한양대학교

졸업프로젝트 결과보고서

프로젝트명	Processing-in-Memory 활용 기술 개발
프로젝트 요약	Processing-in-Memory의 개념을 이해하고, UPMEM-PIM의 구조 및 UPMEM-PIM SDK를 분석하여 개발 방법을 파악한다. 이를 바탕으로 PIM을 이용한 sort-merge join 알고리즘을 고안 및 구현하고, PIM의 유무에 따른 성능을 비교한다. 데이터의 크기가 커질수록 PIM을 통한 유의미한 성능 향상이 있음을 확인하였다.
프로젝트 기간	2024.03.04. ~ 2024.10.31.
산출물	졸업 작품 (○), 졸업 논문 ( )

학과	학번	학년	이름	연락처
컴퓨터소프트웨어	2021019961	4	장서연	tjdus092222@hanyang.ac.kr 010-2658-4217
컴퓨터소프트웨어	2021097356	4	김한결	gyeol0719@hanyang.ac.kr 010-3136-7140

한양대학교

목 차

1. 프로젝트 개요

1.1 프로젝트 목적 및 동기

본 프로젝트에서는 Processing-in-Memory(PIM)를 활용하여 sort-merge join 알고리즘을 구현하고, PIM의 유무에 따른 성능의 차이를 알아본다.

프로젝트 진행을 위해 데이터베이스, 컴퓨터 구조에 대한 이해가 필요하다. 대용량의 데이터를 처리하는 과정에서 기존의 폰 노이만 구조로는 제한된 대역폭(bandwith)와 에너지 부족, 시스템 지연 등의 여러 한계점이 존재하며, 병목현상(bottleneck)이 발생한다. 이를 해결하고자, CPU와 DDR4 DRAM memory를 DRAM memory 다이에 통합시킨 PIM이 등장했다. PIM은 데이터가 있는 곳에서 연산을 수행할 수 있으며 CPU와 DRAM 사이에서는 데이터 자체가 아닌 프로그래밍 명령과 결과만이 이동한다.

데이터베이스 분야에서 활용되는 join 알고리즘 중 하나인 sort-merge join 알고리즘 또한 대규모 데이터 이동과 상당한 연산이 요구되므로 기존의 컴퓨터 구조로는 병목현상이 존재할 수 있다. 따라서 PIM 활용이 적합한 케이스이며, 성능을 향상할 수 있도록 PIM을 활용한 HW 설계를 고려해 프로그래밍할 필요성이 있다고 판단하였다. 또한, 데이터 집약형(data-intensive) application의 병목현상에 대한 해결이 요구되므로 PIM을 활용한 연구, 개발은 대용량 데이터를 다루는 분야의 산업적, 학술적 발전에 이바지할 수 있을 것이다. 뿐만 아니라 PIM의 유무에 따른 성능의 차이를 비교하며 PIM의 기술적 의의를 확인할 수 있기에 해당 프로젝트를 시작하게 되었다.

1.2 프로젝트 목표

본 프로젝트의 최종 목표는 UPMEM사의 PIM을 이용하여 PIM을 활용한 sort-merge join 알고리즘을 구현하고 PIM의 유무에 따른 성능을 비교하는 것이다. 이를 달성하기 위해 다음과 같은 세부 목표가 필요하다. PIM(Processing-in-Memory)의 개념을 이해하고, 본 프로젝트에서 사용할 UPMEM-PIM의 구조 및 UPMEM-PIM SDK 분석을 바탕으로 PIM을 이용한 개발 방법 파악한다. 이를 바탕으로 PIM을 이용한 sort-merge join 알고리즘의 성능을 향상할 방법 고안 및 구현하고, PIM의 유무에 따른 성능을 비교한다.

2. 프로젝트 내용

다음은 프로젝트를 진행하기 위해 필요한 조사 결과와 연구개발 계획에 관한 내용이다.

2.1 Processing-In-Memory(PIM)

데이터 집약형 워크로드는 CPU와 메인 메모리 간에 많은 데이터 이동을 요구하므로, 지연과 에너지 측면에서 상당한 오버헤드를 발생시킨다. 이러한 병목 현상으로 인해 산술 연산과 메모리 접근 간의 격차는 점점 더 커지고 있으며 메모리 접근 비용 역시 비싸지고 있다. 연구 결과에 따르면, 데이터 이동이 전체 시스템 에너지의 62%(2018년), 40%(2014년), 35%(2013년)를 차지한다고 한다.

병목 현상을 완화하리라 기대되는 방법의 하나가 메모리 내부에 프로세스를 가지는 processing-in-memory(PIM)이다. 3D-stacked 메모리에 DRAM 레이어와 로직 레이어를 통합하는 processing-near-memory(PNM), SRAM, DRAM 또는 비휘발성 메모리의 메모리 셀의 아날로그 운영 특성을 활용하여 특정 유형의 작업을 효율적으로 수행하는 processing-using-memory(PUM)이 있었지만, 전자는 높은 비용, 제한된 용량, 로직 레이어로 의한 임베디드 프로세싱 컴포넌트에 제약이 존재한다는 이유로, 후자는 복잡한 연산이 어렵고 규칙적인 계산에 더 적합하다는 한계 때문에 두 방법을 기반으로 하는 완벽한 PIM 시스템은 상용화가 되지 않았다.

2.2 UPMEM-PIM

UPMEM PIM은 실제 하드웨어에서 상용화된 첫 번째 PIM이다. PNM과 PUM의 한계를 극복하기 위해, 2D DRAM 배열을 사용하고 이를 동일한 칩에 DRAM Processing Units(DPUs)와 결합한 구조를 가진다. UPMEM은 비교적 깊은 파이프라인을 가지고 멀티스레드로 처리되는 DPU 코어를 사용한다.

UPMEM PIM 구조는 다른 PIM 방식에 비해 다음과 같은 장점을 가진다. 첫째로, 2D DRAM을 사용하므로 PNM에서 발생하던 문제를 피할 수 있고, 둘째, DPU가 다양한 연산 및 자료형을 지원한다. 셋째, 스레드가 독립적으로 실행될 수 있기에 불규칙적 연산에 적합하고, 넷째, UPMEM에서 C 언어로 DPU 프로그램을 작성할 수 있게끔 정보를 제공한다.

아래 그림은 호스트 CPU와 UPMEM PIM 시스템을 보여준다. PIM을 사용하는 메모리는 N개의 UPMEM PIM 모듈로 이루어지며, 이 모듈은 랭크 당 8개(총 16개)의 PIM 칩을 가진다. PIM 칩은 8개의 DPU로 이루어지며, 각각의 DPU는 메인 메모리인 MRAM, 명령어를 위한 IRAM, 스크래치 패드나 MRAM을 위한 캐시로 사용되는 WRAM을 가진다. MRAM은 호스트 CPU에서 데이터를 복사하고(CPU-DPU) 결과를 찾는데(DPU-CPU), 이는 MRAM에서 전송되는 버퍼의 크기가 동일할 때 병렬로 일어날 수 있다. 그렇지 않다면, 하나의 MRAM에서 전송이 완료된 후 다른 MRAM에서의 전송이 시작되는 식으로 순차적으로 전송이 발생한다. 모든

DPU는 호스트 CPU에 대한 병렬 보조 프로세서로 함께 작동하며 DPU 간의 직접적인 통신은 불가능하다. 모든 DPU 간의 통신은 호스트 CPU를 통해 DPU에서 CPU로 결과를 검색하고 CPU에서 DPU로 데이터를 복사하는 방식으로 이루어진다. DPU는 24개의 하드웨어 스레드를 가지며, 14단계의 파이프라인 깊이를 가지지만 마지막 세 단계만이 DISPATCH 및 FETCH 단계와 병렬로 실행될 수 있다. 따라서 파이프라인을 완전히 활용하려면 적어도 11개의 스레드가 사용해야 한다.

