[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 10월 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연
2021097356 김한결

2024.10.20.

목차

1. 10월 계획

- UPMEM-PIM을 활용한 sort-merge join 알고리즘 구현 완료
  - ■ 다른 table에도 적용
  - ■ join 수행 과정을 마친 최종 sort-merge join 알고리즘
- 성능 측정
- 프로젝트 마무리

2. 10월 진행 사항

10월 계획을 진행하였다. 설계에 따른 구현을 완료하였고, 일부 과정은 설계와 다르게 수정하였다. 이로써 PIM을 활용한 sort merge join 알고리즘 구현을 완료하였다.

본 보고서에서 첫 번째 table은 table 0, 두 번째 table은 table 1이라고 한다.

2.1. 구현

새롭게 구현한 부분으로는 아래와 같다. join 과정을 설계에 맞게 구현하여 전체 알고리즘의 구현을 완료하였다. 성능 측정을 위해 CPU에서의 sort merge join 실행 파일을 생성하였고, 성능 측정을 포함한 전체 프로세스 실행을 위한 쉘 스크립트 파일을 확인할 수 있다.

2.1.1. join

PIM을 활용한 sort merge join 알고리즘의 마지막 단계인 join 과정을 구현했다. 기존 설계에서는 두 개의 테이블에 대해 join될 row를 먼저 찾아 기록하고, 이후 전체 tasklet이 동시에 MRAM에 join 결과를 쓰도록 설계했다. 동시성을 활용한 시간 단축과 전체 배열을 탐색하는 것을 최소화하기 위함이다. 그러나 실제 구현 과정에서 WRAM의 크기 한계로 설계를 수정하였다. 데이터셋의 크기가 커질수록 join될 row의 인덱스를 저장할 배열 또한 크게 동적할당 해야 하므로 WRAM의 크기를 넘어서게 된다.

따라서 설계를 다음과 같이 수정하였다. 먼저 각 tasklet이 담당한 영역에서 join될 row의 개수를 세고, 모든 tasklet이 이를 마치면 그 숫자에 맞춰 MRAM에 기록할 위치를 찾는다. 이후 모든 tasklet이 동시에 다시 테이블의 담당한 영역에서 join되는 결과를 MRAM에 write한다.

2.1.1. CPU ver

cpu_app.c에서 해당 파일을 확인할 수 있다. DPU에서 실행하는 것과 마찬가지로 csv

파일을 읽고 select, sort, join 순서를 거쳐 ouput으로 csv 파일을 도출한다. sort는 DPU에서 사용하는 정렬 방식과 동일한 방법을 사용한다. 시간 측정은 select, sort, join 부분에서만 일어난다.

2.1.2. 데이터 생성 및 실행

데이터를 생성한 파일은 generate_data.py에서 확인할 수 있다. join key는 파일 둘 다 0이라고 가정하였고 해당 key는 unique하다. 또한 column 개수는 4개로 동일하다. 프로그램을 통해 row 개수가 각각 10000개, 100000개, 200000개, 300000개, 500000개, 700000개, 1000000개인 csv 파일을 2개씩 생성하였다. 실행 결과에 따라 데이터는 추후 변경될 수 있다.

코드 상의 구현과 알고리즘에서 데이터의 row 개수를 제한할 필요는 없다. 실험을 위해 테이블의 row 개수를 고정하고 실행 파일(run.sh)에서 row 개수를 출력하도록 작성했지만, 실제 실행은 각 테이블의 row 개수를 제한하지 않고 동작하며, 입력 시 row 개수와 col 개수를 읽어낸다.

여러 데이터를 입력으로 사용하기 위해 파일 이름을 main에서 인자로 받도록 수정하였으며 run.sh에서 파일 순서를 지정하였다. run.sh를 실행하게 되면 데이터에 따른 sort merge join 소요 시간을 확인할 수 있다. CPU 버전의 sort merge join 알고리즘을 실행한 후, PIM을 활용한 sort merge join 알고리즘에서는 CPU에서 DPU로의 이동 시간, DPU에서의 실행, DPU에서 CPU로의 이동 시간, 3개의 부분으로 나누어 측정하며 총합 시간을 확인할 것이다.

