M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Final Project

Due : 2024/12/20 (Fri)

2024-28413 이정현

1. GPU 통신 설계

이번 과제에서 모델을 최적화하기 위해서 총 4개의 노드를 사용하였고, 대략적인 흐름은 다음과 같다.

- Embed / Permute - 노드 1개 (GPU 1개)
- Convolution layer - 노드 4개 (GPU 16개)
- Concat - 노드 1개 (GPU 1개)
- Linear layer - 노드 1개 (GPU 4개)

각 노드가 처리하는 부분은 다음과 같다. (각 노드를 rank0, rank1, rank2, rank3 으로 지칭)

[Rank 0]	[Rank 1]	[Rank 2]	[Rank 3]
-Embed 실행 -Permute 실행			
-다른 rank 들에 broadcast	-Rank0으로부터 receive	-Rank0으로부터 receive	-Rank0으로부터 receive
-Convolution Layer 0	-Convolution Layer 1	-Convolution Layer 2	-Convolution Layer 3
-다른 rank 결과 받기	-Rank0로 결과 보내기	-Rank0로 결과 보내기	-Rank0로 결과 보내기
-Concat			
-Linear layer 0 -Linear layer 1 -Linear layer 2 -Linear layer 3			

Convolution 연산 전후로 각 rank 사이에서 데이터 전송을 위해서는 MPI 통신을 이용하였다. Rank0에서 permute 값을 전송하기 위해서는 MPI_Bcast, convolution이후 다시 rank0 로 결과를 전송하는 과정에서는 MPI_Send/Recv 를 이용해주었다.

```
    //step 2: e&p data to host
    CHECK_CUDA(cudaMemcpy(host_processed_data, permute_a->buf, batch_size * EMBEDDING_DIM * SEQ_LEN * sizeof(float), cudaMemcpyDeviceToHost));

    //step 3: broadcast e&p data to all ranks
    MPI_Bcast(host_processed_data, batch_size * EMBEDDING_DIM * SEQ_LEN, MPI_FLOAT, 0, MPI_COMM_WORLD);

    //step 4: copy received data back to device
    CHECK_CUDA(cudaMemcpy(permute_a->buf, host_processed_data, batch_size * EMBEDDING_DIM * SEQ_LEN * sizeof(float), cudaMemcpyHostToDevice));
}
else{ //rank1, rank2, rank3
    MPI_Bcast(host_processed_data, batch_size * EMBEDDING_DIM * SEQ_LEN, MPI_FLOAT, 0, MPI_COMM_WORLD);
    CHECK_CUDA(cudaMemcpy(permute_a->buf, host_processed_data, batch_size * EMBEDDING_DIM * SEQ_LEN * sizeof(float), cudaMemcpyHostToDevice));
}
//all ranks have e&p data for convolution
```

<Permute 값을 모든 rank에 전달하는 과정 MPI_Bcast 이용>

4개의 convolution layer들은 각각 병렬 처리가 가능했기 때문에, layer 당 하나의 노드를 배정해 주었다. 또한, 연산량이 매우 컸음으로 효율적으로 계산하기 위해 thread를 사용해서 각 노드 안에서 4개의 GPU를 모두 활용하도록 하였다.

```
size_t distribute_B = (batch_size + NUM_GPU - 1) / NUM_GPU;
auto use_gpu = [&](int gpu_idx) {
    size_t start_idx = gpu_idx * distribute_B;
    if (start_idx >= batch_size){
        return; //end
    }
    size_t currB = std::min(distribute_B, batch_size - start_idx); //get current distributed batch

    float *d_in, *d_w, *d_b, *d_max_out;
    CHECK_CUDA(cudaSetDevice(gpu_idx));

    // Create a CUDA stream for this GPU
    cudaStream_t gpu_stream;
    CHECK_CUDA(cudaStreamCreate(&gpu_stream));
```

```
std::vector<std::thread> gpu_threads;
for(int gpu_idx = 0; gpu_idx < NUM_GPU; gpu_idx++) {
    gpu_threads.emplace_back(use_gpu, gpu_idx);
}
for (auto &t : gpu_threads) t.join();
```

<convolution kernel을 부르기 전 4개의 GPU로 workload 분산>

또한, linear layer 계산을 할때도 GPU 4개를 모두 활용하게 하였다. (노드는 1개). 데이터를 적절히 분산시켜서 4개의 GPU에서 정확한 계산이 일어날 수 있도록 해주었다. Linear layer들은 각각 순차적으로 진행되고, 하나의 layer가 계산될 때 4개의 GPU가 사용된다.

```
size_t distribute_B = (batch_size + NUM_GPU - 1) / NUM_GPU; //distribute batch
auto use_gpu = [&](int gpu_idx) {
    size_t start_idx = gpu_idx * distribute_B;
    if (start_idx >= batch_size){
        return;
    }
    size_t currB = std::min(distribute_B, batch_size - start_idx);
```

Linear layer도 처음에는 노드 하나당 layer 1개를 할당하는 방법으로 구현했으나, 순차적으로 진행된다는 특징과, 노드 사이 data transfer overhead가 너무 커져서 성능이 잘 나오지 않는 것을 확인하고, 하나의 노드, 4개의 GPU 방법으로 구현하였다.

