Processing-in-Memory를 위한 코드 생성 및 데이터 레이아웃 변형 기법

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

Code Generation and Data Layout Transformation for Processing-in-Memory

Hayun Lee°, Gyungmo Kim, Dongkun Shin
Department of Electrical and Computer Engineering, Sungkyunkwan University

요 약

Processing-in-Memory (PIM)은 메모리 내부의 병렬성과 대역폭을 활용하여 메모리 집약적인 연산에서 CPU 또는 GPU와 비교하여 좋은 성능을 달성한다. 그러나, 다양한 PIM 구조가 제안된 것에 비해 PIM 컴파일러에 대한 연구는 부족한 상황이다. 다양한 PIM의 구조를 고려하여 코드를 생성하기 위해서는 일반적인 PIM 스케줄 프리미티브와 PIM 메모리에 데이터를 저장하는 레이아웃을 고려해야 한다. 본 논문에서 제안하는 PIM 컴파일러는 일반적인 PIM 구조를 정의하여 이러한 고려 사항을 해결하고, 추가로 레지스터 재사용 최적화를 통해 다양한 GEMV 연산에서 최대 5.75배 성능 개선을 달성한다.

1. 서 론

딥 러닝은 자연어 처리, 음성 인식 등 다양한 분야에서 사용되고 있으며, 높은 정확도를 달성하기 위해 DNN의 크기는 점점 커져 왔다. 이와 더불어 DNN에서 많은 부분을 차지하는 RNN 레이어, 임베딩 룩업 등의 연산은 큰 메모리 요구량에 비해 연산량이 적어 기존 폰 노이만 구조의 연산 장치에서는 메모리 대역폭의 한계로 최대 성능을 끌어내지 못하는 문제가 있다.

최근 이와 같은 한계를 극복하기 위해 메모리 내부에서 연산하는 Processing-In-Memory (PIM)이 제안되었다. PIM은 메모리 내부의 병렬성 및 대역폭을 활용하여 기존 가속기와 비교하여 memory-bound 연산에서 낮은 전력 소모와 빠른 실행 시간을 달성한다. 그러나, 다양한 PIM 구조가 제안된 것과 달리 여전히 PIM 코드 생성을 위한 컴파일러 연구는 부족한 상황이다.

PIM 코드를 생성하기 위해서는 두 가지 고려 사항이 존재한다. 먼저, 다양한 PIM의 인터페이스를 고려하여 일반적인 PIM 스케줄 프리미티브를 정의해야 한다. 또한, PIM은 연산 시 데이터의 저장된 위치가 중요하기 때문에 PIM 특성을 고려하여 데이터 레이아웃을 변형하여 저장할 필요가 있다.

본 논문에서 제안하는 PIM 컴파일러는 앞서 언급한 두 개의 고려 사항을 해결하였으며, 추가 성능 개선을 위해 레지스터 재사용 최적화를 수행한다. 실험에서는 기존에 제안된 PIM-HBM[1]에서 최적화된 코드 생성만으로 최대 5.75배 성능 개선을 보여준다.

이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)

2. 배 경

2.1 Processing-in-Memory

최근 다양한 분야에서 메모리 대역폭에 따른 성능 저하 문제를 해결하기 위해 다양한 PIM 구조가 제안되었다. 특히, 본 논문에서는 메모리 주소를 포함한 요청으로 GEMV 연산을 수행하는 PIM[1, 2]을 대상으로 하며 그림 1과 같은 구조를 갖게 된다.

이러한 구조에서는 호스트에서 보내는 하나의 메모리 요청을 통해 하나의 DRAM 채널 안의 모든 PIM 프로세서가 메모리 주소를 기반으로 뱅크와 레지스터의 데이터를 피연산자로 사용하여 연산을 수행한다. 글로벌 버퍼는 보통 입력 벡터의 값을 저장하며, PIM 구조에 따라 존재하지 않을 수 있다. 본 논문에서는 글로벌 버퍼, 레지스터로 고려하지 않고, 통일성을 위해 입력 레지스터와 출력 레지스터로 정의한다. 예를 들어, PIM-HBM[1]에서는 GRF_A 또는 SRF_M을 입력 레지스터로, GRF_B를 출력 레지스터로 정의할 수 있다. 또한, PIM 구조에 따라 각 레지스터의 종류는 스칼라 또는 벡터가 될 수 있다. 스칼라의 경우 1개의 데이터가 모든 연산에 사용되며, 벡터는 DRAM 칼럼의 크기만큼 데이터를 가지고 있다.

