[졸업 프로젝트]

Processing-in-Memory 활용 기술 개발

- 월별 보고서 -

한양대학교 컴퓨터소프트웨어학부
2021019961 장서연 2021097356 김한결

2024.04.19.

목차

1. 4월 계획

- UPMEM-PIM SDK 환경 세팅
- UPMEM-PIM 프로그래밍 방법 숙지

2. 진행 사항

지난 3월에는 제안서를 작성 및 제출하였으며, 졸업프로젝트 진행을 위한 기본 지식을 숙지하였다.

이후 4월에는 프로그래밍을 시작하기 위해서 UPMEM-PIM SDK simulator를 사용할 수 있는 환경 설정을 완료했다. OS는 Ubuntu 20.04.6 LTS로 통일하며, UPMEM DPU toolchain version 2023.2.0으로 설정했다.

UPMEM-PIM 프로그래밍 방법을 숙지하기 위해 sdk documentation을 참고하여 공부했다. 이 과정에서 제안서 작성을 위해 공부했던 이론들을 실제 실습으로 확인할 수 있었다. 공부한 주제는 다음과 같다.

- Tasklet management & Synchronization
- Memory management
- Exceptions
- Controlling the execution of DPUs from host applications
- Communication with host application
- Advanced Features of the Host API
- Logging

DPU에서 돌아가는 프로그램을 어떻게 만드는지, host application과 어떻게 상호작용하는지 파악하여 DPU 프로그램에 쓰이는 runtime library와 host application의 시작 및 DPU의 실행을 제어하는 Host API 사용을 익히는 것을 목적으로 한다. 또한 코드로 직접 실행해보며 활용하였다.

2.1 Tasklet management & Synchronization

Primitive thread, 즉 tasklet을 통한 추상화를 제공한다. Tasklet들은 같은 메모리 공간을 공유하고 있으며 여러 synchronization primitives와 다양한 기법을 통해 tasklet을 관리하고 메모리 synchronization을 수행한다.

Mutex는 사용하여 tasklets 사이의 critical sections를 정의하기 위한 가장 간단하고 빠른 방법이다. 이 장에서 mutex 사용의 새로운 두 가지 방법을 알 수 있었는데, 첫 번째는 virtual mutex이다. mutex를 통해 소프트웨어적인 abstraction으로 virtual mutex를 구현 가

능하며, virtual mutex는 WRAM에 있는 state bit에 대한 접근을, 하드웨어 mutex는 이 state bit들에 대한 접근을 막기 위해 사용한다. acquire, release하는 cost는 하드웨어 mutex보다 높지만 critical section 안에서 false conflicts를 피할 수 있다. 두 번째 방법은 pool of mutexes이다. 여러 mutex가 각각 일정 규칙에 따라 여러 객체를 보호한다. 이를 virtual mutex와 비교했을 때, 더 많은 하드웨어 mutexes를 필요로 하고 critical section에서 false conflict에 대해 자유롭지는 않다는 단점이 있다. 그러나 locking과 unlocking의 속도가 상대적으로 빠르다. 따라서 critical section의 길이와 conflict 확률에 따라 성능이 다르다.

이 외에도 semaphore, barrier, handshake를 통해 synchronization을 이룰 수 있다.

2.2 Memory Management

WRAM은 프로그램 실행 메모리로 stack, global variable 등이 위치한다. MRAM은 외부 주변장치로 간주하고, runtime library를 통해 WRAM의 동적 관리와 MRAM, WRAM 사이의 transaction을 관리할 수 있는 도구를 사용할 수 있다.

tasklet은 자체적인 목적을 위해 메모리에 버퍼를 가져오거나 공동 작업을 위해 미리 예약된 일부 공유 메모리에 액세스가 가능한데, WRAM에 메모리 할당을 위해 incremental allocator, fixed-size block allocator, buddy allocator가 이미 구현되어 있다. 각각 기능이 다르므로 구현에 맞게 사용하면 된다.

UPMEM DPU에서 정의한 alignment와 size 제약 조건을 고려해 WRAM과 MRAM 사이의 transaction을 처리해야 하는데, 이를 단순화한 것들이 이미 구현되어 있다. mram 함수를 통해 MRAM 자원들에 대해 낮은 수준에서 접근할 수 있고(MRAM에 직접 접근) sequential readers를 통해 MRAM을 WRAM에 매핑할 수 있다.

2.3 Exceptions

DPU 프로그램은 3가지 이유로 멈출 수 있으며, dpu-lldb를 사용해서만 그 이유를 알 수 있다. Host API는 오류 여부만 알려주며, 원인은 출력하지 않는다.

- Memory fault : DPU가 WRAM을 벗어나거나 크기에 맞지 않게 정렬된 주소에서 load 혹은 store를 수행하려 할 때 발생한다.
- DMA fault : DPU가 WRAM 밖의 WRAM 주소로 DMA 연산을 수행하려 할 때 발생한다. 전송의 대상이 되는 전체 범위에 적용된다.
- Fault : 8가지 경우에 대해 DPU Runtime Library가 런타임에 잘못된 것을 감지할 때 발생한다.

2.4 Controlling the execution of DPUs from host applications

DPU host API는 DPU 동적 확보, DPU에 프로그램 로드, DPU를 시작하고 실행 상태를

가져오는 기능을 제공함으로써 host application과 DPU의 상호작용을 원활하게 한다.

