PIM 장치의 효율적 활용을 위한 데이터 이동량 정적 분석 기반 병렬화 알고리즘

곽준호 [1], 조정훈 [2]
[1] 경북대학교 전자전기공학부 박사과정
[2] 경북대학교 전자공학부 교수

junho7513@knu.ac.kr, jcho@knu.ac.kr

Parallelization Algorithm Based on Static Analysis of Data Movement for Efficient Utilization of PIM Devices

Junho Kwak[1], Jeonghun Cho[2]
[1] School of Electronic and Electrical Engineering, Kyungpook National University
[2] School of Electronics Engineering, Kyungpook National University

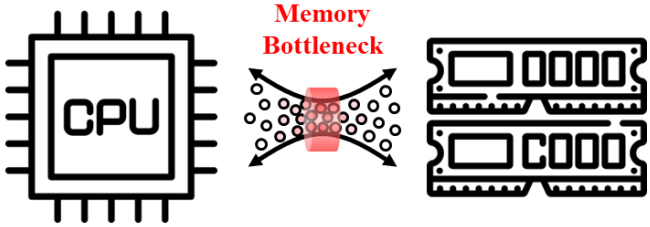
요 약

최근 인공지능, 빅데이터 분석 등 데이터 기반 애플리케이션이 빠르게 발전함에 따라, 잦은 메모리 접근으로 인한 메모리 병목 현상이 중요한 문제로 대두되고 있다. 이를 해결하기 위한 방안으로 메모리 내에서 연산을 수행하는 Processing-in-Memory (PIM) 아키텍처가 주목받고 있지만, 병렬 처리를 전제로 한 구조적 특성으로 인해 실제 활용에는 어려움이 따른다. 본 논문에서는 PIM 장치를 효율적으로 활용하기 위한 데이터 이동량 정적 분석 기반 병렬화 알고리즘을 제안한다. 제안하는 알고리즘은 연산을 최소 단위로 분할한 후, 데이터 이동량 기반의 비용 함수를 통해 각 타일의 비용을 정량적으로 분석한다. 이를 바탕으로 병렬화 수준을 자동으로 결정하며, UPMEM PIM 서버에서의 실험을 통해 제안 알고리즘이 도출한 병렬화 수준이 실제 최적의 수준과 유사함을 확인하였다.

1. 서론

최근 인공지능 (AI), 빅데이터 분석 등 데이터 기반 애플리케이션의 규모와 복잡도가 빠르게 증가하고 있다. 이러한 애플리케이션들은 방대한 양의 데이터를 처리해야 하며 이를 위해 잦은 메모리 접근이 필요하여 메모리 집약적 애플리케이션으로 분류된다. 메모리 집약적 애플리케이션의 지속적인 메모리 접근은 기존 컴퓨터 아키텍처인 폰 노이만 (von Neumann) 아키텍처에서 메모리 병목 (memory bottleneck) 현상을 초래한다 [1].

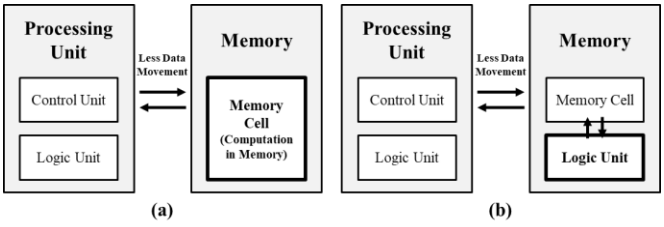
그림 1 은 폰 노이만 아키텍처의 구조와 메모리 병목 현상을 보여준다. 폰 노이만 아키텍처는 연산 장치와 저장 장치가 분리되어 있어 데이터 처리를 위해 데이터의 이동이 필요하다. 그러나 제한된 메모리 대역폭으로 인해 빈번한 메모리 접근은 데이터 이동 병목 현상을 유발하며 이는 시스템 성능을 크게 저하시킨다.



(그림 1) 폰 노이만 아키텍처와 메모리 병목 현상

메모리 병목 현상을 해결하기 위한 대안으로 메모리 내 처리를 수행하는 Processing-in-Memory (PIM) 아키텍처가 주목받고 있다. PIM 아키텍처는 저장 장치에 연산 기능을 통합하여 데이터 이동을 최소화하고 에너지 소모를 줄이는 구조를 가진다 [2]. PIM 아키텍처는 크게 두가지 구조로 구분할 수 있으며 그림 2 는 이를 보여준다. 그림 2(a)는 In-Memory Processing (IMP)으로 기존에는 데이터의 저장만을 수행하던 메모리 셀 안에서 바로 연산을 수행하는 구조를 가진다. 그

림 2(b)는 Near-Memory Processing (NMP)으로 메모리 셀 근처에 간단한 연산 장치 (Logic Unit)를 배치하여 메모리 근처에서 연산을 수행하는 구조를 가진다. PIM 아키텍처는 저장 장치 내 연산을 통해 데이터 이동량을 줄여 메모리 집약적 애플리케이션의 가속화를 가능하게 한다.



