ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 50, No. 8, pp. 639-645, 2023. 8
https://doi.org/10.5626/JOK.2023.50.8.639

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

(Code Generation and Data Layout Transformation Techniques for Processing-in-Memory)

이 하 윤 † 김 경 모 †† 신 동 군 †††
(Hayun Lee) (Gyungmo Kim) (Dongkun Shin)

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. 또한, PIM과 호스트 사이에 발생하는 데이터 이동을 최소화해야 한다. 본 논문에서 제안하는 PIM 컴파일러는 일반적인 PIM 구조를 정의하여 이러한 고려 사항을 해결하고, 추가로 레지스터 재사용 최적화를 통해 다양한 GEMV 연산에서 최대 2.49배 성능 개선을 달성한다.

키워드: Processing-in-Memory (PIM), 딥 러닝, 컴파일러, 코드 생성

Abstract Processing-in-Memory (PIM) capitalizes on internal parallelism and bandwidth within memory systems, thereby achieving superior performance to CPUs or GPUs in memory-intensive operations. Although many PIM architectures were proposed, the compiler issues for PIM are not currently well-studied. To generate efficient program codes for PIM devices, the PIM compiler must optimize operation schedules and data layouts. Additionally, the register reuse of PIM processing units must be maximized to reduce data movement traffic between host and PIM devices. We propose a PIM compiler, which can support various PIM architectures. It achieves up to 2.49 times performance improvement in GEMV operations through register reuse optimization.

Keywords: Processing-in-Memory (PIM), deep learning, compiler, code generation

· 이 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)
· 이 논문은 2022 한국컴퓨터종합학술대회에서 'Processing-in-Memory를 위한 코드 생성 및 데이터 레이아웃 변형 기법'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 성균관대학교 전자전기컴퓨터공학과 학생
lhy920806@skku.edu
†† 비 회 원 : 성균관대학교 반도체디스플레이공학과
7bvcxz@skku.edu
††† 종신회원 : 성균관대학교 전자전기컴퓨터공학과 교수
(Sungkyunkwan Univ.)
dongkun@skku.edu
(Corresponding author임)

논문접수 : 2022년 9월 19일
(Received 19 September 2022)
논문수정 : 2023년 5월 19일
(Revised 19 May 2023)
심사완료 : 2023년 5월 29일
(Accepted 29 May 2023)

정보과학회논문지 제50권 제8호(2023. 8)

1. 서 론

딥 러닝은 자연어 처리, 음성 인식 등 다양한 분야에서 사용되며, 높은 정확도를 달성하기 위해 DNN(Deep Neural Network)의 크기는 점점 커지고 있다. 이 과정에서 RNN 레이어와 임베딩 룩업 같은 연산이 DNN에서 큰 비중을 차지하게 되는데, 이들은 메모리 접근이 많은 메모리 중심의 연산으로, 연산량 대비 메모리 접근 비율이 높다. 따라서, 전통적인 폰 노이만 구조를 사용하는 연산 장치에서는 메모리 대역폭의 제한으로 인해 이러한 연산을 수행할 때 연산 장치의 성능을 충분히 활용하지 못하는 문제가 발생한다.

이를 해결하는 방안으로 최근에는 메모리 내에서 바로 연산을 수행함으로써 호스트와 메모리 간의 데이터 이동을 줄이는 Processing-In-Memory(PIM)이 제안되었다. PIM은 메모리 내부의 병렬성과 대역폭을 활용하여 기존 CPU 또는 GPU보다 메모리 집약적인 연산에서 낮은 전력 소모와 빠른 실행 시간을 가능하게 한다.

PIM은 연산이 수행되는 위치에 따라 분류할 수 있으며, 감지 증폭기(sense amplifier) 이전에 연산을 수행하는 경우를 아날로그(analog) PIM, 감지 증폭기 이후에 연산을 수행하는 경우를 디지털(digital) PIM이라고 한다. 아날로그 PIM은 노이즈, 확장성, 값 정확도 등에 대한 문제로 인해 실제 상용화에 여러 제약이 따른다. 이러한 이유로 본 논문에서는 아날로그 PIM 대신 디지털 PIM을 중점적으로 다루며, 특히 뱅크 근처 디지털 PIM[1,2]을 대상으로 한다. 이는 딥 러닝의 핵심 연산인 매트릭스-벡터 곱셈을 최대한 병렬적으로 처리하기 위해 DRAM 뱅크 내 또는 주변에 위치해야 하기 때문이다[2]. 뱅크 근처 디지털 PIM에 해당하는 PIM 구조로는 삼성의 PIM-HBM[1], SK 하이닉스의 Newton[2] 등이 있다.

