creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

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

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

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

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

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

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

Disclaimer

dCollection

공학석사 학위논문

Dynamic Grain Size Control for PIM Command Scheduling

PIM 명령어 스케줄링을 위한

동적 grain 크기 제어

2021 년 08 월

서울대학교 대학원

전기 정보 공학부

이 상 현

Dynamic Grain Size Control for PIM Command Scheduling

지도 교수 이 혁 재

이 논문을 공학석사 학위논문으로 제출함
2021 년 08 월

서울대학교 대학원
전기 정보 공학부
이 상 현

이상현의 공학석사 학위논문을 인준함
2021 년 08월

위 원 장 _____ 김 태 환 _____

부위원장 _____ 이 혁 재 _____

위 원 _____ 김 장 우 _____

초 록

최근 Deep learning network는 성능을 증가시키기 위해 layer의 수와 크기가 증가하고 있다. 특히 NLP(Natural Language Processing)나 recommendation deep learning network는 layer의 크기가 크고 data reuse의 비율이 적어 memory bandwidth에서 bottleneck이 발생하고 있다.

위 문제를 해결하기 위해 최근 대두되고 있는 PIM (Processing In Memory)는 메모리의 내부 혹은 근처에서 연산을 수행하고, 이를 통해 deep learning이 가지고 있는 memory bandwidth bottleneck을 극복하고 있다. 이러한 PIM의 대표적인 예시로는 NEWTON memory가 있다. NEWTON memory는 데이터를 저장하는 main memory와 연산을 수행하는 PIM의 기능을 수행할 수 있다.

본 논문에서는 host processor에서 메모리에 요청하는 normal command와 PIM에서의 연산을 수행시키는 NEWTON command가 동시에 접근할 때, NEWTON memory의 구조와 특징을 고려하여 normal과 NEWTON command scheduling하는 새로운 PIM command scheduling policy를 소개한다. 본 연구에서는 NEWTON memory의 특성을 고려하지 않은 경우 생기는 문제점인 all bank activation overhead를 설명하고, 제안하는 PIM controller에서 MAC address table을 이용하여 normal command를 grain단위로 issue하는 dynamic command scheduling을 적용하였다.

실험을 통하여 dynamic command 방식은 기존의 FIFO(First In First Out)방식에 비해 SPEC CPU 2017의 execution time이 최대 28% 감소한 모습을 확인할 수 있었고, 이는 fine grain scheduling 방식보다

execution time이 감소한 것을 확인할 수 있었다.

주요어 : PIM, 메모리 request scheduling, memory controller
학 번 : 2019-23220

목 차

제 1 장 서 론 1
1.1 연구의 배경 및 내용 1
1.2 논문 구성 5

제 2 장 관련 연구 6
2.1 새로운 PIM 고제 제안 연구 6
2.2 PIM kernel off loading에 관한 연구 8
2.3 Concurrent memory access에 관한 연구 10

제 3 장 Background 12
3.1 NEWTON memory 구조 12
3.1.1 NEWTON memory overview 12
3.1.2 NEWTON memory data mapping 15
3.1.3 NEWTON command 16
3.2 All bank row activation overhead 18

제 4 장 제안 PIM Controller 22
4.1 PIM controller overview 22
4.2 Fine grain issue policy 26
4.3 MAC address table 29
4.4 Dynamic grain command scheduling 31

제 5 장 실험 결과 및 분석 35
5.1 Experiment methodology 35
5.2 실험 결과 38
5.2.1 RNN execution time 비교 39
5.2.2 Memory contention으로 인한 수행시간 증가 41
5.2.3 grain size에 따른 execution time 변화 42
5.2.4 Dynamic grain size 효과 42
5.2.5 Dynamic grain scheduling policy 효과 44
5.2.6 NEWTON dynamic grain scheduling policy 효과 46
5.2.7 다양한 dynamic grain scheduling policy 효과 47

제 6 장 결론 49

참고문헌 50

Abstract 53

표 목차

[표 3. 1] NEWTON command.....17
[표 3. 2] Matrix size에 따른 activation latency 비율.....20

[표 5. 1] 실험 system configuration.....35
[표 5. 2] 실험 DRAM timing parameters.....36
[표 5. 3] SPEC CPU 2017 benchmarks list.....37

그림 목차

[그림 3. 1] NEWTON architecture overview.....14
[그림 3. 2] Memory timeline without request intevene.....19
[그림 3. 3] Normal memory request로 인한
all bank activation overhead.....19
[그림 3. 4] Memory contention 발생.....21

[그림 4. 1] PIM controller overview.....24
[그림 4. 2] Large grain size issue policy.....28
[그림 4. 3] Small grain size issue policy.....28
[그림 4. 4] MAC Adress Tble 예시.....30
[그림 4. 5] Dynamic grain issue policy.....33
[그림 4. 6] NEWTON command FSM.....34

[그림 5. 1] NEWTON simulator overview.....36
[그림 5. 2] RNN execution time in CPU and GPU.....39
[그림 5. 3] GEMV execution time in CPU and GPU.....39
[그림 5. 4] Memory contention 영향.....41
[그림 5. 5] SPEC CPU slowdown and
activatio overhead ratio.....42
[그림 5. 6] Grain size에 따른 SPEC CPU 2017 수행시간.....43
[그림 5. 7] Dynamic grain issue policy를 적용했을 때
SPEC CPU 2017 수행시간 변화.....45
[그림 5. 8] NEWTON command execution time 변화.....47
[그림 5. 9] Large/small size가 바뀐 경우
execution time 변화.....48

제 1 장 서 론

1.1 연구의 배경 및 내용

최근 deep learning network는 성능을 증가시키기 위해 network에 쓰이는 layer의 수와 크기가 증가하고 있다. 특히 NLP(Natural Language Processing)[19]나 recommendation deep learning network[11]는 layer의 크기가 크고 data reuse의 비율이 적어 network를 수행하는 latency가 memory bandwidth에서 bottleneck이 발생하고 있다. 특히 이런 bottleneck을 발생시키는 부분은 matrix-vector multiplication 연산을 의미하는 GEMV(General Matrix Vector Multiplication)에서 가장 두드러진다. GEMV는 CNN(Convolution Neural Network)의 fully connected network 와 RNN(Recurrent Neural Network)에서 많이 쓰인다.

위 bottleneck 문제를 해결하기 위해 최근 대두되고 있는 PIM (Processing In Memory)는 메모리의 내부 혹은 근처에서 연산을 수행하고, 이를 통해 deep learning이 가지고 있는 memory bandwidth bottleneck을 극복하고 있다. PIM은 메모리 내부에서 연산을 수행하는 방식과 메모리 근처에서 연산을 수행하여 빠른 메모리 접근의 이점을 살린 NDA(Near Data Accelerator)로 구분된다. NDA는 기존의 컴퓨터 구조에 비해 메모리 접근 시간이 짧지만 메모리 bandwidth에 여전히 bottleneck이 존재한다. 따라서 최근에는 메모리 근처에서 연산을 수행하는 것이 아닌 메모리 내부에서 수행하는 PIM연구가 진행되고 있다.

이러한 PIM의 대표적인 메모리는 NEWTON memory[9]이다. NEWTON memory는 2020년 MICRO 학회에 발표된 PIM으로 메모리 내부에서 연산을 수행한다. 메모리 내부에서 연산을 수행함과 동시에 기

존의 메모리의 기능인 읽고 쓰는 명령어도 처리를 하기 위해 full core를 메모리 내부에 구현하지 않고 간단한 연산기인 multiplication과 adder만 메모리 내부에 구현하였다. NEWTON memory는 bank내에 있는 multiplication과 adder를 이용하여 matrix vector multiplication을 수행한다.

이러한 NEWTON memory를 비롯한 여러가지 PIM을 실제로 활용하기 위해 다양한 연구가 진행되었다. 대표적으로 함수 또는 커널을 PIM에서 연산을 수행시키는 off loading에 관한 연구와 함께 host processor와 PIM의 PE에서 접근할 때 data의 coherence를 해결하기 위한 data coherency, 그리고 다른 thread를 수행중인 host processor와 PIM이 동시에 메모리에 접근할 때 성능 하락을 최소한으로 하는 concurrent access memory 연구가 있다.

그 중에서 concurrent access memory에 관한 연구는 NEWTON memory를 비롯한 메모리 내부에서 연산을 수행하는 PIM이 기존의 normal DRAM 명령어와 함께 PIM에서 연산을 수행시키는 명령어도 받아들일 때의 성능에 관한 연구이다. 기존의 메모리 스케줄링 방식은 normal DRAM 명령어만을 가지고 스케줄링을 했고 대표적인 방식인 FR-FCFS(First Ready, First Come, First Serve)방식으로 메모리 컨트롤러를 통해 메모리 명령어를 제어하고 있다. 이 방식은 메모리의 bank마다 활성화된 row가 존재하고 활성화된 row에 있는 column에게만 접근(read or write)이 가능하다는 점을 활용한 메모리 스케줄링 방식이다. 기존의 DRAM command는 하나의 명령어를 통해 하나의 bank 안에 있는 하나의 column 혹은 burst length라고 불리는 길이 만큼의 데이터만 읽고 쓰게 된다. FR-FCFS 스케줄링 방식은 이러한 DRAM command의 특성을 이용하여 다양한 상황에서도 latency와 throughput을 위해 적용된 스케줄링 방식이다.

하지만 NEWTON memory는 기존의 DRAM command 외에도 NEWTON command가 추가로 메모리에 들어오게 된다. FR-FCFS는 앞서 말한대로 기존의 DRAM command의 특성만을 반영한 스케줄링 방식이다. 하지만 NEWTOPN command가 새롭게 추가됨으로서, NEWTON command의 특성과 NEWTON memory가 가지는 특성 또한 반영을 해야한다. 기본적으로 NEWTON command는 deep learning network를 위한 command이다. 따라서 NEWTON command나 normal command에 priority를 주어 스케줄링 하는 방식은 priority를 가지지 않는 command의 latency를 증가시킬 수 있다. 따라서 NEWTON command와 normal DRAM command 사이의 스케줄링은 어느 한쪽의 성능만을 고려해서 command를 선택할 수 없게 된다. 또한 NEWTON memory는 기존 DRAM memory가 가지지 않는 특성인 all bank row activation ovehread를 가지고 있다. 기존 DRAM command는 하나의 row만 activation을 시킨 것에 반해 NEWTON command 중 하나인 MAC command는 bank parallelism을 위해 all bank operation으로 동작을 수행하고, 이를 위해 MAC command 이전에 all bank row activation이 필수적이다. 이때 all bank row activation이 추가로 발생하는 것을 바로 all bank row activation overhead라고 부른다. 이처럼 NEWTON memory만이 가지고 있는 특성은 기존의 스케줄링 방식이나 다른 PIM을 위한 스케줄링 방식을 적용하였을 경우 normal command와 NEWTON command 모두의 성능 저하를 일으키게 된다.

따라서 본 논문에서는 NEWTON memory의 특성인 all bank row activation에 대하여 자세히 살펴보고 activation overhead가 어떤 상황에서 발생하는지 알아보려고 한다. 또한 이를 바탕으로 all bank activation overhead를 줄이기 위하여 먼저 fine grain scheduling policy를 제안한다. 이 방식은 normal command를 grain size만큼 모아

서 NEWTON memory에 issue를 하는 방식으로 grain size에 따라 성능의 차이를 보이게 된다. 이후에 본 논문에서 제안하는 방식인 dynamic grain scheduling policy를 제안한다. Dynamic grain scheduling policy는 먼저 제안한 fine grain scheduling policy에 더하여 grain size를 NEWTON command와 normal command의 상황에 따라 grain size를 변화시키는 방식이다. 이를 위하여 본 논문에서는 MAT(MAC Address Table)이라는 새로운 컨트롤러 내부의 모듈을 제안하고 이용하였다. 이 MAT을 이용하여 컨트롤러에서는 dynamic grain scheduling policy를 제안하였다.