UPMEM PIM 시스템의 일반적인 프로그래밍 권장 사항은 다음과 같다. 첫째, 호스트 CPU와 빈번한 상호 작용을 피하고 가능한 한 긴 병렬 코드를 DPU에서 실행한다. 둘째, 워크로드를 독립적인 데이터 블록으로 분할하여 사용한다. 셋째, DPU를 최대한 많이 사용한다. 넷째, 각 DPU에서 적어도 11개의 tasklet을 사용한다.

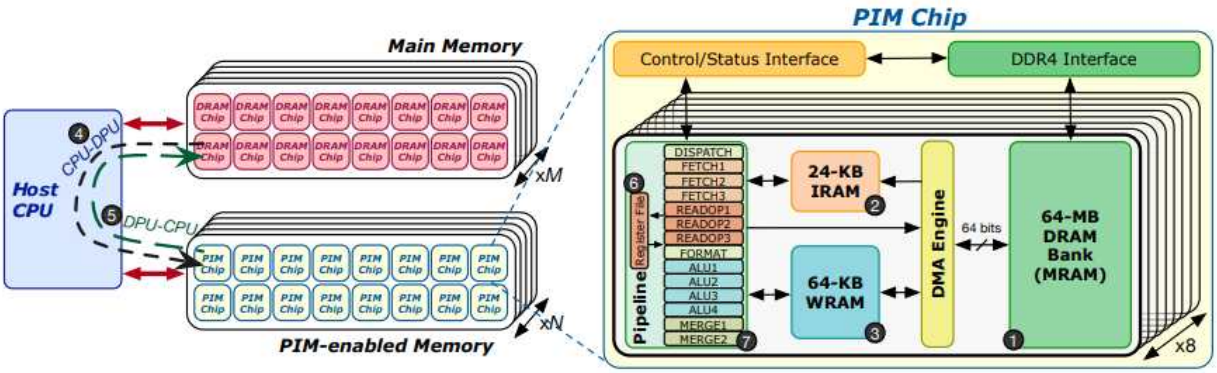


그림 1 UPMEM-based PIM system with a host CPU, standard main memory, and PIM-enabled memory (left), and internal components of a UPMEM PIM chip (right) [2].

2.3 UPMEM-PIM SDK

UPMEM사는 개발자들이 PIM 프로그래밍에 쉽게 적응할 수 있도록 Software Development Kit(SDK)와 toolkit, 디버거를 제공한다.

첫 번째로 functional library이다. 이는 tasklet을 통해 소프트웨어 추상화와 기본적인 하드웨어 스레드에 스택과 같은 시스템 기능을 수행한다. 이 기능을 기반으로 runtime libraray는 mutex, semaphore 등 다양한 동기화 기본 요소를 제공한다. 또한, WRAM 동적 관리, MRAM과 WRAM 사이의 트랜잭션을 관리하며 MRAM에 접근을 수행할 수 있고, tasklet으로 자신의 목적을 위해 작업 메모리에 버퍼를 가져오거나 공동 작업을 위해 미리 예약된 일부 공유 메모리에 접근이 가능하다.

두 번째는 communication library이다. 호스트 측에서, C API를 사용해 호스트 메모리와 DPU 메모리(IRAM, WRAM, MRAM) 중 임의의 메모리 사이에 데이터를 전송하는 기능을 제공한다.

세 번째는 LLVM 컴파일러와 LLDB 디버거이다. DPU를 타겟으로 한 컴파일러는 clang 10.0.0을 사용하는 LLVM을 기반으로 한다. PIM 아키텍처는 수천 개의 프로세서를 한 번에 디버깅해야 하므로 lldv 10.0.0을 기반으로 효율적인 디버깅 툴을 고안했다. 이는 코드 구조

유지를 위해 CPU와 DPU를 동시에 작용하고, 그룹 또는 개별 DPU 단위로 작동하여 정확한 파악이 가능하며, 성능을 고려한 낮은 수준의 프로파일링으로 성능 저하가 발생하는 특정 DPU를 파악할 수 있다.

네 번째는 functional simulator이다. toolkit은 instruction trace와 DPU 그라인드로 구성된 functional simulator를 제공한다. functional simulator에는 cycle-accurate가 없기 때문에 trace 카운터를 cycle 카운터로 해석해서는 안 된다. 따라서 성능 평가가 허용되지 않으며, 평가를 위해서는 클라우드 서버나 평가 서버가 필요하다.

마지막은 PIM driver이다. linux driver를 사용하면 DRAM에 데이터를 읽고 쓸 수 있으며 DRAM 내장 처리 장치와 통신이 가능하다.

2.4 sort-merge join algorithm

데이터베이스에서, join은 두 개 이상의 테이블을 하나의 집합으로 만드는 연산이다. sort-merge join 알고리즘은 두 개의 테이블에서 각각 join 대상을 먼저 읽은 후 정렬하고 merge하여 join을 수행한다. 각 테이블에 대해 동시에 독립적으로 데이터를 먼저 읽어 들이고 읽힌 각 테이블의 데이터를 join column 기준으로 정렬한 후, join 작업을 수행한다.

NL Join을 수행할 경우에는 random access 방식으로 데이터를 읽기 때문에 넓은 범위의 데이터를 처리할 때 부담이 된다. 반면 sort-merge join은 PGA에서 수행되어 random access 부하가 없고, full table scan 방식으로 넓은 범위의 데이터를 처리할 때 유용하다. 일반적으로 대량의 join 작업에서 정렬 작업을 요구하는 sort-merge join보다 CPU 작업 위주로 처리하는 hash join이 성능상 유리하지만, join 조건으로 equal 연산자를 사용하는 Equi Join에서만 사용 가능한 hash join과 달리 Non-Equi Join에 대해서도 작업이 가능하다. 또한 join column의 인덱스가 존재하지 않을 경우에도 사용할 수 있다는 장점이 있다. 그러나 정렬할 데이터가 많아 메모리에서 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역으로 디스크를 사용하기 때문에 성능이 저하될 수 있다.

따라서 sort-merge join 알고리즘의 성능 개선점은 다음과 같다. 데이터가 대량일 때 정렬 단계에서 더 효율적으로 메모리를 관리하며, 테이블에 접근하는 속도와 정렬 속도를 향상하고, 두 테이블 중 어느 한쪽이라도 정렬 작업이 종료되지 않으면 한쪽이 대기 상태가 되고 다른 한쪽의 정렬이 완전히 끝날 때까지 join 작업이 시작될 수 없으므로 양쪽의 정렬 완료 시점을 맞춘다.

2.5 연구개발 계획

본 프로젝트에서 활용할 UPMEM-PIM의 구조 및 특징을 파악한다. 이를 토대로 UPMEM-PIM 프로그래밍 방법을 숙지하고, UPMEM-PIM의 특징을 고려한 sort-merge join 알고리즘을 개발 및 구현한다. 구현한 알고리즘의 성능을 실험하기 위해 자체 dataset을 만들고 실제로 적용 후 결과를 분석한다. 이후 실험 결과를 바탕으로 UPMEM-PIM을 사용하지 않았을 때와 성능을 비교한다.

3. 프로젝트의 기술적 내용

3.1 개발환경

- PIM 하드웨어: V1B
- OS: Ubuntu 20.04.6 LTS
- UPMEM DPU toolchain(version 2023.2.0)

3.2 알고리즘

PIM을 활용한 sort merge join 알고리즘은 크게 select, sort, join 세 단계로 구분할 수 있다. 두 개의 테이블에 대해 select와 sort과정을 진행하고 해당 결과에 대해 join을 수행한다. 이때, 두 개의 테이블에 대해 select와 sort 과정을 병렬로 처리하기 위해 데이터의 row 개수를 사용 가능한 DPU 개수만큼 나누어 할당한다.

