Processing-in-Memory를 위한 효율적인 행렬 연산 기법

김경모○, 이하윤, 신동군
성균관대학교
7bvcxz@gmail.com, lhy920806@gmail.com, dongkun@skku.edu

Efficient Matrix Computation at Processing-in-Memory

Gyungmo Kim, Hayun Lee, Dongkun Shin
Sungkyunkwan University

요 약

최근 빅데이터 처리 및 머신 러닝 등에서 대량의 데이터를 다루는 연산이 증가함에 따라, 메모리 내부에서 데이터 연산을 수행하는 PIM(Processing In-Memory)기법이 활발히 연구되고 있다. 하지만, PIM은 제한된 내부 자원때문에 제한된 형태의 연산만을 지원하고, 각 PIM 장치가 제공하는 고유의 동작 방식과 특성에 맞게 프로그래밍 되어야 한다. 본 논문에서는 최근에 DRAM 업체에 의해서 제안된 PIM 장치를 대상으로 딥 러닝에서 많이 사용되는 행렬 연산을 효율적으로 수행하기 위한 프로그래밍 기법을 제안하고, DRAM 시뮬레이터를 활용하여 PIM으로 인한 연산 성능 향상을 검증한다.

1. 서론

최근 빅데이터 처리 및 머신 러닝을 위한 연산에서 사용되는 데이터의 크기가 점점 커지고 있다. 그 결과, CPU나 GPU에서 연산을 수행하기 위해 메모리로부터 많은 양의 데이터를 읽어와야 하므로 데이터 전송 비용이 연산 비용에 못지않게 커지고 있다. 이러한 비용을 줄이기 위해 연산을 Data 근처에서 수행하여 데이터 이동을 최소화하는 NDP(Near Data Processing)가 큰 주목을 받고 있다. 그 중, PIM(Processing In-Memory)은 메모리 내부에 연산 장치를 추가하여 CPU로의 데이터 이동 없이 메모리 자체에서 연산을 실행하는 NDP 기법 중 하나이다. PIM은 특히 메모리 집약적인 응용을 수행할 때 메모리 대역폭에 의한 병목현상을 피할 수 있다.

PIM은 메모리 내부의 제한된 공간과 전력소모를 고려하여 제한된 기능을 실행하도록 설계된다. 또한, PIM 장치를 개발하는 DRAM 업체마다 고유의 ISA(Instruction Set Architecture)를 사용하며, 프로그램의 실행 방식도 다르다. 그러므로, 이러한 제약사항에 맞춰서 효율적인 연산을 실행하는 소프트웨어 기법이 마련되어야 한다.

본 논문에서는, 머신러닝 Application에서 자주 사용되는 행렬 연산인 ADD와 GEMV 연산을 PIM에서 효율적으로 수행하기 위해, PIM의 특성을 고려한 효율적인 연산 기법을 제안한다.

2. 배경 및 관련 연구

PIM에 대한 연구는 최근에 활발히 진행되고 있어 아직 표준적인 구조와 인터페이스가 정해지지 않았으며, 다양한 형태의 구조가 제안되었다. 특히, HBM(High Bandwidth Memory)에 PIM을 도입하는 여러 연구가 있다. 삼성전자는 Memory 내부에 프로그래밍 가능한 연산장치를 추가한 PIM-DRAM[1]을 제안하였다. 특히, PIM-DRAM은 DRAM Interface 수정없이 동작 가능하도록 DRAM 명령어 기반 PIM 연산 방식을 채택했다. 반면, 하이닉스의 Newton[2]는 DRAM Interface에 Command를 추가하여 PIM의 동작을 수행하며 같은 동작을 수행하는 Command들을 하나의 Command로 묶어서 수행하여 연산에 필요한 총 Command 수를 감소시켰다.

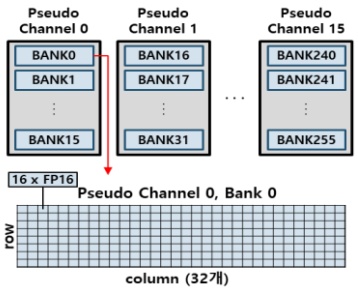
본 논문에서는, 앞서 설명한 PIM 구조 중 대표적으로 PIM-DRAM의 구조를 가정하여, PIM의 특성을 고려한 연산 기법을 제안한다.

