GPGPU (CUDA) 환경에서 행렬-벡터 곱셈 연산 최적화 실험 및 결과 분석

By taeguk (http://taeguk.me)

가. 실험 설명

32x32 행렬과 Nx32 의 행렬을 곱하는 프로그램을 CPU 와 GPU(CUDA)를 이용해서 구현한다. 행렬의 메모리상 저장 방식이 AOS(Array Of Structure)일 경우와 SOA(Structure Of Array, AOS 의 Transposed Version)일 경우에 대해 각각 프로그램을 작성하며 memory access pattern 에 따른 성능을 분석한다. 그리고 GPU 의 경우 CUDA Memory Architecture 상에서 global memory, shared memory, constant memory 에 대한 good practice 를 따라서 최적화를 진행하여 과연 이론과 실제가 일치하는 지, 그리고 실험결과에 대한 이론적 분석을 진행한다.

나. 실험 개요

A. 실험 환경

Windows 10
Visual Studio 2013 (Release, x64)
CUDA 7.5
GPU : GT630 (Compute Capibility = 3.0)
CPU : Intel® Core™ iy-3770 CPU @ 3.40GHz
RAM 8GB
위 환경에서 실험하였다.

Vector 의 개수는 2^19 (=524288) 개. Vector 내의 Element 개수는 32 개로 하였다.

B. CPU를 이용한 구현

1. CPU_AOS

- 자료구조의 저장방식이 AOS일 때 CPU 구현.

2. CPU_AOS_2

- CPU_AOS에서 cache를 고려해 loop index를 뒤바꾼 것.

3. CPU_SOA

- 자료구조의 저장방식이 SOA일 때 CPU 구현.

4. CPU_SOA_2

- CPU_SOA에서 cache를 고려해 loop index를 뒤바꾼 것.

C. GPU를 이용한 기본적인 구현

1. GPU_1_AOS__VECTOR_PER_THREAD

- 자료구조의 저장방식이 AOS일 때 GPU구현.
- Thread 1개당 Vector 1개를 계산한다. (총 32개의 Element)
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.

2. GPU_1_AOS__ELEM_PER_THREAD

- 자료구조의 저장방식이 AOS일 때 GPU구현.
- Thread 1개당 Element 1개를 계산한다.
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.

3. GPU_1_SOA__VECTOR_PER_THREAD

- 자료구조의 저장방식이 SOA일 때 GPU구현.
- Thread 1개당 Vector 1개를 계산한다. (총 32개의 Element)
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.

4. GPU_1_SOA__ELEM_PER_THREAD

- 자료구조의 저장방식이 SOA일 때 GPU구현.
- Thread 1개당 Element 1개를 계산한다.
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.

D. AOS방식, ELEM_PER_THREAD일 때, Shared Memory와 Constant Memory를 사용한 구현

1. GPU_2_AOS__SHARED_X

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.

2. GPU_2_AOS__SHARED_X_M

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X와 행렬 M을 위해 Shared Memory를 사용.

3. GPU_2_AOS__SHARED_X_CONSTANT_M

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.
- 행렬 M을 위해 Constant Memory 사용.

E. AOS방식, ELEM_PER_THREAD일 때, Shared Memory와 Constant Memory를 사용한 구현

1. GPU_3_SOA__SHARED_X

- GPU_1_SOA__VECTOR_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.

2. GPU_3_SOA__SHARED_X_M

- GPU_1_SOA__ VECTOR _PER_THREAD를 기반으로 한다.
- 벡터들 X와 행렬 M을 위해 Shared Memory를 사용.

3. GPU_3_SOA__SHARED_X_CONSTANT_M

- GPU_1_SOA__ VECTOR _PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.
- 행렬 M을 위해 Constant Memory 사용.

4. GPU_3_SOA__CONSTANT_M

- GPU_1_SOA__ VECTOR _PER_THREAD를 기반으로 한다.
- 행렬 M을 위해 Constant Memory 사용.

