Assignment 2

Subject : Computing 2

Prof : 성효진 교수님

Name : 최규빈

Student No : 2024-24934

Major : 데이터사이언스

Date : 2025.10.23.

Contents

1. Introduction

본 과제는 CUDA를 통해 Matrix 연산 및 Tiling 기법을 구현하는 것을 목표로 한다.

해당 구현을 통해 shared memory를 통해 operational intensity를 높여 성능 향상을 얻는 과정을 확인 가능하다.

2. Implementation

실제 구현의 경우 template.cu 파일을 통해 제출되며 각 코드의 구현 내용은 Questions 챕터의 5번 항목에서 설명된다.

3. Questions

(1) How many floating operations are being performed by your kernel?

행렬 곱셈에서 출력 행렬 C의 각 원소는 A의 한 행과 B의 한 열의 모든 원소를 곱하고 그 결과를 모두 더해서 계산된다.

따라서 한 원소를 계산할 때마다 곱셈 연산과 덧셈 연산이 2회 수행된다.

각 부동소수점 연산(Floating-point operation)으로 본다면, 각 결과 원소당 2 회의 연산이 수행된다.

따라서 전체 연산 수는 A의 행 개수, A의 열 개수, B의 열 개수를 모두 곱한 값에 2를 곱한 값이 된다.

FLOPs = 2 × (numARows) × (numAColumns) × (numBColumns)

(2) How many global memory reads are being performed by your kernel?

본 과제에서 구현한 커널은 shared memory tiling 기법을 사용하여 global memory 접근 횟수를 최소화한다.

각 thread block 은 A 와 B 의 일부 데이터를 global memory 에서 한 번씩 읽어 shared memory 에 저장한다.

그 후 shared memory 내에서 여러 thread 가 데이터를 재사용하므로, 같은 원소를 여러 번 읽을 필요가 없다.

따라서 전체적으로 보면 행렬 A 와 행렬 B 의 모든 원소가 각각 한 번씩 global memory 에서 읽히게 된다.

Global Reads = (numARows × numAColumns) + (numBRows × numBColumns)

(3) How many global memory writes are being performed by your kernel?

출력 행렬 C 의 각 원소는 하나의 thread 가 계산을 담당하며, 모든 계산이 끝난 후 global memory 에 한 번만 기록된다.

즉, C 의 원소 개수만큼 global memory 에 쓰기 연산이 발생한다.

이 과정에서는 중복 기록이나 추가적인 write 연산이 존재하지 않는다.

Global Writes = (numCRows × numCColumns)

(4) Describe what further optimizations can be implemented to your kernel to achieve a performance speedup

.

전역 메모리 접근을 `float4` 등의 형태로 vectorization 하여 대역폭을 활용한 향상이 가능하다.

CUDA 의 WMMA(텐서 코어)나 curated 라이브러리(cuBLAS)등을 활용하며 하드웨어 행렬 연산 유닛을 사용 가능하다.

또한, concurrency 를 확보하기 위해 `cudaMemcpyAsync`와 스트림을 이용해 데이터 전송 겹치기 또는 Occupancy 튜닝으로 SM 활용률 개선 등이 가능하다.

(5) Explanation about your version of template.cu

5-1. Allocate device memory

먼저, GPU 에서 연산을 수행하기 위해 입력 행렬 A 와 B, 그리고 결과 행렬 C 에 해당하는 device 메모리를 동적으로 할당하였다.

각 행렬의 크기를 계산하여 그에 맞는 float 단위의 메모리를 확보하였으며,

모든 CUDA 메모리 관련 함수 호출에 대해 오류 검사를 수행하여 안정성을 보장하였다.

이 단계는 GPU 연산을 위한 초기 환경을 설정하는 부분으로, 이후 단계에서 필요한 데이터 전송과 커널 실행을 위한 기반을 마련한다.

5-2. Copy host memory to device

다음으로, host 메모리에 저장된 입력 행렬 A 와 B 를 GPU 메모리로 복사하였다.

이 과정을 통해 커널 실행 시 GPU 가 직접 데이터를 참조할 수 있게 되며,

데이터 전송 방향은 Host → Device 로 수행된다.