NEWTON memory는 아직 제품으로 출시되지 않았기 때문에 실제로 NEWTON command를 NEWTON memory에 issue를 시킬 수 없다. 따라서 본 논문에서는 dynamic grain scheduling policy의 성능을 측정하기 위하여 host processor는 gem5[16] 시뮬레이션을 이용하여 host processor의 성능을 측정하였다. Host processor에서 수행을 하는 application은 SPEC CPU 2017의 21종으로 설정하였다. 이 SPEC CPU 2017 벤치마크의 실행시간을 측정함으로서 dynamic grain scheduling policy가 기존의 방식보다 얼마나 효율적으로 스케줄링을 하였는지를 측정하였다. NEWTON memory의 시뮬레이션을 위하여 본 논문에서는 DRAMsim3[17]를 이용하였다. 기존의 DRAMsim3는 normal DRAM command만을 시뮬레이션 할 수 있기 때문에 본 논문에서는 DRAMsim3를 수정하여 NEWTON command가 issue될 때, bank state와 활성화된 row가 바뀌도록 수정하였다. 또한 DRAMsim3에 있는 controller를 수정하여 본 논문에서 제안하고, 비교하려는 여러가지 policy를 구현하였다.

실험 결과 본 논문에서 제안하는 dynamic grain scheduling policy는 baseline policy인 FIFO(First In, First Out)방식 대비 최대 28%

SPEC CPU 2017의 실행시간이 감소하였다. 또한 dynamic grain scheduling policy는 NEWTON command의 실행시간이 최대 47%가 감소한 모습을 확인할 수 있었다. 이는 NEWTON command에 priority를 준 방식인 coarse grain scheduling policy대비 7%의 실행시간 증가가 나타난 것이데, SPEC CPU 2017의 성능이 28% 증가한 것에 비하여 미미한 실행시간 증가가 일어난 것이다. 이를 통해 dynamic grain scheduling policy는 특정 command를 우선하는 policy에 비하여 두 가지 command의 성능을 모두 고려한 policy라고 할 수 있다.

1.2 논문 구성

본 논문은 2장에서 PIM과 그와 관련한 연구에 대하여 소개한다. 이후 제 3장인 background에서는 본 연구에서 PIM으로 설정한 NEWTON memory에 대하여 설명한다. 제 4장에서는 본 연구에서 제안하는 fine grain scheduling policy와 이를 이용한 dynamic grain scheduling policy를 설명하고 마지막 제 5장에서 실험 결과와 그에 대한 분석을 진행한다.

제 2 장 관 련 연 구

PIM 연구는 메모리 내부의 구조를 바꿔서 PIM을 구현하는 연구와 기존 host에서 연산을 수행하던 것을 PIM에 어떻게 분배할지를 찾는 연구, 그리고 PIM에 host 명령어와 PIM 명령어가 동시에 접근할 때 그 둘 사이의 스케줄링에 관한 연구[15]로 나뉜다. 본 장에서는 이러한 연구들의 특징과 경향성에 대하여 알아본다.

2.1 새로운 PIM 구조 제안 연구

본 연구에서 PIM으로 설정한 NEWTON 외에도, DRAM의 내부 구조를 변화시켜 새로운 PIM 메모리를 제안한 연구가 활발히 진행되었다. 이러한 연구들은 RRAM(Registered Random Access Memory)[2]나 PCM(Phase Change Memory)등 DRAM 이외의 메모리를 사용하여 새로운 PIM을 제안하거나 HMC(Hybrid Memory Cube)[3]를 이용하였다.[4] 그 중에서 DRAM의 1T 1C를 이용한 구조를 변경하여 3T 1C의 구조를 이용하거나[1] sense amplifier에 LUT와 같은 logic을 추가하는 방법[12]을 제안하였다. 이외에도 bank 외부에 PE(Processing Element)를 추가하여 연산을 수행[13][5]을 하는 연구도 활발히 진행되었다.

이러한 연구는 PIM을 위해 새로운 메모리 구조를 제안하여 memory intensive한 네트워크의 성능을 더 향상시켰다. 하지만 이러한 연구는 DRAM이 아닌 다른 메모리를 이용하여 PIM을 구현한 것으로, DRAM 메모리보다 면적당 저장을 할 수 있는 bit의 수가 적거나, 메모리에 있는 데이터를 read 혹은 write을 할 때 걸리는 시간이

늘어나게되어 실제 메인 메모리로 쓰일 경우 기존에 실행중인 application의 성능을 저해할 수 있다. DRAM을 이용하지만 bank 외부에 PE를 두어서 PIM을 구현한 연구들의 경우, 이러한 단점을 가지지 않지만, bank parallelism을 이용하지 못한다. Bank 외부에 PE가 있기 때문에 PIM연산은 rank 단위로 연산이 수행되고, rank마다 16개의 bank가 있는 최근의 메모리와 비교하면 rank parallelism은 bank parallelism에 비하여 성능 향상이 작게 된다.

2.2 PIM kernel off loading에 관한 연구

PIM을 이용하여 deep learning network를 가속할 경우 해결해야할 문제는 크게 2가지로 나뉜다. 첫번째는 어떤 함수 혹은 커널을 PIM에서 수행시킬지에 대한 연구이다. PIM에서 연산을 수행할 수 있다는 것은 기존의 host processor에서도 연산을 수행시킬 수 있다는 의미이다. Memory intensive하지 않은 함수 혹은 커널은 data reuse가 많기 때문에 PIM이 아닌 기존의 host processor에서 연산을 수행하는 것이 성능 향상에 도움이 된다. PIM은 memory intensive하고 data reuse를 많이 하지 않는 함수 혹은 커널에서 성능 향상이 나타나게 된다. 그렇기 때문에 프로그래머가 자신의 코드를 어떤 연산기에서 수행을 시킬지 결정해야하거나 이를 도와준 OS나 하드웨어의 도움이 필요하다. 대표적으로 [6]연구에서는 HMC를 이용한 near data processing을 제안하였다. 이 연구에서는 GPU에서 수행중인 커널 중에서 어떤 커널을 PIM에서 수행을 시킬지를 결정하는 off loading에 관한 연구를 진행하였다.

두번째는 memory coherence문제이다. PIM을 이용할 경우 multi core computing과 같이 여러 개의 core에서 하나의 데이터에 접근하는 coherence 문제가 발생한다. 기존의 multi core computing은 RMW(Read Modify Write) command를 이용하여 이 방법을 해결하였지만 PIM에서 RMW를 이용할 경우 memory bandwidth가 줄어들 수 있다. 따라서 여러 연구에서는 이러한 data coherence 문제를 해결하기 위하여 다양한 방법을 시도하였다. 그 중에서 [7]는 우선적으로 PIM에서 데이터를 이용하여 연산을 수행하고, 이를 끝마쳤을 때 연산을 위해 가져왔던 데이터가 변경되었을 경우 수행했던 커널을 다시 수행하는 optimistic한 방법을 제안하였다. [14]

연구에서는 memory controller 주위에 존재하는 locality monitor를 이용하여 함수의 데이터가 locality가 없다면 메모리 내부에 있는 core를 이용하여 빠른 데이터 접근을 이용한 연산을 수행한다.

이렇게 PIM off loading에 관한 연구는 두 가지 문제를 해결하기 위해 다양한 방법을 제시하였다. 하지만 이러한 연구는 data, 함수 혹은 커널의 특징에 따른 PIM off loading에 관한 방법을 제시하였고 동시에 host processor와 PIM core가 메모리에 접근할 때 발생할 수 있는 문제점에 대한 해결책을 제시하지는 못하였다.

2.3 Concurrent memory access에 관한 연구

PIM을 이용하여 가속을 하게 될 경우 host processor와 PIM의 PE에서 동시에 접근을 할 경우가 발생한다. 만약 두 개의 core가 같은 데이터에 접근을 할 경우 발생하는 문제는 2.2장의 PIM off loading에서 다룬 coherence 문제와 같다. 하지만 host processor와 PIM의 PE에서 다른 application을 수행할 경우, 두 개의 core는 서로 다른 데이터에 접근을 하게 되고 이는 memory contention 문제가 발생한다. 이러한 memory contention을 줄이는 memory scheduling policy는 기존 컴퓨터 구조에서도 중요한 문제였으며 PIM을 활용한 구조에서도 중요한 문제가 되고 있다.

이전 연구[18]에서는 PIM 이전의 다른 종류의 core가 존재하는 heterogeneous system에서 각각 application을 실행할 때 발생할 수 있는 memory contention 문제를 다루고 있다. 이 연구에서는 application에 따라 접근하는 bank를 제한하는 virtual bank divide 방식을 선택하고 있다. 이 방식은 OS의 도움을 통해 application마다 bank를 할당하였다.

이후 PIM을 위한 system연구인 [8]에서는 data movement energy를 줄이기 위한 PIM을 제안하면서 host processor와 PIM에서 동시에 접근할 경우 memory ownership transition 방식의 memory request scheduling 방식을 제안하였다. 마지막으로 [15]연구에서는 PIM에 host와 PIM processor가 동시에 접근할 때, performance 향상을 위한 연구를 진행하였다. 이 연구에서는 fine grained interleave를 통한 memory request scheduling을 제안하였다. 하지만 이러한 concurrent memory access연구에서는 각 연구에서 설정한 PIM에 대한 optimal한 방법론을 제시할 수는 있어도 NEWTON

memory의 특성을 반영한 memory scheduling policy를 제시하지 못하고 있다는 한계가 있다. 따라서 본 연구에서는 NEWTON memory의 특성을 반영한 PIM command scheduling policy를 제안하려고 한다.

제 3 장 Background

3.1 NEWTON memory 구조

3.1.1 NEWTON memory architecture

본 장에서는 대표적인 PIM 메모리인 NEWTON 메모리의 구조와 특징에 대하여 알아본다. 그림 3.1은 NEWTON 메모리의 모습을 보여준다. NEWTON 메모리에는 일반적인 DRAM cell과 sense amplifier에 더하여 sense amplifier에 있는 1024개의 FP16 중에서 16개의 word(FP16)를 선택하는 MUX가 그 밑에 존재한다. 또한 16개의 word의 MAC 연산을 진행할 MAC unit과 그 결과를 저장하는 result buffer는 그 밑에 배치되어 있다. Result buffer는 latch로 되어 있기 때문에 MAC 연산을 수행할 때마다 이전 결과와 새롭게 input vector와 weight를 곱한 결과가 accumulation된다. 이를 통해 NEWTON memory는 GEMV 연산에 필요한 matrix vector multiplication과 accumulation 연산을 수행한다.

NEWTON 메모리의 특징은 all bank operation이 가능하다는 점인데, 이러한 all bank operation을 가능하게 하는 global buffer가 존재한다. Global buffer는 모든 bank에서 접근이 가능하기 때문에 모든 bank가 하나의 global buffer에 있는 값을 동시에 읽을 수 있다. 따라서 global buffer에 값을 저장하게 된다면 bank의 개수만큼 동시에 연산을 수행할 수 있다. Global buffer의 크기는 DRAM array의 row size와 동일하게 2KB이다. 두 개의 크기가 동일해야 matrix vector multiplication을 수행할 때 operand의 수가 동일하게 된다. 만약 matrix vector의 크기가 2KB(FP 16 기준 1024 size)보다 크게 될 경우 새롭게 global buffer에 input vector를 저장하고 연산을 수행해야

한다. MAC unit은 기존의 논문에 따르면 tree 구조로 되어있다. MUX를 거쳐서 나온 16 개의 word 와 16 개의 global buffer 에서 나온 word 가 muliply unit 과 adder unit 을 거쳐서 결과가 나오게 된다. 기존 논문에서는 이러한 tree구조의 중요성을 강조하지만 본 논문에서는 NEWTON의 구조에 대해서 자세히 다루지 않고 그러한 특성도 적용을 하지 않기 때문에 본 논문에서는 그림 3.1과 같이 하나의 MAC unit 으로 치환하기로 하였다. NEWTON 메모리의 구조 중에서 global buffer는 SRAM으로 이루어져 있고 bias는 latch로 구성되어 있기 때문에 값을 읽거나 쓰기 전에 activation이나 precharge를 하지 않아도 된다.