- **select** (select.c)

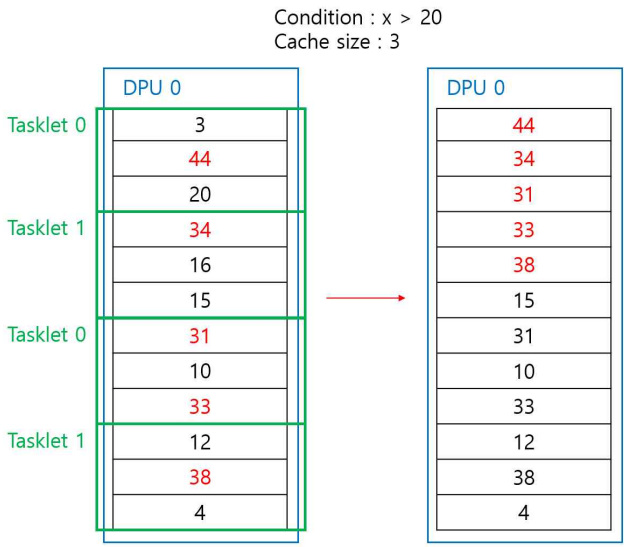


그림 2 select.c 알고리즘 예시

여러 DPU에서 실행하므로 전체 데이터를 DPU 개수에 맞춰 분할하여 할당한다. 각 DPU 내의 tasklet에서는 CACHE_SIZE를 참고하여 cache_A, cache_B를 정의하고 for문을 통해 select를 실행한다. select는 MRAM에서 일정 사이즈의 데이터를 읽어와서 WRAM에 저장한 후, 조건에 맞는 row를 선별하고 MRAM에 적는 방식으로 이루어진다. 모든 tasklet은 동시에 read를 수행하며 write될 위치를 알아야 하므로 handshake_sync를 활용하여 tasklet id가 작은 순서대로 write를 수행한다. 이를 할당받은 데이터에 대해 전부 select가 마칠 때까지 수행하여 MRAM에는 최종적으로 select를 마친 데이터가 존재한다.

- **sort** (sort_dpu.c, merge_dpu.c)

select 과정을 마친 데이터는 정렬이 되지 않은 데이터이다. sort-merge join은 정렬된 두 테이블에 대해 join을 수행하므로 정렬 과정이 필요하다. sort_dpu.c에서는 현재 DPU에서 갖고 있는 데이터에 대한 정렬을 수행하며, merge_dpu.c에서는 sort_dpu.c를 거쳐 정렬된 두 개의 DPU에서 도출한 데이터를 하나로 통합한 정렬된 데이터로 만드는 과정을 수행한다. merge_dpu.c를 토너먼트와 같은 방식으로 반복 수행해 각 DPU에 분할되어 있는 데이터들을 통합하여 정렬해 하나의 정렬된 테이블로 만든다.

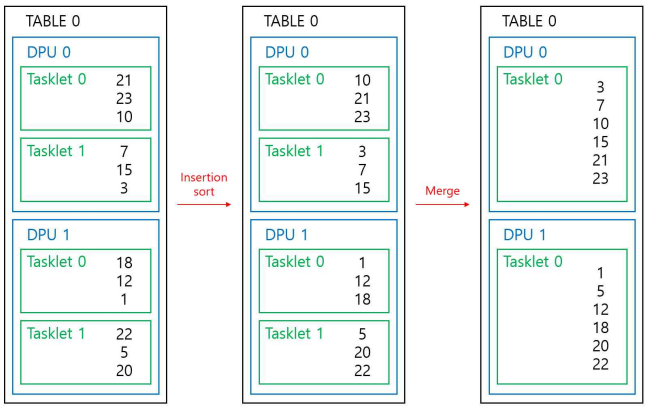


그림 3 sort_dpu.c 알고리즘 예시

sort_dpu.c에서는 tasklet에 할당된 데이터를 insertion sort를 통해 정렬 후, 정렬된 tasklet의 결과를 merge하면서 정렬한다. 이 과정은 merge sort 알고리즘 중 분할 과정이 제거된 방식과 유사하다. 또한 재귀 호출이나 추가적인 공간 사용이 없도록 in-place하게 동작할 수 있는 알고리즘을 설계하고 구현하였다. 그림 5와 같이 merge를 진행한다.

merge_dpu.c에서는 두 개의 DPU에 저장되어 있는 결과를 하나의 정렬된 데이터로 만들어 더 작은 번호의 DPU에 저장한다. 이때의 merge 과정은 sort_dpu.c에서 tasklet의 결과를 merge하는 과정과 같으며, host에서 while문을 반복하며 DPU에 데이터를 할당하는 것을 반복한다. 이를 통해 select와 sort를 거친 table 0과 table 1의 결과는 각각 DPU 0과 DPU pivot_id에서 가져올 수 있다.

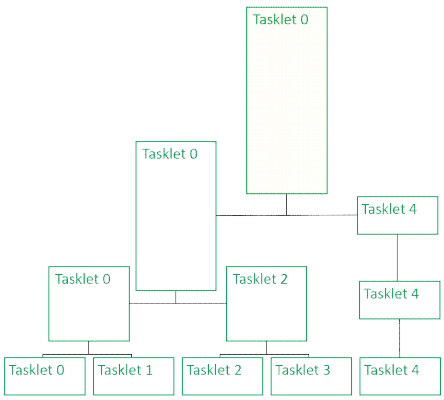
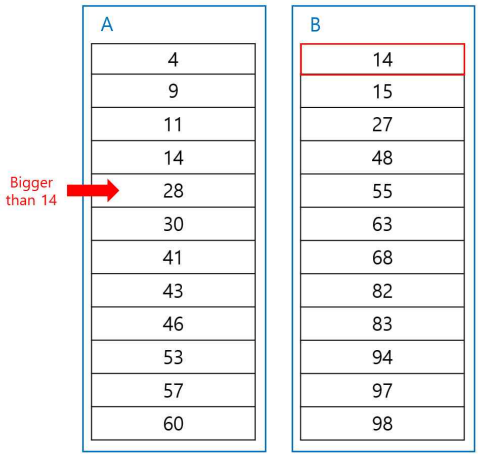
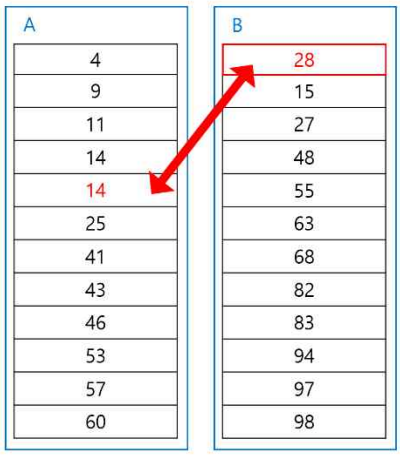


그림 4 merge.dpu의 토너먼트 방식

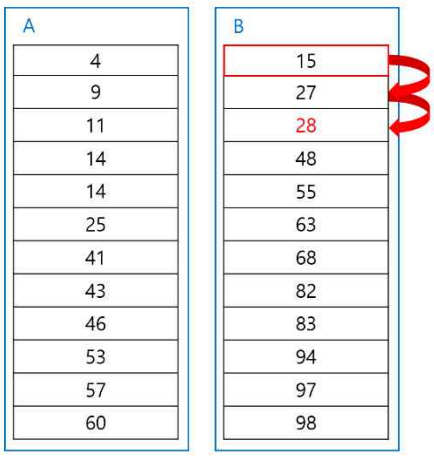
Step #1. Find the value bigger than the first value of A in B.



