[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 최종 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연
2021097356 김한결

2024.10.31.

목차

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)

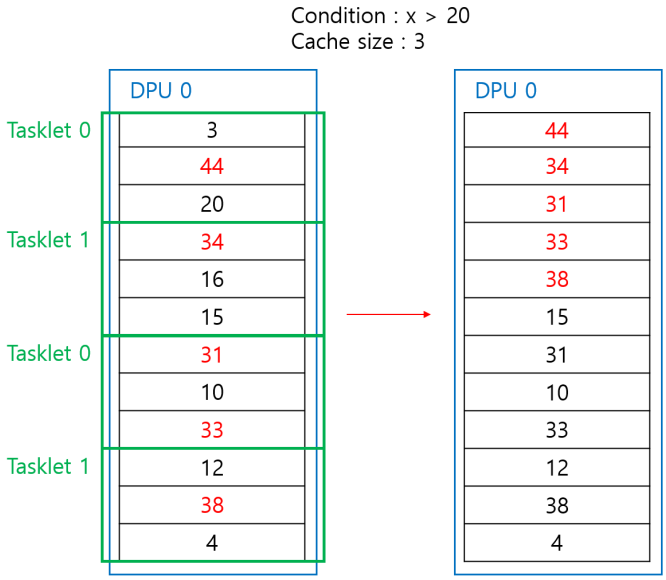


그림 1

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.

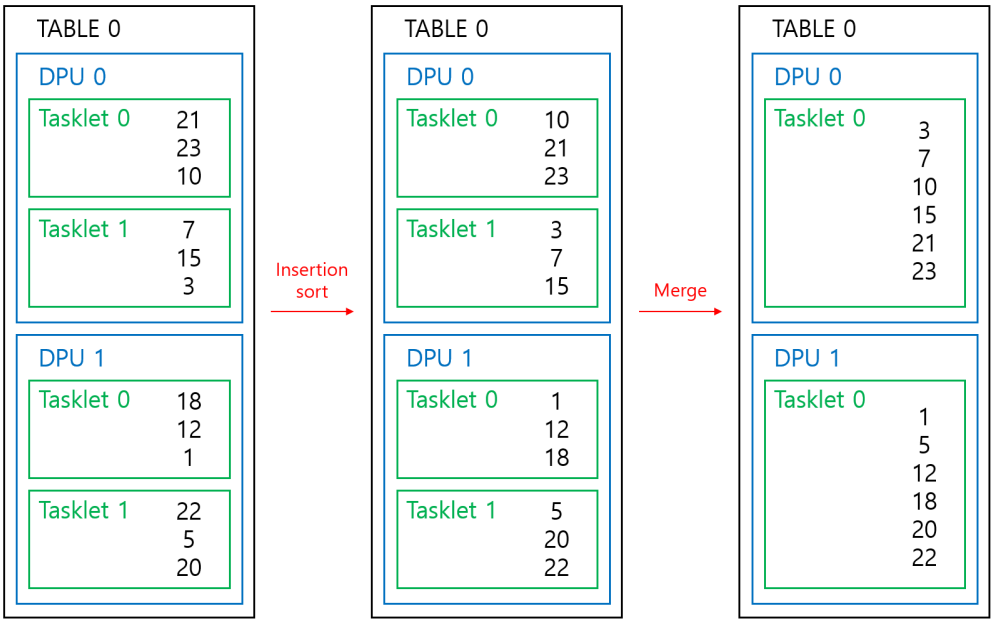
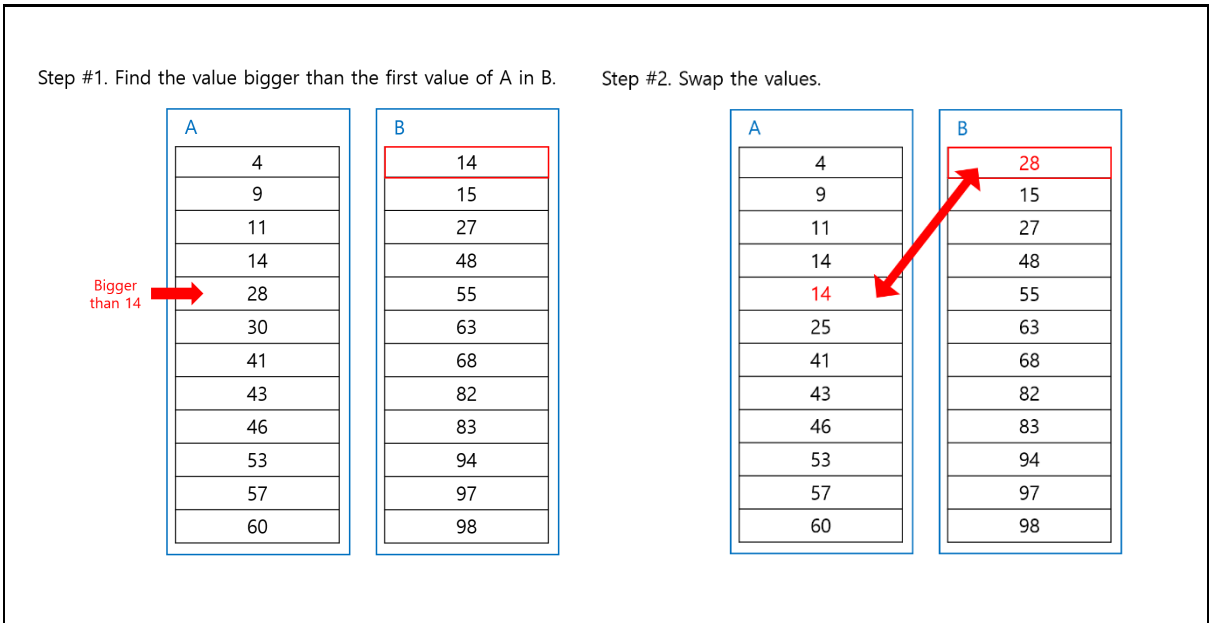


