CSED211: Lab 8 & 9 (due Nov. 27)

손량(20220323)

Last compiled on: Monday 27th November, 2023, 12:15

1 개요

캐시 메모리의 동작에 대해 이해하기 위해, 메모리 접근 정보가 주어졌을 때 캐시 메모리의 동작을 소프트웨어로 시뮬레이션 해 보고, 특정 크기의 캐시에 대해 전치 행렬 연산을 최적화해 본다.

2 Part A: Cache Simulator

이번 lab에서 구현할 캐시 시뮬레이터는 다양한 크기의 캐시 메모리를 지원해야 하고, aligned access 상황만 고려해도 된다. 또한, 캐시 메모리에 저장되는 정보는 시뮬레이션에 필요 없기 때문에 메타데이터만 저장하면 된다.

캐시 메모리의 메타데이터를 저장하기 위하여 다음과 같은 구조체를 정의하였다.

```
struct cache_line {
  bool valid;
  uint64_t tag;
  int last_accesed;
};

struct cache_set {
  bool valid;
  int line_count;
  struct cache_line *lines;
};

struct cache_store {
  int set_bits;
  int lines_per_set;
  int block_bits;
  struct cache_set *sets;
};
```

이 구현에서는 다양한 크기의 캐시 사이즈를 지원하기 위해 `calloc`을 사용하여 동적으로 메모리 할당을 받도록 코딩하였다. 또한 메모리를 낭비하지 않기 위하여 프로그램을 초기화할 때 캐시 메모리의 각 set에 해당되는 메모리만 다음과 같이 `cache_store_init` 함수에서 초기화하도록 구현하였다.

```
void cache_store_init(struct cache_store *store, int set_bits,
```

```
                    int lines_per_set, int block_bits) {
  store->set_bits = set_bits;
  store->lines_per_set = lines_per_set;
  store->block_bits = block_bits;
  store->sets = calloc(1 << set_bits, sizeof(struct cache_set));
}
```

캐시 메모리에 대한 접근을 시뮬레이션하는 코드는 `simulate_access` 함수에서 수행된다. 앞서 설명하였듯 초기화 과정에서는 각 set에 해당하는 메모리만 할당되기 때문에, 만약 아직 메모리가 할당되지 않은 set을 참조한다면 메모리를 할당하는 코드를 작성하였다. 캐시에 접근할 때, 우선 주어진 주소에 해당하는 캐시 라인이 있는지 tag를 비교하여 판정하고, LRU policy에 맞게 작동할 수 있도록 마지막으로 접근한 시간을 업데이트해 준다. 그렇지 않은 경우에는 invalid line을 찾아 tag와 마지막으로 접근한 시간을 업데이트하고, 만약 모든 line이 valid라면 LRU policy에 따라 가장 접근한지 오래된 line을 evict하도록 코딩하였다.

테스트 결과, 레퍼런스 케시 시뮬레이터와 같은 동작을 함을 확인할 수 있었다.

```
$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6
         traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2
         traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1
         traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67
         traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29
         traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10
         traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0
         traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743
         traces/long.trace
    27

TEST_CSIM_RESULTS=27
```

3 Part B: Optimization

3.1 32 by 32

우선 기본적인 전략으로, 8 by 8 블록으로 행렬을 나누어 locality를 보장하도록 하였다. 하지만 주어진 캐시 크기의 제약 때문에, 모든 대각선상 위에 있는 블록에 대해 `A`의 블록의 각 줄과 `B`의 블록의 각 줄이 같은 cache line을 차지한다는 문제점이 있었다. 이 경우 블록의 원소를 순차적으로 복사한다면 cache miss가 대각선 블록에서 다량으로 발생하기 때문에, 대각선상의 블록을 `A`에서 `B`로 복사하는 경우에는 블록의 upper triangle과 lower triangle을 따로 복사하여 cache miss 개수를 줄였다. 이러한 access pattern에서는 각 triangle의 각 row

를 복사할 때 처음 iteration을 제외하고는 적은 수의 cache miss가 발생하기 때문에 cache miss 개수를 최적화할 수 있었다. `test-trans`를 실행한 결과는 다음과 같다.

```
$ ./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183,
    evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

3.2 64 by 64

64 by 64의 경우, 8 by 8 블록으로 행렬을 나누면 우선 각 블록의 n번째 row와 $n+4$번째 row가 같은 cache line을 차지한다. 또한, 대각선상의 블록들은 `A`의 n번째 row와 `B`의 n번째 row 또한 같은 cache line을 차지하기 때문에 순차적으로 원소를 복사할 때에는 다량의 cache miss가 발생한다. 이를 해결하기 위해 지역 변수 8개를 사용하여, 블록의 첫 4줄을 우선 4 by 4 단위로 transpose 한 결과를 `B`에 저장하고, 그 다음 4줄을 transpose할 때 `B` 내부에서 앞서 transpose된 4 by 4 블록을 옮기는 방식으로 구현하여 cache miss 개수를 최작화하였다. `test-trans`를 실행한 결과는 다음과 같다.

```
$ ./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9138, misses:1107, evictions:1075

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723,
    evictions:4691

Summary for official submission (func 0): correctness=1 misses=1107

TEST_TRANS_RESULTS=1:1107
```

3.3 61 by 67

61 by 67의 경우, 앞선 경우와 달리 행렬의 크기가 8의 배수가 아니기 때문에 굳이 access pattern을 collision을 피해 구성할 필요 없이 순차적으로 복사해도 되었다. 주어진 캐시 메모리를 모두 활용할 수 있도록 블록 크기를 16 by 8으로 잡아 cache miss 개수를 줄일 수 있었다.

```
$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6226, misses:1953, evictions:1921

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3756, misses:4423,
    evictions:4391

Summary for official submission (func 0): correctness=1 misses=1953

TEST_TRANS_RESULTS=1:1953
```