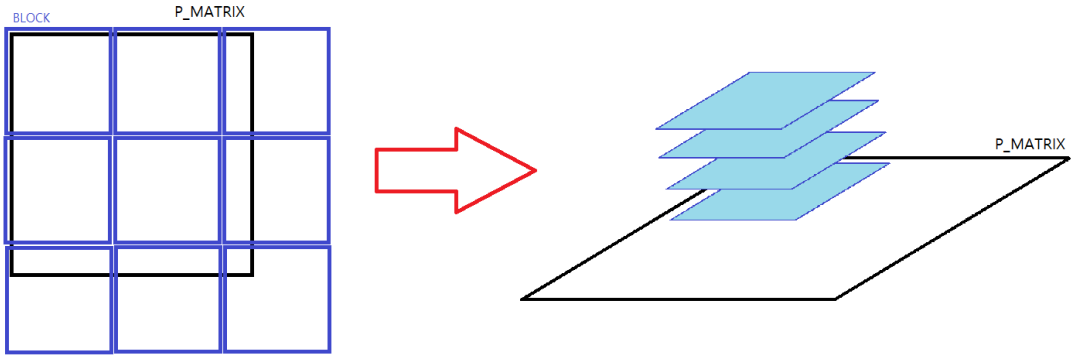
집중교육2- Ptheard Game of Life (5차과제 보고서)

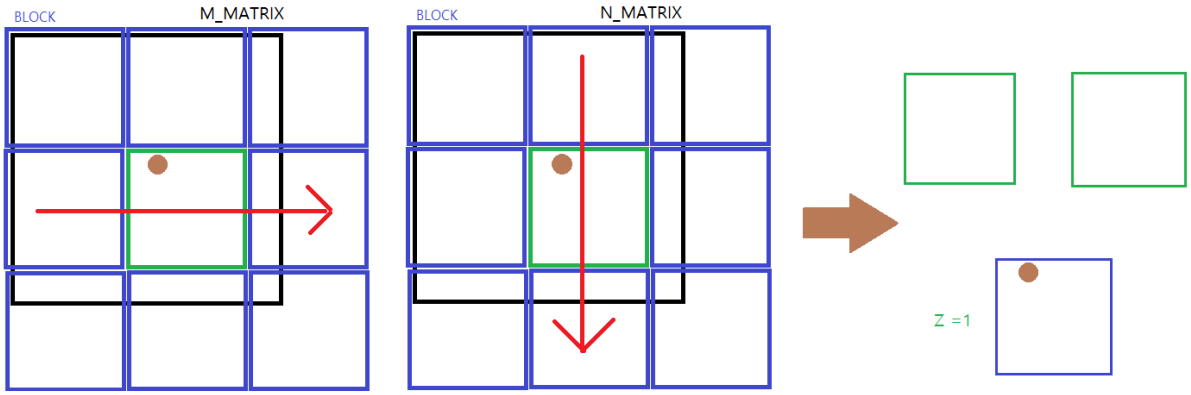
201521032 한태희

1. 디자인과 구현

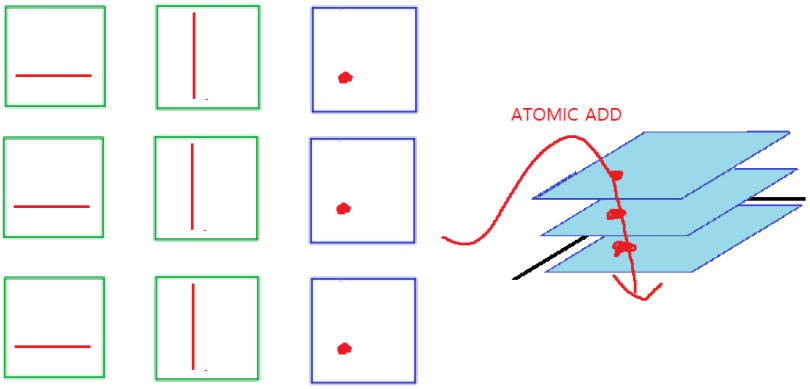
샘플로 제공된 행렬곱 함수는 행렬의 일부분을 정사각 모양의 Block으로 나눈 뒤에, 블록 내부의 스레드들이 결과 행렬의 하나의 요소를 담당해 해당 요소를 업데이트하기 위한 연산을 순차적으로 담당한다. 이 방법은 하나의 스레드가 GPU의 글로벌 메모리를 반복해서 참조하기에 메모리 최적화가 되어있지 않다. 실습 안내에서는 이를 해결하기 위해, 블록 공유 메모리에 연산에 필요한 값을 캐싱하는 방법을 제시했고, 해당 내용대로 새로운 함수를 작성했다.



