M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Homework #6

Due : 2024/12/09 (Mon)

2024-28413 이정현

1. Matrix Multiplication Challenge

-병렬화 방식에 대한 설명

[Kernel]

Kernel의 기본 틀은 앞서 진행했던 multiple GPU를 사용하는 5번 과제에서 구현한 것을 이용하였다. 이 kernel에는 shared memory, tiling, 그리고 하나의 thread당 여러개의 element 연산 담당 이라는 등등의 기본적이면서도 중요한 최적화가 매우 효율적으로 적용되어 있었다.

이번 과제에서는 20000 GFLOPS를 위해서 kernel에 두가지 최적화를 추가하였다.

(1) Vectorized memory access

Float4 데이터 타입을 사용해서 vectorized memory operation을 실행할 수 있도록 코드를 변형하였다. 이는 각 thread가 4개의 float를 동시에 load/store할 수 있기 때문에 매우 효율적이다.

이와 함께 조금 더 성능 극대화를 위해서 추가한 몇 가지를 언급하자면,

- __restrict__ pointer 사용 : 컴파일러가 A,B,C행렬의 pointer alias 없다고 가정하게 함
- 행렬 크기의 약수이자 4의 배수인 Block_size를 이용했고, 모든 불필요한 boundary check를 없앴다

(2) Loop Unrolling

#pragma unroll 코드를 for loop이전에 작성함으로써, 컴파일러가 loop unroll을 실행할 수 있게 해주었고, instruction-level parallelism을 극대화 하였다.

[GPU-Host 통신 설계]

이 부분에서 가장 중요했던 최적화 방법은 하나의 GPU당 두개의 cuda stream을 사용하는 것이었다. 두개를 사용함으로써, data transfer를 kernel execution과 overlapping하는 것이 가능했고, 이는 idle time을 줄이고 GPU utilization을 극대화할 수 있다는 큰 장점이 있다. (하나의 stream이 data transfer를 담당하고 있을때, 다른 stream은 kernel execution 혹은 host로 데이터를 가져오는 부분을 수행할 수 있다)

오류 없이 이를 구현하기 위해서 각 stream마다 device memory buffer를 할당해주었다 (thread safety와 data race 방지). 그리고 synchronization을 이용해서 모든 data transfer와 kernel execution이 끝날때까지 코드가 기다리도록 하였다.

-성능 최적화를 위한 적용한 방법 및 고려 사항들 논의

이번 과제에서 가장 크게 고려했던 부분은 GPU 자원을 얼마나 사용할 것인지에 대한 설계였다. 과제를 시작하는 시점에서는 많은 GPU를 사용하면 무조건 더 성능이 좋게 나올 것이라고 생각했었다. 그러나 실제 구현과 실험을 통해서 몇 가지 이유들로 성능이 GPU개수에 무조건적으로 비례하지 않는다는 점을 깨달았다. 아래는 대표적으로 두 가지 이유를 정리하였다.

1. Data transfer overhead

이번 과제에서는, 효율적이라고 알려진 MPI_Iscatterv, MPI_Igatherv 같은 방법들을 적용했을때에도 노드 간 통신 overhead가 너무 큰 비중을 차지하였다. 행렬의 크기도 컸고 (특히 B행렬은 행렬 곱셈 상 항상 통째로 필요했기 때문에 분할하지 못하고 전체를 전달해야만 했어야하는 점), 행렬 곱셈은 scatter와 gather 통신이 무거운 workload로 분류될 수 있기에 4개의 노드를 사용했을 때 얻는 computational efficiency가 data transfer overhead에 가려진다는 결론을 내리게 되었다.