이러한 PIM은 단일 DRAM 커맨드로 여러 뱅크에서 동시에 연산을 수행하며, 이를 통해 뱅크 수준의 병렬성을 활용하여 높은 연산 성능을 달성한다. 하지만 제한된 공간과 저전력 소비를 고려하여 설계되어야 하므로, 이들은 일반적으로 제한된 Instruction Set Architecture (ISA)를 가진 프로세서[1] 또는 단순한 로직[2] 형태로 구현되어 있다. 따라서 연산 수행을 포함하여 연산 배분, 데이터 이동 등의 모든 제어는 호스트 프로세서가 담당해야 한다. 결과적으로 PIM의 처리량은 호스트 측 코드의 실행에 의존적이므로, 이런 코드를 효율적으로 생성 가능한 PIM 컴파일러의 필요성이 대두된다.

과거에도 PIM 컴파일러[3,4]에 대한 연구가 진행되었으나, 대부분은 HMC(Hybrid Memory Cube) 기반의 PIM을 대상으로 하였으며, HMC 기반 PIM에서 발생하는 볼트(vault) 간의 또는 볼트 내의 데이터 이동 최소화에 중점을 두었다. 하지만, 본 논문에서 대상으로 하는 PIM은 PIM 내부의 PIM 유닛 간의 데이터 이동이 불가능하며, 모든 데이터 이동은 호스트를 거쳐야 하는 특성이 있다. 이러한 특성 때문에, PIM과 호스트 간의 데이터 이동을 최소화하는 PIM 컴파일러를 새롭게 개발할 필요성이 있다.

이러한 PIM을 위한 컴파일러를 개발하려면 몇 가지 어려운 점이 있다. 첫째, 다양한 PIM 장치를 위한 컴파일러를 만들려면 PIM 내부 구현에 대한 이해 없이 사용할 수 있는 추상화된 인터페이스(interface)가 필요하다. 둘째, PIM과 호스트 사이의 데이터 이동을 최소화하는 PIM 코드를 생성해야 한다. 셋째, 연산 수행 시 데이터의 접근 패턴을 고려하여 PIM 내에 데이터를 배치해야 한다.

본 논문에서는 위에서 언급한 어려움을 해결하고 다양한 PIM을 위한 효율적인 코드를 생성하는 PIM 컴파일러를 제안한다. 제안하는 PIM 컴파일러의 주요 특징은 다음과 같다. 첫째, PIM 구조에 상관없이 딥 러닝 컴파일러를 통해 코드를 생성하기 위해 GEMV 스케줄링 템플릿과 PIM 함수 연결을 위한 스케줄 프리미티브를 제안한다. 둘째, 입력 및 출력 레지스터의 값을 재사용하여 PIM과 호스트 간의 데이터 이동을 줄인다. 셋째, 연산 수행 시 데이터 이동량을 고려하여 최적의 스케줄을 찾는 알고리즘을 제안한다. 마지막으로, PIM의 데이터 접근 패턴을 기반으로 데이터의 레이아웃을 변형한다.

본 논문에서 제안하는 PIM 컴파일러는 다양한 PIM 구조를 위해 PIM 코드를 생성하며, 실험에서는 이전에 제안된 PIM-HBM[1]을 대상으로 최적화된 코드를 생성하여 최대 2.49배의 성능 향상을 보여주었다.

2. 배경 및 관련 연구

2.1 Processing-in-Memory (PIM)

최근의 다양한 분야에서 발생하는 메모리 대역폭 제한에 따른 성능 저하 문제를 해결하는 방안으로 PIM 기술이 주목받고 있다. 본 논문에서는 GEMV 연산을 수행할 수 있는 뱅크 근처 디지털 PIM[1,2]을 대상으로 하며 그림 1과 같은 구조로 구성되어 있다.