이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)

3. PIM-DRAM 구조

3.1 HBM2를 기반으로 설계된 PIM-DRAM

PIM-DRAM은 HBM2를 기반으로 설계되었으며, [그림 1]과 같이 16개의 Pseudo Channel과 각 Pseudo Channel마다 16개의 Bank를 가지고 있다. 각각 2개의 Bank(Even Bank, Odd Bank)에 대한 연산을 담당하는 128(= Bank개수/2)개의 Processing Unit(PU)들이 있다. Host가 보낸 메모리 접근 명령어에 128개의 PU들이 동시에 병렬적으로 자신의 bank 데이터를 처리할 수 있으며, 각 PU는 16개의 16-bit Floating Point(FP16) 연산장치를 가지고 있어서 16xFP16로 구성된 1개의 column 데이터가 vector 연산으로 한 번에 처리될 수 있다.



[그림 1] PIM-DRAM의 Pseudo Channel, Bank 구조

3.2 Processing Unit

각 PU는 PIM 연산에 사용되는 CRF(command register file), GRF(global register file), SRF(scalar register file)라는 Register file들과 FP16 연산을 수행할 수 있는 MUL와 ADD 연산장치를 16개씩 가지고 있다. CRF는 PIM Instruction들로 구성된 Micro-Kernel을 저장하며, PPC(PIM program counter)를 통해서 Micro-Kernel에서 현재 수행할 PIM Instruction의 Offset을 표시한다. GRF와 SRF Register는 연산에 필요한 데이터를 저장하고 있으며, 크기가 32Byte(=

16xFP16)인 Register를 GRF_A, GRF_B가 각각 8개씩 가지고 있고, 크기가 2Byte(=FP16)인 Register를 SRF_A, SRF_M이 각각 8개씩 가지고 있다.

3.3 PIM-DRAM 연산 수행 과정

PIM-DRAM에서는 Single-Bank, All-Bank, All-Bank-PIM mode의 3가지 Bank mode를 통해 서로 다른 Bank-level Parallelism을 제공한다. Single-Bank mode에서는 일반적인 메모리와 같이 DRAM 접근 명령어에 대해서 하나의 bank만 접근되지만, All-Bank mode와 All-Bank-PIM mode에서는 1개의 메모리 접근 명령어에 대해 모든 Bank가 동시에 동작하며, 특히 All-Bank-PIM mode는 추가로 PIM 연산을 수행한다.

PIM 연산을 위해서 아래와 같은 과정을 거친다. ① Single-Bank mode에서 연산 대상인 데이터들을 각 Bank 및 PIM Unit Register에 저장한다. ② All-Bank mode로 전환하여 수행할 Micro-Kernel을 각 PIM Unit의 CRF에 저장한다. ③ All-Bank-PIM mode로 전환하여 Micro-Kernel의 각 PIM Instruction에 알맞은 메모리 접근 명령을 메모리에 전달하여, PIM Instruction을 하나씩 수행한다.

3.4 PIM Instruction

각 PIM Instruction은 수행하고자 하는 PIM 연산 명령어와 DST(Destination)/SRC(Source) 정보로 이루어져 있다.

PIM Instruction의 PIM 연산 명령어로는 데이터를 메모리 혹은 PIM Register(CRF, GRF, SRF)로 이동시키는 **MOV**, 설정한 횟수만큼 명령어들을 반복 수행하여 Loop를 구현하는 **JUMP**, All-Bank-PIM mode를 마치는 **EXIT**, 메모리와 PIM Register에 저장된 Data를 이용하여 다양한 연산을 수행하는 **ADD**, **MUL**, **MAC**, **MAD**(Multiply and Add: X += A x B + C)가 있다.

PIM Instruction의 DST/SRC로는 메모리 BANK와 PIM Register가 가능하다. 메모리 Bank의 주소로는, PIM 명령어를 실행하기 위해서 All-Bank-PIM mode에서 Host가 전송하는 DRAM 명령어내의 Row, Column Address 값을 사용하며, 각 register의 index는 GRF_A[5]와 같이 program code에 직접 기입할 수도 있으며, iteration마다 index가 변경되는 loop내에서는 자동으로 0~7사이의 값으로 변경된다. 그리고, SRF Register를 연산자로 사용할 경우, Processing unit내의 16개 FP16 연산 장치에 동일한 값이 broadcast된다.

