[Project-2] 3D Convolution

소프트웨어학과 201921085 곽수정 소프트웨어학과 201720707 나용성

1. Describe how to design your 3D convolution

지난과제에서 배운 2D convolution에서 z축이 더해진 3D convolution을 수행할 수 있는 코드를 작성하기 위해서 필요한 변경점을 먼저 생각해보았고 다음과 같은 추가사항이 필요했다.

1) input, output, kernel matrix를 3차원 매트릭스로 사용할 수 있어야 한다.
   => 이를 위해 input, output , kernel matrix의 크기를 height, width의 크기의 곱 만큼 할당하고 height 보다 작은 값 i와 width보다 작은 값 j 두 값을 이용해 2차원으로 접근하던 것을 3차원으로 활용할 수 있도록 z축의 길이를 의미하는 depth값을 추가하여 depth, height, width의 크기의 곱 만큼 matrix의 크기를 할당하고 depth이내의 값을 추가하여 i, j, k 세 값을 이용해 3차원방식으로 index를 활용할 수 있어야 한다.

2) 2차원으로 사용하던 cuda thread block과 grid를 3차원으로 확장시킨다.
   => 3차원 gird는 x, y, z축에 대해 2차원 grid에서와 같은 방식으로 output matrix의 depth, height, width를 tile size로 나눈 값들을 x, y, z축의 길이로 갖고, 각 thread block은 x, y, z축의 크기는 모두 block size로 같게 구성하는데 이때 block size는 tile size + kernel size - 1이 되도록 한다. 이렇게 구성한 이유는 각 thread block은 output matrix에서 x, y, z축 길이가 tile size인 만큼의 구간에 대해 input matrix와 kernel matrix를 이용해 3d convolution을 수행하도록 하여야 하기 때문에 output matrix의 각 축을 tile size로 나눈 만큼의 thread block을 갖는 grid를 생성했고 thread block이 x, y, z축의 길이가 tile size 만큼의 output을 구하려 할 때 output을 계산하는 구간에서 각 가장자리 구간은 해당 구간의 바깥으로 (kernel - 1)/2만큼의 구간의 데이터가 더 필요하므로 요구사항에서 block shared memory를 활용한 tiling기법을 만족하기 위해 계산하려는 구간 바깥의 데이터도 block shared memory에 저장하는 cuda thread가 더 필요하므로 block size는 tile size + kernel size - 1이 된다.

```
dim3 dimGrid(ceil((float)o_width/TILE_SIZE), ceil((float)o_height/TILE_SIZE), ceil((float)o_depth/TILE_SIZE));
int block_size = TILE_SIZE + (k_size - 1);
dim3 dimBlock(block_size, block_size, block_size);
```

그림1. grid와 thread block의 크기를 위에서 설명한 값으로 설정하는 코드

추가로 저번 과제에선 kernel size가 define에 정의되어 static하게 정의되어 있어 block shared memory의 크기를 미리 지정해줄 수 있었지만 이번 과제에선 입력된 파일 경로에 따라 kernel의 크기가 달라지기 때문에 block size가 정적으로 정해지지 않아 block shared memory의 크기를 kernel함수를 실행할 때 파라미터로 입력하여 block shared memory의 크기를 유동적으로 할당할 수 있도록 했다.

```
Convolution<<<dimGrid, dimBlock, sizeof(float) * block_size * block_size * block_size>>>
```

그림2. kernel함수를 실행할 때 tiling기법을 위해
block shared memory의 크기를 동적으로 할당하는 코드

3) 추가적으로 저번 과제와 달리 사용하는 파일에 따라 kernel의 크기가 달라지는 이번 과제에서 constant memory에 kernel을 저장할 수 있기 위해서 Mc배열을 충분히 큰 크기로 미리 선언해두고 kernel을 kernel.txt에서 읽어낸 다음 kernel의 크기 만큼만 Mc에 cudaMemcpyToSymbol api로 load시킬 수 있도록 설계했다.

```
fscanf(kernel_file, "%d", &k_size);
M = (float*)malloc(k_size * k_size * k_size * sizeof(float));
for (int i = 0; i < k_size * k_size * k_size; i++)
        fscanf(kernel_file, "%f", &M[i]);
cudaMemcpyToSymbol(Mc, M, sizeof(float) * k_size * k_size * k_size);
```

그림 3. kernel.txt파일에서 kernel size를 확인하고 프로세스로 읽어온 다음
gpu의 constant memory에 load하는 파트의 코드

이 때 충분히 큰 크기를 정하기 위해 서버gpu의 스펙을 확인했다. 이번 과제에서 kernel의 최대 크기를 제한한 요소는 thread block당 최대 thread의 개수였다. 이번 과제에서 thread block의 크기는 block size의 세제곱만큼이다. 그리고 서버 gpu의 thread block당 최대 thread의 개수는 1024개였기 때문에 이를 초과하는 크기의 thread block을 생성하려 하면 프로그램이 제대로 동작하지 못할 것이다.

```
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
```

그림 4. deviceQuery를 이용해 Maximum number of threads per block을 확인했다.

