**CSED211 Cache Lab Report**

20210643 김현준

**<Lab 8 & 9 – Cache Lab>**

Lab8과 lab9 시간에 cache와 cache lab 관련 설명을 듣고 Cache-Lab을 진행하였다.

먼저, Memory Address는 Block offset bits와 Set index bits, Tag bits로 구성이 된다. 각각을 b, s, t로 표현을 하며, 전체 physical address bit가 m일 때, m=b+s+t가 된다. 이 Memory Address를 이용하여, Cache memory로 접근을 할 수 있는데, Cache Memory는 principle of locality를 만족하면서 data또는 instruction을 저장하는 메모리이다. 메모리에서 직접 가져다 쓰는 것 보다 효율적으로 데이터에 접근하기 위해서, locality를 이용하여 자주 쓰는 데이터를 캐시에 저장해 두는 것이다. Principle of locality에는 spatial, temporal locality가 있다.

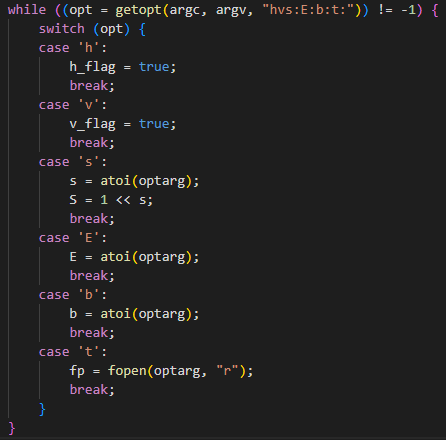
Cache memory는 valid bit, tag bit, cache block으로 구성되고, E는 set 당 line 수, S는 set의 개수를 나타낸다. Cache에 access하였을 때, valid bit가 1이고, tag가 일치하면, offset에 있는 data에 접근할 수 있다. Valid bit가 1이고 tag가 일치하여 Cache에 access되면 hit이고, 그렇지 못하면 miss이고, 해당 set의 line들에 모두 값들이 들어 있지만 miss인 경우에는 eviction이 된다. Eviction이 되는 경우에는 이미 있는 블록을 하나 바꾸어야 하기 때문에 Cache Replacement Policy가 필요하다. 대표적으로 가장 오래 전에 접근했던 캐시 블록을 새것으로 바꾸는 LRU 방식이 있는데, LRU 방식은 Part A에서 Cache Simulator를 만들 때에 사용한다.

Hit Ratio는 Cache memory에서 miss에 비해 hit이 얼마나 이뤄졌는지 나타내는 비율이다. Hit ratio가 높도록 프로그램을 설계하면 더 효율적으로 cache memory에 접근한다고 생각할 수 있다. 예시로 32bytes direct mapped cache, block size 16bytes에서 Row-major order로 구현할 경우 75%의 hit ratio가 나타나지만, column-major order로 구현할 경우 0%의 hit ratio가 나타나는 것을 확인할 수 있다. Matrix Multiplication을 수행하는 함수에서 생각해 보았을 때에도 for loop에서 row와 column을 inner loop로 할 지 바깥 loop로 할 지에 따라 hit ratio가 굉장히 달라지는 것을 확인할 수 있다. 또한, Blocking을 하여 각각의 Block에 대해서 계산을 하게 되면 조금 더 hit ratio가 높아지도록 구현할 수 있다. Hit ratio가 높아지도록 효율적인 transpose matrix 프로그램을 구현하는 것이 Part B: Optimizing Matrix Transpose에서 해야 할 과제이다.

**<Part A: Writing a Cache Simulator>**

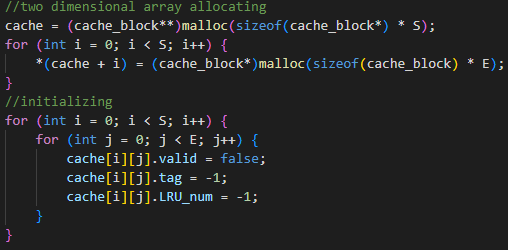
먼저, Part A에서는 캐시 시뮬레이터를 작성하여야 했다. 시뮬레이터가 해야 할 일은 trace 파일에서 “L 10, 1”과 같이 되어 있는 input을 받아 Parsing하고, Cache memory의 S, E, b bit 값을 받아서 실제 Cache가 동작하는 것처럼 hit가 발생하는지 miss가 발생하는지 전부 counting하도록 하는 것이었다.

