[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 최종 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연
2021097356 김한결

2024.10.20.

목차

1. 알고리즘
2. 실행 과정
3. 실행 결과
4. 개선 사항
5. 결론
6. 참고자료

1. 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 사용 가능한 DPU의 개수를 데이터의 row 개수에 비례하여 나누어 할당한다.

- **select** (select.c)

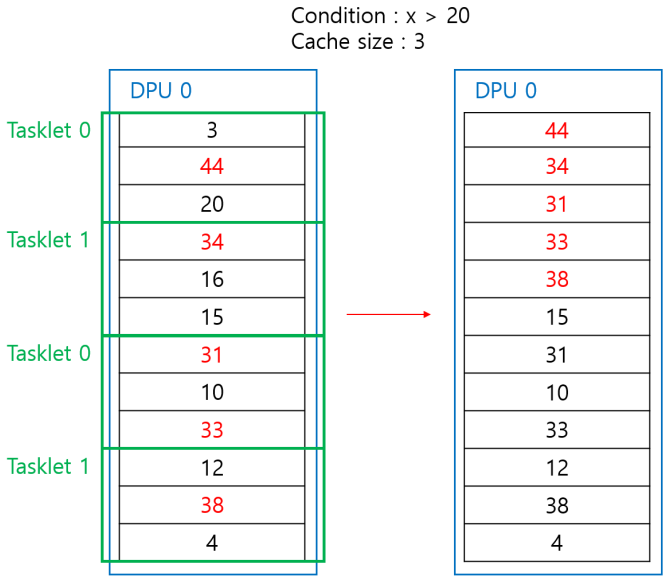


그림 1

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 할당 받은 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 과정

을 수행한다. merge_dpu.c를 토너먼트와 같은 방식으로 반복 수행해(그림 4) 각 DPU에 분할되어 있는 데이터들을 통합하여 정렬해 하나의 정렬된 테이블로 만든다.

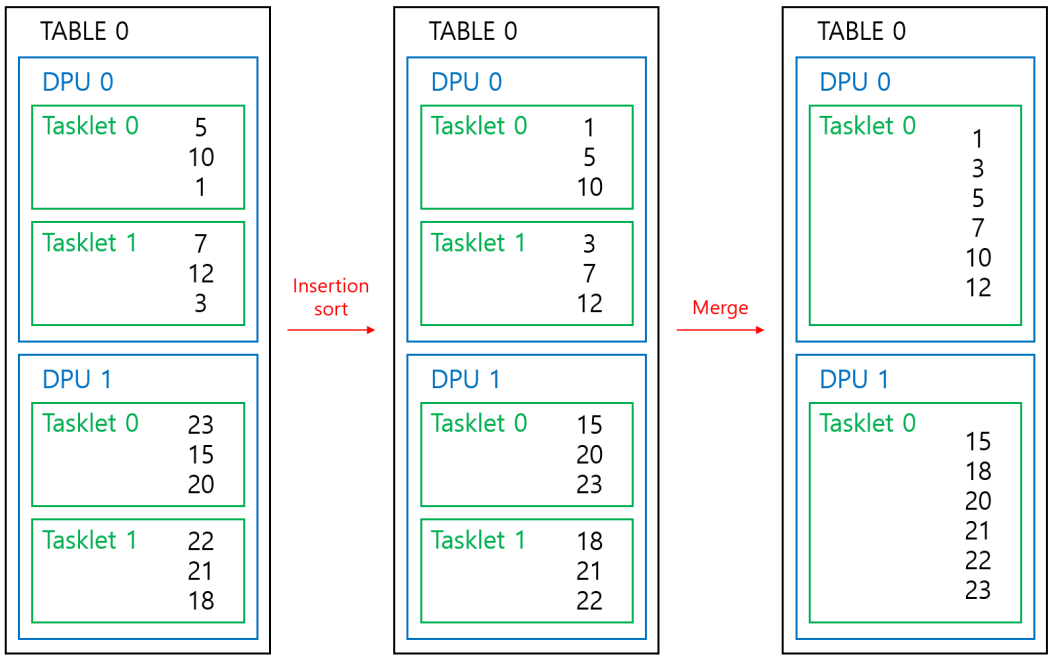
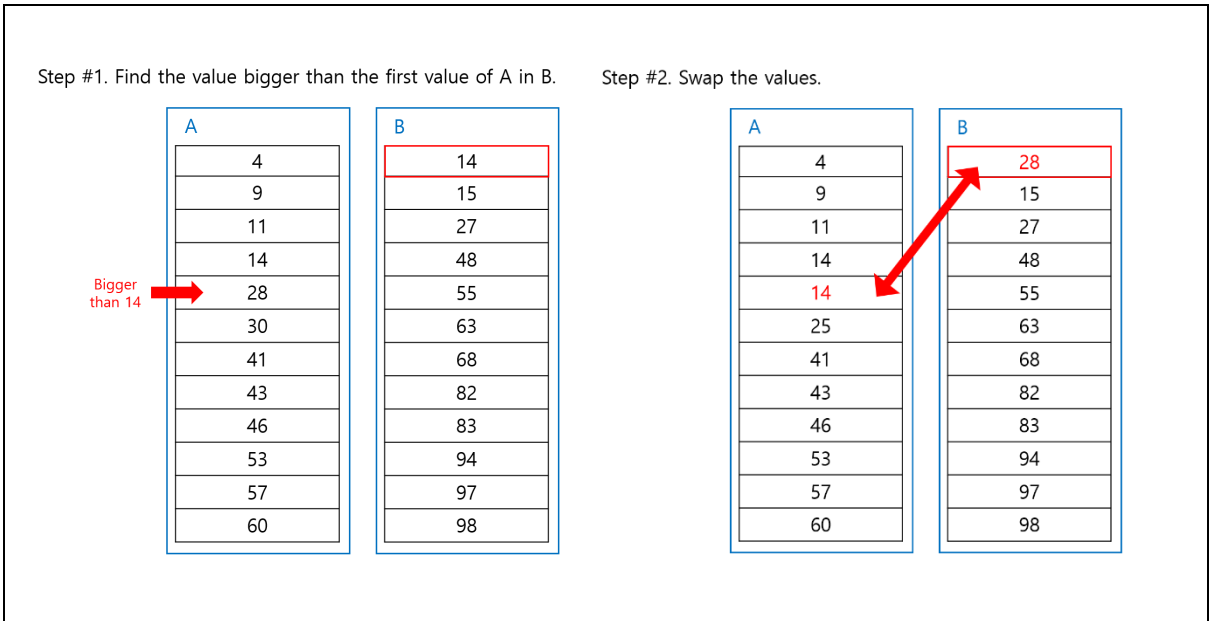


