[Project-2] 3D convolution Report

201920739 소프트웨어학과 박지윤
201920784 소프트웨어학과 강윤지

1. Describe how to design your 3D convolution

1) single-thread(AVX)

AVX는 한 번에 8개의 원소를 불러와서 8개의 원소에 대해 같은 instruction을 수행한다. 그렇기 때문에 AVX의 성질을 최대한 살려주기 위하여 일반적으로 output의 한 원소를 구해주는 convolution 방식과는 다른 방식을 택하였다.

우리가 수행해야 할 convolution은 3차원이지만, 간단하게 생각하기 위하여 1차원에 대한 convolution 연산을 가정해보자. 1~16까지의 원소가 (1 *16)의 size를 가진 input으로 주어지고, a~e까지의 원소가 (1 * 5)의 size를 가진 kernel로 주어지고, output matrix 또한 (1*16)의 size로 구한다고 하였을 때, 이 convolution 연산의 결과는 [그림 1] 과 같다.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
			1a	2a	3a	4a	5a	6a	7a	8a	9a	10a	11a	12a	13a	14a	15a	16a
		1b	2b	3b	4b	5b	6b	7b	8b	9b	10b	11b	12b	13b	14b	15b	16b	
	1c	2c	3c	4c	5c	6c	7c	8c	9c	10c	11c	12c	13c	14c	15c	16c		
1d	2d	3d	4d	5d	6d	7d	8d	9d	10d	11d	12d	13d	14d	15d	16d			
1e	2e	3e	4e	5e	6e	7e	8e	9e	10e	11e	12e	13e	14e	15e	16e			

그림 1 (1-16) input과 (a-e) 원소의 convolution 연산의 결과

이를 AVX처럼 8개의 원소를 가져온다고 생각했을 때, [그림2] 와 같이 나타낼 수 있다.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
			1a	2a	3a	4a	5a	6a	7a	8a	9a	10a	11a	12a	13a	14a	15a	16a
	1b	2b	3b	4b	5b	6b	7b	8b	9b	10b	11b	12b	13b	14b	15b	16b		
1c	2c	3c	4c	5c	6c	7c	8c	9c	10c	11c	12c	13c	14c	15c	16c			
1d	2d	3d	4d	5d	6d	7d	8d	9d	10d	11d	12d	13d	14d	15d	16d			
1e	2e	3e	4e	5e	6e	7e	8e	9e	10e	11e	12e	13e	14e	15e	16e			

그림 2 convolution의 결과를 8개씩 쪼갠 모습

이때, input에서 맨 앞부터 8개의 원소를 가져와서 kernel의 각 원소와 곱하여 더할 때, [그림3]의 노란색 원소에서 볼 수 있는 것처럼 load한 위치가 아닌, 다른 위치에서 8개를 load했을 때 구할 수 있는 값이라는 문제가 생긴다.

index			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
					1a	2a	3a	4a	5a	6a	7a	8a	9a	10a	11a	12a	13a	14a	15a	16a
				1b	2b	3b	4b	5b	6b	7b	8b	9b	10b	11b	12b	13b	14b	15b	16b	
			1c	2c	3c	4c	5c	6c	7c	8c	9c	10c	11c	12c	13c	14c	15c	16c		
		1d	2d	3d	4d	5d	6d	7d	8d	9d	10d	11d	12d	13d	14d	15d	16d			
	1e	2e	3e	4e	5e	6e	7e	8e	9e	10e	11e	12e	13e	14e	15e	16e				

그림 3 convolution의 결과를 8개씩 쪼갰을 때, 다른 위치에서 load해야 하는 원소를 표시한 모습

[그림 3]에서 확인할 수 있듯, 8개의 원소를 가져와서 왼쪽 혹은 오른쪽으로 shift를 해준 후 바로 더해주면, 노란색 원소의 값은 누적해서 더해지지 않는다는 문제점이 있다. 따라서 8개의 원소씩 가져오긴 하되, input과 kernel을 연산한 값을 x축 길이만큼 선언된 배열에 차례대로 저장한다.

```
/* input에서 x축 기준 8개의 elments를 load한다 */
num = _mm256_load_ps(&input[i * imat_x * imat_y + j * imat_x + k]);

/* load한 원소와 kernel의 각 원소와 연산한 후, 연산한 값을 imat_x의 크기를 가진 배열에 맞는 위치에 넣어준다 */
for (int p = 0; p < kernel_size * kernel_size * kernel_size; p++) {
    result[p] = _mm256_mul_ps(ker_arr[p], num);
    memcpy(&line[p * imat_x + k], &result[p], sizeof(float) * 8);
}
```