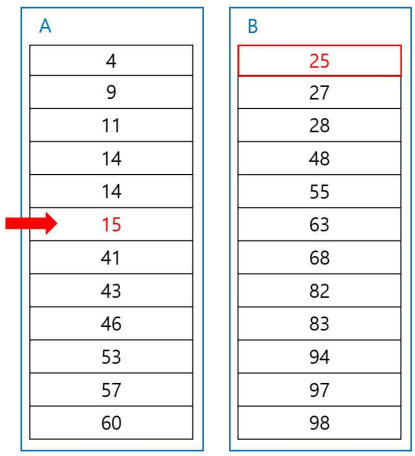
Step #2. Swap the values.



Step #3. Make sequential comparisons and change positions.



Step #4. Move to the next value of A, perform Step 1 ~ 3.



Step #5. If A is fully traversed, terminate.

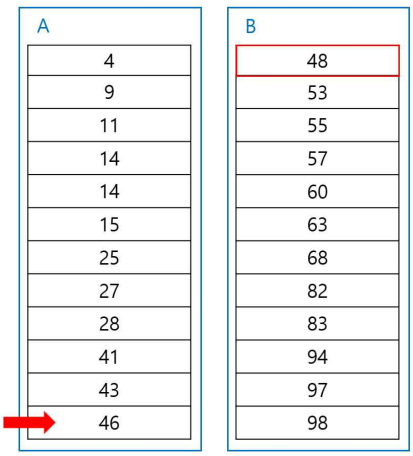


그림 5 merge 알고리즘 예시

- **join** (join.c)

DPU pivot_id에서 나온 결과는 두 번째 table이므로, DPU 0부터 DPU pivot_id – 1만큼의 DPU를 사용하여 join 과정을 수행한다. 먼저 host에서는 사용할 DPU의 개수에 비례하여 table 0의 데이터를 나눈 후, 각 DPU에 할당할 데이터의 join column의 값 중 가장 큰 값이 table 1의 전체 데이터 중 어디에 위치하는지 binary search로 탐색한다. 이 결과를 바탕으로 각 DPU에 할당될 table 0의 일부 데이터가 join을 수행할 table 1의 일부 데이터의 영역을 확인한다.

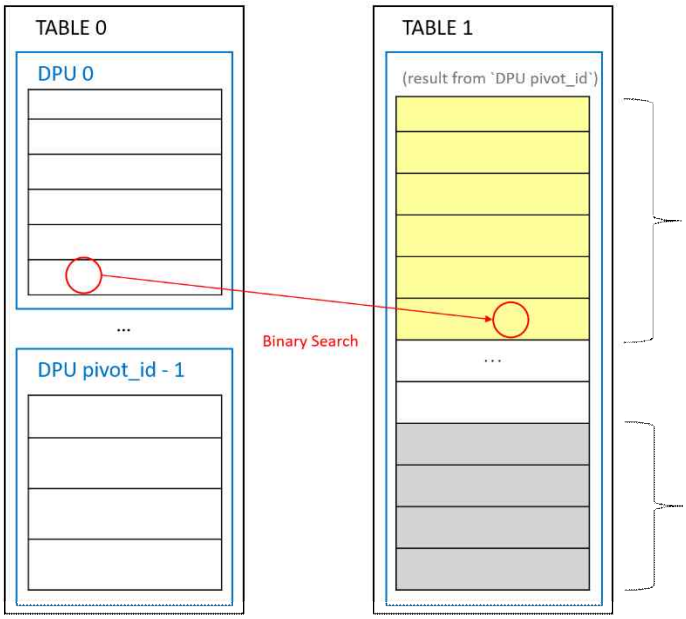


그림 6 join.c 알고리즘 예시

각 DPU에서 처리할 table 0의 데이터와 table 1의 데이터의 정보는 각각 input_args[i]와 input_args[pivot_id + i]에, 실제 데이터는 dpu_result[i]와 dpu_result[pivot_id + i]에 저장한다. (i = 0, ..., pivot_id – 1)

DPU에서의 과정은 merge_dpu.c와 app.c에서 join할 데이터를 할당하는 과정과 유사하다. tasklet의 개수에 맞춰 table 0 데이터를 할당한 후, 해당 데이터를 table 1 데이터에 어느 영역에 해당하는지 binary search를 통해 찾는다. 이후 각 tasklet은 두 테이블에서 join될 row가 몇 개 있는지 확인 후, 이를 바탕으로 join결과를 write할 MRAM 주소를 계산하고, 다시 두 테이블을 sequential하게 탐색하며 매치되는 row를 join한다. join 결과는 앞서 계산한 MRAM 주소부터 모든 tasklet이 동시에 작성한다. 이로써 PIM을 활용한 sort merge join의 동작을 완료하였다.

3.3 실행 과정

github repository(https://github.com/5eoyeon/upmem-pim)에서 산출물을 확인할 수 있다. main branch가 최신 버전이며, upmem-pim/upmem_env.sh을 source 후 실행 가능하다.

- **알고리즘 실행**

pim-sort-merge-join-main/sort-merge-join/data에 sort-merge join을 적용할 데이터 파일(data1.csv, data2.csv)을 넣는다.

```
#define DEBUG

#define NR_DPUS 64
#define NR_TASKLETS 16

#define SELECT_COL1 0
#define SELECT_VAL1 5000

#define SELECT_COL2 0
#define SELECT_VAL2 5000

#define JOIN_KEY1 0
#define JOIN_KEY2 0
```

그림 7 user.h

pim-sort-merge-join-main/sort-merge-join/user.h에서 사용자 커스텀이 가능하다.

- DEBUG 모드
- 사용할 DPU, tasklet 개수 (NR_DPUS, NR_TASKLETS)
- data1.csv에서의 select column과 select 기준값 설정(SELECT_COL1, SELECT_VAL1)
- data2.csv에서의 select column과 select 기준값 설정(SELECT_COL2, SELECT_VAL2)
- data1.csv에서의 join column(JOIN_KEY1)
- data2.csv에서의 join column(JOIN_KEY2)

pim-sort-merge-join-main/sort-merge-join/run.py을 실행하면 input으로 넣은 데이터에 대한 결과를 확인할 수 있다. 결과는 pim-sort-merge-join-main/sort-merge-join/data/result.csv로 저장된다. 아래는 시뮬레이터에서 실행한 예시이다.

```
seoyeon@LAPTOP-MD12S88L ~/upmem-pim(main)$ source upmem_env.sh
Setting UPMEM_HOME to /home/seoyeon/upmem-pim and updating PATH/LD_LIBRARY_PATH/PYTHONPATH
seoyeon@LAPTOP-MD12S88L ~/upmem-pim(main)$ cd sort-merge-join
seoyeon@LAPTOP-MD12S88L ~/upmem-pim/sort-merge-join(main)$ python3 run.py

========================================
Command: make
Output:
gcc -o cpu_app cpu_app.c
gcc --std=c99 app.c     -o app `dpu-pkg-config --cflags --libs dpu`
dpu-upmem-dpurte-clang -DNR_TASKLETS=16 -o select select.c
dpu-upmem-dpurte-clang -DNR_TASKLETS=16 -o sort_dpu sort_dpu.c
dpu-upmem-dpurte-clang -DNR_TASKLETS=16 -o merge_dpu merge_dpu.c
dpu-upmem-dpurte-clang -DNR_TASKLETS=16 -o join join.c

========================================
Command: ./cpu_app ./data/data1.csv ./data/data2.csv
Output:
######### CPU #########
### SORT-MERGE-JOIN ###
       EXEC TIME
Time (ms): 356.093000
########################

========================================
Command: ./app ./data/data1.csv ./data/data2.csv
Output:
######### PIM #########
### SORT-MERGE-JOIN ###
         EXEC TIME
CPU-DPU  156.782000
DPU      23491.133000
DPU-CPU  133.827000
----------------------
TOTAL 23781.742000
########################

========================================
Command: make clean
Output:
rm -f cpu_app app select sort_dpu merge_dpu join

========================================
```

그림 8 시뮬레이터 환경에서의 실행 결과(1)

3.4 실험 결과

해당 실험에서는 input으로 4개의 column과 각각 10만, 20만, 30만, 50만 개의 row로 이루어진 두 개의 csv 파일을 사용하였다.

