

AVX, Pthread, CUDA를 이용한 3D CONV 구현

소프트웨어학과 201521032 한태희
소프트웨어학과 201720733 신승현

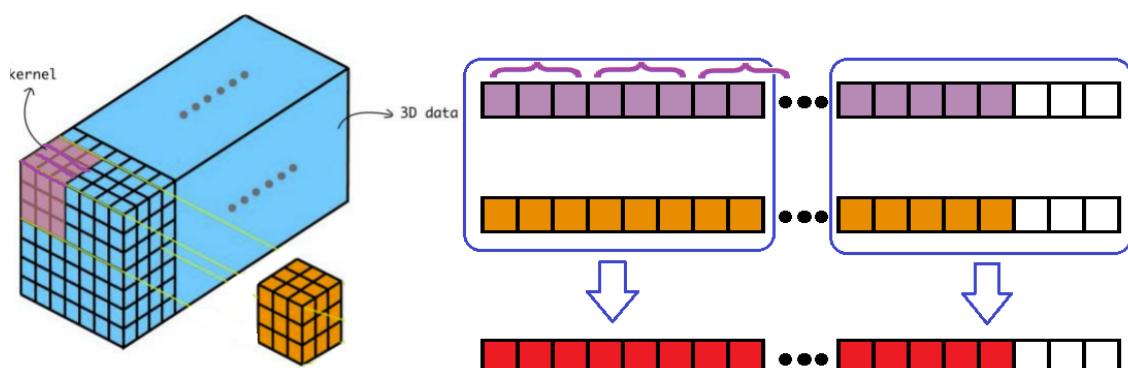
1. 개요

이전 실습에서 제작하였던 2D Convolution을 cuda를 이용하여 GPU에서 연산하는 소스코드와, 지금까지 학습한 AVX 벡터 연산, Pthread를 이용한 병렬처리를 이용하여 3D Convolution을 Single Context에서의 AVX 연산, multiThreads에서의 AVX연산, 그리고 CUDA를 이용한 GPU연산 3가지 버전으로 구현한다. 그 뒤, 3가지 버전의 성능을 측정하고 비교와 분석을 진행한다.

2. 구현 내용

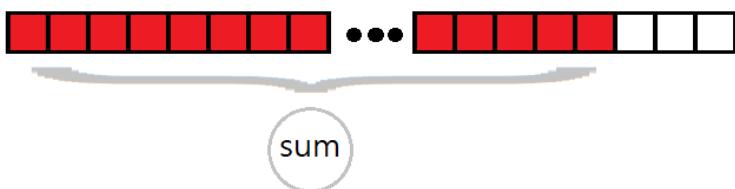
2.1 Single Context에서의 AVX 연산

3D Convolution 연산은 2D Convolution 연산과 원리는 똑같다. 커널이 입력 어레이에 포함된 모습으로 3차원 좌표를 이동한다. 그리고 각 이동 단계마다 커널의 각 요소들과 입력 어레이의 각 요소들 중 동일한 좌표를 가진 요소들을 모두 곱한 다음에 하나로 더한다. 이 하나의 결과값을 커널의 이동 좌표를 나타내는 출력 어레이에 저장한다. 이때, 커널의 크기가 1보다 크다면 출력 어레이는 입력 어레이보다 크기가 작아지게 된다. 이번 실습에서는 입력 어레이에 0으로 채워진 패딩(남는 공간)을 집어넣어 입력 어레이와 출력 어레이의 크기가 같아지게 만드는 구성도 구현했다. 커널은 정사각형 형태만 사용한다고 상정하므로, 각 Depths(z축), Rows(y축), Columns(x축)에는 $(\text{kernel width} - 1) / 2$ 크기만큼의 패딩이 0보다 작은 구역과 최고값보다 큰 구역에 각각 추가되어, 결론적으로 $\text{Depths} * \text{Rows} * \text{Columns}$ 크기의 입력 어레이가 $(\text{Depths} + 2\text{pad}) * (\text{Rows} + 2\text{pad}) * (\text{Columns} + 2\text{pad})$ 크기만큼으로 늘어나게 된다. 패딩이 불허진 배열을 새로 할당하는 것은 오버헤드가 매우 크므로, 실제 연산은 커널이 패딩 공간까지 고려하여 이동하면서 본래 입력 어레이에서 포함하지 않는 위치는 0으로 간주하도록 구현하였다.