우선, 타일 캐싱을 위해선 기존 블록 구조에 Z 축을 추가한 3차원 블록 구조가 필요하다. **Z 축의 크기는 ceil(MUL_line/TILE_WIDTH) 만큼 할당한다.** MUL_line은 두 피연산 행렬이 같아야 하는 행과 열의 크기이다. **타일의 사이즈는 피연산 구역과 연산 구역 모두 일괄적으로 입력받은 타일 크기대로 정사각형으로 만들어진다.**



한 블록은 자신의 Z축을 통해서 잘린 피연산 행렬의 부분 중 어떤 것을 캐싱해야하는지 알 수 있다. 이때, 2개의 캐시 공간과 1개의 블록 공간은 동일한 타일 크기로 잘라 크기가 같으므로, 블록 내부 스레드들이 각자 자신과 로컬 위치가 같은 값을 공유 메모리로 복사한다. 그리고 모든 스레드들은 자신과 같은 블록의 다른 스레드들이 값을 붙혀넣는것을 기다리고, 모든 캐싱이 완료되면 계산이 시작된다.



결국 같은 평면 좌표를 가지는 스레드들은 서로 같은 위치의 인덱스에 값을 업데이트 하는 작업을 분담하게 된다. 이때 결과는 ATOMIC만 보장되면 되고, 각 실행 순서는 연관이 없으므로 Z축으로 분할된 각 스레드들이 자신이 얻은 연산 결과를 ATOMIC ADD로 반환할 행렬에 업데이트 하면 된다.

2. 검증

```
M Matrix:::
-3.700000 -3.610000
2.300000 4.500000
-1.350000 0.370000

N Matrix:::
4.360000 2.030000 -0.210000 1.980000
-1.730000 4.110000 -4.500000 -3.540000


P Matrix:::
-9.886700 -22.348101 17.021999 5.453400
2.243000 23.164001 -20.733000 -11.376000
-6.526100 -1.219800 -1.381500 -3.982800
```

$$\begin{pmatrix} -3.7 & -3.61 \\ 2.3 & 4.5 \\ -1.35 & 0.37 \end{pmatrix} \cdot \begin{pmatrix} 4.36 & 2.03 & -0.21 & 1.98 \\ -1.73 & 4.11 & -4.5 & -3.54 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{-98867}{10000} & \frac{-223481}{10000} & \frac{8511}{500} & \frac{27267}{5000} \\ \frac{2243}{1000} & \frac{5791}{250} & \frac{-20733}{1000} & \frac{-1422}{125} \\ \frac{-65261}{10000} & \frac{-6099}{5000} & \frac{-2763}{2000} & \frac{-9957}{2500} \end{pmatrix}$$

왼쪽은 랜덤으로 생성한 2* 3 행렬과 4*2 행렬의 곱을 1의 Tiled 연산으로 구한 결과를 출력한 것이고, 오른쪽은 해당 결과를 웹에서 할 수 있는 행렬 계산으로 검증한 것이다.

