과제 #5

M1522.006700 확장형 고성능 컴퓨팅 (001)
M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

박찬정
서울대학교 전기정보공학부
2023-24013

1 GPU 정보 확인하기

(a) 주어진 명령을 수행하여 Fig. 1과 같은 출력을 얻었다. 긴 출력의 경우 앞부분만 첨부하였다.

첫 번째 명령어는 4개의 GPU를 할당받아 `nvidia-smi`를 실행한 결과이다. `nvidia-smi`는 NVIDIA System Management Interface로, NVIDIA GPU를 관리 및 모니터링하는 도구이다. Fig. 1a에서 볼 수 있듯이, 4개의 GPU가 할당되었고 각 GPU의 상태를 확인할 수 있다.

두 번째 명령어는 4개의 GPU를 할당받아 `nvidia-smi -q`를 실행한 결과이다. `-q` 옵션은 query 명령어로, 각 GPU의 상태를 더 자세히 확인할 수 있다. Fig. 1b에서 볼 수 있듯이, 4개의 GPU가 할당되었고 각 GPU의 세부 정보를 제공하고 있다. 굉장히 긴 출력이므로, 여기서는 앞 부분만 첨부하였다.

세 번째 명령어는 4개의 GPU를 할당받아 `clinfo`를 실행한 결과이다. `clinfo`는 OpenCL을 지원하는 플랫폼 및 디바이스의 정보를 출력하는 도구이다. Fig. 1c에서 볼 수 있듯이, CUDA 플랫폼을 인식하고 있으며 4개의 GPU가 할당되었음과 각 GPU의 세부 정보를 확인할 수 있다. 마찬가지로 굉장히 긴 출력이므로, 여기서는 앞 부분만 첨부하였다.

(b) 이전 명령어의 출력들로부터 계산 노드에 장착된 GPU의 모델이 NVIDIA TITAN RTX임을 확인할 수 있다.

(c) `nvidia-smi`의 출력을 통해 GPU의 global memory가 24576 MiB임을 확인할 수 있다. 또한 `clinfo`의 출력을 통해 더 자세한 global memory의 크기를 알 수 있는데, 25396969472 bytes임을 확인할 수 있다. 이는 앞서 확인한 24576 MiB에 근사한 수치이다.

(d) GPU의 maximum power limit은 `nvidia-smi`의 출력을 통해 280 W임을 확인할 수 있다. 또한 maximum SM clock speed는 `nvidia-smi -q`의 출력을 통해 2100 MHz임을 확인할 수 있다.

(e) `clinfo`의 출력을 통해 Max work item dimension, Max work item size, Max work group size가 각각 3, $(1024 \times 1024 \times 64)$, 1024임을 확인할 수 있다.

2 Matrix Multiplication with OpenCL

- **병렬화 방식.** 주어진 행렬곱 연산은 우선 워크로드 단위로 쪼개진다. 출력 행렬 `C`는 미리 설정된 `TSM`, `TSN` 값에 따라 타일 단위로 쪼개져 각 워크 그룹에 할당된다. 각 타일은 한번 더, 미리 설정된 `WPTM`, `WPTN` 값에 따라 쪼개져 각 쓰레드에 할당된다. 연산 수행시에는 `float4` 자료형을 사용하여 벡터화된 데이터를 사용하여 여러 데이터에 대한 연산을 한 번에 수행한다. 또한, 입력 행렬 `A`와 `B`를 로컬 메모리에 캐싱하는 과정도 각 쓰레드마다 영역을 지정함으로써 병렬적으로 수행하도록 구현하였다.

- **`matmul.c`의 각 부분 설명.**

  `matmul_initialize` 함수에서는 다음과 같은 API가 사용되었다.

  - `clGetPlatformIDs`: 현재 시스템에서 사용 가능한 OpenCL 플랫폼의 ID를 얻는다.

- `clGetDeviceIDs`: 선택한 OpenCL 플랫폼에 대한 디바이스 ID를 얻는다.
- `clCreateContext`: OpenCL 컨텍스트를 생성한다.
- `clCreateCommandQueue`: 컨텍스트에 대한 커맨드 큐를 생성한다.
- `clCreateKernel`: 특정 프로그램으로부터 커널을 생성한다.
- `clCreateBuffer`: 버퍼 객체를 생성하여 메모리를 할당한다.