대상 PIM은 N_{CH} PIM 채널로 구성되어 있으며, 각 채널은 독립적으로 동작하고 병렬적으로 수행될 수 있다. 각 PIM 채널은 N_P PIM 그룹으로 구성되어 있고, 각 PIM 그룹에는 PIM 유닛, 피연산자를 위한 레지스터, 그리고 N_B 뱅크가 포함되어 있다. 또한, 각 뱅크는 N_{RO} 로우와 N_{CO} 칼럼으로 이루어져 있다. 대상 PIM은 DRAM과 동일하게 READ, WRITE 커맨드를 사용하여 각 뱅크로부터 데이터를 읽거나 쓸 수 있으며 그 크

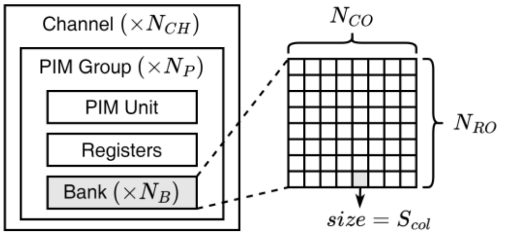

그림 1 뱅크 근처 디지털 PIM 구조
Fig. 1 Near-bank digital PIM architecture

기는 S_{col}이다.

대상 PIM은 내부 병렬성을 극대화하기 위해 하나의 DRAM 커맨드로 여러 뱅크에서 동시에 데이터를 읽거나 쓸 수 있도록 설계되었다. 따라서, 산술 연산을 위한 DRAM 커맨드가 각 PIM 그룹으로 전달되었을 때, 각 PIM 그룹은 각자의 뱅크로부터 S_{col} 크기의 칼럼 데이터를 읽어 레지스터에 저장된 데이터와 연산을 수행한 후 연산 결과를 다른 레지스터에 저장한다. 이때 뱅크로부터 읽어오는 데이터의 크기(예를 들어 32바이트)는 PIM에서 연산하는 데이터 종류(예를 들어 float16)의 크기보다 크기 때문에, 대상 PIM은 기본적으로 벡터 연산을 지원한다.

2.2 딥 러닝 컴파일러

딥 러닝 컴파일러는 다양한 연산과 하드웨어에 대한 코드 생성을 효율적으로 수행하기 위해 제안되었다. 특히, TVM[5]은 다양한 딥 러닝 프레임워크에서 정의하는 모델을 입력으로 받아 그래프 수준 최적화와 연산자 수준 최적화를 수행하며, 최종적으로 장치 별 최적화를 통해 코드를 생성한다. TVM은 연산자 수준 최적화 과정에서 연산의 정의와 스케줄 최적화를 분리한다. 연산 정의와 스케줄 최적화를 위해 각각 텐서 표현(tensor expression)과 미리 정의된 스케줄 프리미티브(schedule primitive)를 제공한다. 이러한 스케줄 프리미티브를 통해 정의된 연산에 대해 루프 타일링(loop tiling), 메모리 할당, 데이터플로우(dataflow) 변형 등을 수행할 수 있다. 그림 2는 GEMV를 위한 연산 정의 후 tile 스케줄 프리미티브를 사용하여 타일링을 수행하는 예를 보여준다.

본 논문에서 제안하는 PIM 컴파일러는 TVM을 기반으로 개발되었으며, PIM을 위한 스케줄링 템플릿과 스케줄 프리미티브를 추가하여 PIM 코드를 생성한다.

2.3 PIM 컴파일러

과거에도 PIM 컴파일러는 제안되었다. iPIM[3]은 이미지 처리를 위한 PIM 컴파일러를 제안했으며, PIM에 특화된 레지스터 할당 및 메모리 접근 순서를 고려하여 코드를 생성한다. PRIMO[4]는 PIM 내부의 데이터 이동을 최소화하기 위한 레지스터 할당 기법을 제안한다. 이들 PIM 컴파일러는 모두 HMC 기반 PIM을 대상으로 하고 있으며, 그들이 제안하는 PIM 구조에 맞춰 HMC에서 발생하는 PIM 내의 데이터 이동을 최소화하는 것에 초점을 맞추고 있다. 하지만, 본 논문에서 대상으로 하는 PIM은 PIM 유닛 간의 데이터 이동이 지원되지 않기 때문에, PIM과 호스트 간의 데이터 이동이 주로 발생하며, 이를 최소화하는 것이 중요하다.