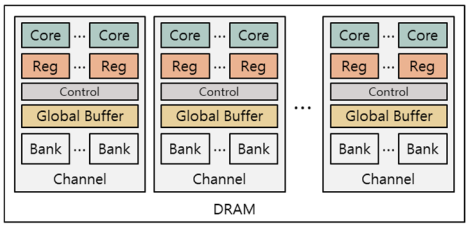


그림 1 일반적인 뱅크 단위의 PIM 구조

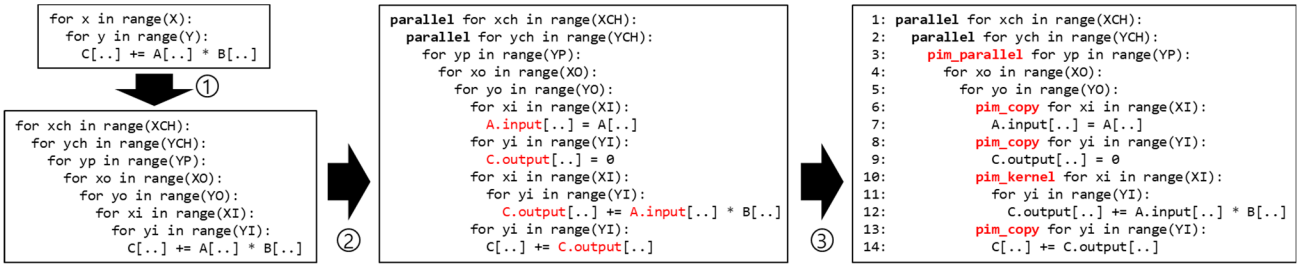


그림 2 PIM-HBM[1] 코드 생성을 위한 GEMV 스케줄링 템플릿

2.2 딥 러닝 컴파일러

딥 러닝 컴파일러는 다양한 연산 및 하드웨어에 대한 코드 생성을 효율적으로 수행하기 위해 제안되었다. 특히, TVM[3]은 다양한 딥 러닝 프레임워크에서 정의하는 모델을 입력으로 받아 그래프 수준 최적화와 연산자 수준 최적화를 수행하며, 최종적으로 각 장치 별 최적화를 통해 코드를 생성한다. TVM에서는 연산자 수준 최적화 과정에서 연산의 정의와 스케줄 최적화를 분리하며, 스케줄 최적화를 위해 미리 정의된 스케줄 프리미티브 (schedule primitive)를 제공한다. 스케줄 프리미티브를 통해 정의된 연산에 대해 루프 블로킹 (loop blocking) 수행 및 메모리 할당, 데이터플로우 (dataflow) 등을 변형할 수 있다.

기존에도 PIM 컴파일러[4]는 제안되었지만, 특정 PIM 구조에 최적화되어 있으며 메모리 레이아웃에 대한 고려가 거의 없다.

3. PIM 컴파일러

3.1 PIM 스케줄 프리미티브

본 논문에서 제안하는 PIM 컴파일러는 TVM을 기반으로 한다. 그림 2는 PIM-HBM[1] 구조에서 입력 벡터 A와 가중치 매트리스 B에 대해 출력 벡터 C를 구하는 GEMV 연산의 코드 생성을 위한 스케줄링 과정을 보여준다.

