[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 월별 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연 2021097356 김한결

2024.09.20.

목차

1. 9월 계획

- UPMEM-PIM을 활용한 sort-merge join 알고리즘 구현 완료
  - 메모리 관리
  - 다른 table에도 적용
  - join 수행 과정을 마친 최종 sort-merge join 알고리즘
- 성능 측정

2. 진행 사항

9월 계획을 진행 중에 있다. join 수행 과정을 구현 중이며, 그 외 계획은 완료하였다.

2.1. 구현

tasklet 별 결과를 merge하는 장소와 이에 따른 DPU 별 산출된 결과를 merge하는 방식에 대해 논의, 변경하였다. 해당 코드는 github repository에서 확인할 수 있다.

2.1.1. 정렬 횟수 최적화

기존 구현에서는 데이터를 taskelt 단위까지 분배 후 select 및 quick sort를 한 번에 진행하였다. 정렬된 데이터는 dpu 내에서 합쳐진 후 host로 전달되어 다음 과정을 위해 분배되었는데 이때 select 이후 데이터의 크기가 줄어들게 되므로 dpu 간 이동하는 데이터의 크기를 재조정해 줄 필요가 있었다. 그러나 이로 인해 정렬이 깨지게 됐고 그다시 한 번 quick sort를 해야 하는 불필요성이 발생하였다. 따라서 최적화를 위해 구조를 변경하였다. 변경한 구조는 다음과 같이 설명할 수 있다. 우선 첫 번째 과정에서는 select만 진행된다. select를 마친 데이터는 host로 와 합쳐지고 dpu 크기만큼 나누어 다시 tasklet 단위로 보내진다. 두 번째 과정에서 dpu 내에서의 sort가 진행되고 수행이 끝나면 host로 전달된다. 마지막 과정에서 dpu끼리 merge를 통해 table 2개에 대한 select와 sort가 완료된 결괏값을 받아볼 수 있다.

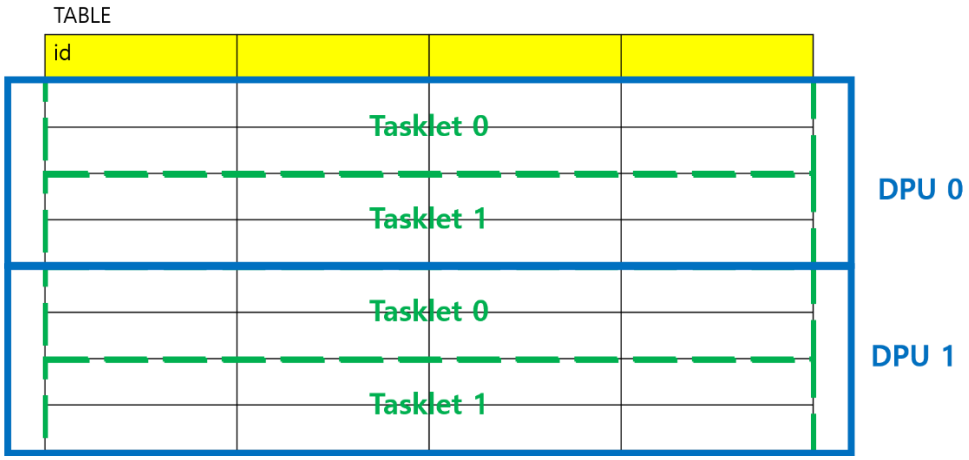
또한 select 내에서 오류가 있었기에 수정하였다. 원래 메모리의 빈 공간을 제거하기 위해 select된 데이터가 write될 위치를 찾고 동시에 write되었으나 이 경우 전 tasklet의 데이터가 덮어씌워지는 오류가 있었다. 해당 오류를 발견하고 select는 순차적으로 진행되도록 변경하였다.

2.1.2. quick sort 관련 trouble shooting(sort_dpu.c)

select된 데이터를 정렬하는 역할을 하는 sort_dpu에서 데이터의 크기가 커지게 되면 write error가 발생하였다. dpu-lldb를 사용해 dpugrind -i trace-0000-00로 오류를 추적해 본 결과 quick sort에서 재귀함수를 통해 해당 오류가 계속해서 발생하고 있음을 확인하였다. 재귀함수를 사용해 quick sort를 구현하게 되면 함수가 계속해서 stack 영역의 메모리, 즉 wram의 메모리를 차지하기에 오버플로우가 발생하였다고 추정할 수 있었다. 따라서 재귀함수를 사용하지 않고 stack을 통해 quick sort를 구현하게 되었다.