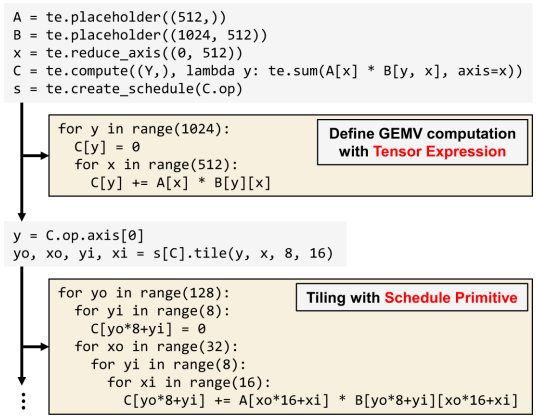

그림 2 딥 러닝 컴파일러에서의 연산 정의와 스케줄링
Fig. 2 Defining computation and scheduling in deep learning compiler

3. PIM 컴파일러

3.1 PIM 스케줄 템플릿과 프리미티브

그림 3은 입력 벡터 A와 가중치 행렬 B에 대해 출력 벡터 C를 계산하는 PIM 기반 GEMV 연산의 코드 생성을 위한 스케줄링 과정을 보여준다.

DRAM 채널과 DRAM 채널당 PIM의 개수는 각각 $N_{CH}(=X_{CH}\times Y_{CH})$, $N_P(=Y_P)$이고, PIM 커널은 PIM 유닛에서 수행되는 연산의 최소 단위로 $X_I\times Y_I$ 크기의 데이터를 한 번에 처리한다고 가정한다. 여기서 ①은 전체 GEMV 연산을 그림 4와 같이 분할하는 루프 변환 스케줄링 과정을 나타낸다. 이 분할은 가중치 행렬을 기준으로 수행되며, $X\times Y$ 크기의 행렬을 먼저 X_{CH}와 Y_{CH}로 각 DRAM 채널마다 나눈다. 그 후, 각 PIM 유닛을 위해 Y_P로 나누고, 마지막으로 각 PIM 커널을 위해 X_O와 Y_O로 나눈다. 이 과정에서 각 PIM 커널의 크기는 $X_I\times Y_I$로 결정된다. X_O와 Y_O 루프의 순서는 변경할 수 있으며, 그림 3은 X_O 루프가 바깥쪽에 위치한 경우를 보여준다. 앞서 설명한 내용에 따라 각 변수는 다음과 같은 조건을 갖게 된다.

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

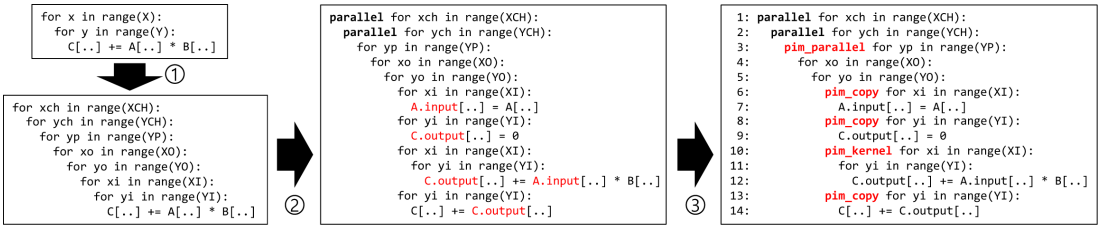

그림 3 PIM 코드 생성을 위한 GEMV 스케줄링 템플릿

Fig. 3 GEMV scheduling templates for PIM code generation

PIM 커널의 실행을 위해 입력 벡터와 출력 벡터를 PIM 레지스터로 쓰고 읽는 과정이 필요하다. 이를 위해, ②는 캐시 스케줄 프리미티브를 사용하여 입력 레지스터(input)와 출력 레지스터(output)를 사용하도록 변형한다. 또한, parallel 스케줄 프리미티브를 사용하여 각 DRAM 채널이 서로 다른 스레드로 실행될 수 있게 한다.

본 논문에서는 ②까지 완료된 스케줄을 기반으로 PIM 코드를 생성하기 위한 세 가지 PIM 스케줄 프리미티브를 제안한다. 첫째, **pim_parallel**은 PIM의 뱅크 수준 병렬성을 고려하여 해당 루프(3줄)를 Y_P번이 아닌 한 번만 실행하도록 변형한다. 단, 내부 루프들(4~14줄) 중 뱅크 수준 병렬성을 활용하지 못하는 루프(13~14줄, 출력 레지스터 값 읽기)의 경우 Y_P번 실행이 필요하므로, 이에 대한 루프를 추가(12줄과 13줄 사이)한다. 두 번째로, **pim_copy**는 루프 내의 앞선 과정에서 캐시 스케줄 프리미티브 적용시 지정한 캐시의 이름을 보고, PIM 레지스터로 읽거나 쓰는 대상 PIM의 함수로 대체한다. 마지막으로, **pim_kernel**은 $X_I \times Y_I$ 크기의 GEMV 연산을 수행하는 대상 PIM의 PIM 커널 함수로 대체한다.