AVX의 `_m256` 자료형은 256바이트의 데이터를 동시에 처리한다. 실습에서 구현한 3D Convolution은 float 자료형을 사용하므로, 한번에 8개의 곱셈을 병렬로 처리할 수 있다. 곱셈을 병렬로 처리하기 위해서, 데이터와 커널을 1차원 배열로 변환한다. 3D Convolution을 진행하기 위해선 각 위치 요소들의 곱을 합하기만 하면 되므로, 각 곱해져야하는 배열 요소의 맵핑만 일치한다면 1차원 어레이에서 곱을 진행해도 결과는 같기 때문이다. 1차원 배열로 변환된 데이터와 커널은 본래의 3차원 위치 정보와 관계없이 8개씩 AVX Multiple 연산을 할 수 있게 된다.

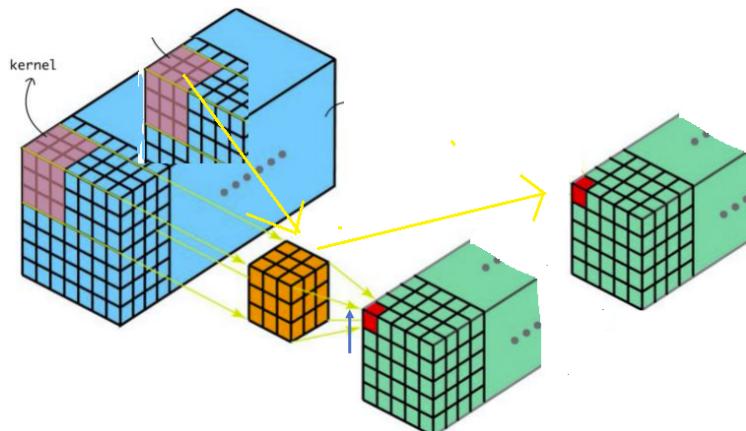
이때, 커널의 크기가 8의 배수가 아니라면 연산 공간에 빈 공간이 생기는데, 해당 위치는 커널의 값을 0으로 할당하여 곱셈은 진행하지만 계산 결과엔 영향을 미치지 않도록 한다. 이는 float 형태의 AVX 256 연산은 무조건 8개의 단위로 계산해야하기 때문에 발생하는 오버헤드이다.



모든 커널 요소의 곱셈이 끝났으면 각 결과들을 모두 더해 output 배열에 집어넣을 한개의 float 값을 구한다. 덧셈 과정은 순차적으로 각 요소들을 더해 결과를 구하도록 구현하였다.

2.2 Multi thread에서의 AVX 연산

Multi Thread에서의 AVX 연산은 2.1.에서 설명한 기존 방식의 연산에서 각 스레드들이 z축을 분할하여 연산을 진행하는 것으로 구현했다.



입력 어레이와 커널은 한번 값이 주어지면, 그 값이 변경될 일이 없다. 따라서 입력 어레이와 커널은 읽는 용도로만 사용이 되므로, 수많은 스레드들이 동시에 읽기를 진행해도 동기화 문제가 발생할 일이 없다. 출력 어레이는 한 출력 어레이 위치의 계산 과정에선 모든 곱들을 더한다는 순차적인 연산이 필요하지만, 서로 다른 위치의 출력 어레이 요소들끼리는 연관, 의존 요소가 전혀 없다. 따라서 멀티 스레드로 3D Convolution 작업을 분할하는 것은 각 출력 어레이의 구간을 분배해서 할당된 출력 어레이 구간의 값을 구하는 것이 타당할 것이다.