- Obtaining DPUs : DPU는 rank로 그룹화되고, rank 내에서 각 연산은 한 번에 한 개 혹은 여러 개의 DPU에서 처리될 수 있다. 그러나 종종 모든 rank의 모든 DPU가 같은 작업을 하게끔 하고 싶을 수 있으며, 가끔 이는 rank 단위로 작업하는 것보다 성능을 저하시키지도 않는다. 결과적으로 host API는 다수의 DPU rank를 포함한 DPU의 집합에서 작동한다.
- Loading programs : 이 연산은 `dpu_load`를 사용하여 한 세트의 모든 DPU를 프로그래밍한다. 함수는 이진 파일 경로를 입력으로 가져와 프로그램을 지정된 DPU에 로드한다. 이 프로그램은 DPU 메모리에 지속적으로 저장되면, 원하는 만큼 재부팅되고 동일한 코드를 실행한다.(global constant 유지) 또한 application은 언제든지 새 프로그램으로 DPU를 다시 로드할 수 있다.
- Executing programs : `dpu_launch`를 호출하여 지정된 세트의 모든 DPU를 부팅할 수 있다. 일부 리소스는 부팅 전 초기화되며, application은 DPUs를 동기적으로 또는 비동기적으로 실행할 수 있다:

2.5 Communication with host application

C host API는 host 메모리와 DPU 메모리 사이에 데이터를 전송하는 함수(dpu_copy_from, dpu_broadcast_to, dpu_push_xfer)를 제공한다. 이 함수를 사용하기 위해서는 대상 메모리에 따라 정렬 제한이 있다. IRAM과 WRAM은 8바이트로 정렬되어야 하며, WRAM은 4바이트로 정렬되어야 한다. 다만 전체 rank로 전송하는 성능을 유지하면서 DPU 간 다른 데이터를 전송하려 할 때 정밀도를 제공하지 못하므로 그 경우에는 dpu_prepare_xfer, dpu_push_xfer를 사용할 수 있다.

2.6 Advanced Features of the Host API

- Multiple Ranks Transfer : 여러 DPU rank의 DPU 집합에서 copy 함수를 사용하는 것은 각 DPU rank에 대해 함수를 호출하는 것보다 효율적이다. Multi-ranks operation을 수행하지 않고 thread를 생성하지 않으려면, 다음과 같이 전달할 수 있다. `dpu_alloc(DPU_ALLOCATE_ALL, "nrThreadsPerRank=0", &dpu_set);`
- Asynchronism : `dpu_broadcast_to`와 `dpu_push_xfer`는 `DPU_XFER_ASYNC` flag를 통해 비동기적으로 동작할 수 있다.
- Callbacks : `dpu_callback` 함수를 통해 일부 비동기 작업(copy, launch) 사이의 함수에 대한 호출을 예약할 수 있다.

2.7 Logging

stdio header를 추가한 후, 다음 기능 중 하나를 이용하여 logging을 사용할 수 있다.

- `void printf(const char *format, ...)` : 형식이 지정된 문자열을 stdout buffer에 write
- `void puts(const char *str)` : 끝에 '₩n'이 추가됨
- `void putchar(int x)` : character를 write

위의 학습한 내용들을 바탕으로 코드를 실행해보며 익히는 과정에서 해결하지 못한 의문점이 있다.

```
#include <barrier.h>
#include <defs.h>
#include <stdint.h>

BARRIER_INIT(my_barrier, NR_TASKLETS);

uint8_t coefficients[128];

/* Computes the sum of coefficients within a tasklet specific range. */
int compute_checksum() {
  int i, checksum = 0;
  for (i = 0; i < 32; i++)
    checksum += coefficients[(me() << 5) + i];
  return checksum;
}

/* Initializes the coefficient table. */
void setup_coefficients() {
  int i;
  for (i = 0; i < 128; i++) {
    coefficients[i] = i;
  }
}

/* The main thread initializes the table and joins the barrier to wake up the
 * other tasklets. */
int master() {
  setup_coefficients();
  barrier_wait(&my_barrier);
  int result = compute_checksum();
  return result;
}

/* Tasklets wait for the initialization to complete, then compute their
 * checksum. */
int slave() {
  barrier_wait(&my_barrier);
  int result = compute_checksum();
```

```
  return result;
}

int main() {
  return me() == 0 ? master(): slave();
}
```

해당 코드는 barrier를 활용해 `coefficient` 바이트 배열 부분의 checksum을 계산하는 코드이다. 0번 tasklet은 0+…+31을, 1번 tasklet은 32+…+63을, 2번 tasklet은 64+…+91을, 3번 tasklet은 92+…+127을 계산한다. 이때 2번 tasklet이 64+…+95까지 32개를 계산하고 마지막 tasklet도 마찬가지로 32개를 계산할 것이라 예측했는데, 2번 tasklet은 28개를 계산하고 3번 tasklet이 나머지를 계산했다. 이 부분이 왜 그런지는 알지 못했다.

3. 5월 계획

남은 4월은 gdb-lldb를 활용한 디버깅과 시뮬레이터를 활용한 성능 측정 방법을 공부할 것이다. 이후 5월은 공부한 것을 바탕으로 PIM을 활용한 sort-merge join 알고리즘을 디자인하는 것을 목표로 한다.

4. 참고 문헌

UPMEM SDK, https://sdk.upmem.com/2023.2.0/