[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 월별 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연 2021097356 김한결

2024.07.18.

목차

1. 7월 계획

2. 진행 사항

2.1. 구현 진행 사항

2.2. simulator 환경에 관한 논의

2.3. 성능 측정 방법

3. 8월 계획

1. 7월 계획

- UPMEM-PIM을 활용한 sort-merge join 알고리즘 구현 시작

2. 진행 사항

7월에는 계획대로 구현을 시작하였다.

2.1. 구현 진행 사항

https://github.com/5eoyeon/upmem-pim

구현이 진행 중인 github repository이다. 7월부터 구현을 시작해 dpu.h를 참고하여 다음 내용을 구현했다.

- dpu 할당 및 tasklet 생성
- tasklet마다 개별 배열 할당
- tasklet의 연산
- 위의 내용에 lock을 추가

tasklet마다 개별 배열 할당의 경우, 현재 코드 상에서는 100개 int로 구성된 배열을 테스트 케이스로 사용했으며 이미 코드 상에 명시되어 있다. 이후 sort-merge join 알고리즘 구현에서는 size를 입력 또는 data의 전체 크기를 먼저 파악하는 것으로 전체 데이터 개수를 파악할 것이다. 전체 data 개수를 할당할 DPU 개수로 나누고 할당한다.

```
  int chunk_size = SIZE / NR_DPUS;
  DPU_FOREACH(set1, dpu1, dpu_id) {
    int offset = dpu_id * chunk_size;

    DPU_ASSERT(dpu_prepare_xfer(dpu1, test_array + offset));
    DPU_ASSERT(dpu_push_xfer(dpu1, DPU_XFER_TO_DPU, "test_array", 0, chunk_size *
sizeof(int), DPU_XFER_DEFAULT));
  }
```

각 DPU별 할당된 배열은 다시 각 tasklet에 나누어서 할당된다. tasklet의 sysname을 통해 DPU가 할당 받은 배열 중 어느 부분을 연산에 사용할 것인지 결정하였다.

```
    int start = tasklet_id * CHUNK_SIZE;
    int end = start + CHUNK_SIZE;
```

이후 구현에서는 CHUNK_SIZE 또한 host application에서 계산하여 전달할 것이다. tasklet 별 연산을 마치고 연산이 끝난 데이터를 다시 받는다.

```
  DPU_FOREACH(set1, dpu1, dpu_id) {
    int offset = dpu_id * chunk_size;
```

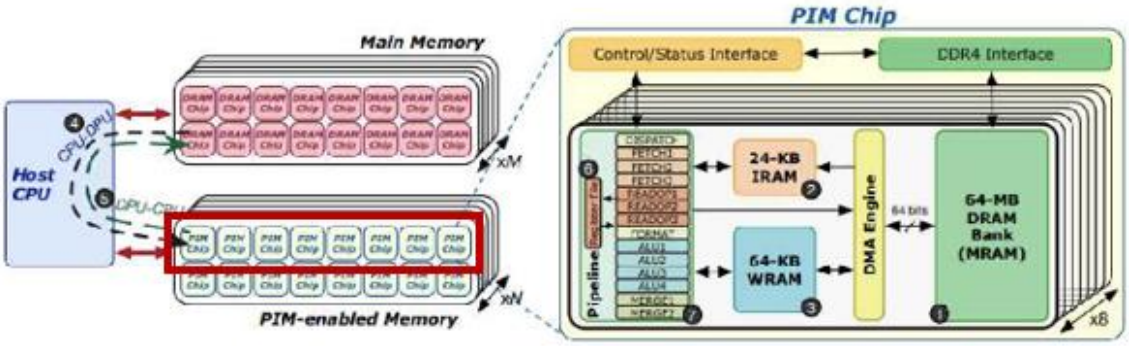
```
    DPU_ASSERT(dpu_prepare_xfer(dpu1, test_array + offset));
    DPU_ASSERT(dpu_push_xfer(dpu1, DPU_XFER_FROM_DPU, "test_array", 0, chunk_size *
sizeof(int), DPU_XFER_DEFAULT));
  }
```

추가로 lock을 적용하여 결과를 print하는 부분을 tasklet별로 확인할 수 있게 하였다. (복잡한 내용은 아니지만 lock을 사용해보는 것에 의의를 두었다.) 또한 upmem-pim sdk에서 제공하는 lock의 활용과 케이스를 공부하였다. sort-merge join 알고리즘의 구현에서는 mutex, barrier, handshake를 사용해 구현할 계획이다.

