CSED211: CacheLab report / 20230784 jaewonyu 유재원

1. 개요

   본 과제는 직접 캐시의 동작을 구현해보는 과정에서 캐시의 동작을 이해하고, 이해한 동작을 바탕으로 matrix transpose 에 대하여 cache miss 가 최소가 되도록 하는 알고리즘을 짜보며 캐시의 동작을 더욱 확실하게 이해하는데 그 의의가 있다.

2. Part A

   간단한 캐시 시뮬레이터를 만드는 문제이다. 문제 해결을 위하여 다음과 같이 구조체를 정의하고 시작하였다

```
typedef struct
{
    int hits;
    int misses;
    int evictions;
} simulate_result;

typedef struct cache_line
{
    long tag;
    struct cache_line *next;
} cache_line;

typedef struct
{
    int num_lines;
    cache_line *lines;
} cache_set;

typedef struct
{
    int s;
    int E;
    int b;
    char *t;
    cache_set *sets;
} cache_storage;
```

   문제 조건 상 line 의 valid tag 등은 필요가 없기 때문에 구현하지 않았다. 또한 각종 변수들의 자료형의 경우 대부분은 int로 가정하고 문제를 풀어도 괜찮을 것 같아(오버플로가 일어나지 않을 것으로 예상되어) 대부분 int로 잡았다. 하지만 tag의 경우 주소의 일부분이기에 long 자료형을 이용하였다. cache 의 경우 LRU policy를 따르도록 구현해야 하는데, writeup 파일, ppt 등을 참고해 보았을 때 최종 접근을 어떤 방식으로든 기록하여 구현한다는 것으로 보인다. 그러나 어떤 방식으로 기록할 지 잘 모르겠어 linked list로 구현 후, 항상 head 에 추가하고, 지울 때에는 가장 마지막 node를 지우는 방식으로 LRU policy를 따르도록 구현하였다. 그것을 위한 보조 함수는 다음과 같이 준비하였다.

```
void add_line(cache_set *set, long tag)
{
    cache_line *line = malloc(sizeof(cache_line));
    line->tag = tag;
    line->next = set->lines;
    set->lines = line;
    set->num_lines++;
}

void remove_line(cache_set *set, int index)
{
    if (index == 0)
    {
        cache_line *temp = set->lines;
        set->lines = set->lines->next;
        free(temp);
        set->num_lines--;
        return;
    }

    cache_line *line = set->lines;
    for (int i = 0; i < index - 1; i++)
    {
        line = line->next;
    }
    cache_line *temp = line->next;
    line->next = line->next->next;
    free(temp);
    set->num_lines--;
}
```

이제 이 구조를 바탕으로, 캐시 시뮬레이팅을 수행하기만 하면 된다. 캐시 초기화는 다음과 같이 구현하였다.

```
void init_cache(cache_storage *cache)
{
    double S = pow(2, cache->s);
    cache->sets = (cache_set *)calloc(S, sizeof(cache_set));
}
```

사실 이 때에도 S 를 int 라 가정하여도 되었을 것 같다. pow 함수의 기본 반환이 double 형이기에 그대로 이용하였지만 딱히 의미가 있는 것은 아니다. calloc은 초기화를 함께 해준다 알고있어 따로 초기화를 진행하진 않았다. 다음으로 main 함수의 동작은 다음과 같다.

```c
int main(int argc, char *argv[])
{
    cache_storage cache;
    simulate_result result = {0, 0, 0};
    int opt;
    while ((opt = getopt(argc, argv, "s:E:b:t:")) != -1)
    {
        switch (opt)
        {
        case 's':
            cache.s = atoi(optarg);
            break;
        case 'E':
            cache.E = atoi(optarg);
            break;
        case 'b':
            cache.b = atoi(optarg);
            break;
        case 't':
            cache.t = optarg;
            break;
        default:
            break;
        }
    }
    init_cache(&cache);

    char buffer[256];
    FILE *file = fopen(cache.t, "r");
    while (fgets(buffer, 256, file) != NULL)
    {
        if (buffer[0] == 'I')
        {
            continue;
        }
        char instruction = buffer[1];
        long address = strtol(buffer + 3, NULL, 16);
        simulate(&cache, &result, instruction, address);
    }
    fclose(file);

    printSummary(result.hits, result.misses, result.evictions);
    return 0;
}
```