그래서 먼저 Parsing부터 구현하고자 하였다. unistd.h 라이브러리에 있는 getopt를 이용하여 Parsing 부분을 만들었다. opt라는 integer 변수에 getopt함수의 리턴값을 받아 switch case 문을 돌게 하여 h, v의 flag와 s, E, b, t에 따라서 각 값들을 변수에 저장시키고 파일을 여는 실행 부분을 작성하였다. 입력 끝까지 읽으면 getopt가 -1를 리턴하므로, while문에서 나갈 수 있도록 구현하였다.

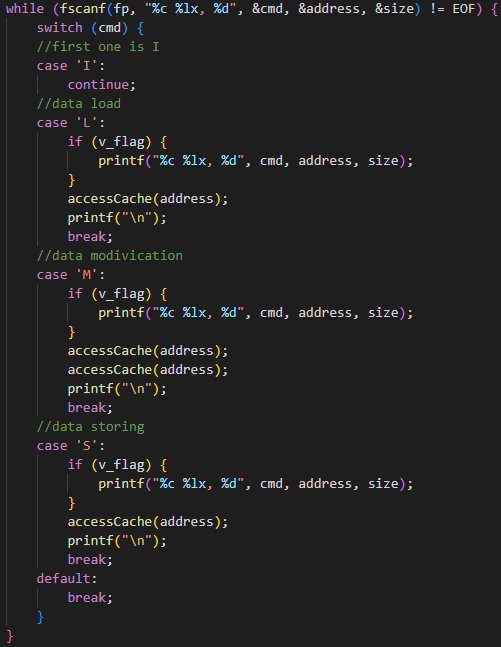


다음으로 필요한 변수들을 정리해보았다. Verbose 와 help mode를 나타낼 bool 자료형 변수를 선언하였고, hit, miss, eviction 횟수를 count할 변수, cache의 s, b, E를 저장할 변수, address 변수 등을 global variable과 local variable로 선언하였다. 그리고 cache memory를 시뮬레이션 하기 위해 캐시 블록을 나타내는 구조체를 정의하여 valid, tag, LRU\_num를 구조체 안에서 선언하였고, 그 structure의 2차원 배열로 캐시를 구현해보았다.

다음으로는 간단하게 file error와 help mode일 경우 출력되는 메시지를 작성하여 보았다. 그리고 cache 2차원 배열의 allocating 및 initializing을 해 주었다.

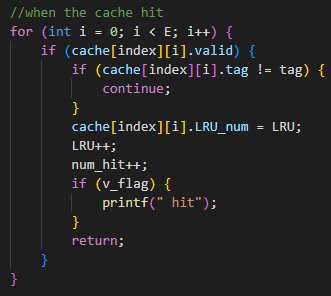


그러고 나서, trace 파일에서 입력 내용을 읽어 들어올 부분의 코드를 작성하였다. Fscanf 함수와 while, switch문으로 구현하였다. 파일의 내용이 L 10, 1과 같은 형식이기 때문에, 먼저 맨 앞에 오는 문자들을 판별하고 다음 값들을 address와 size 변수에 저장하였다. 그리고 각 문자가 L인지 M인지 S인지에 따라 함수를 호출하도록 하였다.