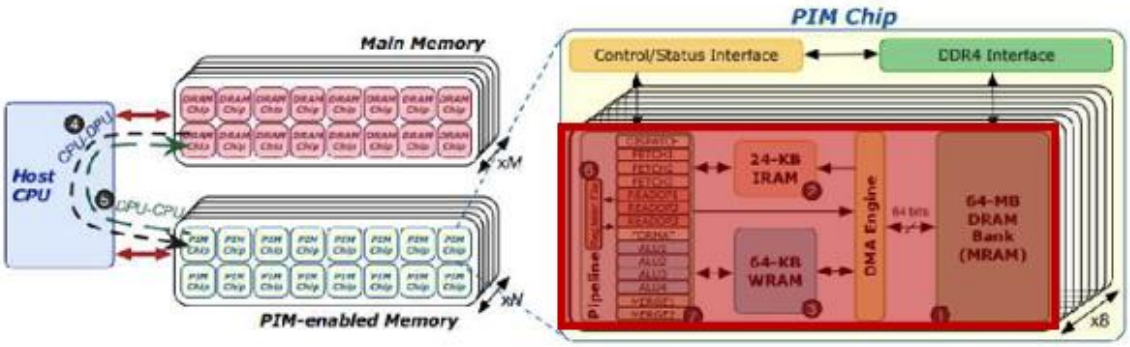
현재는 divide-and-conquer를 적용한 tasklet 별 정렬을 구현하고 있다.

2.2. simulator 환경에 관한 논의

현재 서버가 없는 관계로 UPMEM simulator를 이용하고 있으나 여기서 오는 의문이 존재하였기에 관련 논의를 진행하였다.



1번 그림: 8개의 PIM Chip으로 구성된 하나의 rank만 사용 가능



2번 그림: PIM Chip 내에는 하나의 DPU만 존재

시뮬레이터에서는 총 64개의 DPU를 할당하여 사용할 수 있다. 또한 `dpu_alloc`함수를 이용해 파라미터 `nr_dpus`만큼의 dpu를 하나의 set으로 전달할 수 있다. 의문점이 생기는 부분은, rank와 chip간의 관계이다.

set에는 특정 개수의 rank가 할당된다. 이때 rank는 동시에 접근할 수 있는 dpu의 집합으로 이해하였다. PIM Chip에는 8개의 dpu가 포함되어 있는데, 이 경우 64개의 dpu를 사용할 수 있는 simulator는 1번 그림과 같은 상황인지, 2번 그림과 같은 상황인지, 혹은 그 외의 상황인지 모호하

다.

1번 그림의 경우, 8개의 dpu가 포함된 PIM Chip 8개로 구성된 랭크 하나가 있는 모듈 하나만을 사용할 수 있다. 2번 그림의 경우, PIM Chip 내부를 제외한 나머지 물리적인 구성은 같으나 PIM Chip 내부에는 하나의 dpu만이 존재한다. simulator에서는 어느 경우에 포함되는지 파악하지 못했다.

2.3. 성능 측정 방법

PrIM(https://github.com/CMU-SAFARI/prim-benchmarks) 구현을 살펴보며 성능 측정 방법에 대해 논의했다.

timer를 이용해 UPMEM-PIM을 적용했을 때의 시간과 적용하지 않은 경우의 시간을 측정하여 성능을 비교한다. 중점적으로 이야기를 나눈 부분은 다음과 같다.

- timer 사용 시점
- UPMEM-PIM을 적용하지 않은 CPU 환경에서의 실행과 UPMEM-PIM을 적용한 환경에서의 실행 비교

UPMEM-PIM을 적용한 경우, CPU에서의 실행, CPU-DPU 이동 실행, DPU에서의 실행, DPU-CPU 이동 실행을 구분하여 timer를 start, stop한다. UPMEM-PIM을 적용하지 않은 CPU 환경에서는 직접적인 연산 실행 부분을 timer로 측정한다.

또한, 이를 객관적으로 비교하기 위해서는 동일한 실행 과정을 거쳐야 한다. PrIM의 Vector Addition 구현을 보면, vector addition을 직접적으로 수행하는 부분은 동일한 과정이다. 우리가 구현하는 sort-merge join 또한 연산 수행 함수의 내용은 동일할 것이다.

그러나 CPU 환경과 UPMEM-PIM 환경을 어느 정도로 동일하게 만들어야 하는지에 대해서는 해결되지 않았다. UPMEM-PIM에서의 병렬 처리가 CPU 환경에서도 동일하게 적용되어야 하는가 또한 하나의 기준이며, 동일해야 한다면 (사용한 DPU 개수)*(각 DPU의 tasklet 개수)만큼의 CPU 환경에서의 threads 생성과 이를 뒷받침할 자원 등이 우려된다.

3. 8월 계획

- UPMEM-PIM을 활용한 sort-merge join 알고리즘 구현 완료