다. 실험결과

A. CPU를 이용한 구현

1. CPU_AOS

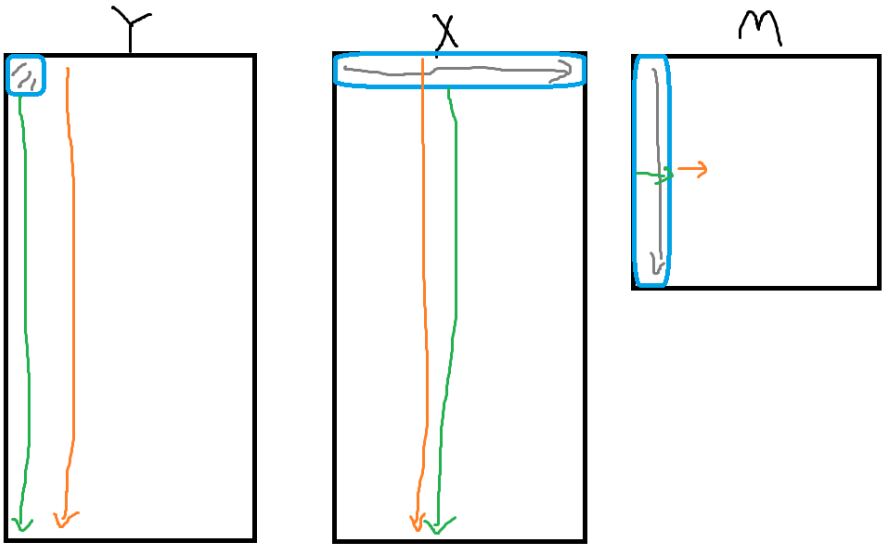
- 자료구조의 저장방식이 AOS일 때 CPU 구현.

CPU_AOS는 총 3중 loop를 도는데, 맨 바깥 loop의 index는 vector를 의미하고, 중간 loop의 index는 vector내의 element를 의미하고, 가장 안쪽 loop의 index는 element 1개를 계산하기 위한 loop이다.

CPU_AOS에서는 안쪽의 2중 loop동안, X의 row 1개가 access pattern이 일정해서 cache의 이득을 본다.

2. CPU_AOS_2

- CPU_AOS에서 cache를 고려해 loop index를 뒤바꾼 것.



위와 같은 memory access pattern을 가진다.

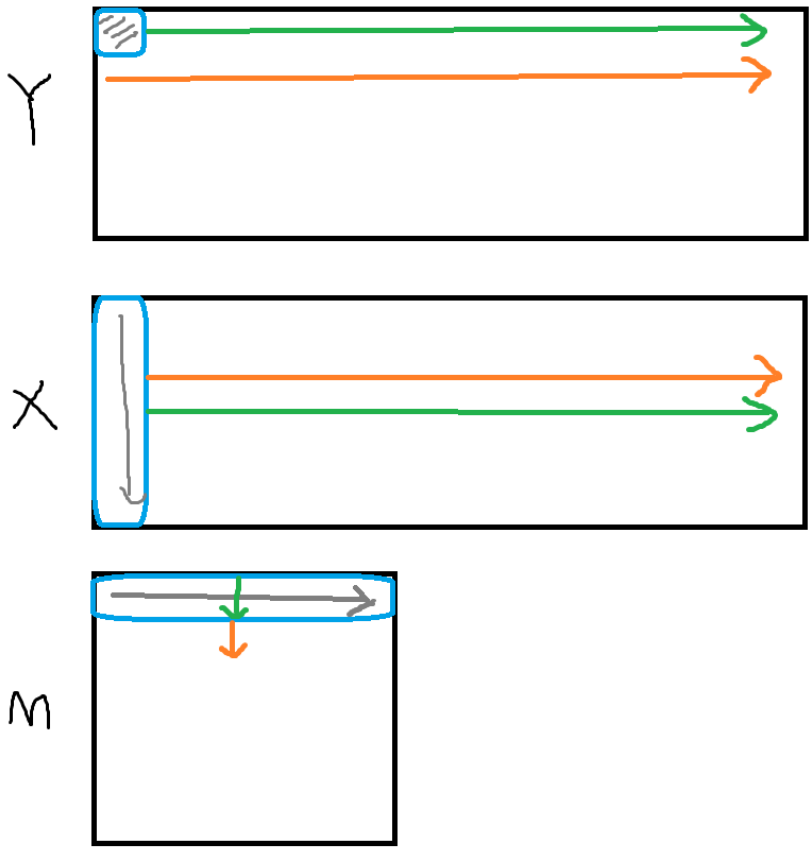
CPU_AOS_2는 CPU_AOS에서 맨 바깥쪽 loop랑 중간의 loop를 서로 교체한 version이다.

CPU_AOS_2에서는 안쪽의 2중 loop동안, M의 column 1개가 access pattern이 일정해서 cache의 이득을 본다.