이 때 block size의 세제곱이 1024를 초과하지 않는 최대 block size는 10이 된다. (10^3 = 1000 < 1024, 11^3 = 1331 > 1024) block size는 tile size + kernel size - 1이므로 tile size가 최소인 1일로 설정할 때 kernel size는 block size와 같으므로 10까지 사용할

수 있게된다. 이번 과제의 kernel은 3차원 kernel이므로 kernel size가 10일 kernel의 크기는 1000이 되므로 Mc를 1000개의 float까지 load할 수 있는 배열로 선언해두었다.

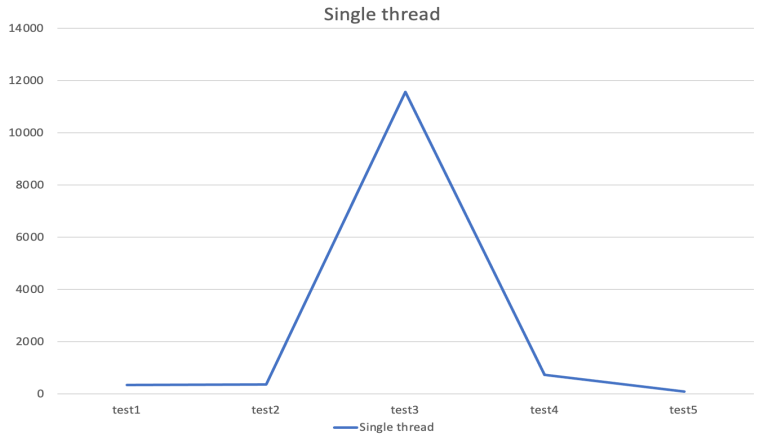
```
#define MAX_KERNEL 1000
#define TILE_SIZE 8

__constant__ float Mc[MAX_KERNEL];
```

그림 5. constant memory를 사용하는 변수 Mc의 크기가 1000으로 충분히 큰 크기로 미리 설정된 모습

2. **Performance comparison for single thread (AVX), multiple threads (w/AVX), GPU**

Single thread (AVX), multiple threads (AVX), GPU에서의 성능 측정을 위해 __rdtsc 함수와 cudaEventElapsedTime API를 사용하였고, test 결과마다 10번씩 측정한 시간(ms)의 평균을 구하여 기록하였다.

- Single thread (AVX)



	Single thread
test1	333.730824
test2	353.822223
test3	11558.5585
test4	725.728003
test5	83.455834

그림6. test1~5까지의 single thread에서의 test 결과

Single thread가 multiple threads와 GPU의 모든 test 결과와 비교했을 때 가장 성능이 좋지 않았다. 특히 input의 크기가 가장 큰 test3에서 많은 시간이 걸리는 것을 볼 수 있었다.

- Multiple threads (AVX)