필요한 구조체를 선언하고, 조교님께서 알려주신 대로 문자열을 파싱하였다. writeup 파일에는 보조 기능을 추가하는 것을 권장하였지만, 그다지 어려운 구현이 아닐 것으로 보여 처음부터 구현하지 않는 방향으로 잡았다. 사용자의 입력을 모두 받은 후, 그 입력값에 맞추어 캐시를 초기화해 준 후, 파일의 내용대로 시뮬레이팅을 진행하도록 코드를 작성하였다. 사실 종료 전 메모리를 free 해주거나 하는 등 작업을 해 주었다면 좋았겠지만, 요구된 사항이 아니기에 특별히 메모리 성능 등을 따지지는 않았다.

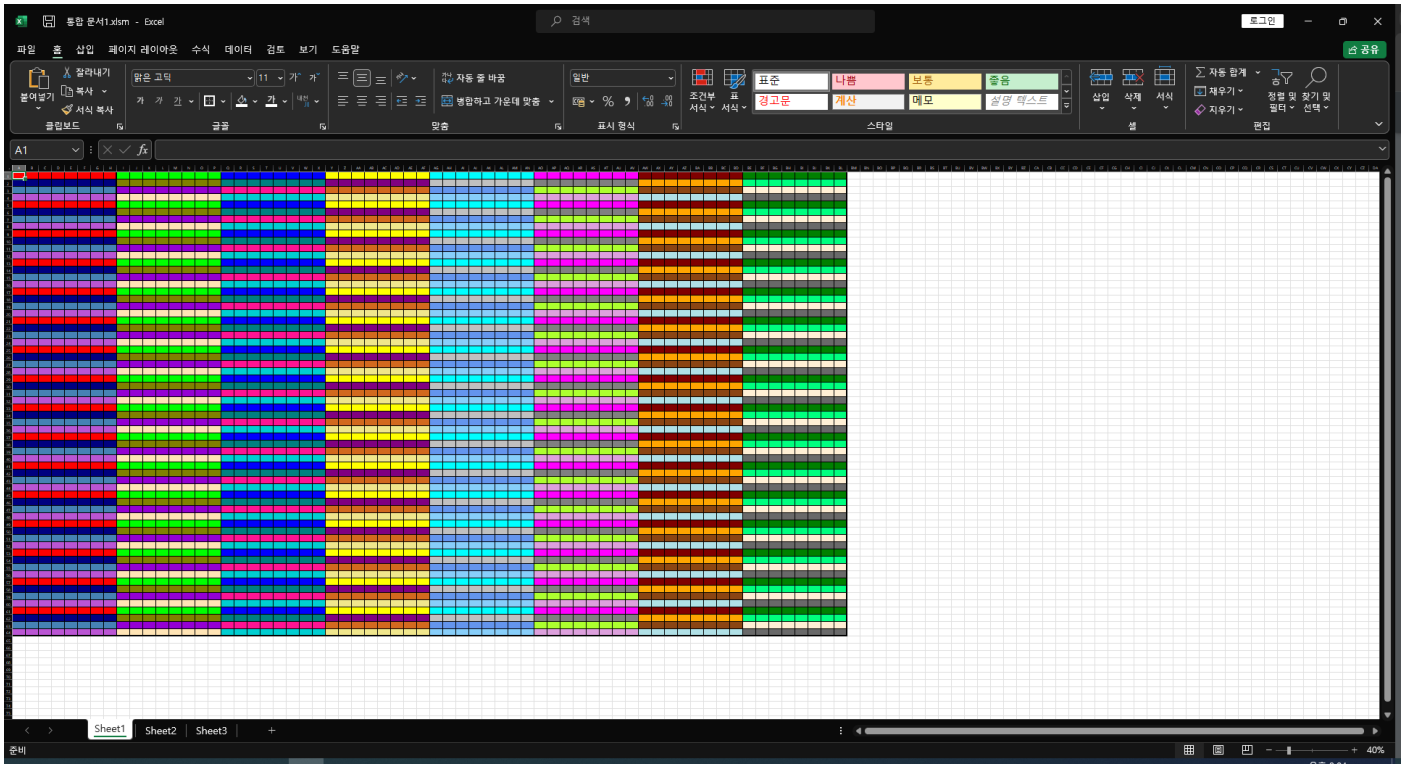
```
void access_cache(cache_storage *cache, simulate_result *result, long address)
{
    long tag = address >> (cache->s + cache->b);
    int set_index = (address >> cache->b) & ((1 << cache->s) - 1);
    cache_set *set = &cache->sets[set_index];
    cache_line *line = set->lines;
    for (int i = 0; i < set->num_lines; i++)
    {
        if (line->tag == tag)
        {
            result->hits++;
            remove_line(set, i);
            add_line(set, tag);
            return;
        }
        line = line->next;
    }
    result->misses++;
    if (set->num_lines < cache->E)
    {
        add_line(set, tag);
        return;
    }
    result->evictions++;
    remove_line(set, cache->E - 1);
    add_line(set, tag);
}

void simulate(cache_storage *cache, simulate_result *result, char instruction, long address)
{
    switch (instruction)
    {
    case 'M':
        access_cache(cache, result, address);
    case 'L':
    case 'S':
        access_cache(cache, result, address);
        break;
    }
}
```