근데 CPU_AOS_2이 CPU_AOS보다 아주아주 약간 빠른데 그 이유는, CPU_AOS_2에서는 X가 어차피 row wise access라 cache의 이익을 여기서도 받는데에 반해, CPU_AOS에서는 M이 column wise access라 cache의 이익을 못 받기 때문으로 생각된다.

3. CPU_SOA

- 자료구조의 저장방식이 SOA일 때 CPU 구현.



위와 같은 memory access pattern을 가진다.

CPU_SOA는 CPU_AOS에 비해 속도가 10배이상 느리다. 그 이유는 X에 대한 access pattern이 column wise이기 때문이다. AOS에서도 M에 대해서 column wise access를 하지만, M의 크기가 작기때문에 cache가 M를 잘 담고있어서 성능피해가 적은 것으로 생각된다. 그러나 X는 크기가 크므로 access pattern이 column wise이면 피해가 커서 속도가 상당히 느리게 나온 것으로 보인다.

4. CPU_SOA_2

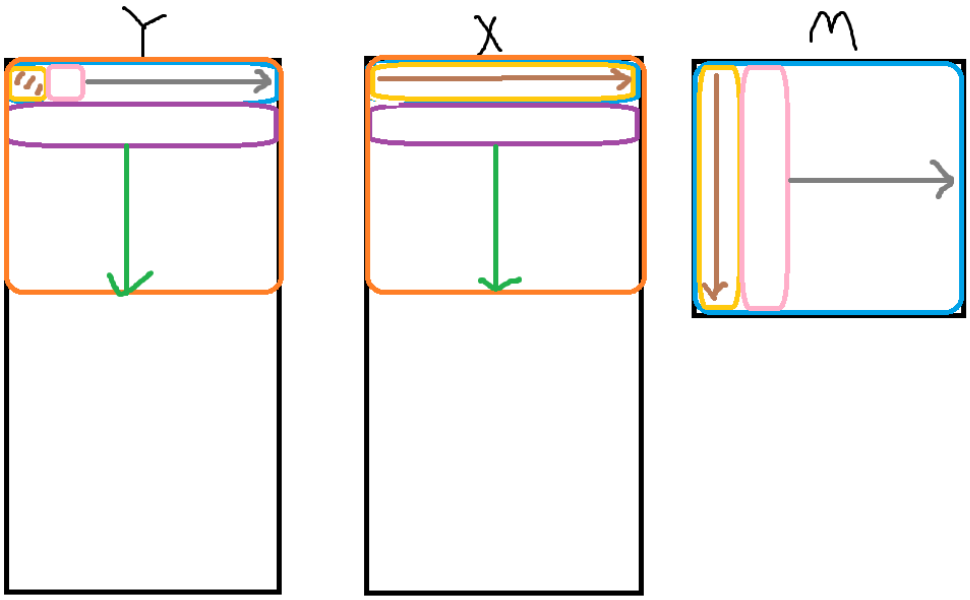
- CPU_SOA에서 cache를 고려해 loop index를 뒤바꾼 것.

CPU_AOS와 CPU_AOS_2의 차이와 비슷한 이유로 CPU_SOA보다 약간 더 빠르다.

B. GPU를 이용한 구현

1. GPU_1_AOS_VECTOR_PER_THREAD

- 자료구조의 저장방식이 AOS일 때 GPU구현.
- Thread 1개당 Vector 1개를 계산한다. (총 32개의 Element)
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.