- **실험 1 - 영역에 따른 실행 시간**

아래는 256개 DPU, 16개 tasklet을 사용하여 실행한 결과이다.

(시간 단위: ms)

	100000	200000	300000	500000
CPU-DPU	337.067	575.17	813.559	1363.736
DPU	36982.893	146000.728	326592.392	899462.524
DPU-CPU	577.183	1099.546	1626.682	2668.132

표 1

PIM을 활용한 실험에서는 CPU에서 DPU로의 이동 시간, DPU에서의 실행 시간, DPU에서 CPU로의 이동 시간을 나누어 측정하였다. 데이터의 규모가 크더라도 데이터 이동 시간의 부담이 크지 않다는 것을 확인했다. 실험 이전에는 host 코드 내에서 CPU와 DPU간의 데이터 이동이 많아 이 부분에 대한 성능 우려가 있었으나, 결과 확인 후 데이터 규모의 증가에 따른 DPU에서의 오버헤드가 성능 저하의 가장 큰 원인이라고 판단했다. 따라서 DPU 실행 처리를 더 최적화할 필요가 있다.

- **실험 2 - PIM 적용 유무에 따른 실행 시간**

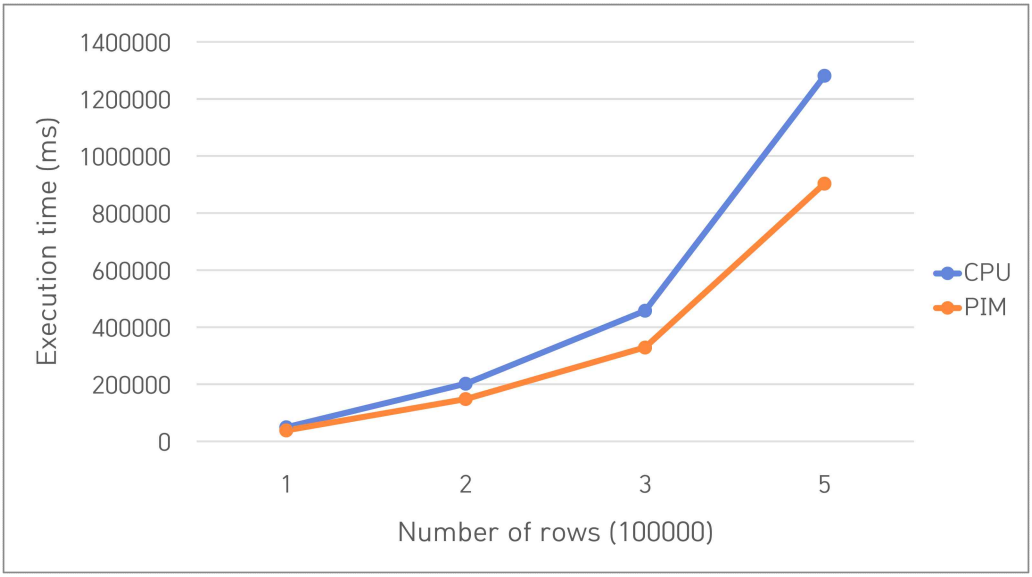
아래는 PIM을 적용하지 않은 CPU 버전의 sort merge join 코드로 실행한 결과와 비교한 것이다. 256개 DPU, 16개 tasklet을 사용하여 실행한 결과이다.

(시간 단위: ms)

	100000	200000	300000	500000
CPU	49493.047	202203.173	457831.413	1281479.55
PIM	37897.143	147675.444	329032.633	903494.392

표 2

실제 PIM 서버에서의 결과를 보면, 데이터셋의 크기가 증가할수록 CPU 환경에서의 실행 시간과 PIM 활용한 실행 시간의 차이가 증가함을 확인할 수 있다. 따라서 처리할 데이터의 양이 증가할수록 PIM을 사용하는 것이 효율적이다.



그래프 1 데이터 크기에 따른 CPU와 PIM 환경에서의 실행 시간

- **실험 3 - DPU 개수에 따른 실행 시간**

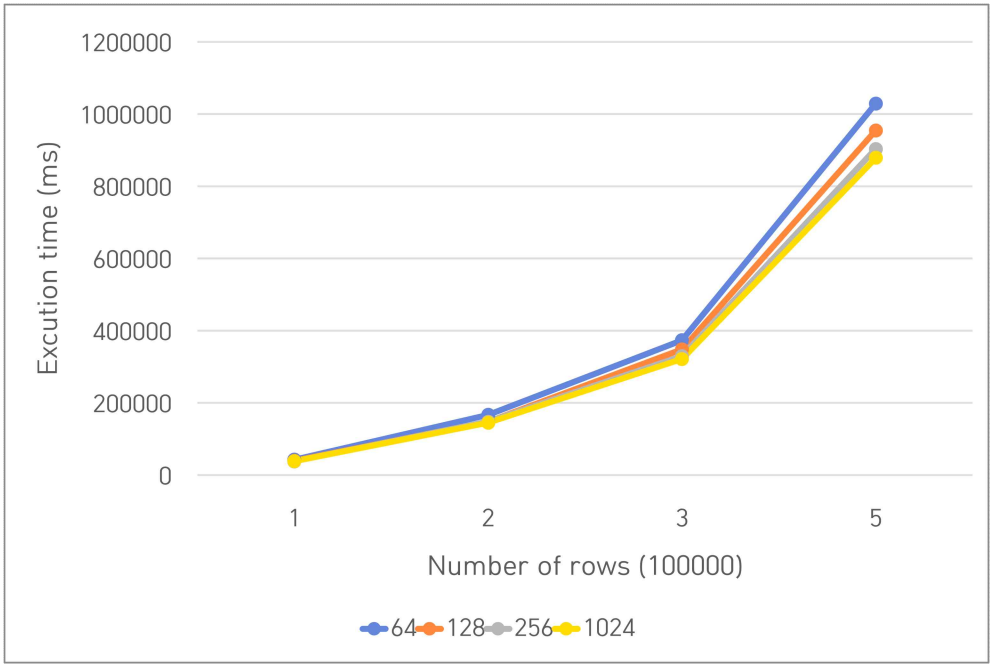
아래는 각 데이터셋에 대해 64개, 128개, 256개, 1024개 DPU, 16개 tasklet을 사용하여 실행한 결과이다.