그림 2

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.



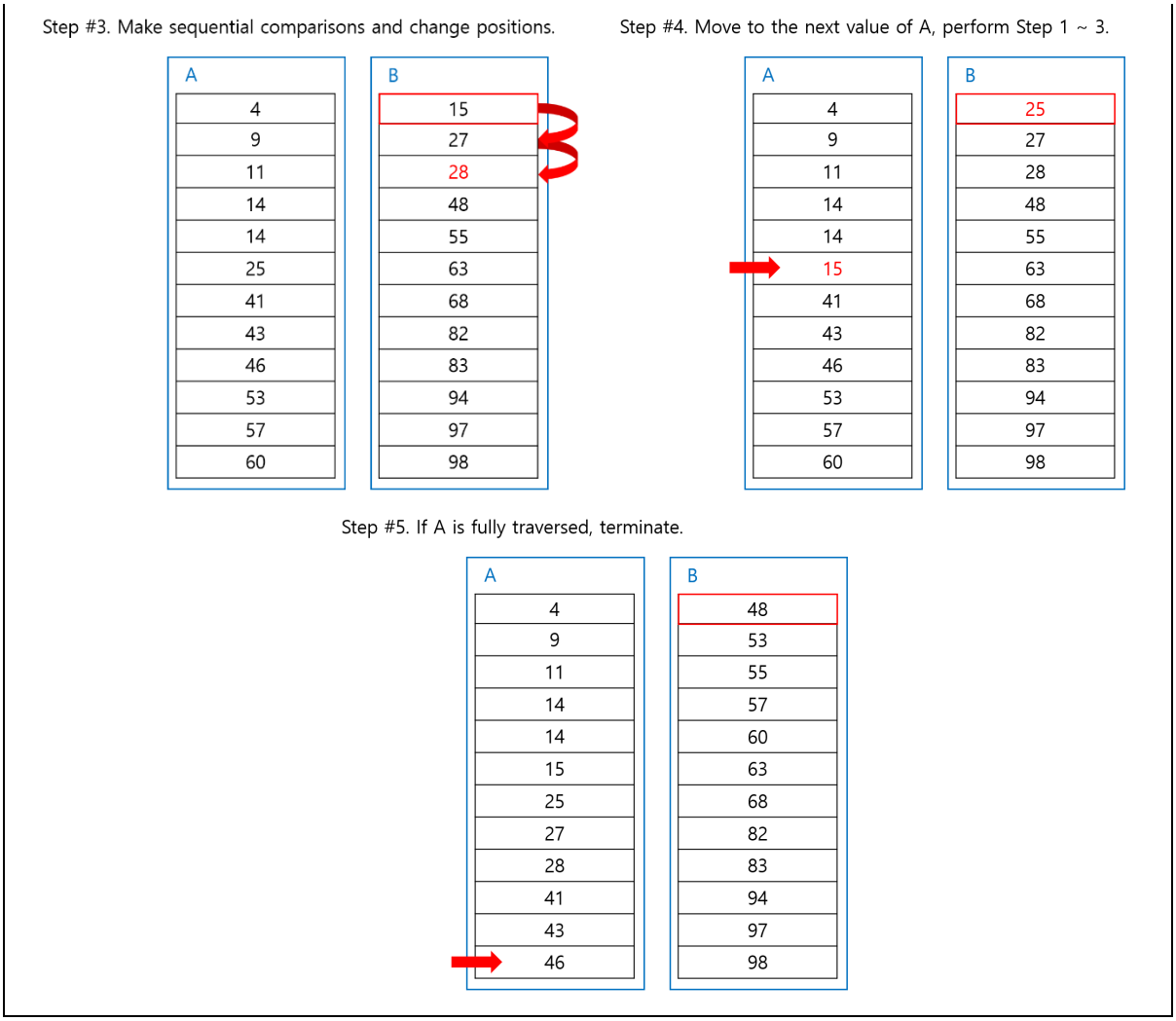


그림 3

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.

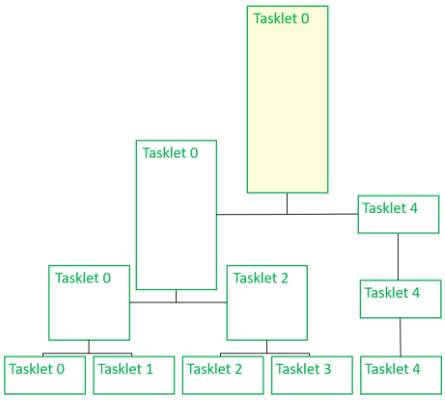


그림 4

