컴퓨터 SW 시스템 개론
LAB 5
Cache Lab

20190650 김관호

1. 배운 것

캐시가 어떻게 구현되는지를 배웠다. 캐시는 Set -> Line -> Block 순으로 세부적으로 나누어지고, Block은 valid bit, tag bit, block bit으로 구성된다. 캐시에 접근하는 방법은 먼저 참조하고자 하는 메모리의 주소값의 binary bits를 캐시의 명세에 맞게 tag, set, block offset으로 자른다. 그리고 이에 해당하는 set으로 가서 tag와 동일하고 유효(valid)한 block이 찾고자 하는 메모리 데이터이다. 만약 그런 block이 존재한다면 cache hit라고 하고, 존재하지 않는다면 cache miss라고 한다. cache miss가 일어나면 메모리로 가서 해당 데이터를 찾아 캐시에 넣어준 후 데이터를 부른 곳으로 되돌려준다.
캐시는 특히 반복문 속에서 중요한 역할을 한다. 같은 값이 계속 참조될 때 해당 값이 캐시에 있다면 메모리까지 가지 않아도 되므로 효율적이게 된다. 이를 temporal locality라 한다.
배열을 사용할 때 배열의 특정 index를 참조하면 캐시는 해당 index의 값만 들고오는게 아니라 block size만큼 전후의 배열 값들도 함께 들고온다. 이렇게 직접 불리지 않았는데 간접적으로 캐시에 올라가고 나중에 이 값이 참조되는 상황에서도 메모리까지 가지 않아도 되므로 효율적이게 된다. 이를 spatial locality라 한다.
이번 과제에서는 set size, line size, block size와 메모리 참조 명령들이 주어질 때 캐시의 상황을 모사해보고, 전치 행렬을 구하는 함수를 캐시 친화적으로 구현해보며 캐시의 동작 원리와 캐시 친화적으로 코드를 짠다는 것이 무엇인지 알아볼 것이다.

2. 구현 과정

Part A

캐시를 모사해야 한다. set, line, block size가 주어졌을 때 동적할당을 이용해 적절하게 캐시를 만들어주고, 주소값이 주어졌을 때 행위를 흉내 내야 한다. 다음은 Part A에서 사용된 구조체 및 함수이다.

0. main

명령행 인수를 처리하고 캐시 시뮬레이션을 수행한다. getopt 함수를 사용해 명령행 인수를 처리한다. -s, -E, -b, -t 옵션을 인식하고, 각각에 대한 값을 설정한다.
그리고 set size인 S를 계산하고, cache 배열을 초기화한다. cache 배열은 **Block** 2차원 배열이다. 각 Block은 is_valid=false, tag=0, placed_time=time_count으로 초기화된다. time_count는 0에서 시작해 시뮬레이션이 실행될 때마다 1씩 증가한다. Eviction할 block을

찾을 때 사용된다.
다음으로 trace 파일에서 명령을 읽어들인다. 명령이 'M'인 경우에는 operate 함수를 두 번 호출한다. 각 명령에 대해 time_count를 증가시키고 operate 함수를 호출하여 캐시 시뮬레이션을 수행한다. 마지막으로 결과를 출력한다.

1. Struct Block

캐시 블록을 나타내는 구조체이다. 유효한지를 확인하는 bool is_valid, 태그인 unsigned int tag, 그리고 캐시에 저장된 시각인 unsigned int placed_time을 가진다. 어떤 블록이 참조 및 저장에 쓰이는지 확인하기만 하면 되므로 저장되는 데이터 같은 다른 변수는 필요없다.

2. void operate(unsigned int address)

메모리 주소를 받아와 해당 주소에 대한 캐시 조작을 수행한다.
메모리 주소를 주어진 S, E에 맞게 나눠 set_index와 tag를 계산한다. find 함수를 호출해 해당 블록이 캐시에 존재하는지 확인하고, hit 여부를 결정한다. hit이면 hit_count를 증가시키고, 해당 블록의 time_count를 갱신한다. miss이면 miss_count를 증가시키고, 캐시에 새로운 블록을 추가(place)하거나 교체(evict)한다.

3. bool find(unsigned int set_index, unsigned int tag)

set_index 내의 블록 중 특정 태그를 가진 블록을 찾아 hit 여부를 확인한다.
해당 set의 모든 블록을 확인하면서 유효하면서 태그가 일치하는 블록이 있는지 찾는다. 발견된 경우, 해당 블록의 저장 시각을 갱신하고 true를 반환한다. 발견되지 않은 경우, false를 반환한다.