(그림 2) PIM 아키텍처 구조

PIM 아키텍처는 충분한 가능성을 보이지만 기존 시스템 구조와의 차이로 인해 PIM 아키텍처에 애플리케이션을 효과적으로 오프로딩 하는 것은 여전히 어려운 문제로 남아있다. 특히, PIM 아키텍처는 메모리 내 각자의 메모리 공간을 담당하고 있는 다수의 경량 처리 장치가 있기 때문에 오프로딩을 위해 연산을 병렬화 하는 방법이 중요하나 이는 고도의 설계가 요구된다. 메모리 접근 패턴을 고려하지 않은 단순한 워크로드 분할은 오히려 데이터 이동량을 증가시켜 성능을 저하시킬 수 있다. 따라서, PIM 하드웨어의 특성을 고려한 지능적인 병렬화 전략이 필수적이다.

본 논문에서는 PIM 아키텍처를 효율적으로 활용하기 위한 데이터 이동량 정적 분석 기반 병렬화 알고리즘을 제안한다. 제안하는 알고리즘은 병렬화 수준에 따른 데이터 이동량을 정적으로 분석하여 비용을 최소화할 수 있는 적절한 병렬화 수준을 결정한다. 실제 PIM 장치에서의 실험을 통해 제안하는 알고리즘이 적절한 병렬화 수준을 찾아 연산을 효율적으로 분배하고 성능을 향상시킴을 검증한다.

2. 데이터 이동량 정적 분석 기반 병렬화 알고리즘

데이터 이동량 정적 분석 기반 병렬화 알고리즘은 PIM 아키텍처에서의 효율적인 동작을 위해 데이터 이동량 정적 분석을 기반으로 대상 연산에 대한 적절한 병렬화 수준을 결정한다. 대상 연산에는 Elementwise 연산, General Matrix-Vector multiplication (GEMV) 연산, 그리고 General Matrix-Matrix multiplication (GEMM) 연산이 포함된다.

그림 3 은 제안하는 알고리즘의 의사코드를 보여준다. 알고리즘은 병렬화 대상이 되는 연산의 종류와 피연산자에 대한 정보를 입력 받는다. 그후, 크게 네 단계에 걸쳐 병렬화를 수행한다.

Algorithm 1 Parallelization Based on Static Analysis of Data Movement

Input: *op_type*: Operation type
operand_num: Number of operands
operand_dtype: Data type of operands
operand_shape: Shape of each operand
Output: *tile_info*: Optimal tiling configuration

/* Step 1: Data Preprocessing */
if *op_type* is ELEMENTWISE **then**
  operand_shape ← flattened 1D shape
else if *op_type* is GEMM **then**
  Second *operand_shape* ← transposed
end if

/* Step 2: Parallel Unit Construction */
output_shape ← computed from *operand_shape*
Find minimal unit shape satisfying:
  - Parallelizable (no data dependency between units)
  - Input size 8-byte alignment
  - Output size 8-byte alignment

/* Step 3: Tile Extension and Cost Evaluation */
tile_shape ← unit shape
Compute *max_extension_count* based on unit shape and unit count
Evaluate cost of initial tile configuration
Add (tile shape, cost) to candidate_tiles
for *extension_count* = 1 to *max_extension_count* **do**
  Attempt to extend tile by adding one unit
  if extended tile fits within memory budget **then**
    tile_shape ← extended tile shape
    Generate tiles from *tile_shape*
    Evaluate cost of the new tile configuration
    Add (tile shape, cost) to candidate_tiles
  end if
end for

/* Step 4: Search for Optimal Tiling */
tile_info ← tile configuration with the minimal total cost
return *tile_info*

(그림 3) 데이터 이동량 정적 분석 기반 병렬화 알고리즘 의사코드

첫번째 단계는 입력 데이터에 대한 전처리를 수행한다. Elementwise 연산의 경우, 각 요소별로 연산이 수행되기 때문에 피연산자의 데이터 차원이 중요하지 않다. 그렇기 때문에 이후의 과정에서 처리하기 용이하도록 1차원 데이터로 변환한다. GEMM 연산의 경우, 연산을 수행할 때 데이터가 연속된 공간에 배치될 수 있도록 두번째 피연산자 행렬을 Transpose 한다.