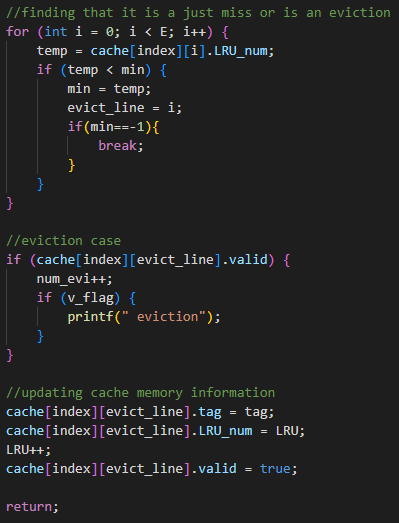
처음에 구상할 때는 세 경우의 함수를 따로 짜야 할 것 같다는 막연한 생각이 있었는데, 그렇지 않고 똑 같은 함수를 호출하여도 될 거라는 생각이 들었다. 따라서 accessCache라는 하나의 함수로 세 경우를 모두 구현하였다. M의 경우에는 캐시에 접근을 해서 load를 하고, Store도 해야 하기 때문에 L+S의 경우이므로 두 번 accessCache 함수를 호출하였다. accessCache 함수에서는 LRU policy와 hit, miss, eviction을 판별 및 counting하는 기능을 구현하였다. 따라서 위의 while문을 통과한 이후는 free로 배열의 메모리를 할당 해제하고 printSummary함수로 최종 결과를 보내는 내용을 작성하였다.

accessCache 함수는 address 값을 parameter로 받아오도록 설계하였다. s와 b bit가 전역변수로 선언되어 있어서, 그 값들을 이용하여 우선 address의 index와 tag을 구하였다. Index는 address를 b만큰 right shift하고, mask를 씌워서 tag bit 부분을 0으로 만들어 구하였고, tag는 s+b만큼 shift하여 구하였다. 그러고 필요한 변수들을 선언한 다음 Hit를 판별하는 부분을 작성하였다.



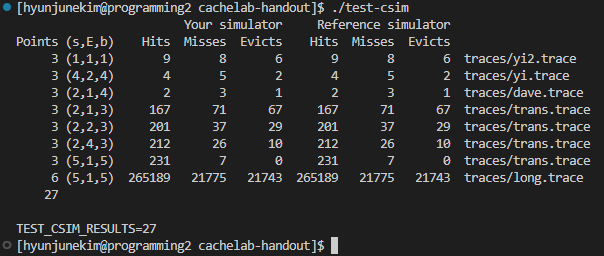
For문으로 line들을 탐색하였고, valid bit가 1이면서 tag가 일치하지 않으면 continue하고, tag가 일치하면 LRU값을 업데이트 하고, hit 횟수를 1회 늘렸다. 그리고 verbose mode의 경우 과정이 출력되어야 되므로 hit을 출력해주었다. 이 때 LRU값은 main에서 0으로 설정을 해 주었고, 블록에 access할 때마다 1씩 늘려서 LRU값이 작을수록 더 오래전에 access한 블록임을 알 수 있도록 하여 lru policy를 구현하고자 하였다. for문의 마지막에는 return을 넣어 hit을 판별하면 바로 리턴되도록 하였다.

Hit이 아닌 경우는 곧 miss이므로 바로 아래에 miss 부분을 구현하였다. Miss의 경우는 단순하게 num\_miss만 1 증가시켜 주었다. 그 다음으로 miss가 단순 miss인지 eviction이 있는지 판별하는 부분을 구현하였다. Cache의 블록 구조체에 저장되어 있는 LRU\_num 값이 작을수록 예전에 access된 값이기 때문에, E값에 대해서 탐색을 하면서 가장 minimum의 LRU\_num을 가지고 있는지 탐색하였다. 그리고, miss의 경우 어느 블록에 access해야 할 지 알아야 했는데, 그 부분은 min이 -1일 경우에 해당했다. 왜냐하면, 처음에 블록들을 initializing 해줄 때에 LRU\_num을 -1로 하였기 때문이다. 탐색을 하면서 E가 0부터 증가하면서 블록들의 LRU값을 찾는데, 0 이상인 경우 한 번 이상 access 된 것이고 -1로 남아 있으면 비어 있는 블록이기 때문이다. 이 부분은 아래와 같은 코드로 구현하였다.