그림 2

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



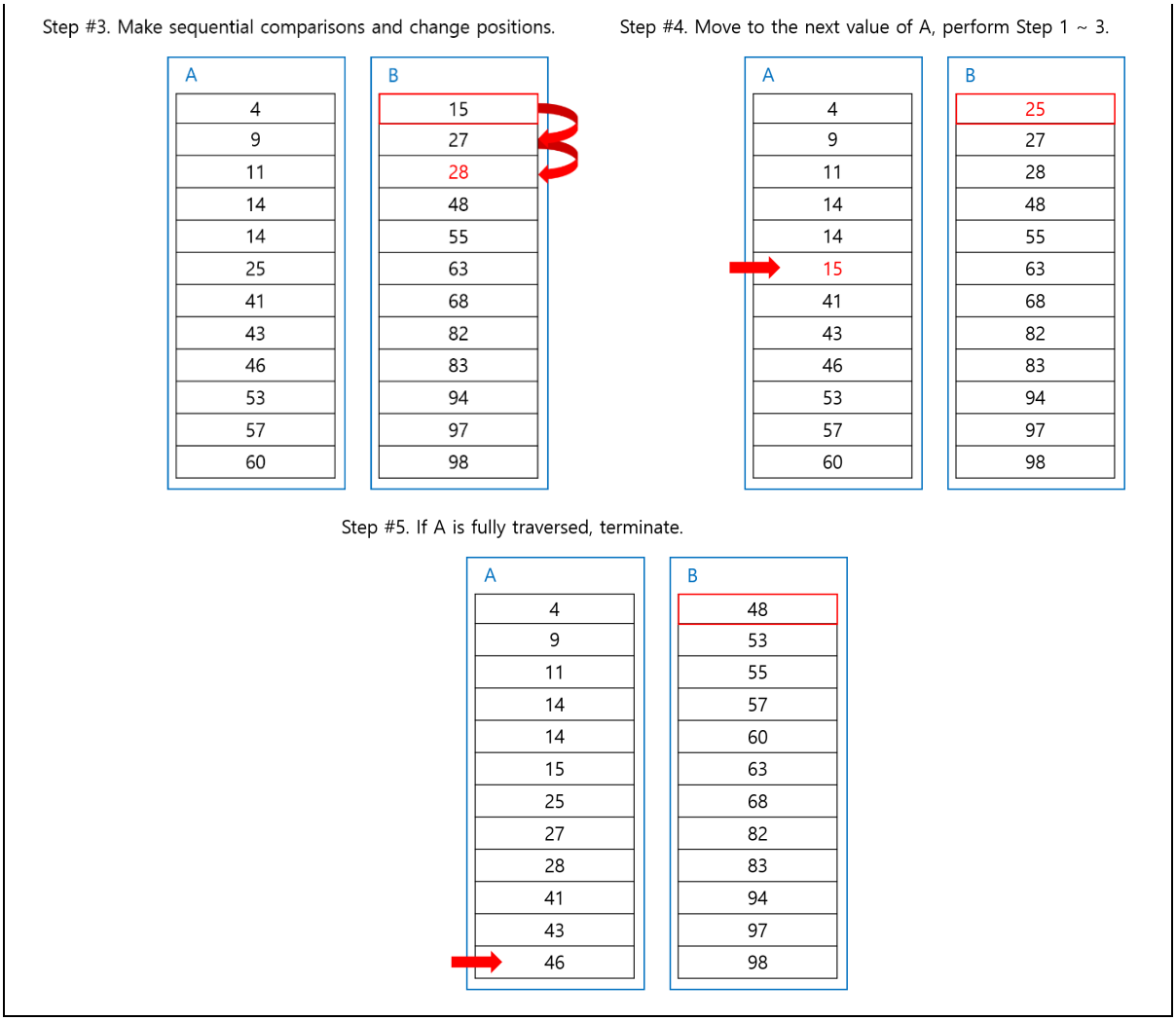


그림 3

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.

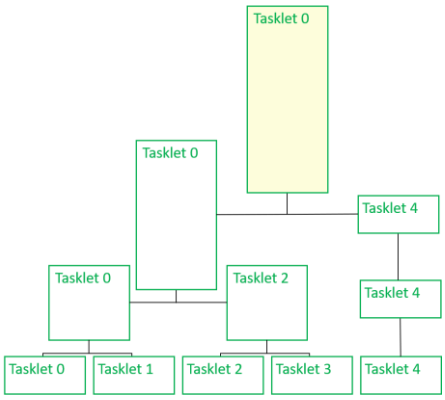


그림 4

- **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.

