boldsymbol{PD_{TH}}$ 값이 0인 경우의 결과로 normalize 되었다.....58
[그림 4.6] 최적 $\boldsymbol{PD_{TH}}$ 값 선택 알고리즘.....61
[그림 4.7] 순전파 과정에서의 host와 PIM의 pipeline 수행 방법.....64
[그림 4.8] CNN 학습 역전파 과정에서 pipeline 적으로 함께 수행되는 CONV와 non-CONV의 구분.....64
[그림 4.9] host-PIM 간 mode 전환 알고리즘에 따른 각 layer 별 CONV 및 non-CONV 수행 시간.....71
[그림 4.10] 역전파 과정에서 non-CONV job 분배 여부에 따른 각 layer 별 CONV 및 non-CONV 수행 시간.....71
[그림 4.11] 각 layer 별 Step 1과 Step 2의 non-CONV job 대비 CONV job의 비율 간 차이.....72
[그림 4.12] 전체 network level에서의 serial 수행 대비 수행 시간 변화 (a) TS는 1/2 RB로 고정, batch size를 32, 64로 변화시킨 경우 (b) batch size는 32로 고정, TS를 1/8, 1/4, 1/2 RB로 변화시킨 경우.....73
[그림 4.13] 최적 $\boldsymbol{PD_{TH}}$ 최초 추정값과 실제 최적값과의 상관 관계 74
[그림 5.1] SM에서 L2 cache로의 traffic과 SM에서 DRAM으로의 traffic의 비교.....80
[그림 5.2] SM에의 CTA 할당 예시 (a) RR 기반 할당 예시 (b) pattern aware 할당 예시.....80
[그림 5.3] Image processing benchmark들의 L1 cache miss rate.....81
[그림 5.4] PIM stack과 제안된 PIM GPU 구조.....82
[그림 5.5] CTA 할당 방법 (a) row-major (b) column-major (c) tile-based 83
[그림 5.6] 소프트웨어 기반 row-major pattern aware CTA 할당 예시 84

[그림 5.7] (a) baseline code (b) pattern-aware CTA 할당이 적용된 코드 (c) pattern-aware CTA 할당과 synchronization이 적용된 코드 87
[그림 5.8] 명시적 동기화가 없는 경우 3가지 PIM configuration에서 시간에 따라 수행된 instruction의 수 (a) 16 SMs + 8 KB L1 cache (b) 16 SMs + 32 KB L1 cache (c) 8 SMs + 8 KB L1 cache 89
[그림 5.9] Pattern-aware prefetcher의 구조 91
[그림 5.10] Warp 수에 따른 실행 시간 변화 92
[그림 5.11] 제안한 방법들을 각각의 baseline configuration에 적용했을 때 Base-8 대비 속도 향상 정도 100
[그림 5.12] Base-8, Base-8-BCS, Base-8-PL, Base-8-LT, Base-8-LT-BCS, Base-8-LT-PL, IP-8, IP-16의 Base-8 대비 속도 향상 정도 100
[그림 5.13] Base-8과 IP-8의 L1 cache miss rate 103
[그림 5.14] Base-8의 SM-L2 traffic 값으로 normalize 된 Base-8 및 IP-8의 memory traffic 103
[그림 5.15] Base-8의 값으로 normalize된 Base-8, Base-8-BCS, Base-8-LT, IP-8, IP-16의 에너지 소모량 104
[그림 5.16] Base-8, IP-8, IP-16의 전력 소모량 106
[그림 5.17] Base-8, IP-8, IP-16의 면적 당 성능 106

제 1 장 서 론

1.1 연구의 배경

최근 convolutional neural networks (CNNs)는 image processing [1], computer vision [2], 자율 주행 application [3] 등에서 매우 높은 성능을 보이며 널리 활용되고 있다. 관련 연구가 활발히 진행됨에 따라 다양하고 더 깊은 CNN 구조가 제안되고 있기에, CNN의 적용을 위해 추론 및 학습을 가속하는 것이 점점 중요해지고 있다. CNN을 가속하기 위해 대표적으로 GPU가 사용되고 있지만, CNN의 높은 연산량과 memory 사용량은 GPU에서 CNN의 성능을 제한하고 있다 [4]. 또한, 점점 높아지고 있는 image 및 video application의 화질은 GPU 등의 processor와 off-chip memory간의 data movement에 의한 overhead를 증가시킨다.

Processing-in-memory (PIM) [5], [6], [7], [8], [9] 또는 near-data-processing (NDP) [10], [11]은 data movement로 인한 overhead를 감소시키기에 적합한 방법이다. 최근에 와서, TSV의 발전을 통한 3D stacked memory의 등장 및 기술의 발전으로 DRAM die 또는 logic die에 연산기를 탑재하는 PIM이 다시 주목받게 되었다. PIM을 활용하기 위해서는 PIM에 적합한 연산을 PIM에서 수행할 수 있도록 해야 한다. DRAM die에 탑재되는 PIM은 DRAM 공정을 사용하기에 host에 비해 낮은 성능을 보인다. 또한 bank level에서의 parallelism을 활용할 수 있어 DRAM 외부에서의 접근보다 훨씬 넓은 bandwidth를 활용 가능하다. 따라서, 복잡하고 computation intensive한 연산보다는 간단하고 memory intensive한 연산이 PIM에 적합하다. Logic die에 탑재되는 PIM은 DRAM die에 탑재되는 PIM보다는 상대적으로 복잡한 연산을 수행 가능하지만, 역시 logic die의 면적 및 전력 등의 제약 조건을 잘 고려해야 한다.

CNN은 convolution layer와 batch normalization, ReLU, skip connection 등으로 구성된 non-convolution layer로 구분할 수 있다. 먼저 DRAM die PIM에 적합한 연산의 후보군을 생각해보면, floating point 기반의 convolution 연산은 computation intensive하기에, DRAM die에서 가속하기에는 적합하지 않다. 따라서, 기존 연구들은 convolution 대신 transformer 또는 LSTM 등에서 주로 사용되고 convolution보다 memory intensive한 matrix-vector 연산의 가속에 집중하였다 [12], [13]. DRAM die의 subarray 단에 가속기를 탑재하여 convolution을 가속하는 연구들도 있지만 [14], 해당 연구는 weight이 -1, 0, 1 중 하나의 값을 가지는 ternary weight에 대해서만 동작 가능하여 범용성이 떨어진다. 따라서, convolution을 PIM을 활용하여 가속하기 위해서는 convolution 연산 그 자체 대신, CNN의 특성을 활용하여 convolution과 결합될 다른 연산을 PIM으로 수행하는 것이 적합하다. 반면, non-convolution layer는 Basic Linear Algebra Subroutines (BLAS) level 1 수준의 낮은 복잡도와 높은 memory intensity를 가지기에 연산 그 자체로도 PIM에 적합하다. CNN의 추론 (inference) 과정에서는 fully connected layer를 포함하더라도 non-convolution layer가 그리 큰 비중을 차지하지는 않지만, 학습 (training) 과정에서는 non-convolution layer 또한 40% 내외의 높은 비중을 차지하기에 PIM을 사용하여 non-convolution layer를 가속하는 방법이 연구되었다 [15]. 그러나, 해당 연구는 convolution과 non-convolution layer가 순차적으로 동작하는 상황만을 가정하였기에 non-convolution layer를 PIM에서 수행하였을 때의 성능 향상 정도만 파악이 가능하였으며, 전체 학습 과정에서의 추가적인 최적화 방안은 연구되지 않았다. 다음으로 logic die PIM에 적합한 구조를 생각해보면, GPU를 탑재 시 최근의 deep learning에서 널리 사용되고 있는 CUDA 등을 사용할 수 있어 deep learning을 위한 system에 적합하다. 그러나, 기존의 GPU 기반 PIM 연구들은 일부 unit만을 PIM에 탑재하거나 [16], GPU 구조의 변경

없이 processing element의 수만을 감소시키거나 [17] fixed point 기반의 간단한 PIM 용 GPU 구조를 제안하였다 [18]. 해당 연구들은 PIM의 면적 및 전력 제약 조건은 만족하지만, 일부 연산만에 적합하며 일반적인 image processing application들에 대해서는 충분한 성능을 보이지 못한다. 따라서, PIM의 제약 조건과 성능을 동시에 고려한 GPU 구조 및 변경된 구조를 활용하기 위한 guideline이 필요하다.

1.2 연구의 내용

본 논문에서는 CNN 추론 및 학습을 가속하기 위한 PIM-enabled GPU System을 제안한다. CNN을 구성하는 convolution layer와 non-convolution layer를 PIM을 활용하여 가속하며, logic die의 제약 조건을 고려한 GPU 구조 및 활용법을 제안한다.

첫째, PIM에서의 data 근사를 통해 convolution을 가속하는 방법을 제안한다. CNN은 error에 의한 영향이 적어 근사를 통해 convolution 가속할 수 있다. 그러나, 근사 연산을 수행 시 arithmetic intensity가 감소하여 memory bottleneck이 증가하며, GPU의 warp 간 load imbalance가 발생한다는 문제점이 있다. 제안 방법은 DRAM die 단에 위치한 PIM에서 data간 유사도를 비교한 후 일부 대표 data만 host로 보냄으로써 memory bottleneck을 완화시킨다. PIM에서의 data 비교에 소요되는 시간을 감소시킬 수 있는 PIM 구조를 제안하며, 비교 후 적절한 address에 대표 data를 할당함으로써 host GPU에서의 warp 간 load imbalance를 완화한다.

두번째로, PIM에서 non-convolutional layer를 수행하여 CNN 학습을 가속하는 방법을 제안한다. 이때, non-convolutional layer를 convolutional layer와 각각 host와 PIM에서 pipelining하여 수행할 수 있도록 CNN 학습 flow를 제안한다. 또한, host와 PIM이 동시에 동작할 때 host와 PIM 간의

동시 memory access로 인해 발생하는 conflict를 고려하여 전체 수행 시간을 최소화하기 위해, bank group 단위로 host와 PIM 간 접근권을 조정하는 방법을 제안한다. 해당 방법은 PIM mode로 동작한 시간과 대기 중인 host request의 수를 모두 고려하여 host와 PIM 간 bank 접근권을 조정한다. 이를 통해, convolutional 및 non-convolutional layer의 순차적 실행 대비 성능을 향상시킨다.

마지막으로, logic die의 제약 조건을 고려하면서 기존 GPU 이상의 성능을 보이는 PIM 용 GPU 구조를 위해 CNN을 포함한 image processing application이 알고리즘 적으로 일정한 data access pattern을 보인다는 점을 이용한다. Software적으로 코드를 수정하여 각 SM에 처리될 data가 access pattern에 따라 할당되게 하며, 이를 이용하여 기존 GPU의 SM에 비해 간단한 구조를 가지지만 높은 정확도의 prefetching을 활용하여 비슷한 성능을 보이는 SM을 탑재한다.

1.3 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 배경 지식을 설명하며, 3장에서는 CNN 중 convolution을 PIM으로 근사적으로 가속하는 방법을 설명한다. 4장에서는 PIM을 활용하여 CNN 학습 중 non-convolution을 가속하는 방법을 설명하며, 5장에서는 convolution 및 image processing을 처리하기 위한 logic die 용 PIM GPU 구조를 설명한다. 마지막으로 6장에서 본 연구의 결론을 맺는다.

제 2 장 연구의 배경 지식

2.1 High Bandwidth Memory

HBM은 through-silicon vias (TSVs) 기술을 활용하여 여러 memory die들을 수직으로 적층하는 2.5D 또는 3D memory이다. 각 HBM의 DRAM die들과 logic die (또는 buffer die)는 TSV를 통해 3D로 적층되어 있다. Logic die는 I/O 회로, built-in-self-test (BIST), [19, 73] 등을 포함한다. HBM은 silicon interposer를 통해 host processor와 연결된다. 그림 2.1은 GPU와 같은 host processor와 interposer를 통해 2.5D로 연결된 HBM 구조를 나타낸다. HBM2의 경우 각 DRAM die 당 4개의 pseudo channel로 구성되어 있으며, 각 channel에는 16개의 bank가 존재한다. 4개의 bank가 1개의 bank group을 구성하여 총 4개의 bank group이 존재하며, DRAM die의 가운데에는 TSV I/O 등의 periphery 회로가 존재한다. 각 bank는 data bit을 저장하는 DRAM cell array 및 row decoder, column decoder 등으로 구성된다. HBM에의 접근은 256 bit 단위로 이뤄지며, 접근을 위한 명령어는 DDR의 그것과 유사하다.

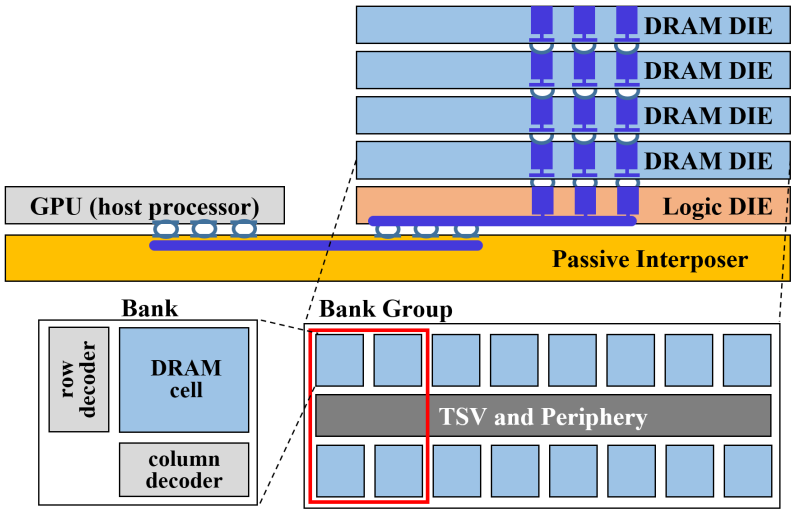

그림 2.1 GPU와 interposer를 통해 연결된 HBM 구조

2.2 Processing-In-Memory

Processing-in-memory는 memory 근처에 연산기를 위치시킴으로써 연산기와 memory 간의 거리를 가깝게 하여 data 이동에 인한 overhead를 감소시키기에 적합한 방법이다. 1990년대부터 2000년대 초반의 PIM 연구는 동일한 DRAM die에 processor와 DRAM을 집적하는 방법을 제안하였다 [5], [8], [20]. DRAM에 processor를 집적함으로써 data transfer에 소요되는 에너지 소모를 줄일 수 있었지만, 기술의 한계로 집적 cost가 커 현실적이지 못하였다. 최근에 와서, TSV의 발전을 통한 3D stacked memory의 등장 및 기술의 발전으로 DRAM die 또는 logic die에 연산기를 탑재하는 PIM이 다시 주목받게 되었다. PIM은 data 이동에 소요되는 energy 및 latency를 감소시킬 뿐 아니라 더 많은 memory bandwidth를 활용 가능하다. 3D stacked memory 상에서 PIM 연산기를 위치시킬 수 있는 장소 후보로는 logic die, DRAM die bank의 column decoder 단, bank의 subarray 단으로 3가지가 있다. 먼저, logic die에 PIM이 위치할 경우 PIM은 logic 공정을 통해 구현되며 DRAM die에 위치하는 PIM에 비해 상대적으로 제약이 적기에 복잡한 연산을 수행 가능하다. 따라서, convolution [21] 또는 graphic texture 처리 [16] 등의 연산을 logic PIM에서 수행하여 host 대비 넓은 bandwidth를 활용하고, off-chip memory 이동에 의한 overhead를 감소시킬 수 있다. Logic die에 PIM을 위치시킬 경우, logic die의 면적 및 전력 등의 제약 조건을 고려해야 한다. 다음으로, DRAM bank의 column decoder 단에 위치하는 PIM은 logic die PIM에 비해서는 면적 등의 제약이 크다. 그러나, 각 bank에 저장된 data에 접근 시 DRAM의 shared bus를 통하지 않아도 되기에 bank level에서의 parallelism을 활용할 수 있어 logic die PIM보다 넓은 bandwidth를 활용 가능하다. 따라서, computation intensive한 연산보다는 memory intensive하고 parallelism을 활용하기 좋은 matrix-vector 연산 [12] 또는 SIMD 연산을

[19] 수행하기 적합하다. 마지막으로, DRAM의 subarray 단에 위치하는 PIM은 DRAM cell level에서의 parallelism을 활용 가능하여 3가지 PIM 중 제일 넓은 bandwidth를 사용할 수 있지만, 면적 제약을 제일 심하게 받는다. 또한, floating point 연산과 같은 복잡한 연산에는 부적합하며, cell array의 구조를 활용하여 ternary weight CNN 또는 copy 등의 bit 단위 연산을 가속하는 방법들이 연구되었다 [14].

2.3 GPU의 구조 및 동작 모델

GPU는 다수의 streaming multiprocessor (SM)들로 구성되어 있으며, 각 SM은 하나의 instruction으로 여러 thread들이 동작하는 SIMT 구조를 가진다. SM 내부의 load-store (LDST) unit은 private L1 cache, 읽기 전용 texture memory, constant cache, 그리고 shared memory로 구성되어 있다 [22]. 각 SM 별로 private한 L1 cache와는 달리, L2 cache는 interconnect network를 통해 여러 SM이 공유하며, 한 개 이상의 memory controller를 통해 off-chip DRAM인 device memory와 연결된다.

GPU 제조사 중 NVIDIA에서 제공하는 CUDA의 경우, kernel 호출을 통해 GPU를 활용한다. 이하 용어들은 NVIDIA의 GPU 구조에서의 용어를 기반으로 한다. 각 kernel은 thread block 또는 cooperative thread array (CTA)라 불리는 thread들의 group들로 구성된다. CTA는 각 SM에 할당되는 기본 단위로, 각 CTA간 dependency는 존재하지 않는다. CTA는 warp라 불리는 subgroup으로 나눠진다. 각 warp는 32개의 thread들로 구성되어 있으며, 동일한 warp 내에 속한 thread들은 동일한 instruction을 수행한다. 즉, warp는 instruction fetch 및 decoding의 기본 단위이다. GPU는 memory에 접근하는데 소요되는 latency 또는 floating point 나눗셈 등의 긴 연산의 latency를 감추기 위해 여러 warp들을 바꿔 가며 수행한다.

이때, 각 warp의 정보는 빠른 context switching을 위해 register file에 저장된다. 알려진 바에 따르면, load balancing을 위해 CTA들은 SM들에 다음과 같이 round-robin (RR)에 기반한 방법으로 scheduling되어 할당된다 [23]. 먼저, 특정 SM이 CTA를 할당 받기 위해 충분한 양의 자원을 가지고 있는지 판별한다. 만일 자원이 충분하다면, CTA는 해당 SM에 할당된다. 만일 자원이 충분하지 않다면, RR 순서로 다음 SM이 충분한 자원을 가지고 있는지 판별된다. CTA scheduling은 kernel이 종료될 때까지 반복된다. SM에의 CTA 할당 가능 여부는 각 SM이 CTA를 끝내는 시간에 기반하기에, SM에의 CTA 할당 순서는 예측하기 어렵다.

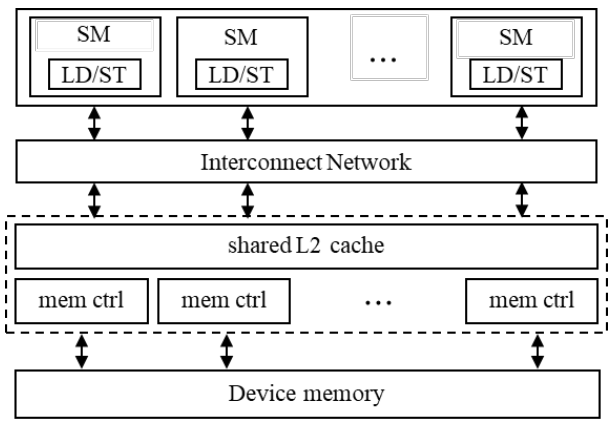

그림 2.2 GPU의 구조

제 3 장 PIM을 활용한 근사적 데이터 비교 및 근사 연산을 통한 Convolution 가속

해당 장에서는 PIM 및 GPU를 활용한 approximate computing을 통해 convolution 연산을 가속하는 방법을 제안한다. CNN은 데이터 중 일부 오류가 발생하더라도 이로 인한 성능 저하가 적다는 특성을 지닌다 (error tolerant characteristic) [24]. 이를 활용하여, approximate computing을 통해 convolution layer 등의 연산량을 감소시킴으로써 CNN을 가속할 수 있지만, 연산량 감소로 인해 memory intensity가 증가하여 memory bottleneck이 증가하게 된다. 또한, 어떤 data가 approximate 될지는 run time에 정해지기에 GPU의 warp 간 load imbalance가 발생하게 된다. 이를 해결하기 위해, 본 연구에서는 memory로의 data 이동량과 host GPU에서의 convolution 연산량을 동시에 감소시키는 Approximate Data Comparison PIM (ADC-PIM)이 탑재된 GPU system을 제안한다. ADC-PIM은 main memory에 위치한 PIM에서 data를 approximate하게 비교하고, 비교 후 비슷하다고 판단된 data 중 대표 data만을 similarity 정보와 함께 host GPU로 전송한다. GPU는 전송된 대표 data에 대해서만 convolution 연산을 수행하고, 함께 전송된 similarity 정보에 기반하여 연산 결과를 warp 단위로 재사용한다. ADC-PIM의 효율적인 동작을 위해서는 두가지 문제가 해결되어야 한다. 첫번째로, ADC-PIM으로 인한 memory access latency의 증가가 최대한 작아야 한다. 더 많은 연산량 감소를 위해서는 한번에 비교하는 data의 범위가 증가해야 하지만, 이는 PIM에서의 비교 시간 (PIM comparison latency)을 증가시킨다. PIM comparison latency를 줄이기 위해, 본 연구에서는 3D stacked DRAM의 bank 및 TSV 단을 모두 활용하는 2-level PIM 구조를 제안한다. 두번째로, 근사로 인해 발생하는

GPU 내 warp 간 load imbalance가 해결되어야 한다. 각 warp는 수행하는 data의 address가 ADC-PIM에서 전송된 대표 data의 address 일때만 연산을 수행하며, 그렇지 않은 경우에는 연산을 skip하기 때문에 warp 당 job의 양은 대표 data를 적절한 address에 할당함으로써 조절될 수 있다. 이를 위해, ADC-PIM은 data 간 similarity 비교 결과에 기반하여 적절한 address에 대표 data를 할당하는 data reorganization을 수행함으로써 GPU에서의 추가적인 작업 없이 load imbalance를 완화한다.

3.1 관련 연구

3.1.1 CNN에서의 Approximate Computing

Hardware 기반의 approximate computing 연구는 크게 voltage overscaling, approximate arithmetic unit, value similarity 기반으로 나뉜다. Voltage overscaling은 회로의 전압을 timing violation이 발생할 때까지 낮추어 연산의 오류를 일부 허용하는 대신 전력 소모량을 감소시킨다 [25], [26], [27], [28]. Approximate arithmetic unit은 precision을 낮추어 정확도를 희생하는 대신 속도를 향상시킨다 [25], [29], [30], [31], [32]. 본 논문의 제안 방법은 해당 방법들과는 orthogonal하며 병행되어 사용될 수 있다. 제안하는 방법과 비슷하게 value similarity를 활용하는 논문들의 경우, CNN에서 비슷한 연산들이 자주 반복된다는 점을 활용한다 [33], [34], [35], [36], [37]. Content addressable memory (CAM) 또는 bloom filter (BF)가 lookup table (LUT)로 사용되며 floating point unit에 인접하게 탑재된다. 자주 반복되는 연산들의 input data와 output result는 LUT에 저장된다. 연산을 수행할 때, input data는 FPU에서 연산을 수행하는 동시에 approximate pattern matching을 통해 LUT를 확인한다. 만일 LUT에

저장된 연산과 matching이 된다면, FPU 연산은 중지되며 FPU 연산 결과 대신 LUT에 저장된 output result가 해당 연산의 결과로 대체된다. 이와 같이 LUT가 FPU에 인접한 경우, 연산을 skip 할지 판단하기 위해 data는 memory에서 processor로 load 되어야 한다. 따라서, 연산을 skip하더라도 memory와 processor 간의 data 이동량은 감소되지 않는다. Approximation을 통해 연산량이 감소하였기에 memory intensity (Byte/Ops)는 증가하며, 이는 memory bandwidth에 부담을 주게 된다. 이에 더해, 해당 연구들은 연산 approximation으로 인해 발생하는 load imbalance 문제를 고려하지 않았다. 일부 sparse accelerator 관련 연구들은 centralized 된 load balancing method들을 제안하였다 [38], [39]. 그러나, 해당 방법들을 GPU에 직접적으로 적용할 경우 새로운 instruction을 생성한 후 각 warp에 배정해야 하며, 이는 instruction cache 및 buffer에서의 flush를 야기하여 성능 하락의 원인이 된다. 언급한 두가지 문제점으로 인해 기존 approximate computing 방법들은 감소한 연산량에 비해 속도 향상이 적은 반면, 제안 방법은 PIM으로 활용하여 memory bottleneck 및 load imbalance를 효과적으로 완화하여 감소한 연산량에 가깝게 속도를 향상시킨다.

3.1.2 Processing In Memory를 활용한 CNN 가속

3D stacked memory의 발전으로, 다양한 PIM 연구들이 computing unit을 logic die 또는 DRAM die에 탑재하여 CNN을 가속하는 방법을 제안하였다 [6], [7], [9], [21], [40]. Neurocube [40] 및 Tetris [21]는 CNN의 memory bottleneck을 완화하기 위해 processing elements를 3D stacked memory의 한 종류인 HMC의 logic die에 탑재하였다. 그러나, HMC와 달리 HBM의 경우 logic die에서의 memory bandwidth는 logic die 밖의 processor가 경험하는

memory bandwidth보다 크게 높지 않다는 단점이 있다. 일부 연구들의 경우, Logic die에 PIM을 탑재하는 대신, DRAM 내부의 넓은 bandwidth 및 parallelism을 활용하기 위해 DRAM die에 가속기를 탑재하였다 [12], [13]. [14], [41]. McDRAM [12] 및 Newton [13] 은 matrix-vector 곱셈을 가속하기 위해 DRAM의 bank 단에 가속기를 탑재하였으며, DrAcc [14] 의 경우 DRAM의 subarray 단에 가속기를 탑재하여 ternary weight neural network를 가속하였다. 그러나, 해당 연구들의 경우 가속기가 DRAM 내부에 위치하여 있기에 high precision 연산과 같은 복잡한 연산을 수행하지 못한다는 단점이 있다. 반면, 제안하는 방법은 convolution 연산은 기존과 동일하게 GPU에서 수행하고, PIM에서는 data 간의 comparison 만을 수행하기에 복잡한 연산 또한 처리가 가능하다. CNN을 가속하는 연구는 아니지만, Buffered compares [41] 은 DRAM 내에서 data를 비교한다는 점에서 제안 방법과 개념적으로 유사하다. Buffered compares는 big data 응용 프로그램에서 흔하게 사용되는 compare-n-op operation을 PIM에서 수행한다. 해당 operation은 주어진 범위의 data를 scan하여 single key와 비교해가며 정해진 operation을 수행하고, 연산 결과만을 host로 보낸다. 그러나 해당 연구의 경우, 한번에 하나의 data와의 비교만을 수행하기 때문에 본 연구의 target인 data 간 유사도를 비교하여 clustering을 하기 위해서는 범위 내 data들을 반복적으로 읽어야 한다. 이는 많은 시간을 소요하기에 GPU의 memory access latency를 매우 증가시켜 GPU의 성능 하락을 야기한다. 반면, 제안하는 ADC-PIM은 comparison latency를 효과적으로 감출 수 있는 방법을 제안함으로써 GPU의 성능을 저하시키지 않고 data 간 유사도를 비교한다.

3.2 Motivation

3.2.1 GPU에서 Convolution 연산 시의 Approximation 기회

CNN의 output feature map (OFM) 결과는 input feature map (IFM)과 weight 간의 convolution 연산을 통해 얻어진다. 따라서 만일 동일 weight에 대해 convolution 연산을 하는 IFM data들이 유사하다면 대표 input data에 대한 output 값만 연산하고 나머지 input의 결과는 이를 재사용하여 계산할 수 있다. 그림 3.1은 근사 convolution 연산의 개념도를 나타낸다. A_i, B_i, C_i, D_i는 IFM 및 OFM data를 나타내며, W_i는 weight을 나타낸다. 이때, OFM B_1과 C_1에 대한 convolution 연산은 그림 3.1의 (b) 및 (c)와 같이 계산된다. 만일 IFM A_i와 B_i가 유사하다면, weight W_0, W_1, W_2와 OFM C_1과의 multiply-accumulate (MAC) 연산은 (그림 3.1 (c)의 빨간색) OFM B_1의 MAC 연산 결과로 (그림 3.1 (b)의 파란색) 그림 3.1 (d)와 같이 대체될 수 있다. 즉, convolution 연산을 그림 3.1 (e)와 같이 row-wise로 분리하면 각 row 단위로 연산을 skip 가능하다.

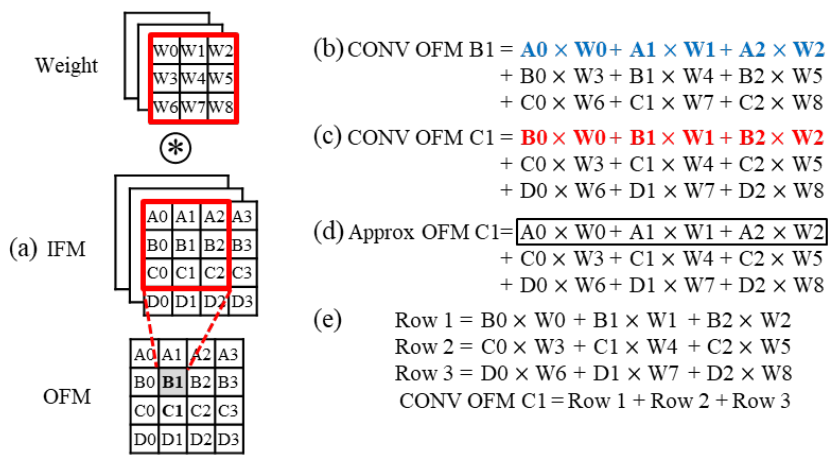

그림 3.1 Approximate convolution operation의 예시
(a) 기본 convolution 연산 (b) OFM B_1에 대한 convolution 연산 (c) OFM C_1에 대한 convolution 연산 (d) OFM C_1에 대한 approximate convolution 연산 (e) row-wise convolution 연산

GPU는 thread 들의 group인 warp 단위로 instruction을 수행하기 때문에, GPU에서 연산을 가속하기 위해서는 warp 단위로 연산을 skip해야 한다. 예를 들어, warp가 4개의 thread로 구성되어 있고 각 thread가 하나의 OFM pixel에 대해 convolution 연산을 수행하는 경우를 생각해보자. 그림 3.1에서 warp 1과 warp 2가 각각 OFM B_i와 OFM C_i $(0 \leq i \leq 3)$에 대해 convolution 연산을 수행한다고 가정하자. 상술하였듯이, IFM A_i와 B_i가 유사하다면, OFM C_1의 Row 1은 OFM B_1의 Row 1의 결과를 재사용하여 대체 가능하다. 따라서, warp 2는 Row 1에 대한 연산을 수행하는 대신, warp 1의 Row 1에 대한 연산 결과를 재사용함으로써 해당 연산을 skip할 수 있다.

3.2.2 Approximate Convolution 연산에서 발생하는 문제점

Approximate convolution을 통해 연산량을 감소시킬 수 있지만, 실제 수행 시간의 감소량은 감소된 연산량에 비해 적다. 즉, instruction pre cycle (IPC)가 연산을 skip하지 않은 baseline에 비해 감소하는 현상이 발생하며, 크게 두가지 문제점에 의해 해당 현상이 나타난다.