2. Synchronization overhead

노드의 개수가 늘어날수록 consistency를 위해 동기화 작업의 중요성이 높아지는데, multi-node 환경에서는 barrier와 같은 synchronization cost가 상승한다. 이는 프로세스의 개수가 많아짐에 따라 관리해야 할 대상이 많아지기 때문이다. 또한, 실습 서버에서 노드 간의 성능 차이가 다소 있다는 점을 확인하였는데, 성능이 좋은 노드는 계산을 빠르게 마치고 성능이 느린 노드를 기다리는 상황이 발생할 수도 있기 때문에 노드를 많이 사용하는 것이 항상 더 좋지는 않았다.

위에 정리한 두가지 이유들을 바탕으로, 이번 과제에서는 하나의 노드(4개 GPU)만을 사용해서 행렬 곱셈을 최적화 하였다.

마지막으로, 하나의 노드만 사용해서 구현하였기 때문에 성능 측정시 더 빠른 자원 할당을 받기 위해 run.sh 파일안에 있는 노드의 개수를 1개로 바꾸고 측정하였다. (: ${NODES:=1})

-matmul.cu의 각 부분에 대한 설명

(1) matmul_initialize에서 사용된 API List

-cudaGetDeviceCount : 사용할 수 있는 GPU의 개수를 구한다

-cudaGetDeviceProperties : GPU 정보를 알려준다

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaMalloc : GPU에 memory를 할당한다

-cudaStreamCreate : GPU stream을 생성한다

(2) matmul에서 사용된 API List

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaMemcpyAsync : host와 device 사이의 memory copy를 비동기적으로 수행해준다

-cudaStreamSynchronize : 지정된 stream의 모든 작업이 완료될때까지 CPU 대기시킨다

(3) matmul_finalize에서 사용된 API List

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaFree : GPU memory를 해제해준다

-cudeStreamDestroy : GPU stream을 해제한다

-최적화 방식의 분류 및 각각에 대한 성능 실험 결과

이 부분에서 측정된 성능은 모두 **run.sh에 M=N=K=4096** 옵션을 주고 측정되었다.

- 4개의 Node 사용, MPI communication 사용 **(6160 GFLOPS)**

  Kernel은 Hw5 - Multi GPU에 사용되었던 동일 kernel 사용하였다.

  Thread가 하나의 원소를 계산하는 것이 아닌, 4x4 크기의 sub-block 계산을 담당하도록 구현하였다. 앞서 다뤘던 것 처럼, thread당 연산 양을 늘려주면 데이터의 재사용이 늘어나고 한 번의 memory access당 더 많은 연산이 일어나기 때문에 전체 성능에서 memory latency의 영향을 상대적으로 감소시켜주는 효과가 있다. GPU의 병렬 연산 특성을 효율적으로 다룰 수 있는 방법이기 때문에 상당한 성능 향상을 목격할 수 있었다. (BLOCK_SIZE = 64)

  위와 같은 최적화가 적용되어있는 kernel을 사용하였고, 총 4개의 MPI 프로세스를 생성해서 GPU하나당 배정하였다. Host에서 GPU로의 데이터 분배와 수거는 MPI_Scatter, MPI_Gather를 사용하였다.

- MPI communication 최적화 **(6390 ~6700 GFLOPS)**

  Blocking operation인 scatter와 gather 대신 non-blocking operation 인 MPI_Iscatter, MPI_Igather를 사용하였다. 동기화를 위해 MPI_Wait을 함께 사용하였다. 처음보다는 성능 향상을 목격할 수 있었다.

- 1개의 노드 사용, GPU 당 2개의 cuda stream **(14600 - 16580 GFLOPS)**

  행렬 곱셈을 최적화 하는 과정에서 여러개의 노드를 사용할때의 computational efficiency가 노드 사이 통신 overhead에 가려지고 있다고 판단하였다. 특히, 행렬 곱셈 과정에서 B행렬같은 경우, 분할할 수 없어서 4개의 노드에 중복되게 전송해야 했는데, 이 과정도 비효율적이라고 느꼈다.

