PIM 아키텍처를 위한 GEMM 최적화 기법 탐구: UPMEM 사례 연구

이찬 [1], 최희림 [1], 김한준 [2]
[1] 연세대학교 전기전자공학과 통합과정
[2] 연세대학교 전기전자공학부 교수

chan.lee@yonsei.ac.kr, heelim@yonsei.ac.kr, hanjun@yonsei.ac.kr

Exploring GEMM Optimization Techniques for PIM Architecture: A Case Study on UPMEM

Chan Lee[1], Heelim Choi[1], Hanjun Kim[1]
[1]Dept. of Electrical Electronic Engineering, Yonsei University

요 약

이 연구는 PIM(Processing-in-Memory) 아키텍처를 활용하여 General Matrix Multiplication(GEMM)의 최적화 기법을 UPMEM PIM 을 통해 탐구한다. 본 연구는 CPU 에서 경험하는 메모리 대역폭의 제한을 극복하고 병렬 처리 구조를 활용함으로써 GEMM 연산에서 PIM 의 잠재적 이점을 확인한다. 또한 연속된 세 개의 행렬 곱셈에 대한 효율성을 평가하고, 데이터 전송 시간이 성능 최적화의 주요 병목 현상으로 작용하는 것을 확인한다. CPU 에서 UPMEM 커널로 전송되는 데이터의 양을 한 번에 늘리면서 전송 횟수를 줄이는 방법을 사용하여 CPU 에 비해 성능을 최대 6.57 배 향상시켰다.

1. 서론

PIM(Processing-in-Memory)은 메모리 내에 프로세서를 통합한 구조로, CPU 에서 메모리 간의 데이터 이동 병목 현상을 근본적으로 해결하기에 적합하다. 그래프 처리 및 신경망과 같은 데이터 집약적인 작업의 경우 CPU 와 메모리 간의 통신이 지연 시간 및 에너지 측면에서 부하를 야기하기 때문에 이에 대한 해결책으로 PIM 이 유리하게 활용될 수 있다. 그래프 처리나 신경망을 연산할 때 가장 기본적으로 사용되는 연산과정인 GEMM(General Matrix Multiplication)은 일반 행렬 곱셈으로, 두 행렬의 곱셈을 통해 새로운 행렬을 생성하는 연산이다. GEMM 연산을 수행할 때 행렬의 행과 열 간의 독립성을 활용하여 병렬 컴퓨팅을 수행할 수 있으며, 이는 병렬 연산에 능한 PIM 구조에 활용하기에 적합하다.

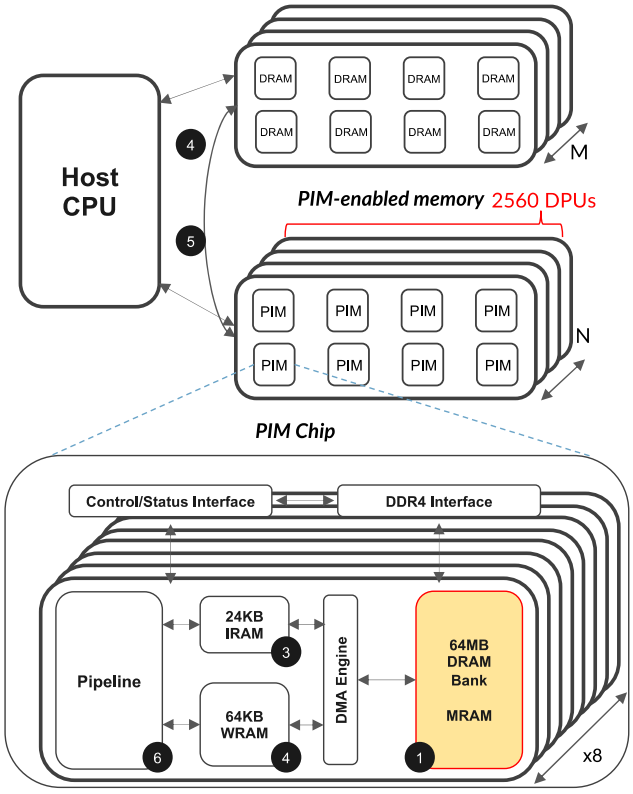
상업적인 PIM 중 하나인 UPMEM 아키텍처 [1], [2], [3]는 메모리 뱅크 근처에 DPU(DRAM Processing Unit)를 내장한 DIMM(Dual Inline Memory Module) 구조를 가지고 있다. 그러나 UPMEM 을 사용한 실험에서는 기대와 달리 PIM 이 CPU 에 비해 성능이 떨어지는 것을 발견했다. 이는 CPU 와 PIM 간의 데이터 전송을 PIM 의 메모리 사용 공간에 비해 충분히 활용하지 않기 때문이다. 이를 해결하기 위해 본 연구는 CPU 와 PIM 간의 통신 병목 현상이 발생하는 위치를 조사하고, 다양한 환경 변수를 조정하여 GEMM 연산을 최적화하는 방법을 모색했다. 이 과정에서 GEMM 연산을 위한 데이터 배치 방식을 설계하고 성능을 비교하여 가장 효율적인 방법을 찾아내고자 하였다.