4. bool is_full(unsigned int set_index)

set_index의 set이 가득 찼는지 확인합니다.
해당 set의 모든 블록의 is_valid를 확인하면서 빈 블록이 있는지 확인한다. 빈 블록이 없으면 세트가 가득 찬 상황으로, true를 반환한다. 빈 블록이 있는 경우, 아직 가득 차지는 않았으므로 false를 반환한다.

5. void evict(unsigned int set_index, unsigned int tag)

set에서 가장 최근에 사용되지 않은(Least Recently Used) 블록을 교체합니다.
해당 set의 모든 블록을 확인하면서 LRU 블록을 찾아내고, 해당 블록의 태그를 새로운 태그로 교체하고, 교체된 블록의 저장 시각도 함께 갱신한다.

6. void place(unsigned int set_index, unsigned int tag)

주어진 set에 빈 블록이 있으면 새로운 블록을 추가한다.
해당 세트의 모든 블록을 확인하면서 is_valid=false인 빈 블록을 찾는다. 빈 블록이 있다면, 해당 블록의 태그, 저장 시각을 갱신한다. is_full()=false인 상황에서만 호출되므로 빈 블록이 없는 경우는 존재하지 않는다.

Part B

전치 행렬(transpose)을 계산할 때 최대한 cache miss를 줄이는 캐시 친화적인 방법을 찾아내야 한다. 핵심 방법은 수업시간에 배운 blocking이다. 주어지는 캐시는 s=5, E=1, b=5인 directed map cache로, 한 블록당 int값 8개를 담을 수 있다. set은 총 32개다. 문제의 32x32, 64x64, 61x67의 경우에서 모두 blocking을 적용해 cache miss를 줄인다. 다음은 문제를 해결하기 위해 사용된 함수들이다.

1. void simple_trans(int M, int N, int A[N][M], int B[M][N], int block_size)

해당 함수는 blocking을 적용해 전치 수행 중에 생기는 cache miss를 최소화한다. 수업 시간에 배운대로 전체 배열을 블록 단위로 나누고 해당 블록 안의 값들을 캐시에 올려 전치를 수행한다. 해당 함수는 32x32, 61x67 두 경우 모두에 block_size만 달리 하여 적용된다. 먼저, 바깥쪽 두 개의 반복문(i, j)은 주어진 행렬을 block_size x block_size 크기의 작은 블록으로 나누어 순회한다. block_size는 전체 행렬을 나누는 작은 블록의 크기다. 이어서 두 개의 내부 반복문(r, c)은 현재 블록 내부의 행과 열을 순회한다. 이 반복문에서는 현재 블록 내에서 각 원소에 접근하여 전치를 수행한다. 행과 열이 같은 원소(대각 원소)의 경우, 그대로 캐시에 올려놓게 되면 conflict miss가 계속해서 발생하므로 임시 변수(tmp)에 저장하고 캐시에 올려놓지 않고 다음 열로 넘긴다. 나중에 해당 위치에 채워넣을 것이다. 대각 원소를 제외한 나머지 원소에 대해 현재 원소의 위치를 바꾸어 전치를 수행한다. c에 대한 가장 안쪽의 반복문이 끝난 후, 대각 원소가 있는 상황(i==j)이었다면 임시 변수에 저장한 값을 이전에 저장한 위치에 넣는다. 이를 행 개수만큼 반복하면 전치가 완료된다.
추가로 61x67 배열은 행과 열의 개수가 다르므로 안쪽 두 개의 반복문에 for (int r = i; r < i + block_size **&& r < N**; r++), for (int c = j; c < j + block_size **&& c < M**; c++) 와 같은 조건을 추가해 주어진 행, 열 개수를 넘기지 않도록 했다.

2. void complex_trans_normal(int M, int N, int A[N][M], int B[M][N])

해당 함수는 64x64 행렬에 대한 전치를 수행하며, simple_trans와 같이 blocking을 적용하되 8x8 block이 아닌 4x4 block을 기본으로 진행한다. 이유는 행렬의 크기가 커지면서 k번째 행과 k+4번째 행이 속하는 set_index가 같아졌기 때문이다. 이 때문에 8x8 block을 이용해 simple_trans와 같이 넣어줄 수 없다. 행을 4개씩 읽을 때마다 conflict miss가 계속 일어날 것이기 때문이다. 또한 해당 함수에서는 지역 변수를 이용해 미리 행렬값 몇 개를 빼놓고 반복문을 거치지 않고 따로 넣어주어 conflict miss를 추가로 줄이고자 했다.
그러나 miss_count는 1,668로 2,000보다는 적었지만 만점을 받지는 못했다. 다른 방법을 고안해야 했다.