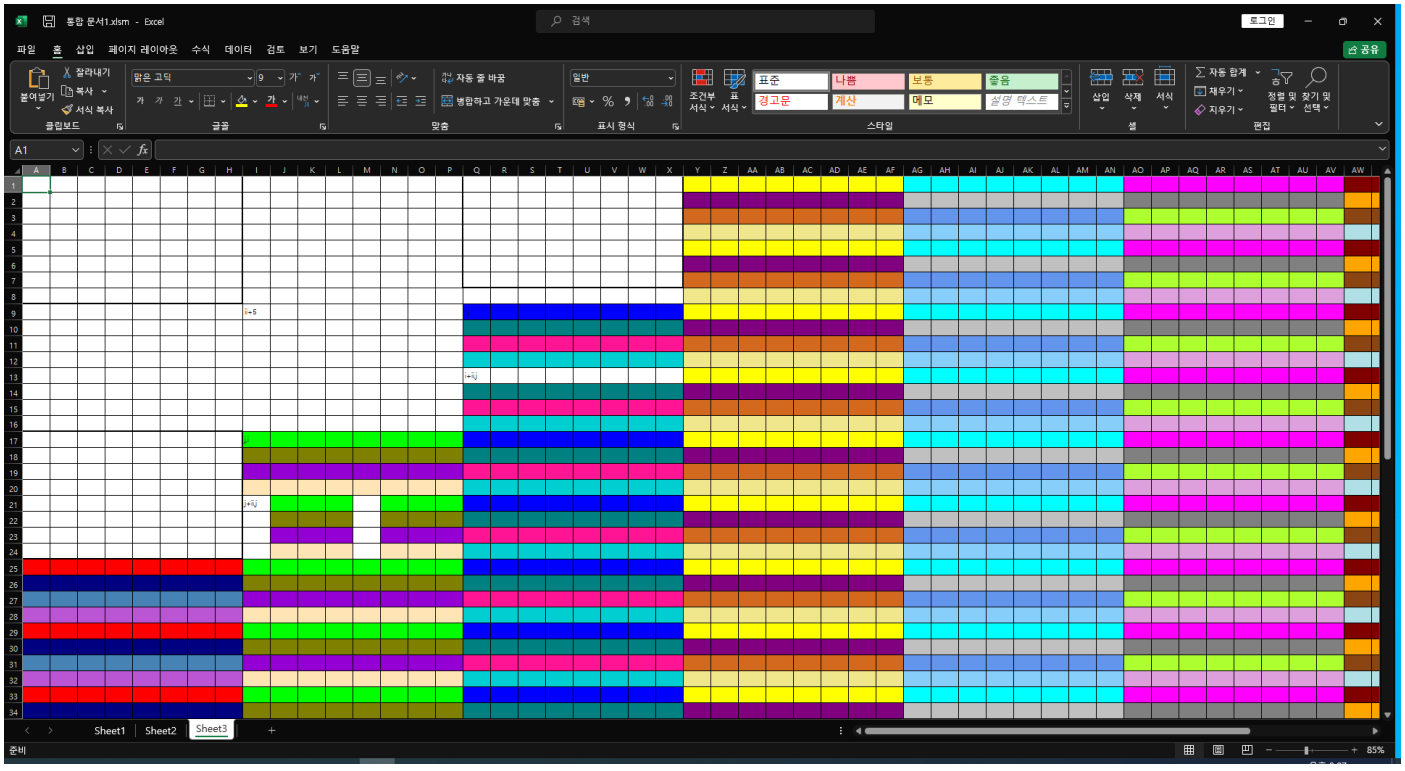
본 과제의 요구 상, M 은 그저 access 두 번, L과 S는 access 한 번 하는 것으로 생각하여도 좋다. 따라서 위와 같이 switch 문을 구성하였다. M일 경우 fall-throw로 access를 두 번 하도록 구현하였다. access_cache 함수의 경우 간단하게 hit 일 경우 처리, miss 일 경우(아직 캐시가 다 차지 않았을 때) 처리, eviction 일 때 처리로 나누어 구현하였다. linked list 로 캐시를 만들었기에 그저 추가, 삭제를 신경써 주는 것으로 LRU policy를 따르도록 구현할 수 있었다.