결론적으로, eviction 탐색과 함께, 어느 블록의 값을 바꾸어야 할 지 탐색을 마치고 마지막 부분에서 각 값을 update한 후 리턴하는 것으로 accessCache함수를 구현하였다. Part A를 수행하면서는 이 부분에서 약간 어려움을 겪었는데, LRU policy를 어떻게 구현해야 할 지 고민이 있었다. 그래서 처음에 구현을 해 보았을 때에는 LRU가 제대로 작동하지 않아서 모든 test case들이 directed-mapped cache인 것처럼 작동하는 오류가 있었다. 위에서 설명한 것과 같은 방식으로 문제를 해결할 수 있었다.

이렇게 작성한 코드를 ./test-csim을 이용하여 테스트해 보았다. 그 결과 27points가 제대로 나오는 것을 확인할 수 있었고, 이렇게 Part A: Cache Simulator를 완성할 수 있었다.

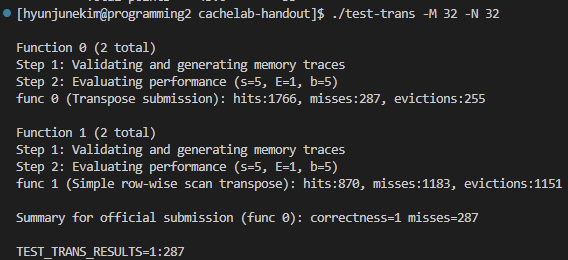


**<Part B: Optimizing Matrix Transpose>**

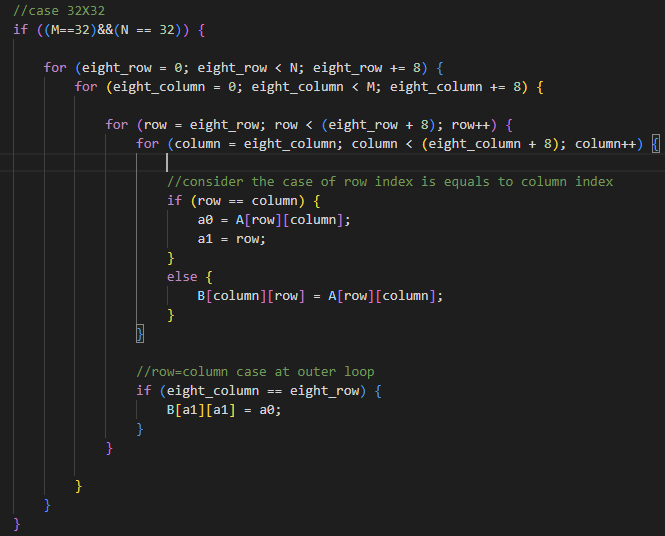
두 번째 파트에서는 Matrix Transpose를 계산하는 함수를 최적화하여 miss가 기준치 아래로 내려가도록 해야 했다. 32X32, 64X64, 61X67의 사이즈를 가지는 matrix에 대해서 각각 최적화하여야 했다. 사용되는 Cache memory의 s=5, E=1, b=5이다. 먼저, 12개 이내로 local variable들을 선언해준 다음, transpose\_submit 함수에서 각 경우에 따라 작동할 수 있도록 구현하였다. 먼저 32X32를 구현하였다.

우선, 기본적으로 32X32의 경우 8X8 size의 matrix로 blocking하여 transpose를 계산하였다. 왜냐하면 b=5이기 때문에 2^5=32byte의 data가 캐시에 저장될 수 있고, 계산하는 것이 integer 배열이므로, 4byte씩 총 8개가 한 블록에 저장될 수 있기 때문이다. 그렇게 함으로써 hit rate를 높일 수 있을 것이다. 따라서 각 블록의 시작 index인 eight\_row와 eight\_column 변수를 선언하고, row, column 변수로 반복문에서 각 전치행렬 부분을 구해주었다. 해당 loop 안에서, 이제 B[column][row]=A[row][column]으로 transpose matrix를 구하여야 하는데, row와 column이 같을 때 miss eviction이 발생하게 될 것이다.