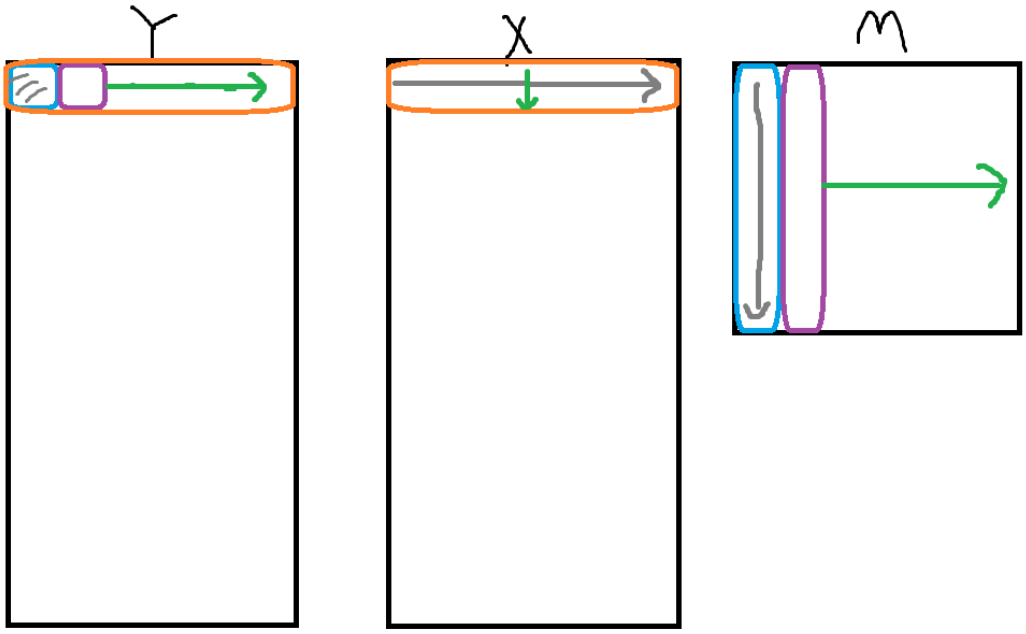


위와 같은 memory access pattern을 가진다.

성능이 매우 안좋다. X에 대한 접근이 coalescing 이 아니라서 warp내에서 X에 대한 데이터를 읽기 위해 32번의 memory transaction이 필요하다. Warp내에서 M에 대한 접근이 항상 same address라는 점은 좋지만, X 에 대한 접근이 치명적으로 비효율적이라서 전체적인 성능은 매우 안좋다. Warp내에서 X에 대한 접근이 항상 same address이므로, constant memory를 사용하면 상당부분 성능이 향상될 것으로 보이는 구조이다.

2. GPU_1_AOS_ELEM_PER_THREAD

- 자료구조의 저장방식이 AOS일 때 GPU구현.
- Thread 1개당 Element 1개를 계산한다.
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.



위와 같은 memory access pattern을 가진다.

GPU_1_AOS_VECTOR_PER_THREAD 보다 훨씬 좋다.

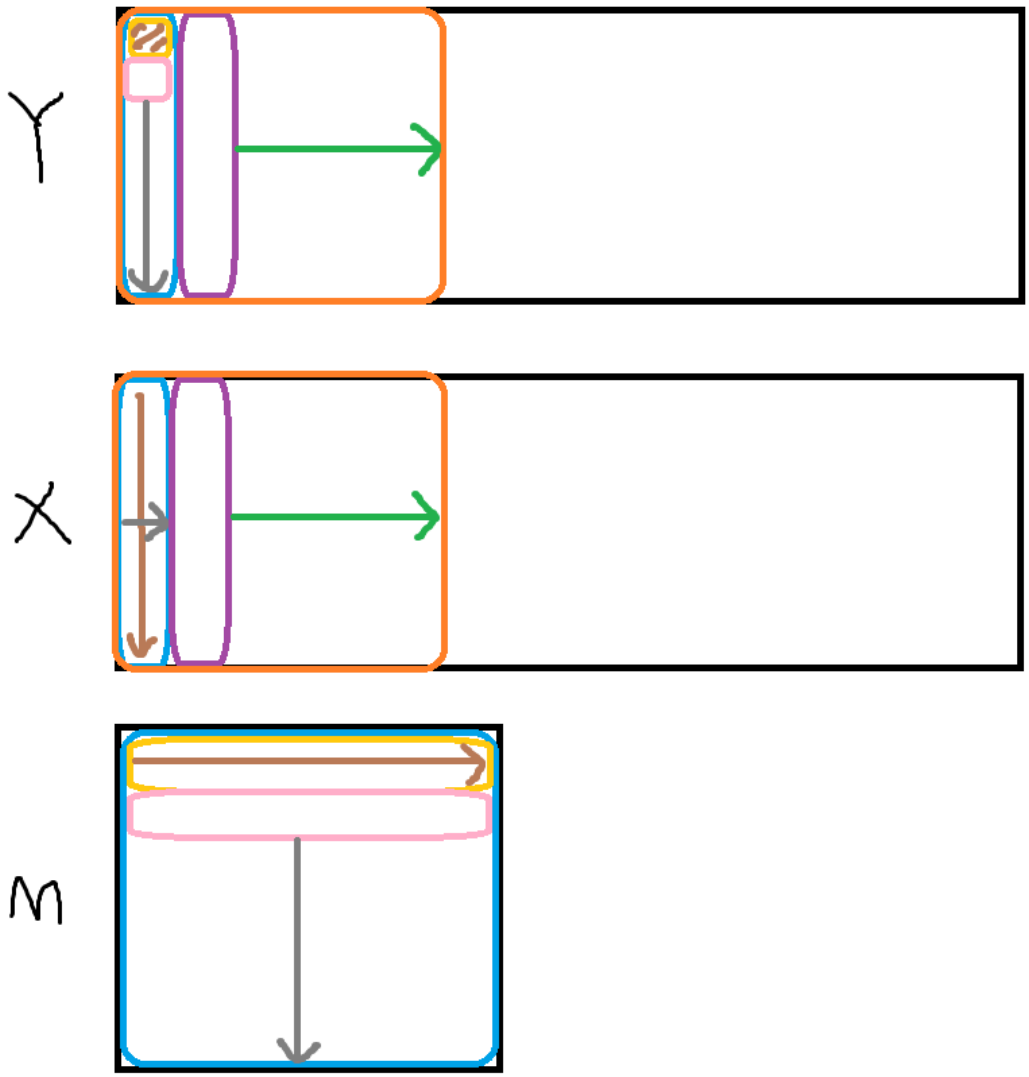
Warp내에서 X에 대한 접근이 항상 same address라서, broadcast가 일어나 매우 효율

