과제 #6

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

박찬정
서울대학교 전기정보공학부
2023-24013

1 Matrix Multiplication Challenge

- **병렬화 방식.** 행렬 A가 M=65536으로 큰 편이므로, 행렬 A를 행 단위로 나누어 노드 및 GPU 단위로 병렬화하였다. 행렬 B는 모든 노드 및 GPU에게 동일하게 분배하도록 하였다. 노드간의 통신은 MPI를 사용하였고, GPU로의 통신은 stream을 GPU의 수에 맞추어 생성하여 사용하였다.

- **성능 최적화 방법 및 고려 사항.** 성능 최적화를 위해 다음과 같은 방법을 적용하였다. 노드간 병렬화, 프로세서간 병렬화, 글로벌 메모리 coalescing, 공유 메모리 캐시 블로킹, work per thread 증가, OpenMP를 사용한 GPU 통신 병렬화, 메모리 접근 vectorize, hyperparameter 튜닝.

  이러한 기법을 사용하는 과정에서 매우 많은 hyperparameter를 설정해야 했고, 일차적으로는 GPU 구조에 대한 지식을 바탕으로 설정한 뒤 실험을 통해 최적의 hyperparameter를 찾아내었다. 각 방법에 대한 자세한 설명 및 적용 효과는 마지막 문항에서 더 자세히 다룬다.

- **`matmul.c`의 각 부분 설명.**

  `matmul_initialize` 함수에서는 다음과 같은 CUDA API를 사용하였다.

  - `cudaGetDeviceCount`: 사용 가능한 GPU의 수를 가져온다.
  - `cudaGetDeviceProperties`: GPU의 정보를 가져온다.
  - `cudaStreamCreate`: GPU stream을 생성한다.
  - `cudaEventCreate`: CUDA event를 생성한다.
  - `cudaMalloc`: GPU 메모리를 할당한다.

  `matmul` 함수에서는 다음과 같은 CUDA API를 사용하였다.

  - `cudaMemcpyAsync`: GPU 메모리에 데이터를 복사한다.
  - `cudaEventRecord`: CUDA event를 기록한다.
  - `cudaStreamWaitEvent`: CUDA event를 기다린다.
  - `cudaGetLastError`: CUDA error를 가져온다.
  - `cudaStreamSynchronize`: GPU stream을 동기화한다.

  `matmul_finalize` 함수에서는 다음과 같은 CUDA API를 사용하였다.

  - `cudaFree`: GPU 메모리를 해제한다.
  - `cudaStreamDestroy`: GPU stream을 해제한다.
  - `cudaEventDestroy`: CUDA event를 해제한다.

- **OpenCL vs. CUDA.** CUDA의 장점으로는 손쉬운 개발 환경 및 높은 성능을 들 수 있다. 이번 과제에서 최적화를 위해 사용한 기법들은 대부분 이전 과제에서 OpenCL을 사용할 때에도 적용한 것들인데, 같은 기법을 적용하였을 때의 성능은 CUDA가 훨씬 더 좋았다. 그 원인으로 추측되는 것은,

OpenCL은 GPU의 구조를 추상화하여 사용하는 반면 CUDA는 직접적으로 다룰 수 있기 때문이라고 생각한다. 이러한 하드웨어와 프로그래머 사이의 좁은 거리가 CUDA의 장점이라고 생각한다.

OpenCL의 장점으로는, CUDA와 달리 다양한 GPU에서 사용할 수 있다는 점이 있다. 개인적으로 모바일 환경에서의 컴퓨팅에 관심을 가지고 있는데, 이 경우 기기마다 제각각의 환경을 가지고 있고 NVIDIA와는 무관한 환경이 대부분이기 때문에 이 경우 거의 모든 상황에 CUDA가 아닌 OpenCL을 사용하게 될 것이다. 이러한 범용성이 OpenCL의 장점이라고 생각한다.

- **최적화 방식에 따른 성능 비교.** 다음과 같은 순서로 최적화 방식을 적용하였고, 각 단계에서의 성능을 측정하였다. 성능 측정은 a00 노드에서 수행하였다.