Memory Bottleneck 문제. 기존 방법과 같이 host로 data를 load 한 후, data의 skip 여부를 판단하는 경우 연산이 skip되더라도 data 이동량이 감소하지는 않는다. Convolution 연산은 computation intensive하다고 알려져 있지만, approximate computing을 통해 연산량은 감소하는 반면 data 이동량은 그대로인 경우 memory intensity가 증가하여 memory bottleneck을 유발하게 된다. 그림 3.2는 skip 되는 연산의 정도 (skip level)와 memory bandwidth (BW)의 변화에 따른 cycle과 IPC의 변화를 나타낸다. 막대 그래프는 수행 시간 (cycle)을 나타내며, 노란색 세모 모양은 IPC를 나타낸다. Skip 된 연산의 비율이 각각 0% (Baseline), 25% (Skip25), 50%

(Skip50) 인 3가지 skip level에 대해 실험하였다. 또한, load imbalance에 의한 영향을 없애기 위해 모든 연산들은 균일하게 skip 되었다. 이상적으로는, Skip25와 Skip50의 경우 cycle 또한 각각 25%, 50%가 감소해야 한다. 그러나, 실제로는 각각 10%, 25%만 감소한 것을 확인할 수 있다. 이는 IPC로도 확인할 수 있는데, Skip25와 Skip50의 IPC는 각각 baseline의 85%, 75% 뿐이었다. 또한 Skip50의 IPC는 Skip25의 IPC보다 낮았는데, 이는 더 많은 연산이 skip 될수록 memory intensity가 증가하여 memory bottleneck이 심해졌기 때문이다. Memory bottleneck을 완화하기 위해, bandwidth를 1.5배 및 2배 증가시킨 상황 또한 실험되었다. Baseline의 경우 memory bandwidth가 bottleneck이 아니기에 cycle이 1%만 감소하였지만, Skip25의 경우 cycle이 23% 감소하여 skip 된 연산의 비율인 25%와 거의 동일하게 감소하였으며, IPC 또한 Baseline의 98%로 회복된 것을 확인할 수 있다. 그러나 Skip50의 경우 여전히 memory bandwidth가 부족하였기에, bandwidth를 2배로 증가시키고 나서야 IPC가 회복된 것을 확인할 수 있다. 따라서, approximate computing에 의한 성능 향상을 모두 얻기 위해서는 data movement를 감소시키는 것이 중요하다.

Load Imbalance 문제. 각 warp는 해당 warp가 처리하는 data의 값에 따라 연산을 skip한다. 그러나, approximate되는 data의 분포가 균일하지 않기에 각 warp가 수행하는 job의 양은 동일하지 않고, 이는 load imbalance를 야기한다. Data의 approximate 여부를 기반으로 각 warp가 적합한 address에 access하도록 새롭게 instruction들을 생성하여 이를

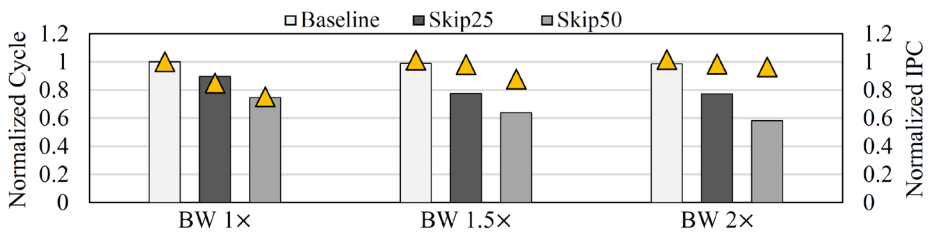

그림 3.2 Memory 대역폭, skip level에 따른 cycle 및 IPC 변화

완화할 수 있지만, 이는 instruction cache 및 buffer에 기존에 load 된 instruction들을 flush해야 하기에 GPU의 성능을 악화시킬 수 있다. 따라서, 기존에 이미 compile된 instruction을 최대한 활용하여 warp 간 load balance를 맞추는 것이 중요하다. 그림 3.3은 상술한 문제점 및 이를 해결하기 위해 제안한 방법의 효과를 Cycle-Ops 그래프와 roofline 그래프 상에서 나타낸 것이다. 그림 3.3 (a)와 (b)에서 동일한 숫자 표기는 동일한 상태를 나타낸다. 먼저, approximation을 통해 Baseline (1)에 비해 연산량이 감소하면 이상적으로는 수행 시간도 그만큼 감소해야 한다 (검은색 화살표 : 1 → 2). 그러나, memory bottleneck과 load imbalance로 인해 수행 시간은 그만큼 감소하지 못하며 (빨간색 화살표 : 2 → 3), 제안하는 방법을 통해 수행 시간을 감소시키게 된다 (파란색 화살표 : 3 → 4). 최종적으로, 본 논문의 목표는 (4)의 Ops/s (또는 IPC)를 (1)에 가깝게 향상시키는 것이다. 그림 3.3 (b)는 이를 더 자세히 설명한다. Approximation을 통해 수행 시간을 감소시키는 경우, 이상적으로는 Ops/s는 유지되면서 arithmetic intensity만 감소해야 한다 (1→2). 그러나, load imbalance (2→2')과 memory bottleneck (2'→2'')으로 인해 Ops/s가 감소하게 되고, 이를 본 논문에서는 PIM으로 memory에서 data 간 비교를 수행하여 memory bottelneck을 완화한다 (3→3'). 이에 더해, PIM에서의 data 비교로 인해 증가하는 memory access latency를 완화시키기 위한 PIM 구조를 제안하며 (3'→3''), PIM에서의 data의 address에의 재배치 (data reorganization)을 통해 load imbalance 문제를 해결한다 (3''→4).

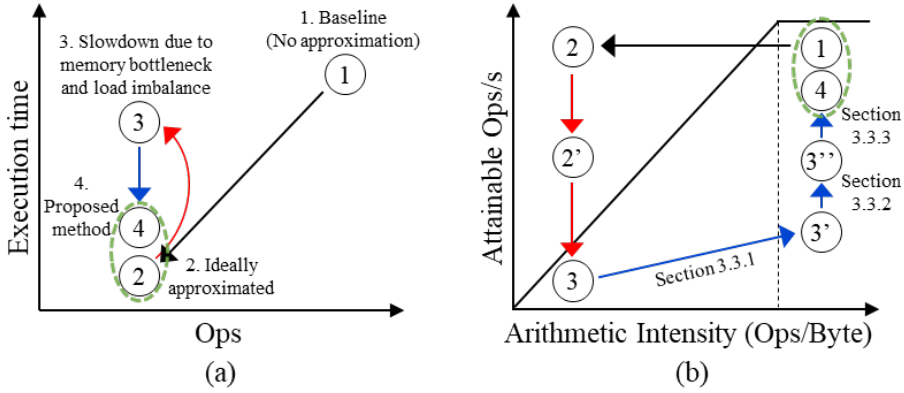

그림 3.3 Approximate computing으로 인해 발생하는 상황에 대한 (a) Cycle-Ops 그래프 (b) Roofline 그래프

3.3 제안하는 ADC-PIM Design

3.3.1 Overview

그림 3.4는 제안하는 ADC-PIM 구조의 전반적인 구조를 나타낸다. DRAM으로는 HBM을 사용하였다. ADC-PIM은 HBM의 bank I/O 단과 TSV 단에 위치하며, DRAM data transaction 단위인 32 byte씩 IFM data를 비교한다. Data transaction 간 유사도 비교를 위해서는 hash 기반의 유사도 비교 방법이 사용되었다 (3.3.2 장). Data들을 비교하는 데 걸리는 시간을 단축시키기 위해, 유사도 비교는 여러 bank에 나눠져서 수행되며, 각 bank에서의 비교 결과는 TSV 단에서 합쳐지는 2-level 구조를 가진다 (3.3.3 장). 최종 비교 결과는 대표 data들과 similarity 정보를 포함하며, host GPU로 전송되기 전에 대표 data가 적절한 address에 재배치된다 (3.3.4 장). 최종 비교 결과가 host GPU로 전송되면 GPU에서는 similarity 정보에 기반하여 대표 data에 대해서만 연산을 수행하고, 나머지 연산을 skip 한다.

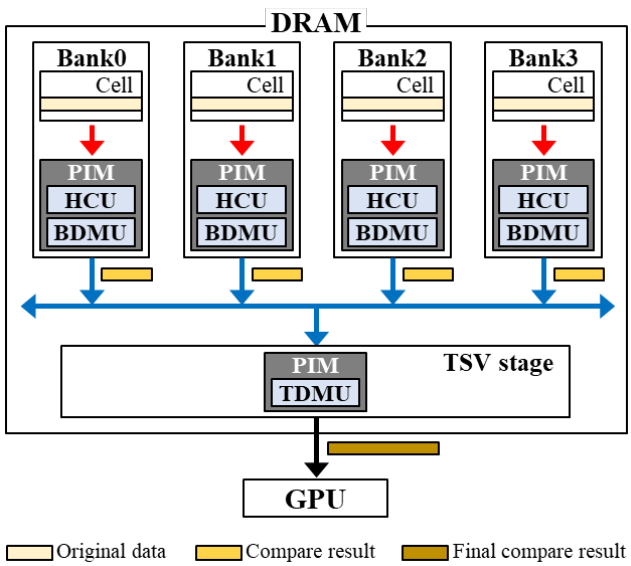

그림 3.4 ADC-PIM 시스템 overview

3.3.2 Data 간 유사도 비교 방법

Random Projection 기반 비교. 상술하였듯이, data 간 유사도 비교는 DRAM의 atom transaction 단위로 수행되며, 이하 "data group" 또는 "transaction" 이라고 명명한다. Data group 간 비교를 위해, random projection [42] 기반의 hashing 방법이 사용되었다. Data group은 32 byte로, floating point 기준으로 8개, 8 bit integer 기준으로는 32개 data로 구성되어 있기에 고차원 벡터로 볼 수 있다. Random projection은 고차원 벡터를 binary 벡터로 mapping하는 방법으로, 고차원 벡터 간의 유사도 비교에 효과적이라고 알려져 있다. 또한 hash 값만으로 data 간 유사도를 판별할 수 있기에, data 값 간 직접 비교를 할 필요가 없다. 따라서 제안하는 방법은 random projection을 사용하여 data 간 유사도를 판별한다. 임의의 k개의 임의의 d차원 벡터 v_i $(1 \le i \le k)$에 대해, 어떤 d차원 벡터 x는 k차원의 binary 벡터 $h(x) = (h_{v1}(x), h_{v2}(x), \dots h_{vk}(x))$로 hashing 된다. 각 $h_{vi}(x)$ 값은 식 (3.1)과 같이 v_i와의 내적 값을 통해 계산된다.

$$h_{vi}(x) = \begin{cases} 1, & if\ v_i \cdot x \ge 0 \\ 0, & if\ v_i \cdot x < 0 \end{cases} \tag{3.1}$$

Hashing을 통해 얻어진 binary 벡터는 binary 숫자로 된 hash index로 취급되며, 동일한 hash index를 가지는 data group은 유사하다고 판단된다. 예를 들어, 그림 3.5에서 data group A와 data group C는 동일한 hash index를 가지기에 유사하다고 판단되지만, data group A와 B는 다른 hash index를 가져 유사하지 않다고 판단된다.

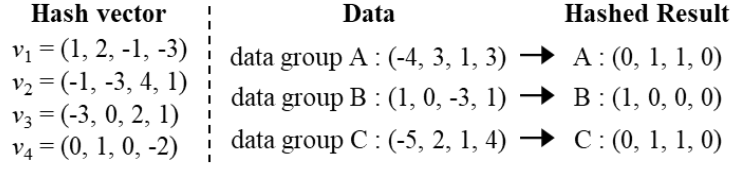

그림 3.5 Random projection 기반 hashing의 예시

Data 비교 범위 선정. 비슷하다고 판단되는 data group의 비율은 비교되는 data의 범위와 random projection에 사용되는 hash 벡터의 수에 따라 달라진다. 그림 3.6은 VGG-16의 두번째 convolution layer에 대해 approximation을 적용한 경우 classification accuracy와 유사하다고 판단된 data의 비율을 나타낸다. 동일 비교 범위에 속한 data group 간에만 비교가 수행되었으며, 유사한 data group들을 동일 cluster로 취급하여 해당 cluster의 centroid 값으로 연산을 수행하여 결과 값을 대체하여 정확도를 판단하였다. 그림 3.6의 범례의 숫자는 비교 범위의 크기와 사용된 hash 벡터의 수의 범위를 나타낸다. 정확도가 동일한 경우 비교 범위가 증가할수록 유사하다고 판단된 data의 비율이 증가하는 것을 확인할 수 있다. 예를 들어, 비교 범위가 4개의 data group인 경우 71% 정확도에서 57%의 data group이 유사하다고 판단되었지만, 동일 정확도에서 비교 범위가 32개의 data group인 경우 69%의 data group이 유사하다고 판단되었다. 따라서, 주어진 정확도에서 더 많은 data를 approximate하기 위해서는 data 비교 범위를 증가시켜야 한다. 그러나, 너무 큰 data 비교 범위를 사용할 경우 두가지 문제점이 있다. 첫번째로, 비교 범위가 클수록 data 비교에 소요되는 시간이 증가하며, 이는 GPU의

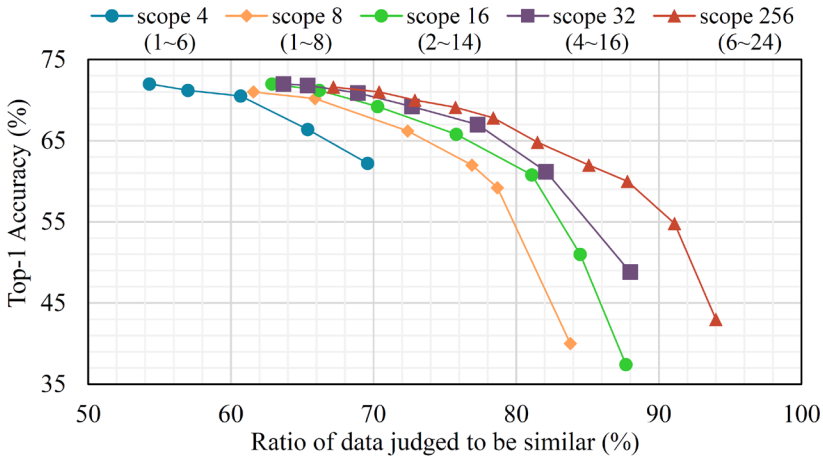

그림 3.6 Data 비교 범위와 hash vector 수에 따른 classification 정확도 및 유사하다고 판단된 data group의 비율

시간이 길어진다. 비록 GPU가 latency hiding을 통해 긴 시간이 소요되는 operation들의 latency를 효과적으로 감출 수 있지만, memory access 시간이 일정 이상으로 증가하게 되면 성능 저하를 피할 수 없다. 두번째로, 비교 범위가 일정 이상으로 증가할 경우 유사하다고 판단되는 data의 비율이 크게 늘어나지 않는 반면, 근사에 의한 정확도 감소를 회복하기 위해 더 많은 hash vector의 수가 필요하게 되고, 이는 PIM의 area 및 power overhead와 비교에 소요되는 cost를 증가시킨다. 예를 들어, 비교 범위가 32인 경우와 256인 경우를 비교하면 71%의 정확도를 위해 필요한 hash vector의 수는 각각 14개, 24개로 70% 넘게 차이가 나는 반면, 유사한 data group의 비율은 2.8% 밖에 차이 나지 않는다. 이를 고려하여, 본 논문에서는 32개의 data group을 비교 범위로 설정하였다.

3.3.3 ADC-PIM 아키텍처

ADC-PIM은 hash calculation units (HCU), bank-level data merge unit (BDMU), 그리고 TSV level data merge unit (TDMU)로 구성된다. HCU와 BDMU는 DRAM의 각 bank 단에 위치하며, TDMU는 TSV 단에 1개 위치한다. HCU는 data group의 hash 결과를 계산하며, BDMU와 TDMU는 hash 결과를 기반으로 동일 hash 값을 가지는 data group들을 clustering한다. 대표 data group의 값을 계산하기 위해 필요한 data group들의 합과 centroid 값은 각각 BDMU와 TDMU에서 계산된다.

2-level PIM 구조. Data group들 간 유사도를 판별하기 위한 제일 단순한 방법은 비교 범위에 속하는 모든 data group들을 하나의 DRAM bank에서 비교하는 것이다 (1-level PIM으로 명명). 그러나, 그와 같은 구현은 비교 범위 내 모든 data를 하나의 bank에서 차례로 읽어야 하기에 data comparison latency를 증가시킨다. 이에 비해 2-level PIM

구조는 각 bank가 전체 비교 범위 중 일부 data만을 읽으면 되기에 1-level PIM에 비해 낮은 comparison latency를 지닌다. 3.3.2 장에서 상술하였듯이, hash 기반 비교 방법의 경우, 유사도 판별을 위해 직접적인 data 값 간의 비교가 필요하지 않으며, 비교 후에도 data group의 hash 값은 동일하다. 즉, data group의 hash 값은 divide-and-conquer 방법을 적용하더라도 변하지 않으므로, 비교 범위 내 모든 data group들을 한번에 비교해서 얻은 결과와 비교 범위를 여러 bank에 나눈 후 비교한 결과가 동일하다. 이러한 특성은 전체 비교 범위를 일부분 씩 여러 bank에 나눈 후 각 bank에서의 부분 비교 결과를 TSV 단에서 합쳐 최종 비교 결과를 생성하는 2-level PIM 구조를 가능케 한다. 2-level PIM 구조의 comparison latency를 최소화하기 위해, 비교에 사용될 bank의 수는 적절하게 선택되어야 한다. PIM에서의 비교에 소요되는 시간은 DRAM cell에서 data를 읽는 시간과 (*Process 1*, 그림 3.4의 빨간색 화살표) 비교된 data를 DRAM shared bus를 통해 TSV 단으로 이동시키는 시간의 (*Process 2*, 그림 3.4의 파란색 화살표) 합으로 계산될 수 있다. 소요 시간은 이동하는 transaction의 양에 비례하기에, 총 read traffic을 계산하여 추정할 수 있다. 각 bank에 동일하게 비교 범위가 나눠져 할당되기에, *Process 1*의 traffic은 (T_{P1}) 전체 traffic을 bank 수로 나눈 것과 동일하다. *Process 2*의 경우, 각 bank는 hash 값과 cluster 내 data group들의 합을 TSV로 전송한다. 모든 bank가 DRAM shared bus를 공유하기에, 모든 bank의 output traffic의 합이 Process 2의 전체 traffic (T_{P2})과 동일하다. 이를 고려하면, 전체 traffic (T_{tot}) 값은 아래 식 (3.2)와 같이 계산될 수 있다.

$$
\begin{aligned}
T_{P1} &= C/B \\
T_{p2} &= B \times (1 + (C/B) \times R_c) \\
T_{tot} &= T_{p1} + T_{p2} = \frac{C}{B} + B + C \times R_c
\end{aligned}
\tag{3.2}
$$

이때, C, B, R_c는 각각 비교 범위, 비교에 사용된 bank의 수, 비교 후 남은 data group의 비율을 의미한다. *Process 2*에서 hash 값 전송을 위한 추가적인 traffic은 bank 당 1개로 계산되었으며, 각 bank의 R_c 값은 동일하다고 가정하였다. 여기서 $C \times R_c$ 값은 bank 수 B와 상관 없기에, 전체 traffic을 최소화하는 B는 $C/B + B$ 값을 최소화하는 B이다. C가 32이고 (3.3.2 장), 산술 기하 평균에 의해 $C/B = B$ 일 때 해당 값이 최소가 되기에 B가 4 또는 8 일때 전체 traffic T_{tot}이 최소이다. GPU가 cache line을 256 byte 단위로 interleave하기에, $B = 4$를 사용하여 8개의 data group이 하나의 bank에서 비교될 수 있도록 하였다.

Hash Calculation Unit의 구조. 그림 3.7은 HCU와 BDMU를 포함하는 ADC-PIM unit의 구조를 나타낸다. HCU는 벡터 내적 연산기, 비교기, 그리고 index 생성기로 구성되어 있다. 먼저 벡터 내적 연산기는 hash 벡터와 DRAM에서 read 한 data group 간의 내적 값을 계산한다. 다음으로, 비교기는 각 내적 값을 0과 비교하여 hash 값을 설정하며, index 생성기는 각 hash 벡터와의 hash 값을 엮어 최종 hash index를 구한다. 마지막으로, 생성된 hash index와 data 값은 BDMU로 전송된다. 그림 3.7은 4개의 hash 벡터로 구성된 예시를 HCU의 나타낸다. 각 hash 벡터와의 내적 값이 -1.5, 2, -4.9, 3.1 이라면, hash 값은 0, 1, 0, 1이 되며 최종 hash index는 0101_2=5가 된다. Hash 벡터는 유사도 확인 성능을 증가시키기 위해 서로 orthogonal하게 Gram-Schmidt 방법에 의해 생성된다 [43]. 정확도는 hash 벡터의 수를 변경하여 조정될 수 있다. Hash 벡터의 수가 증가하면 정확도는 증가하지만, 근사되는 data의 양이 감소하며, hash 벡터의 수가 감소할 때는 반대의 경우가 발생한다. 본 논문에서는 1% 미만의 정확도 감소를 위해 16개의 hash 벡터를 사용하였다.

Bank Data Merge Unit의 구조. BDMU는 index buffer, cluster table, 합 연산기로 구성된다. Index buffer는 각 data group의 hash index를 data group을

읽은 순서대로 저장한다. Cluster table은 hash index와 해당 index를 가지는 data group의 수와 data group들의 합을 저장한다. 합 연산기는 8개의 덧셈기로 구성되어 있다. Cluster table의 각 entry는 각각 16 bit, 3 bit, 32 byte인 hash index, 해당 index에 속하는 data group의 수, 해당 index의 data group들의 합으로 구성되어 있다. BDMU가 HCU로부터 계산된 hash index와 data group을 받으면, hash index를 index buffer의 끝에 저장하는 동시에 cluster table을 업데이트 한다. 해당 index에 속하는 data group의 수가 1 만큼 증가하며, data group들의 합이 element-wise로 합 연산기에 의해 계산되어 업데이트 된다. 그림 3.7에서, 새로 업데이트 된 부분은 빨간색으로 표기되어 있다. 새로 들어온 data group의 hash index가 5이기에, index 5가 index buffer의 끝에 저장되며, hash index 5에 해당하는 data group의 수와 합이 cluster table에 업데이트 된다. 해당 bank에서의 비교가 끝나면, cluster table과 index buffer에 저장된 값은 DRAM bank 밖으로 내보내 진다.

TSV Data Merge Unit의 구조. TDMU는 BDMU와 비슷한 구조를 가지지만, cluster table의 entry의 구성이 다르며 합 연산기 대신 다른 종류의 연산기를 탑재하였다. 또한 data reorganization 작업에 사용되는 대표 address 연산 모듈 (Representative Address Calculator, i.e., RAC)과 결과

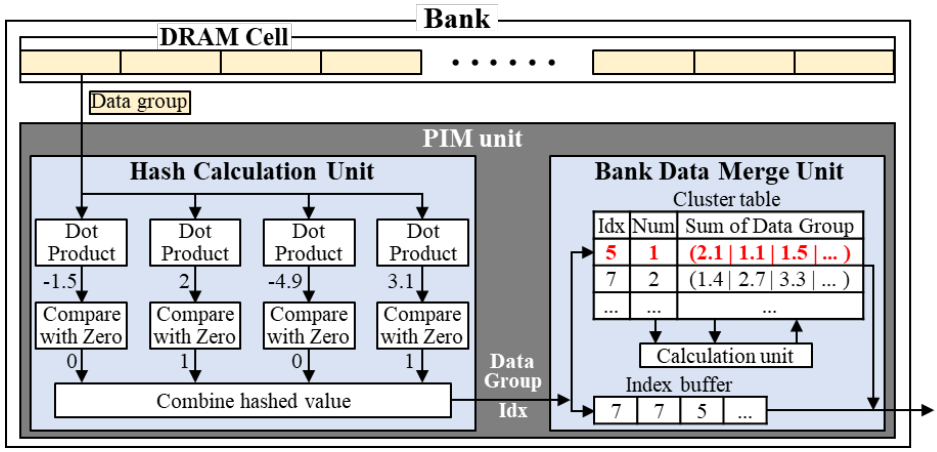

그림 3.7 Hash calculation unit (HCU)와 bank data merge unit (BDMU)으로 구성된 ADC-PIM 구조

transaction 생성기 모듈(Result Transaction Generator, i.e., RTG)을 포함한다. TDMU의 연산기에는 동일 index에 속하는 data group의 centroid 값을 계산한다. 이를 계산하기 위해, TDMU의 연산기는 8개의 덧셈기, 8개의 곱셈기, 그리고 32 entry로 구성된 1개의 LUT로 구성되어 있다. 복잡한 나눗셈기 대신, 나눗셈은 역수와의 곱으로 구현되며, 1부터 32까지의 역수 값이 LUT에 저장되어 있다. 비교가 끝난 후, RAC는 대표 address를 계산하고, RTG는 계산된 대표 address, index buffer, centroid 값으로 구성된 최종 비교 결과를 연속된 transaction으로 생성한다. RTG는 double buffering 구조로 구현되어 있어, 현재 생성된 transaction을 전송하면서 새롭게 전송될 transaction을 생성할 수 있다.

DRAM Address Mapping 방법. ADC-PIM에서의 data 비교를 위해, DRAM address mapping은 그림 3.9 (a)에서 그림 3.9 (b)로 변경되었다. 비교 범위 내의 data들을 동일한 channel에 할당하기 위해 channel bit들은 MSB 쪽으로 이동되었으며, 2-level PIM 구조에서 여러 bank를 동시에 활용하기 위해 bank bit 중 일부가 LSB 쪽으로 이동되었다.

ADC-PIM 명령어 및 동작 흐름. ADC-PIM을 지원하기 위해, 3개의 PIM 용 DRAM 명령어가 표 3.1과 같이 추가되었다. ADC-PIM의 동작 흐름은 다음과 같다. 먼저, GPU가 PIM comparison을 위한 read 요청을 보내게 되면, memory controller는 RDBP 명령어를 보내 해당하는 bank들이 data를 읽고 비교할 수 있도록 한다. DRAM의 명령어 bus가 botteleneck이

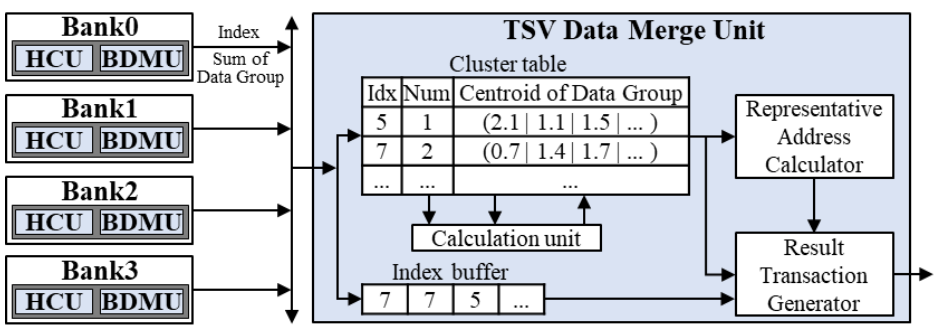

그림 3.8 TSV data merge unit (TDMU)의 구조

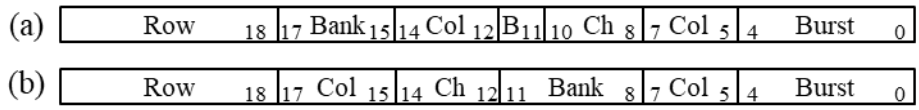

그림 3.9 DRAM address mapping (a) 기존 (b) 변경된 mapping

표 3.1 ADC-PIM 동작을 위해 추가된 DRAM 명령어

CMD	Description
RDBP	Read and compare data in each bank PIM
MBT	Move data from bank to TSV PIM and merge
RDT	Read merged data from TSV PIM

되는 것을 방지하기 위해, 한번의 RDBP 명령어로 8개의 DRAM transaction이 읽어지고 비교 된다. 이때, DRAM cell에서 data를 읽는 것과 HCU 및 BDMU에서 data들을 비교하는 것은 병렬적으로 수행된다. Bank-level PIM에서의 data 비교가 완료된 후, hash index와 data group의 합으로 구성된 data는 MBT 명령어를 통해 두번째 level인 TDMU로 전송된다. Memory controller는 PIM comparison 이후에 감소한 data group의 수를 알 수 없기 때문에, 비교 후의 정보를 포함하는 제일 앞의 transaction을 먼저 읽어 비교 후 감소된 transaction의 수를 파악한다. 이후, memory controller는 감소된 transaction의 수만큼 읽기 명령어를 실행하여 TDMU로의 data 이동을 수행한다. TDMU는 수신한 transaction을 기반으로 cluster를 업데이트 한다. 비교를 수행한 모든 bank에서의 transaction들이 TDMU로 이동하고 TDMU가 merge 동작을 완료한 후, 최종 비교 결과가 RDT 명령어를 통해 host GPU로 전송된다. MBT 명령어의 경우와 동일하게, 제일 앞의 transaction을 먼저 읽어 정보를 파악한 후, 감소된 transaction 수만큼 나머지 data group을 읽는다.

3.3.4 Load Balancing을 위한 Data Reorganization

TDMU에서, 최종 비교 결과는 그림 3.10과 같이 연속된 transaction들로 변환되어 GPU로 전송된다. 최종 비교 결과는 start tag (32 bit), metadata (32 bit), 대표 data (i.e., rep data, 32 byte), index data (6 bit), 그리고

그림 3.10 Start tag, metadata, 대표 data, index data, end tag로 구성된 최종 비교 결과 형식

end tag (32 bit)으로 구성된다. 이때, 대표 data는 ADC-PIM에서 비슷하다고 판단되어진 cluster의 centroid 값을 의미하며, TDMU의 cluster table 중 “Centroid of Data Group”에 해당한다. Index data는 ADC-PIM에서 계산되어진 각 data group의 hash index 값을 나타낸다. Metadata는 32개의 각 data group이 대표 data group 인지 아닌지를 저장하며, 각 bit은 i번째 data group이 대표적인지 아닌지를 의미한다. 만일 i번째 bit 값이 1이면 data group의 대표 data 값과 index data가 모두 기록되며, bit 값이 0인 경우 index data만 기록된다. Start tag와 end tag는 비교 결과의 시작과 끝을 나타내며, 특정 고정된 bit들로 구성된다. Tag 값이 일반 data 값과 겹치는 것을 방지하기 위해, [44]에서 제안된 충돌 방지 알고리즘이 사용되었다. 만일 data가 tag와 겹친다면, 32번째 bit이 1로 대체되고, 원래 bit 값은 memory의 특정 구역에 저장된다. Tag의 길이가 32 bit이기에, 충돌 확률은 $1/2^{32}$ 로 매우 낮아 이로 인한 overhead는 무시할 수 있다. 어떤 address의 data group에 대표 data 값이 할당되는지에 따라 GPU에서의 동작이 변하게 된다. 만일 warp가 접근하는 data가 대표 data라면 warp는 연산을 수행하지만, 아닌 경우 연산을 skip한다. 따라서, 적절한 대표 address (i.e., centroid 값이 할당되는 address)를 선정함으로써 host

GPU에서의 load balancing을 달성할 수 있다. 3.3.3 장에서 설명되었듯이, 대표 address는 TDMU의 RAC 모듈에 의해 계산된다. 비교 후 감소된 data group의 수가 N_d이고, 비교 범위가 C인 경우 (본 논문의 경우 32), 각 centroid 값들은 식 (3.3)에 의해 계산된 i번째 address에 할당된다.

$$i = 3k + 2 \ (mod\ C) \ \ (0 \leq k < N_d) \tag{3.3}$$

그림 3.11은 대표 address 할당에 따른 GPU에서의 approximate computing 상황을 나타낸다. Data group A는 DG_A로 표기되었다. 그림 3.11 (a)의 data $A_i, B_i, C_i, \dots H_i$는 각각 $DG_A, DG_B, DG_C, \dots DG_H$로 표기되며, convolution의 IFM data들을 나타낸다. 단순화를 위해, data group 단위로 address가 할당되며, DG_A 부터 DG_H가 순서대로 address 0 부터 7에 할당된다고 가정하였다. Data group은 ADC-PIM에서의 approximate comparison을 통해 2개의 data group DG_X 및 DG_Y로 clustering 된다. DG_X와 DG_Y가 할당될 address는 ADC-PIM의 data reorganization에 의해 정해진다. 그림 3.11 (b)는 convolution의 weight을 나타내며, weight의 각 row는 Weight Row (WR)로 명명된다. OFM DG_B에 대한 convolution 결과는 그림 3.11 (c)와 같다. OFM pixel B1에 대한 convolution 결과는 (A_0, A_1, A_2)와 WR_0, (B_0, B_1, B_2)와 WR_1, (C_0, C_1, C_2)와 WR_2의 내적 값의 합으로 얻어진다. DG_B의 다른 data에 대해서도 비슷하게 적용하면, OFM DG_B에 대한 convolution 값은 $DG_A * WR_0, DG_B * WR_1, DG_C * WR_2$의 1D convolution 값의 vectored 합으로 나타낼 수 있다. 더 직관적인 이해를 위해, WR_0, WR_1, 그리고 WR_2와의 convolution은 각각 빨간색, 파란색, 노란색으로 표시하였다. 그림 3.11 (d)는 6개 warp가 있는 상황에서 data 근사가 없는 기본적인 convolution을 나타낸다. 셀 내 수식의 첫번째 항은 address를 나타낸다. 예를 들어, warp 1의 파란색 셀 (i.e., $2 * WR_1$)은 address 2에 해당하는 data group DG_C와 WR_1과의 1D convolution을 의미한다. Approximate computing이 적용된 경우, DG_X와 DG_Y에 대한 convolution 연산만이 수행되면 된다.