왜냐하면 A배열의 시작 주소를 a라 하고 B의 시작 주소를 b라고 하면, b=a+0x100일 것이기 때문이다. Tag bit는 신경쓰지 않고 s=5, b=5를 고려해보면 A와 B의 행이 같을 때, 즉 A의 row와 B에서의 행인 column이 같을 때 set index는 같고 tag는 다른 상황이 생긴다. 즉 이 경우가 miss eviction이고 나머지 계산의 경우 hit가 많을 것으로 생각하였다. 따라서 루프에서 row와 column이 같은 경우를 따로 빼 주어 temporary 변수로 row, column index와 해당 값을 저장해 놓은 다음 이를 다시 B에 대입하도록 하여 함수를 돌려 보았는데, 기준인 300회 미만의 miss 횟수가 나타나는 것을 확인할 수 있었다.



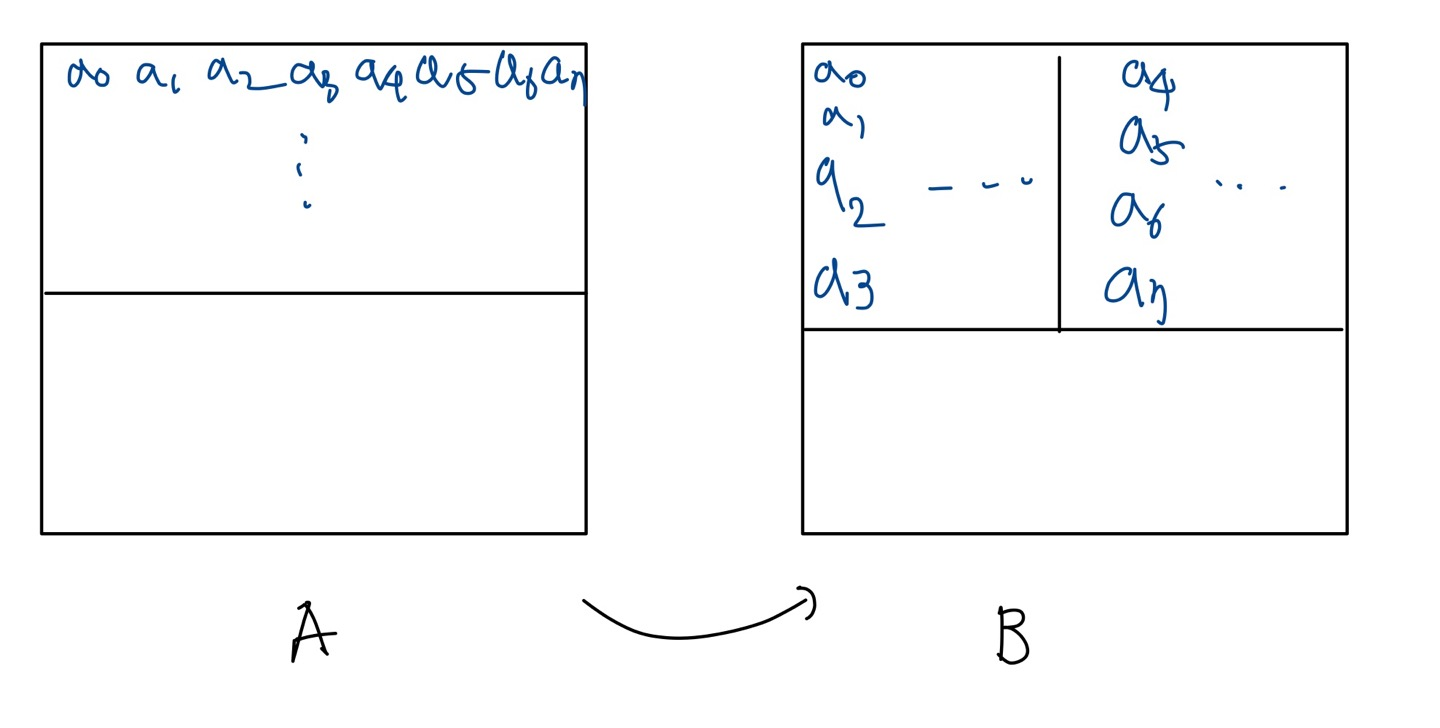
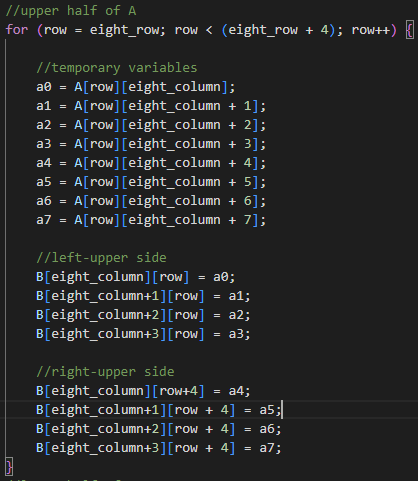
구현 코드는 아래와 같이 작성하였다. 4중 for 반복문을 사용했고, 가장 inner for문에서 row와 column이 같은 지를 확인 후, 같으면 row와 해당 A 값을 저장했다가 loop 이후 diagonal 부분인지 체크 후 B에 값을 넣어주었다.