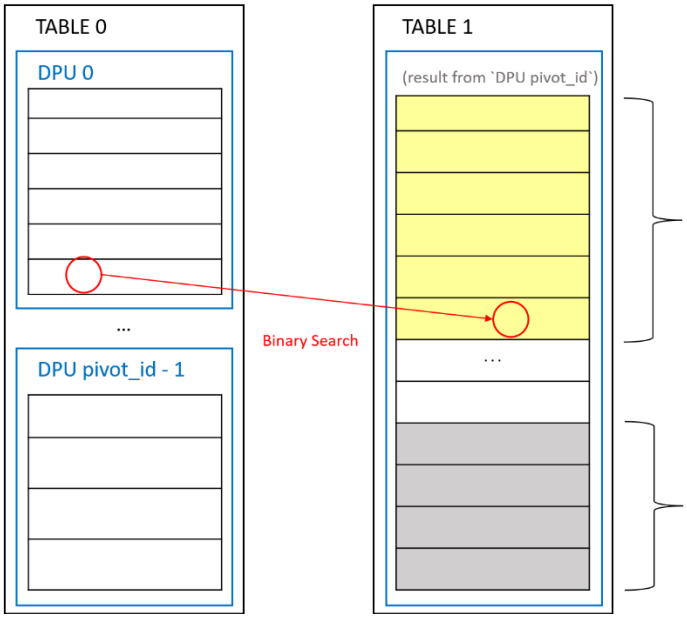


그림 5

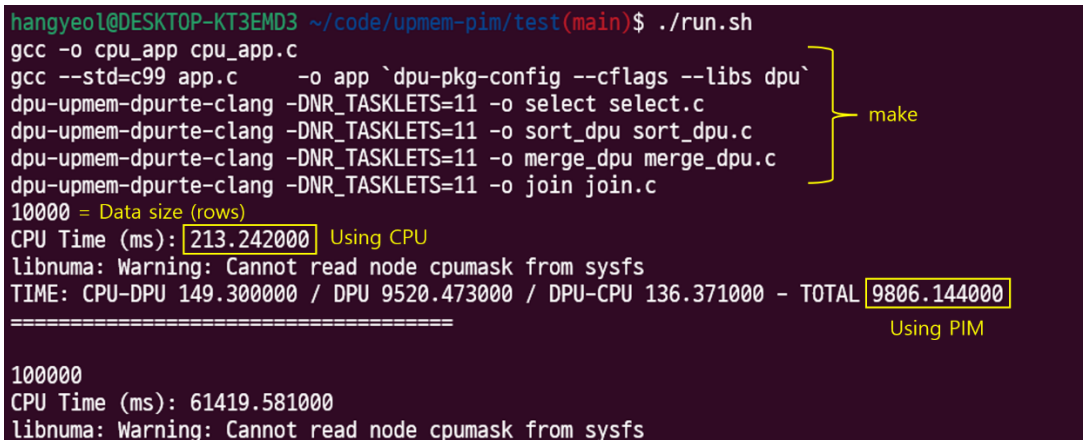
각 DPU에서 처리할 table 0의 데이터와 table 1의 데이터의 정보는 각각 input_args[i]와 input_args[pivot_id + i]에, 실제 데이터는 dpu_result[i]와 dpu_result[pivot_id + i]에 저장한다. (i = 0, ..., pivot_id – 1)

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.

2. 실행 과정

본 프로젝트에 대한 github repository(https://github.com/5eoyeon/upmem-pim)가 있다. 현재 main branch에서 upmem-pim/의 upmem_env.sh을 source 후, upmem-pim/test/run.sh을 실행하면 실행 결과를 확인할 수 있다.

아래는 시뮬레이터에서 실행한 예시이다.



3. 실험 내용 및 결과

OS는 Ubuntu 20.04.6 LTS로 통일하며, UPMEM DPU toolchain version 2023.2.0을 사용했다. 시뮬레이터를 사용해 개발 및 실험을 진행하였다.

64개 DPU, 11개 tasklet을 사용하여 실행한다. Input은 4개의 column과 각각 5천, 1만, 5만 개의 row로 이루어진 csv 파일을 input으로 사용하였다.

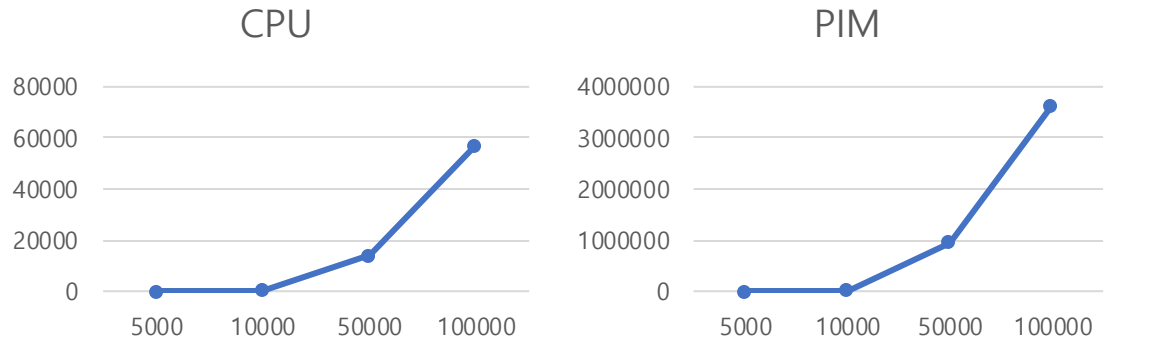
개인 PC에서 시뮬레이터를 사용할 때에는 성능과 하드웨어의 한계로 상대적으로 더 큰 규모의 데이터에 대해서는 실행을 하지 못하였다. 아래는 시뮬레이터에서 실험한 결과이다.

row	5000	10000	50000	100000
CPU-DPU	133.713	131.429	208.768	248.401
DPU	4011.246	9903.442	943980.487	3629379.55
DPU-CPU	129.676	103.032	171.124	187.795

PIM을 활용한 실험에서는 CPU에서 DPU로의 이동 시간, DPU에서의 실행 시간, DPU에서 CPU로의 이동 시간을 나누어 측정하였다. 이동 시간이데이터셋의 크기와 정비례하는 것은 아니므로 데이터셋의 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, 결과를 확인 후 DPU의 실행 시간이 성능 저하의 가장 큰 원인이라고 판단했다. 따라서 DPU 실행 처리를 더 최적화할 필요가 있다.