그림 4 input을 8개씩 가져와서 연산 후, 이를 저장하는 코드

x축 길이만큼 input에서 원소를 load하여 kernel과 연산을 해 저장이 완료되었다면, 현재 kernel 원소의 index에 따라 오른쪽 혹은 왼쪽으로 shift해주고, 빈 칸은 0으로 초기화해주었다. 이때, 단순히 for문을 돌려서 원소 하나하나씩 shift를 해주는 것이 아니라, pointer를 사용하여 메모리의 위치를 변경해주는 식으로 코드를 구현하여 성능상 이점을 얻을 수 있도록 구현하였다.

```
/* imat_x의 크기를 가진 배열에서 kernel의 element의 위치에 따라 왼쪽 혹은 오른쪽으로 shift 한다 */
for (int p = (kernel_size / 2); p < kernel_size * kernel_size * kernel_size; p += kernel_size) {
    for (int t = 1; t <= padding; t++) {
        memmove(&line[(p - t)*imat_x + t], &line[(p - t)*imat_x], sizeof(float) * (imat_x - t));
        memmove(&line[(p + t)*imat_x], &line[(p + t)*imat_x + t], sizeof(float) * (imat_x - t));
        for (int s = 0; s < t; s++) {
            line[(p - t)*imat_x + s] = 0;
            line[(p + t)*imat_x + imat_x - 1 - s] = 0;
        }
    }
}
```

그림 5 [그림 2]처럼 원소를 kernel의 index 위치에 맞게 왼쪽/오른쪽으로 shift해주는 코드

shift작업이 완료되었다면, 이제는 누적해서 더해줄 차례이다. 이때, 단순히 convolution을 해주는 것이 아니라, input size와 output size가 같아야 하기 때문에 padding을 추가해주었다. 또한, 일반적으로 생각하는 convolution의 과정처럼 input의 한 부분을 가져와서 kernel과 convolution을 구하여 한 원소를 구하는 것이 아니라, input의 x축 길이만큼 가져와서 이를 kernel의 각 원소와 곱한 후, 이를 output matrix에 kernel 큐브 크기만큼 더해주는 식으로 하였다. 이때, 바깥쪽 input을 가져와서 연산을 해주고, output에 누적을 해주는 경우 kernel 큐브

크기만큼 더하게 되면 output matrix의 사이즈를 벗어나는 곳에 접근하게 된다. 따라서 이를 방지해주기 위해 누적합을 해주려는 위치가 output matrix의 사이즈를 벗어난다면 다음 현재 index는 건너뛰고, 다음 index를 수행해주도록 하였다. 이때, 누적합을 해주는 과정은 다른 데이터에 대해서 같은 instruction을 수행하는 것이기 때문에 AVX를 사용하여 좀 더 빠르게 수행될 수 있도록 하였다.

```
/* output에 값을 더한다 */
for (int k = 0; k < imat_x; k += 8) {
    for (int p = 0; p < kernel_size * kernel_size * kernel_size; p++) {
        /* output에 넣을 위치가 matrix 범위를 넘어가는지 계산하기 위한 변수 dist, updown */
        dist = kernel_size / 2 - p / (kernel_size * kernel_size);
        updown = kernel_size / 2 - (p % (kernel_size * kernel_size)) / kernel_size;

        /* output에 넣을 위치가 matrix 범위를 넘어간다면 continue */
        if (imat_y - j <= padding && updown > 0 && updown >= (imat_y - j)) { continue; }
        if (i + dist < 0 || i + dist >= imat_z || j + updown < 0 || j + updown > imat_y) continue;

        temp = _mm256_load_ps(&line[p * imat_x + k]);

        /* 들어가야 할 위치에 kernel과 input과 연산한 값을 더하여 넣어준다 */
        temp2 = _mm256_load_ps(&single_output[(i+dist)*imat_x*imat_y + (j+updown)*imat_x + k]);
        res = _mm256_add_ps(temp, temp2);
        memcpy(&single_output[(i+dist)*imat_x*imat_y + (j+updown)*imat_x + k], &res, sizeof(__m256));
    }
}
```