2.2. 개선 및 trouble shooting

실제 PIM 서버를 사용하며 다양한 에러와 개선점을 발견하였다. 입력 데이터의 크기에 따라 발생하는 문제와 소요 시간을 줄이기 위해 아래와 같은 내용과 각종 에러에 대한 trouble shooting을 진행했다.

2.2.1. quick sort 변경

전에 quick sort를 재귀로 구현하였으나 함수의 재호출로 인해 WRAM이 overflow되는 문제를 인식하고 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 돌아가는 걸 생각하면 스택의 정의만으로 WRAM의 범위를 넘어갈 수 있다. 그렇다고 STACK_SIZE가 너무 작아지면 quick sort를 돌리는 게 불가능하므로 다른 방법으로 quick sort를 구현하거나 정렬 방식을 변경할 필요성을 느끼게 되었다.

새로운 방식의 quick sort는 기존의 재귀 호출 또는 스택을 사용한 quick sort에 비하면

비효율적이나, 추가적인 공간의 사용이 없다는 점에서 효율적이다. 0부터 n까지의 인덱스를 가진 데이터가 있다고 가정할 때, 중간의 n/2 인덱스에 해당하는 값을 pivot으로 삼아 기존의 quick sort와 같이 swap을 수행하여 pivot을 중심으로 왼쪽에는 pivot보다 작은 값이, 오른쪽에는 pivot보다 큰 값이 위치하도록 한다. 그 다음에는 0부터 n/2에 대해, n/2 + 1부터 n에 대해 같은 과정을 수행하고, 이후에는 4번, 8번, 16번, ... 이를 반복하며 하나의 데이터에 대해 수행하게 될 때까지 반복한다. 기존의 quick sort와 유사하며 추가적인 공간 사용이 없으나, 상대적으로 더 오랜 시간이 소요되며 전체 데이터에 대해 log N번 탐색을 반복하게 된다.

따라서 quick sort를 제외한 bubble sort, selection sort, insertion sort와 같은 다른 in-place 알고리즘을 적용해 보았다. 시간 복잡도에서는 우위를 가지지 못하지만 해당 알고리즘은 10만 개를 기준으로도 문제없이 동작하는 것을 확인하였다. 현재 최적화보다는 정상 동작과 성능 비교를 우선시하여 CPU와 DPU에서 모두 insertion sort를 쓰는 것으로 결정하였다.

2.2.2. select 최적화

select에서 걸리는 시간을 줄이기 위해 handshake_sync를 사용하여 cache size 단위로 처리하도록 변경하였다. 이는 prim benchmark에서의 방법과 거의 유사하다. 백만 개의 row 개수를 가지는 파일에 대하여 73386.717000ms에서 14154.213000ms로 소요 시간이 유의미하게 감소하였음을 확인하였다.

2.2.4. merge 최적화 시도

merge 부분에서 걸리는 시간을 줄이기 위해 버퍼를 추가해 보았다. 이 merge는 정렬 결과를 유지한체 합치는 부분을 의미하며 sort_dpu.c와 merge_dpu.c에 구현되어 있다. 원래는 값을 swap한 후 sequential하게 내려가며 들어갈 자리를 찾았지만 이 과정에서 버퍼를 추가하여 row를 n개씩 확인하게 하였다. 버퍼의 마지막 줄의 key 값이 swap 된 값보다 크면 해당 버퍼 안에서 sequential하게 탐색이 이루어지며 아니라면 다음 버퍼로 넘어가게 된다. 구현을 마무리하였으나 기존의 것보다 성능상으로 유의미한 차이를 가지지 못했으며 오히려 소요 시간이 더 증가하였기에 보류하였다.