(시간 단위: ms)

	100000	200000	300000	500000
64	42574.258	166677.818	373294.603	1029115.293
128	39492.563	147569.903	348234.689	955048.738
256	37897.143	147675.444	329032.633	903494.392
1024	38444.007	145124.691	321721.173	878914.237

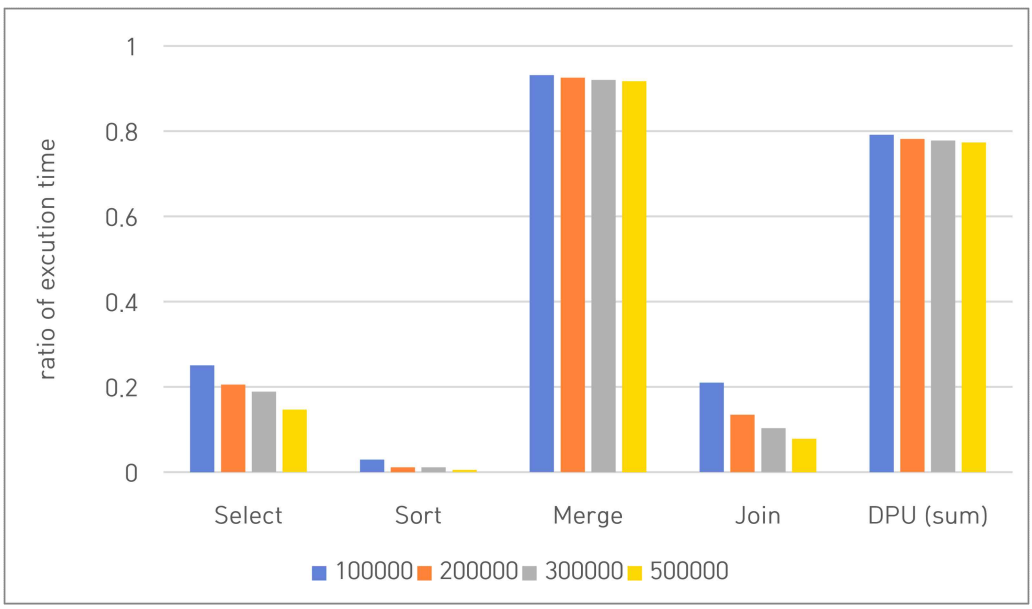
표 3

데이터가 클수록 DPU 사용 개수에 따른 실행 시간의 차이가 더 크다. 즉 처리할 데이터가 많다면 더 많은 DPU를 사용하는 것이 효율적이다. 그러나 비교적 규모가 작은 데이터에 대해 DPU 사용 개수 증가에 따른 실행 시간 감소폭이 크지는 않음을 확인할 수 있다. 이는 아래 그래프 2를 통해 확인할 수 있다.



그래프 2 DPU 개수에 따른 실행 시간

그 이유는 merge_dpu.c(DPU에 나눠져 있는 table의 데이터를 하나로 병합하는 과정)을 그림 4와 같이 반복하기 때문이다. 실험 1의 결과를 보면, DPU에서의 실행 시간이 전체 실행 시간에서 많은 부분을 차지하고 있다. 64개 DPU, 1024개 DPU를 사용해 각 데이터셋에 대해 실행하였을 때(모두 16개 tasklet를 사용), select, sort, merge, join 각각의 과정에서 소요된 시간을 측정하고 실행 시간의 비율을 구하였다. 아래 그래프 3은 그 값을 그래프로 표현한 것이며, 0에 가까울수록 실행 시간의 차이가 크다.



그래프 3 DPU 개수에 따른 알고리즘 별 성능 향상 정도

그래프 3을 보면 merge 과정에서는 실행 시간이 줄어들지 않았으며 오히려 비율이 1을 넘어가 약소하게 증가한 것을 볼 수 있다. 이는 merge가 DPU에서 처리한 데이터를 병합하는 알고리즘이고, DPU의 개수가 늘어날 수록 더 많은 단계를 필요로 하기 때문이다. 전체 table을 더 작게 나누어 병렬로 실행하므로 select, sort, join에 대해서는 성능 개선이 있었고 이는 데이터의 크기가 커질수록 뚜렷했다.

반대로 merge 과정에서 시간이 감소하는 경우도 있었는데, 이는 전체 table을 더 작게 나누어 정렬하기 때문에 정렬 자체의 실행 시간은 감소하였기 때문이다. 즉 DPU를 사용했을 때 증가하는 경우는 loop의 깊이가 더 커졌기 때문이고, 감소하는 경우는 더 작은 크기의 데이터셋에서 정렬을 마치며 병합하기 때문에 이후 각 단계의 정렬에서 더 빠르게 수행이 가능하기 때문이다. 이 두 가지 양상으로 인해 실행 시간의 증감이 상쇄되어 DPU 개수에 따른 실행 시간의 큰 차이가 없다.

- **실험 4 - tasklet 개수에 따른 실행 시간**

아래는 256개 DPU, 16개 tasklet을 사용하여 실행한 결과이다.

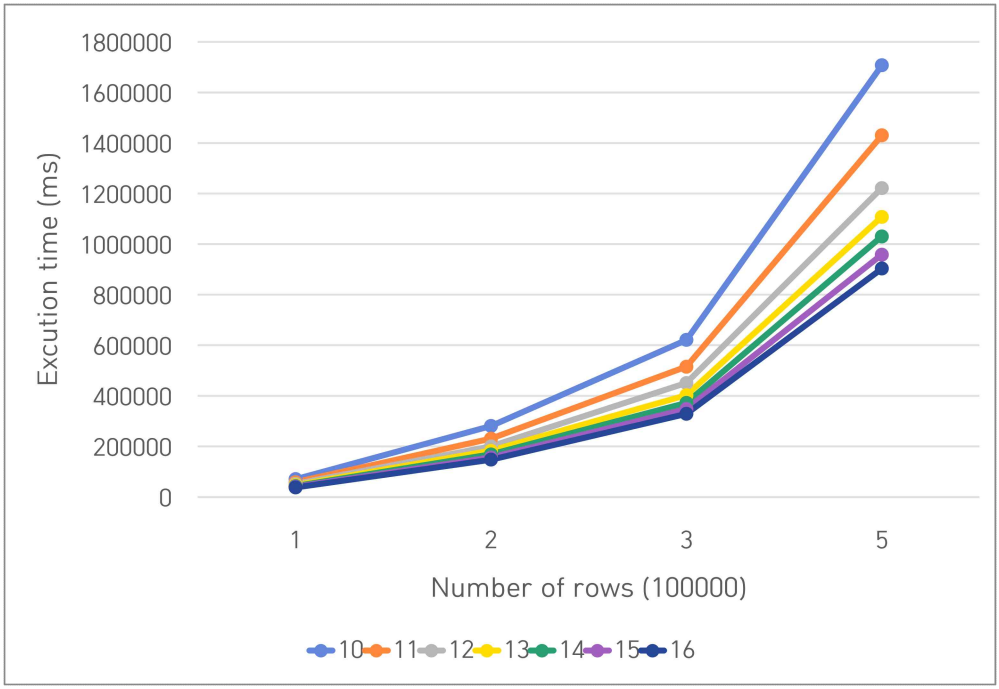
(시간 단위: ms)

	100000	200000	300000	500000
10	70743.954	281219.737	621620.182	1708639.173
11	58696.843	230888.953	515656.332	1430548.578
12	51339.027	199869.255	450711.793	1222208.989
13	46330.609	182801.673	402281.93	1108123.268
14	42896.735	168266.786	371853.85	1030884.233
15	40456.86	156176.676	348053.827	958821.114
16	37897.143	147675.444	329032.633	903494.392

표 4

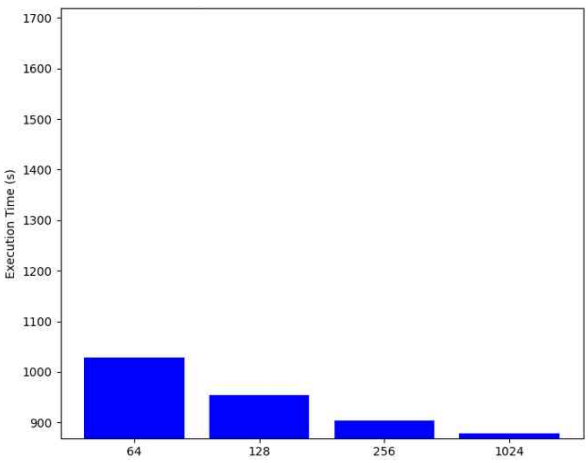
실험 결과, 사용하는 tasklet의 개수가 많을수록 실행 속도가 더 빠르다는 것을 확인할 수 있다. 그러나 현재까지 PIM에 대해 진행된 연구들에 따르면, 11개 이상의 tasklet에서는 성능 향상 측면에서 큰 차이가 없었다. pipeline이 11개 cycle로 구성되기 때문이다.