여기서 stack size를 처음에 2n으로 설정하였기에 당연하게도 wram write error가 발생하였다. n으로 설정한 이유는 quick sort의 경우 평균적으로 logn의 깊이를 가지고, 그 때 서브 배열의 수는 n개를 가지기에 stack에 시작과 끝 인덱스를 저장한다 생각하면 pop을 고려하지 않았을 때 최대 2n의 크기를 가져야 한다고 생각하였다. 하지만 실제로 이미 처리된 배열에 대해서는 해당 인덱스가 stack에서 사라지기에 logn으로 보는 것이 더 합당하며 고정 사이즈 100으로 설정하였다.

2.1.3. DPU 별 결과 merge 구현(merge_dpu.c)



기존의 방식: DPU를 단위로 merge (5월 보고서에 첨부된 자료)

2.1.1.의 내용을 바탕으로, 마지막 DPU를 제외한 각 DPU는 동일한 개수의 row를 가지고 있다. merge_dpu.c에서는 각 DPU의 결과를 하나로 합쳐 하나의 table에서 select와 sort를 마친 상태로 만든다. DPU-i와 DPU-(i+1)의 결과를 sort-merge하며 DPU0에 최종 결과가 저장된다.

merge_dpu,c에서 이루어지는 과정은 다음과 같다.

a) binary search

DPU-i의 결과에 대해 각 tasklet이 할당 받은 영역 중 마지막 row를 기준으로, DPU-(i+1)에서 join column의 값이 매치되거나 현재 타겟 값보다 작은 수 중 가장 큰 수(lower bound)를 반환하는 binary search를 수행한다. 이를 통해 해당 tasklet에서 할당 받은 영역이 DPU-(i+1) 내 처리해야할 영역을 알 수 있다.

b) sort

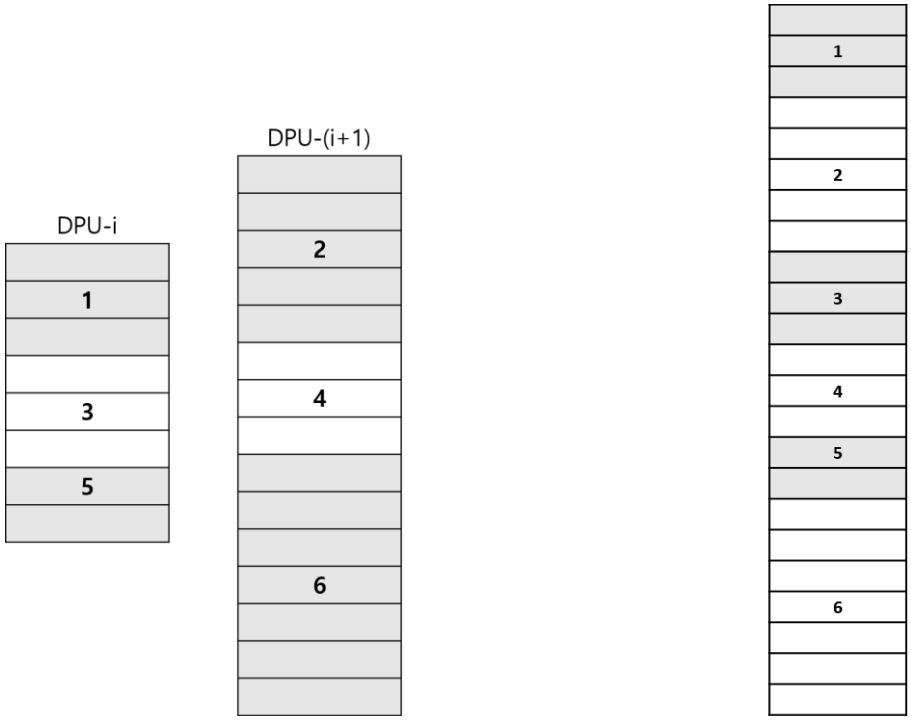
각 tasklet이 a의 결과를 바탕으로 정렬을 수행한다. 여기서 정렬 방식은 sort_dpu.c와 같은 방식을 사용했으며, 추가 공간 없이 in-place로 수행할 수 있다. 정렬 방식은 다음과 같다. 현재 tasklet이 처리해야할 DPU-i의 영역을 A, DPU-(i+1)의 영역을 B라고 하자.