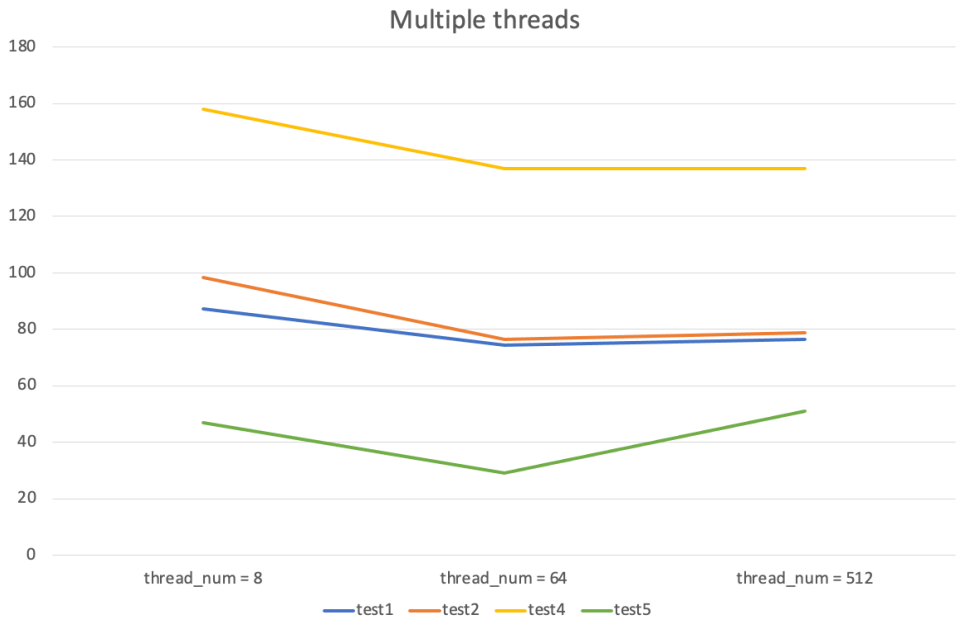


그림 7. test1, 2, 4, 5의 multiple threads에서의 test 결과

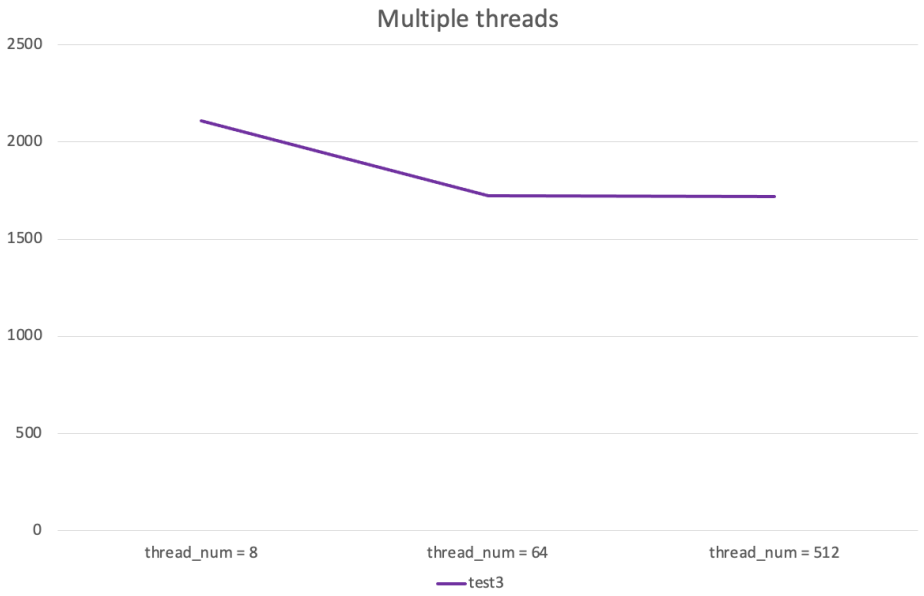


그림 8. test3의 multiple threads에서의 test 결과

thread_num	test1	test2	test3	test4	test5
8	87.318825	98.48304	2107.1067	157.835125	46.858926
64	74.342957	76.539036	1724.78898	136.963252	29.259284
512	76.506407	78.891292	1720.09071	137.08379	50.911508

그림 9. multiple threads에서 thread 개수에 따른 각 test에서의 실행시간 표

thread_num	test1	test2	test3	test4	test5
8	3.82198024	3.59272239	5.48551172	4.59801329	1.78100185
64	4.48907116	4.62276822	6.70143342	5.29870598	2.85228559
512	4.3621291	4.48493381	6.71973776	5.29404682	1.6392332

그림 10. multiple threads에서의 실행시간을 single thread에서의 실행시간과 비교하여 성능향상률을 배수로 표시한 표

Multiple threads에서는 각 test마다 thread_num = 8, TILE_DIVIDE = 2와 thread_num = 64, TILE_DIVIDE = 4, 그리고 thread_num = 512, TILE_DIVIDE = 8로 thread 수를 늘려가며 측정하였다. 모든 test에서 thread가 64개일 때는 thread 수가 8일 때보다 성능이 좋아졌지만, thread가 512일 때는 오히려 시간이 오래 걸렸다. 이는 각각의 thread가 하는 일은 줄어들었지만 thread를 512개 생성하고, context switch 하는 등의 오버헤드가 더 크기 때문에 성능이 좋지 않게 나온 것으로 보인다.

Multiple threads의 성능을 single threads와 비교해봤을 때 일의 크기가 가장 큰 test3에서 최대 약 6.71배의 성능향상이 있었고 일의 크기가 가장 작은 test5에서 최대 약 2.85배의 성능향상이 있었다. 즉 일의 양이 클 수록 thread의 개수가 많아도 각각의 thread가 처리할 일의 양이 충분히 많으므로 성능향상에 유리하다는 것을 알 수 있었다. 이는 test 1, 2, 4, 5에서는 thread의 개수가 64개일 때가 512개일 때보다 성능이 좋았지만 일의 크기가 훨씬 큰 test3에선 4ms의 차이지만 성능이 앞서는 점을 통해 다시 한번 알 수 있었다.