2. 배경

UPMEM PIM 아키텍처는 기존 DRAM 과 DPU 가 단일 칩에 통합되어 있다. UPMEM PIM 모듈은 다른 PIM 모듈에 비해 몇 가지 장점이 있다. 첫째, 3D 적층 메모리 기술을 기반으로 한 PIM [4] 은 시뮬레이션을 기반으로 작동되지만 UPMEM 은 2D DRAM 설계 및 제조 공정으로 개발된 실제 하드웨어이다. 둘째, DPU 는 범용으로 다양한 연산과 데이터 유형을 제공한다. 셋째, 스레드가 서로 독립적으로 실행되므로 비정형 연산에 적합하다.

UPMEM PIM 시스템의 구조 [5]는 (그림 1)과 (그림 2)에서 확인할 수 있다. PIM 을 지원하는 메모리는 하나 또는 여러 메모리 채널에 위치할 수 있다. UPMEM 모듈은 64 개의 PIM 칩으로 구성된 표준 DDR4-2400 DIMM 이 20 개로 구성되어 있다. 각 PIM 칩 내부에는

8 개의 DPU 가 있다.



(그림 1) 호스트 CPU, 표준 메인 메모리 및 PIM 지원 메모리가 있는 UPMEM 기반 PIM 시스템 및 UPMEM PIM 칩의 내부 구성 요소(아래쪽)

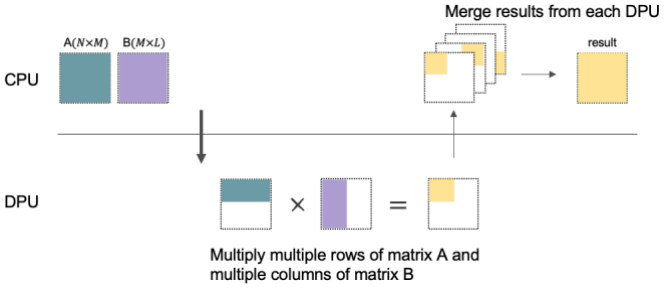
각 DPU 는 ❶ Main RAM(MRAM), 64MB DRAM 뱅크, ❷ 명령어 RAM(24KB 명령어 메모리), ❸ 64KB 의 Working RAM (WRAM, 64KB 의 스크래치 패드 메모리)에 대한 독점 액세스 권한이 있다. WRAM 은 64KB 용량을 가지고 있어 DPU 는 8, 16, 32, 64 비트의 명령어를 로드하고, WRAM 에 액세스하여 값을 저장할 수 있다. ISA 는 직접 메모리 액세스 명령어를 제공하여 MRAM 뱅크에서 IRAM 로 명령어를 이동하고 MRAM 뱅크와 WRAM 간에 데이터를 제공한다. ❹MRAM 은 호스트 CPU 로부터 입력 데이터를 복사하여 가져오고 MRAM 뱅크에서 WRAM 로 데이터를 전달한다. ❺WRAM 에서 연산한 후 DMA 를 통해 데이터를 다시 MRAM 뱅크로 수신하고, MRAM 뱅크는 그 결과를 호스트 CPU 에 다시 전달한다. CPU-DPU 및 DPU-CPU 데이터 전송은 여러 MRAM 뱅크에서 동시에 수행될 수 있으며, DPU 간의 직접 통신은 지원되지 않으므로 DPU 간의 모든 통신은 호스트인 CPU 를 통해 이루어진다. DPU 는 ISA 가 있는 32 비트 RISC 코어이고, 24 개의 하드웨어 스레드가 있으며, 각 스레드에는 32 비트 범용 레지스터 24 개가 있다. 각 하드웨어 스레드는 피연산자를 저장하기 위해 명령어 메모리(IRAM)와 스크래치 패드 메모리(WRAM)를 공유한다. ❻DPU 의 파이프라인은 14 단계로 구성된다. 그러나 파이프라인의 마지막 세 단계만 동일한 스레드에서 다음 명령어의 DISPATCH 및 FETCH 단계와 병렬로 실행할 수 있다. 따라서 동일한 스레드의 명령어는 11 주기 간격으로 DISPTACH 되어야 하며, 이를 완전히 활용하려면 최소 11 개의 스레드가 필요하다.