아래는 PIM을 적용하지 않은 CPU 버전의 sort merge join 코드로 실행한 결과와 비교한 것이다.

Row	5000	10000	50000	100000
CPU	75.282	158.353	14175.884	57219.937
PIM	4274.635	10137.903	944360.379	3629815.746



(시간 단위: ms)

수정 전 이전 코드이지만 연구실의 실제 서버를 사용한 결과는 다음과 같다. 이전 코드의 경우 수정이 필요하였으나, 해당 데이터에 대해서는 select와 sort, join까지 모두 정상적으로 수행 완료하였으므로 참고할 수 있는 데이터라고 판단하여 첨부한다. 256개의 DPU와 11개의 tasklet을 사용한 결과이다.

```
CPU Time (ms): 50315.754000
TIME: CPU-DPU 327.231000 / DPU 58649.939000 / DPU-CPU 609.678000 - TOTAL 59586.848000
```

(10만 개 row)

```
CPU Time (ms): 203050.490000
TIME: CPU-DPU 569.347000 / DPU 229318.394000 / DPU-CPU 1172.744000 - TOTAL 231060.485000
```

(20만 개 row)

시뮬레이터에서의 결과와 서버를 사용한 결과를 보았을 때, 시뮬레이터에서는 DPU를 64개까지 사용할 수 있다는 제한이 있고, 개인 PC 환경에서 실행하여 상대적으로 큰 데이터셋에 대해서는 실행 결과를 확인하기가 어려워 실험 결과의 차이가 크다. 유의미한 실험 결과를 위해서는 서버 사용이 필요할 것으로 보인다.

실험 결과, 데이터셋 크기에 따른 PIM 서버(시뮬레이터)에서의 소요 시간이 CPU 환경에 비해 더 가파르게 증가한다. 그러나 실제 PIM 서버에서의 결과를 참고하면, 이는 1. 시뮬레이터에서 실행한 결과이며, 2. 데이터셋의 크기가 작고, 3. DPU 개수 사용의 제한으로 인한 양상이라고 판단된다.

또한 실제 PIM 서버에서의 결과를 보면, 데이터셋의 크기가 증가할수록 CPU 환경에서의 결과와 비교한 PIM 활용한 결과의 차이 비율이 감소함을 확인할 수 있다. 따라서 처리할 데이터의 양이 증가할수록 PIM을 사용하는 것이 효율적이다. 또한 PIM의 성능이 CPU에 비해 더 좋아지는 시점이 있을 것이라고 생각하며, 이는 실제 서버를 사용한 결과를 첨부할 예정이다.

추가적인 실험(데이터셋의 크기에 따른 결과, DPU 개수에 따른 결과)은 이후 수정하여 첨부할 예정이다.

4. 개선 사항

- 현재는 main에서 인자로 두 개의 파일명만을 받고 있다. 그리고 common.h에서 NR_DPUS, NR_TASKLETS, SELECT_COL, SELECT_VAL, JOIN_KEY 등을 상수로 선언하여 사용하고 있다. 하지만 데이터에 따라 이러한 값들을 유연하게 바꾸는 편이 이상적이기에 main에서 해당 값들을 전부 인자로 받아 sort merge join을 수행하게 함으로써 편의성 측면을 개선할 수 있을 것이다. select할 때의 조건 값이 고정되어 있으므로 범위가 만, 10만, 20만으로 늘어날 때마다 동일 비율로 데이터가 줄어들지 않는 점도 실험에 있어서 문제가 될 수 있다.

- 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)지만 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 과정 역시 개선점이 될 수 있다. 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 방식(e.g. heap sort, shell sort 등)을 사용할 수 있을 것이다.

- load imbalance 문제도 존재한다고 본다. tasklet이나 DPU의 data를 하나로 합치는 과정에서 최종적으로는 tasklet 0, DPU 0에 모이게 되므로 merge에 한하여 작업을 균등하게 할당 받지 못하게 된다. 하지만 이는 DPU 간의 데이터 공유가 불가능하기에 발생한 문제이므로 현 구조상 변경이 불가능하기도 하다. 다른 설계 방식을 통해 해결할 수도 있을 것이다.

5. 참고 자료

[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] UPMEM SDK, https://sdk.upmem.com/2024.2.0/

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