본 프로젝트에서는 사용하는 tasklet 개수가 증가할수록 실행 시간이 감소하였다. 이는 구현 상에서 사용하는 tasklet의 개수가 많아짐에 따라 처리할 데이터의 양이 줄어들기 때문이다. tasklet의 개수가 11개를 초과하면서 pipeline이 full 상태가 되어 처리 과정에서 clock을 공유하지는 못하는 것과는 별개로, 각 tasklet이 처리해야 할 데이터가 줄어듦에 따라 처리에 필요한 cycle 수 또한 줄어들어 tasklet의 종료 시점이 빨라지기 때문이라고 생각한다. 따라서 pipeline 활용 최대치를 초과하더라도 각 tasklet의 처리량 자체를 감소시키면 효율성을 높일 수 있을 것이다.

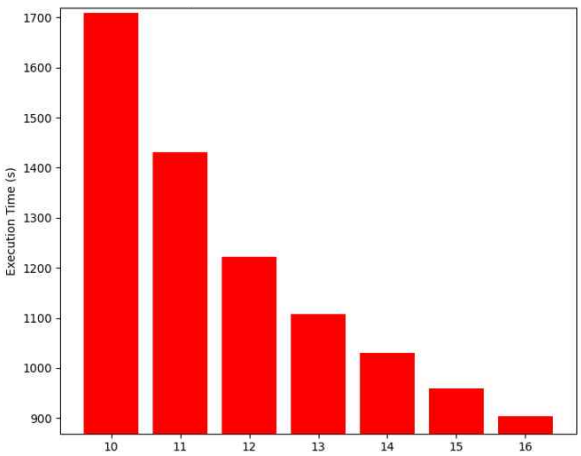


그래프 4 tasklet 개수에 따른 실행 시간

아래의 그래프 5는 DPU의 개수에 따른 실행 시간의 변화이고, 그래프 6은 tasklet의 개수의 따른 실행 시간의 변화이다. 두 그래프를 비교하면, 본 코드에서는 DPU의 개수 증가보다 tasklet의 개수 증가가 성능 향상에 더 큰 영향을 준다는 것을 확인할 수 있다.



그래프 5 DPU 개수에 따른 실행 시간



그래프 6 tasklet 개수에 따른 실행 시간

3.5 Trouble Shooting

- **WRAM을 사용한 data transfer 중 발생하는 overflow 해결**

기존 코드에서는 MRAM을 사용하지 않고 DPU 코드 내에서 데이터를 주고 받기 위해 배열을 최대 크기로 초기화하고 있었다. WRAM의 크기가 제한되어 있어 이러한 방식으로는 처리할 수 있는 범위를 넘어선다. host가 dpu에게 직접적으로 주는 정보는 테이블의 column

개수와 row 개수 뿐이며 데이터는 MRAM의 `DPU_MRAM_HEAP_POINTER_NAME`(DPU 프로그램의 사용되지 않은 MRAM의 시작) 부분에 쓰여진다. DPU는 받은 정보를 통해 크기를 알 수 있으므로 `mram_read`를 통해 읽어올 수 있다. DPU에서 host로의 전달도 동일하게 `col_num`과 `row_num`만 전달하며, 같은 위치에 write하여 host에서 접근이 가능하다. 이를 통해 메모리가 낭비되는 문제는 해결할 수 있었으며, 크기가 커지면 WRAM overflow로 인해 컴파일부터 되지 않는 에러를 픽스하였다.

- **tasklet 내 sort 방식 변경**

기존 설계에서는 재귀를 사용한 quick sort를 구현하였으나 함수의 재귀 호출로 인해 WRAM이 overflow되는 문제가 발생하였다. 이에 스택을 사용해 while문을 돌도록 변경하였다. 그러나 스택은 처리해야 할 범위의 처음과 마지막을 저장해야 하고 그로 인해 어느 정도의 고정 크기를 가지고 있어야 하기에 WRAM의 공간을 차지하는 문제는 동일하다. 또한 tasklet마다 quick sort가 실행되며 스택의 정의만으로 WRAM의 범위를 넘어갈 수 있다. 반대로 STACK_SIZE가 너무 작아지면 quick sort를 돌리는 게 불가능하다. 이에 다른 방법으로 quick sort를 구현하거나 정렬 방식을 변경해야 했다. 새로운 방식의 quick sort는 전체 데이터에 대해 데이터의 앞에서부터 순차적으로 한번 정렬하는 데이터의 크기를 지정하여 정렬이 이루어지며 그 외의 과정은 일반적인 quick sort와 같다. 반복만을 사용하여 추가적인 공간의 사용이 없다는 점에서 효율적이지만 전체 데이터에 대해 logN번 탐색을 반복하여 비효율적이다. 따라서 quick sort를 제외한 bubble sort, selection sort, insertion sort와 같은 다른 in-place 알고리즘을 적용해 보았다. 시간 복잡도에서는 우위를 가지지 못하지만 대규모 데이터에 대한 정상 동작과 성능 비교를 우선시하여 CPU와 DPU에서 모두 insertion sort를 쓰는 것으로 결정하였다.

- **할당받은 데이터가 없는 경우 예외 처리**

데이터의 row 개수가 DPU의 개수보다 적은 경우와 DPU에 할당된 row 개수가 tasklet의 개수보다 적은 경우에 대해 동작이 불가능했다. 전자의 상황이 발생한다면, 사용하는 DPU 개수를 2개로 변경하여 동작하도록 하였다. DPU 별로 적은 데이터를 할당할 경우 데이터 이동에 대한 오버헤드로 인해 하나의 DPU가 처리하도록 하는 것이 더 효율적이라는 판단에 근거한다. 후자의 상황에서는 대응 방법이 과정마다 상이하다. select.c에서는 cache size가 전체 데이터 크기를 넘어가지 않도록 처리하였으며 sort_dpu.c와 join.c에서는 DPU 개수를 변경했던 것처럼 사용하는 tasklet 개수를 수정하였다. merge_dpu.c에서는 해당 문제와 관련이 없기 때문에 변경점이 존재하지 않는다.

- **'stack smashing detected' 에러 해결**

만 개, 10만 개, 20만 개 등의 데이터를 처리할 때 join까지 완료되었음에도 불구하고 일부 데이터에 대해 stack smashing detected 에러가 발생했다. stack에서 에러가 발생하였기에 app.c를 검토한 결과 join 과정 처리 중 `input_args`과 `dpu_result`에서 배열 사이즈의 범위를 벗어난 값을 참조하고 있음을 확인하였다. 이는 pivot_id(table 1을 처리하는 첫 번째 DPU id)가 NR_DPUS / 2라고 가정하고 구현하였기에 발생한 오류였으며 논리의 문제는 없

었기에 단순히 `input_args`과 `dpu_result`의 배열의 크기를 늘려주어 해결하였다.

- **join.c 내 'DPU is in fault' 에러 해결**

해당 에러는 첫 번째 row에서 join_key가 동일하지 않다면 발생하지 않았다. 그러나 join.c에서 동일한 데이터를 join할 때 DPU is in fault 에러가 발생하였고, dpu-lldb를 사용하여 원인을 파악하였다. join을 수행하는 while문에 진입하면, join 전에 join할 row의 개수를 세기 위한 while문에서 마지막으로 read한 row를 사용하고 있어 발생한 문제였다. 따라서 join을 수행하는 while문에 진입하기 전에 각 테이블의 첫 번째 row를 read하도록 수정하여 해결하였다.