  1. **Baseline.** 주어진 과제 코드를 그대로 실행하였다. 이 코드에서는 전체 workload를 단일 노드에서 실행하며, 한 노드에서는 4개의 GPU에 같은 크기의 workload를 분배하였다. 성능은 약 3400 GFLOPS였다.
  2. **MPI.** 전체 workload를 4개 노드에게 동일하게 분배하도록 하였다. 행렬 A는 `MPI_Scatter`를 사용하여 각 노드에게 분배하였고, 행렬 B는 `MPI_Bcast`를 사용하여 각 노드에게 동일하게 분배하였다. 성능은 약 7500 GFLOPS였다.
  3. **GMEM Coalescing.** 행렬곱 커널에서 블록 개념을 적용하여, 같은 warp 안의 쓰레드가 같은 글로벌 메모리에 접근하는 경우를 최대화하도록 하였다. 성능은 약 8650 GFLOPS였다.
  4. **SMEM Cache Blocking.** 공유 메모리를 사용하여, 같은 블록 안의 쓰레드가 행렬 A와 B의 같은 영역을 공유 메모리에 저장하도록 하였다. 성능은 약 9070 GFLOPS였다.
  5. **Work per Thread.** 각 쓰레드가 행렬 C의 하나의 원소를 계산하는 대신, 하나의 쓰레드가 행렬 C의 인접한 여러 원소를 계산하도록 하였다. 이 과정에서 블록의 형태를 정사각형에서 임의의 직사각형 형태가 되도록 수정하였다. 성능은 약 9237 GFLOPS였다.
  6. **Work per Thread(2D).** 각 쓰레드가 행렬 C의 일정 크기의 블록을 담당하여 계산하도록 하였다. 성능은 약 9863 GFLOPS였다.
  7. **Using Only 1 Node.** 행렬곱의 latency를 각 파트별로 측정한 결과, 총 0.222934 ms 중 행렬 A 분배에 0.080922 ms 로 36.3%, 행렬 B 분배에 0.015000 ms 로 6.7%, 행렬 C 합치기에 0.062237 ms 로 27.9 %가 소요되었다. 즉 MPI에 의한 통신이 전체 시간의 70.9 %를 차지하였다. 이를 해결하기 위해서는 root 노드가 아닌 노드에 분배하는 workload의 양을 적절히 작게 설정하여 통신의 비중을 줄여야 한다. 다만 문제를 더 단순화하기 위해, 이번 과제에서는 노드간의 통신을 제거하고, 한 노드에서만 실행하도록 하였다. 성능은 약 14000 GFLOPS였다.
  8. **OpenMP & Concurrent Streams.** Host와 device 사이의 데이터 전송을 담당하는 스트림과 kernel의 실행을 담당하는 스트림을 분리하였다. 이들 사이의 동기화는 event를 사용하여 수행하였다. 또한, 각 GPU에서의 스트림 관리를 병렬화하기 위해 OpenMP를 사용하였다. 성능은 약 16022 GFLOPS였다.
  9. **Vectorize.** 공유 메모리 접근을 최적화하기 위해 행렬 A를 transpose하여 공유 메모리에 저장하였다. 또한, 글로벌 메모리와 공유 메모리에 접근하는 부분을 vectorize하였다. 성능은 약 18660 GFLOPS였다.
  10. **Hyperparameter Tuning.** 이전 단계에서 적용한 기법들은 여러 hyperparameter를 요구하였다. 이를 바꾸고 성능을 측정하는 실험을 반복적으로 수행하여 성능을 최대화하도록 하였다. 이를 통해 얻은 hyperparameter는 다음과 같다.
      - `BLOCK_M=128`: 한 블록이 담당하는 행렬 C의 행의 수.
      - `BLOCK_N=128`: 한 블록이 담당하는 행렬 C의 열의 수.
      - `BLOCK_K=8`: 한 블록이 담당하는 행렬 A의 열, 행렬 B의 행의 수.
      - `THREAD_M=16`: 한 쓰레드가 담당하는 행렬 C의 행의 수.
      - `TRHEAD_N=8`: 한 쓰레드가 담당하는 행렬 C의 열의 수.

      성능은 약 22004 GFLOPS였다.