3.2 PIM 레지스터 재사용 최적화

그림 3에서 레지스터의 값이 재사용되는 것을 고려하지 않을 경우, 같은 값이 입력 레지스터에 덮어쓰게 되거나, 출력 레지스터의 부분합을 미리 가져오게 된다. 이에 따라, 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. 이를 통해, PIM 코드 수행 시 PIM과 호스트 간의 데이터 이동을 줄일 수 있다.

3.3 데이터 이동 최소화 알고리즘

PIM 레지스터 재사용 최적화를 통해 PIM과 호스트 간의 데이터 이동을 줄일 수 있지만, 여전히 전체 스케줄 공간을 탐색해야 하는 문제가 남아 있다. 본 논문에서는 데이터 이동 비용을 계산하여 데이터 이동을 최소화하는 알고리즘을 제안한다.

먼저, 각 데이터플로우에 따른 데이터 이동 비용을 계산해야 한다. 데이터 플로우가 input stationary인 경우 각 PIM 채널에서 발생하는 데이터 이동 비용은 다음과 같다.

$$X_O X_I + Y_O Y_I X_O Y_P = \frac{X}{X_{CH}} + \frac{XY}{N_{CH} X_I}$$

여기서 X_{CH} 외에는 모두 상수이므로, 데이터 이동 비용을 최소화하는 문제는 X_{CH}를 최대화하는 문제로 변환할 수 있다. 또한, output stationary에 대해서도 마찬가지로 데이터 이동 비용을 다음과 같이 계산할 수 있으며, 이 경우 Y_{CH}를 최대화하는 문제로 바꿀 수 있다.

$$X_O X_I Y_I + Y_O Y_I Y_P = \frac{XY}{N_{CH} N_P Y_I} + \frac{Y}{Y_{CH}}$$

따라서, 최적의 X_{CH}와 Y_{CH}는 input stationary와 output stationary에 대해서 각각 다음과 같이 구할 수 있다.

$$X_{CH}^{IS} = \min\left(N_{CH}, \frac{X}{X_I}\right), Y_{CH}^{IS} = \frac{N_{CH}}{X_{CH}}$$

$$Y_{CH}^{OS} = \min\left(N_{CH}, \frac{Y}{Y_I Y_P}\right), X_{CH}^{OS} = \frac{N_{CH}}{Y_{CH}}$$

이때, X/X_I와 $Y/(Y_I Y_P)$는 각각 PIM 커널과 PIM 유닛을 고려했을 때 할당할 수 있는 X_{CH}와 Y_{CH}의 최대 크기를 나타낸다. 최종적으로 각 데이터플로우에 대한 최적의 X_{CH}와 Y_{CH}를 계산하여, 데이터 이동 비용이 최소화되는 데이터플로우를 찾을 수 있다.

3.4 2단계 데이터 레이아웃 변형

PIM 컴파일러는 코드 생성뿐만 아니라 데이터의 레이아웃을 PIM 연산에 적합하게 사전에 변환하고 저장하는 과정 또한 중요하다. 왜냐하면 PIM 연산의 최소 단위가 DRAM 칼럼의 크기만큼의 벡터 연산이고, PIM