4. 프로젝트의 역할 분담

4.1 개별 임무 분담

번호	학과	학번	학년	이름	담당업무
1	컴퓨터소프트웨어	2021019961	4	장서연	계획, 설계, 개발, 실험
2	컴퓨터소프트웨어	2021097356	4	김한결	계획, 검증, 개발, 실험

4.2 개발 일정

구분	추진내용	수행기간											
		1월	2월	3월	4월	5월	6월	7월	8월	9월	10월	11월	12월
계획	졸업 프로젝트 제안서 작성												
분석	PIM 활용 프로그래밍 및 알고리즘 분석												
설계	PIM을 활용한 sort-merge join 설계												
	설계 검증												
개발	PIM을 활용한 sort-merge join 구현												
	unit 단위 구현 검증												
실험	PIM 서버를 사용한 실험 진행 및 분석												
종료	최종 보고서 작성												
논문작성	논문 작성												

5. 결론 및 기대효과

5.1 활용 기대효과

UPMEM PIM을 활용한 sort-merge join 알고리즘을 구현하였다. 또한 CPU 환경에서의 성능과 PIM을 사용하였을 때의 성능을 비교하여 분석한 결과를 도출하였다. 이를 바탕으로 데이터베이스 분야에서 대규모 데이터에 대해 PIM을 활용하여 성능을 향상시킨 사례를 제시하였고, 활용 가능성을 살펴보았다. 또한 PIM을 활용하여 구현 후 CPU 환경과 비교하며 PIM의 기술적 의의를 확인하였다. 이와 더불어 알고리즘 분야에서는 sort-merge join 알고리즘의 성능 개선점을 제시했으며 이것으로 다양한 환경에서도 성능을 개선할 수 있을 것이다.

5.2 추후 계획

추후 계획은 현재 완료한 프로젝트에 대해 부가적인 개선과 논문 작성이다. 아래와 같은 개선 사항을 들 수 있다.

- **cache size 증가**

큰 데이터에서는 cache size를 증가시키고 cache를 사용하는 부분을 늘림으로써 성능을 개선할 수 있다. 현재는 cache size는 256byte(혹은 이와 가장 가까운 column * sizeof(T)의 배수)이지만 일정 데이터 크기 이상이 되면 이보다 큰 수가 효율적일 수 있다. 또한 select에서만 cache를 사용하여 진행하는데 다른 알고리즘에서도 이런 방식을 차용할 수 있을 것이다. merge_dpu.c와 join.c에 적용하여 구현도 해보았지만 10000개 row의 데이터에 대해 오히려 더 성능이 저하되어 연구가 필요해 보인다.

- **정렬 알고리즘 개선**

본래 속도를 위해 quick sort를 사용하였지만 재귀적인 수행으로 인해 WRAM 크기를 뛰어넘는 문제가 있었고 우선 in-place 알고리즘 중 하나인 insertion sort를 사용하였다. 하지만 해당 방법의 시간 복잡도는 최선의 경우 O(n), 최악의 경우 O(n^2)이므로 빠르다고 하기 어렵다. quick sort를 다른 방법으로 구현하거나 다른 정렬 알고리즘(e.g. heap sort, shell sort 등) 또는 정렬 방식을 적용할 수 있을 것이다.

- **자원 활용률 개선**

tasklet이나 DPU의 data를 하나로 합치는 과정에서 최종적으로는 tasklet 0, DPU 0에 모이게 되므로 병합하는 작업에서 실제 동작이 없는 tasklet 또는 DPU가 존재한다. 하지만 이는 병합 구조에 의해 발생한 문제이므로 현 설계상 변경이 불가능하기도 하다. 다른 설계를 통해 해결할 수 있을 것이다.

- **테이블 병합 과정의 loop 개선**

실험 4의 결과를 보면, 각 테이블의 데이터를 DPU에서 나눠서 정렬한 후 다시 하나의 테이블로 병합하는 과정에서 사용하는 DPU의 개수에 따라 loop가 깊어지며 오버헤드가 커진다. 해당 부분에서 loop의 깊이가 얕아지거나 merge_dpu.c의 반복을 줄이는 것이 DPU에서의 실행 시간을 감소하는 데 큰 영향을 줄 수 있을 것이라고 생각한다.

5.3 경험 및 느낀점

프로젝트 진행 과정에서 기록의 중요성을 체감할 수 있었다. 개발 과정에서 동일한 오류가 발생하는 문제가 있었는데, 이전에 해결하였음에도 불구하고 마땅한 기록 체계가 없어 불필요한 오버헤드가 발생하였다. 해결 당시에는 명확하고 사소하다고 생각하여 넘겼으나 프로젝트가 진행되며 확실한 기록이 없다는 점에서 아쉬움을 느꼈다. 동시에 상대방의 코드를 확인하는 과정에서도 기록은 중요하게 작용할 수 있었는데, 보통 말로만 전달하는 방식을 취했기에 이해에 대한 착오가 있었다. 기록을 아예 하지 않은 것은 아니지만 발생 오류, 해결 방법, 진행 내용에 있어서 더 중점을 두고 코드 주석과 함께 자세히 적었다면 진행 시간을 단축할 수 있었을 것이다.

또한 정확한 설계와 구현의 검증이 필요함을 느꼈다. 프로젝트에 관련된 사전 지식을 제대로 학습하고, 설계 과정에서 더 정확하고 최적화된 구조와 이를 검증하여 튼튼한 설계를 완성했다면 보다 빠른 구현이 가능했을 것이라고 생각한다. 이후 개발 과정에서는 unit 단위의 정밀한 기능 검증과 디버깅을 통해 다양한 예외 상황 및 에러 상황을 정확하고 빠르게 처리할 수 있는 능력을 갖출 수 있었다.

이 모든 과정을 팀 프로젝트로 진행하며, 협업할 때에는 무엇보다도 형상 관리가 어렵다고 생각했다. branch 관리와 conflict 처리가 까다로워 앞으로 경험할 큰 규모의 프로젝트에서는 형상 관리가 중요함을 배웠다.

대규모 데이터의 빠른 처리가 요구됨에 따라 Processing-In-Memory가 더 큰 활약을 할 수 있을 것이라 기대된다. 본 프로젝트를 통해 학부 수업에서 배웠던 병렬 처리와 동기화, 메모리 및 자원 관리, C언어 등을 활용 및 적용하였고, PIM에 대한 새로운 지식과 기술을 습득할 수 있었다. 이를 설계부터 개발, 검증, 실제 적용까지의 단계에서 경험하며 연구자와 개발자의 발판을 마련할 수 있었다.

6. 참고 문헌

[1] UPMEM, “UPMEM Processing In-Memory (PIM): Ultra-efficient acceleration for data-intensive applications”, UPMEM, 2022.

[2] Juan Gómez-Luna, et al., "Benchmarking a new paradigm: experimental analysis of a real processing-in-memory architecture.”, https://doi.org/10.48550/arXiv.2105.03814, 2022.

[3] Juan Gómez-Luna, et al., https://github.com/CMU-SAFARI/prim-benchmarks/tree/main/SEL

[4] 곽재혁, “UPMEM PIM의 HPC 분야 적용 가능성 연구”, 한국정보처리학회, 2022.

[5] Gavin JT Powell, “Oracle Data Warehouse Tuning for 10g”, Elsevier Sciencd, 2011.

[6] UPMEM SDK, https://sdk.upmem.com/2023.2.0/