2. Parameter / Activation Pre-allocation

특정 노드들만 담당하는 연산 layer들이 있었기 때문에, 모든 노드에 똑같은 Parameter/activation을 할당해주지 않고, alloc_and_set_parameters() 와 alloc_activation()을 수정해서 원하는 노드에 필요한 parameter/activation들이 pre-allocate되어있도록 해주었다. 이를 위해서는 tensor class의 수정이 필요했고, 생성자에 device_id를 추가적으로 받을 수 있게 해서 원하는 device에 바로 메모리 할당이 가능하게 해주었다. Pinned memory로 할당해서 최적화에 기여하였다. (free함수도 맞춰서 변형) 또한, predict_sentiment안에서 필요한 중간 변수들도 (예를 들어서 permute/concat과 같은 결과를 임시로 저장할 host buffer 등) 함수 밖에서 미리 선언하고, 대부분 pinned memory를 사용해주었다.

```cpp
//constructor for float buffers with host data
Tensor::Tensor(const std::vector<size_t> &shape_, float *host_buf_, int device_id_)
    : ndim(shape_.size()), buf(nullptr), buf_int(nullptr), total_elements(1), is_gpu(true), device_id(device_id_) {
    assert(ndim <= 4 && "Tensor constructor supports up to 4 dimensions.");
    for(size_t i = 0; i < ndim; i++) { shape[i] = shape_[i]; }
    for(size_t i = ndim; i < 4; i++) { shape[i] = 1; }
    for(size_t i = 0; i < ndim; i++) { total_elements *= shape[i]; }

    CHECK_CUDA(cudaSetDevice(device_id));
    CHECK_CUDA(cudaMallocHost(&buf, total_elements * sizeof(float))); //allocate device memory for float data

    if (host_buf_ != nullptr) { //if host buffer exists, copy data to device
        CHECK_CUDA(cudaMemcpy(buf, host_buf_, total_elements * sizeof(float), cudaMemcpyHostToDevice));
    }
}
```

<tensor.cu 수정한 부분. Device에 바로 할당>

```cpp
//allocate embedding weights
emb_w = new Parameter({21635, EMBEDDING_DIM}, param + pos, rank); //[21635, 4096]
pos += 21635 * EMBEDDING_DIM;

//allocate conv parameters
int kernel_sizes[NUM_WORKERS] = {3, 5, 7, 9}; //kernel size differs for each rank

for(int r = 0; r < NUM_WORKERS; r++) {
    if(r == rank){ //current rank (weights + bias)
        conv_w[r] = new Parameter({N_FILTERS, EMBEDDING_DIM, static_cast<size_t>(kernel_sizes[r])}, param + pos, rank); //[1024, 4096, K]
        pos += N_FILTERS * EMBEDDING_DIM * kernel_sizes[r];
        conv_b[r] = new Parameter({N_FILTERS}, param + pos, rank); //[1024]
        pos += N_FILTERS;
    }
    else{ //conv parameters of other ranks ignored. But their position is accumulated
        pos += N_FILTERS * EMBEDDING_DIM * kernel_sizes[r]; //weight
        pos += N_FILTERS; //bias
    }
}
```

<alloc_and_set_parameters 일부. 각 rank별로 conv_w와 conv_b를 할당한다>

```cpp
emb_a = new Activation({BATCH_SIZE, SEQ_LEN, EMBEDDING_DIM}, (float*)nullptr, rank); //embedding
permute_a = new Activation({BATCH_SIZE, EMBEDDING_DIM, SEQ_LEN}, (float*)nullptr, rank); //permute

if(rank == 0){ //rank 0 handles all linear layers
    conv_out_a_workers[0] = new Activation({BATCH_SIZE, N_FILTERS}, (float*)nullptr, rank); //conv0
    concat_a = new Activation({BATCH_SIZE, 4096}, (float*)nullptr, rank);
    linear0_a = new Activation({BATCH_SIZE, 2048}, (float*)nullptr, rank);
    linear1_a = new Activation({BATCH_SIZE, 1024}, (float*)nullptr, rank);
    linear2_a = new Activation({BATCH_SIZE, 512}, (float*)nullptr, rank);
    linear3_a = new Activation({BATCH_SIZE, 2}, (float*)nullptr, rank);
}
else{ //other ranks -> only conv
    conv_out_a_workers[rank] = new Activation({BATCH_SIZE, N_FILTERS}, (float*)nullptr, rank);
}
```