4. PIM-DRAM의 연산을 위한 Micro-Kernel 정의

4.1 Micro-Kernel Programming Rule

PIM-DRAM에서 연산을 수행하기 위해서는 각 PIM Unit마다 수행해야 할 PIM 동작을 나타내는 PIM Instruction들인 Micro-Kernel을 정의해야 한다. Micro-Kernel 내부의 PIM Instruction들에 따라, 연산을 수행할 입력 데이터의 분포와 PIM 연산과정이 정해진다.

효율적인 PIM 연산을 위해 입력 데이터 재사용을 극대화할 수 있도록 Micro-Kernel Programming Rule을 정하고, 이에 기반하여 각 연산의 Micro-Kernel을 정의했다. Micro-Kernel Programming Rule은 3가지로, ⓐ Bank mode 전환에 의한 overhead를 최소화하기 위해서, Micro-Kernel을 수행할 때 최대한 많은 입력데이터를 재사용 가능하도록 Micro-Kernel을 구성하는 것. ⓑ 연산 대상인 데이터들을 동일 bank에 배치하는 것. ⓒ 여러 Bank에서의 병렬적 수행을 위해 동시 연산 가능한 데이터를 여러 Bank에 분산 저장하는 것이다.

4.2 ADD Micro-Kernel

Vector Z = Vector X + Vector Y 연산을 PIM을 이용하여 수행하기 위해서 아래 ADD Micro-Kernel을 사용할 수 있다.

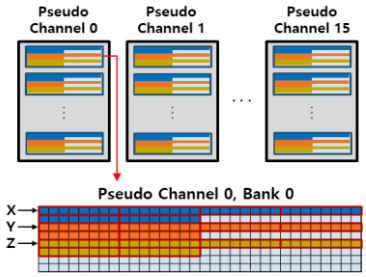
ADD_ukernel

```
0 : MOV   GRF_A  BANK          // GRF_A[0] ~ GRF_A[7]에 Vector
1 : JUMP  -1     7             // X[i] ~ X[i + 7] 값을 저장한다.
2 : MAC   GRF_A  BANK  SRF_M   // GRF_A[0] ~ GRF_A[7]와 Vector
3 : JUMP  -1     7             // Y[i] ~ Y[i + 7]를 더해서
                               // GRF_A[0] ~ GRF_A[7]에 저장한다.
4 : MOV   BANK   GRF_A         // GRF_A[0] ~ GRF_A[7]의 값을
5 : JUMP  -1     7             // Z[i] ~ Z[i + 7]에 저장한다.
6 : EXIT
```

[그림 2] ADD_ukernel 정의

1번 행의 JUMP 명령어의 경우, PPC=PPC-1 동작을 총 7번 반복하는 PIM Instruction으로 결과적으로 0번 행의 MOV을 8번 수행하게 된다. SRF_M은 MAC연산을 수행하기 위해 1로 초기화하여 연산한다.

[그림 3]의 경우에, Vector X, Y, Z의 크기가 384KB(= 256 x 48 x 16 x FP16)일 경우의 Data Partition을 나타낸다. 각 Bank에 48 x 16 x FP16만큼의 데이터를 분산 저장하여 128개의 PU에서 병렬 처리가 가능하도록 한다. [그림 2]의 ADD_ukernel을 한번 수행하여 X, Y의 16xFP16 데이터 8개를 연산할 수 있으므로, 전체 수행을 위해 ADD_ukernel이 6번(=48/8) 실행되어야 한다.