실험에서 사용한 CPU 는 Intel Xeon Gold 5222 를 사용하였으며, UPMEM DIMM 은 16 개로, 총 2048 개의 DPU 를 사용하였다.

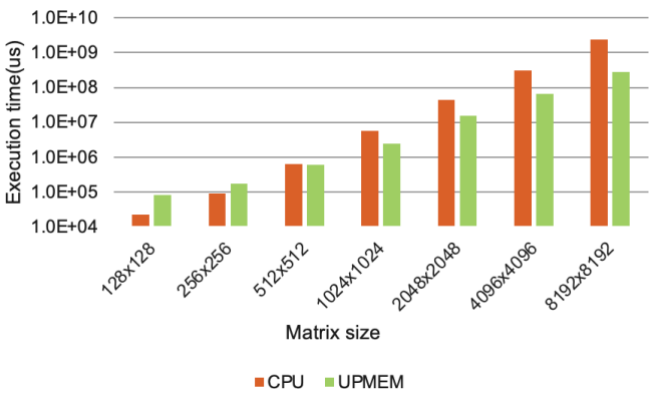
3. 설계 및 구현

UPMEM 의 하드웨어 설계에 초점을 맞춰, UPMEM 에서 수행 가능한 적합한 연산 방법을 탐색했다. 특히, DPU 의 MRAM 을 이용해 CPU 와 DPU 간 데이터를 전송하는 구조에 주목했다. 각 DPU 의 MRAM 크기는 64MB 로, 일반적인 행렬 크기에 비해 상당히 크다. 한 번에 더 많은 데이터를 MRAM 에 올리고, 각 커널 실행 시 더 많은 연산을 수행하게 함으로써, 전체 커널에서 수행되는 연산량이 동일하다고 가정할 때 성능을 향상시킬 수 있을 것으로 예측했다.

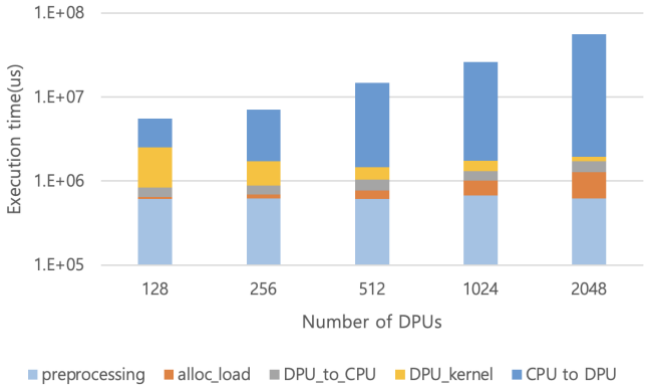
이를 바탕으로, 단일 행렬을 최적화한 방법은 다음과 같다. 전체 행렬을 각 DPU 의 MRAM 로 전송할 때, 64MB DPU MRAM 의 크기를 고려하여, (그림 3)와 같이, 행렬 A 의 n 행과 행렬 B 의 m 열을 각 DPU 로 전송해 부분 결과 행렬을 연산했다.



(그림 2) 행렬 A 의 n 개 행과 행렬 B 의 m 개 열을 곱하는 연산



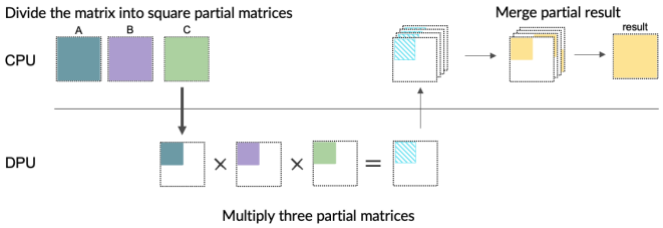
(그림 3) CPU 와 UPMEM 에서의 행렬 크기 증가에 따른 실행 시간 비교