두번째 단계는 최소 병렬화 단위 (parallel unit)를 결정한다. 최소 병렬화 단위는 병렬화가 가능한 가장 작은 형태로 이후 병렬화 수준을 바꿀 때 기본 단위가 된다. 최소 병렬화 단위를 결정하기 위해 먼저 출력 데이터 형태를 계산한다. 이후, 출력 데이터의 각 요소를 기본 단위로 그룹을 묶어 나가면서 병렬화 가능 조건을 확인하고, 병렬화 가능 조건을 만족하는 가장 작은 크기의 그룹을 최소 병렬화 단위로 설정한다. 이때, 빠른 탐색을 위해 출력 데이터 요소 1 개부터 시작해 하나씩 그룹에 추가해가며 조건을 확인한다. 병렬화가 가능하기 위한 조건으로는 기본적인 데이터 비의존성 외에도 메모리 정렬 조건이 포함된다. 병렬화를 위해 여러 경량 처리 장치에 데이터를 전송할 때 메모리 정렬 조건을 만족해야 문제없이 데이터를 전송할 수 있다. 본 알고리즘에서는 그룹의 연산

에 필요한 입력 데이터의 크기와 결과인 출력 데이터의 크기가 메모리 정렬에 맞는지 확인한다. 만약, 시스템에 따라 추가할 조건이 있다면 추가할 수 있다.

세번째 단계는 최소 병렬화 단위를 조합하여 여러 형태의 타일을 형성하고 타일의 비용을 계산한다. 처음에는 타일을 최소 병렬화 단위로 초기화하여 가장 작은 단위로 병렬화를 수행했을 때의 비용을 계산한다. 이후, 타일은 최소 병렬화 단위로 하나씩 확장을 하며 비용을 계산한다.

Elementwise 연산의 경우, 출력 데이터의 형태가 1 차원이기 때문에 최소 병렬화 단위 또한 1 차원 형태를 가지고 있어 하나의 방향으로 타일을 확장한다. GEMV 연산도 마찬가지로 출력 데이터의 형태가 1 차원이기 때문에 해당 방향으로 확장한다. 반면 GEMM 연산의 경우, 출력 데이터의 형태가 2 차원이기 때문에 탐색 공간이 매우 넓어진다. 2 차원 탐색 공간에 대해 모든 확장을 계산하는 것은 연산량이 매우 많아지기 때문에 이를 방지하기 위해 타일을 확장할 때 Greedy 알고리즘을 적용한다. 타일 확장 시점에 두 차원 중 비용이 더 적은 차원으로의 확장을 선택하고, 그 이후에도 동일한 반복을 통해 비용이 적은 방향으로만 탐색을 수행한다. 만약 최대 확장 횟수가 넘어가거나 메모리 제약으로 인해 더 이상 타일을 확장할 수 없다면 비용 계산이 종료된다.

마지막 네번째 단계는 앞서 계산한 비용을 바탕으로 최소 비용을 가지는 타일을 찾는다. 최소 비용을 가지는 타일의 형태로 병렬화 수준을 결정한다.

$$Cost = T_{scatter} + T_{compute} + T_{gather} \quad (1)$$

수식 (1)은 알고리즘의 세번째 단계에서 비용을 계산할 때 사용하는 비용 함수를 보여준다. 실행 시간을 비용으로 계산하며 병렬 연산 시간 $T_{compute}$ 뿐만 아니라 병렬 연산을 위해 데이터가 이동하는 시간 또한 포함한다. $T_{scatter}$는 다수의 경량 처리 장치에 데이터를 전송하는 시간이고, T_{gather}는 병렬 연산 후 결과를 전송받는 시간이다. 잘못된 병렬화에 의해 데이터 이동량이 많아진다면 이는 $T_{scatter}$, T_{gather}를 증가시켜 그대로 비용에 반영된다.

$$T_{scatter} = \alpha_{scatter} \cdot N_{tiles} + \frac{NBYTE_{scatter}}{BW_{scatter}} \quad (2)$$

$$T_{compute} = \max_{PEs}\left(\frac{N_{ops}}{MOPS} + T_{boot\ (option)}\right) \quad (3)$$

$$T_{gather} = \beta_{gather} \cdot N_{tiles} + \frac{NBYTE_{gather}}{BW_{gather}} \quad (4)$$