이런 NEWTON의 구조는 다른 PIM과 비교하여 가지는 장점이 있다. Global buffer를 활용하여 모든 bank에서 접근이 가능하고 이를 활용하여 input vector를 한번만 저장을 하게 된다면 모든 bank에서 input vector를 이용한 연산이 가능해진다. 다른 rank parallelism을 이용한 PIM은 NEWTON의 bank parallelism에 비해 동시에 수행 가능한 PE의 수가 줄어들고, 데이터를 bank의 I/O bus를 통해 bank 외부로 가져가야한다. 따라서 rank parallelism을 활용한 PIM은 NEWTON에 비하여 I/O bus에 bottleneck이 생기게 된다.

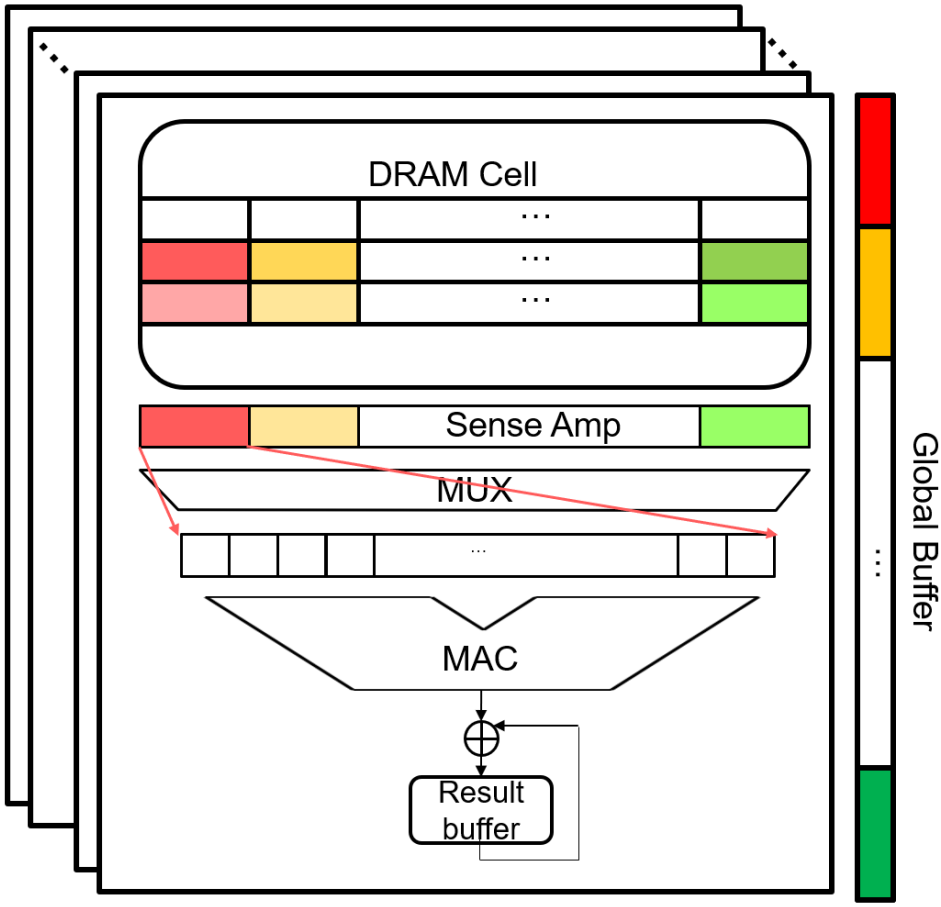

그림 3. 1. NEWTON architecture overview

3.1.2 NEWTON memory data mapping

본 장에서는 NEWTON 메모리에 GEMV의 operand 데이터가 어떻게 매핑이 되는지를 설명한다. Matrix-vector multiplication을 뜻하는 GEMV는 deep learning 연산에 있어서 필수적이다. GEMV는 fully connected network 뿐만 아니라 RNN 계열에서 벡터화 된 input을 학습된 weight와 곱하는데도 쓰인다. 이러한 GEMV연산은 weight와 input vector 모두 메모리 내부에 저장이 되어 있고 그 크기가 클수록 메모리에서 읽어오는 양이 많기 때문에 GEMV연산을 많이 하고 data reuse의 양이 적을수록 memory intensive한 네트워크가 된다. 따라서 memory intensive한 deep learning network의 속도 향상은 GEMV연산을 얼마나 빠르게 수행을 하느냐에 따라 달려있고 데이터 매핑을 어떤 방법을 사용하는지에 따라 성능이 달라지게 된다.

GEMV 연산은 식 3.1과 같이 weight와 input vector가 곱해진 값에 bias가 더해져서 결과 값이 나오게 된다. A는 weight를 나타내는 것으로 matrix 행렬이고 본 논문에서는 그 크기를 1024 x 1024로 설정하였다. ‘x’는 input vector를 의미하는 것으로 weight A와 차원을 맞추기 위하여 1024 크기의 벡터로 설정하였다. 만약 이 벡터의 크기가 변화한다면 weight의 크기도 바뀌어야 한다. ‘b’는 bias를 뜻하는 것으로 fully connected layer에서 사용되는 bias를 의미한다. 이 bias의 크기 또한 input vector의 크기와 동일하게 된다. ‘y’는 output vector를 의미하는 것으로 1024 크기의 input vector와 동일하게 된다. NEWTON 메모리는 기본적으로 training을 위한 연산기를 지원하지 않기 때문에 inference를 위한 PIM이다. Inference를 위한 PIM이므로 weight의 값은 변하지 않고 input vector만 변하게 된다. 따라서 weight는 DRAM cell 내부에 저장되어 있는 값을 사용한다. Input vector는 global buffer에 저장이 되는데, 이를 통해 all bank operation을 수행 하지만 global

buffer에 모든 bank가 접근이 가능하기 때문에 input vector를 모든 bank에 넣어주지 않아도 된다는 장점이 있다. 마지막으로 bias값은 latch인 result buffer에 저장이 되어, weight와 input vector가 곱해진 값에 계속해서 accumulation이 된다.

$$y = A \cdot x + b \quad \text{(식 3.1)}$$

3.1.3 NEWTON 명령어

본 장에서는 NEWTON 명령어의 종류와 그 역할에 대해서 설명한다. 기존 main 메모리는 호스트에서 메모리에 접근하는 명령어는 read와 write만 있었다. 이 두 가지 명령어 외에도 메모리 컨트롤러에서 DRAM을 컨트롤하기 위한 activation 과 precharge 명령어가 존재한다. 또한 DRAM 셀에 저장된 정보를 잃어버리지 않기 위한 refresh 명령어도 존재한다.

NEWTON 메모리는 이 명령어들에 더해 GEMV 연산을 위한 4 가지 명령어를 추가로 받아들인다. 이 4 가지 명령어는 GEMV 연산을 위해 데이터를 저장하거나 MAC 연산을 수행 또는 수행된 결과를 읽어오는 역할을 한다. 이러한 명령어는 표 3.1과 같이 write 명령어와 비슷한 WRGB와 WRBias가 있고 MAC연산을 진행하는 ABMAC이 있다. 마지막으로 MAC연산을 수행한 결과를 읽어오는 RDMAC command가 존재한다. WRGB는 write global buffer를 의미하는 것으로 global buffer 에 값을 쓰는 동작을 하는 명령어로서 matrix 의 크기에 따라 달라진다. WRBias는 write bias를 의미하는 것으로 result buffer에 bias값을 쓰는 동작을 하는 명령어로서 보통 메모리에 있는 bank의 수 만큼 명령어가 반복이 된다. ABMAC은 all bank MAC 을 의미하는 것으로 sense

amplifier와 global buffer에 있는 값을 읽어와서 MAC unit에서 연산을 수행시키는 명령어로서, WRGB 명령어와 마찬가지로 matrix의 크기에 따라 명령어의 횟수가 달라진다. 마지막으로 RDMAC은 read MAC을 의미하는 것으로 result buffer에 있는 값을 읽어오는 명령어이다.

Command name	역 할	target
WRGB	Write to global buffer	Global buffer
WRBIAS	Write to bias	Result buffer
ABMAC	All bank MAC operation	MAC unit
RDMAC	Read result	Result buffer

표 3. 1 NEWTON command

3.2 All bank row activation overhead

본 장에서는 NEWTON 메모리의 특징 중 하나인 all bank row activation overhead를 소개한다. NEWTON memory 기존 연구에서 설정한 PIM과는 달리 bank parallelism을 위한 all bank operation 동작을 수행한다. 그 중에서 NEWTON의 command 중에서 MAC command는 all bank 동작이고, MAC command는 activation 명령어에 의해 DRAM cell의 데이터가 sense amplifier으로 올라간 뒤, 그 값을 이용하여서 연산을 수행한다. 그렇게 때문에 MAC command는 all bank row activation이 필요하다.

그림 3.2과 그림 3.3은 이러한 동작을 보여준다. 이 그림의 가로 축은 메모리의 수행 시간을 나타내고 세로 축은 그 명령어의 종류를 나타낸다. 명령어를 나타내는 사각형의 가로 길이는 그 명령어를 수행하는데 걸린 시간을 의미한다. 그림 3.2의 경우에는 중간에 삽입된 normal request가 없기 때문에 all bank row activation을 1번만 수행하게 된다. 하지만 그림 3.3에서 MAC1의 실행을 위해 all bank row activation이 실행되었지만 중간에 삽입된 normal command에 의해 activate된 row가 바뀌게 된다. 모든 MAC command를 수행하기 위해서는 모든 bank의 activated row가 MAC command의 target row이어야 한다. 만약 normal command가 없었다면 MAC1에서 activation된 row를 target row와 같게 맞추었기 때문에 MAC2를 위한 all bank activation이 필요 없겠지만, normal command에 의해 activation이 필요하게 되었고 이는 all bank row activation의 시간만큼 overhead가 발생한 것을 의미한다. All bank row activation으로 발생한 overhead는 NEWTON command의 수행시간도 증가시키지만 MAC2 command의 수행시간이 길어지기 때문에 MAC2 command 이후에

나오는 normal command의 latency도 증가시키는 결과를 초래한다. 이렇게 NEWTON의 특성인 all bank row activation overhead를 고려하지 않은 memory controller의 설계는 overhead를 늘리고 결국은 성능이 좋지 않은 memory controller를 만들게 된다.

표 3.2는 matrix 크기에 따른 all bank activation이 차지하는 latency를 나타낸 것이다. all bank activation은 row를 바꿀 때 발생하는 timing인 tRRD와 4개의 row를 activation 시킬 경우 발생하는 tFAW로 인해 다른 timing(tCCD)에 비해 그 latency가 길게 된다.

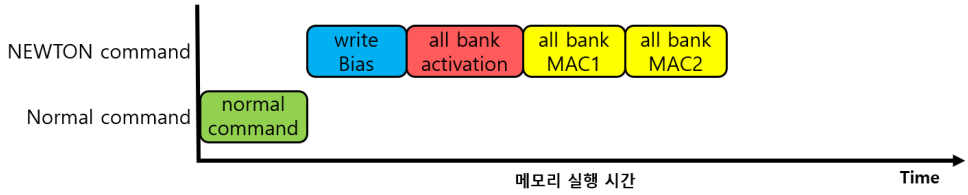

그림 3. 2. Memory timeline without request intervene

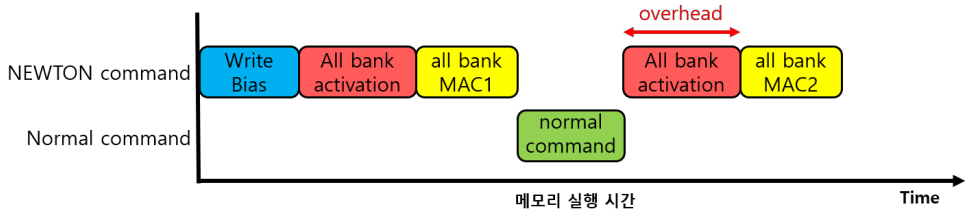

그림 3. 3. Normal memory request로 인한 all bank activation overhead

Size	ACT
256	44.9%
512	38.1%
1024	28.1%
2048	19.8%
4096	20.3%

표 3. 2 matrix size에 따른 activation latency 비율

이러한 activation overhead는 그림 3.4와 같이 여러 개의 host processor core에서 다른 thread를 실행시킬 때 memory contention을 발생시킨다. 하나의 메모리에 여러 개의 thread가 동시에 접근을 하게 된다면 메모리 내부의 bank의 활성화된 row가 바뀌어야 한다. 기존의 normal DRAM command는 다른 thread에서 온 command가 반복이 된다면, 활성화 되어 있지 않은 bank만 수정을 하면 되었다. 하지만 NEWTON의 MAC command는 하나의 bank만이 아니라 모든 bank를 precharge 동작 후에 다시 activation을 시켜야 한다. 이는 다른 PIM과는 다른 NEWTON memory 만의 특징으로 만약 이러한 특징을 반영하지 않고 memory scheduling policy를 설정한다면, NEWTON memory를 완전히 활용하지 못하고 성능 저하를 일으키게 된다. 따라서 본 연구에서는 NEWTON memory의 특징인 all bank activation overhead를 고려하여 PIM command scheduling policy를 제안하려고 한다.