2개의 data group과 3개의 WR에 대해 convolution 연산이 수행되기에, 총 수행되어야 하는 연산의 수는 6개이다. 6개의 warp가 존재하기에, 이상적인 상황이라면 그림 3.11 (e)와 같이 각 warp가 1개의 연산을 수행하면 된다. 이런 이상적인 상황을 달성하기 위해, X와 Y가 적절한 address에 할당되어야 한다. 그림 3.11 (f)는 DG_X와 DG_Y가 address 2와 3에 할당된 경우이다. 이 경우, 그림 3.11 (d)와 같이 각 warp 가 원래 수행한 연산 중 재사용 될 수 있는 것은 warp 0의 $2 * WR_2$, warp 1의 $2 * WR_1$과 $3 * WR_2$, warp 2의 $2 * WR_0$와 $3 * WR_1$, 그리고 warp 3의 $3 * WR_0$이다. Warp 4와 warp 5의 경우 기존 연산 중 재사용 가능한 것이 없기에 해당 warp 들에는 연산이 할당되지 않은 반면, warp 1에는 2개의 연산이 할당된다. 따라서, warp 간 load를 맞추기 위해 새로운 instruction이 생성되어 각 warp들에 분배되어야 한다. 그림 3.11 (g)는 제안하는 data reorganization에 의해 새로운 instruction의 생성 없이 load balancing이 된 경우를 보여준다. DG_X, DG_Y는 각각 address 2와 5 (즉 식 (3.3)에서 $C = 8, N_d = 2$인 경우)에 할당되었다. $2 * WR_0, 2 * WR_1, 2 * WR_2, 5 * WR_0, 5 * WR_1, 5 * WR_2$ 가 각각 warp 2, 1, 0, 5, 4, 3의 기존 instruction에 의해 수행될 수 있으며, 모든 warp가 동일하게 1개의 연산만을 수행하게 된다.

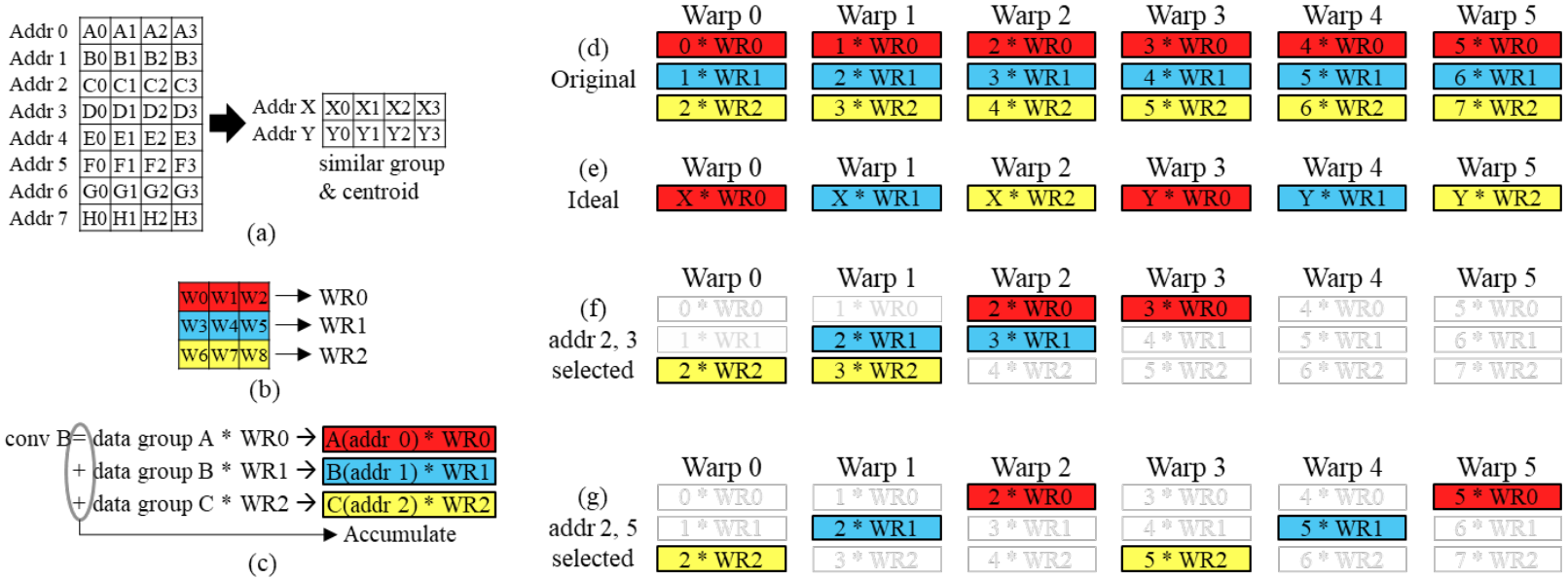

그림 3.11 Data reorganization에 따른 warp 간 load 분배의 예시 (a) IFM data (b) weight data (c) 각 warp가 기본적으로 수행하는 convolution (d) approximate computing이 적용되지 않은 경우 각 warp가 원래 수행하는 연산 (e) 이상적으로 warp들에 job이 분배된 경우 (f) 대표 data가 주소 2와 3에 할당된 경우 (g) 대표 data가 주소 2와 5에 할당된 경우

3.4 GPU에서의 Approximate Convolution

3.4 장에서는 GPU에서의 approximate convolution의 수행 방식 및 이를 지원하기 위한 GPU 구조를 제안한다. GPU는 ADC-PIM에서 보내진 정보에 기반하여 근사 가능한 data를 파악한다. 각 warp들은 근사 가능한 data에 대한 연산을 skip하며, 대신 다른 warp의 결과를 재사용한다 (3.4.1 장). ADC-PIM의 data reorganization 덕분에 load balancing은 새로운 instruction의 생성 없이 행해진다. 3.4.2 장에서는 approximate convolution을 지원하기 위해 추가 및 변경된 구조를 설명한다.

3.4.1 Instruction Skip을 통한 Approximate Convolution

Baseline code로는 direct convolution을 사용하였다. Block 단위로 동기화를 하며 연산하며, 하나의 thread가 하나의 output feature map에 대한 convolution 연산을 수행한다. 동일 warp내 thread들은 연속적인 pixel에 대해 연산을 수행한다. 제안하는 GPU 구조에서, 연산의 skip 및 재사용은 instruction skip과 register renaming을 통해 행해진다. Instruction skip은 program counter (PC)를 변경함으로써 수행된다. Vector 형태인 warp의 operand value들을 모두 저장하는 것은 많은 저장 공간을 필요로 하지만, register renaming을 통해 이런 추가적인 저장 공간의 필요성을 없앨 수 있다. 그림 3.12는 그림 3.1 (e)에 instruction skip과 register renaming을 적용하는 pseudo code를 나타낸다. 그림 3.1 (e)의 row를 수행하는 instruction들의 집합은 ‘InstBlock’으로 명명하였다. PC가 100인 경우, 먼저 address B0를 확인하여 data가 근사 가능한지 확인한다. 만약 근사 가능한 것으로 판명된 경우, 해당 InstBlock의 나머지 instruction들은 skip된다. 이때, next PC는 다음 InstBlock의 첫번째 instruction의 주소인 136으로 설정된다. Skip 된 InstBlock의 결과가 저장될 예정이었던 register

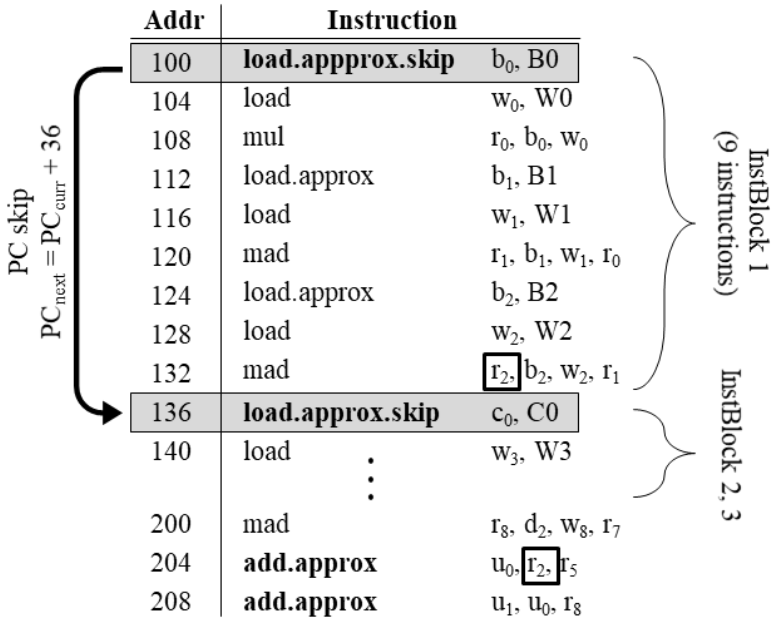

그림 3.12 Convolution을 위한 instruction skip의 예시

132의 instruction의 destination register (그림 3.12의 r2)는 재사용 될 결과값을 계산하는 warp의 destination register로 renaming 된다. 각 InstBlock의 연산이 끝나면, InstBlock의 결과들은 address 204와 208에서 합해진다. 이때, skip 된 InstBlock의 결과 register에 접근하는 대신, rename 된 register에 접근하여 값을 가져온다. 또한, InstBlock의 첫번째 instruction, data를 load하는 instruction, InstBlock의 결과들을 합하는 instruction들을 다른 instruction들과 구분하기 위해 load.approx.skip, load.approx, add.approx instruction을 새로 추가하였다.

3.4.2 Approximate Convolution을 위한 구조적 지원

그림 3.13은 instruction skip과 register renaming을 지원하기 위해 변경된 GPU의 pipeline을 보여준다. 회색으로 칠해진 부분들이 새롭게 추가된 모듈들이며, 비교 결과 table (comparison result table, i.e., CRT), 대표 warp의 결과를 저장하는 table (representative warp result table, i.e., RWRT),

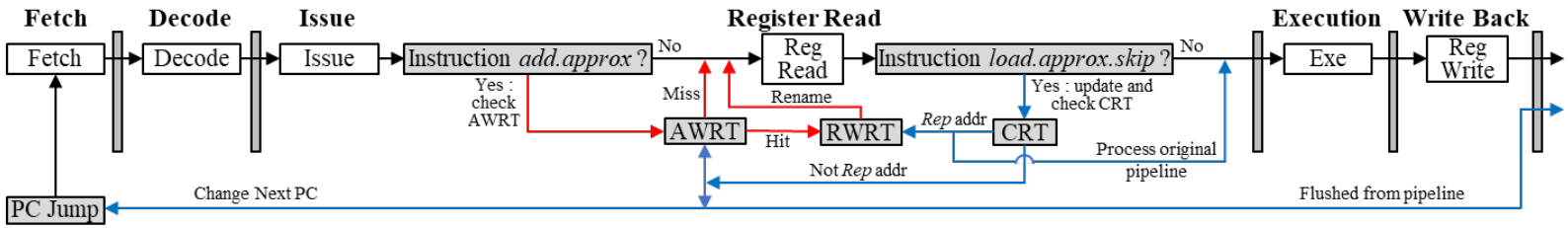

그림 3.13 연산 skip을 지원하기 위해 변경된 GPU의 pipeline 구조

연산이 skip 된 warp의 renaming 결과를 저장하는 table (approximated warp rename table, i.e., AWRT), 그리고 PC jump unit으로 구성된다. CRT, RWRT, AWRT는 cache와 비슷한 구조로, 각각 32, 64, 64개의 entry로 구성되어 있다. 각 정보는 threada block (또는 CTA) 단위로 저장되며, tag와 저장되는 data는 그림 3.14와 같다. Tag에 해당하는 부분은 짙은 글씨로 표기되어 있다. 만일 thread block이 작업을 완료하면, CRT, RWRT, AWRT에 저장되어 있는 해당 thread block의 data는 무효화된다. 변경된 pipeline은 load.approx.skip (그림 3.13의 파란색 선)과 add.approx (그림 3.13의 빨간색 선)에 대해서만 기존 GPU와 다른 동작을 한다. Data group 내 data의 수와 warp의 thread 수를 동일하게 하기 위해, warp의 크기는 8 thread로 설정되었다.

load.approx. 만일 load.approx 명령어가 탐지되는 경우, 먼저 CRT가 업데이트 된다. CRT는 ADC-PIM에서의 비교 결과에 기반하여 [address]를 tag로, [hash index, 대표 address 여부]를 data로 하여 업데이트 된다. 이때, data group의 첫번째 data의 address만이 기록된다. 업데이트 과정이 끝나면, load 될 data group의 address를 나타내는 instruction의 source register 값으로 CRT를 확인한다. 만일 address가 대표 address인 경우, RWRT가 [register id, hash index]의 tag와 [warp id, register id]의 data로 업데이트 되며, 기존의 pipeline을 따라 나머지가 수행된다. 반대의 경우, AWRT가 [register id, warp id]의 tag와 [hash index]의 data로 업데이트 되며 instruction은 skip된다. 현재 instruction은 pipeline에서 flush되며, next PC 값이 현재 PC 값 + 4에서 다음 InstBlock의 첫번째 instruction의 address인 현재 PC 값 + 36으로 변경된다.

add.approx. 해당 명령어가 실행되면, [sourse register id, warp id]를 tag로 사용하여 AWRT를 확인한다. 만일 AWRT에서 miss가 나게 되면, 일반적인 pipeline을 따라 수행한다. Hit이 나는 경우, 얻어진 hash index로 [source register id, hash index]의 tag를 구성하여 RWRT에 접근한다. 만일

RWRT에서 hit이 된다면 원래 register 대신에 새롭게 얻어진 warp와 register에 접근한다. 만일 miss가 난다면, 해당 warp는 RWRT의 값이 채워져 hit이 될 때까지 계속 접근한다.

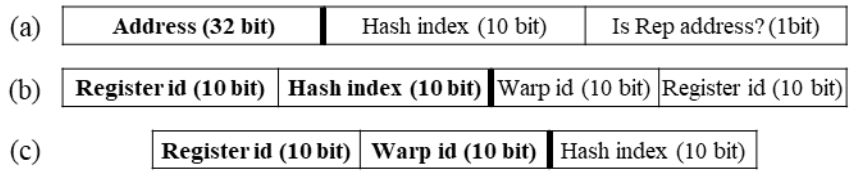

그림 3.14 추가된 table들의 구조 (a) CRT (b) RWRT (c) AWRT

3.5 실험 결과 및 분석

3.5.1 실험 환경 구성

실험을 위해, cycle accurate한 시뮬레이터인 GPGPU-Sim V3.2.2 [45]를 수정하였다. GPU 구조는 NVIDIA Fermi GTX480 아키텍처를 기반으로 SM 수를 변경하였으며, GDDR 대신 HBM에 적합하게 변수들을 변경하였다. Main memory로는 하나의 HBM stack을 가정하였으며, 제안한 ADC-PIM을 위한 memory 명령어들을 지원하도록 수정하였다. 표 3.2는 실험에 사용된 GPU 및 DRAM (HBM)의 simulation 변수들을 나타낸다. 새롭게 추가된 PIM memory 명령어의 latency 등과 같이 ADC-PIM으로 인해 추가된 overhead들은 3.5.5 장에서 분석되었다.

GPU의 전력 소모량을 분석하기 위해, GPUWattch [46]가 사용되었으며, 추가된 모듈들의 전력 소모량을 분석하기 위해서는 CACTI [47]가 사용되었다. DRAM이 소모하는 에너지는 DRAM 접근 당 [48]를 참고하여 4.39 pJ/bit로 계산되었다. VGG network와 ResNet에서 사용되는 13개의 대표적인 convolution layer들에 대해 성능을 확인하였으며, 해당 layer들의 자세한 configuration은 표 3.3과 같다. 정확도의 변화를 확인하기 위해, 제안하는 approximation 방법은

표 3.2 GPU 및 DRAM 시뮬레이션 변수

SM parameters	16SM, 1400MHz, 32768 registers, 8 threads/warp, 1024 threads, LRR scheduler
L1 cache	32KB, 128B cache line
L2 cache	768KB, 128B cache line
Memory model	8 channels, 16 bank, 256GB/s, 32B column I/O bit width, FR-FCFS
Memory timing parameters (ns)	tRCD=16, tRP=16, tRAS=28, tRC=45, tCL=16, tRRD=2, tCCDS=2, tCCDL=4, tWR=16, tFAW=12, tWL=2

PyTorch와 cupy [49]를 통해 구현되었으며, ImageNet dataset [50]에 대해 실험하였다. 표 3.4.의 5가지 configuration은 (1) memory bottleneck 감소에 얼마나 효과적인지 (2) two-level 구조의 효율성 (3) data reorganization을 통한 load imbalance 완화의 3가지 항목에 대해 제안하는 방법의 성능을 확인하기 위해 실험되었다.

표 3.3 실험에 사용된 convolution layer configuration

CONV Layer No.	Channel	Filter	Height	Width	Size of Filter	Stride
1	3	64	224	224	3	1
2	64	64	224	224	3	1
3	64	128	112	112	3	1
4	128	128	112	112	3	1
5	128	256	56	56	3	1
6	256	256	56	56	3	1
7	256	256	56	56	3	1
8	256	512	28	28	3	1
9	512	512	28	28	3	1
10	512	512	28	28	3	1
11	512	512	14	14	3	1
12	512	512	14	14	3	1
13	512	512	14	14	3	1

표 3.4 실험된 configuration

Name	Configuration
Baseline	Baseline GPU wihtout approximation
AHG-LB	Approximation on host GPU + load balancing
A1P-LB	Approximation with 1-level PIM + load balancing
A2P	Approximation with 2-level PIM
A2P-LB	Approximation with 2-level PIM + load balancing

3.5.2 제안하는 각 방법의 영향 분석

먼저, 제안하는 2-level PIM 구조는 in-memory data comparison으로 인한 memory latency의 증가를 효과적으로 감소시켰다. 그림 3.15는 Baseline, A1P-LB, A2P-LB (i.e., ADC-PIM)의 memory access latency를 나타낸다. Baseline, A1P-LB, A2P-LB의 평균 memory access latency는 각각 383, 423, 345 cycle이었다. A2P-LB는 PIM에서의 data comparison으로 인해 소요되는 시간을 2-level 구조를 통해 효과적으로 감소시켰으며, PIM을 사용하여 memory로의 data traffic을 감소시켜 memory hierarchy 상에서의 혼잡 또한 줄였기에 memory latency를 Baseline에 비해서 10% 감소시킬 수 있었다. 반면, A1P-LB는 PIM을 통해 memory hierarchy 상에서의 혼잡함은 감소시킬 수 있었지만, PIM data comparison으로 인해 memory latency가 증가하였기에 Baseline보다 긴 memory latency를 보였다. Skip되는 연산량이 감소할수록, A1P-LB의 memory latency는 Baseline 대비 커지는 경향을 보였다. 이는 skip되는 연산량이 감소하게 되면 PIM에 의한 memory traffic 효과는 감소하지만 A1P-LB에 의한 memory latency 증가 효과는 동일하기에 memory hierarchy 상에서의 혼잡을 증가시키기 때문이다. Memory latency에서의 차이는 속도와 IPC에서의 차이를 야기하였다. 그림 3.16과 그림 3.17은 실험한 configuration들의 Baseline 대비 속도와 IPC를 나타낸다. A2P-LB는 A1P-LB 대비 7.3%의 속도 증가와 4.1%의 IPC 증가를 보였다. Memory latency 차이가 커질수록 속도의 차이 또한 증가하였다. A2P-LB와 A1P-LB가 제일 큰 latency 차이를 보였던 layer 4의 경우 A2P-LB는 A1P-LB 대비 19%의 속도 증가와 11%의 IPC 증가를 보였다.

두번째로, 제안하는 PIM data reorganization은 warp 간 load imbalance를 효과적으로 감소시켜 성능을 향상시켰다. A2P-LB는 A2P 대비 11%의 속도 증가와 10%의 IPC 증가를 보였다. 더 많은 연산이 skip 된

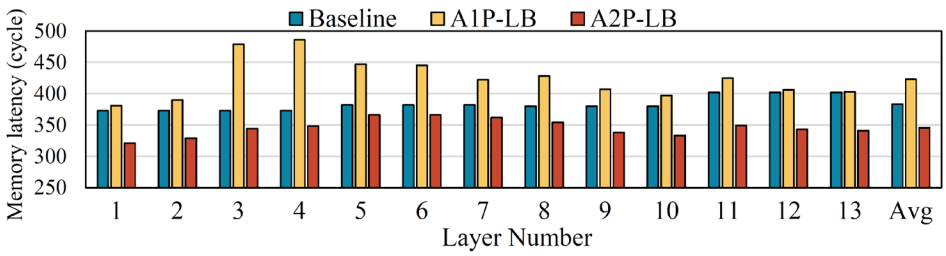

그림 3.15 Baseline, A1P-LB, A2P-LB의 memory access latency

상황일수록 load balancing에 의한 영향 또한 컸다. 그림 3.18은 convolution 연산에 사용되는 instruction과 전체 instruction들에 대해 각각 skip된 instruction의 비율을 나타낸다. Convolution 연산 중 각각 66%와 43%의 연산이 skip된 layer 1과 6을 비교해보면, layer 1의 경우 A2P-LB가 A2P 대비 19%의 속도 향상과 23%의 IPC 증가를 보여준 것이 비해, layer 6의 경우 A2P-LB는 A2P 대비 단지 4.2%의 속도 증가와 3.9%의 IPC 증가만을 달성하였다. 특정 warp에 작업이 집중되면 다른 warp에 할당되는 작업량은 전체 연산량이 감소할수록 줄어든다. 따라서, 더 많은 연산이 skip될수록 남아있는 연산량은 감소하기 때문에, warp간 작업량의 차이는 더 증가하게 되고, 이는 warp 간 load imbalance를 더 증가시킨다. 제안하는 PIM data reorganization 방법은 그러한 상황에서도 load imbalance를 효과적으로 최소화 시키는 것을 확인할 수 있다.

마지막으로, 제안하는 ADC-PIM (i.e., A2P-LB)는 Baseline과 비슷하게 IPC를 유지하면서 성능을 향상시킬 수 있었다. ADC-PIM는 Baseline에 비해 43%의 속도 향상을 보였으며, 95%의 IPC를 달성하였다. ADC-PIM의 높은 성능은 두가지 요인에 기인한다. 첫번째 요인은 GPU에서 instruction skip으로 구현된 approximate computing에 의한 연산량 감소이다. 그림 3.18에 나타나 있듯이, convolution에 사용되는 instruction 중 평균적으로 50%의 instruction이 skip 되었으며, 전체 instruction 중에는 33%가 skip 되었다. 두번째 요인은 PIM에서 일부 data만 전송함으로써 감소한 memory traffic이다. 그림 3.19는 ADC-PIM이 적용된 경우 Baseline

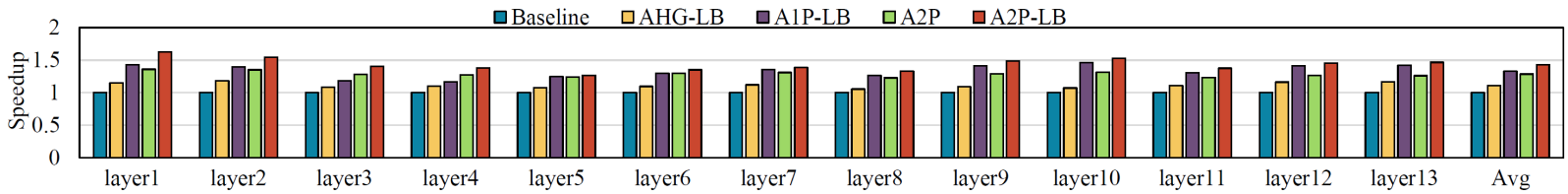

그림 3.16 AHG-LB, A1P-LB, A2P, A2P-LB의 Baseline 대비 속도 향상 정도

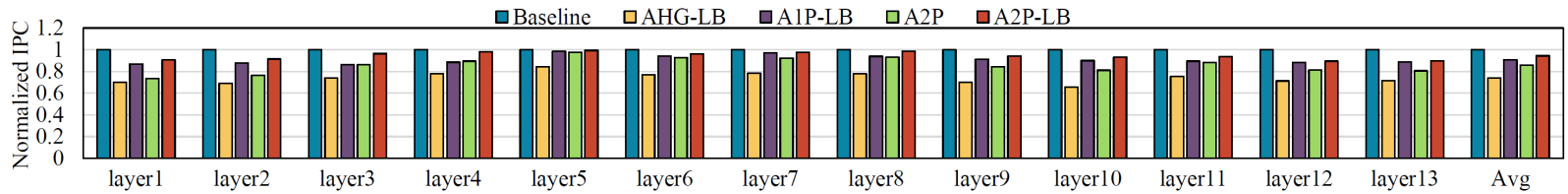

그림 3.17 AHG-LB, A1P-LB, A2P, A2P-LB의 Baseline 대비 IPC

대비 memory traffic을 나타낸다. Memory traffic은 연산이 skip된 비율과 비슷하게, Baseline 대비 평균적으로 39% 감소하였다. 따라서, approximate computing을 적용하였음에도 memory intensity가 동일하게 유지되었기에 memory bottleneck이 발생하지 않았다. 대조적으로, AHG-LB는 memory hierarchy에서의 bottleneck으로 인해 Baseline의 IPC의 74%만을 달성할 수 있었으며, Baseline 대비 속도 증가 또한 11% 밖에 되지 않았다. 제안하는 ADC-PIM이 Baseline과 IPC를 비슷하게 유지할 수 있었지만, 더 많은 연산이 skip 될수록 Baseline 대비 IPC가 감소하는 것을 확인할 수 있었다. Layer 5의 경우, 33%의 convolution 연산만이 skip 되었으며, 이 경우 ADC-PIM을 통해 Baseline 대비 99%의 IPC를 달성할 수 있었다. 반면, layer 13의 경우 61%의 convolution 연산이 skip 되었으며, IPC는 Baseline의 90%로 layer 5의 경우보다 낮았다. 이는 더 많은 연산이 skip 될수록 작업을 할당 받지 못하는 warp의 수가 증가하여 latency hiding에 사용될 수 있는 warp 수가 감소하기 때문이다. 그러나, A2P와 A2P-LB의 비교해서 확인하였듯이 그러한 경우에도 제안 방법은 warp 간 load imbalance를 감소시킬 수 있었다. 종합하면, 제안한 ADC-PIM이 approximate computing 및 in-memory data comparison으로 인해 발생하는 부작용들을 효과적으로 감춰 성능을 향상시킨 것을 확인할 수 있다.

3.5.3 기존 연구와의 성능 비교

제안하는 ADC-PIM의 성능을 확인하기 위해, NNCAM [36]과 성능이 비교 되었다. NNCAM은 CAM을 사용하여 자주 사용되는 연산과의 비교를 수행하지만, 본 논문에서 비교 시에는 CAM search delay를 0으로 가정하여 비교하였다. 이는 비교하는 즉시 연산이 skip될 수 있게 하기에, NNCAM에 유리한 조건이다. 그림 3.20과 3.21은 ADC-PIM의 NNCAM 대

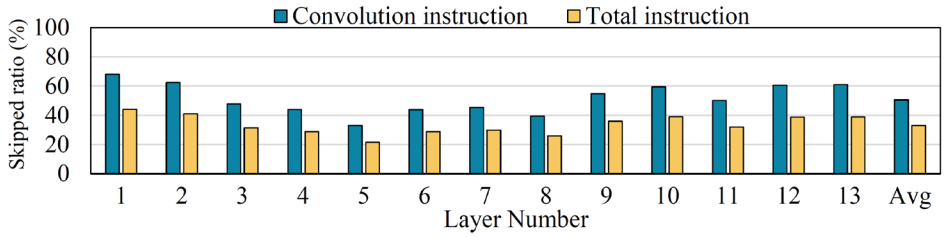

그림 3.18 Convolution에 사용되는 instruction 및 전체 instruction 중 skip된 instruction의 비율

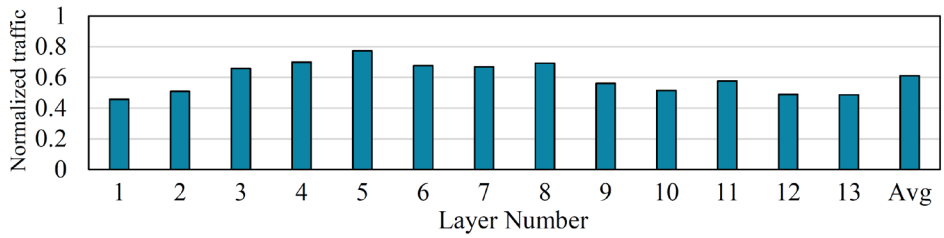

그림 3.19 ADC-PIM의 Baseline 대비 memory traffic

비 속도와 IPC를 나타낸다. 그림 3.22는 NNCAM과 ADC-PIM의 convolution instruction 중 skip된 instruction의 비율을 나타낸다. ADC-PIM은 NNCAM에 비해 더 적은 연산이 skip 되었음에도 불구하고, NNCAM 보다 높은 성능을 보였다. ADC-PIM은 NNCAM에 비해 연산을 11%만큼 덜 skip 하였지만, 22%의 속도 향상 및 35%의 IPC 증가를 보였다. NNCAM은 전체 dataset을 미리 분석하여 자주 사용되는 연산들을 파악하고, associative memory (i.e., LUT)를 해당 연산들로 채운다. 반면, ADC-PIM은 미리 주어진 정보 없이 runtime에 data 간 유사도를 판별한다. 따라서, ADC-PIM은 NNCAM 대비 더 적은 연산을 skip하였다. 그럼에도 불구하고, layer 3, 9, 10의 경우, ADC-PIM은 NNCAM 보다 비슷하거나 더 많은 연산을 skip하였다. 해당 layer들의 경우, ADC-PIM에 의한 memory bottleneck 감소 효과와 load imbalance 완화 효과가 효과적으로 반영되어, NNCAM 대비 ADC-PIM의 속도 향상이 다른 경우보다 높았다. Layer 3, 9, 10의 경우 ADC-PIM은 NNCAM 대비 각각 34%, 40%, 45%의 속도 향상을 보였다. 이외 다른 layer의 경우, 더 많은 연산이 skip 될수록 ADC-PIM이

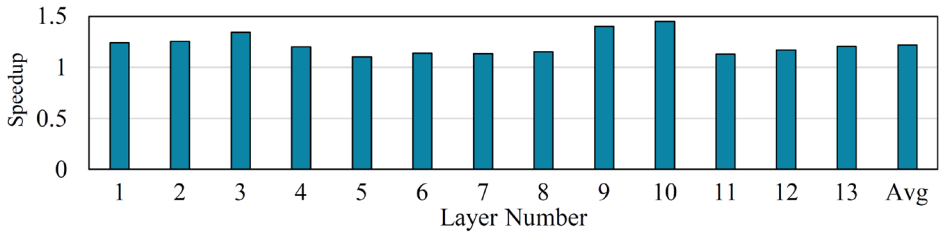

그림 3.20 NNCAM 대비 ADC-PIM의 속도 향상 정도

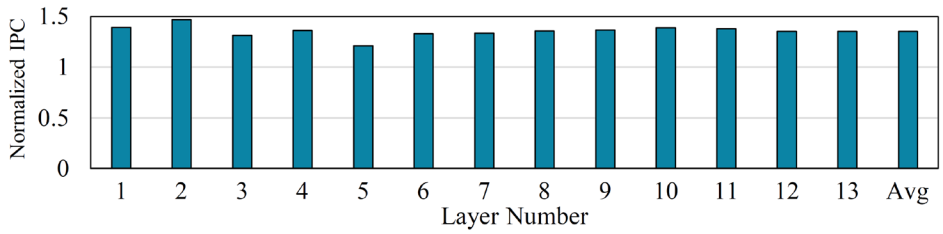

그림 3.21 NNCAM 대비 ADC-PIM의 IPC 향상 정도

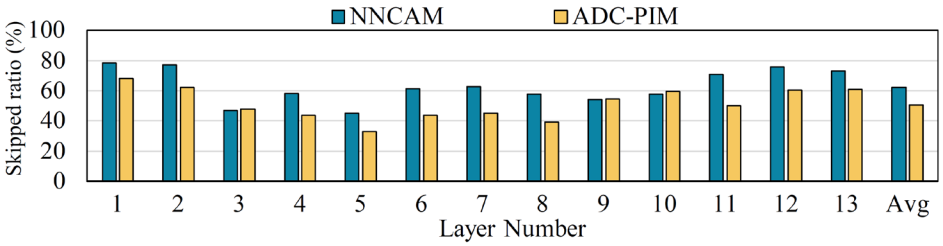

그림 3.22 Convolution에 사용되는 instruction 및 전체 instruction 중 NNCAM과 ADC-PIM에 의해 skip된 instruction의 비율

NNCAM 보다 높은 IPC를 달성하였다. 이는 3.5.2 장에 언급되었듯이, 더 많은 연산이 skip 될수록, ADC-PIM에 의한 memory bottleneck 및 load imbalance 완화 효과가 증가하기 때문이다. Layer 2와 5를 비교해보면, NNCAM의 경우 각각 77%와 45%의 convolution 연산이 skip 되었다. Layer 2의 경우, ADC-PIM은 NNCAM 대비 47%의 높은 IPC를 보였다. 반면, layer 5의 경우 ADC-PIM은 NNCAM 대비 21% 높은 IPC를 보여 layer 2의 경우보다는 낮은 IPC 향상 정도를 가졌다. 그럼에도 불구하고, ADC-PIM은 layer 5의 경우에도 NNCAM 대비 10% 높은 속도를 달성하였다. 종합하면, ADC-PIM은 NNCAM보다 높은 속도 향상을 보였으며, 연산의 pre-profiling 없이도 NNCAM과 크게 차이 나지 않는 연산 skip 정도를 달성하였다.