수식 (2)는 병렬 연산을 위해 데이터를 전송하는 시간의 계산식을 보여준다. $\alpha_{scatter}$는 데이터 전송을 준비하는데 걸리는 시간이고, $BW_{scatter}$는 데이터 전송 대역폭이다. 타일의 형태, 크기, 그리고 개수에 따라 N_{tiles}, $NBYTE_{scatter}$가 결정되고 이 값을 바탕으로 데이터 전송 시간이 계산된다. 수식 (3)은 병렬 연산을 수행하는 시간의 계산식을 보여준다. 다수의 경량 처리 장치가 병렬로 실행되기 때문에 그중 가장 큰 실행 시간이 $T_{compute}$가 된다. $T_{compute}$는 할당된 타일을 계산하기 위한 연산 횟수 N_{ops}와 경량 처리 장치의 연산 능력 $MOPS$에 따라 실행 시간이 계산된다. 만약 경량 처리 장치의 부팅 시간이 필요하다면 T_{boot}를 추가할 수 있다. 수식 (4)는 병렬 연산 결과를 전송받는 시간의 계산식이며 수식 (2)와 구성이 동일하다.

3. 실험 및 성능 평가

제안하는 알고리즘을 검증하기 위해 상용 PIM 장치인 UPMEM PIM 을 활용했다. UPMEM 사에서 UPMEM PIM 이 장착된 서버를 클라우드 서비스로 제공하고 있기 때문에 해당 서비스를 사용하여 실험을 진행했다. 표 1 은 실험을 진행한 UPMEM 서버의 사양을 보여준다.

<표 1> UPMEM 서버 사양

CPU	Intel Xeon
Memory	4x64 GB DDR4-2666 RDIM Dual Rank DRAM (256 GB)
PIM Memory	20xDDR4-2400 PIM modules (160 GB)
DPU	2560 DPUs @ 450 MHz

제안하는 알고리즘이 동작하기 위해서는 비용 함수 내 파라미터 값을 설정해야 한다. 표 2 는 UPMEM 서버에서 실제 측정한 파라미터 값을 보여준다.

<표 2> 비용 함수 파라미터 설정값

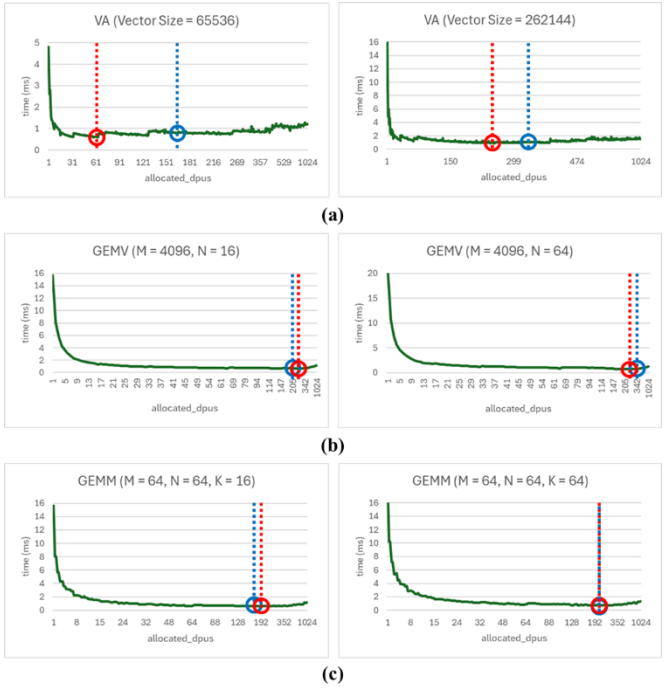
$\boldsymbol{\alpha_{scatter}}$	31.671 ns
$\boldsymbol{BW_{scatter}}$	4.3289 GB/s
$\boldsymbol{\beta_{gather}}$	21.377 ns
$\boldsymbol{BW_{gather}}$	1.7814 GB/s
$\boldsymbol{MOPS}$	42.936 MOPS
$\boldsymbol{T_{boot}}$	276 us

검증에는 메모리 집약적 벤치마크 모음인 PrIM 벤치마크를 사용했다 [3]. 그중, Elementwise 연산을 수행하는 VA (Vector Addition) 벤치마크와 GEMV 연산을 수행하는 GEMV 벤치마크를 사용했다. GEMM 연산의 경우, PrIM 벤치마크에 포함되어 있지 않아 직접 코드를 구현하여 검증을 진행했다.