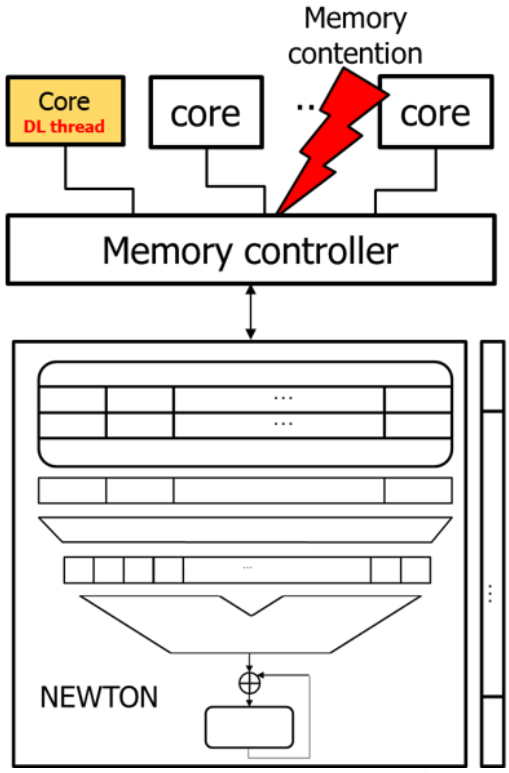

그림 3. 4 Memory contention 발생

제 4 장 제안 PIM command scheduling policy

4.1 제안 PIM controller overview

제안하는 PIM command scheduling policy를 위해 우선 제안하는 PIM controller의 overview는 그림 4.1과 같다. Host processor에 있는 여러 개의 core 중에서 NEWTON thread(Deep learning network)를 실행하는 core가 있다고 가정할 때, PIM 연산을 위해 필요한 PIM command는 L1 cache와 L2 cache를 거쳐 last level cache에 도달하게 된다. NEWTON command는 normal command처럼 메모리에 있는 데이터를 읽고 쓰는 명령어가 아니기 때문에 non-cacheable로 수행된다. 이후 last level cache를 통해 들어오는 memory transaction는 normal transaction인지 newton transaction인지에 따라 normal transaction buffer와 newton transaction buffer로 나뉘어 들어간다. 두 개의 buffer는 모두 memory transaction을 저장한 뒤, transaction이 완료된 후에 read의 경우는 data와 함께 last level cache로 돌아가게 된다.

그림 4.1에서 색칠된 부분이 기존의 memory controller에서 제안하는 PIM controller로 바꾸었을 때 추가된 부분이다. MAT는 MAC Address Table의 약자로서 추후 4.3장에서 설명을 하겠다. NT arbitrator는 NEWTON arbitrator로서 NEWTON command에 필요한 activation이나 precharge command를 arbitration하고 normal command와의 arbitration을 하는 모듈이다. 기존 memory controller에 있는 normal arbitration 모듈은 normal request(Read or Write)를 받아들여 FR-FCFS(First Ready, First Come First Serve)방식을 사용하여 가장 먼저 issue할 수 있는 command를 issue한다. Issue하는

command는 precharge, row activation, cas(read/write)가 있다. Precharge command는 bank의 row가 open이 되어 있을 때 bank는 닫아주는 command이다. Row activation이란 close되어 있는 bank에서 command의 target row를 sense amplifier에 올려주는 activation command를 수행하게 된다. Memory row state는 bank별로 row의 상태가 open인지 close인지를 나타내 주는 것으로, 이를 통해 normal arbitrator 모듈에서 bank row의 state를 보고 어떤 command를 가장 먼저 issue할 것인지를 판단하게 된다. 가장 먼저 issue가 가능하더라도 timing counter 모듈에서 아직 issue하지 못한다고 판단이 되면 issue를 할 수 없다. Timing counter 모듈은 DRAM의 timing parameter를 저장하고 command가 issue될 때마다 어떤 command가 issue해도 안전한지를 판단한다.

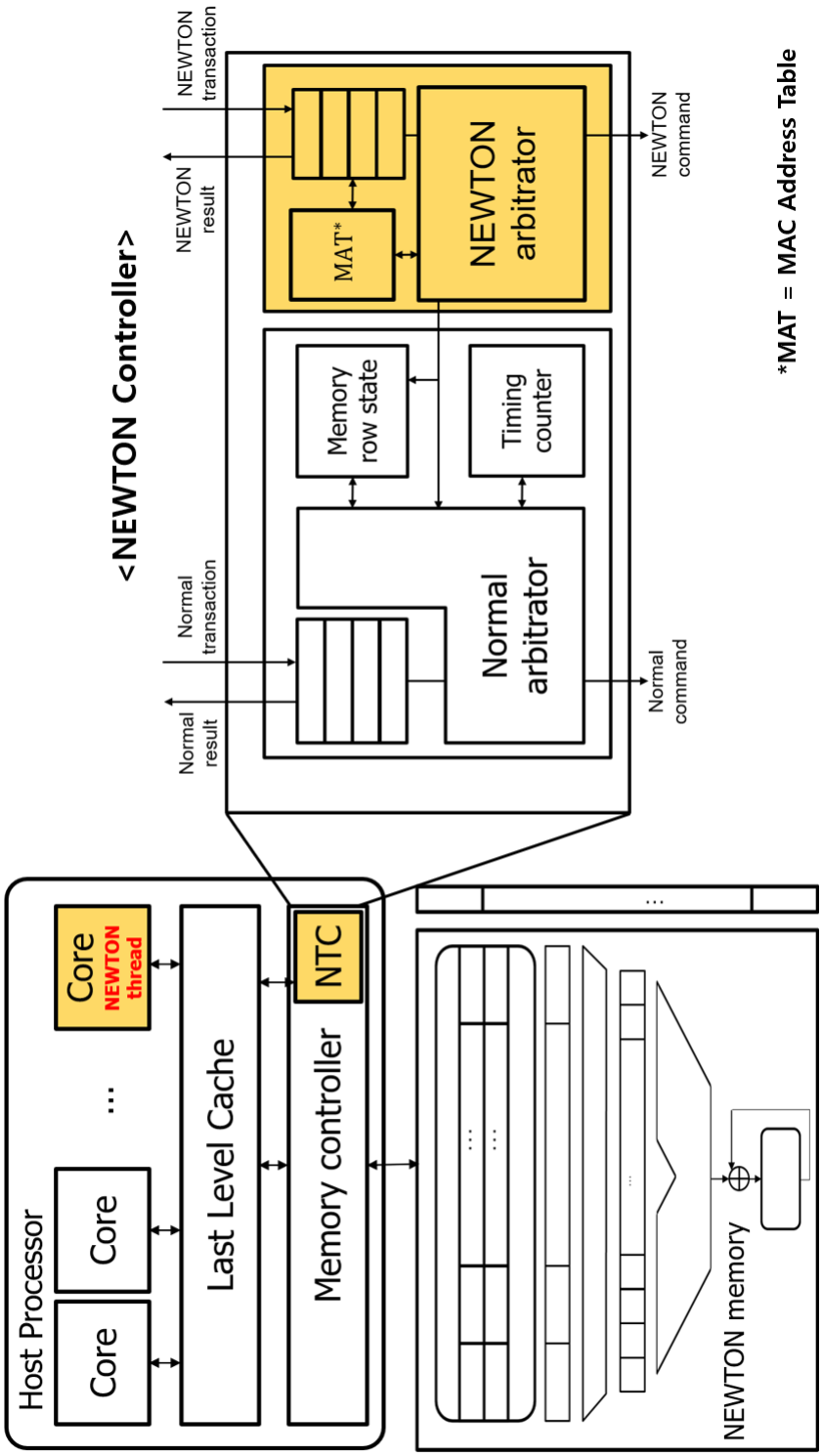

그림 4. 1. PIM controller overview

NEWTON arbitrator 모듈에서는 normal arbitrator 모듈과 같이 들어온 NEWTON command 중에서 어떤 command를 issue하면 좋을지를 판단한다. Normal command와는 달리 NEWTON command는 out of order 방식이 불가능 하기 때문에 in order 방식으로 issue를 하지만 all bank operation인 MAC command의 경우 모든 bank에 있는 row가 open이 되어 있어야 하고 row또한 MAC command의 target address와 같아야 한다. 이러한 이유 때문에 NEWTON arbitration 모듈에서는 NEWTON command에 필요한 precharge와 row activation command를 issue하게 된다. 또한 result buffer나 global buffer에 접근하는 Write bias, Write global buffer, Read result command와 다른 read, write, MAC command사이에는 all bank precharge가 필수적이다. 이를 위해서도 NEWTON command는 row state에 따른 command를 issue 하게 된다. 따라서 NEWTON arbitrator는 normal command를 위한 normal arbitrator와 timing, 현재 bank의 state를 고려하여 어떤 command를 issue할 지 결정하게 된다.

4.2 Fine grain scheduling policy

PIM controller에서 all bank row activation overhead를 줄이기 위해 normal command를 grain size 단위로 issue 하는 방법을 생각할 수 있다. 그리고 grain size보다 작은 개수의 normal command가 있다면 normal command를 issue하지 않고 NEWTON command buffer에 존재하는 NEWTON command를 issue한다. 예를 들어 grain size가 8이라고 가정해보자. 이때 NEWTON command를 issue하는 도중 normal command buffer에 8개의 normal command가 있다면 이 8개의 normal memory request를 모아서 한번에 NEWTON memory에 issue를 하는 방식이다. Fine grain scheduling policy는 normal command를 grain size만큼 issue를 하기 때문에 baseline 방식인 FIFO방식보다 MAC command사이에 issue가 되는 normal command의 수가 줄어들게 된다. MAC command 중간에 issue가 되는 normal command는 all bank activation overhead를 발생시키기 때문에 이 policy는 all bank activation overhead를 줄일 수 있다.

Fine grain scheduling policy는 all bank activation overhead를 줄일 수 있지만 grain size에 따라 성능이 달라질 수 있다. 만약 grain size가 큰 경우 normal memory request를 큰 grain size만큼 모아서 issue를 하기 위해 NEWTON command를 먼저 issue를 하게 된다. NEWTON command 중간에 issue가 되는 normal command가 줄어들기 때문에 all bank activation overhead는 줄어들게된다. 따라서 fine grain scheduling policy는 메모리의 수행시간을 줄어들게 만들어 NEWTON command의 성능을 향상시킨다. 하지만 너무 큰 grain size 방식은 NEWTON thread의 성능 향상을 할 수 있지만 그림 4.2와 같이 normal request의 latency가 길어지거나 return되는 normal

command가 적어지기 때문에 normal command의 starvation이 발생할 수 있다.