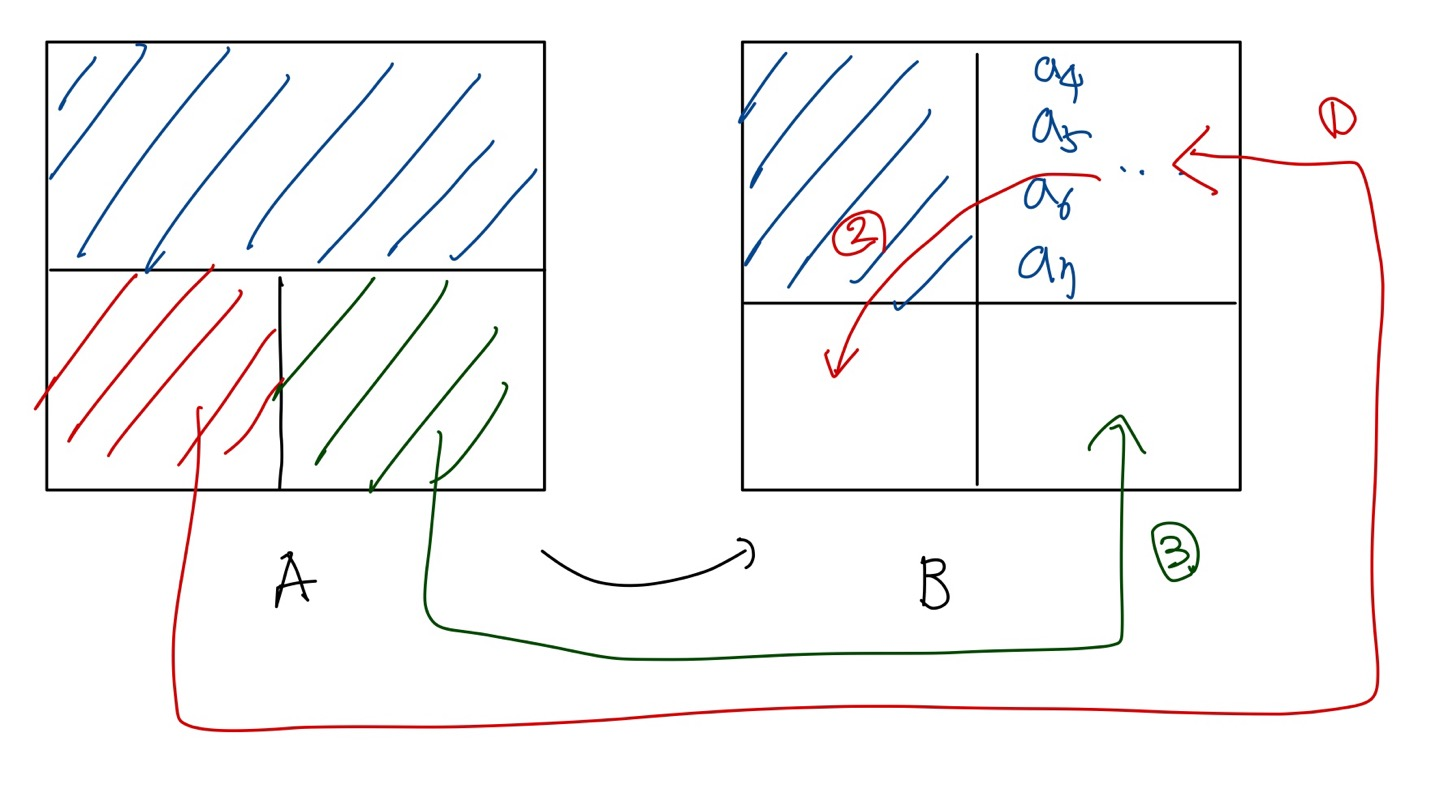
다음으로 64X64 case를 구현하였다. 이 경우도 마찬가지로 8X8에서 blocking하여 계산을 해 주려고 하였는데, 더 최적화가 필요했다. 32X32때와 같은 방식으로 row와 column이 같은 경우에만 loop에서 빼서 시도해보았으나 아래와 같이 4000여개의 miss가 나왔고, 기준치는 3000미만으로 구현해야 하기 때문이었다.