3.5.4 에너지 소모량 비교

그림 3.23은 Baseline에 normalize 된 에너지 소모량을 나타낸다. B와 P는 각각 Baseline과 ADC-PIM을 나타낸다. 새롭게 추가된 unit들의 에너지 소모량은 3.5.5 장에서 자세하게 분석되었다. 제안하는 방법은 Baseline에 비해 68%의 에너지만을 소비하는 것을 확인할 수 있다. 이러한 효율적인 에너지 소모에는 2가지 이유가 있다. 첫번째로는, 연산 skip에 의한 GPU의 pipeline 및 FPU 에너지의 감소이다. GPU의 core에서 소모하는 에너지는 Baseline에 비해 평균적으로 31% 감소하였다. 두번째 이유는 memory traffic의 감소에 의한 NoC 및 DRAM 에너지의 감소이다. 상술하였듯이, 제안하는 방법으로 memory traffic을 감소시킬 수 있으며, 이는 NoC 및 DRAM 에너지를 39% 감소시켰다. ADC-PIM으로 인한 에너지 소모량은 증가는 전체의 0.7% 뿐으로 무시할 만하다.

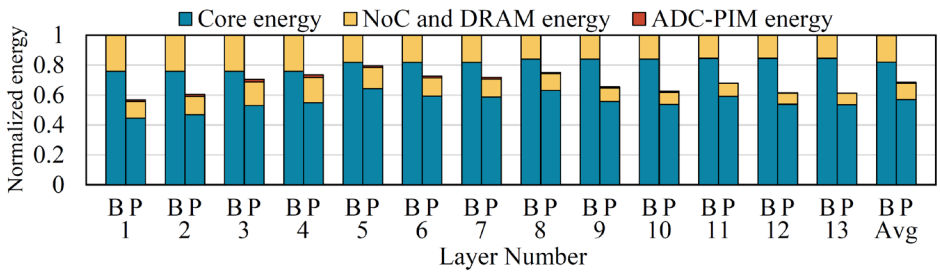

그림 3.23 Baseline (B)와 ADC-PIM (P)의 에너지 소모량

3.5.5 Design Overhead 분석

본 논문에서는 22nm 공정 및 single precision floating point의 data type을 가정하였다. Half precision이나 integer data type의 경우에는 overhead가 더 감소할 것이다. 새로 추가된 부분들의 latency, 면적, 에너지는 표 3.5의 값과 같다.

1) DRAM overhead 분석. ADC-PIM은 HCU, BDMU, 그리고 TDMU로 구성된다. HCU를 구성하는 부분 중 제일 비중이 큰 것은 16개의 벡터 내적 연산기이다. 벡터 내적 연산기의 구조로는 [51]에서 제안된 구조가 사용되었다. DMU는 cluster table, index buffer와 연산기로 구성이 되어 있다. BDMU의 연산기는 8개의 덧셈기로 구성되어 있으며, TDMU의 연산기는 8개의 덧셈기, 8개의 곱셈기와 1024 bit의 LUT로 구성되어 있다. BDMU의 cluster table과 index buffer는 8개의 entry를 가지며, 따라서 size는 각각 2200 bit (8×(16+3+256))과 128 bit (8×16)이다. TDMU의 경우 cluster table과 index buffer는 32개의 entry를 가지며, 따라서 각각 8864 bit (32×(16+5+256)), 512 bit (32×16)을 차지한다. RTG의 double buffer는 512 bit (256×2)의 크기를 차지한다.

1-1) Latency 분석. HCU와 BDMU의 latency (t_{HCU}, t_{BDMU})는 각각 2.67 ns, 0.89 ns로 둘의 합은 3.56 ns이며 이는 t_{CCDL}보다 작다. 따라서, RDBP 명령어에서 이전에 읽은 data의 비교와 새로운 data의 읽기는 병렬적으로 수행될 수 있으며, t_{HCU}와 t_{BMDU}는 마지막 data를 읽을때만 반영된다. RDBP 명령어로 8개의 data를 읽는 것은 $t_{CL} + 8 \times t_{CCDL}$ 시간을 소요하기에, ADC-PIM으로 인한 latency의 증가는 10% 이내이다.

1-2) 면적 분석. HBM DRAM의 die 면적은 84.4 mm^2이다 [19]. HCU와 BDMU는 DRAM의 각 bank마다 탑재되어 있고, TDMU는 각 channel마다 탑재되어 있기에 총 32개의 HCU, 32개의 BDMU, 그리고 2개의 TDMU가 DRAM die 당 탑재되어 있다. 추가로 탑재된 unit들의 총 면적은 die 당 3.9 mm^2으로, 전체 DRAM die의 4.6%를 차지한다.

표 3.5 추가된 구조의 latency, area 및 에너지 소모량

		Latency (ns)	Area (mm^2)	E/op (pJ)
DRAM (ADC-PIM)	HCU	2.67	0.114	111.2
	BDMU	0.89	0.010	5.5
	TDMU	1.69	0.031	15.9
GPU	CRT	0.198	0.008	0.58
	RWRT	0.244	0.014	0.55
	AWRT	0.227	0.012	0.49

1-3) 전력 및 에너지 분석. ADC-PIM의 DRAM die 당 전력 소모량은 742 mW이다. 따라서 ADC-PIM이 탑재되더라도, 기존 HBM의 thermal design power (TDP)를 여전히 만족한다. ADC-PIM은 동작에 0.52 pJ/bit의 에너지를 소모한다. HBM의 bit 당 소모하는 에너지는 4.39 pJ/bit로, ADC-PIM으로 인한 에너지 소모량의 증가는 12% 이다. 그러나, 3.5.4 장에서 언급되었듯이 ADC-PIM으로 인한 에너지 증가량 보다 NoC 및 DRAM에서의 에너지 감소량이 훨씬 크기에, 이는 큰 문제가 되지 않는다.

2) GPU overhead 분석. GPU 쪽에 새롭게 추가된 CRT, RWRT, AWRT의 저장공간 요구량은 SM 당 각각 784, 1408, 1088 byte이다. SM 당 register file 및 L1 cache의 크기가 160 KB에 달하는 것을 고려해보면, 새롭게 추가된 부분은 SM 저장공간의 2% 만을 차지한다. CRT, RWRT, AWRT의 에너지 소모량은 3.5.4 장에 반영되어 있으며, core의 에너지 소모량의 0.1% 미만을 소모한다.

3.5.6 정확도에 미치는 영향

표 3.6은 baseline과 제안된 방법의 top-1과 top-5 정확도를 나타낸다. 대표적으로 쓰이는 CNN 네트워크인 AlexNet, VGG, ResNet-50에 대해 정확도가 측정되었으며, approximation으로 인한 정확도 저하가 top-1과 top-5에 대해 각각 1.0%, 0.6% 미만인 것을 확인할 수 있다.

표 3.6 Approximate computing으로 인한 CNN 정확도 변화

	Baseline		Approximated	
	Top-1 (%)	Top-5 (%)	Top-1 (%)	Top-5 (%)
AlexNet	55.4	77.6	54.4	77.0
VGG	72.8	90.4	72.0	90.0
ResNet-50	76.2	93.0	75.6	92.8

3.6 본 장의 결론

본 장에서는 approximate computing과 결합된 processing-in-memory solution인 ADC-PIM을 제안하였다. Approximate computing에서의 연산 skip으로 인해 memory intensity가 증가하게 되고, 이로 인한 memory bottleneck을 완화하기 위해 data는 GPU에서 비교되는 대신, memory에서 비교되게 된다. Data의 비교 범위는 approximation의 정도와 overhead를 고려하여 선택되었다. PIM으로 인한 memory latency의 증가를 최소화하기 위해, DRAM bank와 TSV 단을 모두 활용하는 2-level PIM 구조가 제안되었다. 이에 더해, ADC-PIM은 GPU로 비교 결과를 전송하기 전에 data reorganization을 통해 data를 적절한 address에 할당함으로써 GPU에서의 warp 간 load imbalance를 완화한다. 실험 결과는 ADC-PIM이 approximate computing으로 인해 발생하는 memory bottleneck을 효과적으로 감소시키는 것을 보여준다.

Future work로, ADC-PIM을 GPU가 아닌 다른 DNN 가속기에 적용할 수 있다. 많은 DNN 가속기들이 높은 memory access overhead를 지니기에, 그와 같은 환경에서도 ADC-PIM이 효율적일 것으로 예상된다.

제 4 장 Convolutional Layer와 non-Convolutional Layer의 Pipeline 실행을 통한 CNN 학습 가속

해당 장에서는 PIM 및 GPU를 활용하여 CNN 학습을 가속하는 방법을 제안한다. CNN 학습은 convolutional (CONV) 및 batch normalization (BN), ReLU 등의 non-convolutional (non-CONV) layer로 구분이 가능하다. 이 중, CONV는 1×1 CONV, Winograd 변환 등 다양한 방법으로 가속하는 연구가 활발히 진행되고 있다 [52], [53]. 이에 따라, CNN 학습에서 non-CONV layer가 차지하는 비중이 점점 증가하고 있다. Non-CONV layer는 memory intensive하며, data reuse가 적기에 가속을 위해서는 memory bottleneck issue 해결이 필수적이다. PIM은 이를 해결하기 적합하기에, PIM에서 non-CONV layer를 수행하고, host GPU에서는 convolutional (CONV) layer를 수행하여 CNN 학습을 가속할 수 있다 [15]. 이때, non-CONV layer는 output feature map의 channel 단위로 수행되기에, 본 연구에서는 CONV가 완료된 channel에 대한 non-CONV를 CONV와 pipeline 적으로 수행하며, 역전파 과정에서 발생하는 weight update와 feature map gradient 계산에서의 CONV와 non-CONV 간 job 균형을 위해 적절하게 non-convolution job을 분배하는 동작 flow를 제안한다. 이에 더해, CONV와 non-CONV가 host와 PIM에서 동시에 수행될 때, host와 PIM의 memory 접근 시의 conflict를 고려한 memory scheduling 알고리즘을 제안한다.

4.1 관련 연구

4.1.1 Non-CONV Layer의 Memory Bottleneck 완화

Non-CONV layer의 memory bottleneck를 완화하기 위해, CNN 학습 시 수행 순서 등을 변경한 연구들이 있다 [55], [56], [57]. [55]는 non-CONV의

연산 과정을 나눠 (fission) CONV와 합쳐 (fussion) CONV을 완료한 data를 non-CONV가 바로 활용할 수 있게 하였으며, 이를 통해 memory로의 data 이동량을 감소시켰다. INPLACE-ABN [56]은 학습의 역전파 과정 (backward path) 시 필요한 중간 결과들을 저장하는 대신, 순전파 과정 (forward path)의 결과를 활용하여 역전파 과정에 필요한 값을 계산하여 memory access를 줄이고 memory 사용량을 줄인다. MBS [57]는 각 mini batch를 sub batch들로 나누고 전체 layer를 일부 layer 씩 grouping하여, on-chip buffer에 sub batch의 inter layer data가 모두 저장되도록 하여 layer 당 memory 사용량을 감소시킨다. 역전파에 필요한 data만 off-chip memory에 저장되고, 나머지는 group 간의 data 이동에서만 off-chip memory를 거치게 된다. 위의 연구들과 같이 학습 동작 flow를 변경하는 대신, Co-ML [15]는 PIM을 활용하여 non-CONV의 memory bottleneck을 완화하였다. Co-ML은 CONV는 host GPU에서 수행하고, non-CONV를 PIM에서 수행하여 CNN 학습을 가속하였다. 그러나, 해당 연구의 경우 host와 PIM의 memory conflict를 피하기 위해 단순히 CONV와 non-CONV가 serial하게 수행된다고 가정하였기에 성능을 추가적으로 향상시킬 여지가 있다. 제안하는 방법은 CONV와 non-CONV를 동시에 수행하여 추가적으로 성능을 향상시킨다.

4.1.2 Host와 PIM 간 Memory Scheduling

대부분의 PIM 연구는 PIM이 host와 동시에 동작하는 경우를 고려하지 않았다. 해당 연구들은 PIM이 host와 time-division되어 PIM에 coarse-grain하게 job을 할당하여 동작한다고 가정하거나 [58], [59], PIM request를 host request와 동일하게 취급하였다 [60], [61]. Host와 PIM이 동시에 동작하는 경우, memory를 공유하게 되기에 memory 접근 시의

conflict를 고려해야 한다. 최근 들어, CHOPIM [62]은 host와 PIM이 동시에 동작하는 경우의 memory scheduling 알고리즘을 제안하였다. Host와 PIM이 rank 단위로 동작하는 상황에서, host request를 PIM 동작에 우선하여 host request가 들어오면 PIM 동작을 지연시켰다. 또한 bank conflict를 감소시키기 위해 bank partitioning을 통해 host 전용 bank와 PIM과 공유하는 bank로 분류하였다. 이에 더해, PIM write 동작 후 read 시 read/write-turnaround delay로 인해 host 성능 저하가 크다는 점을 고려하여 PIM write을 확률적으로 수행하거나 host가 접근할 rank를 예측하여 다른 rank에서 PIM 동작을 수행하는 방법을 제안하였다. MViD [63] 역시 CHOPIM과 비슷하게 bank partitioning을 적용하여 host 전용 bank와 PIM 용 bank로 구분하였다. 이에 더해, PIM bank에 host request가 일정 개수 이상 대기 중이면 PIM 동작을 멈추고 host request를 수행하는 방법을 제안하였다. 그러나, MViD는 대기하는 host request 수의 최적 threshold를 찾기 위한 방법을 제시하지 않았으며, 대기 중인 host request로 인한 host request의 latency 증가를 고려하지 않았다. 또한 해당 논문들에서 제안한 bank partitioning은 PIM이 host 작업의 결과물을 사용하기 위해서는 host 작업이 끝난 후 결과를 PIM bank로 copy해야 한다. Host와 PIM 간 data 공유가 적다면 문제되지 않을 수도 있지만, CONV와 non-CONV는 feature map 및 gradient 값을 공유하기에 이로 인한 다음의 2가지 overhead가 문제가 된다. (1) data copy로 인한 시간 증가 (2) copy되는 만큼 사용 가능한 memory 용량 감소. 특히, GPU의 작은 main memory와 CNN 학습의 높은 memory 사용량을 고려하면 두번째 단점의 영향은 더욱 커진다. 따라서, 본 논문에서는 bank partitioning 대신 host와 PIM이 모든 bank를 공유하는 환경에서 최적의 memory scheduling 알고리즘을 제시하였다.

4.2 Motivation

4.2.1 CONV와 non-CONV의 동시 수행 시 성능 향상 기회

Non-CONV는 output feature map (OFMAP)의 각 channel단위로 독립적으로 수행된다. 이를 활용하여, CONV 연산이 완료되어 memory에 저장된 OFMAP channel에 대해서는 PIM에서 non-CONV를 수행할 수 있다. 즉, host에서는 나머지 OFMAP channel에 대해 CONV 연산을 수행하고, 동시에 PIM에서는 CONV 연산이 완료된 OFMAP channel에 대해 연산을 수행할 수 있다. 그림 4.1은 ResNet50에 대해 CONV와 non-CONV의 수행 위치 및 방법에 따른 수행시간 변화를 나타낸다. 그림 4.1 (b)와 같이 non-CONV를 PIM에서 수행함으로써 host GPU에서 모든 것을 수행하는 경우 대비 17%만큼 수행 시간을 감소시킬 수 있다. 이에 더해 그림 4.1 (c)와 같이 Host와 PIM이 동시에 작업을 수행하더라도 conflict가 발생하지 않아 서로의 수행 시간이 영향 받지 않는 이상적인 경우, pipeline 실행을 통해 Co-ML [15]에서 제안한 CONV와 non-CONV의 serial 수행 대비 28%의 수행 시간 감소가 가능한 것을 확인할 수 있다.

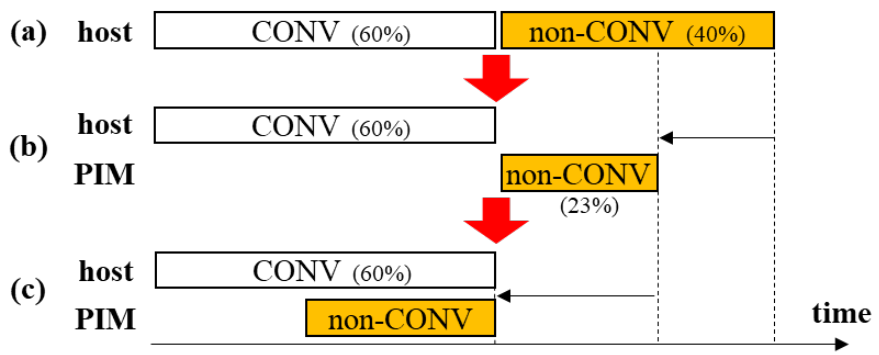

그림 4.1 CONV와 non-CONV의 수행 위치 및 방법에 따른 수행시간 변화 (a) CONV, non-CONV 모두 host에서 수행하는 경우 (b) CONV는 host에서, non-CONV는 PIM에서 serial하게 수행하는 경우 (c) CONV는 host에서, non-CONV는 PIM에서 parallel하게 수행하는 경우

4.2.2 PIM 우선도에 따른 host 및 PIM request의 처리 효율성 변화

덜 memory intensive한 application이 memory intensive한 application과 함께 수행되는 경우, memory intensive application에 의한 성능 저하를 방지하기 위해 기존의 연구들은 덜 memory intensive한 application에 memory 접근 우선권을 준다 [64], [65], [66]. 이와 비슷하게, rank level PIM 환경에서의 host-PIM memory scheduling 연구인 CHOPIM [62] 또한 host의 memory 접근을 PIM보다 우선하여 처리하였다. 이는 rank level PIM의 경우 PIM이 rank에 접근하는 동안 host가 해당 rank에 접근하지 못하기 때문이다. 그러나, 본 논문에서 상정하는 bank level PIM 환경의 경우, 특정 bank에서 PIM이 연산을 수행하는 동안에도 host가 다른 bank에 접근할 수 있다. 또한, PIM이 특정 bank에서 연산을 수행하는 동안, 해당 bank에서 처리될 host request들은 대기하다가 PIM이 해당 bank에서의 연산을 끝내면 한꺼번에 처리된다. 이는 host request의 밀집도를 증가시켜 bank level parallelism (BLP) 등을 증가시킨다. 즉 PIM 동작으로 host request 처리 효율성이 증가한다. 그림 4.2는 PIM mode로 진입 시 PIM의 bank 점유 시간에 따른 host request의 BLP 값을 나타낸다. Bank 점유 시간이 길어질수록 BLP 값이 증가하는 것을 확인할 수 있다. 또한, PIM 점유 시간이 증가할수록 PIM의 row buffer locality 또한 증가한다. 이는 PIM에 일정 기간 memory 접근 우선권을 주는 것이 전체 수행 시간을 감소시킬 수 있다는 것을 의미한다.

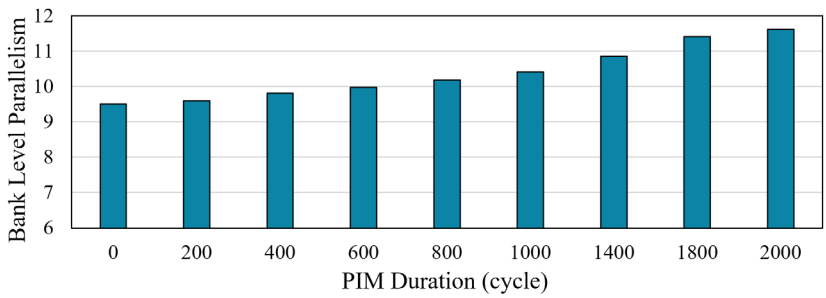

그림 4.2 PIM의 bank 점유 시간에 따른 host request의 BLP

4.3 제안하는 host-PIM Memory Scheduling 알고리즘

4.3.1 host-PIM System Overview

가정하는 host-PIM system의 overview는 그림 4.3과 같다. Host는 NVIDIA의 V100 [67], [68]과 비슷한 구조를 가지는 GPU이며, main memory로는 3D stacked DRAM (HBM)을 사용하였다. PIM은 SIMD 연산기와 temporary storage를 가진다. 이와 비슷한 구조를 가지는 Samsung PIM [19]에서의 PIM의 area overhead를 고려하여, 각 bank group마다 PIM이 위치한다. 즉 4개의 bank가 1개의 PIM unit을 공유하여 사용한다. Host GPU는 CONV layer를 수행하며, PIM은 non-CONV layer를 수행한다. Host와 PIM은 DRAM의 각 bank group 단위로 소유권 (ownership)을 가진다. 특정 bank group에 대한 소유권을 host가 가지는 경우, 해당 bank group은 host request를 처리하며 반대의 경우에는 PIM request를 처리한다. Bank group 단위의 mode 전환을 위해, bank group 단위로 row precharge 및 activation을 하는 명령어가 추가되었다. Host-PIM 간 mode 전환 방법 및 알고리즘은 4.3.2 및 4.3.3 장에서 설명된다.

4.3.2 PIM Duration Based Memory Scheduling

제안하는 알고리즘은 host의 memory request를 집약적으로 처리하여 host request 처리 효율성을 증가시켜 host request를 처리하는 데 소요되는 시간을 감소시키고, 감소된 host request 처리 시간을 활용하여 PIM request를 처리하여 전체 수행 시간을 최소로 하는 것을 목표로 한다. 먼저, CNN 학습이 시작되는 초기 상황에서는 host GPU가 모든 bank group에 대한 소유권을 가진다. 이후, 일부 OFMAP channel에 대한 CONV 연산이 완료되어 해당 channel들에 대한 non-CONV 연산을 위해 PIM

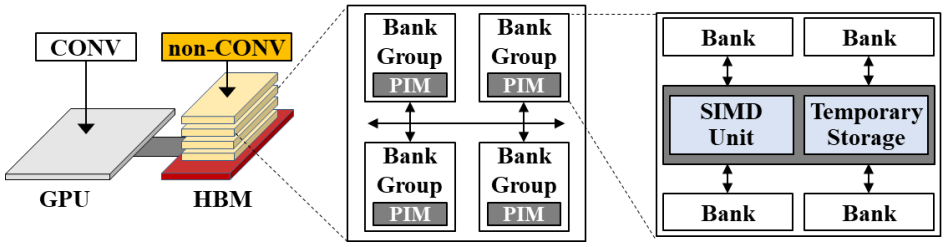

그림 4.3 Host-PIM system overview

request가 생성되기 시작한 후부터 host-PIM 간 mode 변경이 시작된다. 각각의 경우에 전환 방법 및 조건은 다음과 같다.

host mode에서 PIM mode로의 전환 : Host의 DRAM에의 접근 빈도는 시간에 따라 다른 양상을 보인다. 그림 4.4는 시간에 따른 DRAM의 bank group에의 host request 요청 횟수를 나타낸 것이다. 특정 구간에는 host request가 몰려서 들어오기에 처리가 지연되고 있는 host request의 수(pending host request)가 증가하지만, 그 외 구간에서는 request 처리 속도가 요청 속도보다 더 빠르기에 pending host request의 수가 0까지 감소하는 것을 확인할 수 있다. 대기 중인 host request의 수가 충분히 많은 경우, 이미 host memory request가 집약적으로 처리되고 있기에 더 많은 host request를 대기시키더라도 host request의 처리 효율을 증가시키기 어렵다. 반면, 대기 중인 host request의 수가 적은 경우 host request 처리를 중단하고, host request가 충분히 모일 때까지 기다렸다가 처리함으로써 host request 처리 효율을 증가시킬 수 있다. 즉, 대기 중인 host request 수가 적은 경우의 host request 처리 효율을 증가시키기 위해 각 bank group은 해당 bank group에 접근하는 host request가 없는 host idle

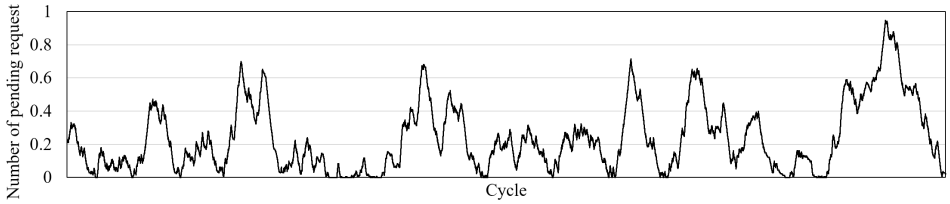

그림 4.4 Cycle에 따른 host의 DRAM 접근 빈도 (bank group 단위)

구간에 진입한 경우에 PIM mode로 진입한다. Host mode에서 PIM mode로 전환을 위해서는 먼저 bank group 단위로 row를 precharge 하고, (BG_PRE) PIM이 처리할 row를 activate (BG_ACT) 한 후 PIM mode에서 PIM 연산을 시작한다.

PIM mode에서 host mode로의 전환 조건 : PIM mode로 진입한 후, 일정 개수의 PIM request를 처리할 때마다 또는 새롭게 row를 open하여 연산을 수행해야 할 때 host mode로의 전환 조건을 만족하는지 확인한다. 이때, 너무 자주 전환 조건을 만족하는지를 확인하게 되면 PIM 동작 시의 row buffer locality (RBL)를 보장할 수 없게 되어 성능 저하를 야기한다. 본 논문에서는 각 bank가 4개의 request를 처리할 때마다 전환 조건 만족 여부를 확인하였다. PIM mode에서 host mode로의 전환 시 고려해야 할 점은 크게 두가지가 있다. 첫번째는 PIM mode 유지 시간이다. PIM mode를 오래 유지할수록 해당 bank group에 접근하는 host request들의 대기 시간 또한 길어지게 되며, 이는 host GPU의 memory access latency를 증가시킨다. GPU는 latency hiding을 통해 긴 latency를 감출 수 있지만, 일정 값 이상의 긴 latency는 감춰지지 않으며 host가 연산 수행에 필요한 data를 받지 못해 동작이 지연된다. 두번째는 대기 중인 host request의 수이다. 대기 중인 host request의 수가 많을수록 host GPU에서 pending되고 있는 memory instruction이 많다는 것을 의미하며, 이 역시 첫번째 원인과 동일하게 GPU의 동작을 지연시키게 된다. 또한, 각각의 host request들이 처리되기 위해서는 앞선 host request들을 처리해야 하므로 각 host request의 대기 시간 또한 증가하게 된다. 위의 두가지를 반영하여, PIM mode를 일정 기간 이상 유지하였거나, 대기중인 host request의 수가 충분히 많아지게 되면 PIM mode를 종료한다. 고려한 전환 조건은 식 (4.1)과 같다.

$$T_P + t_H \times N_H > PD_{TH} \tag{4.1}$$

이때 T_P 는 PIM mode를 유지한 시간, t_H 는 1개의 host request를

처리하는데 평균적으로 소요되는 시간, N_H는 PIM mode 수행 동안 대기 중인 host request의 수, PD_{TH}는 PIM mode 최대 지속시간 (PIM mode duration threshold)이다. T_P는 host request가 1개 이상 대기 중 일때부터 count된다. 수행 시간 최소화를 위해서는 최적의 PIM mode duration threshold를 설정하는 것이 중요하며, 이는 host CONV의 memory 접근 특성에 따라 다르게 설정된다. 정확한 PD_{TH}의 계산 방법은 아래에서 자세히 설명된다. 상술하였듯이, 마지막 host request가 체감하는 latency는 앞의 host request들을 처리하는 데 소요되는 시간도 포함한다. 이를 반영하기 위해, 대기 중인 host request의 수인 N_H에 비례 상수 t_H가 곱해졌다. $t_{RP}, t_{RCD}, t_{CCDs}, RBL$을 각각 DRAM row precharge time, row address to column address delay, column to column delay, application의 row buffer locality라고 할 때, 새롭게 DRAM row를 open하여 request를 처리한다고 가정하면 request 당 처리 시간 t_H는 아래 식 (4.2)와 같이 계산된다.

$$t_H = (t_{RP} + t_{RCD} + RBL \times t_{CCDs})/RBL \quad (4.2)$$

여러 bank가 동시에 동작하는 상황을 고려하여, t_{CCDL} 대신 t_{CCDs}가 사용되었으며 실험을 통해 t_H 값은 4 cycle로 설정되었다. PIM mode에서 host mode로 전환 시, BG_PRE로 bank group의 모든 bank들을 precharge한 후 host mode로 전환된다.

4.3.3 최적 PD_{TH} 값의 계산 방법

식 (4.1)에 기반하여 host-PIM 간 mode를 변경할 때 전체 수행 시간을 최소로 하기 위해서는 최적 PD_{TH} 값을 설정하는 것이 중요하다. 최적 PD_{TH} 값은 host의 memory 접근 특성 및 CONV와 non-CONV의 수행 시간 비율과 관련 있다. Host가 memory에 더 접근하는 빈도가 높을수록, CONV에 비해 non-CONV 수행 시간이 클수록 non-CONV를 처리하기 위해 PIM mode를 더 오래 유지해야 하기에 PD_{TH} 값 또한

커져야 한다. 각 CONV layer 별로 두 요인이 달라지기 때문에, PD_{TH} 값은 각 layer 별로 설정된다. 이때 host와 PIM 간의 load balancing 측면에서, 최적의 PD_{TH} 값은 host와 PIM이 동시에 작업을 끝낼 수 있게 하는 값이다. 만약 PD_{TH}가 최적 값보다 짧아 host에 비해 PIM이 늦게 끝나는 경우, 홀로 동작하게 되는 PIM 부분을 host CONV 내에 더 수행되도록 하여 CONV의 memory 접근 밀집도를 증가시켜 CONV의 memory request 처리 효율을 증가시킬 수 있다. 따라서, CONV의 수행 시간을 감소시켜 전체 수행시간을 줄일 수 있다. 반대로, PD_{TH} 를 과도하게 증가시켜 host에 비해 PIM이 먼저 끝나게 되면, (PD_{TH} = 0인 경우는 제외) host가 과도하게 지연되며 홀로 동작하게 되는 host 부분은 PIM과 함께 동작하는 경우에 비해 낮은 memory request 처리 효율을 가지게 되어 host와 PIM이 동시에 끝나는 경우에 비해 전체 수행 시간이 더 길어지게 된다. 그림 4.5는 PD_{TH} 값에 따른 host와 PIM의 수행 시간을 나타낸다. PD_{TH} 값이 0인 경우로 normalize 되었으며, host와 PIM이 동시에 동작하는 경우 (host concurrent, PIM concurrent), PIM이 홀로 동작하는 경우의 시간 (PIM alone)을 따로 측정하였다. 초기에는 PD_{TH}가 증가할수록 PIM alone 시간이 적어지며 host의 memory request 처리 효율이 증가하여 host concurrent 수행 시간은 감소하는 것을 확인할 수 있다. PD_{TH}가 최적 값에 도달하면 host와 PIM이 동시에 끝나 PIM alone 시간은 0이 되며, 전체 수행 시간은 최소가 된다. PD_{TH} 가 최적 값 이상으로 증가하게 되면 상술한 두가지 이유로 인해 host 시간이 오히려 증가하여 전체 수행 시간이 증가하는 것을 확인할 수 있다.