이 단계는 커널이 연산에 사용할 데이터를 준비하는 필수적인 사전 작업으로,

모든 입력 데이터가 GPU 에 상주하게 된다.

5-3. Initialize thread block and grid dimensions

커널 실행을 위해 thread block 과 grid 의 차원을 초기화하였다.

각 block 은 타일 단위 계산을 담당하도록 TILE_WIDTH × TILE_WIDTH 크기로 설정하였으며,

전체 grid 는 출력 행렬 C 의 크기에 맞춰 block 들이 겹침 없이 전체 영역을 덮도록 구성하였다.

이를 통해 각 thread 는 C 의 한 원소를 담당하며,

모든 연산이 병렬적으로 수행될 수 있도록 효율적인 grid 구조를 설계하였다.

5-4. Invoke CUDA kernel

설정된 grid 와 block 차원을 기반으로 행렬 곱셈을 수행하는 커널을 실행하였다.

커널 호출 시 입력 행렬 A, B, 그리고 출력 행렬 C 의 메모리 포인터와 크기 정보를 함께 전달하였다.

실행 직후 오류 검사를 수행하여 런타임 문제를 방지하였으며,

커널의 완료를 보장하기 위해 동기화 과정을 추가하였다.

이 단계는 전체 GPU 계산의 핵심 실행 부분으로, GPU 가 실제로 행렬 곱 연산을 수행하는 단계이다.

5-5. Copy results from device to host

커널 실행이 완료된 후, GPU 메모리에 저장된 결과 행렬 C 를 다시 host 메모리로 복사하였다.

이 과정을 통해 CPU 측에서 결과를 검증할 수 있으며,

정확한 연산 결과가 host 환경에서도 확인 가능하다.

결과 복사 과정 또한 오류 검사를 포함하여 안전하게 수행되도록 구성하였다.

5-6. Free device memory

결과 복사가 끝난 뒤에는 연산 과정에서 사용된 모든 GPU 메모리를 해제하였다.

A, B, C 각각에 대해 cudaFree()를 호출함으로써 불필요한 자원 점유를 방지하였다.

이는 GPU 리소스 관리 측면에서 필수적인 절차로, 프로그램이 종료될 때 메모리 누수가 발생하지 않도록 보장한다.

5-7. Write the CUDA kernel

마지막으로, shared memory 를 활용한 타일 기반 행렬 곱셈 커널을 구현하였다.

각 thread block 은 A 와 B 의 일부 데이터를 shared memory 에 로드하고,

모든 thread 가 데이터를 불러온 후 동기화를 수행하여 연산의 일관성을 유지한다.

이후 각 tile 단위로 부분 곱셈을 반복 수행하며,

누적된 결과를 최종적으로 C 의 대응 위치에 기록하도록 설계하였다.

이 커널은 shared memory 를 활용함으로써 global memory 접근 횟수를 크게 줄였고,

동시에 thread 간의 협력 계산을 통해 메모리 대역폭을 효율적으로 활용하였다.

이 단계는 본 과제의 핵심 목표인 “tiled matrix multiplication using shared memory”를 실질적으로 구현한 부분이다.