반대로 grain size를 작게 설정한 경우 normal command의 latency는 짧아질 수 있지만 그림 4.3과 같이 normal command의 issue 주기가 짧아지고 이 경우 앞에서 언급한 all bank row activation overhead가 발생하게 된다. 그렇기 때문에 많은 all bank row activation overhead는 normal command와 NEWTON command의 수행시간을 모두 증가시킬 수 있다.

이렇게 grain size의 크기를 크거나 작게 설정하게 되면 추가적인 overhead가 발생하거나 latency와 throughput을 모두 저하시킬 수 있다. 본 논문에서는 이러한 grain size를 하나로 고정시키지 않고 normal command와 NEWTON command의 상황에 따라 바꾸는 dynamic grain scheduling policy 방식을 취한다. 이를 통해 large grain size와 grain grain size의 장점을 모두 극대화할 수 있을 것이다.

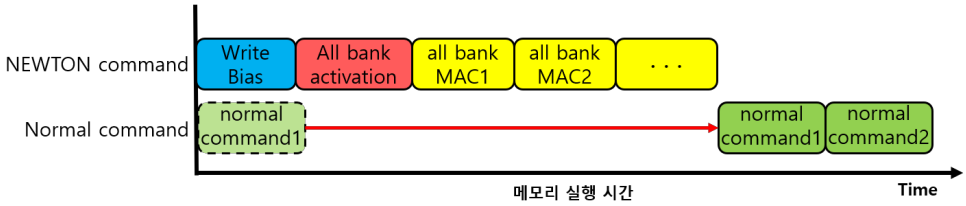

그림 4. 2 Large grain size issue policy

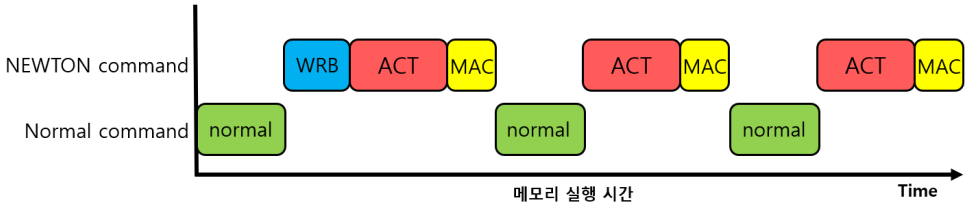

그림 4. 3 Small grain size issue policy

4.3 MAC Address Table (MAT)

Optimal 한 grain size 조절을 위해 본 논문에서는 MAC address table이라는 새로운 요소를 제안한다. MAC address table이란 NEWTON command가 반복이 된다는 점을 고려하여 NEWTON command중 하나인 MAC command의 주소와 반복되는 write bias의 수, 현재까지 issue한 write bias의 수 그리고 현재 grain size state를 저장하고 있다. 그림 4.4.는 이러한 MAC address table의 예시를 보여주고 있다.

MAC address는 MAC command의 target address 중에서 column의 주소를 제외한 address를 저장하는 항목이다. Column address를 제외하는 이유는 column address는 뒤이어 나오는 MAC command에 의해 바뀌기 때문이다. Write bias는 이전 주기에서 반복된 write bias command의 명령어를 저장하는 곳으로 weight의 위치가 변하지 않는다면 그 weight를 사용하는 command의 trace는 일정할 것이므로 write bias의 수도 변하지 않는다. Issued write bias는 만약 write bias를 issue 중에 있다면 연속해서 몇 개의 write bias를 issue하였는지를 저장하게 된다. 이 값은 PIM controller에서 write bias를 issue할 때마다 1씩 증가하게 된다. 또한 최대 개수는 이전 주기에서 반복된 write bias의 수와 같으므로 write bias 항목과 같게 된다면 write bias command의 반복이 끝남을 의미하고, PIM controller에서는 이 값을 0으로 만든다. 마지막으로 grain state항목은 normal command의 grain size가 large인지 small인지를 나타내주는 flag로서 1 bit만 필요하게 된다. 만약 추후 연구에서 grain size의 크기를 좀더 세분화 하고 싶다면 이 flag의 bit 크기를 늘려서 다양한 grain size로 만들 수 있을 것이다. 기존의 memory controller에는

MAT와 비슷한 memory row state가 있다. Memory row state는 memory cell의 row가 open인지 close인지를 나타내 주는 것으로 본 연구에서는 이러한 memory row state를 변형하여 MAT를 만들었다.

MAC address	Write bias	Issued Write bias	Chunk state
0x2000	16	12	Small
0x3000	32	0	Large

그림 4. 4. MAC Address Table 예시

4.4 Dynamic grain command scheduling policy

Dynamic grain command scheduling방식이란 MAC address table을 이용하여 NEWTON memory에 적합한 grain size를 결정하는 방식이다. 본 논문에서는 grain size를 large grain size와 small grain size 두 개로 구분하였고 필요하다면 더 작은 경우로 나눌 수 있다.

NEWTON memory에서는 MAC command를 비롯한 NEWTON command가 반복이 되고 그 주소 또한 일정하게 된다. NEWTON command가 들어온다면 MAC command의 target address를 가장 먼저 보게 된다. 이때 이 target address의 값이 MAT에 없다면 MAT에 새롭게 추가하여 기록을 하게 된다. MAC command 이전에 연속된 write bias의 수가 몇 개인지를 기록하는 register의 값을 write bias 항목에 적게 된다.

이후에 두번째 주기가 반복이 된다면 첫번째 주기에서 issue한 MAC command의 target address와 맞는 주소를 MAC address table에서 찾을 수 있을 것이다. 그리고 write bias를 issue할 때마다 MAC address table에 있는 issued write bias 항목을 1씩 늘려준다. 이런 방식으로 issued write bias항목을 증가시킨다. 이 때, default grain size state는 large로 설정 되어있다. 만약 중간에 issued write bias의 값이 threshold보다 크게 된다면 grain state를 small로 바꾼다. 본 논문에서는 threshold의 값을 [write bias의 항목 – 4]로 설정하였다. Grain size state를 바꾼 후, normal command buffer에 있는 normal command의 숫자가 grain size보다 크다면 grain size만큼 issue를 하게 된다. Issue를 한 후에도 grain size보다 많은 개수의 normal command가 있다면 계속해서 normal command를 issue하게 된다. Normal command를 작은 grain size만큼 issue를 함으로써 normal

command의 latency는 command의 latency가 감소하게 된다. 이후에 issued write bias의 크기가 write bias와 같게 된다면, issued write bias의 항목은 0이 되고 grain size의 크기도 large grain size로 돌아가게 된다.

그림 4.5는 dynamic grain command scheduling policy를 적용한 모습을 보여준다. 가장 먼저 0x2000의 주소에 해당하는 MAC address가 들어오게 된다면 MAT의 0x2000 주소의 칸을 채우게 된다. 이후에 파란색의 write bias가 들어오면 issued write bias의 값을 1씩 증가시키게 된다. 이 값이 threshold 값 이상이 된다면 grain size가 small로 바뀌게 되고, 녹색의 normal command를 small size만큼 issue를 하게 된다. Issue를 한 이후, write bias를 끝까지 issue를 하게 되는데 issued write bias의 값이 write bias와 같게 된다면 write bias가 종료된다. 이 때 grain size는 small에서 다시 large로 바뀐다. 이후 붉은 색의 all bank activation을 issue 하고, 다시 MAC command를 issue하게 된다. Grain size를 small로 바꾸고 issue를 해놓은 상태이기 때문에 normal command buffer에는 normal command의 수가 적게 되고 덕분에 MAC command 중간에 issue되는 normal command의 수가 줄어들게 된다.

Grain size를 large에서 small로 바꾸게 된다면 normal command를 중간에 issue하게 된다. 이 방식을 이용함으로서 MAC command가 수행되기 이전에 memory controller안에 있는 normal command buffer를 small size만큼 issue를 하게 된다. Grain size를 small로 바꾸지 않는다면 buffer에 남아있는 normal command의 수가 계속해서 증가하여 large grain size만큼 도달하게 되는 경우가 증가한다. 이는 normal command를 MAC command 중간에 issue시키게되고, all bank activation overhead를 발생시킨다.

반대로 grain size를 small에서 large로 바꾸게 된다면 small grain size로 인해 발생할 수 있는 normal command issue를 줄이게 되고 MAC command를 위한 all bank activation overhead를 줄이게 된다.

Dynamic grain scheduling policy는 이렇게 NEWTON command와 normal command의 상황에 따라 issue하는 grain size의 크기를 small과 large로 변경함으로서, normal command의 latency 감소와 all bank activation overhead의 감소를 가져올 수 있다. 이러한 dynamic grain scheduling policy를 위해서 NEWTON arbitrator에서 FSM(Finite State Machine)을 구현하였다. 이 FSM에 따라 NEWTON arbitrator와 normal arbitrator는 어떤 command를 issue할지 결정하게 된다. FSM을 나타내는 그림은 4,6과 같다.