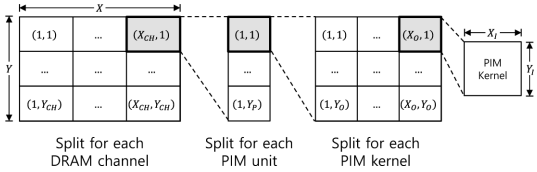

그림 4 PIM 코드 생성을 위한 GEMV 연산 타일링

Fig. 4 GEMV operation tiling for PIM code generation

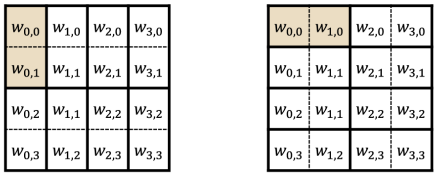

그림 5 GEMV PIM 커널의 입력 레지스터 종류에 따른 데이터 블록 생성

Fig. 5 Data block generation according to input register type of GEMV PIM kernel

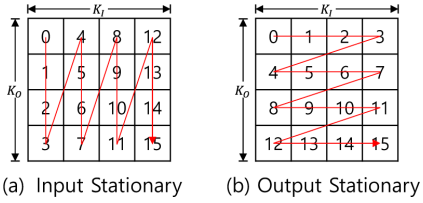

그림 6 GEMV PIM 커널의 데이터플로우에 따른 데이터 블록 배치

Fig. 6 Data block placement according to dataflow of GEMV PIM kernel

커널 수행 시 사용할 레지스터의 인덱싱은 뱅크 데이터의 로우 인덱스와 칼럼 인덱스로 결정되기 때문이다. 따라서 PIM 커널이 정확히 실행되게 하려면, 뱅크에 데이터를 저장하는 과정에서 PIM 레지스터의 특징과 데이터플로우를 모두 고려해야 한다. 본 논문에서는 DRAM 칼럼에 저장된 데이터 집합을 데이터 블록이라 정의한다.

데이터 레이아웃의 변형은 두 단계로 수행된다. 첫 번째 단계에서는 단일 데이터 블록 내의 데이터 배치를 결정하고, 두 번째 단계에서는 데이터 블록 간의 배열 순서를 결정한다.

그림 5는 하나의 DRAM 칼럼에 데이터가 2개 포함될 수 있다고 가정할 때, 입력 레지스터 종류에 따라 데이터 블록을 생성하는 첫 번째 단계를 보여준다. 이 예에서는 총 16개의 데이터로부터 8개의 데이터 블록을 생성하고 있다. 입력 레지스터가 하나의 값을 저장할 수 있는 스칼라 레지스터일 경우, 단일 입력에 대해 다수의 출력 계산이 필요하므로 그림 5(a)에서처럼 다른 출력에 대한 가중치들을 한 데이터 블록으로 생성한다. 입력 레지스터가 다수의 값을 저장할 수 있는 벡터 레지스터일 경우 다수의 입력에 대해 계산이 필요하므로 그림 5(b)에서처럼 다른 입력에 대한 가중치를 한 데이터 블록으로 생성한다.

그림 6은 PIM 커널의 데이터플로우에 따라 데이터 블록 저장 순서를 결정하는 두 번째 단계를 보여준다. 여기서 K_I와 K_O는 PIM 커널에서 사용하는 입력 및 출력 레지스터의 수를 각각 나타낸다. 그림 6의 (a)와 (b)는 Input Stationary(IS)와 Output Stationary(OS) 데이터플로우에의 데이터 배치 순서를 각각 보여준다. 만약, IS 데이터플로우로 수행되는 PIM 커널에서 OS 데이터플로우 데이터 배치를 적용할 경우 잘못된 연산 결과를 얻게 된다.

4. 실 험

4.1 실험 환경

본 논문에서 제안하는 PIM 컴파일러는 TVM[5]을 기반으로 구현되었다. 실험을 위해 PIM-HBM[1] 논문에서 명시된 PIM 설정을 사용하되, 실험의 편의성을 각 뱅크당 하나의 PIM 유닛이 존재한다고 설정하였다. 따라서, 실험에서 사용된 PIM 구조의 파라미터는 N_{CH}=16, N_P=16, N_B=1, N_{CO}=32, N_{RO}=16384, S_{col}=32로 정의되었다.

입력 벡터와 출력 벡터의 크기를 각각 X와 Y일 가정하고, 이에 따른 다양한 $X \times Y$ GEMV 연산을 PIM 컴파일러로 코드를 생성하였다. 그리고 이 코드의 실행 시간은 DRAMsim3[6] 기반으로 구현한 시뮬레이터를 통해 측정하였다. 실험은 기존 PIM-HBM 논문에서 제시된 GEMV 커널로 워크로드 분배에 대한 추가 고려 없이 X_{CH}=1로 수행하였을 때를 기준으로 하였다. 비교 대상은 PIM 컴파일러를 이용한 코드 생성 결과와 그 외에 레지스터 재사용 최적화를 적용한 후의 성능이다. 사용할 수 있는 PIM 커널의 레지스터 수는 K_I,$K_O \in \{1,2,4,8\}$로 설정되었다.