최적 PD_{TH} 값은 CNN 학습이 여러 iteration을 수행한다는 점을 고려하여, 0부터 시작하여 매 iteration마다 PD_{TH} 값을 증가시켜 가며 전체 수행 시간이 최소가 되도록 하는 값으로 선정할 수도 있다. 그러나, 해당 방법은 최적 PD_{TH} 값이 큰 경우 많은 iteration 동안 최적 값이 아닌 상태로 동작하게 된다는 단점이 있다. 이를 해결하기 위해, 본

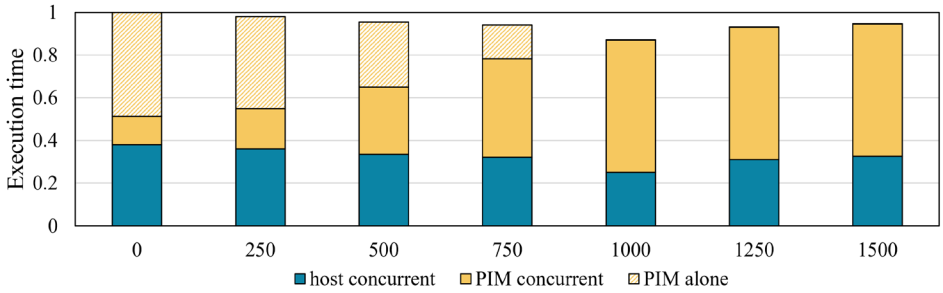

그림 4.5 PD_{TH} 값에 따른 host와 PIM의 수행 시간. PD_{TH} 값이 0인 경우의 결과로 normalize 되었다.

논문에서는 host CONV kernel의 초기 일부분 동안 얻어진 특성을 활용하여 PD_{TH} 값을 계산하는 방법을 제안한다. 먼저, CONV의 PIM Capacity ($PCAP$)라는 것을 정의한다. $PCAP$ 은 PD_{TH} 가 0일 때 host CONV의 수행 시간 동안 처리 가능한 non-CONV의 양을 전체 처리해야 하는 non-CONV의 양으로 나눈 변수이다. 예를 들어, $PCAP$이 0.1이라면 CONV 내에 처리 가능한 non-CONV의 양이 전체 non-CONV의 10%라는 것을 의미한다. $PCAP$ 이 1보다 크다면, PD_{TH} 가 0이어도 CONV 동작 내에 non-CONV를 모두 처리할 수 있다는 것을 의미한다. PD_{TH}가 0이면 PIM mode 중 host request가 1개라도 대기 중이라면 host mode로 전환하게 된다. 즉, $PCAP$ 은 host CONV에 접근 우선 권한을 줄 때 CONV 수행 내에 처리 가능한 non-CONV의 비율이다. Host CONV에 memory 접근 우선권이 있기에 PIM non-CONV는 host가 memory에 접근하지 않는 host idle 구간에만 처리될 수 있으며, 따라서 $PCAP$ 은 식 (4.3)과 같이 계산된다.

$$PCAP = T_C \times BG_{idle}/T_{NC} \tag{4.3}$$

이때, T_C 와 BG_{idle} 은 CONV를 host GPU에서 단독으로 수행할 때 소요되는 총 시간 및 bank group 단위로 host request가 접근하지 않는 idle한 시간의 비율이며, T_{NC} 는 non-CONV를 PIM에서 단독으로 수행할 때 소요되는 총 시간이다. T_C/T_{NC}는 식 (4.4)와 같이 계산될 수 있다.

$$\frac{T_C}{T_{NC}} = \frac{Work_C / Perf_{host}}{Work_{NC} / Perf_{PIM}} = \frac{Work_C}{Work_{NC}} \times \frac{IPC_{host} \times Clk_{host}}{IPC_{PIM} \times Clk_{PIM}} \quad (4.4)$$

이때 $Work_C$ 와 $Work_{NC}$ 는 각각 CONV와 non-CONV의 총 job의 양, $Perf_{host}$ 와 $Perf_{PIM}$ 은 host와 PIM의 하드웨어 성능, IPC_{host} 와 IPC_{PIM} 은 host와 PIM에서 각각 CONV와 non-CONV를 수행했을 때의 IPC, Clk_{host} 와 Clk_{PIM} 은 host와 PIM의 하드웨어 clock을 의미한다. 이중 $Work_C / Work_{NC}$ 는 CONV와 non-CONV의 알고리즘적 복잡도를 기반으로 추정 가능하며, IPC_{host} 는 host에서 CONV를 실행하여 측정 가능하다. 또한, PIM은 data 간 dependency 등이 거의 없으며, DRAM bank 단에 위치하여 memory access latency가 매우 짧기에 IPC_{PIM} 은 일정하다고 가정한다. 식 (4.3), (4.4)를 종합하면 $PCAP$은 식 (4.5)와 같이 계산된다.

$$PCAP = K \times \frac{Work_C}{Work_{NC}} \times IPC_{host} \times BG_{idle} \quad (4.5)$$

K 값은 상수로, $Clk_{host} / (IPC_{PIM} \times Clk_{PIM})$로 계산된다. CONV는 프로그램 전반에 걸쳐 비슷한 종류의 연산을 수행하기에, 프로그램 초기의 특성과 전체의 특성이 비슷하다. 따라서, IPC_{host} 및 BG_{idle} 등의 특성을 얻기 위해 학습의 첫 iteration에 CONV를 host alone으로 수행하는 대신, 첫 iteration부터 PIM과 동시에 수행하되 PIM이 수행되기 전에 host alone CONV의 초기 실행 결과에서 얻어진 특성을 사용하여 PD_{TH} 계산에 필요한 iteration 횟수를 줄인다.

최적 PD_{TH} 추정값은 이렇게 얻어진 $PCAP$ 및 CONV의 memory 특성을 활용하여 계산한다. 먼저 정의에 따라, $PCAP$은 PD_{TH}가 0일 때 host CONV 내에 처리할 수 있는 non-CONV의 비율이기에, 남은 non-CONV의 비율인 $(1 - PCAP)$ 만큼을 PD_{TH} 를 증가시켜 처리해야 한다. 따라서, $(1 - PCAP)$이 클수록 최적 PD_{TH} 값도 커진다. 두번째로, host의 밀집도가 높을수록 최적 PD_{TH}가 커진다. CONV 내 non-CONV PIM의

수행에 따른 host request 처리 효율 증가는 host request를 몰아서 처리함으로써 행해지기에, CONV의 memory request의 원래 밀집도가 높을수록 효율의 증가 정도가 감소한다. Cycle 당 memory access 양에 비해 대기 중인 request의 수가 많을수록 request가 몰려서 들어온 다는 것을 의미하기에, host request의 밀집도는 대기 중인 request의 수를 cycle 당 memory access로 나눠 측정할 수 있다. 마지막으로, PD_{TH}가 클수록 PD_{TH}를 추가적으로 증가시킬 때의 host request 효율 증가의 효과는 작아진다. 따라서, 동일한 양의 non-CONV job을 CONV 내에 추가로 수행하기 위해 증가시켜야 하는 PD_{TH}의 크기가 증가한다. 이를 고려하여 최적 PD_{TH}는 식 (4.6)과 같이 추정할 수 있다.

$$\sqrt{PD_{TH}} = \alpha \times \{(1 - PCAP) \times \frac{N_{pending}}{MAPC}\} + \beta \quad (4.6)$$

이때 $N_{pending}$은 bank group 당 대기 중인 request의 평균 개수, $MAPC$는 memory access per cycle이며, α, β는 실험적으로 구해진 상수이다. 또한 PD_{TH}가 증가할수록 감소하는 host quest 효율성 증가 정도를 $\sqrt{PD_{TH}}$가 잘 반영하는 것으로 측정되어 $\sqrt{PD_{TH}}$를 사용하였다.

최적 PD_{TH} 추정값을 활용한 최적 PD_{TH} 값 선택 알고리즘은 그림 4.6과 같다. 먼저, 첫번째 PIM 연산을 시작하기 전까지 host alone으로 수행될 때의 CONV의 stat들을 활용하여 PD_{TH}를 계산한다. 계산된 PD_{TH}로 첫번째 iteration을 수행한 후, 결과에 따라 PD_{TH}를 변경한다. 먼저 CONV 내에 non-CONV를 모두 수행한 경우, PD_{TH}를 10% 감소시켜 다음 iteration을 수행한다. 만일 PD_{TH}를 감소시킨 경우 CONV 내에 non-CONV를 모두 처리하지 못하였다면 감소시키기 전의 PD_{TH}를 최적 PD_{TH}로 설정한다. 모두 수행하지 못하고 일부 non-CONV가 남은 경우, PD_{TH}에 (전체 non-CONV 작업량 / 처리한 non-CONV 작업량)을 곱해 새로운 PD_{TH}로 설정하여 다음 iteration을 수행한다.

Algorithm 1: Optimal PD_{TH} Selection Algorithm

Input : $init_PD_{TH}$ // estimated PD_{TH}
$PD_{TH} = init_PD_{TH}$
$prevProcessedAll = False$
while $True$ **do**
    Perform one training iteration
    if `ProcessedAllNonConv` **then**
        $new_PD_{TH} = PD_{TH} \times 0.9$
        $prevProcessedAll = True$
    else if $prevProcessedAll == True$ **then**
        $new_PD_{TH} = PD_{TH}/0.9$
        break
    else
        $new_PD_{TH} = PD_{TH} \times \frac{Total}{Processed}$
    end if
end while
return new_PD_{TH}

그림 4.6 최적 PD_{TH} 값 선택 알고리즘

4.4 제안하는 CNN 학습 동작 Flow

4.4.1 CNN 학습 순전파 과정

일반적인 CNN 학습 순전파 과정 (forward path)은 input feature map에 대해 CONV, BN, Activation (ReLU 등)을 순차적으로 수행하는 과정을 반복한다. 이를 고려하여 CNN 학습 순전파 과정에서는 CONV와 그에 뒤따르는 BN 및 ReLU 등의 non-CONV가 pipelining 되어 수행된다. Skip connection으로 인한 element-wise 연산이 있는 경우 non-CONV에 포함되어 수행된다. 이때, PIM에서 non-CONV를 각 OFMAP channel 단위로 수행하게 되면 job의 양의 부족으로 인해 PIM 동작 시의 병렬성을 보장할 수 없어 PIM 동작 시 성능이 하락할 수 있다. 예를 들어, OFMAP image의 height × weight가 14 × 14이고 batch size가 64인 경우, 각 OFMAP channel에서 batch normalization을 수행하기 위해 처리해야 하는 data는 12.3 KB이다. HBM의 각 channel 당 16개에서 32개의 DRAM bank가 존재한다는 것을 고려할 때, 해당 OFMAP channel의 모든 data가 동일 HBM channel에 위치하더라도 각 bank 당 1 KB 이하의 data만 위치하게 된다. V100의 경우 총 24개의 HBM channel이 존재한다는 것을 고려하면 각 bank 당 data는 더 적어지게 되며, 이런 환경에서는 PIM의 bank level parallelism을 활용하기 어렵다. 따라서, 병렬성 확보를 위해 전체 OFMAP channel을 N개의 group으로 나누어, 각 group 단위로 non-CONV를 수행한다. OFMAP channel 수가 F일때, 각 group은 F/N개의 OFMAP channel로 구성되어 있다. CONV와 non-CONV는 각 group 단위로 pipelining되어 그림 4.7과 같이 수행된다. 먼저, host에서 CONV를 수행한다. CONV의 OFMAP의 channel group 1에 대한 연산이 완료되면, 해당 연산 결과는 DRAM에 저장된다. 다음으로, host는 다음 OFMAP channel인 OFMAP channel group 2에 대한 연산을 수행한다. 동시에,

PIM에서는 DRAM에 저장된 OFMAP channel group 1의 결과에 대해서 non-CONV를 수행한다. 이와 같이 pipelining을 통해 OFMAP channel들에 대한 CONV 및 non-CONV 연산을 반복하여, OFMAP channel group N에 대한 CONV 연산 및 OFMAP channel group N-1에 대한 non-CONV 연산을 수행하고, 마지막으로 OFMAP channel group N에 대한 non-CONV 연산을 수행한다. 해당 예시에서는 CONV가 non-CONV에 앞서 수행되는 경우만 설명되었으나, 제안한 방법은 non-CONV가 먼저 수행되고 CONV가 뒤따라 수행되는 형태에도 동일하게 적용 가능하다.

4.4.2 CNN 학습 역전파 과정

CNN의 역전파 과정 (backward path)은 feature map의 gradient를 계산하는 과정 (Step 1)과 weight을 업데이트하기 위해 계산하는 과정 (Step 2)으로 나눌 수 있다. Gradient를 계산하는 과정의 경우 순전파 과정과 동일하게 convolution이 host에서 수행되는 CONV에 해당하며, BN 및 ReLU가 non-CONV에 해당한다. Weight을 업데이트하는 과정의 경우, weight gradient는 matrix multiplication으로 계산되기에 weight gradient를 계산하는 과정이 host에서 수행되며, 계산된 weight gradient를 learning rate과 곱하여 기존 weight에 더하여 weight을 업데이트하는 과정이 PIM에서 수행된다. 편의성을 위해, weight gradient 계산 과정을 CONV, weight 업데이트 과정을 non-CONV로 지칭한다. CONV와 non-CONV를 함께 수행할 때, CONV 대비 non-CONV의 job의 양이 많을수록 CONV 내에 non-CONV를 처리하기 어려워지며, CONV 내에 non-CONV를 처리함으로써 감소하는 전체 수행 시간의 효율 또한 줄어든다. 따라서, 만일 Step 1의 CONV 대비 non-CONV의 job의 양이 Step 2의 CONV 대비 non-CONV의 job의 양에 비해 과도하게 많다면, Step 1의 non-CONV 중

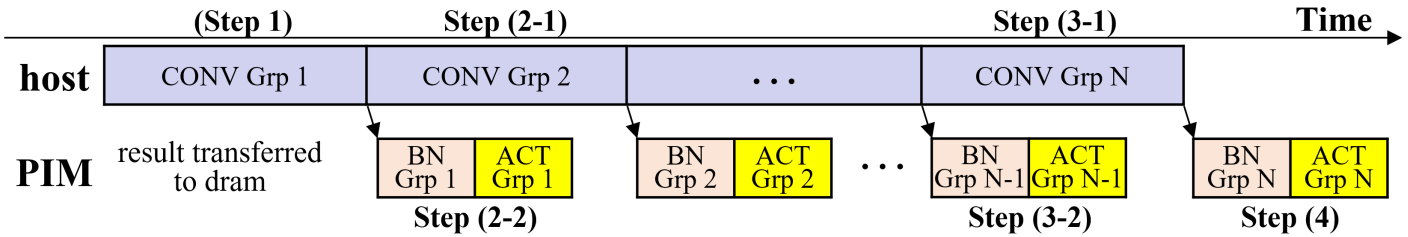

그림 4.7 순전파 과정에서의 host와 PIM의 pipeline 수행 방법

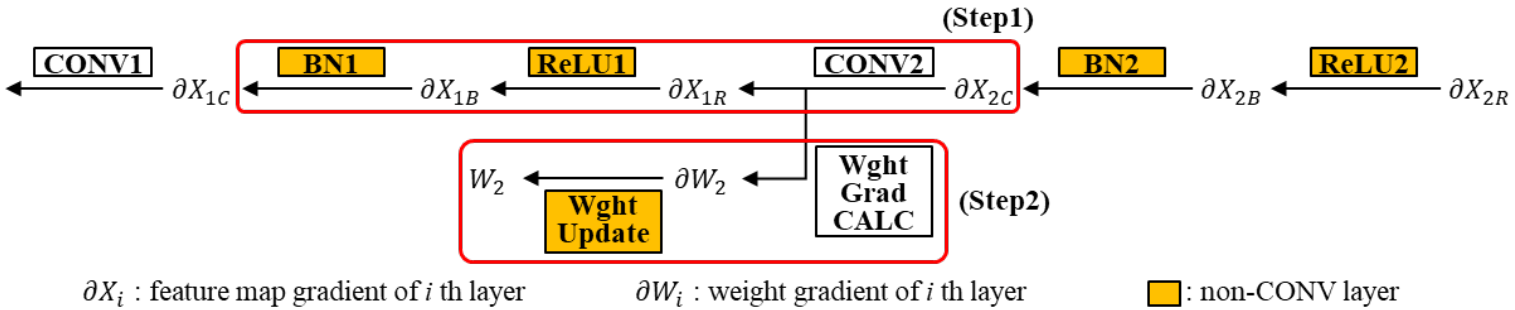

그림 4.8 CNN 학습 역전파 과정에서 pipeline 적으로 함께 수행되는 CONV와 non-CONV의 구분

일부를 Step 2 과정에서 non-CONV 처리 시 함께 처리함으로써 전체 수행 시간을 줄일 수 있다. 상술한 바와 같이 전체 수행 시간을 최소로 하기 위해서는, Step 1과 Step 2의 non-CONV job이 한쪽으로 몰리지 않도록 서로 간에 적절하게 재분배할 수 있어야 한다. 이를 위해서는 feature map gradient를 계산하는 과정과 weight을 업데이트하는 과정 간의 dependency가 없어야 한다. 이를 위해, feature map gradient 계산 시 현재 layer의 CONV와 non-CONV를 함께 수행하는 대신, i+1번째 layer의 CONV와 i번째 layer의 non-CONV를 함께 수행한다. 그림 4.8은 CONV, BN, Activation (ReLU 등)으로 구성된 일반적인 역전파 과정에서 pipelining되어 함께 수행되는 CONV와 non-CONV를 나타낸다. ∂X_i , ∂W_i 는 각각 feature map, weight에 대한 gradient를 나타내며 박스들은 CONV 및 non-CONV 함수를 나타낸다. 동일 숫자의 CONV와 non-CONV는 동일 layer에 속한다. 이때, backward CONV layer의 결과인 feature map gradient $\partial X_{1R} = W_2^T * \partial X_{2C}$ 의 식으로 계산된다. 즉, ∂X_{1R} 의 계산을 위해서는 weight이 업데이트되기 전의 W_2가 필요하기에, weight 업데이트 전에 CONV가 완료되어야 한다. 만일 CONV2, BN2, ReLU2를 함께 pipelining하여 수행하게 되면, non-CONV2에 해당하는 BN2 및 ReLU2를 먼저 수행하게 되어 non-CONV2가 끝나지 않으면 CONV2 또한 완료할 수 없다. 따라서, gradient를 계산하는 과정의 non-CONV job의 일부를 weight을 업데이트하는 과정으로 이동할 수 없게 된다. 반면, 그림 4.8과 같이 동일 layer의 CONV와 non-CONV를 함께 수행하는 대신, i+1번째 layer의 CONV (CONV2)와 i번째 layer의 non-CONV (BN1, ReLU1)를 함께 수행하게 되면 CONV2가 완료된 후 Step 2가 수행되기에 Step 1의 non-CONV job의 일부를 Step 2에 함께 수행할 수 있다.

Step 1과 Step 2의 non-CONV job을 적절하게 배분하기 위해서는 CONV 대비 non-CONV의 job의 양 뿐 아니라, CONV의 memory access 특성 또한 고려해야 한다. 이를 위해, 4.3.3 장에서 제안한 PD_{TH} 를

활용하여 Step 1과 Step 2의 추정 PD_{TH} 값이 동일해지도록 non-CONV job을 배분한다. PD_{TH}는 식 (4.6)과 같이 계산되며, 이 중 PIM에서의 non-CONV 수행에 영향을 받는 변수는 식 (4.5)의 $Work_{NC}$ 뿐이다. $PCAP$ 값은 CONV 대비 non-CONV의 job의 양에 영향을 받으므로, non-CONV job의 양을 조절하여 PD_{TH}를 동일하게 조정할 수 있다.

4.5 실험 결과 및 분석

4.5.1 실험 환경 구성

성능 측정을 위해, cycle accurate한 시뮬레이터인 GPGPU-Sim V4 [69]를 사용하였다. GPU 구조로는 NVIDIA의 V100과 비슷한 구조를 사용하였으며, main memory로는 HBM을 사용하였다. 표 4.1은 실험에 사용된 GPU 및 DRAM의 시뮬레이션 변수들을 나타낸다. PIM의 동작을 위해 bank group 단위의 precharge, activation, PIM 연산을 지원하도록 DRAM 명령어를 추가하여 시뮬레이터를 수정하였다. Bank group마다 4개의 bank가 존재하기에, bank group 단위에서 한번의 PIM 연산을 하기 위해 소요되는 시간은 $t_{CCDL} \times 4$로 가정되었다. Temporary storage를 가지는 PIM 구조를 고려하여 PIM 연산이 효율적으로 수행될 수 있도록 주의 깊게 연산을 PIM에 mapping하였다. 널리 사용되는 CNN network들인 VGG [70], ResNet-50, ResNet-152 [53], DenseNet [71]가 실험되었으며, input으로는 ImageNet [50]을 가정하였다. 네트워크 단위의 성능은 각 layer 단위로 CONV와 non-CONV를 수행한 결과를 더하여 측정되었으며, batch size는 32로 실험하였다.

표 4.1 GPU 및 DRAM 시뮬레이션 변수

SM parameters	80SM, 1200MHz, 65536 registers, 32 threads/warp, 2048 threads, 4 schedulers per core
L1 cache	32B sector, adaptive cache (up to 128KB)
Shared memory	adaptive (up to 96KB)
L2 cache	32B sector, 4.5MB
Memory model	HBM, 24 channels, 16 bank, 652GB/s, 32B column I/O bit width, FR-FCFS
Memory timing parameters (ns)	tRCD=16, tRP=16, tRAS=28, tRC=45, tCL=16, tRRD=2, tCCDS=2, tCCDL=4, tWR=16, tFAW=12, tWL=2

4.5.2 Layer 당 수행 시간 변화

제안하는 host-PIM 간 mode 전환 알고리즘의 성능을 확인하기 위해, 대기 중인 host req의 수만 고려하거나 (NR) [63] PIM mode를 유지하는 시간만을 고려하는 방법 (PD) [72]을 각 layer 별로 비교하였다. NR과 PD의 경우, 전체 수행 시간이 최소가 되는 request의 수 및 PIM mode 유지 시간 값을 구하여 해당 값으로 실험하였다. 실험에 사용된 layer들은 동일 network에서 weight filter의 크기에 따라 ResNet_1 × 1, ResNet_3×3, VGG_3×3, DenseNet_1×1의 4가지 group으로 분류되었으며, 동일 group 내에서 layer 번호가 클수록 network 상에서 후반부에 위치하는 layer이다. 그림 4.9는 ResNet을 구성하는 각 layer 별로 CONV와 non-CONV를 순차적으로 (serial) 수행하였을 때와 pipelining하여 수행했을 때의 시간을 나타낸다. 수행 시간은 순차적으로 수행한 경우의 값으로 normalize되었으며, 수행 시간은 CONV와 non-CONV를 함께 수행한 시간 (편의상 host로 표기)과 non-CONV만 따로 수행한 시간 (편의상 PIM으로 표기)으로 나누어 나타내었다.

두가지 observation이 있다. 먼저, PDNR, PD, NR, baseline 순서로 수행시간이 짧았다. PDNR, PD, NR은 각각 baseline 대비 14.3%, 12.8%, 8.7%만큼 짧은 수행 시간을 보였다. NR의 경우 CONV 동작이 끝나기 전에 non-CONV를 모두 수행하지 못한 경우가 많았다. NR의 경우, bank group에 대기하고 있는 host request의 수가 threshold 값보다 작다면 해당 bank group은 계속 PIM mode를 수행하게 된다. 이에 따라, host request가 처리되지 못해 dead lock이 발생하게 된다. Layer 1, 3, 4, 5, 7, 8, 9, 11, 19, 20, 21, 22, 23, 24 등 대부분의 layer에서 이와 같은 현상이 발생하였다. 상술하였듯이, CONV 동작과 non-CONV 동작 (마지막 channel group의 non-CONV는 제외)이 함께 끝나는 경우 제일 수행 시간이 빠르기에, 이는 NR이 PD 및 PDNR 대비 각각 4.7%, 6.5%만큼 긴 수행 시간을

보이는 주 원인이 되었다. 또한, CONV 내에 non-CONV를 모두 처리한 경우에도 NR은 PDNR보다 낮은 성능을 보였다. NR은 대기 중인 host request의 수만을 고려하여 PIM mode에서 host mode로 전환한다. 따라서, 대기 중인 host request의 수가 threshold보다 적은 상태로 유지된다면 PIM mode로의 전환이 되지 않아 host가 지연된다. 또한, host request를 threshold 값 이상으로 집약하여 처리할 수 없어 host request의 처리 효율성을 증가시키는데 한계가 있다. PD의 경우, 연속하여 수행되는 PIM mode의 최대 시간이 정해져 있어 deadlock 없이 CONV 내에 non-CONV를 모두 처리할 수 있었지만, 역시 PDNR보다는 낮은 성능을 보였다. 이는 PD는 대기 중인 host request의 수를 고려하지 않기 때문이다. 대기 중인 host request 수가 많다는 것은 host GPU의 warp들 중 load/store long memory operation을 수행하고 있는 warp의 비중이 높다는 것을 의미하며, 이는 latency hiding에 사용할 수 있는 남은 warp의 수가 적다는 것을 의미한다. 따라서, host가 연산을 수행하지 못하는 idle time이 증가하게 하게 되고, 이는 전체 수행 시간의 증가를 유발한다. PD와 달리 PDNR은 대기 중인 host request의 수가 많으면 그만큼 PIM mode 수행 시간이 감소하기에 그와 같은 상황에서 더 잘 대응할 수 있었다.

두번째로, CONV (host) 대비 non-CONV (PIM)의 시간이 길수록 PDNR 적용 시 성능 향상이 컸다. 즉, 1 × 1 filter의 layer의 경우가 3 × 3 filter layer보다 성능 향상이 컸다. ResNet 및 DenseNet에서 1×1 filter layer들은 PDNR 적용 시 baseline 대비 수행 시간이 각각 14.1%, 30.1% 감소한 반면, 3 × 3 filter의 경우 각각 6.5%, 14.4%만 감소하였다. 또한 초기 layer일수록 channel 수 대비 feature map size가 커 non-CONV의 비중이 높기에 후반부에 layer에 비해 제안하는 방법에 의해 수행 시간이 더 많이 감소하였다. ResNet 1 × 1 group에서 layer 1은 baseline 대비 수행 시간이 18.2% 감소한 반면, layer 11은 10.5% 감소에 그쳤다. CONV 대비 non-CONV의 시간이 길수록 CONV 내에 non-CONV를 처리하기 위해

CONV를 더 많이 지연시켜야 한다. 그럼에도 불구하고 non-CONV 시간이 길수록 수행 시간이 감소한 비율이 더 컸다는 것은, 지연 시간이 증가하더라도 host request의 지연을 통한 처리 효율성 증가 효과가 계속 유지된다는 것을 의미한다. 따라서, 제안하는 host-PIM 간 memory scheduling 방법이 모든 layer들에 대해 효율적으로 동작한다고 결론 내릴 수 있다.

4.5.3 역전파 과정에서의 non-CONV job 배분 효과

그림 4.10은 역전파 과정에서 feature map gradient를 계산하는 과정과 weight을 업데이트하는 과정 간 non-CONV job을 분배하지 않았을 때와 분배하여 수행했을 때의 수행 시간을 나타낸다. Job을 분배하지 않고 수행한 경우 수행 시간이 baseline 대비 8.4%만 감소하였지만, job을 적절하게 분배하여 수행한 경우 수행 시간이 13.4% 감소하였다. Job 분배에 따른 성능 증가는 학습의 역전파 과정에서 gradient를 계산하는 과정 (Step 1)과 weight을 계산하고 업데이트하는 과정 (Step 2) 중 한쪽의 non-CONV job의 양 대비 CONV job의 양이 다른쪽에 비해 클수록 (즉, 한쪽으로 non-CONV job이 몰린 경우일수록), 총 non-CONV job의 양 대비 CONV job의 양이 많을수록 컸다. 초기 layer일수록 feature map의 크기가 크고 input 및 output channel의 수가 적어 BN, ReLU의 연산량은 많은 반면 weight을 update하는 연산량은 적기에, 초기 layer일수록 제안하는 job 재분배 방법이 효과적이었다. 그림 4.11은 Step 1과 Step 2의 non-CONV job / CONV job의 차이를 나타낸다. ResNet_1×1 group 중 해당 값이 제일 크고, 총 non-CONV job의 비중 또한 제일 큰 layer 2의 경우, job 재분배를 통해 17.4%의 제일 높은 speedup을 보였다. 반면, 해당 값이 작은 layer 12의 경우 job 재분배를 통한 성능 향상이 1% 미만으로

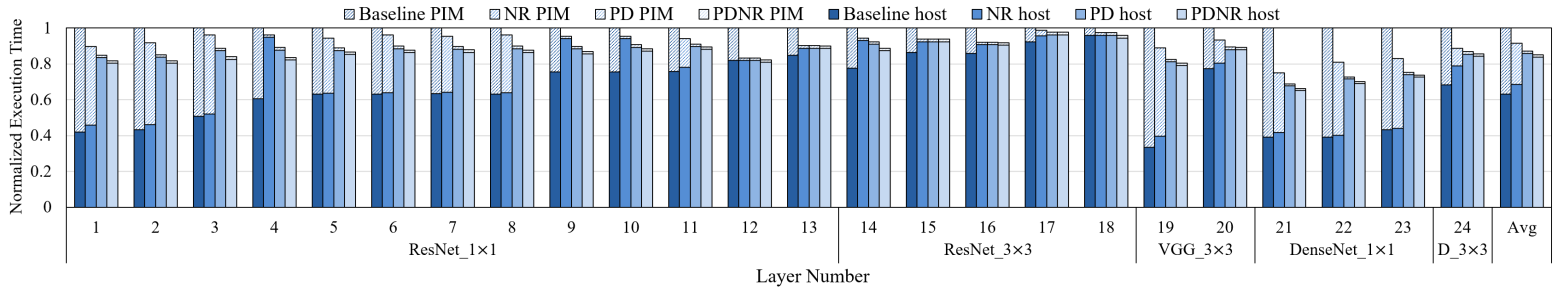

그림 4.9 host-PIM 간 mode 전환 알고리즘에 따른 순전파 과정에서의 각 layer 별 CONV 및 non-CONV 수행 시간

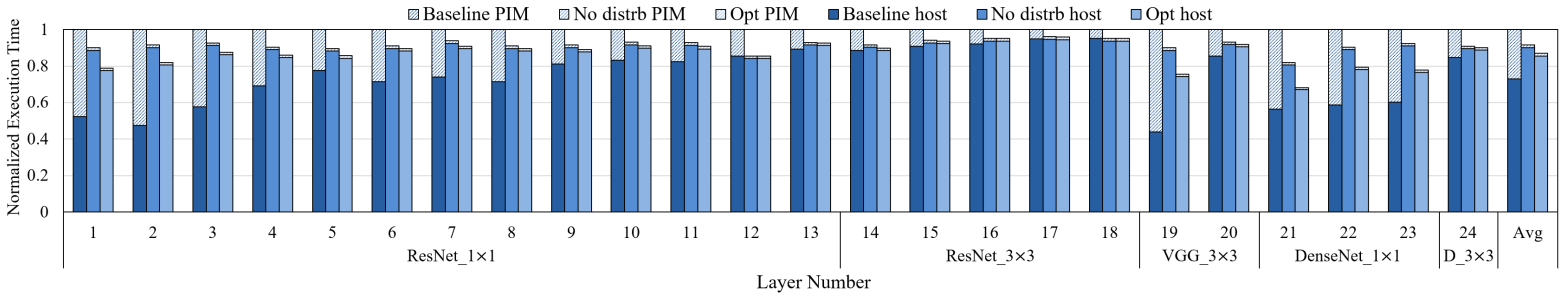

그림 4.10 역전파 과정에서 non-CONV job 분배 여부에 따른 각 layer 별 CONV 및 non-CONV 수행 시간

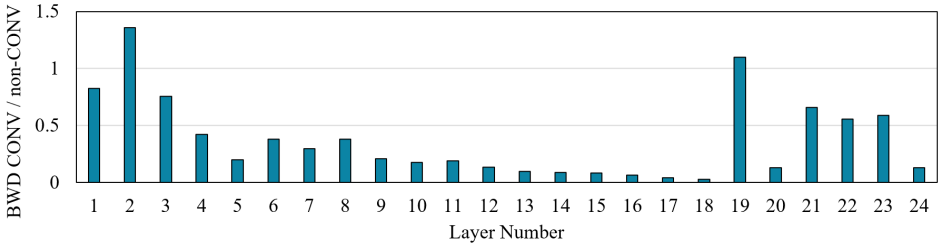

그림 4.11 각 layer 별 Step 1과 Step 2의 non-CONV job 대비 CONV job의 비율 간 차이

미미하였다. 유사하게, VGG_3 × 3 group에서도 layer 19는 job 재분배를 통해 20%의 성능 향상을 보인 반면, layer 20은 1.3%의 성능 향상만을 보였다. 성능 향상 정도의 차이는 있었지만, 모든 layer에서 job 재분배를 통해 성능을 향상시킬 수 있었기에 제안하는 job 재분배가 수행 시간 감소에 효과적이라고 결론 지을 수 있다.