- GPU

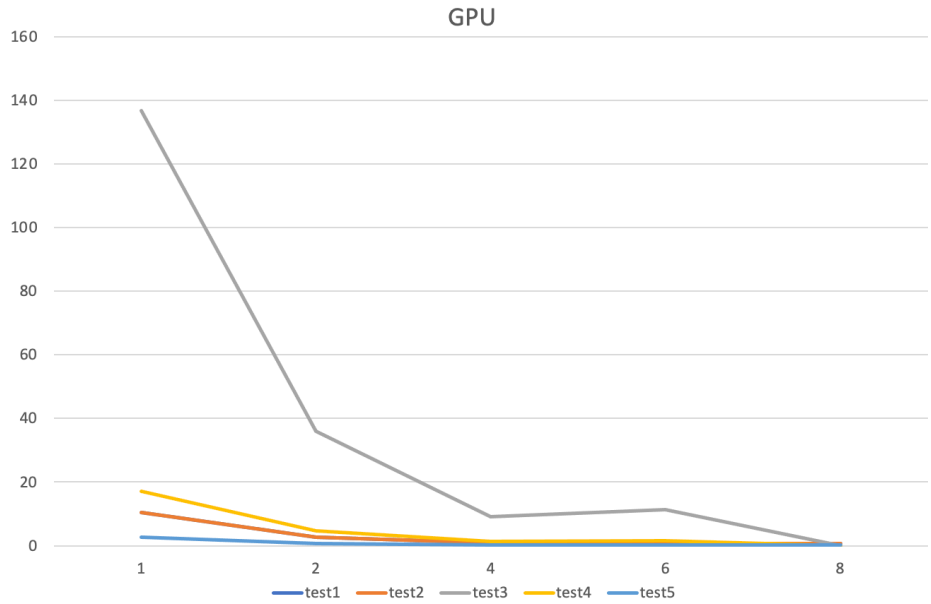


그림11. test1~5까지의 gpu에서의 test 결과

GPU	test1	test2	test3	test4	test5
1	10.479808	10.477696	136.777054	17.176064	2.653408
2	2.697568	2.698208	36.038303	4.573088	0.695104
4	0.91584	0.909728	9.208576	1.260608	0.259168
6	0.779264	0.772736	11.444096	1.615968	0.259232
8	0.601792	0.613056	X	X	0.181408

그림 12. multiple thread에서 TILE_SIZE에 따른 각 test에서의 실행시간 표

GPU	test1	test2	test3	test4	test5
1	31.8451277	33.7690865	84.5065613	42.252288	31.4523187
2	123.715444	131.132301	320.729822	158.695394	120.062371
4	364.398611	388.931882	1255.19499	575.696809	322.014423
6	428.264136	457.882411	1010.00188	449.098004	321.934923
8	554.561749	577.145029			460.044948

그림 13. gpu에서의 실행시간을 single thread에서의 실행시간과 비교하여 성능향상률을 배수로 표시한 표

GPU에서는 tile size를 1, 2, 4, 6, 8로 늘려가며 시간을 측정하였다. tile size가 증가할수록 성능이 좋아지는 경향을 보였지만, tile size가 8일 때 test3와 test4에서는 제대로 동작하지 않았으며, TILE_SIZE가 더 커졌을 때 모든 test가 동작하지 않는 것을 볼 수 있었다. 이는 앞서 말했듯이, block size가 gpu의 사양 중 maximum number of threads per block에 의해 최대 10으로 제한되기 때문에 test3, 4에선 kernel size가 5기 때문에 tile size가 8이 되면 block size가 12가 되어 thread block 하나의 thread개수가 maximum number of threads per block을 초과하기 때문에 정상적으로 작동할 수 없다.

single thread에서와 실행시간을 비교해본 결과 가장 성능향상이 적은 test5에서도 tile size가 1일 때 31배, tile size가 8일 때 460배의 성능향상이 있었으며 일의 크기가 가장 컸던 test3에서는 tile size가 1일 때 84배 tile size가 4일 때 1255배의 성능향상을 보여 multiple thread에서의 성능향상과 비교해도 비약적으로 성능이 많이 향상됨을 확인했다. 이는 multiple thread에서는 sw thread를 생성해 gpu와 비교해 적은 양의 코어에서 스케줄링하며 하나의 스레드가 여러개의 output을 계산해내야 하는 cpu에서의 pthread를 이용한 multiple threading 방식과는 달리 gpu에서는 하나의 cuda thread가 하나의 output을 계산하며 cpu에서보다 많은 양의 코어와 thread가 동시에 parallel하게 SPMD방식으로 동작하기 때문일 것으로 보인다.