- **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.

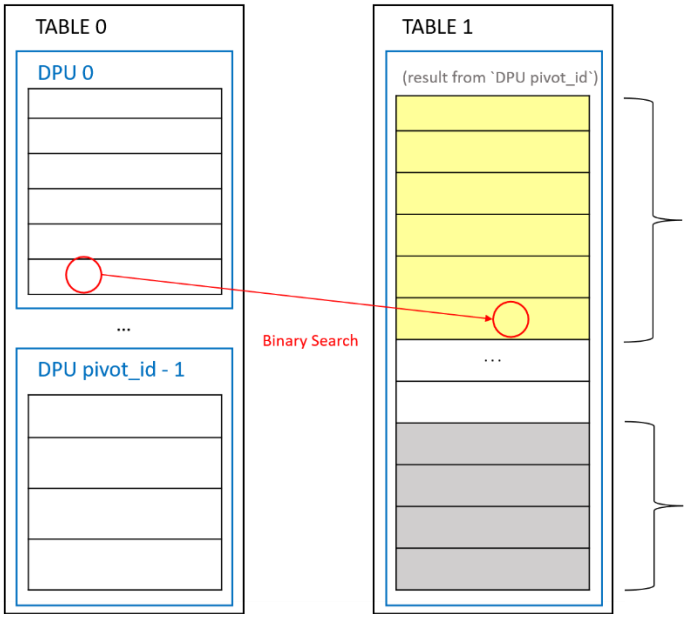


그림 5

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. main branch가 최신 버전이며, upmem-pim/upmem_env.sh을 source 후 실행 가능하다.