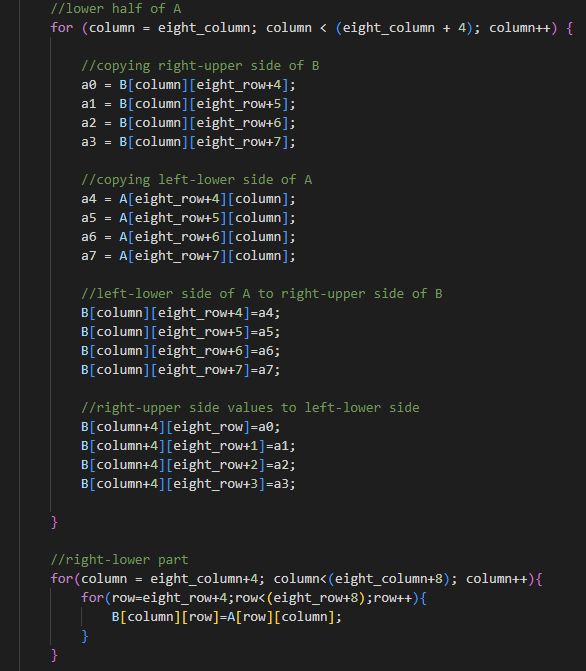
따라서 먼저 8X8 블록 안에서 위쪽 부분과 아래쪽 부분으로 나누어 처리해보려고 생각하였고, A의 위쪽 부분을 B로 옮길 때 모두 B의 행과 매치시키는 것이 아니라 절반은 B의 우측 위 부분에 매치시켜서 miss횟수를 줄여보려고 하였다. 나중에 우측 위 부분 값들을 다시 옮겨주어야 하지만, 이렇게 하면 miss 자체를 줄일 수 있을 것으로 생각하였다



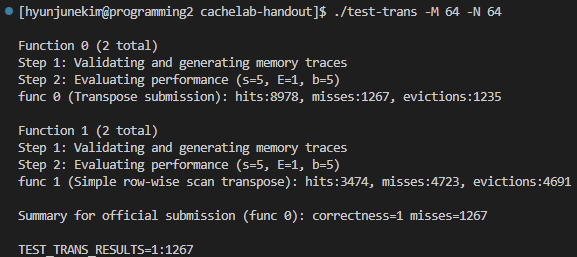
다음으로는 아래쪽 절반 부분을 B로 옮겨야 했는데, 먼저 해야 할 일은 오른쪽 위에 옮겨진 값들을 왼쪽 아래로 내려야 되는 것이었다. 그런데 A의 왼쪽 아래 부분을 B의 오른쪽 위 부분으로 옮겨야 하는데, 현재 가장 최근 access한 부분이 B의 column이 0~3의 부분이므로, 여기서 캐시 miss를 시키려면 이 column(B의 행)에 접근을 해야 했다. 따라서 결론적으로 A의 왼쪽 아래 부분을 B의 오른쪽 위 부부분으로 먼저 매치시키고, 다음으로 원래 오른쪽 위에 있던 부분을 왼쪽 아래로 매칭시켰다. 마지막으로 남은 부분인 오른쪽 아래도 따로 매칭시켜 주었다.