3. Part B

본 과제 해결을 위하여 캐시의 구조에 대하여 정확히 파악해야 한다. 캐시의 크기가 한정되어 있기에 특히 64*64 상황에서 고려할 점이 많았으나, 캐시의 동작을 직관적으로 보기 힘든 문제가 있었다. 이를 위해 다음과 같이 엑셀 파일을 준비했다. Chat GPT 가 의도대로 잘 만들어 주었으며, 이는 결론에 링크를 달아놓았다.



이를 이용하면 다음과 같이 캐시의 상황을 직관적으로 그려가며 문제를 해결할 수 있다.



그리고 채점 시 편의를 위하여 다음과 같은 shell script를 작성하여 이용하였다.

```
#!/bin/bash
make clean && make && clear && ./test-trans -N 64 -M 64
```

그때그때 푸는 문제에 따라 조금씩 만져주긴 했지만, 이렇게 사용하니 편했다.

본론으로 돌아가, 본 과제 해결 시에 switch 문을 이용해, N 에 따라 case 를 나누어 세 가지 코드로 작성하였다. 먼저 N=32 인 경우이다.

```
for (int i = 0; i < 32; i += 8)
{
    for (int j = 0; j < 32; j += 8)
    {
        if (i != j)
        {
            for (int ii = 0; ii < 8; ii++)
                for (int jj = 0; jj < 8; jj++)
                {
                    B[j + jj][i + ii] = A[i + ii][j + jj];
                }
        }
        else
        {
            for (int ii = 0; ii < 8; ii++)
            {
                for (int jj = 0; jj < 8; jj++)
                {
                    if (ii != jj)
                    {
                        B[j + jj][i + ii] = A[i + ii][j + jj];
                    }
                }
                B[j + ii][i + ii] = A[i + ii][j + ii];
            }
        }
    }
}
```

코드가 예쁘지 못한 것 같지만, 더 나은 방법을 생각할 수 없었다. size=8인 block 을 잡아 처리하도록 코드를 구상하였다. diagonal 부분과 그 외 부분을 따로 처리해야 한다. diagonal 부분의 경우 같은 cache line을 공유하게 되기에, 일반적인 방법으로 옮긴다면 불필요한 cache miss가 다수 발생한다. 따라서 diagonal 에 대해서는 cache miss 가 적게 나도록 세심하게 원소를 옮겨주고, 나머지 케이스에 대하여는 평범하게 처리해 준다.

N=64 인 경우를 보기 앞서, N=67 인 경우를 먼저 보도록 하겠다.

```
for (int i = 0; i < 67; i += 16)
{
    for (int j = 0; j < 61; j += 16)
    {
        for (int ii = 0; ii < 16 && i + ii < 67; ii++)
        {
            for (int jj = 0; jj < 16 && j + jj < 61; jj++)
            {
                B[j + jj][i + ii] = A[i + ii][j + jj];
            }
        }
    }
}
```

matrix가 8의 배수로 떨어지지 않는 특이한 상황이다. 별다른 고려 없이 block을 잘 잡아 주기만 하여도 최대 점수를 받을 수 있었다. block size의 경우 4, 8, 16 등을 시도해 보았는데, 16일 때에는 최대 점수를 받을 수 있었고, 그 외의 경우는 miss rate 가 16일 때보다 높았다.