<알고리즘 실행>

upmem-pim/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
```

<user.h>

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

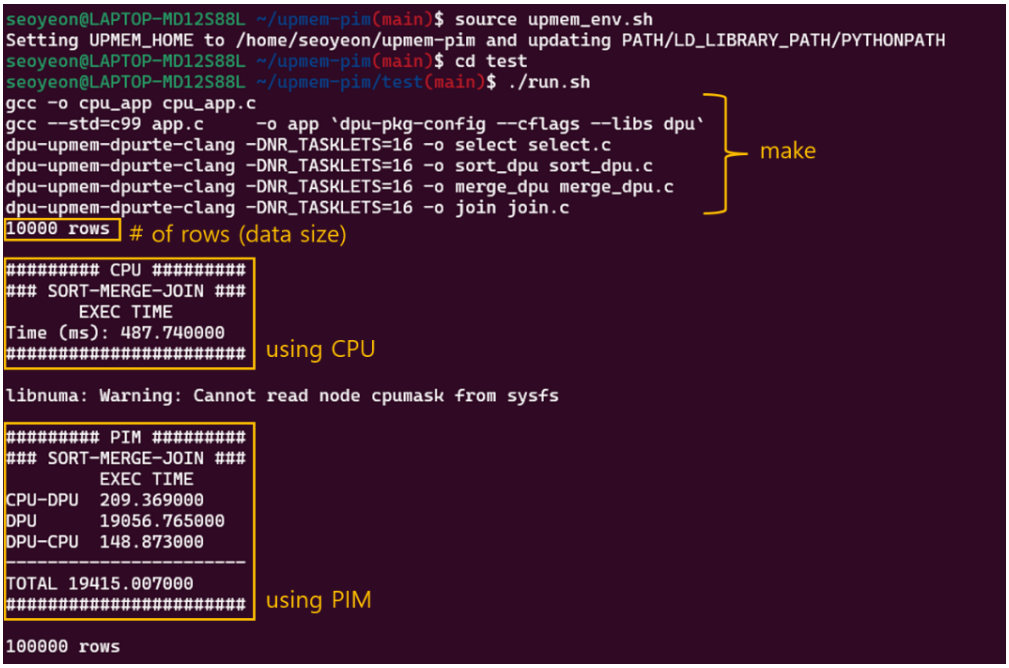
- DEBUG 모드
- 사용할 DPU, tasklet 개수 (NR_DPUS, NR_TASKLETS)
- data1.csv에서의 select column과 select 기준값 설정(SELECT_COL1, SELECT_VAL1)
- data2.csv에서의 select column과 select 기준값 설정(SELECT_COL2, SELECT_VAL2)
- data1.csv에서의 join column(JOIN_KEY1)
- data2.csv에서의 join column(JOIN_KEY2)

upmem-pim/sort-merge-join/run.py을 실행하면 input으로 넣은 데이터에 대한 결과를 확인할 수 있다. 결과는 upmem-pim/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

========================================
seoyeon@LAPTOP-MD12S88L ~/upmem-pim/sort-merge-join(main)$
```

<실험 결과 확인 – PIM 적용 여부에 따른 실행 시간 비교>

upmem-pim/test/run.sh을 실행하면 동일한 데이터에 대해 PIM을 적용하지 않았을 경우의 실행 시간과 PIM을 적용하였을 경우의 실행 시간을 확인할 수 있다. input 데이터는 upmem-pim/test/data에 있으며, generate_data.py를 통해 제작된 일정 범위 내 무작위의 숫자로 이루어진 데이터이다. 아래는 시뮬레이터에서 실행한 예시이다.



3. 실험 및 결론

- PIM 하드웨어: V1B
- OS: Ubuntu 20.04.6 LTS
- UPMEM DPU toolchain (version 2023.2.0)
- Input: 4개의 column과 각각 10만, 20만, 30만, 50만 개의 row로 이루어진 두 개의 csv 파일

<실험 1 - 영역에 따른 실행 시간>

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

	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

(시간 단위: ms)

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 DPU 간의 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.