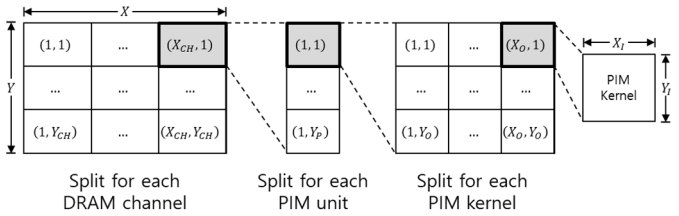


그림 3 PIM을 위한 GEMV 연산 타일링

DRAM 채널과 DRAM 채널당 PIM의 개수는 각각 $N_{CH}(= X_{CH} \times Y_{CH})$, $N_P(= Y_P)$ 이고, PIM 커널은 한 번에 $X_I \times Y_I$ 의 데이터를 연산할 때, ①은 루프 변환 스케줄 프리미티브를 통해 GEMV 전체 연산을 그림 3과 같이 분할하는 과정이다. 분할은 가중치 매트리스를 기반으로 수행하며, 먼저 각 DRAM 채널을 위해 $X \times Y$ 매트리스에 대해 X_{CH}와 Y_{CH}로 나눈다. 그 다음 각 PIM 유닛을 위해 Y_P로 나눈다. 마지막으로 각 PIM 커널을 위해 X_O와 Y_O로 나눈다. 이 때, 각 PIM 커널에 대한 크기는 $X_I \times Y_I$가 된다. X_O와 Y_O 루프의 순서는 바뀔 수 있으며, 그림 2는 X_O가 바깥쪽에 있는 경우이다. 앞서 설명한 내용에 따라 각 변수는 다음과 같은 조건을 갖게 된다.

$$X = X_{CH} \times X_O \times X_I, \quad Y = Y_{CH} \times Y_P \times Y_O \times Y_I$$

PIM 커널을 수행하기 위해서는 입력 벡터와 출력 벡터를 각각 PIM 레지스터로 쓰고, 읽는 과정이 필요하다. 이를 위해, ②는 캐시 스케줄 프리미티브를 통해 입력 레지스터(**input**)와 출력 레지스터(**output**)를 사용하도록 지정한다. 또한, **parallel**을 통해 각 DRAM 채널마다 서로 다른 thread로 수행되도록 할 수 있다.

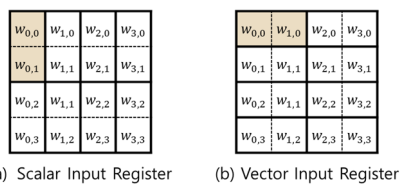
본 논문에서는 ②까지 완료된 스케줄로부터 PIM을 위한 코드를 생성하기 위해 3가지 PIM 스케줄 프리미티브를 제안한다. 먼저, **pim_parallel**은 PIM의 뱅크 수준 병렬성을 고려하여 해당 루프(3줄)를 Y_P번이 아닌 한 번만 수행하도록 변형한다. 단, 내부 루프들(4~14줄) 중 뱅크 수준 병렬성을 활용하지 못하는 루프(13~14줄, 출력 레지스터 값 읽기)의 경우 Y_P 번 수행이 필요하므로 루프를 추가(12줄과 13줄 사이)한다. **pim_copy**는 루프 내의 캐시의 이름을 보고 PIM 레지스터로 읽거나 쓰는 대상 PIM의 함수로 대체한다. 마지막으로, **pim_kernel**은 $X_I \times Y_I$ GEMV 연산을 수행하는 대상 PIM의 함수로 대체한다.

3.2 2단계 데이터 레이아웃 변형

PIM 컴파일러는 코드 생성과 함께 PIM을 고려하여 데이터의 레이아웃을 미리 변형 후 저장하는 과정도 중요하다. 왜냐하면, PIM 연산의 최소 단위는 DRAM 칼럼의 크기이며, PIM 커널 수행 시 계산에 사용되는 데이터의 주소를 통해 입력 및 출력으로 사용되는 레지스터의 인덱스를 계산하기 때문이다. 따라서, PIM 커널이 올바르게 수행되도록 데이터를 저장하기 위해서는 PIM 레지스터의 특성 및 데이터 플로우를 고려해야 한다. 본 논문에서는 DRAM 칼럼 하나에 저장되는 데이터들을 데이터 블록이라 정의한다.