4.5.4 전체 Network Level에서의 수행 시간 변화

그림 4.12은 batch size와 temporary storage (TS)를 변화시켰을 때 CONV와 non-CONV를 serial하게 수행한 경우 대비 전체 network level에서의 speedup을 보여준다. PIM은 non-CONV를 타겟으로 하기에, 전체 network 중 non-CONV 비중이 클수록 network level에서 높은 성능 향상을 보였다. 그림 4.12는 batch size가 32로 고정된 상황에서 TS가 row buffer 크기의 1/8, 1/4, 1/2 일 때 serial 수행 대비 pipeline 수행의 speedup을 나타낸다. TS가 작을수록 CONV와 non-CONV의 pipeline 수행을 통해 serial하게 수행한 경우 대비 높은 성능 향상을 보였다. 예를 들어 ResNet-50의 경우, TS가 1/8, 1/4, 1/2일 때 각각 serial 수행 대비 17.9%, 14.1%, 12.2%의 speedup을 보였다. TS가 작을수록 DRAM row의 open이 빈번하게 발생하여 non-CONV 비중이 증가한다. 4.5.2장에서 언급하였듯이, 제안하는 방법은 non-CONV의 비중이 클수록 효율이

증가하기에 TS가 작을수록 serial 수행 대비 높은 성능 향상을 보였다. 결론적으로, TS size가 변하더라도 제안하는 방법이 유효함을 확인할 수 있다.

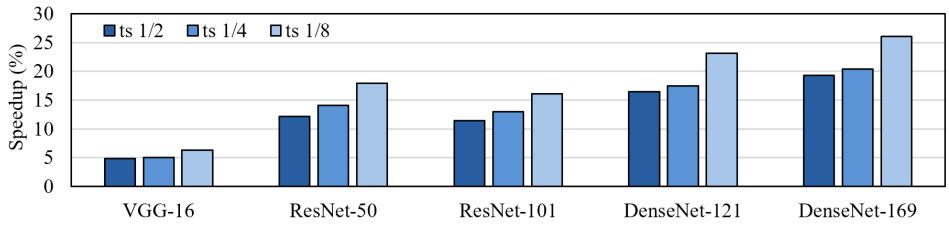

그림 4.12 TS를 1/8, 1/4, 1/2 RB로 변화시킨 경우 전체 network level에서의 serial 수행 대비 수행 시간 변화

4.5.5 제안하는 최적 PD_{TH} 추정 방법의 정확도 및 선택 알고리즘의 수렴 속도

PD_{TH} 선택 알고리즘의 성능을 확인하기 위해, 최적 PD_{TH}와 추정 PD_{TH} 간 유사도 및 최적 PD_{TH}를 얻기 위해 소요된 iteration의 수를 확인하였다. 그림 4.13는 각 layer 별로 최적 PD_{TH} 값과 추정 PD_{TH} 간 상관 관계를 나타낸다. 최적 PD_{TH} 값과 추정 PD_{TH} 값은 0.986의 상관 관계를 가졌으며, 4.1%의 NRMSE (normalized root mean square error) 값을 가졌다. 이는 제안하는 PD_{TH} 추정 계산식이 초기 PD_{TH} 값을 최적 PD_{TH} 값에 근접하게 추정하였음을 의미한다. 최적 PD_{TH} 값을 구하기 위해 진행된 평균 iteration 횟수는 2.5번이였으며, 최대 iteration 횟수는 6번이었다. 따라서, 최적 PD_{TH} 값이 큰 경우에도 제안하는 추정 방법을 통해 초기 PD_{TH} 값을 최적값에 가깝게 추정함으로써 적은 iteration 횟수로 최적값에 도달한 것을 확인 가능하다.

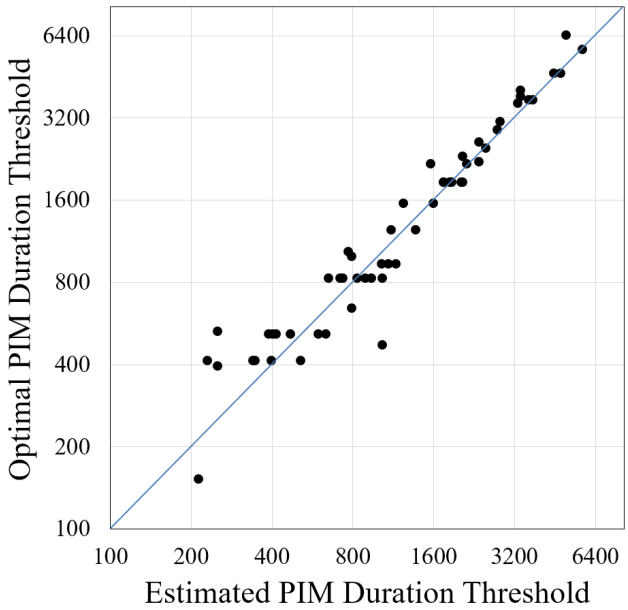

그림 4.13 최적 PD_{TH} 최초 추정값과 실제 최적값과의 상관 관계

4.6 본 장의 결론

본 장에서는 CNN training을 가속하기 위해 convolution (CONV)과 non-convolution (non-CONV)을 각각 host와 PIM에서 동시에 수행하는 방법을 제안하였다. Non-CONV는 output feature map의 channel 단위로 수행되기에, CONV와 non-CONV를 병렬적으로 수행하기 위해 output feature map의 channel들을 grouping하여, channel group 단위로 CONV와 non-CONV를 pipelining하여 수행하는 training flow를 제안한다. 또한, host와 PIM이 memory에 동시에 접근하는 상황에서 전체 수행 시간을 최소로 하기 위해 host와 PIM이 일정 조건을 만족할 때까지 각 bank group 단위로 소유권을 가지는 host-PIM 간 memory scheduling 알고리즘을 제안한다. 실험 결과는 제안하는 방법을 통해 CONV와 non-CONV를 순차적으로 실행하는 경우에 비해 성능을 향상시킬 수 있음을 보여준다.

제 5 장 Image processing의 데이터 접근 패턴을 활용한 PIM에 적합한 lightweight GPU 구조

해당 장에서는 HBM의 logic die에 적합한 GPU 구조 및 최적화 기법을 제안한다. HBM [73] 및 HMC [74]과 같은 3D stakced memory의 발전에 따라, logic die와 DRAM die가 공존할 수 있게 되어 logic die에 프로세서의 탑재가 더 현실에 가까워지게 되었다. Logic die에 탑재될 PIM 중, 프로그래밍이 가능한 GPU 기반의 PIM는 다양한 application들을 처리할 수 있고, CUDA 및 openCL과 같은 잘 정립된 software development kits (SDKs)를 사용할 수 있기에 유망하다. PIM의 넓은 bandwidth를 활용하기 위한 제일 간단한 방법은 충분한 수의 computing unit을 logic die에 탑재하는 것이다. GPU는 parallelism 및 latency hiding을 위해 많은 수의 thread들을 context switching한다. 따라서 context들을 임시로 저장하기 위해 매우 큰 크기의 register file을 필요로 하며, cache 크기 또한 증가한다. Logic die의 면적 제한을 고려하면, 이러한 큰 용량의 on-chip SRAM은 logic die에의 GPU의 탑재를 어렵게 한다. 본 논문에서는 CNN의 convolution과 같은 image processing이 알고리즘 적으로 일정한 패턴을 보인다는 것을 활용하여 PIM에 적합한 lightweight한 GPU 구조와 함께 제안하는 구조를 위한 최적화 기법을 제안한다. 제안하는 최적화 기법은 image processing의 memory access 패턴과 data locality가 보존되도록 processing unit에 GPU의 thread block (또는 CTA)를 할당한다. 이런 예측 가능한 CTA 할당을 기반으로, 접근 패턴을 활용한 prefetcher를 탑재하였으며, GPU에 탑재되는 hardware 자원들을 감소시킨다. 이를 통해 충분한 수의 processing unit들을 탑재하여 lightweight한 구조임에도 높은 성능을 확보할 수 있도록 한다.

5.1 관련 연구

5.1.1 Processing In Memory

Top-PIM [9]는 3D stacked memory의 logic die에 GPU의 SM을 탑재하였으며, 이에 따른 PIM의 성능을 예측하는 기법을 제안하였다. 그러나 해당 연구는 PIM에 적합한 operation인지 여부를 확인하지 않았으며, SM의 구조를 PIM에 적합하게 변경하지 않았다. Xie et al. [16]은 graphic rendering을 위해 texture unit을 탑재한 PIM 구조를 제안하였으며, PIM에 적합하게 rendering 알고리즘을 변경하였다. 그러나, 해당 연구는 3D rendering라는 특정 application을 가속하는 것을 중점적으로 수행하였으며, 전반적인 image processing 알고리즘을 target으로 하지 않았다. 그 외 graph processing에 적합한 PIM 구조를 연구한 논문들도 있지만 [7], 해당 연구들은 image processing에는 적용하기 어렵다. Panda et al. [17]는 memory access pattern을 분석하여, row buffer locality를 향상시키기 위한 PIM을 제안하였다. 이에 더해, L2 cache로의 prefetching을 활용하여 성능을 향상시켰다. 그러나 GPU의 CTA scheduling을 고려하지 않았으며, 이에 따라 성능 향상 정도가 제한적이었다. 또 다른 연구들은 PIM이 탑재된 GPU system에서의 task scheduling 기법을 제안하였다. TOM [61]는 PIM에 offload 될때의 bandwidth 이득을 추정하여 instruction level에서 PIM과 host간 scheduling을 수행하였으며, Pattnaik et al. [58]는 workload를 kernel 단위로 regression을 통해 PIM에 적합한지 파악하였다.

5.1.2 GPU에서의 CTA Scheduling

Kayiran et al. [75]는 application의 각 phase마다 적합한 CTA의 수가 다르다는 것에 착안하여 CTA 수를 제한하였다. 그러나, PIM에 탑재되는 GPU는 resource의 한계로 활용할 수 있는 CTA의 수가 적기에 CTA의 수가 감소된다면 성능이 하락한다. Lee et al. [76], A. Jog et al. [77]는 inter-CTA locality를 활용하는 CTA scheduler를 제안하였다. 연속한 두 CTA를 하나의 SM에 할당함으로써 inter-CTA locality를 활용할 수 있었지만, 이는 data access pattern을 보존하지는 못한다. Li et al. [23]는 inter-CTA locality의 원인을 분류하여 실제 GPU에서 사용될 수 있는 CTA clustering framework를 제안하였다. 제안하는 방법은 [23]와 같이 복잡한 CTA remapping function 없이, 한 개의 CTA만을 각 SM에 할당하도록 최적화 기법을 제시한다.

5.1.3 GPU에서의 Prefetching

많은 기존 연구들이 GPU에서의 prefetching 방법을 제안하였다 [78], [79], [80]. 그러나 CTA가 SM에 할당되는 순서가 알려지지 않았기에, prefetch의 범위가 CTA 내로 제한되었다. Jeon et al. [81]는 CTA aware prefetching을 제안하였다. 각 CTA의 leading warp가 먼저 schedule 된 후, 나머지 warp와의 차이를 기반으로 stride를 설정하였다. 그러나 해당 연구는 첫번째 warp에 대한 prefetching이 되지 않는다는 단점이 있다. 따라서 기존 연구들은 CTA 내 warp 수가 충분히 많을 때 효과적이며, thread 수가 부족한 PIM GPU 환경에 적용되기 어렵다.

5.2 Motivation

5.2.1 PIM GPU system에서 Image Processing 알고리즘 처리 시 기존 GPU 구조의 비효율성

초기 GPU는 graphic processing을 위해 설계되었지만, general application을 처리하기 위해 다양한 module들이 추가되었다. 이러한 module들은 GPU가 logic die의 면적 및 전력 제약 조건을 맞추기 어렵게 한다. 그러나, image processing의 예측 가능한 패턴을 활용할 경우, 기존 GPU의 비효율적인 부분들을 해결할 수 있다.

CTA scheduling. 각 SM에 할당된 CTA들이 작업을 끝내는 순서는 SM에 할당된 순서와 다르며, round-robin (RR) scheduling은 여러 SM에 CTA들을 빈 CTA slot이 발생하는 대로 할당한다. 이는 두가지 문제점을 야기한다. 첫번째로, data locality를 활용하기 어려워진다. 대부분의 image processing application들은 인접한 block의 data를 연속적으로 읽는다. GPU는 global memory로의 load와 store를 warp 단위로 합친다. CTA가 영역의 경계 부분에 있는 data를 load 하는 경우, coalescing으로 인해 인접한 data 또한 함께 load 된다. 이 경우, 인접한 block을 처리하는 CTA를 동일한 SM에 할당하여 data locality를 증가시킬 수 있다. Filtering 알고리즘이 대표적인 예이다. 이렇듯 data locality를 활용할 수 있는 기회가 있음에도 불구하고, baseline CTA scheduling 알고리즘은 인접한 block에 대해 연산을 수행하는 CTA들을 다른 SM에 두어 SM 내에서 data reuse를 어렵게 한다. 인접한 CTA들이 다른 SM에 할당되는 경우, L2 cache에서 data를 중복하여 읽어야 한다. 이는 memory hierarchy에서의 혼잡을 증가시키며, 추가적인 에너지를 소모한다. 그림 5.1은 SM에서 L2 cache로의 traffic과 SM에서 DRAM으로의 traffic을 나타낸다. 전자의 경우가 후자의 경우보다 request 수가 높은 것을 확인할 수 있다. 이는

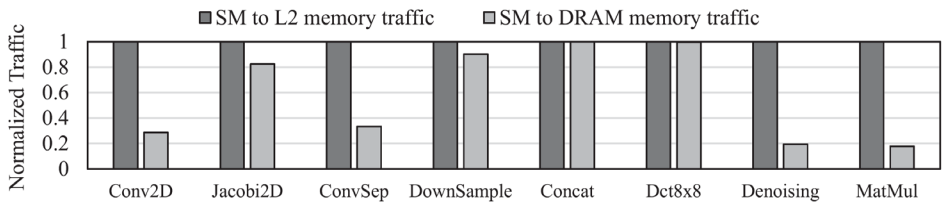

그림 5.1 SM에서 L2 cache로의 traffic과 SM에서 DRAM으로의 traffic의 비교

SM이 data를 중복하여 읽는다는 것을 의미한다. 두번째로, image processing의 data access pattern을 활용하기 어렵다. RR scheduling이 사용되는 경우, SM에 할당되는 CTA의 순서는 data access pattern과 달라져, 예측 가능했던 원래의 pattern이 사라지게 된다. 그림 5.2 (a)는 x 축으로 raster scan 순서로 SM마다 하나의 CTA가 할당되는 예시를 나타낸다. 큰 직사각형은 전체 image를 나타내고, 작은 직사각형은 CTA 단위로 수행하는 block을 의미한다. 직사각형 내부 숫자는 CTA의 index를 나타내며, 색깔은 어떤 SM에 할당되었는지를 의미한다. 첫번째 행에서는 CTA 0, 1, 2, 3이 각각 SM 0, 1, 2, 3에 할당되었다. 이후, SM1이 SM0보다 먼저 할당된 작업을 끝내면서, CTA4는 SM1에 할당되었다. 이는 SM에의 CTA 할당 순서 예측이 어렵다는 것을 보여준다. SM0에 할당된 영역의 간격은 6 → 2 → 3 → 2 순서로 변한다. 그림 5.2 (b)의 경우, CTA 할당이 application의 data access pattern을 고려하여 행해진다. 각 SM에 할당된 영역의 간격은 모두 1로 동일하다. 이러한 예측 가능한 data access는 memory로부터의 data load 효율을 증가시킬 수 있는 기회를 제공한다.

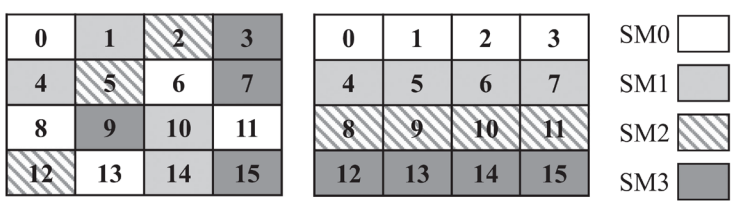

그림 5.2 SM에의 CTA 할당 예시 (a) RR 기반 할당 예시 (b) pattern aware 할당 예시

많은 수의 thread. GPU는 memory load와 같은 긴 latency를 가지는 instruction을 효율적으로 처리하기 위해 많은 수의 thread를 가지고 있다. Thread 간 빠른 context switching을 위해 매우 큰 크기의 register file을 가지고 있으며, 이는 면적과 leakage 전력량을 증가시킨다. 이에 더해, 수많은 thread들이 L1 cache를 공유하기 때문에, thread 당 실질적인 cache 점유율은 매우 낮고, 이는 잦은 cache eviction을 일으켜 높은 miss rate을 야기한다. 그림 5.3은 여러 image processing 벤치마크의 L1 cache miss rate을 나타낸다. 평균적으로 69%의 높은 miss rate을 보이는 것을 확인할 수 있다. 이는 image processing의 높은 data locality가 L1 cache 단에서 잘 활용되지 못하고 있음을 의미한다.

L2 cache. L2 cache는 L1 cache와 main memory 간의 latency 및 bandwidth 차이를 경감해준다. 그러나, PIM의 경우 높은 memory bandwidth를 활용할 수 있기에 L1 cache와 memory 간 bandwidth 차이가 크지 않으며, L2 cache의 넓은 면적은 PIM의 면적 및 전력 제약 조건을 맞추기 어렵게 한다.

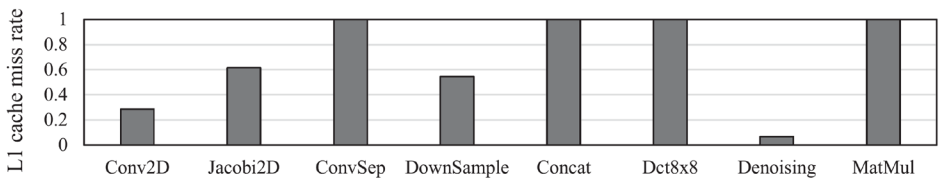

그림 5.3 Image processing benchmark들의 L1 cache miss rate

5.3 제안하는 GPU 기반 PIM System

5.3.1 Overview

해당 장에서는 PIM의 면적 및 전력 제약 조건을 만족하는 GPU 기반 architecture를 제안하며, 이를 효율적으로 활용하기 위한 최적화 기법이 함께 제안된다.

PIM System 구조. 그림 5.4는 PIM stack의 logic die에 위치한 PIM GPU 구조를 나타낸다. PIM GPU는 main GPU system을 보조하는 형태로 사용될 수도, 혼자 사용될 수도 있다. Memory가 TSV로 바로 연결되어 사용되기에, 낮은 latency와 높은 bandwidth로 memory에 접근 가능하다. Logic die의 물리적인 제약 조건을 고려하여, SM 수, register 크기, L1 cache 크기는 일반적인 fermi 기반 GPU의 1/4로 설정되었으며, thread의 수는 1/6으로 설정되었다. Image processing application의 access pattern을 활용하는 prefetcher가 SM 별로 탑재되어 있으며, memory hierarchy가 단순화되었다.

PIM Programming 모델. 제안하는 PIM GPU는 CUDA 기반의 programming model을 사용한다. PIM GPU에 적합한 kernel을 찾는 연구가 많이 있지만 [58], 본 연구의 관심은 아니며, 프로그래머가 PIM으로 offload 할 kernel을 선택하도록 하였다. PIM GPU를 활용하기 위해, 사용자는 제공되는 간단하지만 효율적인 최적화 guideline에 따라 application을 프로그래밍 한다. 이를 통해 data access pattern을 활용할 수 있도록 CTA가 할당된다.

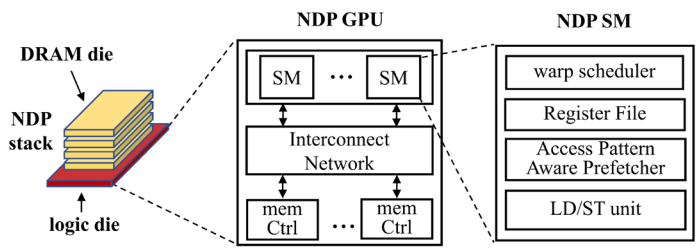

그림 5.4 PIM stack과 제안된 PIM GPU 구조

5.3.2 Access Pattern을 고려한 CTA 할당

CTA-to-SM 할당은 CTA가 처리하는 data 영역의 정보에 기반하여 행해진다. 할당 방법은 그림 5.5와 같이 3가지로 구분된다. Row-major와 column-major의 경우 행 또는 열로 연속한 영역이 동일한 SM에 할당되며, tile-based의 경우 전체 image가 SM 수만큼의 tile로 나눠져 각 tile이 하나의 SM에 할당된다. 표 5.1은 application의 access pattern에 따른 CTA 할당 방법을 나타낸다 Raster scan 또는 tensor 기반 application pattern의 경우, scan 방향에 따라 row 또는 column-major 방법이 선택된다. Tile-based 할당은 block-based scan에 사용된다. 제안하는 CTA 할당 방법은 하나의 SM이 연속된 data 영역을 처리하도록 하여 data locality를 모두 활용할 수 있게 하고 data access pattern을 예측 가능하게 한다. Application의 특성을 활용함으로써, SM 내 thread 수가 감소하더라도 성능을 유지할 수 있게 한다. CTA 할당은 코드 변경을 통해 소프트웨어적으로 구현되었다 (software CTA scheduling). 소프트웨어 적인 방 CTA scheduler를 변경하여 CTA를 SM에 특정 순서로 할당하는 방법으로도 구현할 수 있다 (hardware CTA scheduling). 하드웨어 적인 방법은 사용자가 코드를 수정하지 않아도 된다는 장점이 있지만, 간단한 코드 변경을 통해 instruction의 수를 감소시켜 추가적으로 성능을 향상시킬 수 있다. 이는 Guideline 1에서 더 자세히 설명된다. 본 장의 나머지 부분에서는 pattern-aware CTA 할당을 위한 4가지 guideline을 제시한다.

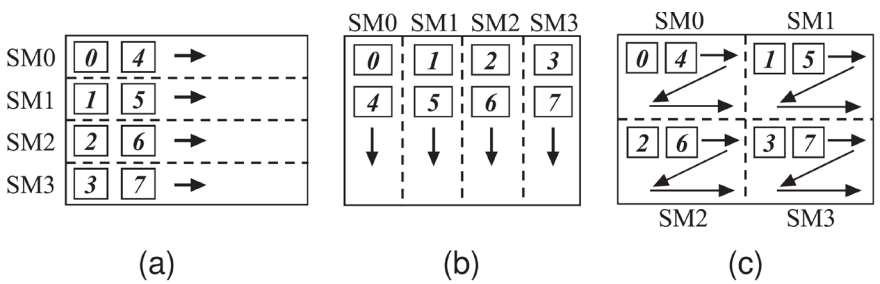

그림 5.5 CTA 할당 방법 (a) row-major (b) column-major (c) tile-based

표 5.1 Application의 access pattern에 따른 CTA 할당 방법

Data access pattern of application	CTA allocation scheme
Raster scan	Row or column major
Block based	Tile based
Tensor	Row or column major
Data dependent	Depending on local pattern

Guideline 1은 CTA-to-SM mapping을 위한 software 기반 방법을 제시하며, Guideline 2, 3, 4는 PIM GPU의 적은 하드웨어 자원을 효율적으로 활용하기 위한 프로그래밍 방법을 제시한다. 이해를 돕기 위해 Convolution 2D 알고리즘의 pseudo 코드를 사용한 예시가 함께 제공된다.

Guideline 1. SM에의 CTA 할당 방법. 소프트웨어 CTA 할당 방법은 하나의 CTA만을 각 SM에 할당한다. CTA는 for loop으로 x, y, z 방향으로의 iteration 범위를 조정하여 의도한 pattern대로 data 영역을 이동하며 동작한다. 그림 5.6은 전체 image가 4개의 영역으로 나눠져 각 SM에 할당된 경우를 나타낸다. 그림 5.6의 오른쪽은 SM1에 할당된 이미지 영역을 확대한 것으로, 하나의 SM에 할당된 영역 내에서 data가

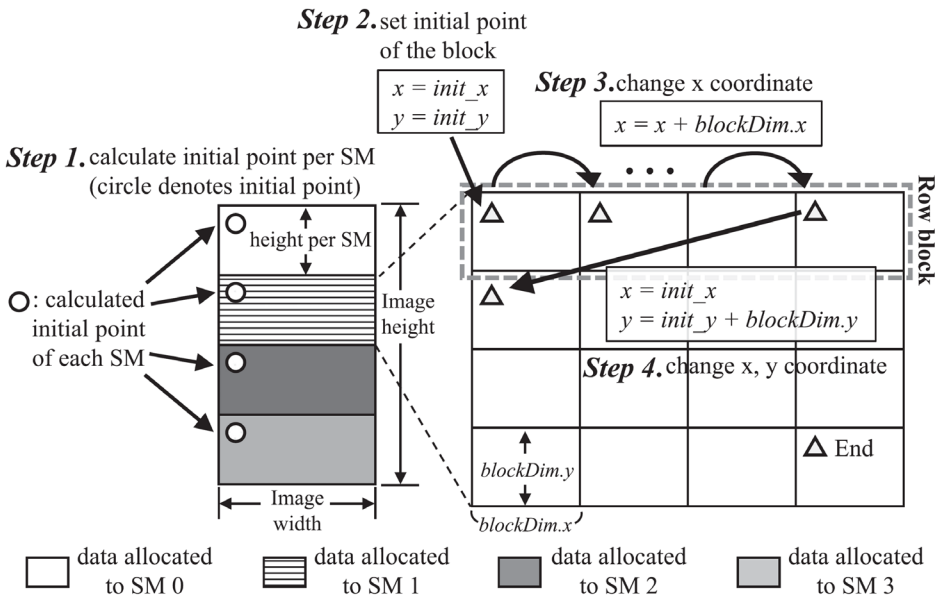

그림 5.6 소프트웨어 기반 row-major pattern aware CTA 할당 예시

처리되는 순서를 나타낸다. 그림의 작은 직사각형은 한번에 처리되는 영역을 나타내며, *blockDim.x*와 *blockDim.y*의 너비와 높이를 가진다. 회색 점선으로 둘러쌓인 *image width* × *blockDim.x*의 영역은 *row block*으로 명명된다. 먼저, 각 SM에 할당된 영역의 초기 좌표 (*init_x*, *init_y*)가 계산된다 (*Step 1*). 이는 그림 5.6에서 원으로 표기되었다. *Step 2*에서, SM에 할당된 영역의 첫번째 block의 x 및 y 좌표가 변수 *x*와 *y*에 할당된다. 이후, 해당 영역에 대해 convolution 연산이 수행된다. 해당 영역에 대한 연산이 완료되면, 다음 block으로 이동한다. *Step 3*는 row-major 방향으로 다음 block으로 이동하는 과정을 보여준다. 변수 *x*의 값은 다음 block의 x 좌표의 값인 *x* + *blockDim.x*으로 변경된다. *Step 3*는 *row block*의 끝에 도달할때까지 반복된다. *Row block*의 끝에 도달하게 되면, *Step 4*와 같이 다음 *row block*으로 이동하여 convolution 연산을 수행한다. *Step 4*에서, 다음 *row block*의 첫번째 block의 x 및 y 좌표가 변수 *x*와 *y*에 할당된다. 즉, (*x*, *y*) = (*init_x*, *y* + *blockDim.y*)가 된다. *Step 3*와 *Step 4*는 SM1에 할당된 영역을 모두 처리할 때까지 반복된다.

원본 코드를 위의 과정대로 수정하기 위해, (1) kernel을 시작할 때, CUDA의 *gridDim*을 SM 수와 동일하게 설정하여 SM 당 하나의 CTA만 할당되게 한다. (2) CTA 당 SM의 수를 Guideline 3에 명시된 바와 같이 256으로 설정한다. (3) CTA가 application의 access pattern을 따라 영역을 이동할 수 있도록 for 문을 추가한다. 위의 3가지를 제외하고, 알고리즘이 실제 연산을 수행하는 부분의 코드는 변경이 거의 필요하지 않아, 다양한 알고리즘에 적용하기 용이하다.

그림 5.7 (a)는 baseline 2D convolution 알고리즘의 pseudo 코드이다. 2D convolution은 row 방향으로의 raster scan access pattern을 보이기에, row-major CTA 할당 방법대로 코드가 변경되어야 한다. 그림 5.7 (a)의 2, 3 번째 줄에서 하나의 thread가 각 pixel에 할당된다. Convolution filter 연산은 5~9 번째 줄에서 수행된다. 그림 5.7 (b)는 row-major CTA 할당을

위해 변경된 코드이다. 그림 5.6의 *Step 1*을 위한 초기 좌표 계산은 3, 4 번째 줄에서 수행된다. 첫번째 row block을 위한 for 문을 수행하기 전에, 7번째 줄에서 변수 *y*의 값을 *init_y*로 설정하며, 9번째 줄은 변수 *x*의 값을 *init_x*로 설정한다. 이는 *Step 2*에 해당한다. 11~15번째 줄은 하나의 CTA 영역에 대한 convolution 연산을 수행하며, 17줄은 변수 *x*를 변경하여 다음 CTA 영역으로 이동한다. 10번째 줄은 row block의 끝까지 처리하여 Step 3가 완료되었는지 파악한다. *Step 4*를 위해 19줄에서 변수 *y*가 변경되며, 변수 x가 9번째 줄에서 다시 *init_x*로 변경된다. CTA 할당을 위해 코드를 변경하였음에도 불구하고, 그림 5.7 (a)에서 convolution을 수행하는 5~9번째 줄은 그림 5.7 (b)의 11~15번째 줄로 동일하게 보존된 것을 확인할 수 있다.

상술하였듯이, 소프트웨어 적인 CTA 할당 방법은 instruction의 수를 감소시킬 수 있다. 그림 5.7 (a)의 코드를 수행할 때, 하드웨어 CTA scheduler는 SM에 더 이상 CTA를 할당할 수 없을 때 까지 CTA를 할당한다. 이 경우, 매번 CTA가 새로 할당될 때마다 *pixel_Pos x*, *y* 와 같이 *blockIdx* 및 *threadIdx*와 연관된 변수들은 새로 계산되어야 한다. 반면, 그림 5.7 (b)와 같이 소프트웨어 적으로 CTA가 할당되는 경우 해당 변수들은 초기 위치를 계산할 때 한번만 계산되며, index 계산은 *blockDim*을 더하는 연산만 수행하면 된다. 즉, 소프트웨어 적인 CTA 할당은 index 계산에 소모되는 instruction의 수를 감소시킨다.