  하나의 노드 (GPU 4개)를 사용하는 대신, cuda stream을 두개 사용해서 Host-to-Device (H2D) transfers, kernel execution, and Device-to-Host (D2H) transfers, 위 3개의 과정을 겹치게 수행할 수 있도록 구현하였다. 하나의 stream이 데이터 전송을 하는동안 다른 stream은 kernel execution과 같은 작업을 할 수 있기 때문에 매우 효율적인 최적화 방법이라고 요약할 수 있다. 엄청난 성능 향상을 목격할 수 있었다.

- Kernel안에서 float4 이용한 vectorization **(20300 GFLOPS)**

  우선, 노드를 하나만 사용했기 때문에 **run.sh에 노드 개수를 (: ${NODES:=1}) 로 수정**하였다. 그리고 kernel에 몇가지 최적화를 더 해주었다. 가장 먼저 float4 데이터 타입을 사용해

서 vectorized load가 가능하게 구현하였다. 동시에 4개의 값을 load/store할 수 있기 때문에 kernel이 한층 더 최적화되었다. Shared memory에 load하는 과정을 효율적으로 만들 수 있었다. 추가적으로, #pragma unroll을 사용해서 컴파일 과정에서도 loop관련 최적화가 이루어질수 있도록 구현하였다. 결과적으로 목표 성능인 20000 GFLOPS를 목격할 수 있었다.

노드 별 성능 차이가 꽤 큰것같다는 생각에 다양한 노드를 할당받아서 측정한 결과는 다음과 같다

*a0 : 20300 GFLOPS

*a2 : 20435 GFLOPS

*a3 : 20446 GFLOPS

*a4 : 13852 GFLOPS

*a8 : 20295 GFLOPS

-성능 : ./run.sh 65536 4096 4096 -n 20 -v 실행결과 20446 GFLOPS

```
shpc106@login2:~/skeleton/hw6$ ./run.sh 65536 4096 4096 -n 20 -v
salloc: Pending job allocation 1104532
salloc: job 1104532 queued and waiting for resources
salloc: job 1104532 has been allocated resources
salloc: Granted job allocation 1104532
(a03) Hello world, rank 0 out of 1
Options:
  Problem size: M = 65536, N = 4096, K = 4096
  Number of iterations: 20
  Print matrix: off
  Validation: on

[rank 0] Initializing matrices...Done!
Using 4 device(s)
GPU 0: NVIDIA TITAN RTX
GPU 1: NVIDIA TITAN RTX
GPU 2: NVIDIA TITAN RTX
GPU 3: NVIDIA TITAN RTX
[rank 0] Calculating...(iter=0) 0.108025 sec
[rank 0] Calculating...(iter=1) 0.107871 sec
[rank 0] Calculating...(iter=2) 0.107211 sec
[rank 0] Calculating...(iter=3) 0.107939 sec
[rank 0] Calculating...(iter=4) 0.106731 sec
[rank 0] Calculating...(iter=5) 0.107784 sec
[rank 0] Calculating...(iter=6) 0.107080 sec
[rank 0] Calculating...(iter=7) 0.107111 sec
[rank 0] Calculating...(iter=8) 0.108259 sec
[rank 0] Calculating...(iter=9) 0.106897 sec
[rank 0] Calculating...(iter=10) 0.108512 sec
[rank 0] Calculating...(iter=11) 0.107090 sec
[rank 0] Calculating...(iter=12) 0.107526 sec
[rank 0] Calculating...(iter=13) 0.107706 sec
[rank 0] Calculating...(iter=14) 0.106968 sec
[rank 0] Calculating...(iter=15) 0.108719 sec
[rank 0] Calculating...(iter=16) 0.107277 sec
[rank 0] Calculating...(iter=17) 0.107758 sec
[rank 0] Calculating...(iter=18) 0.108038 sec
[rank 0] Calculating...(iter=19) 0.106463 sec
Validating...
Result: VALID
[rank 0] Avg. time: 0.107548 sec
[rank 0] Avg. throughput: 20446.843023 GFLOPS
salloc: Relinquishing job allocation 1104532
```