그림 4 는 각 벤치마크에 대한 실험 결과를 보여준

다. 그래프의 가로축은 병렬화 수준 (개수)을 나타내고, 세로축은 연산의 전체 실행 시간을 나타낸다. 모든 실험에서 최대 1024 개까지 병렬화 수준을 증가시키며 실행 시간을 측정했다. 각 그래프의 빨간색 표시는 실제 실험에서 가장 짧은 실행 시간을 보여준 최적의 병렬화 수준이고, 파란색 표시는 본 논문에서 제안하는 알고리즘이 결정한 병렬화 수준이다.

그림 4(a)는 VA 벤치마크의 실험 결과이다. 알고리즘이 결정한 병렬화 수준과 최적의 병렬화 수준이 차이가 있다는 결과를 보여준다. 하지만 실행 시간에서는 거의 차이가 없음을 확인할 수 있다. 그림 4(b)는 GEMV 벤치마크의 실험 결과이고, 그림 4(c)는 직접 구현한 GEMM 벤치마크의 실험 결과이다. 두 결과 모두에서 알고리즘이 결정한 병렬화 수준이 최적의 병렬화 수준과 거의 유사한 것을 확인할 수 있다.



(그림 4) 벤치마크 실험 결과

<표 3> 실행 시간 비교

(ms)	VA		GEMV		GEMM	
	left	right	left	right	left	right
Optimal	0.591	0.915	0.627	0.748	0.578	0.720
Max	1.201	1.551	1.148	1.319	1.099	1.263
Algorithm	0.803	1.048	0.643	0.788	0.632	0.720

표 3 은 그림 4 의 실험 결과에서 측정한 실행 시간의 비교를 보여준다. 그림 4 를 기준으로 각 벤치마크의 두가지 실험을 'left', 'right'로 표시하였다. Optimal 은 최적의 병렬화 수준으로 그림 4 의 빨간색 표시에 해당하는 실행 시간이고, Algorithm 은 제안하는 알고리즘이 결정한 병렬화 수준으로 그림 4 의 파란색 표시에 해당하는 실행 시간이다. Max 는 최대 병렬화 수준 (1024)에 해당하는 실행 시간이다.

표 3 은 알고리즘이 제안하는 병렬화 수준이 최적과 유사하거나 그에 살짝 미치는 못한다는 결과를 보여준다. 이는 비용 함수가 시스템의 동적인 요소까지 반영을 하지 못했기 때문이다. 하지만 모든 실험에서 최대 병렬화 수준에 비해 우수한 결과를 보여줬다. 대부분의 병렬화 작업에서는 보통 자원에 맞춰 최대 병렬화를 하는 것이 일반적이다. 하지만 제안하는 알고리즘은 데이터 이동량을 고려한 비용을 바탕으로 적절한 병렬화 수준을 결정하는 결과를 보여주었다.

4. 결론

본 논문에서 우리는 PIM 장치의 효율적인 활용을 위해 데이터 이동량 정적 분석을 기반으로 적절한 병렬화 수준을 결정하는 병렬화 알고리즘을 제안하였다. 본 알고리즘은 최소 병렬화 수준부터 단계적으로 타일을 형성하였고, 비용 함수를 통해 각 타일의 비용을 계산하였다. 비용 함수는 병렬 연산 시간뿐만 아니라 데이터 전송 시간까지 포함하여 데이터 이동량의 영향도 반영하였다. 이와 같이 계산한 비용을 바탕으로 적절한 병렬화 수준을 결정했고, 실제 PIM 장치인 UPMEM 서버에서 알고리즘을 검증하였다. 검증 결과에서 알고리즘이 결정한 병렬화 수준은 최적의 병렬화 수준과 거의 유사한 실행 시간을 보여주었다.

우리는 더욱 정교한 비용 함수를 설계하여 최적의 병렬화 수준을 찾을 수 있게 연구를 발전시킬 것이며, 추가적으로 컴파일러와 연동하여 컴파일 단계에서 병렬화가 자동으로 적용되도록 연구를 수행할 계획이다.

ACKNOWLEDGMENT

이 논문은 2025 년도 정부(산업통상자원부)의 재원으로 한국산업기술진흥원의 지원을 받아 수행된 연구임 (RS-2024-00415938, 2024 년 산업혁신인재성장 지원사업).

참고문헌

[1] Asif Ali Khan "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview" arXiv, 2024

[2] Gagandeep Singh "Near-memory computing: Past, present, and future" Microprocessors and Microsystems, Volume 71, 2019

[3] J. Gómez-Luna "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System" in IEEE Access, vol. 10, pp. 52565-52608, 2022