RR scheduling 대신 제안하는 방법을 사용할 경우, SM 간 imbalance가 발생할 수 있다. 이는 성능 하락으로 이어지기에, imbalance를 완화하는 것이 중요하다. Imbalance의 원인으로는 첫번째, data가 SM에 동일하게 분배되지 않을 수 있으며, 두번째, 각 SM이 data를 처리하는 시간이 동일하지 않을 수 있다. 그러나, 이는 제안하는 CTA 할당 방법에서는 문제가 되지 않는다. 제안하는 방법은 image가 SM에 균일하게 배분될 수 있는 image processing application을 타겟으로 한다. 또한, data를 처리하

```
1   __global__ void conv2D(image input, image output, coeff convKernel){
2      pixel_Pos x = blockIdx.x * blockDim.x + threadIdx.x
3      pixel_Pos y = blockIdx.y * blockDim.y + threadIdx.y
4      int i, j
5      for(i = -kernel_width/2; i<kernel_width/2; i++)
6         for(j = -kernel_width/2; j<kernel_width/2; j++)
7            conv_Result = input[x-i][y-j] * convKernel[i][j]
8         endfor
9      endfor
10     output[x][y] = conv_Result
11  }
```

(a)

```
1   __global__ void conv2D(image input, image output, coeff convKernel){
2      int height_per_SM = input.height/number_of_SM
3      pixel_Pos init_x = blockIdx.x * input.width + threadIdx.x
4      pixel_Pos init_y = blockIdx.y * height_per_SM + threadIdx.y
5      int iter_row, iter_col
6      pixel_Pos x, y
7      y = init_y
8      for(iter_col = 0; iter_ col < height_per_SM; iter_ col += blockDim.y)
9         x = init_x
10        for(iter_row = 0; iter_row < input.width; iter_row += blockDim.x)
11           for(int i = - kernel_width / 2; i < kernel_width / 2; i++)
12              for(int j = - kernel_width / 2; j < kernel_width / 2; j++)
13                 conv_Result = input[x-i][y-j] * convKernel[i][j]
14              endfor
15           endfor
16           output[x][y] = conv_Result
17           x += blockDim.x
18        endfor
19        y += blockDim.y
20     endfor
21  }
```

(b)

```
1   __global__ void conv2D(image input, image output, coeff convKernel){
2      int height_per_SM = input.height/number_of_SM
3      pixel_Pos init_x = blockIdx.x * input.width + threadIdx.x
4      pixel_Pos init_y = blockIdx.y * height_per_SM + threadIdx.y
5      int iter_row, iter_col
6      pixel_Pos x, y
7      y = init_y
8      for(iter_col = 0; iter_ col < height_per_SM; iter_ col += blockDim.y)
9         x = init_x
10        for(iter_row = 0; iter_row < input.width; iter_row += blockDim.x)
11           for(int i = - kernel_width / 2; i < kernel_width / 2; i++)
12              for(int j = - kernel_width / 2; j < kernel_width / 2; j++)
13                 conv_Result = input[x-i][y-j] * convKernel[i][j]
14              endfor
15           endfor
16           output[x][y] = conv_Result
17           __syncthreads
18           x += blockDim.x
19        endfor
20        y += blockDim.y
21     endfor
22  }
```

(c)

그림 5.7 (a) baseline code (b) pattern-aware CTA 할당이 적용된 코드 (c) pattern-aware CTA 할당과 synchronization이 적용된 코드

는 시간은 memory에서 data를 load하는 시간과 연산하는 시간을 구분할 수 있다. 이중, data를 load 하는 시간이 특히 SM 간의 load balancing에 영향을 미칠 수 있다. Image processing의 경우, 프로그램을 수행하는 동안 arithmetic intensity가 거의 균일하게 유지된다. 따라서, data를 load하는데 소요되는 시간이 거의 일정하다. 또한, SM 간 연산 능력은 거의 동일하기에, 각 SM이 data를 처리하는데 소요되는 시간은 거의 차이가 나지 않는다. 따라서, SM간 load imbalance는 거의 발생하지 않으며, 5.4장에서 실험적으로 SM 간 수행 시간 차이가 1% 이내인 것을 확인한다.

Guideline 2. Syncthreads. Data locality의 부족은 cache 효율을 감소시켜 stall을 발생시킨다. 이를 방지하기 위해, shared memory와 동일하게 L1 cache를 사용할 때도 명시적으로 동기화를 사용하였다. 이는 warp간 처리 속도를 비슷하게 유지하여 data locality의 감소를 방지한다. 알려져 있듯이, GPU는 CTA 단위로 동기화를 지원하지만, 하나의 CTA만이 SM에 할당된 경우 각 CTA는 넓은 영역의 data를 처리하게 된다. 일정 data 영역을 처리할 때마다 명시적인 동기화를 하지 않는 경우, CTA 내에서 warp 간 다른 처리 속도로 인해 data locality가 감소하게 된다. 명시적 동기화 대신, loose-round-robin (LRR) warp scheduling이 사용될 수도 있다. 그러나, LRR은 warp간 우선도를 완전히 동일하게 보장하지는 않는다. 그림 5.8은 PIM configuration 을 변경해가며 명시적 동기화가 없이 LRR warp scheduling을 사용하였을 때 각 SM에서 시간에 따라 수행된 instruction의 수를 나타낸 것이다. 그림 5.8 (a)는 16개의 SM과 SM 당 8KB L1 cache가 있는 경우를 나타낸다. 특정 warp의 수행속도가 빠른 것을 확인할 수 있다. 초기에 schedule된 warp들은 prefetch된 data를 사용하여 빠르게 연산을 수행하지만, 시간이 지남에 따라 memory traffic이 증가하여 memory latency가 증가하게 된다. 따라서, 나중에 prefetch한 data들은 늦게 도착하게 되고, 해당 data를 사용하여 연산하는 warp들은 prefetch를 통한 이득을 보지 못해 느리게

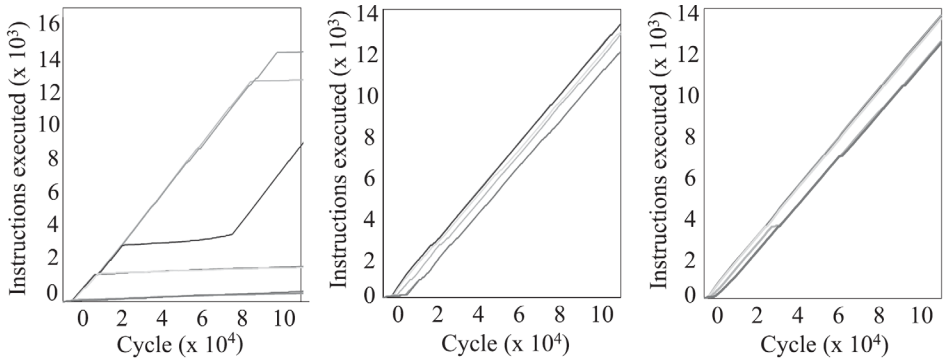

그림 5.8 명시적 동기화가 없는 경우 3가지 PIM configuration에서 시간에 따라 수행된 instruction의 수 (a) 16 SMs + 8 KB L1 cache (b) 16 SMs + 32 KB L1 cache (c) 8 SMs + 8 KB L1 cache

수행된다. 이에 더해, PIM GPU의 작은 cache 크기로 인해, 느린 warp들이 prefetch한 data는 먼저 schedule된 warp들에 의해 cache eviction이 발생할 확률이 높다. Cache 크기가 크거나 traffic이 적은 경우에는 warp 간 속도가 동일하게 유지될 수 있다. 예를 들어 그림 5.8 (b)와 같이 16개의 SM이 있으며 SM 당 cache size가 32KB인 경우, cache eviction등이 거의 발생하지 않아 warp 간 속도가 동일한 것을 확인할 수 있다. 또한, 그림 5.8 (c)와 같이 SM 수가 줄어들어 memory traffic이 감소한 경우에도 warp 간 속도가 거의 동일한 것을 확인할 수 있다.

명시적 동기화의 사용은 일부 warp의 실행 중단을 야기하여 속도 저하를 유발할 수 있다. 이로 인한 overhead는 data miss가 발생하지 않는 perfect memory 환경에서 동기화를 한 경우와 하지 않은 경우를 비교하여 측정될 수 있다. 이런 이상적인 환경에서, 동기화는 5~10%만큼 수행 시간을 증가시켰다. 그러나, PIM GPU와 같이 작은 cache 크기를 가지고 prefetcher를 사용하는 경우에는 그림 5.8 (a)와 같이 명시적 동기화 없이는 warp간 속도를 일정하게 유지하기가 어렵다. 따라서, overhead에도 불구하고 명시적 동기화를 사용하는 것이 더 이득이다. 그림 5.7 (c)는 동기화를 사용한 코드이다. 그림 5.7 (b)와 거의 동일하지만, 17줄에

__syncthreads() 명령어를 추가하여 *Step 2*와 *Step 3* 사이에 동기화를 수행한다.

Guideline 3. CTA 크기 설정. 제안하는 PIM GPU의 thread 수는 256개로 제한되어 있으며, 하나의 SM에 하나의 CTA만이 할당된다. 따라서, CTA의 크기 (*blockDim.x* × *blockDim.y*)는 256으로 설정되어야 한다.

Guideline 4. Shared Memory의 사용. GPU의 shared memory는 사용자가 관리하여 사용하는 on-chip memory이다. 그러나, shared memory의 사용을 위해서는 data load와 shared memory instruction이 모두 사용되어야 하기에 일반적인 cache로의 load에 비해 instruction 수를 증가시킨다. Shared memory는 data가 반복되어 사용되거나 global memory로의 접근이 coalesce되지 않은 경우 사용된다. 제안하는 PIM GPU 구조는 prefetching 및 적절한 CTA 할당을 통해 cache의 효율을 증가시켰기에, coalescing 문제가 없다면 cache를 사용하는 것이 더 권장된다. 예를 들어, ConvolutionSeperable [82] 알고리즘의 경우, shared memory 대신 L1 cache를 사용할 경우 instruction의 수가 28% 감소한다. 한편, texture unit이나 constant cache와 같은 다른 LDST unit은 read-only이며, boundary 처리와 같은 특정 상황에 유용하기에 해당 unit을 사용하는 코드 부분은 변경하지 않는 것이 권장된다.

5.3.3 PIM GPU 구조

SM 구조. PIM GPU의 SM은 전력 및 면적 제약 조건을 만족해야 하기에, 성능과 이를 고려하여 warp의 수, on-chip SRAM의 크기 등을 고려해야 한다.

(1) Pattern-aware prefetcher : L1 cache의 효율을 증가시키고, 높은 memory bandwidth를 활용하기 위해 stride 기반의 prefetcher가 SM마다

탑재되었다. Application의 access pattern이 제안하는 CTA 할당 방법에 의해 보존되었기에, 기존 GPU prefetching과 달리 history table 등을 필요로 하지 않으며, 간단한 stride 기반 prefetcher로 높은 정확도를 달성할 수 있다. 먼저, image 경계가 아닌 부분에서의 stride (*base stride*)와 경계에서의 stride (*modified stride*)가 계산된다. Image 내에서 현재 위치에 따라 두가지 stride 중 한 개가 선택되어 계산에 사용된다. 그림 5.9는 제안하는 pattern-aware prefetcher의 구조를 나타낸다. Input으로는 image의 start address (*image start address*), image의 크기 (*image size*), CTA dimension (*blockDim.x*, *blockDim.y*), core에서의 demand load address가 사용된다. Access pattern의 (x, y, z) 중의 방향에 따라, CTA block의 *blockDim.x*, *blockDim.y*, *blockDim.z* 중 하나가 base stride로 사용된다. 예를 들어 x 축을 따라 raster scan으로 수행되는 경우 *blockDim.x*가 사용된다. Prefetch가 요청된 load address의 image 내에서의 위치는 demand load address에서 image start address를 뺀 값을 *image size*로 나눠 계산된다. *Base stride*가 계산된 위치에 더해지며, 행 또는 열이 변경되는지 파악하기 위해 *image size*와 비교된다. 만일 변경되는 경우, $modified\ stride = base\ stride + image\ width \times blockDim.y$ 가 *base stride* 대신 사용된다. Stride는 demand load address에 더해져 prefetch address로 사용되며, *base stride*와 *modified stride*는 kernel launch 시 계산되어 저장된다.

(2) Active Warp의 수 : Access pattern-aware CTA 할당과 prefetch는 data load로 인한 stall을 효율적으로 감소시킨다. 따라서, baseline GPU보다

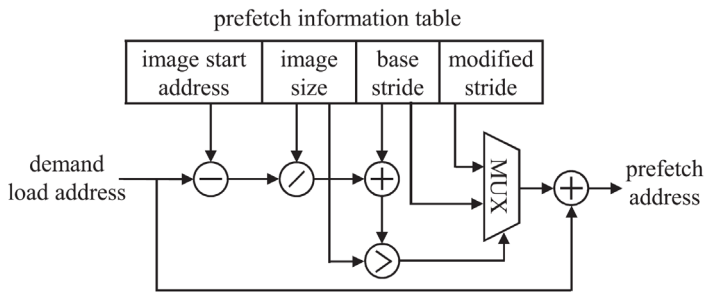

그림 5.9 Pattern-aware prefetcher의 구조

적은 warp 수로도 latency를 감출 수 있다. 그림 5.10은 prefetcher가 탑재된 환경에서 warp 수에 따른 실행 시간을 나타낸다. warp 수가 8개 이상일때부터 실행 시간의 감소가 saturate되는 것을 확인할 수 있다. 따라서, SM 당 warp의 수는 8개 (즉 256개 thread)로 설정되었다.

(3) Warp scheduler : warp 간 속도를 비슷하게 유지하기 위해, LRR warp scheduling이 사용되었다. 5.3.2장의 Guideline 2에서 설명된 문제는 명시적으로 동기화를 함으로써 대부분 해결되었으며, LRR warp scheduling을 통해 이를 보완한다.

(4) On-chip SRAM 크기 : Register file, L1 data cache, shared memory를 포함하는 on-chip SRAM의 크기는 warp 수를 줄임으로써 감소한다. 이는 warp 수가 감소함에 따라 warp를 switching하는데 필요한 register file의 크기가 줄어들며, thread 당 유효 cache size가 줄어들기 때문이다. 이는 leakage 전력 및 필요 면적을 감소시켜 logic die의 제약 조건을 맞추기 용이하게 한다. Register의 크기는 8192, L1 data cache의 크기는 8 KB로 설정되었다.

Off-chip L2 Cache. 제안하는 PIM GPU는 L2 cache를 사용하지 않는다. 첫번째 이유로, PIM GPU의 경우 off-chip DRMA과 on-chip SRAM 간 접근 시 소모되는 energy의 차이가 적기 때문이다. 반대로, baseline GPU의 경우 DRAM access 시 off-chip link를 통해야 하기에 필요한 에너지가 SRAM에 비해 10~100배 이상 크다 [83], [84]. 두번째로, 제안하는 CTA 할당 및 prefetcher를 통해 L1 cache의 hit rate을 100%에 가깝게

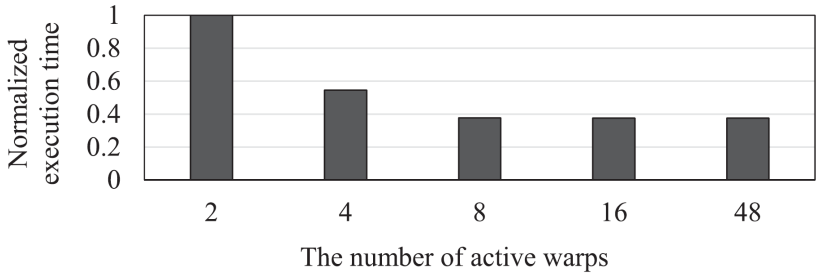

그림 5.10 Warp 수에 따른 실행 시간 변화

증가시킬 수 있기 때문이다. 따라서, L1 cache에서의 miss를 완화하기 위한 L2 cache의 필요성이 감소한다. 세번째로, PIM의 넓은 memory bandwidth 덕분에, L2 cache와 main memory 간의 bandwidth 차이가 기존 GPU에 비해 PIM GPU는 더 작다. 마지막으로, logic die의 면적 및 전력 제약 조건을 고려할 때, L2 cache의 넓은 면적은 logic die에의 탑재를 어렵게 한다. 종합하면, L2 cache의 제거를 통해 memory hierarchy를 단순화할 수 있으며, 전력 소모량 또한 감소시킬 수 있다.

SM의 수. PIM의 넓은 bandwidth를 활용하기 위해, 충분한 수의 SM이 PIM GPU에 탑재되어야 한다. 성능 및 면적, 에너지 소모량이 모두 고려되어야 하며, 이는 SM의 수를 변경해가며 실험되었다. 총 16개의 SM이 탑재되었으며, 이는 5.4 장에서 더 자세히 분석된다.

5.4 실험 결과 및 분석

5.4.1 실험 환경 구성

실험을 위해 cycle-accurate한 시뮬레이터인 GPGPU-Sim [45]이 수정되어 사용되었다. 4가지 기본 configuration은 다음과 같으며, 더 자세한 configuration은 표 5.2와 같다.

- **Base-8**: Baseline PIM GPU로 L2 cache를 탑재하고 있으며 8개의 SM으로 구성됨. SM 구조는 일반적인 GPU와 동일함
- **Base-8-LT**: SM 수, cache 및 register 크기는 Base-8과 동일하지만, SM 당 thread 수가 256개로 감소
- **IP-8-Base**: baseline image processing PIM으로, 256개의 thread, 작은 크기의 SRAM을 가지며, 8개 SM으로 구성
- **IP-8**: 제안하는 image processing PIM으로, IP-8-Base에 더해 pattern-aware CTA allocation, prefetcher를 탑재

PIM의 에너지 소모량은 GPUWattch [46]와 CACTI [47]를 활용하여 측정되었다. PIM stack 당 256 GB/s의 bandwidth를 가지며, 3D stacked DRAM의 parameter는 [48]의 값을 활용하여 시뮬레이션 되었다. [48]에 따르면, 3D stacked DRAM의 datapath는 다음의 3가지로 분류할 수 있다. (1) row buffer에서 global sense amplifier (GSAs), (2) GSAs에서 DRAM I/Os, (3)

표 5.2 실험에 사용된 PIM GPU configuration

	Base-8	Base-8-LT	IP-Base-8
Compute unit	8 SMs	8 SMs	8, 16 SMs
SM parameters	1.4GHz, 32 SIMT width 1536 threads, 32768 registers	1.4GHz, 32 SIMT width 256 threads, 32768 registers	1.4GHz, 32 SIMT width 256 threads, 8192 registers
warp scheduler	GTO (Greedy Then Oldest)	GTO	LRR
L1 cache	32KB, 8-way	32KB, 8-way	8KB, 8-way
L2 cache	512KB, 8-way, 8banks		none
memory bandwidth	256GB/s		

DRAM과 GPU 사이의 I/O channel. 이때, PIM GPU가 3D stacked DRAM의 logic die에 위치하기에, PIM GPU에서 DRAM에 접근할 때 datapath (3)은 거치지 않는다. 이를 고려하여, PIM GPU에서의 DRAM 접근 시 에너지는 row activation 에너지와 경로 (1), (2)의 합인 3.59 pJ/bit로 계산되었다.

실험에 사용된 벤치마크들은 표 5.3과 같다. 8개의 대표적인 image processing 알고리즘이 사용되었다. Pattern-aware CTA 할당 방법의 성능 확인을 위해, 5.3.2 장과 같이 코드가 수정되었다. Conv2D, ConvSep, Denoising은 filtering 알고리즘으로 대표적인 computer vision 알고리즘이다. DownSample 및 Concat 또한 널리 사용된다. Jacobi2D는 stencil 알고리즘으로 scientific computing에서 자주 사용되며, Dct8x8은 JPEG, H.264, HEVC 등에서 사용되는 핵심 모듈이다. MatMul은 deep learning 및 transformation 등에서 흔히 사용된다. 벤치마크들은 특성에 따라 아래와 같이 분류되었다.

- **Type 1**: Memory intensive하며, SM 간 data 공유가 많은 경우 – Conv2D, Jacobi2D
- **Type 2**: Memory intensive하며, SM 간 data 공유가 적은 경우 – ConvSep, DownSample, Concat, Dct8x8
- **Type 3**: Computation intensive한 경우 – Denoising, MatMul

5.4.2 In-Depth Analysis

해당 장에서는 prefetching (PF)와 pattern-aware CTA 할당 (PCAP) 등 제안된 각 방법을 심층 분석한다. 표 5.4의 15가지 configuration이 실험되었다. 제안하는 방법들의 영향을 분석하기 위해, PF를 program counter 기반의 per warp stride prefetching (PWS)로 [77], [78] 교체한 Base-8-PWS-PACA, Base-8-LT-PWS-PACA, IP-8-Base-PWS-PACA가 실험되었다.

그림 5.11은 다양한 configuration들의 Base-8의 값으로 normalize된

표 5.3 실험에 사용된 benchmark 목록

Benchmark	Function	Input	Type
Conv2D	2D convolution	2048×2048	1
Jacobi2D	stencil operation	2048×2048	1
ConvSep	seperable 1D convolution	1024×1024	2
DownSample	average downsampling	2048×2048	2
Concat	multiple image concatenation	Four 2048×2048	2
Dct8×8	discrete cosine transform operates on 8×8 blocks	1024×1024	2
Denoising	denoising using KNN method	2048×2048	3
MatMul	multiplication of two matrices	Two 256×256	3

표 5.4 실험에 사용된 GPU configuration

Name	Configuration
Base-8	Baseline NDP
Base-8-PF	Base-8 with proposed prefetching
Base-8-PACA	Base-8 with pattern-aware CTA allocation
Base-8-PWS-PACA	Base-8 with per warp stride prefetching and pattern-aware CTA allocation
Base-8-PF-PACA	Base-8 with proposed prefetching and pattern-aware CTA allocation
Base-8-LT	Baseline NDP with a lower number of threads
Base-8-LT-PF	Base-8-LT with proposed prefetching
Base-8-LT-PACA	Base-8-LT with pattern-aware CTA allocation
Base-8-LT-PWS-PACA	Base-8-LT with per warp stride prefetching and pattern-aware CTA allocation
Base-8-LT-PF-PACA	Base-8-LT with proposed prefetching and pattern-aware CTA allocation
IP-8-Base	Baseline image processing NDP
IP-8-Base-PF	IP-8-Base with proposed prefetching
IP-8-Base-PACA	IP-8-Base with pattern-aware CTA allocation
IP-8-Base-PWS-PACA	IP-8-Base with per warp stride prefetching and pattern-aware CTA allocation
IP-8	IP-8-Base with proposed prefetching and pattern-aware CTA allocation

속도 결과를 나타낸다. 먼저, Base-8의 경우, PACA 단일로 적용된 경우 성능이 하락하였으며, prefetching이 적용된 경우 PACA 적용 여부와 관계없이 성능이 향상되었다. Base-8-PF는 Base-8 대비 1.02배의 속도 향상을 보였다. PACA가 적용되지 않은 경우 prefetch의 정확도가 낮아 성능 향상 정도가 높지는 않았다. Prefetch 정확도가 낮았음에도 불구하고, Base-8은 충분한 수의 thread를 가져 부정확한 prefetch로 인한 부작용이 드러나지 않았다. Base-8-PACA의 경우, Base-8 대비 0.55배의 속도 향상을 보였다. 이는 resource 활용도가 감소하였기 때문으로, Base-8의 thread

수가 1536개인 반면, PACA는 SM 당 256개 thread로 구성된 한 개의 CTA만을 할당하기 때문이다. Base-8-PF-PACA는 Base-8 대비 1.00배의 속도 향상만을 보였으며, 이는 IP-8 대비 0.85배의 속도이다. Prefetching과 PACA가 모두 적용되었음에도 불구하고, L2 cache로 인해 성능 향상이 없었다. 이는 PACA가 적용되어 L2 cache가 필요하지 않지만, 여전히 존재하기에 L2 cache에서의 혼잡이 발생하였기 때문이다. 이는 prefetch를 지연시켜 성능에 악영향을 미쳤다. 특히, Type 3 벤치마크에 대해 Base-8-PF-PACA는 Base-8 대비 20%의 성능 하락을 보였다. Type 3 벤치마크는 computation intensive하며 SM 간 data 공유가 많아 L2 cache 덕에 prefetch의 효과가 낮았으며, thread 활용도의 감소로 성능이 하락하였다.

두번째로, Base-8-LT의 경우 prefetching과 PACA가 따로 적용된 경우 모두 성능이 향상되었으며, 함께 적용된 경우 높은 성능 향상을 보였다. Base-8-LT-PF는 Base-8-LT 대비 1.12배의 성능을 보였다. SM 당 1536개의 thread를 가지는 Base-8과 달리 Base-8-LT는 256개의 thread만을 가지기에 prefetching을 통한 data load stall의 감소가 성능에 큰 영향을 미쳤다. 이에 더해, thread 당 실질적인 cache 크기가 Base-8보다 더 크기에, 부정확한 prefetching의 부작용이 Base-8보다 덜하였다. Base-8-LT-PACA는 Base-8-LT 대비 1.05배의 속도를 보였다. Base-8-PACA와 달리 Base-8-LT-PACA는 256개의 thread만을 가지기에 PACA를 적용해도 thread 활용도가 감소하지 않았다. 그러나, Base-8-LT에 L2 cache가 존재하기에 PACA로 인한 성능 향상 정도가 Base-8-LT-PF의 그것보다 낮았다. Base-8-LT-PF-PACA는 Base-8-LT 대비 1.91배의 속도 향상을 보였으며, Base-8보다 더 적은 thread 수를 가짐에도 Base-8과 동일한 성능을 보였다. 이는 제안하는 방법이 적은 thread 수의 환경을 효율적으로 활용함을 의미한다. Base-8-PF-PACA의 경우와 동일하게, L2 cache의 존재로 인해 IP-8 대비 성능은 0.85배로 IP-8보다 낮은 성능을 보였다.

세번째로, IP-8-Base의 경우 Base-8-LT의 경우와 동일하게

prefetching과 PACA가 따로 적용되어도 성능이 향상되었으며, IP-8의 성능이 모든 configuration 중 제일 높았다. IP-8-Base-PF는 IP-8-Base 대비 1.01배의 성능을 보였다. Base-8-LT에 PF가 적용되었을 때 1.12배의 성능 향상이 있었던 것에 비하면 향상 정도가 작았는데, 이는 IP-8-Base가 Base-8-LT보다 작은 cache를 가져 prefetch가 부정확했을 때 부작용이 컸기 때문이다. 그러나, ConvSep의 경우 PF를 적용하여 L1 cache miss rate이 98.3%에서 39.4%로 급감하였기에, Base-8 대비 1.36배의 속도 향상을 보였다. IP-8-Base-PACA는 IP-8-Base 대비 1.13배의 성능 향상을 보였다. 이는 PACA가 Base-8-LT에 적용된 경우보다 높은 성능 향상으로, 이는 IP-8-Base에 L2 cache가 없어 PACA로 인한 성능 향상이 온전히 적용되었기 때문이다. IP-8-Base에 PF와 PACA가 모두 적용된 IP-8의 경우, IP-8-Base와 Base-8 대비 각각 2.07배, 1.17배의 성능 향상을 보여 configuration들 중 제일 높은 성능을 보였다. 정리하면, PF와 PACA가 따로 적용되었을 때의 성능 향상은 그리 높지 않지만, 함께 적용된 경우 성능 향상은 높은 것을 확인할 수 있다.

마지막으로, 모든 벤치마크에서 제안한 PF가 PWS보다 높은 성능을 보였다. Base-8-PF-PACA, Base-8-LT-PF-PACA, IP-8은 Base-8-PWS-PACA, Base-8-LT-PWS-PACA, IP-8-Base-PWS-PACA 대비 각각 1.07배, 1.07배, 1.16배 높은 성능 향상을 보였다. 이는 PWS가 image 경계에서의 stride를 제대로 계산하지 못한 반면, 제안하는 PF는 경계에서의 stride를 정확하게 계산할 수 있었기 때문이다.

5.4.3 기존 연구와의 성능 비교

본 연구와의 비교를 위해, BCS [76]와 [17]이 실험되었다. Base-8과 Base-8-LT에 적용되어 실험되었으며, [17]의 경우 직접적으로 비교하는

대신, perfect L2 cache (PL) 환경으로 실험하였다. [17]은 L2 cache로의 prefetching 방법을 제안하였으며, PL 환경을 사용하는 것은 100%의 prefetch 정확도와 coverage를 가지기에 [17]의 상한 성능과 비교한 것과 동일하다. IP-8은 Base-8보다 작은 면적을 가지기에 더 많은 SM을 탑재할 수 있다. 따라서, 16개의 SM을 가지는 IP-16 또한 함께 실험되었다. 정리하면, 다음의 8가지 configuration이 실험되었다: Base-8, Base-8에 BCS를 적용한 것 (Base-8-BCS), Base-8에 perfect L2를 적용한 것 (Base-8-PL), Base-8-LT에 BCS를 적용한 것 (Base-8-LT-BCS), Base-8-LT에 perfect L2를 적용한 것 (Base-8-LT-PL), IP-8, IP-16. IP-8과 IP-16을 함께 지칭하는 경우 IP-PIM으로 표기하였다.

그림 5.12는 Base-8로 normalize 된 각 configuration의 성능을 보여준다. 4가지 현상이 다음과 같이 설명된다. 먼저, 제안하는 IP-8이 Base-8, Base-8-BCS, Base-8-PL보다 1.17배 (최대 1.89배), 1.13배 (최대 1.93배), 1.02배 (최대 1.25배) 높은 성능을 보였다. Type 1 벤치마크에 대해, IP-8은 Base-8, Base-8-BCS, Base-8-PL 대비 1.21배, 1.08배, 1.07배의 성능을 보였다. Base-8-BCS는 Base-8-PL과 비슷한 성능을 보였다. 이는 Type 1 벤치마크의 경우 높은 inter-CTA locality를 가지기에 BCS의 효과가 좋았기 때문이다. Type 2 벤치마크의 경우, IP-8은 Base-8과 Base-8-BCS 대비 1.29배, 1.27배의 성능을 보였지만 Base-8-PL 대비해서는 1.05배의 성능 향상만을 보였다. ConvSep가 제일 높은 속도 향상을 보였다. 이는 기본 ConvSep 코드는 shared memory를 사용하기에, Guideline 4와 같이 shared memory 대신 cache를 사용함으로써 instruction 수를 줄일 수 있었기 때문이다. Type 2 벤치마크는 SM 간 data 공유가 적어 BCS와 같이 inter-CTA locality를 활용하는 방법이 효과적이지 못했다. 반면, perfect L2 cache는 memory latency를 감소시키기에, memory intensive한 Type 2 벤치마크의 성능을 효과적으로 향상시켰다. Type 3 벤치마크의 경우, IP-8은 Base-8, Base-8-BCS, Base-8-PL 대비 0.93배, 0.95배, 0.92배의 성능을

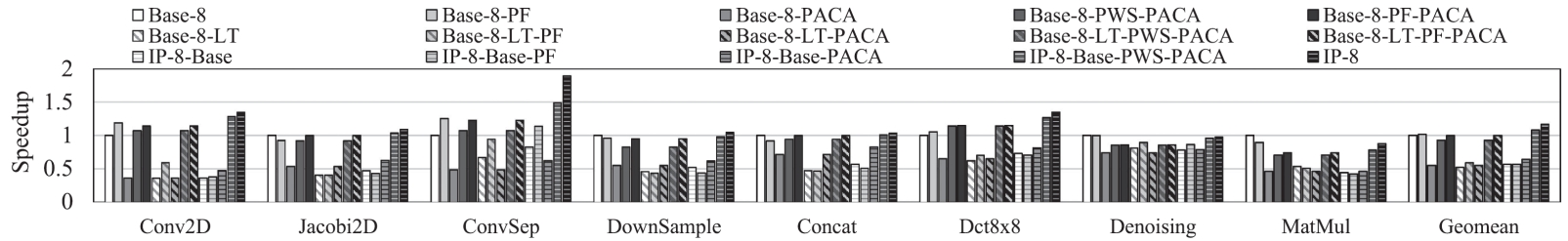

그림 5.11 제안한 방법들을 각각의 baseline configuration에 적용했을 때 Base-8 대비 속도 향상 정도

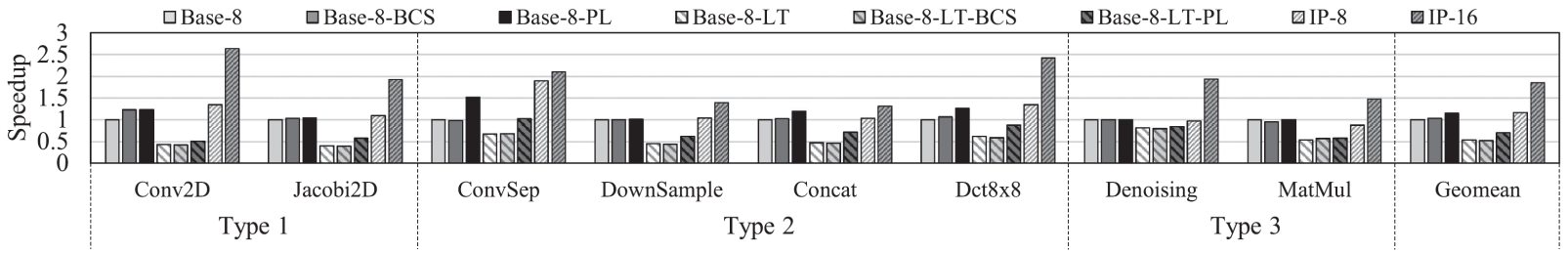

그림 5.12 Base-8, Base-8-BCS, Base-8-PL, Base-8-LT, Base-8-LT-BCS, Base-8-LT-PL, IP-8, IP-16의 Base-8 대비 속도 향상 정도

보였다. Type 3 벤치마크들은 computationally intensive하기에 IP-8의 memory traffic 감소 이득을 보기 어려웠지만, 그럼에도 불구하고 8개의 warp만을 가지는 IP-8이 Base-8 기반과 비슷한 성능을 보인 것은 인상적이다.