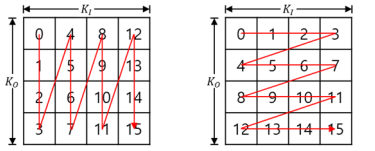
데이터 레이아웃의 변형은 2단계로 수행한다. 1단계에서는 하나의 데이터 블록 안에 데이터를 어떻게 배치할지를 결정하고, 2단계에서는 데이터 블록 간의 순서를 결정한다.

그림 4는 하나의 DRAM 칼럼에 데이터가 2개 들어간다고 가정할 때, 입력 레지스터 종류에 따른 데이터 블록을 생성하는 1단계 과정을 보여준다. 해당 예제에서는 총 16개의 데이터로부터 8개의 데이터 블록을 생성하고 있다. 입력 레지스터가 스칼라일 경우 하나의 입력이 주어지므로 여러



(a) Scalar Input Register (b) Vector Input Register

그림 4 1단계: PIM GEMV 커널의 입력 레지스터 종류에 따른 데이터 블록 생성 방법



(a) Input Stationary (b) Output Stationary

그림 5 2단계: PIM GEMV 커널의 데이터플로우에 따른 데이터 블록 배치 방법

출력에 대한 계산을 수행해야 하므로 (a)와 같이 서로 다른 출력에 대한 가중치들을 하나의 데이터 블록으로 생성하게 된다. 입력 레지스터가 벡터일 경우 여러 입력에 대한 계산을 수행해야 하므로 (b)와 같이 서로 다른 입력에 대한 가중치들을 하나의 데이터 블록으로 생성하게 된다.

그림 5는 PIM 커널의 데이터플로우를 통해 데이터 블록의 저장하는 순서를 결정하는 2단계를 보여준다. 여기서 K_I 와 K_O는 각각 PIM 커널에서 사용하는 입력 및 출력 레지스터의 개수를 의미한다. 그림 5의 (a)와 (b)는 각각 Input Stationary (IS)와 Output Stationary (OS) 데이터플로우에 대한 데이터 배치 순서를 보여준다. 만약, IS 커널에서 OS 방식과 같이 가중치를 배치할 경우 잘못된 연산 결과를 얻게 된다.

3.3 PIM 레지스터 재사용 최적화

그림 2에서 레지스터의 값이 재사용되는 것을 고려하지 않을 경우 입력 레지스터에 같은 값을 덮어쓰게 하거나, 출력 레지스터의 부분합을 미리 가져오게 된다. 따라서, PIM 코드 생성 시 **pim_copy**에서 현재 입력(출력) 벡터의 인덱스가 마지막 입력(출력) 벡터의 인덱스와 다른 경우에만 PIM 입력(출력) 레지스터에 관해 쓰기(읽기)를 수행한다.

4. 실 험

4.1 실험 환경

본 논문에서 제안하는 PIM 컴파일러는 TVM[3]을 기반으로 개발하였다. 실험을 위해 PIM-HBM[1] 논문의 설정과 동일한 PIM을 가정하였다. (단, 실험의 편의를 위해 뱅크마다 PIM 프로세서가 있다고 가정하였다) PIM 컴파일러를 통해 코드를 생성하고, DRAMsim3[5]로 구현한 시뮬레이터로 실행 cycle을 측정하였다. 실험은 기존 PIM-HBM 논문에 나와있는 GEMV 커널 ($K_I = 8, K_O = 1$) 수행 시의 성능을 기준으로 코드 생성 결과와 레지스터 재사용 최적화까지 적용하였을 때의 성능 결과를 비교하였다. 사용 가능한 PIM 커널의 레지스터 개수는 $K_I, K_O \in \{1, 2, 4, 8\}$와 같이 설정하였다.