4.2 실험 결과

우선, PIM 컴파일러로 스케줄 가능한 전체 공간의 성능 분석을 위한 실험을 진행하였다. 그림 7은 PIM 컴파일러를 사용하여 생성할 수 있는 1024x2048 GEMV 연산의 1100개 코드의 성능을 오름차순으로 정렬한 결과를 나타낸다. 레지스터 재사용 최적화 없이 코드를 생성했을 경우, PIM-HBM보다 항상 성능이 떨어진 것을 확인할 수 있었다. 이는 PIM-HBM이 기본적으로 레지스터 재사용을 수행하는 코드를 실행하기 때문이다. 그러나 레지스터 재사용 최적화를 수행하면 기존 PIM-HBM을 뛰어넘는 성능을 달성하는 코드를 생성할 수 있다.

표 1에는 이러한 결과를 분석하기 위해 각 방법에 따라 어떻게 코드가 생성되는지에 대한 스케줄 설정을 보여준다. 이 표에서는 다양한 GEMV 연산에 대해 PIM-HBM과 Compiler(Code Generation+Register Reuse) 방법을 이용한 코드 생성 결과를 비교하고 있다. 표 1의 1024x2048 결과를 관찰하면 선택된 PIM 커널의 크기는 동일하지만, 전체 연산의 분배가 다르다는 것을 알 수

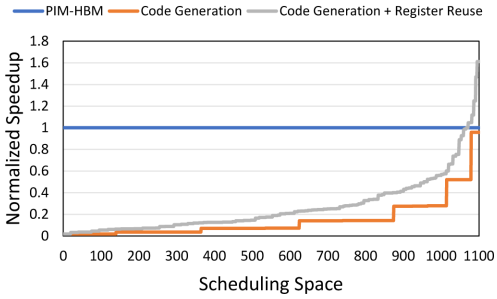

그림 7 전체 스케줄링 공간에서 PIM 코드 성능 비교 (1024x2048 GEMV)

Fig. 7 Comparison of PIM code performances in overall scheduling space (1024x2048 GEMV)

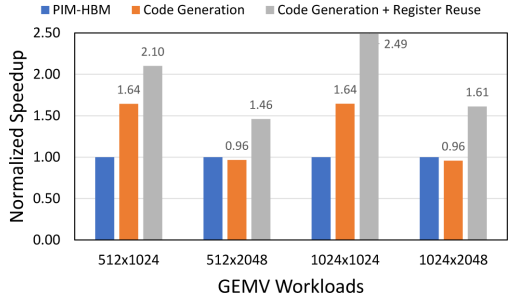

그림 8 다양한 GEMV 연산에 대한 PIM 코드 성능 비교

Fig. 8 Comparison of PIM code performances for various GEMV operations

표 1 다양한 GEMV 연산에 대한 PIM 코드의 타일링 설정 비교

Table 1 Comparison of tiling configurations in PIM code for various GEMV operations

OP	Method	X_{CH}	Y_{CH}	X_O	Y_O	X_I	Y_I
512 x 1024	PIM-HBM	1	16	4	1	128	4
	Compiler	4	4	1	2	128	8
512 x 2048	PIM-HBM	1	16	4	1	128	8
	Compiler	4	4	1	4	128	8
1024 x 1024	PIM-HBM	1	16	8	1	128	4
	Compiler	8	2	1	4	128	8
1024 x 2048	PIM-HBM	1	16	8	1	128	8
	Compiler	8	2	1	8	128	8

있다. PIM-HBM은 출력 벡터에 대해 단순 분배를 진행하므로, 각 PIM 커널 실행마다 서로 다른 입력 벡터를 PIM 레지스터에 로드해야 한다. 그러나 PIM 커널이 128개의 입력에 대해 8개의 출력을 생성하므로, 매번 다른 입력 벡터를 PIM 레지스터에 쓰는 것보다는 동일한 입력 벡터에 대해 다른 출력 레지스터의 값을 읽어오는 것이 데이터 이동량 측면에서 더 효율적이다.