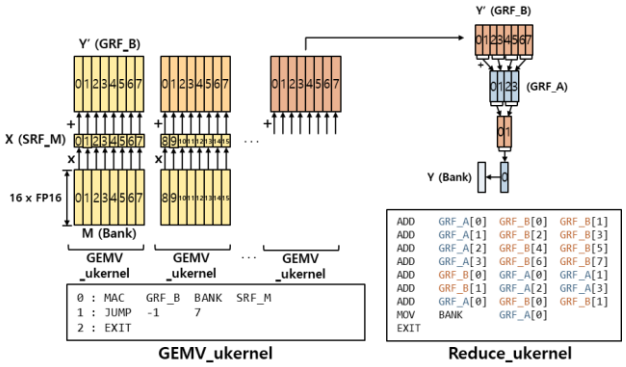
[그림 3] PIM의 ADD 연산을 위한 입력 데이터 분배

4.3 GEMV Micro-Kernel

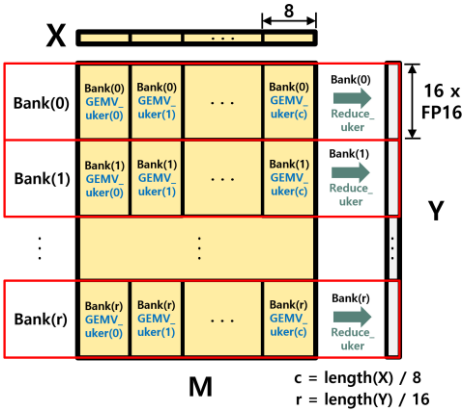
Vector Y = GEMV(Matrix M, Vector X) 연산은 GEMV Kernel 및 Reduce Kernel이 필요하다. Adder tree를 제공하는 Newton[2]과 달리 PIM-DRAM[1]에서는 하드웨어 공간을 고려하여 Adder tree를 제공하지 않기 때문에, GEMV 연산에서 Row 방향 누적을 수행하기 위한 소프트웨어 기반 Reduce 연산이 필요하다.

[그림 4]와 같이 우선 입력 Vector X는 SRF_M 레지스터에 저장하고, 입력 Matrix M은 메모리 Bank에서 읽어오며 결과를 GRF_B 레지스터에 계속 누적한다. 16개 row의 8개 element에 대한 연산을 GEMV_ukernel로 수행하고 이를 여러 번 반복하여 수행하면 같은 row에서 stride 8에 대한 값을 누적할 수 있다. (e.g., GRF_B[2]은 16개 Row의 Column 2, 10, 18, 26, ..의 연산 값이 누적 저장된다) 그리고, GRF_B Register의 값을 Reduce_ukernel을 통해 누적하여 결과적으로 Row에 대한 누적을 수행할 수 있다. [그림 5]와 같이, 각 Bank 마다 16개의 Row에 대해서 [그림 4] 동작을 수행하여 전체 GEMV 연산을 수행한다. Matrix M을 4096(=256개 Bank x 16개 Row)개 Row 연산으로 분할하고, 각 Matrix 부분을 Transpose하여, Bank, Row, Column의 순서로 Interleave하여 16xFP16씩 저장하면 (Row0, Col0, BA0~BA255 → Row0, Col1, BA0~BA255 → …) 각 Bank는 각 Matrix 부분의 16개 Row에 대한 Matrix M 값을 가진다. 이후, Micro-Kernel에

맞게 알맞은 Vector X 값을 각 PU에 전달하여 연산을 수행한다.



[그림 4] GEMV ukernel, Reduce ukernel 정의



[그림 5] 각 Bank에서 병렬로 GEMV 연산을 같이 수행

5. 실험

5.1. 실험환경

메모리 특성을 고려한 Cycle-Accurate DRAMsim3[3]를 PIM 동작을 수행할 수 있도록 수정하여 시뮬레이터를 구현하였다. 본 시뮬레이터를 이용하여, 메모리 명령에 의한 PIM 동작을 시뮬레이션 및 메모리 명령들을 수행하는데 사용된 Bus Clock 횟수를 측정할 수 있다.

메모리의 Timing 관련 값은 PIM-DRAM의 HBM2를 기반으로 설정하였으며, 메모리 구조는 16개의 Pseudo Channel과 각 Pseudo Channel마다 16개의 Bank로 구성하였다. 총 PIM Unit의 개수는 128개이며, Burst Size는 32Byte, Bus Clock Frequency는 250MHz로 설정한다.

CPU의 연산시간은 연산처리 시간이 메모리로부터 데이터를 읽고 쓰는 시간에 의해 Overlap된다고 가정하여 연산 대상 데이터를 읽고 쓰는 시간으로 계산한다. PIM의 연산시간은 연산을 수행하기 위한 Bank mode 전환, Micro Kernel Program, PIM 연산에 필요한 시간을 합한 것으로 정한다. 연산 대상 데이터는 임의의 값을 가지며, 메모리에 저장되어 있다고 가정한다.