그림 6 계산한 결과를 output matrix에 누적합해주는 코드

2) multi-thread(w/AVX)

single-thread의 알고리즘을 기반으로 multi-thread를 구현하였다. Single-thread에서 input matrix에서 x축 -> y축-> z축 순으로 원소를 가져오면서 연산을 하기 때문에, multi-thread에서 z축 기준으로 thread가 해야 할 연산의 범위를 나누어 주었다.

이때, output matrix에 지속적으로 누적해서 값을 더해주어야 하는데, 기존의 single-thread 방식대로 하면 누적해서 더해주기 위해 값을 불러오는 과정에서 여러 thread가 동시에 접근하여 원하는 결과가 나오지 않을 수 있다. 처음에는 이를 방지하기 위해 값을 load할 때 lock을 걸어주었으나, lock을 거는 과정에서 발생하는 overhead가 multi threading으로 줄어든 시간보다 더 커서 오히려 single thread 방식보다 수행 시간이 증가하였다.

따라서, 각 thread들이 할당받은 연산을 수행하고, 이를 누적해서 더해줄 때 동시에 load하는 것을 막아주기 위해 output matrix와 size가 똑같은 multi_temp matrix를 kernel_size만큼 생성해주었다. 그리고 각 thread들이 (자신의 id % kernel_size)에 해당하는 multi_temp matrix에 연산한 결과들을 누적해서 더하도록 하여 thread들이 같은 위치에 접근하는 것을 최소화해주었다. thread들이 join하면 multi_temp에는 thread들이 각자 담당한 범위만큼 연산한 결과들이 저장되어 있으므로, 이를 avx를 사용하여 더해주어 최종적으로 3d convolution한 결과를 구해주었다.

3) GPU

기존의 CUDA 2D convolution을 구현한 내용을 참고하여 kernel 내에서 depth에 대한 처리를 추가해주었다.

```
dim3 dimGrid(ceil((float)imat_x/TILE_SIZE), ceil((float)imat_y/TILE_SIZE), ceil((float)imat_z/TILE_SIZE));
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
```

```
int row_o = blockIdx.y * TILE_SIZE + ty;
```

처음 Gird의 block갯수를 정해줄 때 BLOCk_SIZE로 나눠주었다. 하지만 kernel 안에서는 한 thread당 접근하는 범위가 block 단위가 아니라 tile size 단위이기 때문에 모든 element에 대한 접근이 이루어 지지 않는 문제가 발생하였다. 이를 해결해주기 위해 BLOCK_SIZE대신 TILE_SIZE로 block의 수를 정해주게 되었다.

```
#define TILE_SIZE 4
```

TILE_SIZE는 4로 설정하였다. 한 block 당 들어가는 최대 thread 수는 1024개인데 만약 tile의 size가 8이 되어버린다면, BLOCK_SIZE = TILE_SIZE + KERNEL_SIZE - 1 = 8 + 5 -1 = 12이고 한 block당 들어가는 thread의 수는 다음과 같이 결정되니