(6) 시간 측정

서버 환경

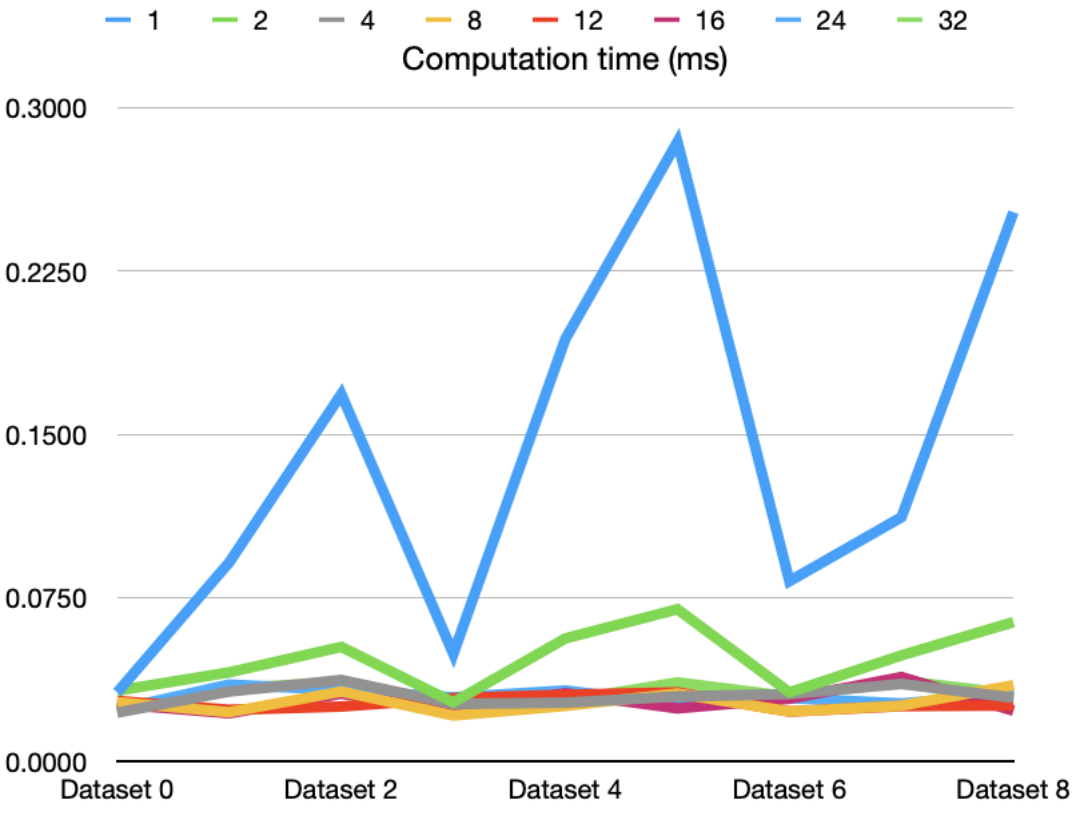
- **GPU: NVIDIA TITAN RTX**
- **메모리 용량: 24 576 MiB (24 GB)**
- **파티션: class 3**
- **할당 노드: a08**
- **CPU 프로세서: Intel(R) Xeon(R) Silver 4216 @ 2.10 GHz**

실행결과

- **GPU 환경 상태**: NVIDIA TITAN RTX, 온도 27 °C, 메모리 24 576 MiB 중 사용 0 MiB, 활용도 0 %로 즉시 사용 가능.
- **SLURM 실행 정보**: 파티션 class3, 노드 a08, 1 분 잡 요청 시 바로 할당됨. 주요 SLURM_* 환경변수 확인 완료.

- **CPU 자원**: Intel Xeon Silver 4216 듀얼 소켓(총 64 vCPU), NUMA 2 노드 구성, AVX512 등 최신 ISA 지원.
- **시스템 메모리**: 총 472 GiB, 사용 11 GiB, 여유 243 GiB, 버퍼/캐시 217 GiB.

Tile width	Dataset 0	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Dataset 6	Dataset 7	Dataset 8
1	0.0314	0.0911	0.1684	0.0491	0.1938	0.2843	0.0825	0.1119	0.2520
2	0.0322	0.0406	0.0523	0.0268	0.0562	0.0696	0.0314	0.0484	0.0637
4	0.0221	0.0318	0.0371	0.0261	0.0267	0.0295	0.0306	0.0353	0.0289
8	0.0269	0.0223	0.0318	0.0208	0.0252	0.0309	0.0226	0.0253	0.0347
12	0.0275	0.0234	0.0249	0.0287	0.0303	0.0314	0.0226	0.0251	0.0254
16	0.0260	0.0218	0.0308	0.0227	0.0310	0.0241	0.0285	0.0385	0.0230
24	0.0235	0.0350	0.0325	0.0291	0.0323	0.0257	0.0288	0.0263	0.0308
32	0.0226	0.0333	0.0361	0.0246	0.0275	0.0360	0.0295	0.0368	0.0306