3. void complex_trans(int M, int N, int A[N][M], int B[M][N])

4x4 blocking의 가장 큰 단점은 16bytes로 계속 끊기므로 크기가 32bytes인 block을 모두 활용하지 못한다는 것이다. 또한, 4번마다 한 번씩 읽는 행이 자주 바뀌므로 그만큼 conflict miss도 자주 발생하게 된다. 이를 해결하기 위해서는 **캐시에 올려진 상태를 최대한 유지하면서 이를 이용해야 한다.** 그러기 위해 **4x4 blocking은 유지한 채로 지역 변수 int tmp[8]을 넣어 배열 A, B간 값 교환을 용이하게 중개할 것이고, B를 캐시처럼 활용할**

것(mocked cache)이며, 최대한 캐시를 늦게 교체(evict)하게 해 spatial&temporal locality를 살릴 것이다.

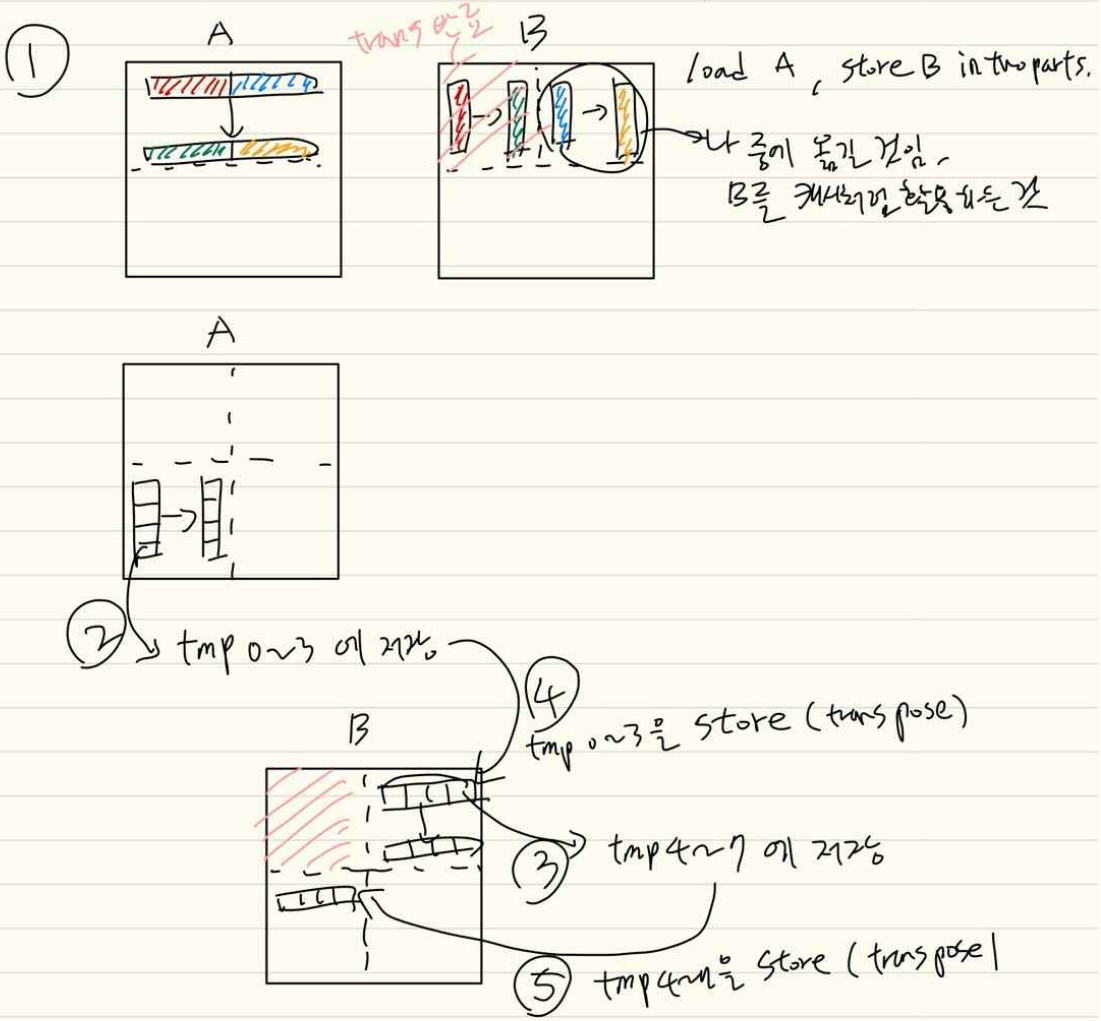
바깥쪽 두 개의 반복문은 complex_trans_normal과 동일하게 64x64 행렬을 8x8 크기의 작은 블록으로 나누어 순회한다. 첫 번째 내부 반복문부터 달라지는데, 여기서 현재 블록 내의 **행을 순회**한다. 각 행에 대해 해당 행의 8개의 원소를 임시 변수 int tmp[8]에 넣은 후 B에 tmp를 넣는다. 이 때 B 블록의 좌상단 부분이 완성된다**(그림 1번)**. B block의 우상단 부분은 제대로 된 값이 들어가있지 않다. 캐시 역할(mocked cache)을 하는 것이다. 나중에 여기 있는 값들이 알맞은 위치인 좌하단으로 들어갈 것이다.