(그림 4) 단일 행렬 연산 시 DPU 개수에 따른 실행 시간

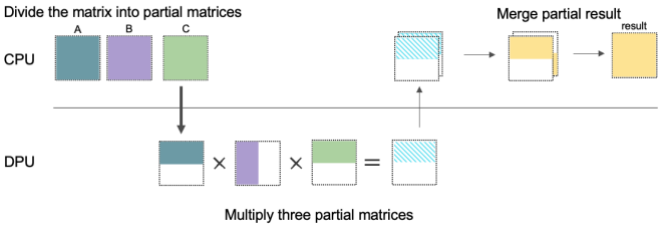
(그림 2)의 실행 결과를 CPU 와 비교해봤을 때의 결과는 (그림 3)과 같다. 512×512 이하 크기의 행렬에서는 CPU 가 더 나은 성능을 보였지만, 행렬 크기가 증가하면서 UPMEM 의 성능이 향상되었다. 특히, 행렬 크기가 8192×8192 일 때, UPMEM 은 CPU 보다 약 8.50 배 빠른 성능을 보였다. 또한 (그림 4)에서 같은 크기의 행렬 연산을 진행할 때 CPU 에서 DPU 로 데이터를 전송하는 시간이 DPU 의 수가 증가함에 따라 지수적으로 증가한다는 것을 알 수 있다. 이는 행렬 연산을 수행할 때, CPU 에서 DPU 커널로의 전송 시간을 최적화하는 것이 중요함을 알 수 있다.

그러므로, 연속된 두 행렬 곱셈의 연산을 최적화하기 위해서는 CPU 와 DPU 간의 전송 시간을 줄이는 것이 중요하다. 이러한 시나리오에서는 총 세 개의 행렬이 필요하다. 첫 번째 행렬 연산에는 A 와 B 가 있으며, 그 다음 연산에는 C 와 D 가 필요하다. 행렬 C 는 행렬 A 와 행렬 B 의 곱셈 결과이다. 또한, 행렬의 크기가 커질수록 하나의 DPU 의 메모리 용량은 전체 행렬을 보유하는 데에 충분하지 않을 수 있다. 이를 해결하기 위해, 행렬 A 을 (전체 행렬 크기) / (DPU 수)의 크기로 부분 행렬로 나누어 각 DPU 로 전송한다. 행렬 B 의 경우, 행렬 A 의 크기를 고려해서 MRAM 크기에 맞추어 데이터를 전송하기 위해, 전체 행렬을 부분 행렬로 나누었다. 연속된 두 행렬 곱셈 시나리오에 대해 두 가지 다른 접근 방식을 비교했다.



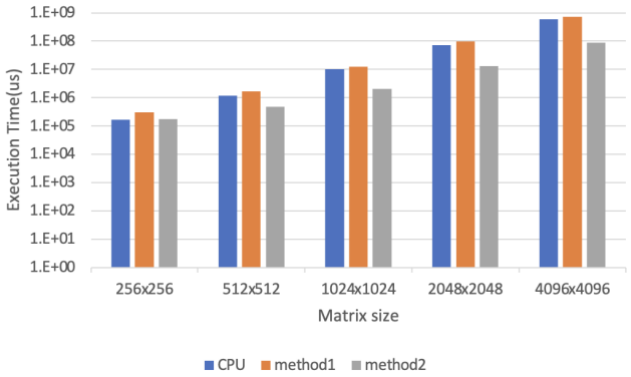
(그림 5) 방법 1: 연속적인 정사각형 부분 행렬의 곱 연산

(1) 방법 1: (그림 5)에서는 CPU 가 각 행렬을 정사각형 부분 행렬로 나누어 DPU 로 전송한다. 각 DPU 는 정사각형 부분 행렬에서 연속된 곱셈 연산을 수행한다. 그 다음 DPU 는 결과를 CPU 로 다시 전송하고, CPU 는 행렬 A, B 및 D 의 받은 부분 행렬 결과를 결합하여 최종 결과 행렬을 생성한다. 이 방법은 총 3 번의 CPU 에서 DPU 로의 전송, 1 번의 DPU 에서 CPU 로의 전송, 그리고 1 번의 CPU 와 DPU 사이의 커널 실행이 필요하다.