그림 8은 다양한 GEMV 연산에 대한 코드 생성 및 레지스터 재사용 최적화 결과의 성능 향상을 보여준다. 단순한 코드 생성만으로도 최대 1.64배까지 성능 향상을 확인할 수 있었다. 이는 Y의 크기가 1024 이하일 때, PIM-HBM에서는 모든 PIM 채널과 PIM 유닛을 활용하여 연산하기 위해 128x4 크기의 PIM 커널을 사용해야 한다. PIM 커널의 크기가 작을 경우 PIM 커널 내에서 레지스터 재사용 효과가 줄어들어 성능이 하락하는 반면 컴파일러를 통해 생성된 코드는 PIM 커널의 크기를 최대로 사용할 수 있도록 연산 분배를 하였기 때문에 PIM-HBM보다 높은 성능을 달성할 수 있었다. 또한, 레지스터 재사용 최적화를 추가로 적용할 경우 최소 1.61배에서 최대 2.49배까지 성능이 향상이 확인되었다. 결론적으로 PIM-HBM보다 성능 개선이 되는 이유는 각 PIM 채널과 PIM 유닛에 데이터 이동을 최소화하면서 효율적인 PIM 커널을 사용할 수 있도록 전체 연산을 최적으로 분배했기 때문이다.

데이터 이동 최소화 알고리즘을 적용하여 찾은 스케줄은 표 1의 결과와 동일함을 확인할 수 있다. 이를 통해, 최적의 PIM 코드를 생성하기 위해 생성할 수 있는 모든 코드에 대해 실행 시간을 측정할 필요 없이, 데이터 이동 최소화 알고리즘을 통해 한 번에 최적의 PIM 코드를 생성할 수 있음을 알 수 있다.

5. 결 론

본 논문에서는 다양한 PIM을 위한 코드를 생성하는 PIM 컴파일러를 제안한다. 제안하는 PIM 컴파일러는 기본적인 PIM 구조를 정의하고, PIM 스케줄 프리미티브를 사용하여 PIM 코드를 생성한다. 또한, 레지스터 재사용 최적화를 통해 데이터 이동을 최적화하며, 데이터 이동 최소화 알고리즘을 통해 모든 스케줄링 공간을 탐색하지 않아도 최적화된 코드 생성을 수행한다. 마지막으로, PIM 연산 장치와 PIM 커널의 특성을 고려하여 메모리 레이아웃 변형을 수행하여, PIM에서 연산 수행 시 발생하는 메모리 레이아웃의 문제를 해결하였다. 본 논문에서 제안한 PIM 컴파일러는 기존 PIM-HBM과 비교하여 최대 2.49 배의 성능 개선을 달성한다.

우리가 제안하는 PIM 컴파일러는 최근 활발히 연구되고 있는 PIM을 추상화하고, 이를 위한 코드 생성 기술을 제안함으로써, 새로운 PIM 구조에 대한 연구가 진행될 때마다 개발자들이 PIM 소프트웨어를 수동으로 최적화할 필요를 줄여준다. 또한, 이 컴파일러는 디자인 공간 탐색(Design Space Exploration)에 활용되어, 이를 통해 최적의 PIM 구조를 찾아내는 것도 가능하다.

References

[1] S. Lee, S. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, S. O, A. Iyer, D. Wang, K. Sohn, and N. Kim, "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)*, pp. 43-56, 2021.

[2] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, T. N. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 372-385, 2020.

[3] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 804-817, 2020.

[4] H. Ahmed, P. C. Santos, J. P. C. Lima, R. F. Moura, M. A. Z. Alves, A. C. S. Beck, and L. Carro, "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 564-569, 2019.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578-594, 2018.

[6] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, Vol. 18, No. 2, pp. 106-109, July-Dec. 2020.

이 하 윤

2017년 성균관대학교 컴퓨터공학과(학사) 2017년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 Processing-in-Memory, 딥 러닝 컴파일러, 딥 러닝 모델 압축

김 경 모

2020년 성균관대학교 반도체시스템공학과(학사). 2023년 성균관대학교 반도체디스플레이공학과(석사). 2023년~현재 삼성전자. 관심분야는 Processing-in-Memory, 딥 러닝, 머신러닝, NPU, DRAM

신 동 군

1994년 서울대학교 계산통계학과(학사). 2000년 서울대학교 전산과학과(석사). 2004년 서울대학교 컴퓨터공학부(박사). 2004년~2007년 삼성전자 소프트웨어 책임연구원. 2007년~현재 성균관대학교 소프트웨어대학 교수. 관심분야는 컴퓨터구조, 운영체제, 머신러닝