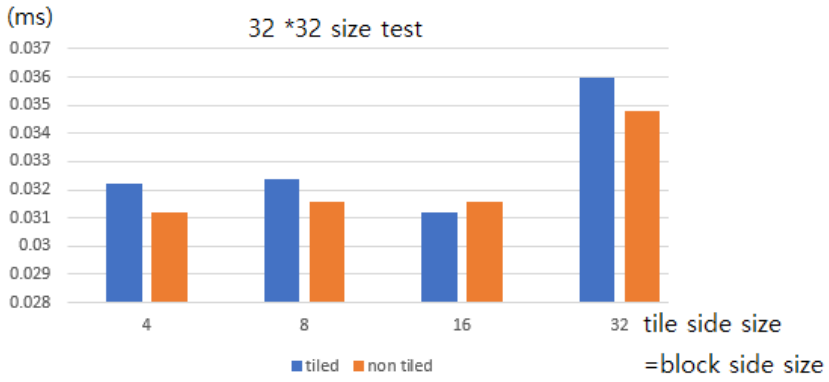
3. 성능 측정과 분석

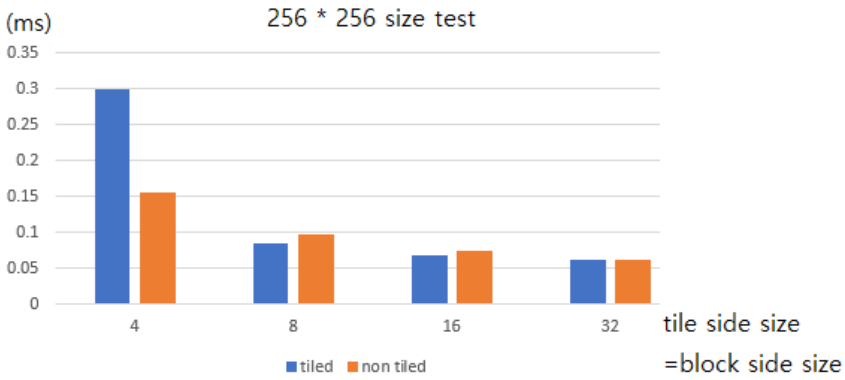
성능 측정은 행렬 곱을 진행하고, 연산이 진행되는 동안의 시간을 측정하는 방식으로 진행했다. 이때 변경하는 독립변수는 아래와 같다.

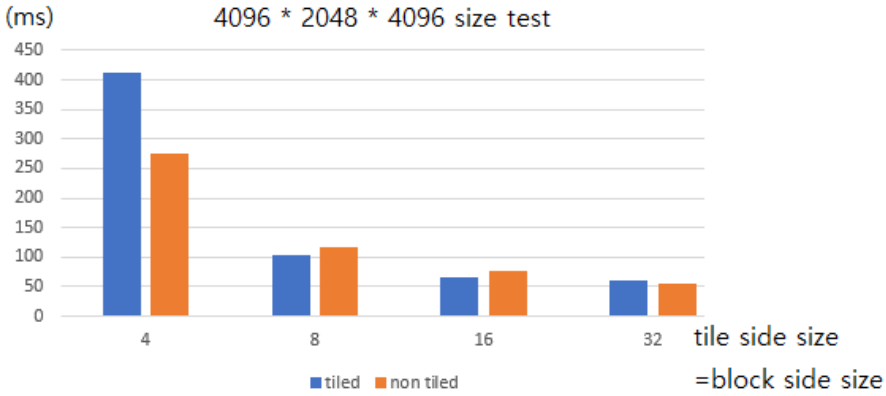
- 입력 행렬의 크기 <small (32*32), middle(256*256), big(4096*2024*4096)> 3가지

- <tile side size> 4, 8, 16, 32 (block side size가 곧 tile side size가 된다.)

-tiled cache 알고리즘 사용 여부 <tiled vs non tiled>







측정 결과 예상과는 달리 tiled cache 방법이 모든 경우에 non tiled cache보다 좋은 성능을 보여주지는 않았다. tile side size가 4일 경우에는 블록이 너무 많아지고 캐시 그리드의 크기가 작아져 tile cache를 사용하지 않았을 때보다 더 안 좋은 실행 결과가 나왔다. 반대로, tile side size를 32로 잡았을 경우에도 유의미한 성능 향상을 보이지 못하였다. tile cache을 사용한 성능 향상을 볼 수 있었던 경우는 middle, big에서 8 크기의 타일인 경우와 모든 경우에서 16개의 타일인 경우였다. 하지만 극적으로 성능이 빠르게 변하진 않았고, 몇 퍼센트의 실행 시간만 단축할 수 있을 뿐이었다.

tile cache를 사용할 때, tile 크기에 따라서도 속도에 큰 변화가 있었다. small에서는 모든 결과가 유의미한 차이를 보이지 않았다. 하지만 middle과 big에서는 4의 사이즈는 매우 긴 실행 시간이 소모되었다. 앞서 언급했던 것처럼 tile cache로 메모리 접근 속도의 이득을 얻는 것보다 block을 너무 잘게 나누어 그것으로 인한 오버헤드가 훨씬 더 커진 상황이기 때문이다.