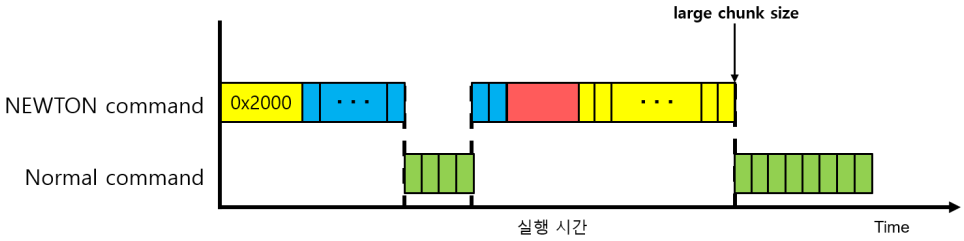

그림 4. 5 Dynamic grain scheduling policy

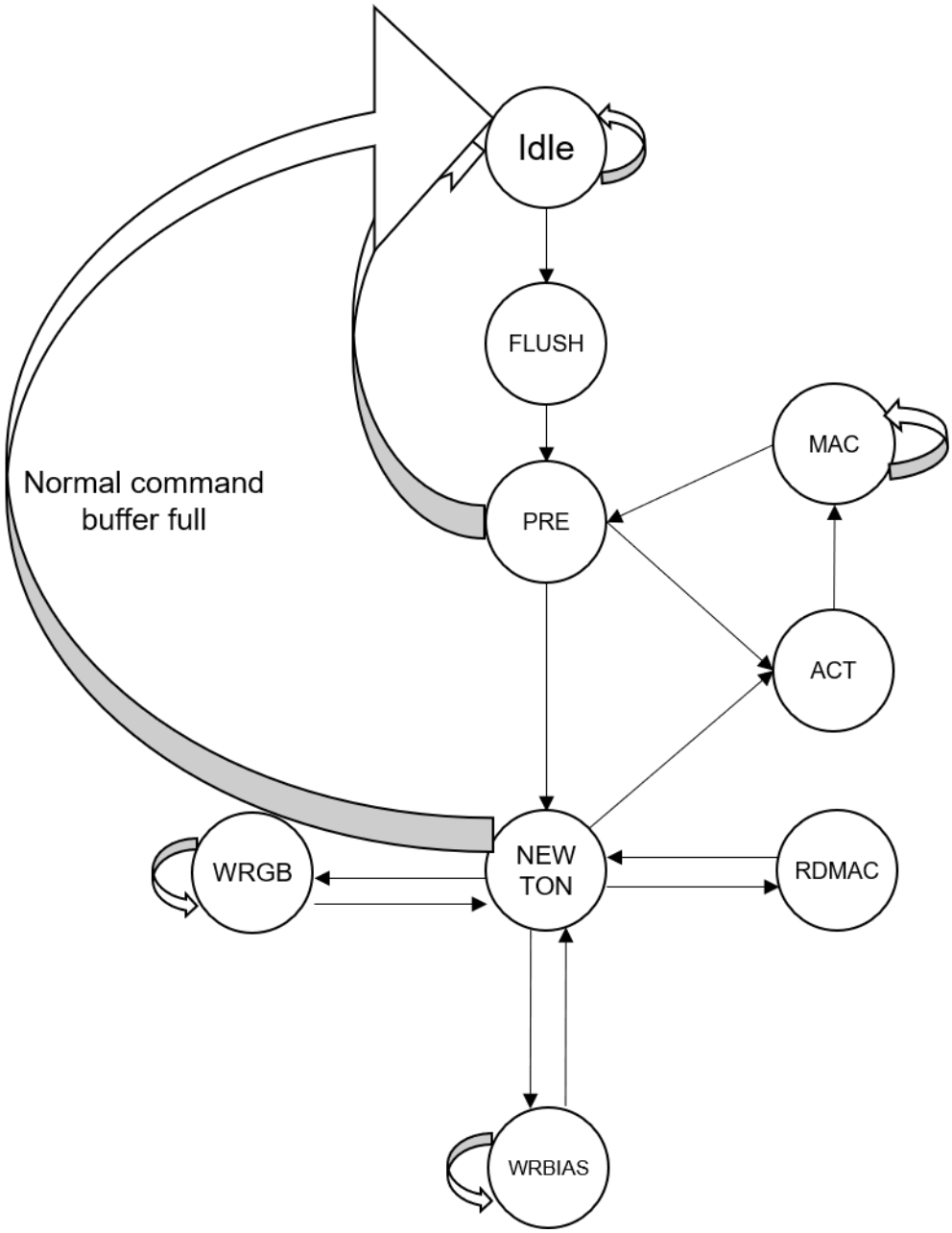

그림 4. 6 NEWTON command FSM

제 5 장 실험 결과 및 분석

5.1 Experiment methodology

Normal command와 NEWTON command사이의 dynamic grain scheduling policy 성능 측정을 위해 본 논문에서는 실험을 gem5와 DRAMsim3를 연결하여 진행하였다. Host procssor를 시뮬레이션 하는 gem5에서 나오는 memory request는 DRAMsim3에 있는 memory controller와 연결이 되고, memory controller는 본 논문에서 제안하는 policy에 혹은 baseline policy에 따라 command를 issue하게 된다. Gem5에서 설정하는 system configuration는 아래의 표 5. 1과 같이 설정하였고, DRAMsim3에서 설정한 DRAM parameter는 표 5. 2와 같이 설정하였다.

System configuration	
Processor	in order CPU, 2 GHz
L1 cache	32KB L1 data cache, 32KB L1 inst cache, 8-way assoc
L2 cache	1MB size private L2, 4 way-Associativity : 4
Last level cache	8MB size, assoc : 16, cache line : 64
DRAM	DDR4, 8Gb, x8, 1ranks, 16 banks, FR-FCFS

표 5. 1 실험 system configuration

DRAM timing parameters
tCK = 0.63, tCL=22, tRCD=22, tRP=22, tRAS=52, tFAW=34, tCCD=4

표 5. 2 실험 DRAM timing parameters

Gem5와 DRAMsim3를 이용한 본 논문 실험의 전체적인 그림은 그림 5.1과 같다. 기존의 Gem5와 DRAMsim3를 연결한 것에 더해서 NEWTON trace가 DRAMsim3의 memory controller에 접근하도록 하여서 memory controller입장에서는 동시에 gem5와 NEWTON trace가 접근을 하는 것처럼 보이게 하였다. Gem5에서 수행할 벤치마크는 SPEC CPU 2017 21개로 선정하였다. 그 목록은 표 3과 같다.

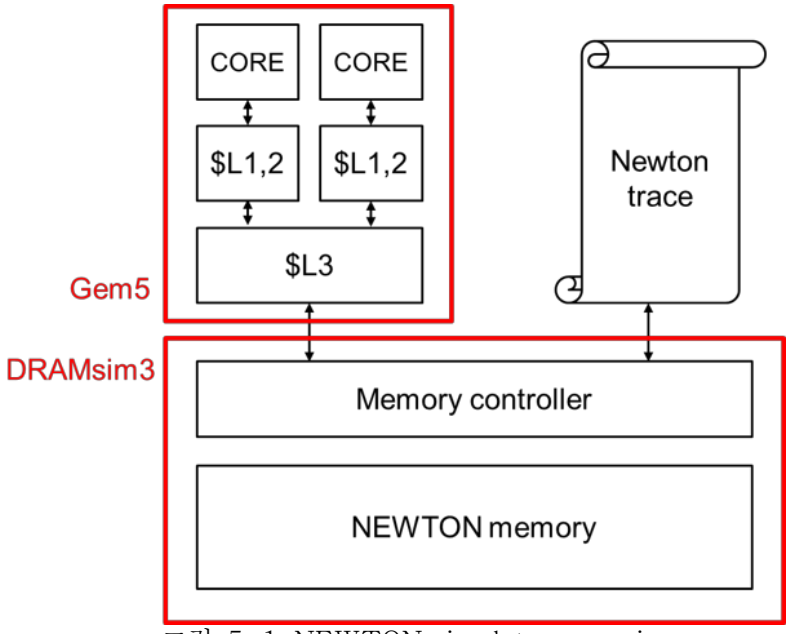

그림 5. 1 NEWTON simulator overview

SPEC CPU 2017
500.perlbench_r, 502.gcc_r, 503.bwaves_r, 505.mcf_r, 507.cactuBSSN_r, 510.parest_r, 511.povray_r, 519.lbm_r, 520.omnetpp_r, 521.wrf_r, 523.xalancbmk_r, 525.x264_r, 526.blender_r, 527.cam4_r, 531.deepsjeng_r, 538.imagick_r, 541.leela_r, 544.nab_r, 548.exchange_r, 549.fotonik3d_r, 554.roms_r

표 5. 3 SPEC CPU 2017 benchmark list

5.2 실험 결과

5.2.1 RNN execution time 비교

Deep learning network중 하나인 RNN을 CPU와 GPU에서 수행하였을 때의 결과는 그림 5.2과 같다. 전체적인 수행 시간은 GPU가 CPU의 약 24배 정도 빠르다. 하지만 그림 5.3와 같이 NEWTON에서 연산을 수행할 수 있는 GEMV의 실행시간 비율을 본다면 CPU에서의 GEMV수행시간이 더 길지만 GEMV를 제외한 함수의 수행 시간은 GPU가 CPU보다 크게 된다. 이를 통해 NEWTON 메모리를 GPU와 함께 수행하는 것보다 CPU에서 수행을 하는 것이 더 바람직함을 알 수 있다.

GPU에 비해 CPU에서 수행시간의 성능향상이 커질 수 있는 이유는 NEWTON 메모리의 특징때문이다. NEWTON 메모리는 연산을 수행하는 역할도 하지만 메인 메모리의 역할도 수행한다. GPU와 NEWTON을 이용하여 RNN을 가속할 경우, CPU의 메인 메모리에 있는 weight와 input vector, bias를 모두 CPU의 메모리에서 GPU의 메모리로 이동시켜야한다. 이 과정을 cuda malloc과 cuda memcpy로 부른다. 연산을 완료한 후에도 GPU 메모리에 있는 결과를 CPU로 이동시켜야한다. 이 과정에서 발생하는 메모리와 메모리 사이의 bandwidth로 인한 bottleneck은 NEWTON을 이용하는 효과를 감소시키기 때문에 GPU와 NEWTON을 이용한 RNN 연산은 성능 향상이 작게 된다. 하지만 CPU와 NEWTON을 이용한 경우, CPU의 메인 메모리가 곧 NEWTON이기 때문에 data movement가 사라지게 된다. 따라서 GPU에 의해 발생했던 cuda malloc과 cuda memcpy와 같은 동작을 수행할 필요가 없게된다.

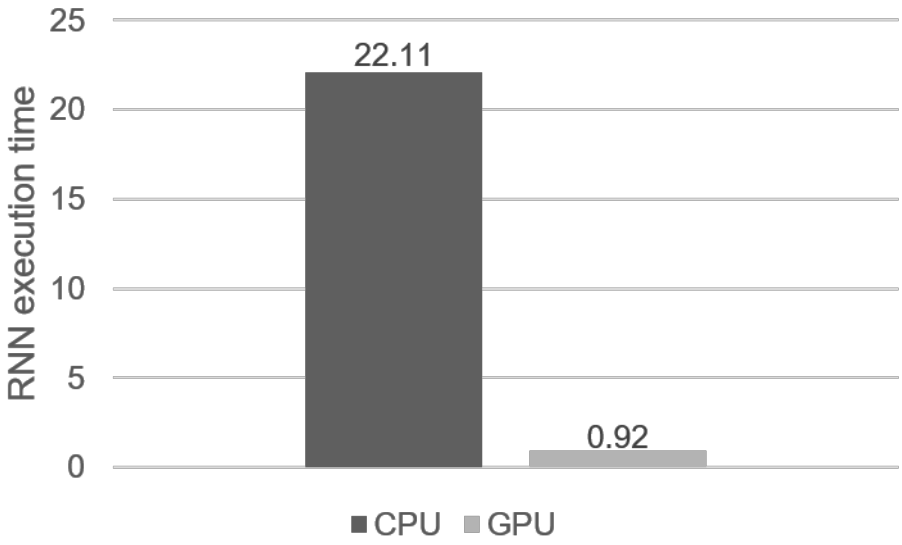

그림 5. 2. RNN execution time in CPU and GPU

Exec time(s)	GEMV	GEMV 제외	total
CPU	21.79s	0.32s	22.11s
GPU	0.13s	0.79s	0.92s

그림 5. 3 GEMV execution time in CPU and GPU

5.2.2 Memory contention으로 인한 수행시간 증가

SPEC CPU 2017의 memory request와 NEWTON command가 memory에 동시에 접근을 하게 된다면 memory contention으로 인해 전체적인 slowdown을 일으키게 된다. 이에 대한 결과는 그림 5.3과 같다. Baseline은 SPEC CPU 2017을 하나의 core에서 SPEC CPU 2017만 수행하였을 때의 수행시간을 1로 잡은 것이다. Ratio는 SPEC CPU 2017과 NEWTON trace의 평균 메모리 접근 횟수를 바꿔가며

측정한 것이다. 예를 들어 ratio 1은 only SPEC만을 수행하였을 때 발생한 normal command사이의 평균 사이 시간과 memory controller에 입력이 되는 NEWTON command의 평균 사이 시간이 같다는 것을 의미한다. Ratio 2는 NEWTON command의 평균 사이 시간이 normal command보다 2배가 된다는 것을 의미한다. 완벽한 시뮬레이션을 위해서는 NEWTON을 위한 컴파일러와 코드가 필요하지만 아직은 상용화 되지 않은 NEWTON 이므로 이렇게 trace 시뮬레이션으로 진행을 하였다.

그림 5.4는 ratio가 작을 수록 slowdown이 더 커지는 것을 보여준다. Ratio가 작다는 의미는 NEWTON을 normal command에 비해 더 자주 controller에 입력이 된다는 의미이므로 NEWTON command의 비율이 높아진다는 것을 의미한다. NEWTON command의 비율이 높으므로 NEWTON command와 normal command가 memory controller에서 memory contention이 더 자주 발생하게 된다. 따라서 NEWTON command의 비율이 가장 높은 ratio 1에서 가장 큰 slowdown이 발생하였다. 또한 FIFO 방식으로 작동하는 baseline이기 때문에 이러한 slowdown은 all bank activation overhead로 인해 발생하였다.