해당 결과물을 보며 눈여겨볼 수 있는 지점인 tile width 가 1 인 시점에서 특정 데이터셋의 computation time 이 갑자기 줄어든 것처럼 보이는 지점이 있다.

dataset 2 – dataset 4 의 지점을 보면 dataset 3 에서 갑자기 time 이 감소하는 것을 볼 수 있는데 실제 각 데이터셋의 FLOPs 를 확인해 보면

Dataset 3: A: 112×48, B: 48×16 → 총 연산량 ≈ 112×16×48×2 = 172,032 FLOPs

Dataset 2: A: 64×128, B: 128×64 → 총 연산량 ≈ 64×64×128×2 = 1,048,576 FLOPs

Dataset 4: A: 84×84, B: 84×84 → 총 연산량 ≈ 84×84×84×2 = 1,181,184 FLOPs

의 형태로 Dataset 3 의 실제 수 및 연산량이 매우 작음을 알 수 있다.

그 격차가 드라마틱해 보이는 더 큰 이유는 tiling width 1 의 경우 shared memory 최적화가 전혀 적용되지 않은 형태이기 때문에 단순히 FLOPs 에 비례하는 시간을 갖기 때문이라고 해석 가능하다.

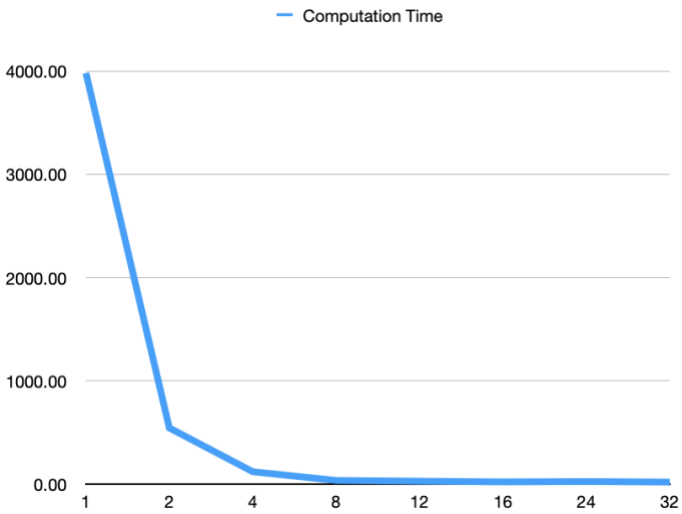
(7) data_large 를 사용한 측정 (2,4,8,12,16,24,32)

서버 환경

- **GPU: NVIDIA TITAN RTX**
- **메모리 용량: 24 576 MiB (24 GB)**
- **파티션: class 3**
- **할당 노드: a08**
- **CPU 프로세서: Intel(R) Xeon(R) Silver 4216 @ 2.10 GHz**

Tile width	Computation Time (ms)
1	3979.51
2	544.76
4	121.31
8	36.68

12	29.32
16	22.60
24	26.14
32	20.75



전체적으로 tile 의 사이즈가 커짐에 따라 computation time 이 지수적으로 빠르게 감소한다. 이는 중복 사용되는 메모리를 tiling 기법을 통해 shared memory 에 미리 넣어둠에 따라 불필요한 global memory access 가 줄어들었음을 의미한다. 그러나 tile width 가 작은 시점에서는 band width 에 따른 memory-bound 로 인해 성능이 낮았지만 tile width 가 충분히 확보된 이후로는 comput-bound 로 넘어간 시점으로 해석 가능하다.

특이한 지점은 tile width 가 24 인 시점에서 tile width 를 16 으로 둔 경우보다 더 computation time 이 오래 소모된다는 점인데 이의 경우 여러 관점에서 해석이 가능하다.

우선 본 과제에서 분석한 데이터셋의 dimension 의 경우 Matrix A = (4096, 8000)이고 Matrix B = (8000, 512)의 사이즈를 가지는데 tile width 16 과 32 는 모두 정수배로 나누어 떨어지지만 tile width 24 의 경우 나누었을 때 정수배로 떨어지지 않는다.