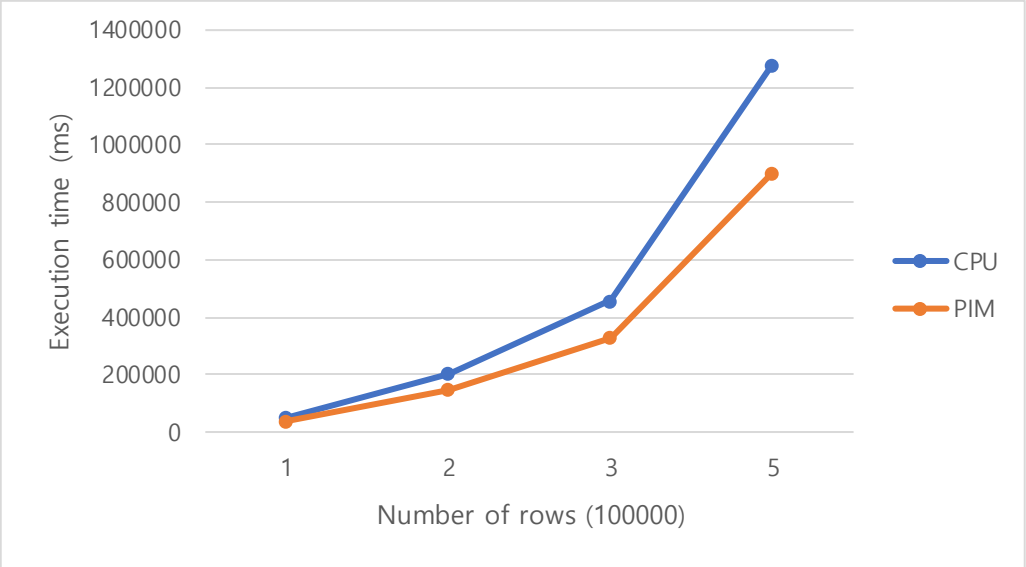
<실험 2 - PIM 적용 유무에 따른 실행 시간 변화>

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

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

(시간 단위: ms)

표 2



그래프 1

또한 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.

<실험 3 - DPU 개수에 따른 실행 시간 변화>

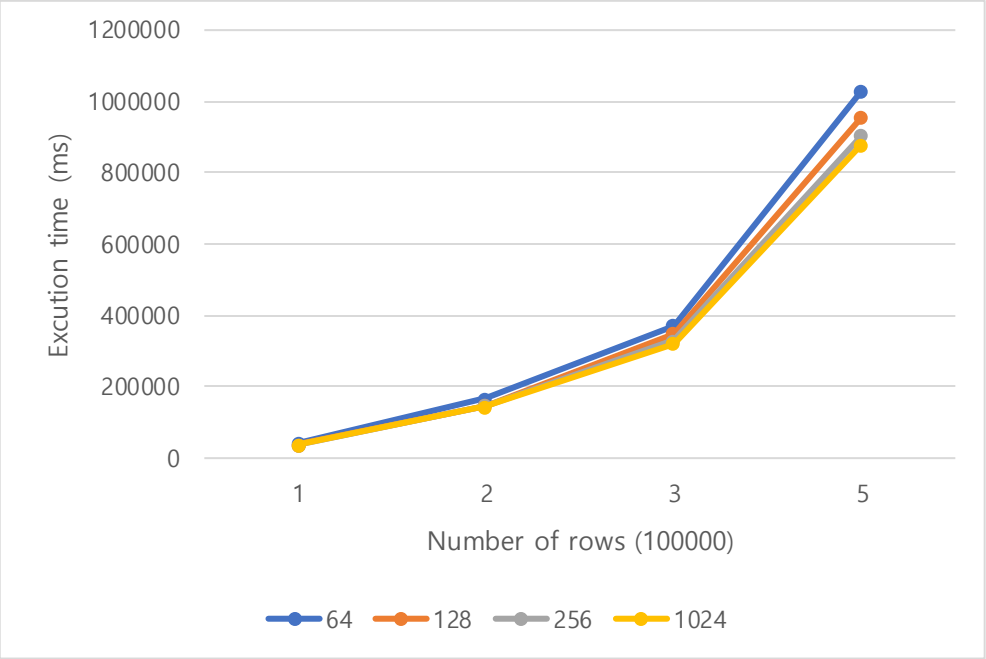
아래는 각 데이터셋에 대해 64개, 128개, 256개, 1024개 DPU, 16개 tasklet을 사용하여 실행한 결과이다.