적이다. 그리고 X의 크기가 커서 constant memory를 이용한 최적화는 불가능하지만, bank conflict가 없을 것이므로 shared memory를 활용하면 성능이 상당히 향상될 것이다.

또한 Warp내에서 M에 대한 접근이 서로 인접한 address라서 coalescing이 일어나 매우 효율적이다. 따라서 bank conflict가 없을 것이므로 shared memory를 활용하면 효과적일 것이다.

3. GPU_1_SOA__VECTOR_PER_THREAD

- 자료구조의 저장방식이 SOA일 때 GPU구현.
- Thread 1개당 Vector 1개를 계산한다. (총 32개의 Element)
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.



위와 같은 memory access pattern을 가진다.

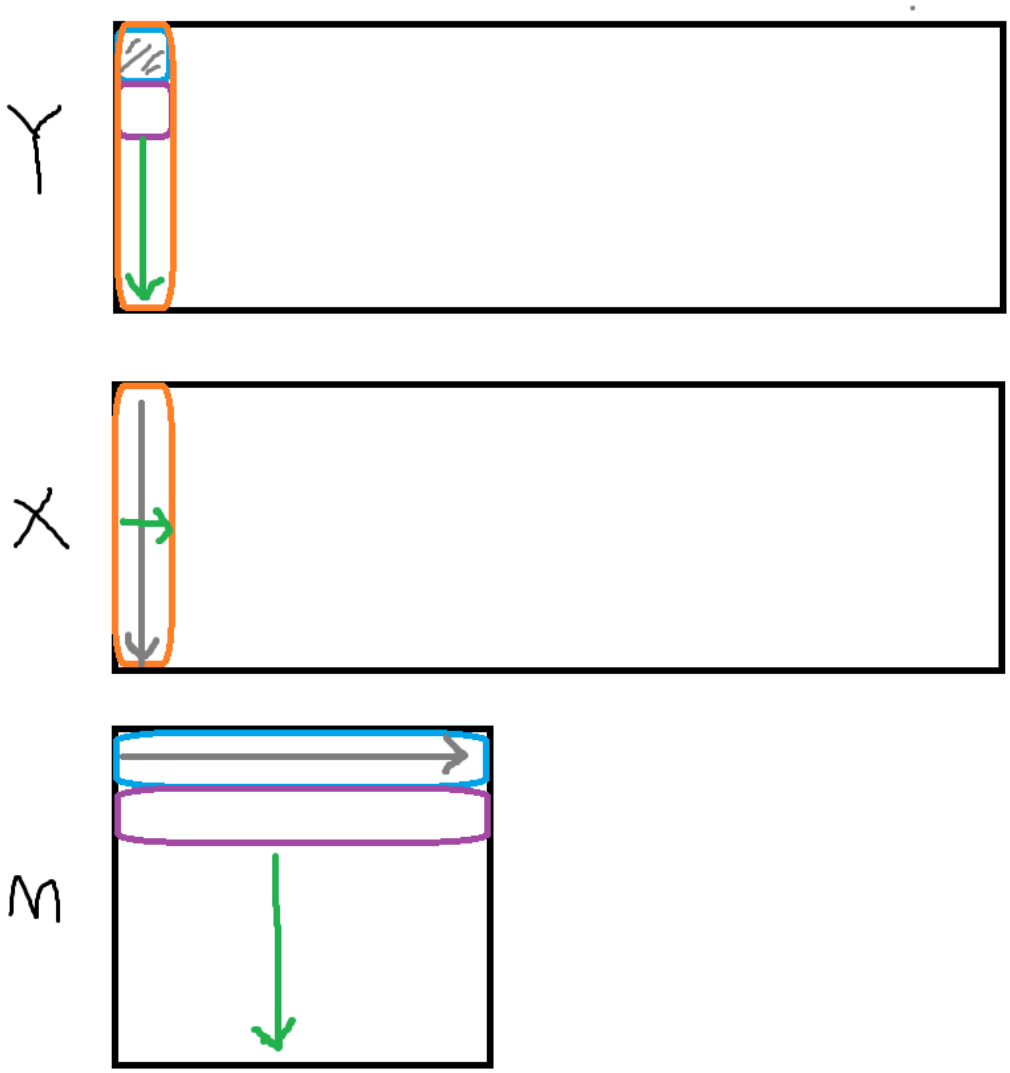
GPU_1_SOA__ELEM_PER_THREAD 보다 훨씬 좋다.