하지만 환경이 한계로 적은 데이터만을 실행시켜보았기에 그랬을 가능성이 존재하며 매우 큰 데이터셋에 대해서는 효율적일 수도 있다. 실행 결과에 따라 추후 적용 여부를 논의할 예정이다.

2.2.5. select에서의 동기화 문제

select는 cache size만큼 for문을 통해 실행된다. 따라서 tasklet 개수가 3개라면 0 -> 1 -> 2 -> 0 -> 1 -> 2...의 순으로 write가 이루어지며, tasklet 1에서 모든 데이터가 처리된

경우, tasklet 2가 먼저 for문 밖으로 나가게 된다. 이렇게 되면 for문 안의 barrier_wait로 인해 tasklet별로 호출하는 barrier_wait의 개수가 달라지게 된다. 따라서 일부 tasklet이 return 0에 도달하지 못하는 문제가 발생하였다. 이를 해결하기 위해 tasklet이 실행해야 할 cycle 수를 미리 구하고 모든 tasklet은 해당 cycle만큼 반복문을 돌도록 하였다. 만약 처리할 게 없는 tasklet(위의 경우에서는 tasklet 2)은 cycle을 도는 동안 별다른 일을 하지 않고 barrier_wait만을 호출함으로써 갇히는 상황이 발생하지 않도록 하였다.

2.2.6. 할당 받은 데이터가 없는 경우에 대한 예외 처리

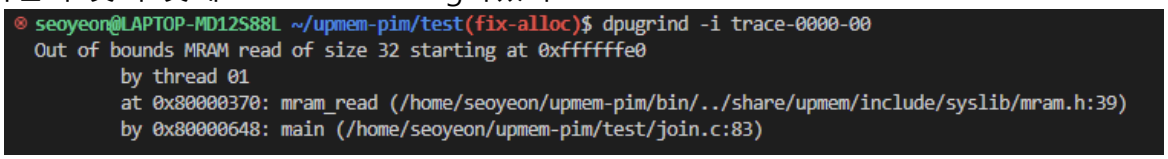
데이터의 row 개수가 DPU의 개수보다 적은 경우, 그리고 DPU에 할당된 row 개수가 tasklet의 개수보다 적은 경우에 대해 오류가 발생하였다. 전자의 상황이 발생한다면, 사용하는 DPU 개수를 2개로 변경하여 동작하도록 하였다. 이는 NR_DPUS가 64고 하나의 table에 32개를 할당하려고 할 때, 31개의 row를 DPU에 하나씩 할당하고 마지막 DPU를 비우는 것보다는 하나의 DPU가 처리하도록 하는 것이 더 효율적이리라는 판단에 근거한다. 후자의 상황에서는 대응 방법이 알고리즘마다 조금씩 상이하다. select.c에서는 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.

2.2.7. 'stack smashing detected' 에러

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 없었기에 단순히 `input_args`과 `dpu_result`의 배열의 크기를 늘려주는 것만으로도 해결할 수 있었다.

2.2.8. join.c 내 'DPU is in fault'

join.c에서 동일한 데이터를 join할 때 DPU is in fault 에러가 발생하였는데 dpu-lldb를 사용하여 원인을 파악할 수 있었다. while문에 진입하기 전 필요한 정보에 대해 선행적으로 mram_read가 일어나지 않았기에 첫 번째 수행에서 잘못된 값을 참조하고 있어 발생한 문제였다. 첫 번째 row에서 join_key가 동일하지 않다면 발생하지 않는 에러이기에 식별이 늦어 늦게 trouble shooting하였다.



3. 추후 계획

PIM 서버를 사용하지 않고 시뮬레이터를 사용해 구현 및 디버깅, 성능 측정을 진행하고 있다. 실제 서버와 결과가 동일하지 않고, 개인 PC의 성능 한계로 인해 규모가 큰 데이터셋은 성능 평가에 활용하지 못하고 있다. 구현을 완료하여 최종 보고를 앞둔 상태이다. 최종 보고서에서는 시뮬레이터의 결과와 연구실의 서버를 사용한 실험 결과를 첨부하여 작성할 예정이다.