	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

(시간 단위: ms)

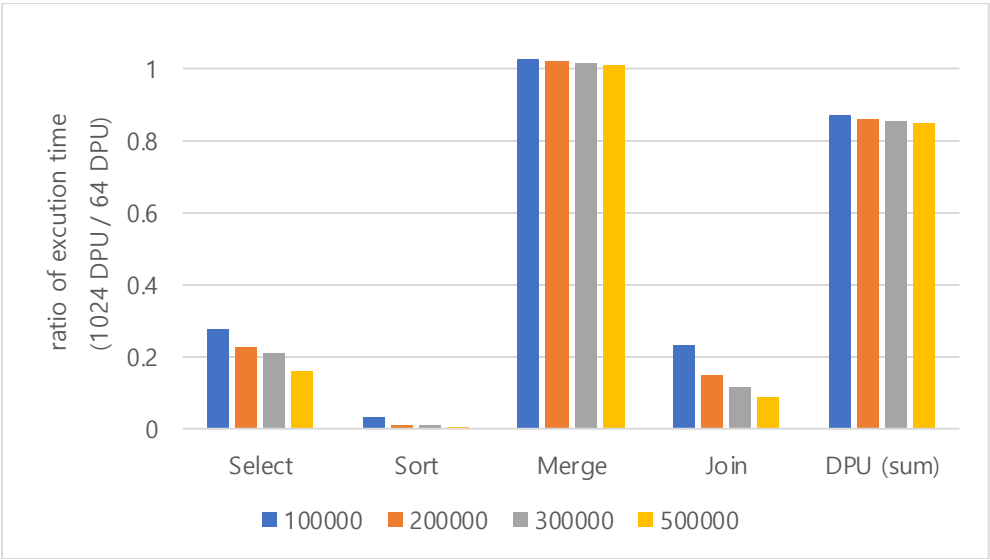
표 4

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 2'를 통해 확인할 수 있다.



그래프 2

그 이유는 merge_dpu.c(DPU에 나눠져 있는 table의 데이터를 하나로 병합하는 과정)을 '그림 4'와 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 3'은 그 값을 그래프로 표현한 것이며, 0에 가까울수록 실행 시간의 차이가 크다.



그래프 3

'그래프 3'을 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 DPU와 10~16개 tasklet을 사용하여 실행한 결과이다.

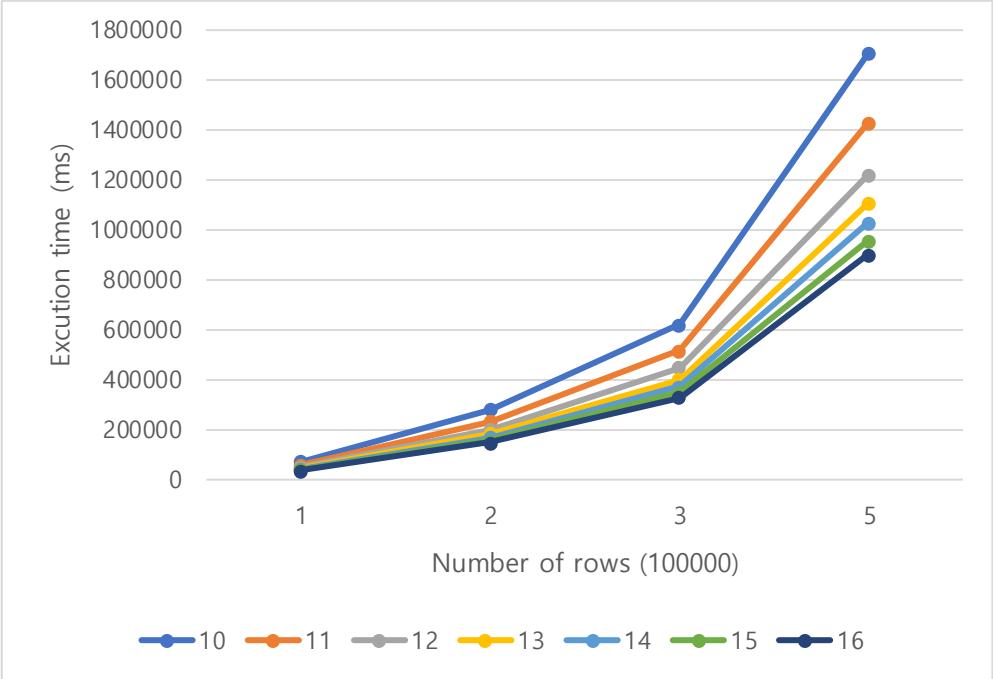
	100000	200000	300000	500000
10	70743.954	281219.737	621620.182	1708639.173
11	58696.843	230888.953	515656.332	1430548.578
12	51339.027	199869.255	450711.793	1222208.989
13	46330.609	182801.673	402281.93	1108123.268
14	42896.735	168266.786	371853.85	1030884.233
15	40456.86	156176.676	348053.827	958821.114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