위에서 설명한 방식으로는 z 축 뿐만 아니라 y축, x축을 분할하더라도 스레드간 중복 계산 없이 명확하게 작동할 것이다. 하지만, 실제 구현은 z축을 분할해서만 작업을 할당하도록 구현했다. 왜냐하면, 3D Convolution은 어레이의 깊이가 보통은 크고, 스레드는 통상적으로 cpu의 동시 context 처리량 이상의 병렬 처리 기능은 기대할 수 없기에 대부분의 경우에는

멀티 스레드의 개수가 z축의 크기보다도 작을것이라고 생각했기 때문이다. 또한 x, y는 상대적으로 데이터에 지역적 요소가 있지만, z축은 데이터의 거리가 멀어 z 축으로 작업을 분할하는 것이 데이터의 지역적 요소를 고려해도 합리적이라고 판단했기 때문이다. 만약 x, y 크기만 크고 깊이는 하나인, 사실상 2D Convolution과 같은 구조에서는 z 축만 분할하는 방식으로는 병렬 처리가 작동하지 않을 것이다.

2.3 CUDA GPU에서의 연산

Shared memory를 활용하는 tiling 기법으로 convolution을 수행한다. 입력과 출력 배열 데이터의 크기가 같게 해주기 위해 커널 크기의 행렬에 (커널 크기 - 1) / 2 크기의 padding을 추가한 형태로 블럭을 설정하여 convolution을 수행한다.

타일의 크기가 너무 큰 값일 경우 한 블럭에 생성되는 스레드가 많아져 테스트 환경 GPU의 최대 블럭 당 스레드 개수인 1024를 초과할 수 있기 때문에 타일 크기를 1~5로 설정할 수 있다고 가정하여 구현한다.

인덱스 계산의 간소화를 위해 실제 convolution 계산을 할 때는 블럭을 (0,0,0) 지점에 위치한 것으로 간주하여 계산하고, 계산 결과를 배열에 저장할 때는 원래 모양의 패딩을 가진 상태로 인덱스를 되돌려 저장한다.

한 블럭에 속한 스레드들은 각각 자신이 속한 블럭의 shared memory를 공유하며, 여기에 값을 저장해 빠른 속도의 데이터 접근이 가능하게 된다. 각 스레드가 shared memory에 데이터를 저장하고 나서 동기화를 위해 __syncthreads() 함수를 호출한다.

커널 데이터는 constant memory에 저장하여 grid에 속한 스레드가 빠른 속도로 데이터를 읽어올 수 있도록 한다. 일반적인 global memory는 read/write 할 시 약 500 cycle이 소요되지만, constant memory의 경우 한 grid에 대한 read only memory로서 약 5 cycle이 소요된다. 커널 데이터는 모든 원소에 대해 곱해지는 데이터이면서 값이 변하지 않는 데이터이므로, convolution 연산 함수 호출 이전에 미리 정해진 크기로 constant memory를 선언하여 커널 데이터에 대한 읽기 속도를 높인다.

3. 연산 결과 검증

연산 결과의 검증은 샘플로 주어진 테스트 케이스의 출력 데이터를 연산 결과 데이터 배열과 비교하여 모든 데이터가 일치하면 검증된 것으로 간주한다. 이 때, 부동소수점 계산 및 타입 변환으로 인한 오차가 생길 수 있기 때문에 데이터 차이의 절댓값이 0.0001 보다 작다면 유효한 데이터로 간주한다.

```
Results are equal!
single thread avx execution time : 100.10100 ms

Results are equal!
5 multi-threads avx execution time : 37.03800 ms

Results are equal!
CUDA GPU execution time : 0.03811 ms
```

위 사진은 test2를 실행한 결과이다. 실행 결과 테스트를 오차 범위 이내로 통과하는 것을 확인할 수 있었다. 다른 3개의 테스트 케이스도 오차 범위 내에서 정상적으로 출력되는 것을 확인할 수 있었다.