두 번째 내부 반복문에서는 현재 블록 내의 **열을 순회**한다. 여기서는 tmp를 통한 데이터 흐름의 순서가 중요하다. 먼저 A의 좌하단 4개 행을 읽어 tmp에 넣는다**(그림 2번)**. 다음으로 캐시 역할을 하는 B 블록의 우상단 부분의 행 하나(열 4개)를 tmp의 남은 공간에 넣는다**(그림 3번)**. 다음으로 2번에서 tmp에 저장된 값을 B에 넣는다**(그림 4번)**. 이 때 **참조되는 B 블록의 행은 이미 캐시에 올려진 상태**이다. 즉, conflict miss가 일어나지 않는다. 이제 나머지 tmp에 저장된 값을 B 블록의 좌하단에 넣어 제대로 위치되게 한다**(그림 5번)**. 이 때 캐시에 올라가는 set_index는 그림 4에서 우상단에서 참조한 행과 같다. 즉, 그림 2에서 수행될 때 다른 set_index에 올라가 있는 A를 망치지 않고 conflic miss 하나만 낼 수 있게 된다. 이렇게 2~5를 4번 반복하면 B 블록의 우상단, 좌하단이 완성된다.

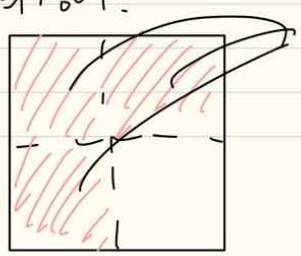
마지막으로, B 블록의 우하단은 simple_trans에서 했던 방법으로 B를 채워준다. 이렇게 8x8짜리 B 블록 하나가 완성되었고 바깥 반복문으로 돌아가 계속 반복하면 결과적으로 locality를 최대한 활용하면서 전치가 완료된다.

캐시에 들어와 있는 상태를 최대한 활용.

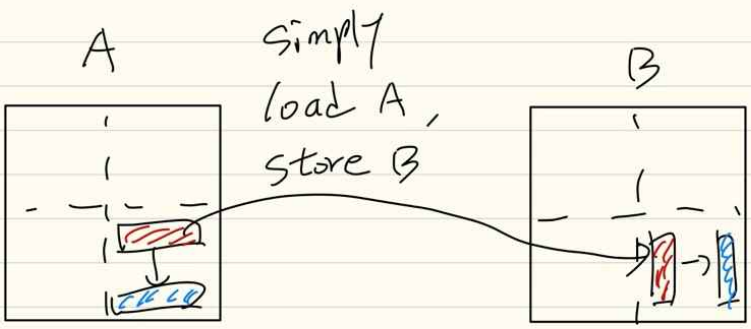
편의상 A, B의 8×8 크기 block을 A, B 라 하자.



⑤ 이후 B의 상태.



⑥



⑥ 이후 B의 상태.

Complete.

이 (해석) 가 for(i=0; i<N; i+=8)
for(j=0; j<M; j+=8)

이 작업을 계속 반복해야 한다.

3. 느낀 점

캐시의 구조에 따라 같은 용량, 같은 명령이라도 hit, miss가 다르게 날 수 있음을 알게 되었다. Part B에서는 Blocking을 통해 작은 블록을 사용하여 배열을 나누는 것이 중요했다. 특히, 작은 블록을 이용해 지역성을 활용하며 메모리에 접근하는 패턴을 최적화할 수 있었다. complex_trans 에서는 행렬의 특정 부분을 블록 단위로 순회하면서 데이터를 읽고 쓰는데, 이를 통해 spatial locality를 활용하여 데이터를 캐시에 유지할 수 있었다. 또한, 임시 변수 tmp를 사용해 값을 저장하고 다시 활용해 temporal locality를 얻을 수 있었다.
캐시 친화적인 코드를 짤수록 가독성이 떨어지는 것을 느꼈다. 간단하게 몇 줄로 전치를 구현할 수 있는데 blocking을 도입하면서부터 가독성이 확 떨어졌고, 특히 complex_trans에서 지역 변수를 넣고 이곳 저곳을 참조하고 데이터가 돌아다니면서 가독성은 더욱 떨어졌던 것 같다. 코드의 가독성을 높이기 위해 어떤 부분을 희생할 것인지, 어떤 최적화 기법을 적용할 것인지에 대한 Trade-off를 생각해야 할 것이다.