(시간 단위: ms)

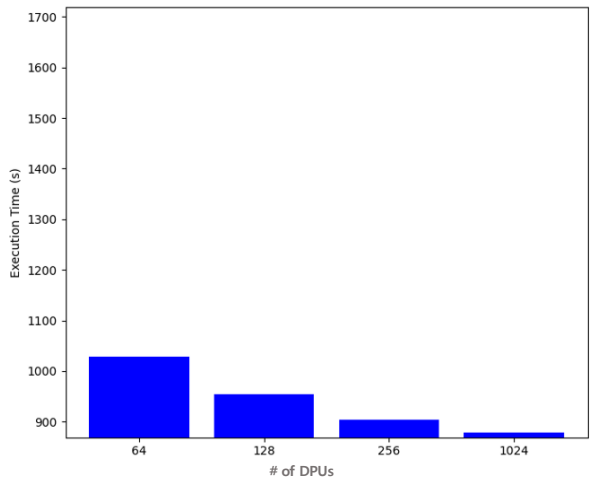
표 5

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.

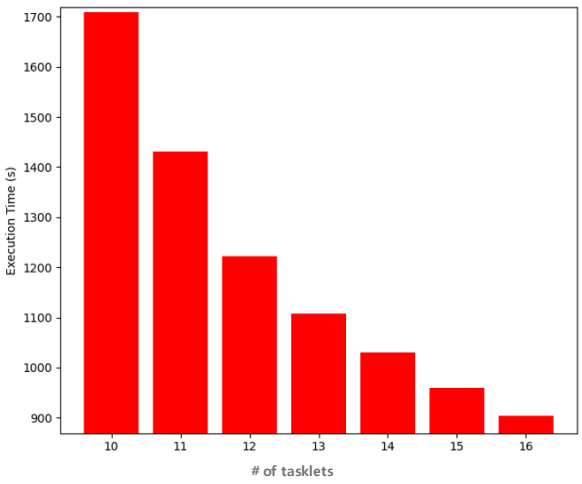
본 프로젝트에서는 사용하는 tasklet 개수가 증가할수록 실행 시간이 감소하였다. 이는 구현 상에서 사용하는 tasklet의 개수가 많아짐에 따라 처리할 데이터의 양이 줄어들기 때문이다. tasklet의 개수가 11개를 초과하면서 pipeline이 full 상태가 되어 처리 과정에서 clock을 공유하지는 못하는 것과는 별개로, 각 tasklet이 처리해야 할 데이터가 줄어듦에 따라 처리에 필요한 cycle 수 또한 줄어들어 tasklet의 종료 시점이 빨라지기 때문이라고 생각한다. 따라서 pipeline 활용 최대치를 초과하더라도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



그래프 4



그래프 5



그래프 6

위의 '그래프 5'는 DPU의 개수에 따른 실행 시간의 변화이고, '그래프 6'은 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.

4. 개선 사항

- cache size의 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

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 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.

예를 들어, 하나의 테이블만 병합하는 방법이 있다. 한 테이블만 병합하고 다른 테이블은 DPU에 분할한 채 기존 방식과 동일하게 binary search 후 join 과정을 수행한다. 이러한 방식을 적용하면, 기존의 설계는 유지할 수 있되 현재 구현(두 개의 테이블의 데이터를 모두 각각 하나로 병합하는 방식)에 비해 병합에 드는 오버헤드를 절반 정도 감소될 것이라고 예상된다.

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