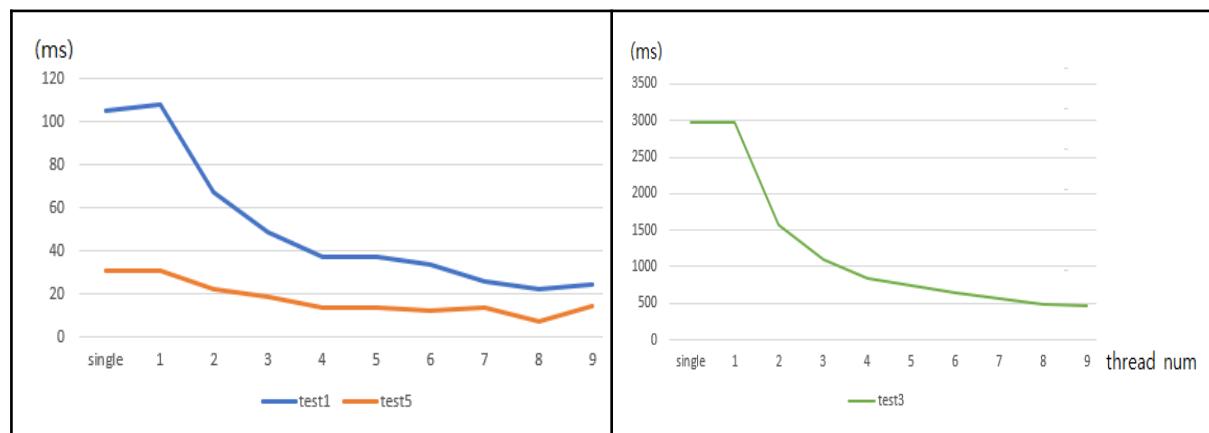
4. 성능 측정 및 비교

성능의 측정은 연산에 걸린 시간을 기준으로, 연산에 필요한 시간이 얼마나 줄어들었는가, 혹은 더 늘어났는가를 확인한다. 연산시간의 측정은 **avx**의 경우 연산 함수를 호출하기 직전과 직후에 **gettimeofday()** api를 통해 측정하며, **gpu**의 경우 커널 함수 호출 직전과 직후에 **cudaEventRecord()** api로 측정한다.

테스트 케이스 1번은 $64 \times 64 \times 64$ 행렬과 $3 \times 3 \times 3$ 의 커널, 3번은 $128 \times 128 \times 128$ 행렬과 $5 \times 5 \times 5$ 의 커널, 5번은 $32 \times 64 \times 32$ 행렬과 $3 \times 3 \times 3$ 의 커널 데이터로 이루어져 있다.

비교는 3가지를 진행한다. 첫번째는 **cpu**에서 작동하는 코드에서 **pthread**가 없는 버전과 **pthread**가 있는 버전 사이의 성능을 비교한다. 이때, **pthread**의 개수는 1개에서 9개까지 변화를 준다. 두번째는 순차 작동, 5개의 **pthread**에서의 병렬 작동, 그리고 **GPU**에서의 작동 3가지를 비교해본다. 세번째는 **GPU** 연산에서 각 테스트 케이스에 대해 타일 크기에 따른 성능 차이를 비교해보았다.