Warp내에서 X에 대한 접근이 서로 인접한 address라서 coalescing이 일어나 매우 효율적이다. 또한 이 경우 bank conflict가 없으므로 shared memory를 활용하면 효과적일 것이다. 라고 생각하기 쉽지만, 실제결과는 shared memory를 사용하면 더 느리다. 이 것은 밑에서 다룰 것이다.

그리고 warp내에서 M에 대한 접근이 항상 same address라서, broadcast가 일어나 매우 효율적이다. 따라서 constant memory를 이용한 최적화를 하면 효과적일 것이다.

4. GPU_1_SOA__ELEM_PER_THREAD

- 자료구조의 저장방식이 SOA일 때 GPU구현.
- Thread 1개당 Element 1개를 계산한다.
- 행렬 M과 벡터들 X를 위해 Global Memory를 사용.



위와 같은 memory access pattern을 가진다.

성능이 상당히 안좋다.

M에 대한 접근이 coalescing 이 아니라서 warp내에서 X에 대한 데이터를 읽기 위해 32번의 memory transaction이 필요하다.

X에 대한 접근이 항상 same address라는 점이 좋지만, M에 대한 접근이 치명적으로 비효율적이라서 전체적인 성능은 매우 안좋다.

C. AOS방식, ELEM_PER_THREAD일 때, Shared Memory와 Constant Memory를 사용한 구현

1. GPU_2_AOS__SHARED_X

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.

위에서 언급한 GPU_1_AOS__ELEM_PER_THREAD의 access pattern대로면, warp내에서 X에 대한 접근이 항상 same address라서 shared memory를 이용하면 broadcast가 일어나 매우 효율적일 것으로 추측된다. 실제로 적용한 결과, 약간 성능향상이 된 것을 확인할 수 있었다. 그러나 그렇게 큰 성능향상은 얻지 못했는데, 그 이유는 어차피 X의 경우 global memory상에서도 broadcast에 의해 성능이 좋았고, X에 대한 접근이 row wise이기 때문에 spacial locality특성에 따라 cache의 득을 봤기 때문으로 생각된다. 따라서 shared memory를 사용했음에도 shared memory에 대한 cost에 비해 엄청나게 큰 성능향상은 없어서 결과적으로 약간만 성능이 향상된 것으로 보인다.

2. GPU_2_AOS__SHARED_X_M

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X와 행렬 M을 위해 Shared Memory를 사용.

GPU_2_AOS__SHARED_X에서 추가적으로 M에 대해 shared memory를 사용한 버전이다. GPU_2_AOS__SHARED_X에 비해 거의 2배에 가까운 성능향상을 보여준다. X는 shared memory를 사용해도 큰 성능향상이 없는데에 반해, M의 경우는 큰 성능향상을 보여준다. 그 이유는 M에 대한 access가 column wise이기 때문에 spacial locality 를 띄지않아서 이에 대한 cache의 이득이 적어 M을 shared memory에 올리면 엄청난 성능향상이 일어나는 것이라고 생각된다.