추가로, DRAM은 효율적으로 메모리 명령을 수행하기 위해서 전달된 메모리 명령의 순서가 뒤바꾸어 수행될 수 있다. 이에 의한 PIM 연산의 오동작을 막기 위하여, 8개의 메모리 명령마다 앞선 8개의 메모리 명령이 수행 완료됨을 기다려서 8개의 메모리 명령 씩 수행하도록 분리하는 Barrier를 추가했다. Barrier는 이전 8개의 메모리 명령들이 수행 완료될 때까지 Bus Clock을 증가시키며, 메모리 명령을 처리한다. 분리된 8개의 메모리 명령들은 순서가 뒤바뀌더라도 올바르게 PIM 연산을 수행하도록 동작한다.

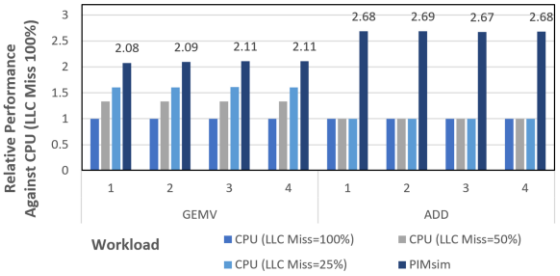
5.2 실험방법

실험을 위한 연산은 아래 [표 1]의 워크로드를 사용하였다. CPU, PIM-DRAM 각각 연산을 수행하기 위해 각 워크로드에 필요한 메모리 명령들을 순차적으로 전달하는 Application을 구현하여, 메모리 명령을 DRAMsim3에게 전달하도록 한다. 그리고, DRAMsim3는 메모리 명령들을 Trace 방식으로 처리하고, 메모리 명령들을 수행하기 위한 총 Bus Clock 횟수를 출력 값으로 반환한다. 그리하여, 총 연산시간은 Bus Clock 횟수 x (1/Bus Clock Frequency)로 계산하였다. Cache Hit에 의한 CPU 성능을 고려하기 위해 다양한 LLC (Last Level Cache) Miss Rate의 CPU와 성능을 비교하였다. LLC Miss Rate의 경우 Application의 전체 메모리 명령 중에서 Miss Rate 비율만을 사용하여 시뮬레이션 한다.

Name	Dimension(input x output)	Name	Dimension
GEMV1	1k x 4k	ADD1	2M
GEMV2	2k x 4k	ADD2	4M
GEMV3	4k x 8k	ADD3	8M
GEMV4	8k x 8k	ADD4	16M

[표 1] 워크로드

5.3. 실험



[그림 6] Workload별 CPU와 PIM의 수행 Cycle 수

LLC Miss Rate가 감소함에 따라 Cache에 의한 GEMV의 Input Vector 재사용량이 증가하기 때문에 CPU 성능이 높아지는 반면, ADD는 모든 연산자가 재사용되지 않기 때문에 CPU 성능이 동일한 결과를 보여준다. Bank-level Parallelism과 PIM 구조에 효율적인 연산 기법에 의해 모든 Workload에 대해서 PIM이 CPU에 비해 월등히 적은 Cycle 수로 Workload를 수행하며, ADD, GEMV 연산에서 각각 CPU 대비 평균 2.68배, 1.6배의 성능개선을 보여준다.

6. 결론 및 향후계획

본 연구에서는 PIM의 특성을 고려한 효율적인 행렬 연산 기법을 제안하고 시뮬레이션을 통해 CPU 대비 높은 성능을 제공하는 것을 보였다. 본 연구를 참조하여, 다른 PIM 장치에서의 연산 기법을 제안할 수 있을 것으로 기대하며, 향후 Full ML Application을 PIM으로 수행하는 연산을 설계 및 평가할 계획이다.

참고문헌

[1] S. Lee et al, “Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology”, *In: International Symposium on Computer Architecture, ISCA, 2021.*

[2] M. He et al, “Newton: A DRAM-maker’s Accelerator-inMemory (AiM) Architecture for Machine Learning,” *In: Int’l Symp. on Microarchitecture (MICRO). 2020.*

[3] University of Maryland, Memory Systems Research, https://github.com/umd-memsys/DRAMsim3