4.1. AVX-싱글스레드, AVX-멀티스레드 방식 비교



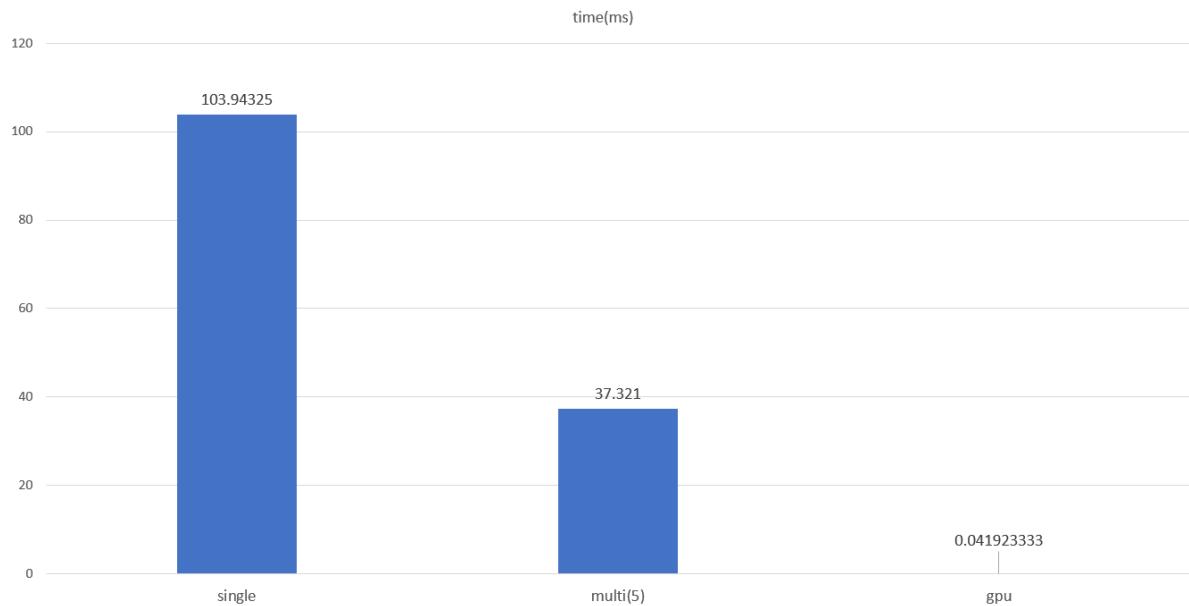
첫번째로, 1번, 3번, 5번 테스트 케이스에서 **thread**의 개수를 변화시켜가며 결과 시간을 측정한 뒤, **thread** 없이 계산했을 때의 결과와 합쳐 10가지 경우를 그래프로 나타내었다.

1번 ($64 \times 64 \times 64$, k=3) 테스트 케이스와 3번 ($128 \times 128 \times 128$, k=5) 테스트 케이스는 비슷한 양상을 보였다. 스레드가 4개가 될 때까지는 시간이 크게 감소하다가, 그 뒤로는 시간 감소 비율이 둔화되는 모습을 보였다. 반대로, 5번($32 \times 64 \times 32$, k=3) 케이스는 멀티 스레드의 증가에도 연산 시간이 크게 개선되지 않는 모습을 보였다.

여기서 데이터와 그 데이터를 통한 연산의 횟수가 충분히 크다면 **pthread**를 이용해서 병렬 처리의 효능을 볼 수 있지만, 5번 케이스 같이 데이터의 크기가 작은 경우엔 **pthread**만으로는 병렬처리를 통한 성능 향상을 기대하기 힘들다는 것을 알 수 있었다. 이는 연산 크기가 작아지면 병렬 처리로 얻을 수 있는 성능 향상 비율은 작아져도, 스레드를 생성하고, 작동하는 오버헤드는 여전히 존재하기에 발생하는 문제이다.

또한 테스트 1번이나 3번 같은 경우에도 스레드 증가에 비례하게 처리 성능이 증가하지 않고, 스레드 개수가 5~6개를 넘어가면 급격하게 처리속도의 증가량이 감소한다는 것을 알 수 있다. 이는 **pthread**를 많이 생성하여 리눅스 운영체제에 동시 처리를 요청하더라도, 여러가지 작업을 요청받는 **cpu**는 모든 **pthread context**를 동시에 실행시켜줄 수 없다는 것을 의미한다.