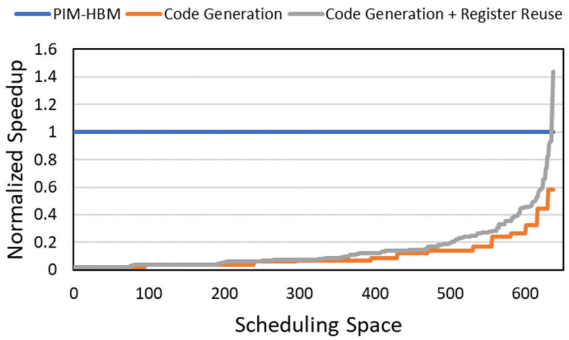


그림 6 1024x2048 GEMV PIM 코드 생성을 위한 전체 스케줄링 공간 성능 비교

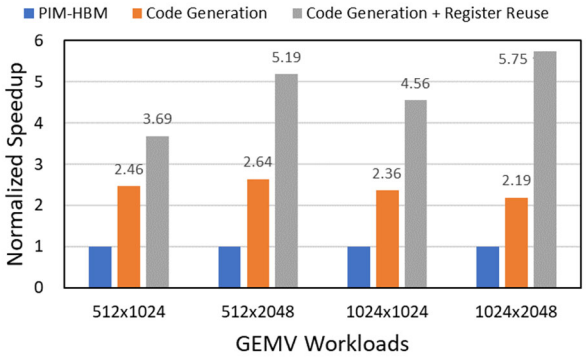


그림 7 PIM 코드 생성 및 최적화에 따른 성능 비교

4.2 실험 결과

먼저 PIM 컴파일러로 스케줄링할 수 있는 전체 공간의 성능을 분석하는 실험을 진행하였다. 그림 6은 1024x2048 GEMV 연산에 대하여 PIM 컴파일러를 통해 생성할 수 있는 전체 코드들의 성능을 오름차순으로 정렬한 결과를 보여준다. 단순하게 코드를 생성하였을 경우 PIM-HBM보다 성능이 항상 낮은 것을 볼 수 있는데, 이는 PIM-HBM에서 기본적으로 레지스터 재사용을 하도록 코드가 수행되기 때문이다. 또한, 레지스터 재사용 최적화를 수행할 경우 기존 PIM-HBM보다 더 높은 성능을 달성하는 코드를 생성하는 것을 확인할 수 있다. 이는 더 효율적인 PIM 커널을 사용하기 때문이다.

그림 7은 다양한 GEMV 연산에 대해 코드 생성 및 레지스터 재사용 최적화까지 수행했을 때의 성능 개선을 보여준다. 단순 코드 생성의 경우 최소 2.19배에서 최대 2.64배까지 성능이 개선되는 것을 확인할 수 있다. 이는, 효율적인 PIM 커널을 통해 연산을 수행하였기 때문이다. 레지스터 재사용 최적화까지 수행할 경우 최소 3.69배에서 최대 5.75배까지 성능이 개선된다.

5. 결 론

본 논문에서는 다양한 PIM을 위한 코드를 생성하는 PIM 컴파일러를 제안한다. 해당 PIM 컴파일러는 기본적인 PIM 구조를 정의하고, PIM 스케줄 프리미티브를 통해 PIM 코드를 생성한다. 또한, PIM 연산 장치와 PIM 커널의 특성을 고려하여 메모리 레이아웃 변형을 수행하며, 레지스터 재사용 최적화를 통해 최적화된 코드 생성을 수행한다. 본 논문에서 제안한 PIM 컴파일러는 기존 PIM-HBM과 비교하여 최대 5.75배의 성능 개선을 달성한다.

참고 문헌

[1] Lee, Sukhan, et al. "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product." 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021.

[2] He, Mingxuan, 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). IEEE, 2020.

[3] Chen, Tianqi, et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.

[4] Gu, Peng, et al. "iPIM: Programmable in-memory image processing accelerator using near-bank architecture." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[5] Li, Shang, et al. "Dramsim3: a cycle-accurate, thermal-capable dram simulator." IEEE Computer Architecture Letters 19.2 (2020): 106-109.