3. GPU_2_AOS__SHARED_X_CONSTANT_M

- GPU_1_AOS__ELEM_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.
- 행렬 M을 위해 Constant Memory 사용.

GPU_2_AOS__SHARED_X에서 추가적으로 M에 대해 constant memory를 사용한 버전이다. GPU_2_AOS_SHARED_X에 비해 약 10배가까히 성능이 '하락'하는데, 이는 M에 대한 access pattern이 same address가 아니라서 memory access가 32번으로 나누어 일어나 성능이 상당히 하락하기 때문이다.

D. AOS방식, ELEM_PER_THREAD일 때, Shared Memory와 Constant Memory를 사용한 구현

1. GPU_3_SOA__SHARED_X

- GPU_1_SOA__VECTOR_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.

GPU_1_SOA__VECTOR_PER_THREAD에서 추가적으로 X에 대해 shared memory를 사용한 버전이다. GPU_1_SOA__VECTOR_PER_THREAD에 비해서 오히려 50%정도 성능이 '하락'하는데 그 이유는 어차피 X에 대한 access가 coalescing에 의해 효율적이였고, (CUDA_WARP_SIZE * ELEM_PER_VECTOR) (=1024) 크기. 즉 4KB에 해당하는 X의 부분 영역이

총 ELEM_PER_VECTOR만큼 iterate되며 반복해서 접근되기에 temporal locality에 의해 cache의 이득을 많이 봤을 것이다.

그런데 X를 shared memory에 올림으로서 thread가 256개일 때, 32768bytes만큼의 shared memory가 할당됨으로 인해, SM내에 resident하는 block의 개수가 1개로 제한되고 (실험환경에서 shared memory의 크기는 48KB라서) 즉, resident warp의 개수는 8개가 될테고, 이로 인해 M을 global memory에서 읽을 때, stall이 발생하면 scheduling될 warp가 적어 이로 인해 성능이 하락되는 것이 아닐까라고 추측된다.

2. GPU_3_SOA__SHARED_X_M

- GPU_1_SOA__VECTOR_PER_THREAD를 기반으로 한다.
- 벡터들 X와 행렬 M을 위해 Shared Memory를 사용.

GPU_3_SOA__SHARED_X에서 추가적으로 M에 대해 shared memory를 사용한 버전이다. 이 경우 GPU_3_SOA__SHARED_X에 비해 성능이 3배정도 향상되는데, 그 이유는 아마 M이 shared memory에 상주함으로서 M에 의한 stall이 발생하지 않아 GPU_3_SOA__SHARED_X에서 언급한 문제가 해결되어 성능이 많이 향상되는 것으로 생각된다.

3. GPU_3_SOA__SHARED_X_CONSTANT_M

- GPU_1_SOA__VECTOR_PER_THREAD를 기반으로 한다.
- 벡터들 X를 위해 Shared Memory를 사용.
- 행렬 M을 위해 Constant Memory 사용.

GPU_3_SOA__SHARED_X에서 추가적으로 M에 대해 constant memory를 사용한 버전이다. 이 경우 GPU_3_SOA__SHARED_X에 비해 약 7~8배, GPU_3_SOA__SHARED_X_M에 비해 약 2~3배, GPU_1_SOA__VECTOR_PER_THREAD에 비해 약 4~5배 정도 빨라진다.

GPU_3_SOA__SHARED_X_M에 비해서 빠른 이유는 M을 shared memory로 놓을 경우, block마다 초기화와 할당이 필요하지만, constant memory는 전체에서 접근이 가능하기 때문이다.

4. GPU_3_SOA__CONSTANT_M

- GPU_1_SOA__VECTOR_PER_THREAD를 기반으로 한다.
- 행렬 M을 위해 Constant Memory 사용.

GPU_1_SOA__VECTOR_PER_THREAD에서 추가적으로 M에 대해 constant memory를 사용한 버전이다. 성능은 거의 같다고 봐도 될 정도로 아주 미세하게 향상하였는데, 아마도 GPU_1_SOA__VECTOR_PER_THREAD에서 이미 M에 대한 cache hit이 많이 떠서, 혹은 cache miss로 인해 stall되도 GPU_3_SOA__SHARED_X에 비해 resident warp가 많아서 scheduling이 되기 때문에 constant memory를 사용하여도 성능향상이 적은 것이 아닐까라고 생각된다.