`matmul` 함수에서는 다음과 같은 API가 사용되었다.

- `clEnqueueWriteBuffer`: 호스트에서 디바이스로 데이터를 복사한다.
- `clSetKernelArg`: 커널에 전달할 인자를 설정한다.
- `clEnqueueNDRangeKernel`: 커널을 실행하기 위해 큐에 명령을 추가한다.
- `clEnqueueReadBuffer`: 디바이스에서 호스트로 결과 데이터를 복사한다.

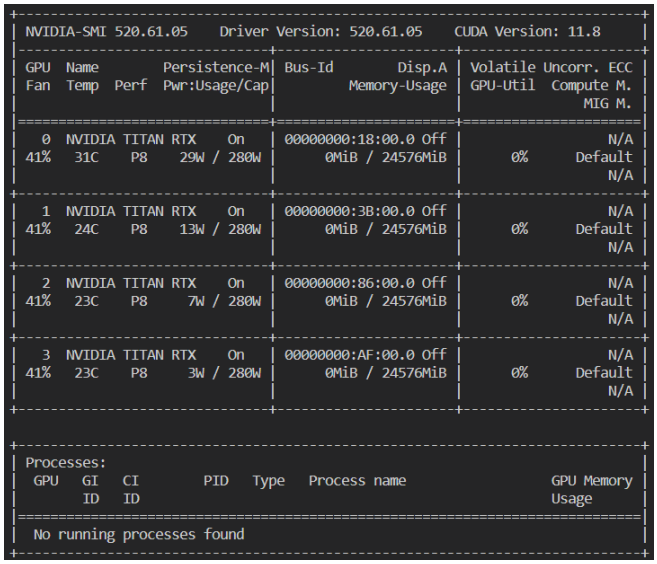
`matmul_finalize` 함수에서는 다음과 같은 API가 사용되었다.

- `clReleaseMemObject`: 할당된 메모리 객체를 해제한다.
- `clReleaseKernel`: 커널 객체를 해제한다.
- `clReleaseProgram`: OpenCL 프로그램을 해제한다.
- `clReleaseCommandQueue`: 커맨드 큐를 해제한다.
- `clReleaseContext`: OpenCL 컨텍스트를 해제한다.

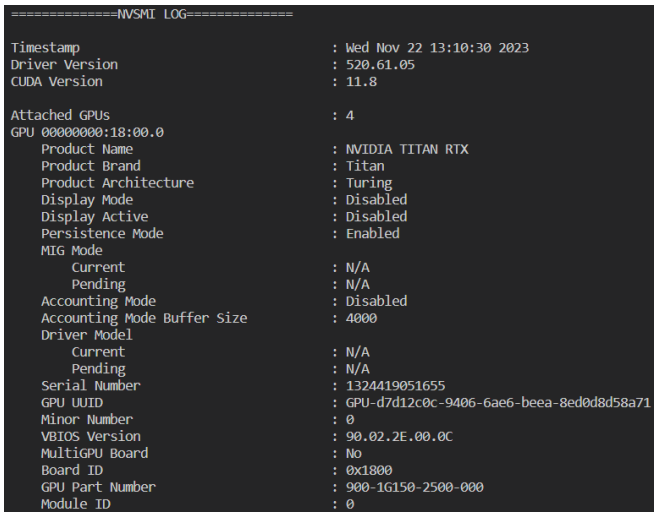
- **최적화 방식.**

  다음과 같은 순서로 최적화하였다.