(그림 6) 방법 2: 연속적인 부분 최대 행렬의 곱 연산

(2) 방법 2: (그림 6)에서는 (전체 행렬 크기) / (DPU 수)의 크기로 부분 행렬로 나누어, CPU 가 A, B 및 D 의 부분 행렬을 각 DPU 로 전송한다. 각 DPU 는 A 와 B 의 부분 행렬 사이의 곱셈 연산을 수행한다. 그 결과 행렬 C 는 A 의 부분 행렬의 행 크기와 C 의 부분 행렬의 열 크기를 곱한 것으로 결정된다. 행렬 D 가 행렬 C 와의 곱셈을 위해 호환되려면 D 의 부분 행렬의 행 크기가 B 의 부분 행렬의 열 크기와 일치해야 한다. 각 DPU 에 전송되는 데이터를 최대화하기 위해 D 의 부분 행렬의 열 크기를 D 의 전체 열 크기와 일치시킨다. 그러나 행렬 C 와 행렬 D 의 부분 행렬 사이의 곱셈을 수행할 때, DPU 는 부분 결과 행렬을 다시 보낸다. 이는 불완전한 상태이다. CPU 는 그런 다음 각 DPU 로부터 받은 부분 행렬 결과를 추가하여 부분 행렬을 완성하고, 이러한 부분 행렬을 결합하여 최종 결과 행렬을 생성한다. 이 방법은 총 3 번의 CPU 에서 DPU 로의 전송, 1 번의 DPU 에서 CPU 로의 전송, 그리고 2 번의 CPU 와 DPU 사이의 커널 실행이 필요하다.



(그림 7) 연속행렬 크기에 따른 CPU 와 method1, method2 의 전체 측정 시간 비교

(그림 7)에 따르면, 행렬 크기가 256×256 일 때 CPU 가 가장 짧은 시간에 연산을 완료한다. 이는 방법 1 과

방법 2 의 CPU 와 DPU 간의 전송 시간이 데이터 연산 시간보다 길기 때문에 CPU 가 더 빠른 성능을 나타내는 것을 알 수 있다. 그러나, 행렬 크기가 512×512 일 때 방법 2 가 CPU 에 비해 약 2.5 배, 방법 1 에 비해 약 3.58 배의 성능을 보임을 알 수 있다. 그 이후로 행렬 크기가 증가함에 따라 방법 2 가 CPU 와 방법 1 에 비해 연산을 빨리 수행하며, 행렬 크기가 4096×4096 일 경우 방법 2 가 CPU 와 방법 1 에 비해 약 6.57 배와 약 8.13 배 빨라지는 것을 알 수 있다. 이는 방법 2 가 CPU 에서 DPU 로 행렬을 전송할 때 MRAM 크기에 맞게 최대한 많은 데이터를 보냈기 때문이다. 또한, DPU 에서 CPU 로 결과를 받아올 땐, CPU 와의 반복적인 통신 횟수를 줄이기 위해 행렬의 부분적 결과를 MRAM 크기에 맞게 최대화해서 보냈기 때문이다.
결과적으로 CPU 에서 DPU 로의 전송 시간을 최적화함으로써 통신 오버헤드를 효과적으로 줄이고 두 연속된 행렬 곱셈을 수행하는 효율을 향상시킬 수 있다.

4. 결론 및 향후 연구 방향

두 행렬 간 연산을 하는 단일 행렬 곱셈의 경우, 각 DPU 에서 MRAM 효율성을 향상시켜 더 많은 데이터를 단일 전송으로 전송함으로써 결과 행렬의 크기가 8192×8192 일 때 성능을 약 8.50 배 향상시킬 수 있다. 또한, 연속된 두 행렬 곱셈에서 각 DPU 의 MRAM 크기에 맞춰 보낼 수 있는 데이터의 양을 최대로 하여 전송하는 방식이 가장 효율적임을 확인하였고, 결과 행렬의 크기가 4096×4096 일 때 CPU 에 비해 성능을 약 6.75 배 향상시킬 수 있다.

본 연구에서는 CPU 와 DPU 간의 단방향 전송만을 고려하였다. 그러나 DPU 간의 양방향 통신의 경우 CPU 의 개입이 불가피하며, 통신의 규모 또한 증가하기 때문에 양방향 통신의 최적화에 대한 연구가 선행되어야 한다. 즉, DPU 간 통신의 최적화를 통해 더 효율적인 데이터 교환 및 처리를 위한 전략을 개발할 필요가 있다. 이는 더 나은 성능과 효율성을 달성하기 위해 고려해야 할 중요한 측면이다.

참고문헌

[1] J. Gómez-Luna, et al., "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture." *IEEE Access, vol. 10, pp. 52565-52608*, 2022.

[2] J. Gómez-Luna, et al., “An experimental evaluation of machine learning training on a real processing-in-memory system.” *arXiv preprint arXiv:2207.07886*, 2023

[3] F. Devaux, “The true Processing In Memory accelerator” in *HCS*, 2019.

[4] Sukhan, Lee, et al., “Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology.” in *ISCA,* 2021.

[5] UPMEM. (2021). UPMEM Software Development Kit (SDK). URL: https://sdk.upmem.com/2021.3.0/.