두번째로, BCS와 PL이 LT 환경에 적용된 경우, IP-8의 경우보다 성능 향상이 상당히 낮았다. 이는 제안하는 방법인 thread 수가 적은 상황에 기존 방법들보다 더 효율적임을 의미한다. IP-8은 Base-8-LT 대비 2.18배 (최대 3.07배)의 성능을 보인 반면, Base-8-LT-BCS는 Base-8-LT 대비 0.99배 (최대 1.01배)의 속도 향상만을 보였다. Thread의 수가 감소한 경우, prefetching 없이는 latency를 감추기 어렵기 때문에, 모든 벤치마크에서 BCS는 성능이 좋지 않았다. Base-8-LT-PL은 Base-8-LT 대비 1.3배 (최대 1.52배)의 성능 향상을 보였다. Perfect L2 cache는 memory latency를 줄여 latency hiding을 더 쉽게 하기에 thread 수가 적은 Base-8-LT에서 BCS 보다 효과적이었다. 그러나 여전히 L1 cache로 prefetch하지 않기에 L2 cache로의 접근 시간으로 인해 성능 향상은 제한적이었다.

세번째로, IP-16은 다른 모든 configuration들보다 높은 성능을 보였다. IP-16은 Base-8과 IP-8 대비 1.85배, 1.58배의 성능을 보였다. Type 1 벤치마크의 경우 IP-16은 IP-8 대비 1.86배의 거의 2배에 가까운 성능 향상을 보였다. Type 1 벤치마크의 경우 SM 간 data 공유가 많아 제안하는 방법으로 memory traffic을 효과적으로 줄일 수 있었기 때문에, IP-16 또한 bandwidth를 효율적으로 사용할 수 있었기 때문이다. 그러나, Type 2 벤치마크의 경우 inter-CTA locality를 활용해도 memory traffic을 충분히 감소시킬 수 없었기에, SM 수가 2배로 증가하였음에도 불구하고 IP-16은 IP-8 대비 1.36배의 속도 향상만을 달성하였다. Type 3 벤치마크의 경우 computation에 의해 bottleneck 되었기에, IP-16은 IP-8 대비 1.83배의 성능 향상을 보여 증가한 SM 수에 가까운 속도 향상을 달성하였다.

마지막으로, [17]가 실제로는 perfect L2 cache의 0.33배의 속도 향상만을 보인 반면, IP-8은 Base-8-PL과 Base-8-LT-PL 대비 1.02배, 1.67배

속도가 향상되었다. 이에 더해, IP-16의 경우 Base-8-PL과 Base-8-LT-PL 대비 1.61배, 2.64배의 속도 향상을 보였다. 따라서, 제안하는 방법이 [17] 대비 뛰어난 성능을 보이며, 감소한 면적을 고려하면 더 많은 성능 차이가 나는 것을 확인할 수 있다.

5.4.4 Cache Miss Rate 및 Memory Traffic

그림 5.13은 Base-8과 IP-8의 L1 cache miss rate을 나타낸다. IP-8은 Base-8 대비 77.5% 감소한 miss rate을 보였다. 또한, 제안하는 prefetcher의 정확도는 98.4%로 불필요한 prefetch가 거의 없었다. 그림 5.14는 Base-8의 SM과 L2 cache 간의 memory traffic (SM-L2 traffic)에 normalize된 IP-8 및 Base-8의 SM과 L2 cache 간의 memory traffic (SM-L2 traffic)과 SM과 main memory 간의 memory traffic (SM-Mem traffic)을 나타낸다. Base-8의 경우, SM-Mem traffic은 L2 cache와 main memory 간의 traffic과 동일하며, IP-8의 경우 L2 cache가 존재하지 않기에 SM-L2 traffic은 표기되지 않았고, SM-Mem traffic과 동일하게 사용되었다. IP-8은 SM-L2 traffic을 Base-8 대비 32.4% 감소시켰다. Type 1 벤치마크에 대해, IP-8은 SM-L2 traffic을 Base-8 대비 61.4% 감소시켰다. 상술하였듯이, Type 1 벤치마크는 SM 간 data 공유가 많기에 제안하는 CTA 할당 방법이 효과적으로 작동하였다. Type 2 벤치마크의 경우, IP-8은 SM-L2 traffic을 Base-8 대비 4.3% 감소시켰다. Type 2 벤치마크들은 SM 간 data 공유가 적어, CTA 할당으로 인한 효과 또한 제한적이었다. SM-Mem traffic의 경우, IP-8은 Base-8 대비 18% 더 많은 traffic을 보였다. Base-8의 L2 cache는 SM들의 main memory로의 불필요한 중복 접근을 줄인다. IP-8은 이를 제안하는 CTA 할당 방법을 통해 방지하지만, 각 SM에 할당된 영역의 경계에서의 중복 접근은 이를 통해 방지하지 못하기 때문이다. MatMul의 경우, input matrices에서 동일

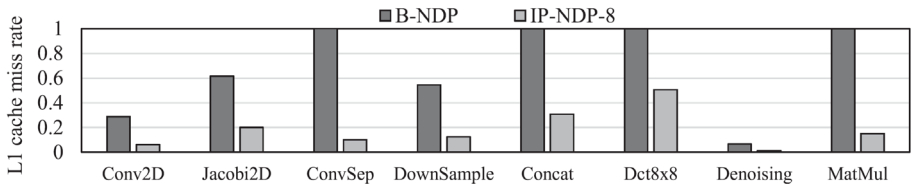

그림 5.13 Base-8과 IP-8의 L1 cache miss rate

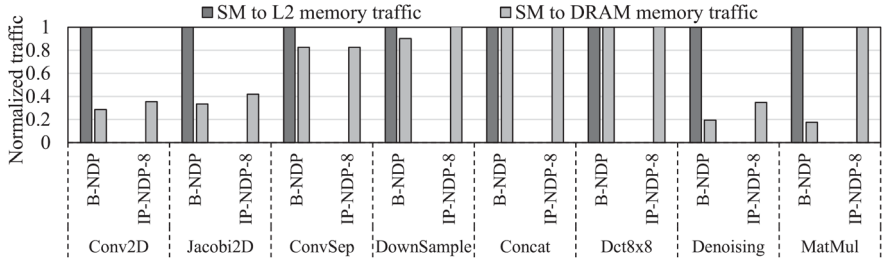

그림 5.14 Base-8의 SM-L2 traffic 값으로 normalize 된 Base-8 및 IP-8의 memory traffic

행과 열이 반복되어 접근되기 때문에 SM 간 data 공유가 많다. 그러나, 이러한 data sharing이 coalescing되어 접근하지 않기에 제안하는 방법으로 traffic이 감소하지 못하였다. 그럼에도 불구하고, IP-8의 SM-Mem traffic은 Base-8의 SM-L2 traffic과 SM-Mem traffic의 합보다 적었으며, 따라서 제안하는 CTA 할당 방법이 memory hierarchy 내에서의 전체 traffic을 효과적으로 감소시킴을 확인할 수 있다.

5.4.5 에너지 소모량 비교

그림 5.15는 6가지 configuration들의 Base-8에 normalize 된 에너지 소모량을 나타낸다. Perfect L2의 경우, main memory 접근 에너지를 구할 수 없기에 제외되었다. 모든 벤치마크에 대해 IP-PIM이 제일 적은 에너지를 소모함을 확인할 수 있다. IP-8은 Base-8과 Base-8-LT 대비 20.9%, 39.5% 적은 에너지를 소모하였다. 첫번째 이유로, IP-8의 빠른 수행 속도가 에너지 효율을 증가시켰다. 두번째로, IP-8의 SM은 Base-

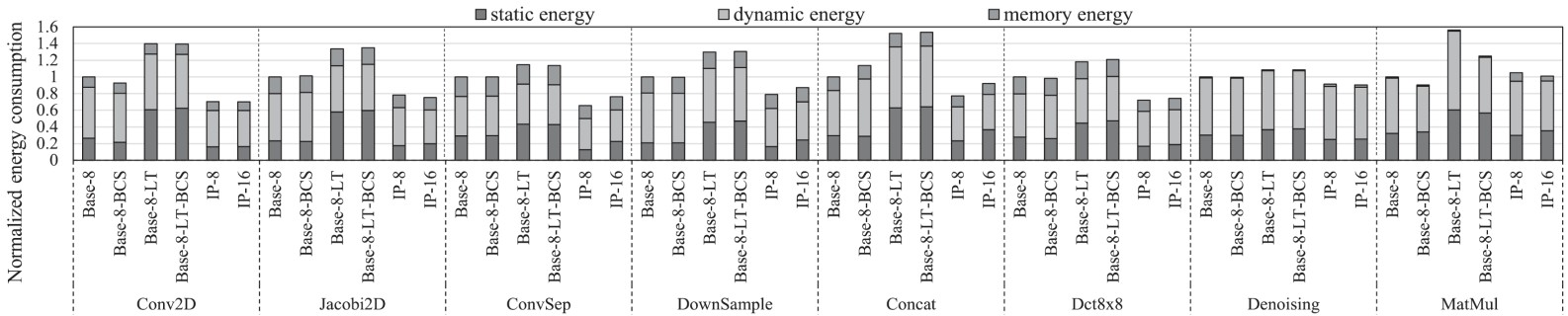

그림 5.15 Base-8의 값으로 normalize된 Base-8, Base-8-BCS, Base-8-LT, IP-8, IP-16의 에너지 소모량

8보다 작은 크기의 SRAM을 탑재하여 Base-8보다 더 낮은 static 전력을 소모하기 때문이다. IP-8의 SM의 static 전력은 Base-8 대비 75.2% 인 것을 확인하였다 (이는 5.4.6 장에서 더 자세히 설명된다). Denoising의 경우, IP-8은 Base-8 대비 0.97배의 속도를 보였지만, 더 낮은 static 전력 소모량 덕에 8.6% 적은 에너지를 소모하였다.

5.4.6 PIM의 면적 및 전력 소모량 분석

PIM의 면적은 40nm에서 SM 당 16 mm^2 의 면적을 가지는 fermi architecture [85]에 기반하여 추정되었다. Baseline PIM의 SM의 면적은 fermi SM을 22nm로 scaling하여 추정되었으며, IP-PIM의 L1 data cache 및 register file 등 SRAM의 크기는 CACTI 6.5를 사용하여 추정되었다. 더 자세한 내용은 [86]를 참조하였다. 계산된 값은 표 5.5에 나타나 있다. IP-PIM은 pattern-aware prefetcher를 탑재하였으며, 이 prefetcher는 mux 1개, 덧셈기 3개, 나눗셈기 1개, 비교기 1개로 구성되어 있다. Image의 시작 주소, base stride, modified stride를 저장하기 위해 각각 32, 20, 12, 12 bits가 필요하다. 이는 전체 SM 면적에 비하면 무시할 만하다. SRAM의 전력 소모량은 CACTI 6.5를 사용하여 측정되었으며, 나머지 요소의 측정에는 GPUWattch가 사용되었다. 측정된 값을 기반으로 계산한 결과, IP-PIM은 logic die의 면적 및 전력 제약 조건을 만족하는 것을 확인하였다.

표 5.5 SM 구성 요소들의 면적 및 전력 소모량

Components		Area (mm^2)	Static Power (mW)
baseline SM	Registers	1.56	68.5
	L1 cache	1.38	17.9
	Total	4.84	869
IP-NDP SM	Registers	0.39	17.1
	L1 cache	0.34	4.49
	Total	2.65	733
L2 cache		38.75	226

면적. HBM의 logic die 중 사용 가능한 면적은 50~60 mm^2 으로 알려져 있다 [87], [88]. Base-PIM의 경우, L2 cache의 면적을 고려하면 4개의 SM만이 logic die에 탑재될 수 있다. L2 cache를 제외하더라도, 12개의 SM만이 탑재될 수 있다. 반면 IP-PIM의 경우, SM 당 면적이 baseline의 55% 뿐이며 L2 cache가 없기에 최대 22개의 SM이 탑재될 수 있다. 따라서, IP-8과 IP-16 모두 면적 제약 조건을 만족한다.

전력. Logic die의 전력 제약 조건은 55W로 알려져 있다 [87]. 그림 5.16은 Base-8, IP-PIM의 전력 소모량을 나타내며, IP-8과 IP-16의 최대 전력 소모량은 30.7 및 50.1W로 logic die의 전력 제약 조건을 만족한다. Power gating 등의 방법은 추가적인 SM의 탑재를 가능케 한다.

마지막으로, 그림 5.17은 면적 당 성능을 나타낸 것으로, Base-8의 값에 normalize 되었다. IP-8과 IP-16의 면적 당 성능은 Base-8보다 4.27배, 3.38배 높다. 따라서, IP-PIM이 면적을 효율적으로 사용하여 높은 성능을 달성함을 확인할 수 있다.

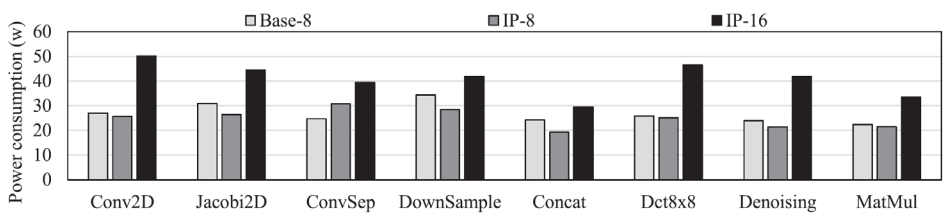

그림 5.16 Base-8, IP-8, IP-16의 전력 소모량

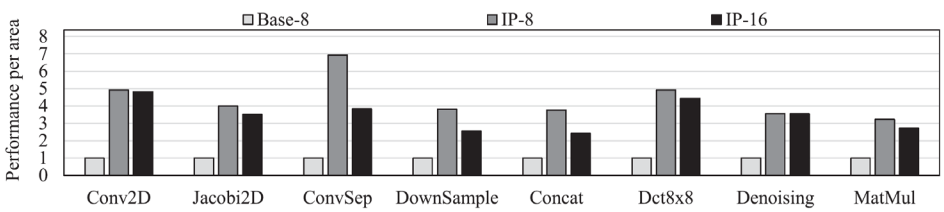

그림 5.17 Base-8, IP-8, IP-16의 면적 당 성능

5.5 본 장의 결론

본 장에서는 3D stacked memory의 logic die에 적합한 image processing 용 lightweight한 PIM GPU 구조와 함께 해당 구조를 활용하기 위한 software 기반의 CTA 할당 알고리즘이 제안되었다. 단순히 GPU를 3D stacked memory의 logic die에 탑재하는 것은 image processing의 성능을 향상시킬 수 있지만, 불필요한 memory access가 많이 발생하게 된다. 이에 더해, logic die의 면적 제한으로 인해 성능 향상이 적다. 이를 해결하기 위해, SM 간 data sharing이 최소화되도록 image processing의 data access pattern에 따라 CTA를 SM에 할당하는 기법이 제안되었다. 동시에, pattern aware CTA 할당을 활용하는 prefetcher를 통해 PIM 친화적인 lightweight한 GPU 구조가 제안되었다.

제 6 장 결 론

본 연구에서는 GPU와 PIM이 함께 탑재된 system을 활용하여 CNN을 가속시키는 방법에 대해 연구하였다. 첫째로, CNN의 convolution 연산을 가속하기 위해 approximate computing에 기반한 PIM solution인 ADC-PIM을 제안하였다. Approximate computing으로 인해 발생하는 memory bottleneck 및 GPU에서의 load imbalance를 완화하기 위해, PIM을 활용하여 memory 내부에서 data 간 유사도를 판별한 후, GPU에서는 대표 data에 대해서만 연산을 수행하게 된다. 이때, DRAM bank와 TSV 단을 모두 활용하여 divide-n-conquer 구조를 가지는 2-level PIM 구조를 제안하여 PIM에서 data 간 비교 시 일정 정확도 하에서 approximate되는 data의 비율을 높이면서 data 간 비교에 소요되는 시간을 최소화하였다. 또한, GPU로 비교 결과를 전송하기 전에 대표 data를 적절한 address에 할당하는 data reorganization을 통해 GPU에의 부담 없이 warp 간 load imbalance를 완화하였다. Memory bottleneck 및 load imbalance 문제를 해결함으로써, 제안하는 방법은 기존의 approximate computing 방법에 비해 성능을 향상시킬 수 있었다.

두번째로, CNN 학습 과정에서의 non-convolution 연산을 가속하기 위해, non-convolution 연산을 PIM으로 가속하여 host와 병렬적으로 수행하는 방법을 제안하였다. Batch normalization 등의 non-convolution은 output feature map의 channel 단위로 수행되기에, output feature map의 channel 단위로 convolution과 non-convolution을 pipelining 적으로 수행하도록 training flow를 변경한다. 이때, PIM에서의 병렬성을 확보하기 위해 channel들은 grouping되어 channel group 단위로 PIM 연산을 수행한다. 이때, 역전파 과정의 경우, weight의 update와 feature map gradient의 계산 과정에서의 non-convolution의 양이 불균형하기에, 두

과정 간의 non-convolution job을 균형있게 배분하여 수행 시간을 감소시킨다. 또한 Host와 PIM이 동시에 동작하는 상황에서 전체 수행 시간을 최소로 하기 위해 host와 PIM이 각 bank group 단위로 소유권을 가지고, 소유 시간 및 대기 중인 request의 수를 고려하여 소유권을 변경하는 memory scheduling 알고리즘을 제안한다. 실험 결과는 제안하는 방법이 기존의 memory intensive한 application을 우선시하는 scheduling 방법에 비해 성능을 향상시킬 수 있음을 보여준다.

마지막으로, PIM 환경에서의 CUDA 기반 전반적인 image processing application의 동작을 위해 logic die에 위치 가능한 PIM GPU 구조를 제안한다. 기존의 GPU 구조를 변경하지 않고 단순히 logic die에 탑재하는 대신, image processing의 data access pattern을 활용하여 각 SM에 data가 예측 가능하게 할당되게 함으로써 SM 간 data sharing을 최소화하고, 각 SM에 높은 정확도로 prefetching이 가능하도록 할 수 있다. 이를 통해 SM 면적 중 높은 비율을 차지하는 on-chip SRAM 및 L2 cache의 크기를 줄이고 warp 수를 감소시킨 간단한 구조의 SM을 사용하고, 더 많은 수의 SM을 PIM에 탑재하여 lightweight하면서 고성능의 PIM GPU 구조를 제안한다. 실험 결과는 제안하는 구조가 SM 수가 동일한 경우에도 기존과 비슷한 성능을 보이며, 더 많은 수의 SM을 탑재할 수 있는 것을 고려하면 기존보다 높은 성능을 보인다는 것을 나타낸다.

본 논문에서는 host로 GPU를 사용하는 system에서의 연구를 진행하였지만, 추후 host로 가속기를 사용하는 system에도 비슷한 방법을 적용하여 성능을 향상시킬 수 있을 것으로 기대 된다.

참고 문헌

[1] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, “Learning and transferring mid-level image representations using convolutional neural networks,” in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014, pp. 1717-1724.

[2] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), 2010, pp. 253-256.

[3] J.Choi, D. Chun, H. Kim, and H. J. Lee, "Gaussian yolov3: An accurate and fast object detector using localization uncertainty for autonomous driving," in Proceedings of IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 502-511.

[4] S. Li, W. Wen, Y. Wang, S. Han, Y. Chen, and H. Li, "An FPGA design framework for CNN sparsification and acceleration." in Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 28-28.

[5] J. Draper, J. Chane, M. Hall, C. Steele, T. Barret, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, “The architecture of the DIVA processing-in-memory chip,” in Proceedings of International Conference on Supercomputing (ICS), 2002, pp. 14–25.

[6] J. Ahn, S. Hong, S. Yoo, O.Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2015, pp. 105-117.

[7] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 457–468.

[8] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: The Terasys massively parallel PIM array,” Computer, vol. 28, no. 4, pp. 23–31, 1995.

[9] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “TOP-PIM: Throughput-oriented programmable processing in memory,” in Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2014, pp. 85–98.

[10] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, “Accelerating linked-list traversal through near-data processing,” in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2016, pp.

113–124.
[11] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2015, pp. 113–124.
[12] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, “Mcdram: Low latency and energy-efficient matrix computations in dram,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 11, pp. 2613–2622, 2018.
[13] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, “Newton: A dram-maker’s accelerator-in-memory (aim) architecture for machine learning,” in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 372–385.
[14] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, “Dracc: a dram based accelerator for accurate cnn inference,” in Proceedings of Design Automation Conference (DAC), 2018, pp. 1–6.
[15] Aga, Shaizeen, Nuwan Jayasena, and Mike Ignatowski. "Co-ML: a case for Co llaborative ML acceleration using near-data processing," in Proceedings of International Symposium on Memory Systems (MEMSYS), 2019, pp. 506-517.
[16] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, “Processing-in-memory enabled graphics processors for 3D rendering,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 637–648.
[17] R. Panda, Y. Eckert, N. Jayasena, O. Kayiran, M. Boyer, and L. K. John, “Prefetching techniques for near-memory throughput processors,” in Proceedings of International Conference on Supercomputing (ICS), 2016, pp. 1-14.
[18] C. D. Kersey, H. Kim, and S. Yalamanchili. "Lightweight SIMT core designs for intelligent 3D stacked DRAM." in Proceedings of International Symposium on Memory Systems (MEMSYS), 2017, pp.49-59
[19] S. Lee, S. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O, Seongil, A. Lyer, D. Wang, K. Sohn, N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product." in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2021, pp. 43-56
[20] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, “FBRAM: A new form of memory optimized for 3D graphics,” in Proceedings of Computer Graphics and Interactive Techniques (SIGGRAPH), 1994, pp. 167–174.
[21] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “TETRIS: Scalable and efficient neural network acceleration with 3D memory,” in Proceedings of International Conference on Architectural Support for Programming Languages and

Operating Systems (ASPLOS), 2017, pp. 751-764.
[22] N. G. GTX, “980: Featuring maxwell, the most advanced GPU ever made,” White Paper, NVIDIA Corporation, 2014.
[23] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, “Locality-aware CTA clustering for modern GPUs,” ACM SIGARCH Computer Architecture News, vol. 45, pp. 297–311, 2017.
[24] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “Dadiannao: A machine-learning supercomputer,” in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014, pp. 609-622.
[25] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi, “A hybrid approximate computing approach for associative in-memory processors,” IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 8, no. 4, pp. 758–769, 2018.
[26] P. K. Krause and I. Polian, “Adaptive voltage over-scaling for resilient applications,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp. 1–6.
[27] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, “Scalable effort hardware design,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 9, pp. 2004–2016, 2014.
[28] A. Ghofrani, A. Rahimi, M. A. Lastras-Monta˜no, L. Benini, R. K. Gupta, and K.-T. Cheng, “Associative memristive memory for approximate computing in gpus,” IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 6, no. 2, pp. 222–234, 2016.
[29] C. Liu, J. Han, and F. Lombardi, “A low-power, high-performance approximate multiplier with configurable partial error recovery,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pp. 1–4.
[30] M. Imani, D. Peroni, and T. Rosing, “Cfpu: Configurable floating point multiplier for energy-efficient computing,” in Proceedings of Design Automation Conference (DAC), 2017, pp. 1–6.
[31] M. S. Ansari, H. Jiang, B. F. Cockburn, and J. Han, “Low-power approximate multipliers using encoded partial products and approximate compressors,” IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 8, no. 3, pp. 404–416, 2018.
[32] I. Qiqieh, R. Shafik, G. Tarawneh, D. Sokolov, S. Das, and A. Yakovlev, “Significance-driven logic compression for energy-efficient multiplier design,” IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 8, no. 3, pp. 417–430, 2018.
[33] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta, “Energy-efficient neural networks using approximate computation reuse,” in Proceedings of Design,

Automation & Test in Europe Conference & Exhibition (DATE), 2018, pp. 1223–1228.
[34] L. Mocerino, V. Tenace, and A. Calimera, “Energy-efficient convolutional neural networks via recurrent data reuse,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 848–853.
[35] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, “Looknn: Neural network with no multiplication,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1775–1780.
[36] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing, “Efficient neural network acceleration on gpgpu using content addressable memory,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1026–1031.
[37] A. Yasoubi, R. Hojabr, and M. Modarressi, “Power-efficient accelerator design for neural networks using computation reuse,” IEEE Computer Architecture Letters, vol. 16, no. 1, pp. 72–75, 2016.
[38] D. Kim, J. Ahn, and S. Yoo, “Zena: Zero-aware neural network accelerator,” IEEE Design & Test, vol. 35, no. 1, pp. 39–46, 2017.
[39] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, “7.7 lnpu: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16,” in Proceedings of International Solid-State Circuits Conference (ISSCC), 2019, pp. 142–144.
[40] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory,” ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 380–392, 2016
[41] J. Lee, J. H. Ahn, and K. Choi, “Buffered compares: Excavating the hidden parallelism inside dram architectures with lightweight logic,” in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016, pp. 1243–1248.
[42] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” in Proceedings of ACM symposium on Theory of computing (STOC), 2002, pp. 380–388.
[43] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian, “Super-bit locality sensitive hashing,” in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2012, pp. 108–116.
[44] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. Healy, “Attache: Towards ideal memory compression by mitigating metadata bandwidth overheads,” in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 326–338.
[45] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing

cuda workloads using a detailed gpu simulator," in Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009, pp. 163–174.
[46] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," ACM SIGARCH Computer Architecture News, vol. 41, no. 3, pp. 487–498, 2013.
[47] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, no. 2, pp. 1–25, 2017.
[48] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017, pp. 41–54.
[49] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2017, [Online].
[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009, pp. 248–255.
[51] H. H. Saleh and E. E. Swartzlander, "A floating-point fused dot-product unit," in Proceedings of International Conference on Computer Design (ICCD), 2008, pp. 427–431.
[52] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 101-108.
[53] K. HE, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, p. 770-778.
[54] W. Jung, D. Jung, B. Kim, S. Lee, W. Rhee, and J. H. Ahn, "Restructuring batch normalization to accelerate CNN training," in Proceedings of Machine Learning and Systems 1 (MLSys), 2019, pp.14-26.
[55] S. R. Bulo, L. Porzi, and P. Kontschieder. "In-place activated batchnorm for memory-optimized training of dnns," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 5639-5647.
[56] S. Lym,A. Behroozi, W. Wen, G. Li, Y. Kwon, and M. Erez, "Mini-batch serialization: Cnn training with inter-layer data reuse," in Proceedings of Machine Learning and Systems 1 (MLSys), 2019, pp. 264-275.

[57] A. Pattnaik X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, “Scheduling techniques for GPU architectures with processing-in-memory capabilities,” in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2016, pp. 31–44.
[58] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. “NDA: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 283–295.
[59] A. Nag, and R. Balasubramonian. "OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations." in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2021, pp. 298-310.
[60] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems,” ACM SIGARCH Computer Architecture News, vol. 44, pp. 204–216, 2016.
[61] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near data acceleration with concurrent host access," in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2020, pp. 818-831.
[62] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J, Park, M. Wi, S. Lee, and J. H. Ahn, "MViD: Sparse matrix-vector multiplication in mobile dram for accelerating recurrent neural networks." IEEE Transactions on Computers, vol. 60, no. 7, pp. 955-967, 2020.
[63] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: exploiting differences in memory access behavior," in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010, pp. 65-76.
[64] R. Ausavarungnirun, K. K. W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2012, pp. 416-427.
[65] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution." in Proceedings of International Symposium on Memory Systems (MEMSYS), 2015, pp. 223-234.
[66] Nvidia Tesla V100 GPU Architecture, The World’s Most Advanced Data Center GPU. NVIDIA Corporation, 2017.
[67] Jia, Zhe, et al. "Dissecting the NVIDIA volta GPU architecture via microbenchmarking." arXiv preprint arXiv:1804.06826 (2018).

[68] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019, pp. 79-92.
[69] K. Simonyan, and A. Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
[70] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp 4700-4708.
[71] H. Wang, R. Singh, M. J. Schulte, and N. S. Kim, "Memory scheduling towards high-throughput cooperative heterogeneous computing," in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2014, pp.331-341.
[72] J. Standard, "High bandwidth memory (HBM) DRAM," JESD235, 2013.
[73] J. T. Pawlowski, "Hybrid memory cube (HMC)," in Proceedings of IEEE Hot Chips Symposium (HCS), 2011, pp. 1–24.
[74] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013, pp. 157–166.
[75] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 260–271.
[76] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2013, pp. 332-343.
[77] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Manythread aware prefetching mechanisms for GPGPU applications," in Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010, pp. 213–224.
[78] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013, pp. 73–82.
[79] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," ACM SIGPLAN Notices, vol. 48, no. 4, pp. 395–406, 2013.
[80] H. Jeon, G. Koo, and M. Annavaram, "CTA-aware prefetching for GPGPU,"

Univ. Southern California, Los Angeles, CA, USA, Comput. Eng. Tech. Rep. CENG-2014–08, 2014.
[81] NVIDIA, “NVIDIA CUDA SDK 4.2,” 2011.
[82] B. Dally, “Power, programmability, and granularity: The challenges of ExaScale computing,” in Proceedings of IEEE International Test Conference, 2011, pp. 12–12.
[83] M. Horowitz, “1.1 Computing’s energy problem (and what we can do about it),” in Proceedings of International Solid-State Circuits Conference (ISSCC), 2014, pp. 10–14.
[84] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, “Energy-efficient mechanisms for managing thread context in throughput processors,” in Proceedings of IEEE International Symposium on Computer Architecture (ISCA), 2011, pp. 235–246.
[85] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, “Power modeling for GPU architectures using McPAT,” ACM Transactions on Design Automation of Electronic Systems, vol. 19, no. 3, 2014, Art. no. 26.
[86] Y. Eckert, N. Jayasena, and G. H. Loh, “Thermal feasibility of diestacked processing in memory,” 2014.
[87] J. Kim and Y. Kim, “HBM: Memory solution for bandwidth-hungry processors,” in Proceedings of IEEE Hot Chips Symposium (HCS), 2014, pp. 1–24.

Abstract

Acceleration of CNN Computation on a PIM-enabled GPU system

Jungwoo Choi

Electrical and Computer Engineering

The Graduate School

Seoul National University

Recently, convolutional neural networks (CNN) have been widely used in image processing and computer vision. CNNs are composed of various layers such as computation-intensive convolutional layer and memory-intensive fully connected layer, batch normalization layer, and activation layer. GPUs are often used to accelerate the CNN, but performance is limited by high computational costs and memory usage of the convolution. Also, increasing demand for high resolution image applications increases the burden of data movement between GPU and memory. By performing computations on the memory, processing-in-memory (PIM) is expected to mitigate the overhead caused by data transfer. Therefore, a system that uses a PIM is promising for processing CNNs.

First, prior studies exploited approximate computing to reduce the computational costs. However, they only reduced the amount of the computation, thereby its performance is bottlenecked by the memory bandwidth due to an increased memory intensity. In addition, load imbalance between warps caused by approximation also inhibits the performance improvement. This dissertation proposes a PIM solution that reduces the amount of data movement and computation through the Approximate Data Comparison (ADC-PIM). Instead of determining the value similarity on the GPU, the ADC-PIM located on memory compares the similarity and transfers only the selected data to the GPU. The GPU performs convolution on the representative data transferred from the ADC-PIM, and reuses the calculated results based on the similarity information. To reduce the increase in memory latency due to the data comparison, a two-level PIM architecture that exploits both the

DRAM bank and TSV stage is proposed. To ease the load balancing on the GPU, the ADC-PIM reorganizes data by assigning the representative data to proposer addresses that are computed based on the comparison result.

Second, to solve the memory bottleneck caused by the high memory usage, non-convolutional layers are accelerated with PIM. Previous studies also accelerated the non-convolutional layers by PIM, but there was a limit to performance improvement because they simply assumed a situation in which the GPU and PIM operate sequentially. The proposed method accelerates the CNN training with a pipelined execution of GPU and PIM, focusing on the fact that the non-convolution operation is performed in units of channels of the output feature map. PIM performs non-convolutional operations on the output feature map where the GPU has completed the convolution operation. To balance the jobs between convolution and non-convolution in weight update and feature map gradient calculation that occur in the back propagation process, non-convolution job is properly distributed to each process. In addition, a memory scheduling algorithm based on bank ownership between the host and PIM is proposed to minimize the overall execution time in a situation where the host and PIM simultaneously access memory.

Finally, a GPU-based PIM architecture for image processing application is proposed. Programmable GPU-based PIM is attractive because it enables the utilization of well-crafted software development kits (SDKs) such as CUDA and openCL. However, the large capacity of on-chip SRAM of GPU makes it difficult to mount a sufficient number of computing units in logic die. This dissertation proposes a GPU-based PIM architecture and well-matched optimization strategies considering both the characteristics of image applications and logic die constraints. Data allocation to the computing unit is addressed to maintain the data locality and data access pattern. By applying a prefetcher that leverages the pattern-aware data allocation, the number of active warps and the on-chip SRAM size of the PIM are significantly reduced. This enables the logic die constraints to be satisfied and a greater number of computing units to be integrated on a logic die.

Keywords : processing-in-memory, convolutional neural networks, approximate computing, GPU, image processing
Student Number : 2016-20982