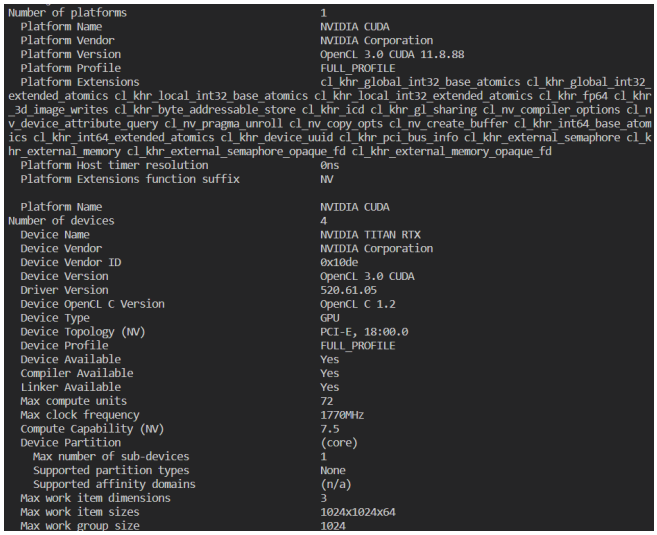
  - Naive 커널 구현: 우선, 매우 단순한 방식으로 행렬곱을 수행하는 커널을 구현하였다. 이 커널은 매우 단순하게 작동하는 대신 임의의 크기의 입력 행렬에 대해 항상 연산을 수행할 수 있다. 이 커널은 `run_performance.sh`에 대해 대략 130 GFLOPS의 성능을 보였다.
  - 조건부 가속: 연산 커널을 수행하기 전 입력 차원을 확인하여, 가속에 용이한 입력 차원이라면 앞으로 구현할 특수 커널을 사용하고, 그렇지 않다면 naive 커널을 사용하도록 구현하였다.
  - 타일링 및 로컬 메모리 사용: 타일의 크기를 지정하여 각 워크 그룹이 하나의 타일을 맡아 연산하도록 구현하였다. 이 때 구현의 편의를 위해 타일의 형태는 정사각형으로 설정하였다. 또한 이 과정에서 입력 행렬을 로컬 메모리에 캐시하여 연산에 사용하도록 하였다. 이를 통해 메모리 접근을 줄이고, 캐시 히트를 높이는 효과를 얻었다. 이 커널은 `run_performance.sh`에 대해 대략 560 GFLOPS의 성능을 보였다.
  - 쓰레드의 담당 영역 설정: 각 쓰레드가 타일의 여러 원소를 담당하도록 구현하였다. 이를 통해 로컬 메모리의 연속된 메모리 영역이 하나의 쓰레드에 의해 사용되도록 하였다. 이 커널은 `run_performance.sh`에 대해 대략 1,000 GFLOPS의 성능을 보였다.
  - 벡터 자료형 사용: 입력 행렬 및 로컬 메모리의 자료형을 `float4`로 설정하여 메모리 접근 및 연산을 벡터화하였다. 이 커널은 `run_performance.sh`에 대해 대략 1,713 GFLOPS의 성능을 보였다.
  - 직사각형 타일링: 타일의 크기를 더욱 최적화하기 위해 직사각형 형태의 타일을 사용하도록 구현하였다. 이 과정에서 행렬 `A`, `B`, `C`의 타일의 크기가 모두 다르게 설정되었다. 이 커널은 `run_performance.sh`에 대해 대략 1,810 GFLOPS의 성능을 보였다.
  - 뱅크 컨플릭트 회피: 로컬 메모리의 행 크기에 크기 2의 패딩을 추가하여 뱅크 컨플릭트를 회피하였다. 이 커널은 `run_performance.sh`에 대해 최고 2,010 GFLOPS의 성능을 보였다.
  - 레지스터 타일링: 연산 과정에서 로컬 메모리에서 레지스터로 데이터를 로드하는 과정을 최적화하기 위해 레지스터에 타일의 데이터를 캐시하도록 구현하였다. 이 커널은 `run_performance.sh`에 대해 최고 2,250 GFLOPS의 성능을 보였다.



(a) `srun --partition=class1 --gres=gpu:4 nvidia-smi` 커맨드 실행 결과.



(b) `srun --partition=class1 --gres=gpu:4 nvidia-smi -q` 커맨드 실행 결과.



(c) `srun --partition=class1 --gres=gpu:4 clinfo` 커맨드 실행 결과.

Figure 1: 각종 커맨드를 실행한 결과.