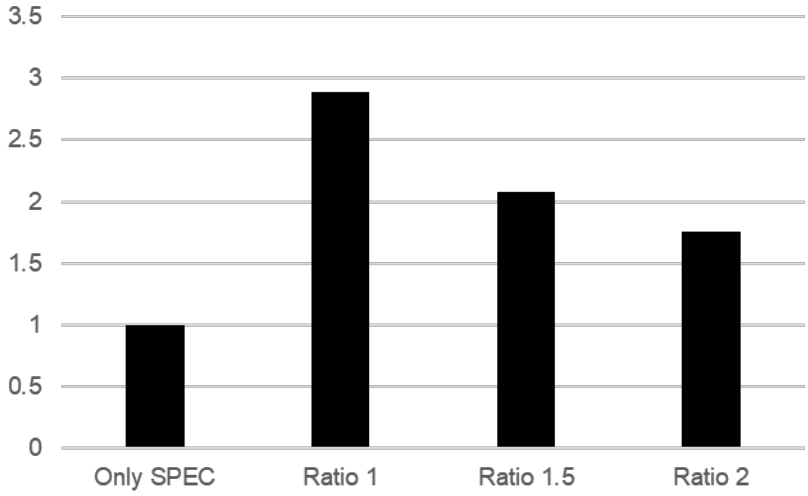

그림 5. 4 Memory contention 영향

5.2.3 All bank activation으로 인한 SPEC CPU의 성능 저하

그림 5.5는 ratio 1에서 SPEC CPU의 21개 benchmark 각각의 slowdown과 그때 발생한 actvation ovehread의 비율을 나타낸다. 막대 그래프는 각 benchmark을 ratio 1으로 수행하였을 때 only SPEC CPU만을 수행하였을 때와 비교하여 얼마나 수행시간이 늘어났는지를 보여주는 결과이다. 이때 only SPEC CPU만을 수행하였을 때의 시간을 1로 normalize하였다. 그래프의 마지막은 각 벤치마크의 결과를 기하평균을 낸 결과를 geomean으로 표시하였다. 꺽은선 그래프는 SPEC CPU와 NEWTON command를 동시에 수행하였을 때 SPEC CPU의 10개 instruction당 몇 개의 all bank activation overhead가 발생하였는지를 나타내는 그래프이다. 그래프에서 볼 수 있듯이 all bank activation overhead가 자주 발생하는 benchmark일수록 실행시간이 늘어나는 것을 확인할 수 있다. 이를 통해 SPEC CPU의 slowdown을 일으키는 요소는 all bank activation overhead의 비율이다. All bank

activation이 일으키는 normal command의 latency 증가가 전체 SPEC CPU를 수행하는데 걸리는 시간을 늘리게 된다.

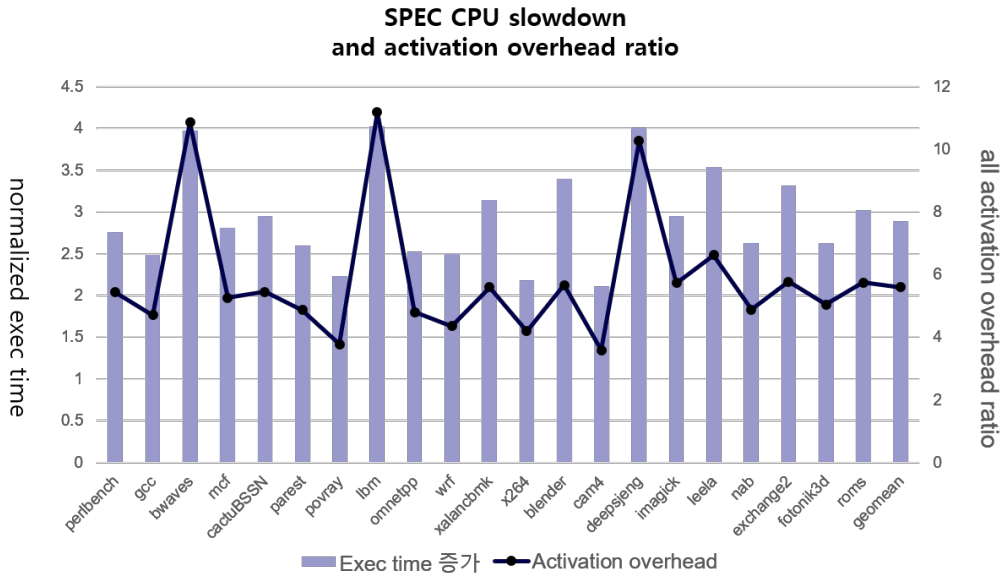

그림 5. 5 SPEC CPU slowdown and activation overhead ratio

5.2.4 grain size에 따른 execution time 변화

그림 5.6은 grain size에 따른 SPEC CPU 2017과 NEWTON의 execution time 변화를 각각 나타낸 것이다. 그래프를 보면 grain size의 크기가 커질수록 execution time이 줄어드는 것을 알 수 있다. Ratio에 따라 변화량이 다른 것도 알 수 있는데, FIFO 방식 대비 최대 10%, 최소 8%가 감소한 것을 알 수 있다. Grain size가 커질수록 SPEC CPU의 execution time이 증가하는 것을 알 수 있는데 이는 SPEC CPU의 data locality로 인해 execution time이 감소한 것을 알 수 있다. 본 연구에서는 gem5의 system emulation을 통해 실험을 진행하였다. System emulation은 SPEC CPU 2017을 multi core로 수행을 하지 못하고 이를 위해서는 full system 시뮬레이션을 통해야만 한다. 추후

본 연구를 발전시킨다면, full system 시뮬레이션을 통해 2개 이상의 SPEC CPU 벤치마크를 동시에 수행하면 grain size가 커지더라도 다른 벤치마크에서 나온 normal DRAM command로 인해 data locality가 줄어들고, 따라서 execution time이 늘어날 것으로 예측한다. 또한 grain size가 36이상인 경우는 그래프에 표시하지 않았다. 이는 grain size가 36이상인 경우 normal command(SPEC CPU)의 starvation이 발생하기 때문이다.

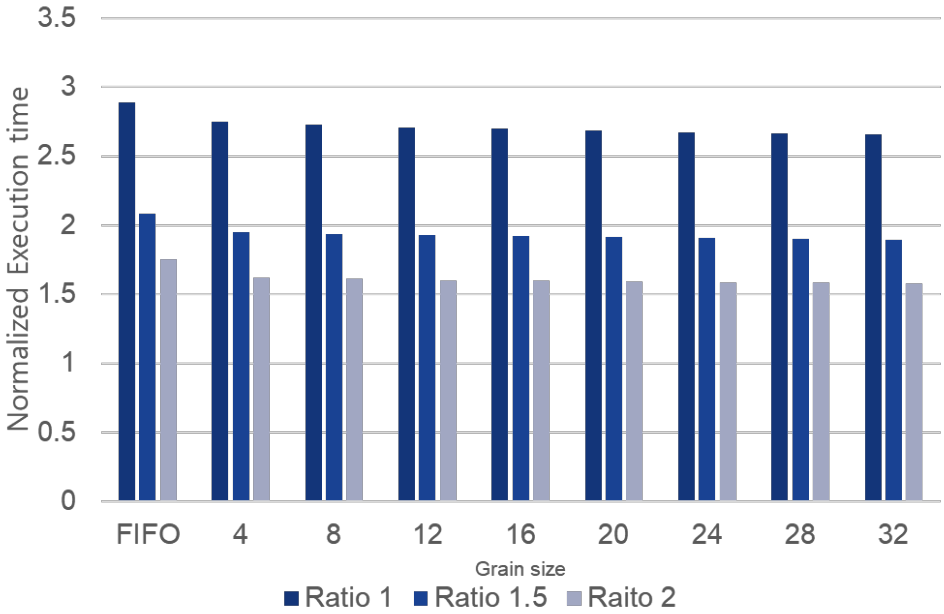

그림 5. 6 Grain size에 따른 SPEC CPU 2017 수행시간

5.2.5 Dynamic grain scheduling policy 효과

그림 5.7은 dynamic grain scheduling policy를 적용했을 때의 SPEC CPU 2017의 execution time 결과를 나타낸다. FIFO는 normal command과 NEWTON command 중에서 먼저 온 command를 우선적으로 처리하는 방식을 의미한다. 그림 5.7의 geomean항목을 통해 FIFO 방식의 execution time이 평균적으로 가장 길고 dynamic grain scheduling 방식이 execution time이 가장 작은 것을 알 수 있다. Dynamic grain command scheduling 방식에서 small grain size는 8, large grain size는 32로 설정하였는데 dynamic grain 방식의 execution time은 8과 32일때의 수행시간보다 모두 낮은 것을 알 수 있다. 또한 Ratio 1에서 수행시간 감소가 가장 큰 것을 알 수 있는데 이는 Ratio 1의 NEWTON command 비율이 가장 높고 이로 인해 policy가 가장 적용이 잘 된 것을 알 수 있다.

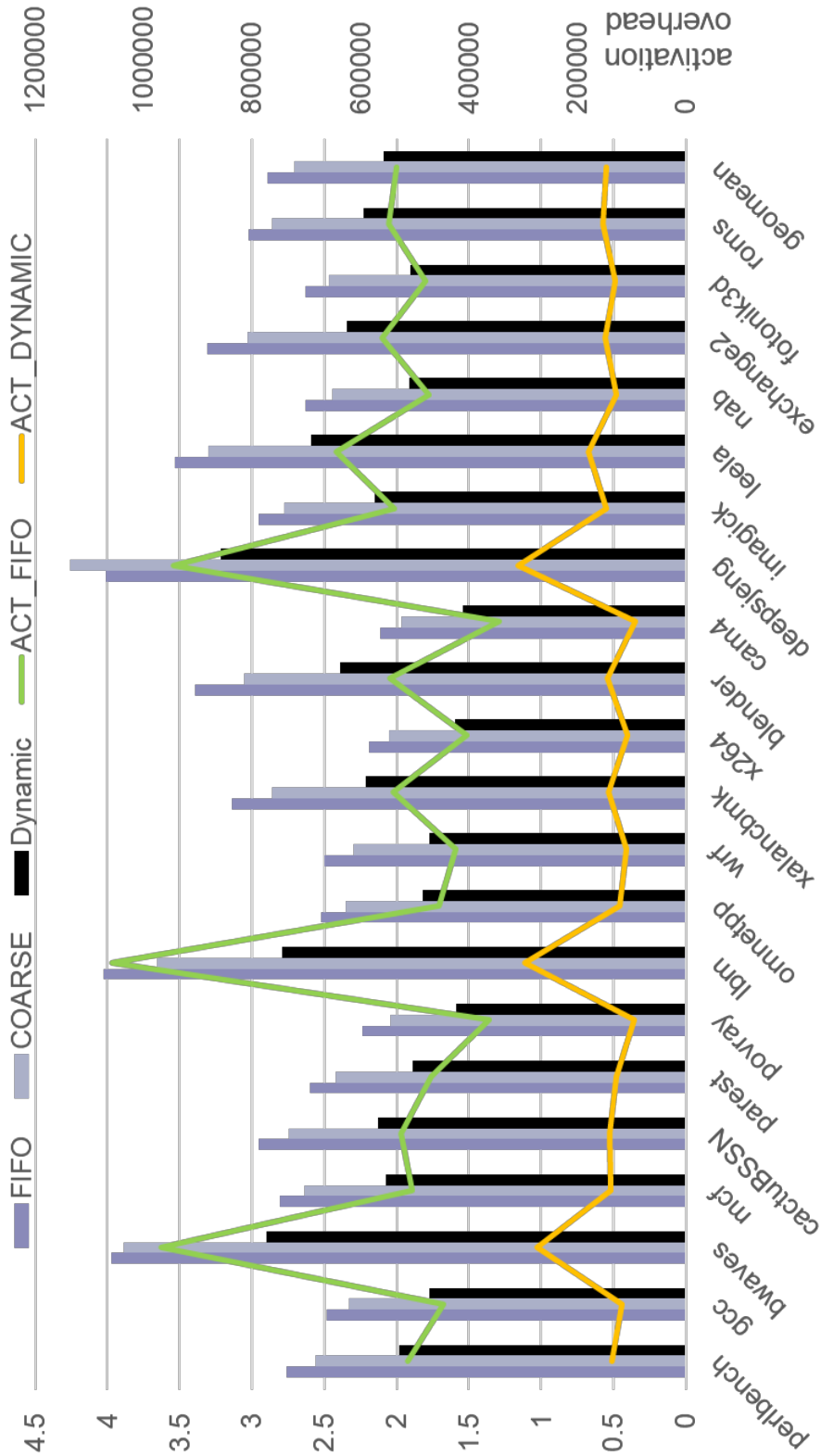

그림 5. 7 Dynamic grain issue policy를 적용했을 때 SPEC CPU 2017 수행시간 변화

5.2.6 NEWTON dynamic grain scheduling policy 효과

그림 5.8은 dynamic grain scheduling policy를 적용하였을 때 각 벤치마크에 따른 policy의 execution time 변화를 나타낸 그래프이다. NEWTON command의 실행시간은 NEWTON command가 memory controller에 들어와서 issue가 되고, NEWTON memory에서 수행이 된 후에 NEWTON command buffer에서 사라졌을 때까지의 시간을 측정한 것이다. 그림 5.8에서 나오는 coarse는 두 개의 command가 있을 경우 NEWTON command에 priority를 준 policy이다. 이 policy를 적용하였을 경우 normal command는 fine grain scheduling policy나 dynamic grain scheduling policy에 비하여 issue가 되는 비율이 낮게되고 반대로 NEWTON command는 coarse 방식에서 가장 나은 성능을 보이게 된다. Dynamic grain scheduling policy를 적용하였을 경우 baseline 방식인 FIFO보다 47% 실행시간이 감소한 것을 볼 수 있다. 반면에 coarse 방식보다는 7% 실행시간이 증가한 것을 볼 수 있는데, normal command의 latency를 감소시키면서 7%만의 실행시간만을 증가한 것은 dynamic grain scheduling policy가 다른 종류의 두 가지 command사이의 scheduling을 잘 했다는 것을 보여준다.