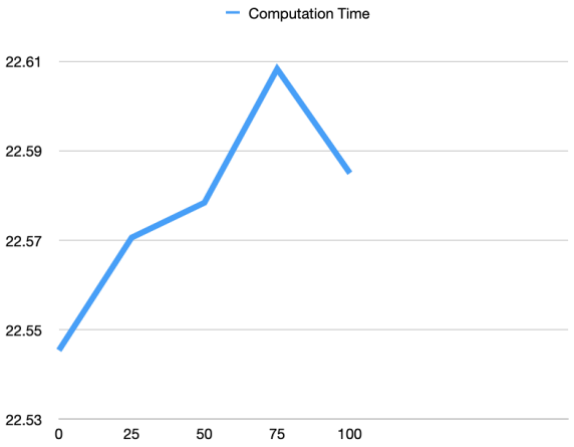
이는 tiling 과정에서 필연적으로 boundary 지점에서 idle thread 가 발생하거나 warp 최적화 효율이 떨어질 가능성을 제공한다. 또한, shared memory 또는 cache 에서 24 라는 불규칙한 단위로 접근할 경우 한번에 가져오는 데이터의 길이가 2 의 배수로 떨어지지 않아 불필요하게 빈 공간을 추가로 할당받는 문제도 발생 가능하다.

(8) 16 width 를 사용하고 shared memory capacity 를 0, 25, 50, 75, 100 에 맞게 측정

서버 환경

- **GPU: NVIDIA TITAN RTX**
- **메모리 용량: 24 576 MiB (24 GB)**
- **파티션: class 3**
- **할당 노드: a08**
- **CPU 프로세서: Intel(R) Xeon(R) Silver 4216 @ 2.10 GHz**

Shared Memory capacity (%)	Computation Time
0	22.55
25	22.57
50	22.58
75	22.61
100	22.59



실제 결과를 봤을 때 그래프를 도시하는 것이 가능하긴 하나, 구체적인 연산 시간은 큰 차이를 가지지 않는다. 이는 shared memory 를 caravout 을 통해 조정하는 것이 본 과제에서는 연산 시간에 크게 영향을 미치지 않았음을 의미한다.

과제에서 사용한 caravout 의 경우 SM 의 L1 cache / shared memory 파티션을 얼마나 공유 메모리 쪽으로 선호할지 요청하는 것으로,

Caravout 이 0 일 경우 L1 캐시를 최대한 활용하여 global memory access 에 중점을 두고 Caravout 이 100 일 경우 SM 에서 shared memory 를 최대한 활용하는 방향으로 최적화를 진행한다.

실제 tile width 를 16 으로 설정할 경우 모델이 float matrix 2 개를 탑재하므로 2 x (16 x 16 x 4B) = 2KB 만 로드하게 되는데 caravout 이 0 일 경우라도 기본적으로 SM 이 가지는 shared memory 를 초과하지 않는 작은 크기의 타일이기 때문에 제약에 걸리지 않게 된다.

또한, 본 과제에서 실험에 활용하는 matrix 는 2 차원의 형태로 존재하기 때문에 CUDA 블록당 최대 쓰레드 1024 에 맞는 tile width 32 로도 해당 shared memory 에 모두 로드하는 것이 어렵지 않다고 판단된다.

4. Conclusion

여러 실험을 하며 Tiling의 중요성을 깨달을 수 있었다.

Tiling을 통해 shared memory을 이용해 memory band width를 효율적으로 사용할 경우 본

과제의 경우 최대 200배에 가까운 성능 향상을 얻을 수 있었고 이는 기존의 Tile width 1으로 설정된 단순 연산의 경우 roof-line 기준 memory-bound에 크게 묶여 있었다는 것을 알 수 있다.

이를 통해 매우 좋은 Peak Flops를 가진 GPU를 사용하더라도 연산 성능을 모두 사용할 수 있도록 충분한 메모리를 공급하지 않으면 그 최대 성능을 활용하지 못하고 낭비할 수 있다는 것을 알 수 있다.