다음은 N=64 인 경우이다. 사실 이 상황에 대한 분석 때문에 엑셀 파일을 만들어 자주 보았다. 작성한 코드는 다음과 같다.

```
        for (int i = 0; i < 64; i += 8)
        {
            for (int j = 0; j < 64; j += 8)
            {
                temp = 56;
                if (i == 56 || j == 56)
                {
                    temp = 0;
                }
                if (j == 0)
                {
                    temp = 8;
                }
                if (i != j)
                {
                    for (int jj = 0; jj < 8; jj++)
                    {
                        B[temp + 4][temp + jj] = A[i + 4][j + jj];
                        B[temp + 5][temp + jj] = A[i + 5][j + jj];
                        B[temp + 6][temp + jj] = A[i + 6][j + jj];
                        B[temp + 7][temp + jj] = A[i + 7][j + jj];
                    }
                    for (int jj = 0; jj < 8; jj++)
                    {
                        B[j + jj][i + 0] = A[i + 0][j + jj];
                        B[j + jj][i + 1] = A[i + 1][j + jj];
                        B[j + jj][i + 2] = A[i + 2][j + jj];
                        B[j + jj][i + 3] = A[i + 3][j + jj];
                        B[j + jj][i + 4] = B[temp + 4][temp + jj];
                        B[j + jj][i + 5] = B[temp + 5][temp + jj];
                        B[j + jj][i + 6] = B[temp + 6][temp + jj];
                        B[j + jj][i + 7] = B[temp + 7][temp + jj];
                    }
                }
            }
        }
```

64*64 의 경우 이전과 같이 일반적인 방법으로 8*8 blocking 을 할 시 cache miss late 가 상당히 커지게 된다. 이유는 위에 첨부해준 엑셀 그림에서 볼 수 있듯, r 번째 row 와 r+4 번째 row 가 같은 cache line을 공유하기 때문이다. 따라서 이외에 다른 방법을 통한 최적화를 해 주어야 한다. diagonal 을 제외하고 먼저 처리하기에 diagonal 이 비어있음을 이용, 먼저 옮길 데이터 중 아래 8*4 만큼을 diagonal 에 복사한 뒤(이 때 복사되는 위치는 다른 cache line에 위치해야 한다.) diagonal 과 원래 A 에 있는 data를 이용하여 B 로 전치해서 옮김으로써 cache late를 줄일 수

있었다.

```
        for (int i = 0; i < 64; i += 8)
        {
            for (int j = 0; j < 64; j += 8)
            {
                if (i == j)
                {
                    for (int ii = 0; ii < 8; ii++)
                    {
                        a0 = A[i + ii][j + 0];
                        a1 = A[i + ii][j + 1];
                        a2 = A[i + ii][j + 2];
                        a3 = A[i + ii][j + 3];
                        a4 = A[i + ii][j + 4];
                        a5 = A[i + ii][j + 5];
                        a6 = A[i + ii][j + 6];
                        a7 = A[i + ii][j + 7];
                        B[j + 0][i + ii] = a0;
                        B[j + 1][i + ii] = a1;
                        B[j + 2][i + ii] = a2;
                        B[j + 3][i + ii] = a3;
                        B[j + 4][i + ii] = a4;
                        B[j + 5][i + ii] = a5;
                        B[j + 6][i + ii] = a6;
                        B[j + 7][i + ii] = a7;
                    }
                }
            }
        }
```

diagonal의 경우 temp val 을 통하여 조금이나마 cache miss를 줄이고자 노력하였다.

4. 결론

본 과제를 통하여 캐시의 동작 원리를 자세히 이해할 수 있었다. 다만 행렬 전치 알고리즘의 경우 특수한 상황에 대하여만 최적화를 하게 되어 아쉬움이 있다.

https://chatgpt.com/share/67470b0b-c608-8004-81e5-14de8d948643

언급한 대로, 캐시의 구조를 직관적으로 파악하기 위해 GPT 에게 적당한 방안을 요청하였다.