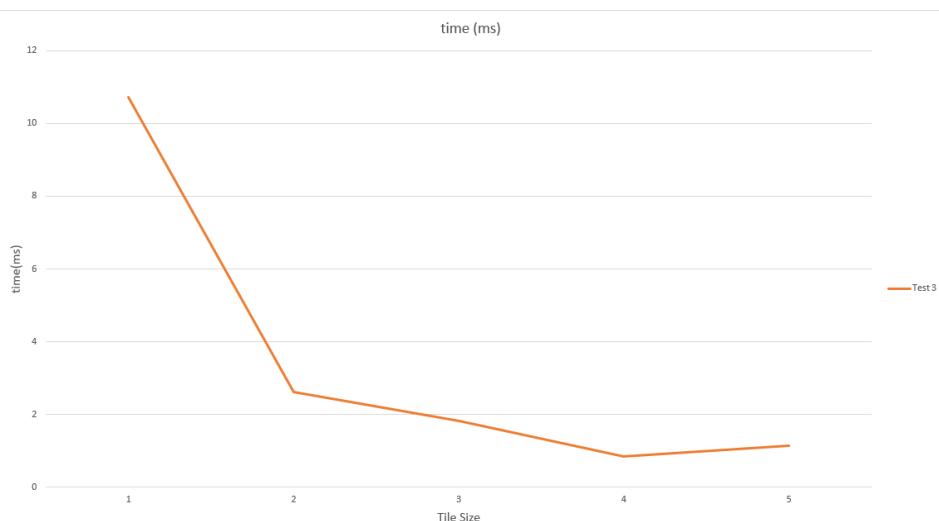
4.2. AVX-싱글스레드, AVX-멀티스레드(5개), CUDA GPU 방식



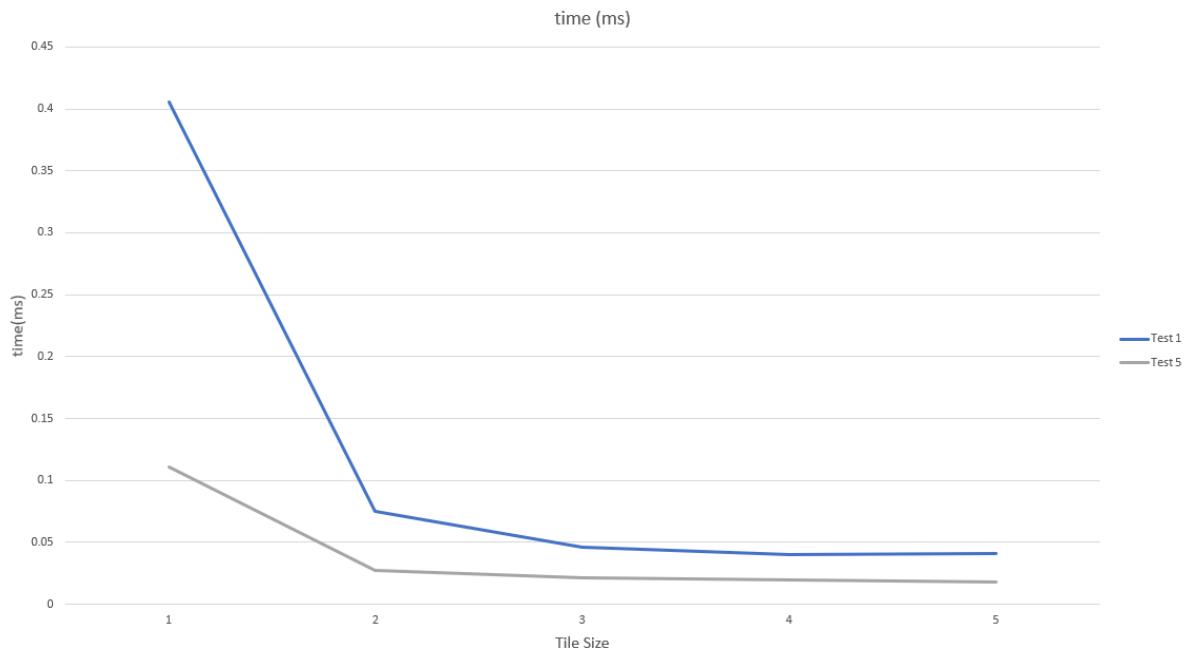
테스트 케이스 1번을 기준으로 각각 AVX 싱글스레드, AVX 멀티스레드(5개), CUDA GPU(타일 크기 5) 방식으로 연산한 시간을 측정하였고, 결과는 모두 샘플 데이터와 일치하였다. 싱글 스레드에 비해 5개의 멀티스레드를 사용한 경우 계산시간이 약 2.78배 빨라졌고, 멀티스레드에 비해 GPU로 계산한 시간이 약 837배 빨라졌다.