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
```

12 * 12 * 12 > 1024 가 되어 문제가 발생한다. 그렇기에 8보다 작으면서 편의상 2의 배수 중 가장 큰 4를 선택하였다.

2. performance comparison for single-thread (AVX), multi-thread (w/ AVX), GPU

1) single-thread(AVX)

single-thread에서 각 test의 시간을 5번 측정하여 낸 평균값은 아래 표와 같다.

[표1] single thread에서 각 test의 수행 시간

	test1	test2	test3	test4	test5
input_size	64*64*64	64*64*64	128*128*128	32*64*64	32*64*64
kernel_size	3*3*3	3*3*3	5*5*5	5*5*5	3*3*3
시간(s)	0.043356	0.043580	1.639774	0.099092	0.011944

각 test들 간의 시간을 통해 연산량이 많아질수록 시간이 늘어남을 알 수 있다. Test3의 경우, input size도 128*128*128로 제일 크고, kernel size도 5*5*5로 convolution 연산을 해야 할 수도 많기 때문에 시간이 제일 오래걸렸다. 같은 input size와 kernel size를 가진 test1과 test2의 경우에는 연산량이 같기 때문에 수행 시간이 비슷하게 걸렸다. 같은 input size이지만 kernel size가 작은 test5의 경우, 연산량이 test4에 비해 적기 때문에 수행 시간이 적게 걸린 것을 확인할 수 있다. 마찬가지의 이유로, kernel size는 같지만 input size가 작은 test5의 경우, 연산량이 test1이나 test2에 비해 적기 때문에 수행 시간이 적게 걸린 것을 확인할 수 있다.

2) multi-thread(w/AVX)

multi-thread에서 thread의 수를 2, 4, 8, 16, 32로 바꾸었을 때, 각 test의 시간을 5번 측정하여 낸 평균값은 아래 표와 같다.

[표2] thread의 수에 따른 각 test의 수행 시간

thread 수	single	2	4	8	16	32
test1	0.043356	0.169893	0.084052	0.043524	0.023993	0.015965
test2	0.043580	0.168170	0.083936	0.043673	0.024500	0.015729
test3	1.639774	1.367621	0.745663	0.404728	0.265868	0.156257
test4	0.099092	0.128673	0.066991	0.037328	0.024607	0.017671
test5	0.011944	0.074409	0.037922	0.019506	0.010886	0.007685

single-thread와 비교하였을 때, thread의 개수가 2개일 때는 test3를 제외한 나머지에서 single-thread가 더 빠른 것을 확인할 수 있다. 이는 test1, test2, test4, test5는 multithreading으로 줄어드는 연산 수행 시간보다, thread를 생성하는데 드는 overhead가 더 커서 그런 것으로 보인다. 다만, test3는 thread를 2개만 생성하였음에도 single thread보다 빨랐는데, test3의 경우 연산량이 매우 많아 thread를 생성하는데 드는 overhead보다 multi threading으로 얻는 speedup이 더 이점이 컸던 것으로 보인다.

thread의 개수가 4일 때, test3 다음으로 연산량이 많은 test4도 single thread보다 빨라졌는데 위와 마찬가지의 이유로, 연산량이 test1, test2, test5보다 많아서 multi threading이 더 효과적이었던 것으로 보인다.

thread의 개수가 8일 때는 모든 test가 수행 시간이 single thread보다 빠르거나 비슷하였다. 이후 thread의 개수를 늘려갈수록, 수행 속도가 빨라지는 것을 확인할 수 있다. z축을 기준으로 thread가 수행해야 할 연산의 범위를 나누었기 때문에, 32개까지밖에 비교를 할 수 없었으나, 만약 z축의 길이가 32개보다 더 커서 더 많은 thread를 생성할 수 있다면, thread의 개수가 많아질수록 수행 시간이 짧아지다가 어느 시점부터는 thread를 생성하는 overhead가 multi threading으로 줄어드는 연산 시간보다 커져서 수행 시간이 길어질 것으로 예상된다.

3) GPU

GPU에서 각 test의 시간을 5번 측정하여 낸 평균값은 아래 표와 같다.

[표3] GPU에서 CUDA Programming을 사용하였을 때, 각 test의 수행 시간

	test1	test2	test3	test4	test5
input_size	64*64*64	64*64*64	128*128*128	32*64*64	32*64*64
kernel_size	3*3*3	3*3*3	5*5*5	5*5*5	3*3*3
시간(s)	0.001355	0.001354	0.024552	0.001594	0.000374

상대적으로 적은 input size를 가지는 다른 test들과 달리 128*128*128의 input size를 가지는 test3을 비교해 보았을 때 연산량이 늘어날수록 완료할 때까지 걸리는 시간이 늘어나는것을 확인할 수 있다. 64*64*64인 test1,2와 32*64*64인 test4,5의 차이도 같은 맥락에서 비교할 수 있으며 tset1과 test2는 input size 및 kernel size가 동일하므로 걸리는 시간의 차이는 거의 없는것을 확인할 수 있다. test4, 5는 같은 input size이지만 kernel의 size가 3*3*3과 5*5*5로 다르다. test5보다는 test4가 연산량이 더 많기에 처리 속도도 더 오래걸리는 것을 확인할 수 있다.