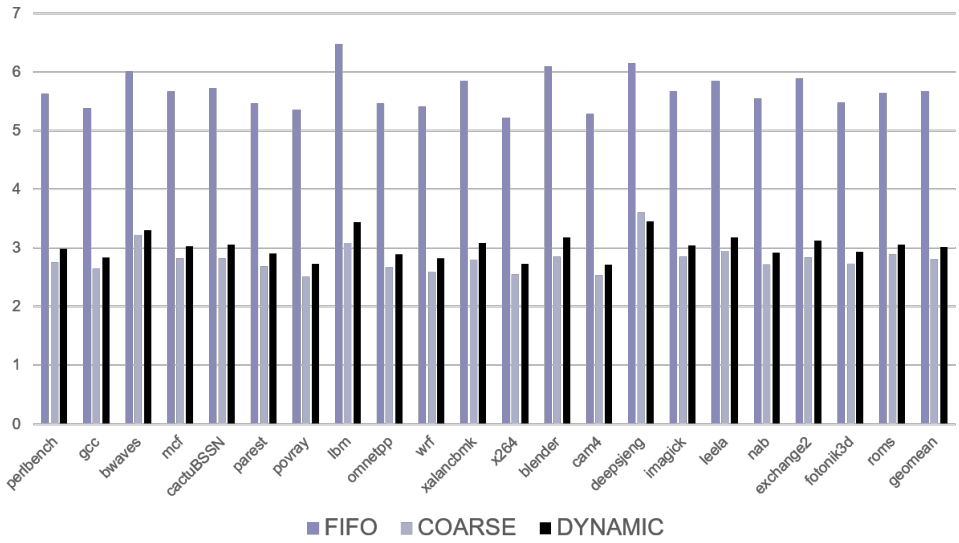

그림 5. 8 NEWTON command execution time 변화

5.2.7 다양한 Dynamic grain scheduling policy 효과

Grain size를 지금까지 large는 32, small은 8로 설정을 하여 execution time을 측정하였다. Large, small grain size는 memory controller를 만드는 제작자에 따라 달라질 수 있다. 필요하다면 grain size state를 나타내는 bit의 수를 현재의 1에서 2나 3으로 늘릴 수 있다. 그렇게 함으로서 grain size가 4개나 8개로 바뀔 수 있다. 더 다양한 grain size는 더 다양한 경우의 NEWTON command 환경에서 적용할 수 있기 때문에 성능 향상을 할 수 있다. 본 장에서는 32와 8로 설정한 grain size를 각각 16과 4로 바꾸어 측정해보았다. 그림 5.9은 grain size를 변화시켰을 때 SPEC CPU 2017의 execution time 변화를 나타낸 그래프이다. 이 그래프에서 execution time은 FIFO policy를 적용하였을 때의 execution time을 1로 정규화하였다. 5.7의 그래프를 보면 일부 벤치마크(lbm, deepsjeng)을 제외한 대부분의 벤치마크에서 large와 small grain size가 32와 8로 설정한 policy가 더 좋은 성능을 보임을 알 수 있다. 이는 grain size를 크게 할수록 normal

comamnd의 locality가 높아지기 때문이다.

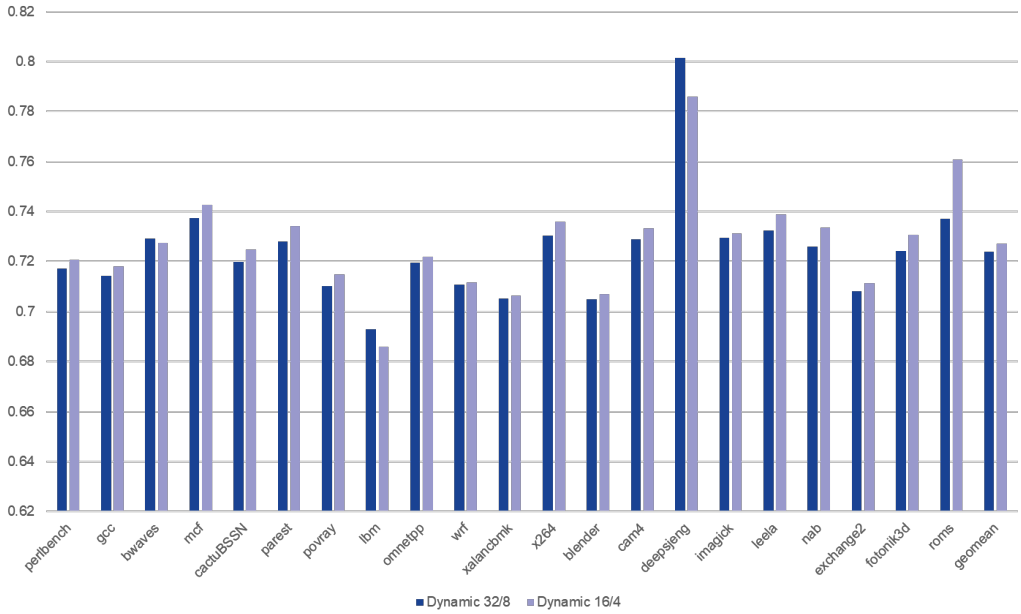

그림 5. 9 Large/small size가 바뀐 경우 execution time 변화

제 6 장 결론

본 논문에서는 host processor의 normal command와 PIM에서의 연산을 수행시키는 NEWTON command가 동시에 접근할 때 NEWTON memory의 구조와 특징을 고려한 PIM controller와 PIM command scheduling policy를 제안하였다. NEWTON memory는 다른 PIM과는 달리 all bank activation overhead가 발생하고 이 overhead가 성능에 미치는 영향이 크다. 따라서 all bank activation overhead를 줄이는게 memory command scheduling policyh에서 중요한 키포인트가 된다. 이를 위해 본 논문에서는 normal command를 grain size만큼 issue를 하는 fine grain scheduling policy를 제안하였다. 또한 grain size에 따른 normal command latency증가와 all bank activation overhead를 줄이기 위하여 grain size를 변화시키는 dynamic grain scheduling policy를 제안하였다. Dynamic grain scheduling policy는 MAC address table을 이용하여 normal command를 다양한 grain단위로 issue하는 PIM command scheduling policy이다. 본 연구에서는 gem5와 DRAMsim3를 이용한 실험을 통하여 dynamic grain scheduling policy는 FIFO(First In First Out)방식에 비해 SPEC CPU 2017은 execution time이 최대 28% 감소한 모습을 확인할 수 있었고, NEWTON command는 최대 47% 감소한 모습을 확인할 수 있었다.

참고 문헌

[1] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan and Yuan Xie, “DRISA: A DRAM-based Reconfigurable In-Situ Accelerator,” in Proc. 50th Annu. Int. Symp. on Microarchitecture (MICRO), 2017, pp. 288–301.

[2] B. Li, L. Song, F. Chen, X. Qian, Y. Chen and H. H. Li, "ReRAM-based accelerator for deep learning," 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, pp. 815-820, doi: 10.23919/DATE.2018.8342118.

[3] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," 2012 Symposium on VLSI Technology (VLSIT), 2012, pp. 87-88, doi: 10.1109/VLSIT.2012.6242474.

[4] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pages 751–764. ACM, 2017.

[5] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 740–753. ACM, 2019.

[6] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, pages 31–44. ACM, 2016.

[7] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In Proceedings of the

46th International Symposium on Computer Architecture, ISCA ’19, pages 629–642, New York, NY, USA, 2019. ACM.

[8] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on, pages 283–295. IEEE, 2015.

[9] M. He et al., "Newton: A DRAM-maker’s Accelerator-in-Memory (AiM) Architecture for Machine Learning," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 372-385, doi: 10.1109/MICRO50266.2020.00040.

[10] B. Y. Cho, Y. Kwon, S. Lym and M. Erez, "Near Data Acceleration with Concurrent Host Access," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 818-831, doi: 10.1109/ISCA45697.2020.00072.

[11] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman,J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov,A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu,V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, “Deep learning recommendation model for personalization and recommendation systems,” CoRR, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[12] S. Li et al., "SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator," 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 696-709, doi: 10.1109/MICRO.2018.00062.

[13] S. Cho, H. Choi, E. Park, H. Shin and S. Yoo, "McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge," in IEEE Access, vol. 8, pp. 135223-135243, 2020, doi: 10.1109/ACCESS.2020.3011265.

[14] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-Enbled Instructions: A

LowOverhead, Locality-Aware Processing-in-Memory Architecture," in ISCA, 2015.

[15] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near data acceleration with concurrent host access," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 818–831.
[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The GEM5 Simulator," SIGARCH Computer Architecture News, vol. 39, no. 2, May 2011.

[17] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, ''DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator,'' IEEE Comput. Archit. Lett., early access, Feb. 14, 2020, doi: 10.1109/ LCA.2020.2973991.

[18] J. Jalle, E. Quiñones, J. Abella, L. Fossati, M. Zulianello and F. J. Cazorla, "A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study," 2014 IEEE Real-Time Systems Symposium, 2014, pp. 207-217, doi: 10.1109/RTSS.2014.23.

[19] T. Mikolov, M. Karafiat, L. Burget, J. ´ Cernock ˇ y, S. Khudanpur. Recurrent neural network ´ based language model, In: Proceedings of Interspeech, 2010.

Abstract

Dynamic Grain Size Control for PIM Command Scheduling

SangHyun Lee
Electrical and Computer Engineering
The Graduate School
Seoul National University

Recently, Deep learning networks have been increasing in number and size of layers to increase performance. In particular, natural language processing (NLP) or recomandation deep learning network causes botleneck in memory bandwidth due to the large size of layers and the small proportion of data reuses.

To address the above issues, the emerging Processing In Memory (PIM) performs operations in or near memory, thereby overcoming the memory bandwidth botleneck possessed by deep learning. A typical example of such PIM is the NEWTON memory. NEWTON memory can function as a main memory for storing data and a PIM for performing operations.

In this paper, we introduce a new PIM command scheduling policy that takes into account the structure and features of the NEWTON memory, when the normal command requesting memory from the host processor and the NEWTON command performing

operations from the PIM are simultaneously accessed. In this study, we describe all bank activation overhead, a problem that arises when the characteristics of the NEWTON memory are not considered, and apply dynamic command scheduling, which uses MAC address table in the proposed PIM controller to determine the normal command by grain.

Experiments showed that the dynamic command method reduced the execution time of SPEC CPU 2017 by up to 28% compared to the conventional FIFO (First In First Out) method, which showed a decrease in execution time compared to the fine-grained method.

Keywords : PIM, memory request scheduling, memory controller
Student Number : 2019-23220