1) A의 현재 row의 join column 값(a)을 B의 첫 번째 join column 값(b)과 비교
   A. 만약 a가 b보다 작거나 같다면 A에서 다음 row로 이동, 1을 수행
   B. 만약 a가 b보다 크다면
      i. B의 첫 번째와 자리를 변경
      ii. B의 정렬이 깨졌으므로 순차적으로 비교하며 알맞은 위치가 나올 때까지 row를 앞으로 하나씩 이동하여 자리 변경
      iii. A의 다음 row로 이동, 1을 수행
2) A의 row 개수만큼 수행했다면 종료

c) re-sort

각 tasklet이 담당한 영역에 대해 정렬을 마쳤으므로 이를 다시 재정렬한다. b번까지 마 친 결과는 왼쪽 그림과 같다. 이를 오른쪽 그림과 같은 결과로 다시 정렬한다. (host에서 공간을 할당할 때 이를 고려하여 할당하였다.)

먼저 A를 재정렬한다. 동시에 작성할 경우 덮어씌워질 수 있으므로 마지막 tasklet부터 역순으로 작성 후, 남은 공간에 대해 모든 tasklet이 B를 동시에 작성한다.기존의 방식은 DPU i와 DPU i+1의 결과를 DPU (i/2)에서 sort 후 merge하는 과정을 하나의 DPU에서 처리하여 하나의 sort된 table을 결과로 낼 때까지 반복하는 방식이었다.



(좌) b까지 수행한 결과 / (우) 전체 수행 결과

구현 과정에서 메모리 영역과 관련한 trouble shooting이 있었다. 'dpu_types.h'에 정의되어 있는 DPU_MRAM_HEAP_POINTER_NAME을 사용하여 데이터를 전송했는데, DPU_MRAM_HEAP_POINTER_NAME은 dpu_push_xfer를 수행할 때마다 변하는 값이라고 생각하여 각종 문제가 발생했었다. 예를 들어, 아래와 같은 코드가 있을 때 두 번째 dpu_push_xfer에서의 DPU_MRAM_HEAP_POINTER_NAME과 첫 번째 dpu_push_xfer에서의 DPU_MRAM_HEAP_POINTER_NAME은 동일한 위치를 가리킨다.

```
DPU_ASSERT(dpu_prepare_xfer(dpu2, dpu_result[pair_index].arr));
DPU_ASSERT(dpu_push_xfer(set2, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0, first_size, DPU_XFER_DEFAULT));
DPU_ASSERT(dpu_prepare_xfer(dpu2, dpu_result[pair_index + 1].arr));
DPU_ASSERT(dpu_push_xfer(set2, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, first_size + second_size, second_size, DPU_XFER_DEFAULT));
```

2.1.4. 다른 table에 적용

지금까지의 구현을 한 table이 아닌 두 개의 table에 대해 수행하도록 수정하였다.

2.2. join 과정의 구체적 설계

host에서 각 DPU에서 담당할 table1의 영역에 대해 table2의 어느 영역까지 필요한지 binary search를 수행한다. 결과를 바탕으로 필요한 영역만큼만 자른 table2의 일부와 DPU 개수와 비례하여 담당할 table1의 일부를 전송한다.

DPU 내에서는 다시 tasklet 별로 영역을 담당하고, 담당한 영역에 대해 merge_dpu.c에서와 같이 binary search를 통해 필요한 table2 영역을 찾아낸다. 이후 linear하게 각 row를 비교하며 매치되는 row를 찾는다. 매치된 row는 join하여 WRAM에 쓰고 host로 전송한다. (공간이 부족할 경우 MRAM에서 관리하도록 수정) 이후 DPU 순서대로, tasklet 순서대로 결과를 받으면 sort merge join을 수행한 최종 결과를 얻을 수 있다.

3. 10월 계획

- UPMEM-PIM을 활용한 sort merge join 알고리즘 구현 완료
  - join 수행 과정을 마친 최종 sort merge join 알고리즘
- 성능 측정
  - PIM이 적용되지 않은 sort merge join 알고리즘 구현
  - 타이머를 통한 성능 비교
- 프로젝트 마무리