E. 성능 측정 결과

```
N = 524288

------------------------------------------------

Finish CPU_AOS calculation
 - Elapsed time: 437.861145 (msec)

Finish CPU_AOS_2 calculation
 - Elapsed time: 395.029266 (msec)

Finish CPU_SOA calculation
 - Elapsed time: 4599.692383 (msec)

Finish CPU_SOA_2 calculation
 - Elapsed time: 4471.449219 (msec)

AOS and SOA(transpose) compare -> Error rate: 0.00%
```

```
------------------------------------------------

Finish GPU_1_AOS__VECTOR_PER_THREAD <<< 2048, 256 >>> calculation
 - Elapsed time: 1307.427124 (msec)
 - Error rate: 0.00%

Finish GPU_1_AOS__ELEM_PER_THREAD <<< 65536, 256 >>> calculation
 - Elapsed time: 96.943169 (msec)
 - Error rate: 0.00%

Finish GPU_1_SOA__VECTOR_PER_THREAD <<< 2048, 256 >>> calculation
 - Elapsed time: 101.670723 (msec)
 - Error rate: 0.00%

Finish GPU_1_SOA__ELEM_PER_THREAD <<< 65536, 256 >>> calculation
 - Elapsed time: 1307.678955 (msec)
 - Error rate: 0.00%
```

```
------------------------------------------------

Finish GPU_2_AOS__SHARED_X <<< 65536, 256 >>> calculation
 - Dynamic shared memory size: 1024 bytes
 - Elapsed time: 80.951134 (msec)
 - Error rate: 0.00%

Finish GPU_2_AOS__SHARED_X_M <<< 65536, 256 >>> calculation
 - Dynamic shared memory size: 1024 bytes
 - Elapsed time: 47.718113 (msec)
 - Error rate: 0.00%

Finish GPU_2_AOS__SHARED_X_CONSTANT_M <<< 65536, 256 >>> calculation
 - Dynamic shared memory size: 1024 bytes
 - Elapsed time: 744.833679 (msec)
 - Error rate: 0.00%
```

```
-----------------------------------------------

Finish GPU_3_SOA__SHARED_X <<< 2048, 256 >>> calculation
 - Dynamic shared memory size: 32768 bytes
 - Elapsed time: 159.539047 (msec)
 - Error rate: 0.00%

Finish GPU_3_SOA__SHARED_X_M <<< 2048, 256 >>> calculation
 - Dynamic shared memory size: 32768 bytes
 - Elapsed time: 58.844418 (msec)
 - Error rate: 0.00%

Finish GPU_3_SOA__SHARED_X_CONSTANT_M <<< 2048, 256 >>> calculation
 - Dynamic shared memory size: 32768 bytes
 - Elapsed time: 23.935968 (msec)
 - Error rate: 0.00%

Finish GPU_3_SOA__CONSTANT_M <<< 2048, 256 >>> calculation
 - Elapsed time: 99.564926 (msec)
 - Error rate: 0.00%
```

```
- Finish
```

CPU 중에 가장 빠른 것 : CPU_AOS_2 (395.03 msec)
CPU 중에 가장 느린 것 : CPU_SOA (4599.69 msec)
GPU 중에 가장 빠른 것 : GPU_3_SOA_SHARED_X_CONSTANT_M (23.94msec)
GPU 중에 가장 느린 것 : GPU_1_SOA_ELEM_PER_THREAD (1307.68msec)

CPU 와 GPU 를 합치면, 가장 빠른 것과 가장 느린 것이 약 192 배 차이 난다.
GPU 만 따졌을 때, 가장 빠른 것과 가장 느린 것이 약 55 배 차이 난다.
이렇듯 memory access pattern 을 고려해 최적화하는 것이 상당히 중요함을 알 수 있었다.

Note)

Global Memory : Warp내의 thread들이 인접한 memory address에 접근해야 한다. 그래야 coalescing에 의하여 좋은 성능을 얻을 수 있다. 최선의 경우 2번의 memory transaction, 최악의 경우 32번의 memory transaction을 수행해야한다.

Shared Memory : Warp내의 thread들이 인접한 memory address에 접근해야 bank conflict가 발생하지 않는다. (최근의 gpu들은 bank가 32개이므로) 만약 bank conflict가 발생하면 memory padding을 통해 해결해야한다.

Constant Memory : Warp내의 thread들이 동일한 memory address에 접근해야 좋은 성능을 발휘한다. (2번의 memory request) 만약 서로 전부 다 다른 memory address에 접근하면 32번의 memory request가 생성된다.