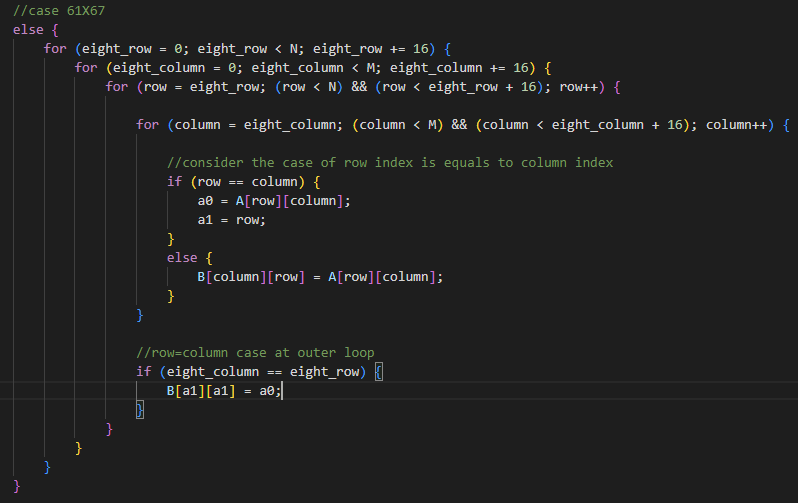
코드는 아래와 같이 8X8의 각 블록을 탐색하는 이중 반복문 안에서 upper, lower, left, right 부분에서 어떻게 작동하는지 나누어 구현하였다.



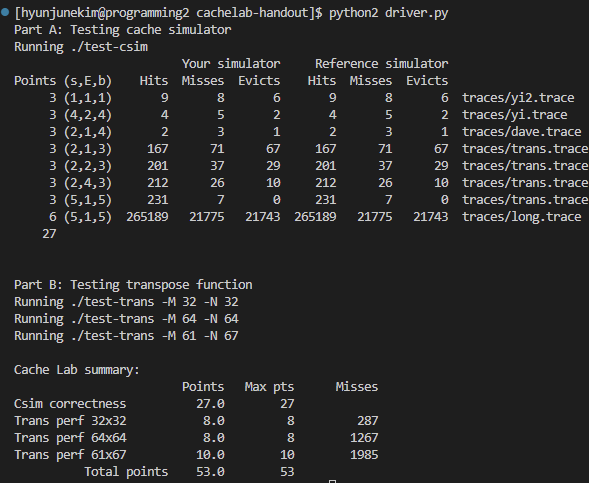
그 결과 test를 해 보니 잘 작동하는 것을 확인할 수 있었다. 기준치가 1300회 miss 미만이었는데, 1267회의 miss를 기록하였다.



마지막으로 61X67의 경우는 처음 32X32를 풀 때처럼, 각 블록으로 나누어서 풀고 row와 column이 같을 때만 따로 빼서 계산해주는 식으로 구현을 해 보았다. 그 결과 바로 기준인 2000회 미만인 1985회의 miss를 얻을 수 있었다. 코드는 아래와 같다.



마지막으로 driver.py를 이용하여 전체 점수를 산출해보았다. 그 결과 아래와 같이 total point인 53점이 나오는 것을 확인할 수 있었다. Part A와 part B 모두 적절히 구현하였다는 것을 확인해볼 수 있었다.



이렇게 Lab 8 & 9 - Cache Lab을 모두 구현하였다. 다만 약간의 문제가 있었는데, Part B의 64X64 case와 61X67 case에서 time out error가 발생하는 경우가 생겼다. 계속 발생하는 것은 아니고 어쩔 때는 time out이 발생하고 어쩔 때는 발생하지 않는 상황이 생겼다. 구현한 코드를 조금 더 최적화해보려고 노력해보았고, Lab Q&A 게시판에서 조교님께서 채점 시 alarm(180)으로 진행하겠다는 답변을 보고, test-trans.c에서 해당 부분을 수정하고 여러 번 다시 돌려본 결과 time out 없이 제대로 점수가 나오는 것을 확인할 수 있었다.