CPU에서 벡터연산을 멀티스레드로 실행한 경우와 GPU 병렬처리 방식으로 실행한 경우의 성능 차이가 상대적으로 매우 컸는데, 이는 CPU의 경우 convolution의 계산과정 중 커널과의 곱셈 부분만을 벡터로 연산하고 z축을 나누는 방식을 사용했지만 GPU는 행렬을 타일로 나누고 각 원소마다 스레드가 할당되어 계산하는 방식이기 때문이다. 행렬의 convolution 계산에서 data parallel하게 나누는 알고리즘의 차이와 물리적인 스레드 개수의 차이, 메모리 접근의 구조적 차이 등의 면에서 GPU가 좋은 성능을 보이기 때문에 이러한 결과가 나왔다고 할 수 있다.

4.3. CUDA GPU 방식



<Test 3 결과>



<Test 1, 5 결과>

테스트 케이스 1, 3, 5번에 대해 CUDA GPU 방식으로 타일크기가 1~5인 경우를 계산한 시간을 측정했으며, 결과는 모두 샘플 데이터와 일치하였다.

상대적으로 데이터의 크기가 큰 테스트 3번의 경우 타일의 크기가 1일 때에 비해 2일 때 급격하게 측정시간이 감소했다가 점차 완만하게 시간의 감소폭이 줄어드는 결과를 얻었다. 이는 타일크기가 1일 때는 블럭 당 스레드 개수가 상대적으로 매우 적어 global memory에 접근하는 횟수가 많아져 측정시간이 길고, 타일의 크기가 커지면 shared memory를 사용하는 비중이 많아져 시간이 단축되지만 병렬 실행으로 단축되는 시간의 한계가 존재하기 때문이다.

테스트 1, 5번도 3번과 같은 이유로 대체적으로 유사한 측정결과를 보였다. 상대적으로 데이터의 크기가 작은 테스트 5번이 다른 두 테스트에 비해 타일 크기 증가에 따른 연산시간의 감소폭이 작은 것은 데이터의 크기가 클수록 병렬 연산의 이점을 크게 얻을 수 있기 때문이다.

5. 결론

CPU에서의 AVX를 이용한 SIMD와 GPU에서의 CUDA thread를 이용한 SIMT의 차이점을 결과를 통해 알 수 있었다. 성능 테스트에 사용한 계산은 수백만 byte 크기의 데이터에 해당하는 3d 행렬을 convolution하는 방식이므로 data-parallel 방식으로 계산이 가능하기 때문에 GPU의 경우가 높은 성능을 보여줄 것이라고 예상했고, 결과적으로 800배 이상의 높은 성능을 얻을 수 있었다. 낮은 메모리 latency, superscalar를 통한 instruction level parallelism이 가능한 CPU에 비해 GPU의 수많은 스레드를 활용한 멀티스레딩 방식은 데이터의 크기가 크고 data parallel 할 수록 더욱 높은 성능차이를 보여줄 것으로 예상된다. 지금까지 진행했던 병렬 프로그래밍의 측정 결과를 바탕으로, 어플리케이션을 작성할 때 동시에 실행 가능한 작업을 스레드로 나누어 별개의 CONTEXT로 실행시키고, data parallel이 있는 연산을 AVX 연산이나 GPU 연산으로 처리하면 훨씬 좋은 성능을 낼 수 있다는 것을 알 수 있었다.