<alloc_activations 일부. 필요한 activation만 rank마다 배정하고, batching을 고려하였다>

3. Batching / Kernel Optimization

한번에 여러개의 input을 동시에 처리하기 위해서 batching을 고려하였다. 이를 고려하기 위해서 모든 커널을 조금씩 바꿨지만, 가장 중요한 부분 중 하나는 concat이였다. Concat을 그냥 적용 시,

B=1	conv0-S1	conv1-S1	conv2-S1	conv3-S1				
B=2	conv0-S1	conv0-S2	conv1-S1	conv1-S2	conv2-S1	conv2-S2	conv3-S1	conv3-S2
Fixed	conv0-S1	conv1-S1	conv2-S1	conv3-S1	conv0-S2	conv1-S2	conv2-S2	conv3-S2

배치가 늘어날수록 위 그림과 같이 interleaving 현상이 일어난다는 것을 파악하고, batched concat이 올바르게 될 수 있도록 구현하였다. (위 현상은 linear input으로 사용될때 문제일어남)

Activation의 경우, batching을 사용하게 되면 크기 변화가 있어야하기 때문에 이를 고려한 크기의 tensor로 선언해주었다.

Kernel의 경우, 지속적으로 함께 불리는 커널들이 존재한다는 것을 확인 후, 총 2개의 Kernel fusion을 적용하였다.

(1) Batched_Conv1D_ReLU_GetMax : convolution의 결과에는 항상 ReLU와 GetMax가 함께 적용되기 때문에, 이 세개의 연산을 하나의 커널로 합쳤다. 각 노드가 한번씩 실행하게 된다.

(2) Batched_Linear_ReLU : linear0,1,2 layer에서는 끝나고 결과에 ReLU가 적용된다. 따라서 이 두개의 연산을 합친 kernel을 만들었다. (마지막 Linear layer 3에는 그냥 Batched_Linear가 적용된다)

마지막으로, convolution 연산의 비중이 가장 높은것을 초기 단계에서 nsys를 사용해서 파악했었는데 성능 향상을 위해 이를 최적화하였다.

```
** CUDA GPU Kernel Summary (gpukernsum):

Time (%)  Total Time (ns)  Instances   Avg (ns)     Med (ns)     Min (ns)   Max (ns)   StdDev (ns)    GridXYZ        BlockXYZ                                              Name
--------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  --------------  --------------  ----------------------------------------------------------------------------------------------------
    99.4        1,821,165          1  1,821,165.0  1,821,165.0  1,821,165  1,821,165          0.0  1024    1    1    14    1    1  Batched_Conv1D_ReLU_GetMax_kernel(float *, float *, float *, int, int, int, int, float *, unsigned …
     0.4            6,624          1      6,624.0      6,624.0      6,624      6,624          0.0     1  256    1    16   16    1  Batched_Permute_kernel(float *, float *, unsigned long, unsigned long, unsigned long)
     0.2            4,287          1      4,287.0      4,287.0      4,287      4,287          0.0     1    1    1   256    1    1  Batched_Embedding_Kernel(int *, float *, float *, unsigned long, unsigned long, unsigned long)
```

먼저, kernel호출 과정에서 하나의 블록을 output channel x batch_size로 설정해주었다. 이를 통해서 병렬성을 높이고, load balancing을 기대할 수 있었다. 각 thread는 multiple position을 실행하게 되고, 하나의 thread안에서 로컬 max값을 업데이트하면서 저장하고 있게 하였다. (효율적인 reduction 가능)

GetMax를 위해서는 두번의 reduction을 적용하였다. 첫번째 warp-level reduction에서는 각 warp 내에서 __shfl_down_sync 함수를 사용하여 thread 간에 thread_max 값을 공유하고 비교하였다. 이 과정이 끝나면 warp안에서의 최댓값을 구할 수 있다. Global memory 접근 없이

warp내에서 구할 수 있기 때문에 성능 향상에 많은 도움이 되었다. 여기서 구한 최댓값은 shared memory에 저장되고, 여기서 또 한번의 reduction을 통해서 block내의 최댓값을 구하도록 구현하였다.

이번 프로젝트에서 대략적으로 추적했던 성능변화는 다음과 같다.

Stage	Throughput (sentences/sec)
Data Transfer 설계완료 (노드 4개 사용)	250
Pinned Memory, Preallocate parameters	610
Kernel Optimization, Hyper-parameter search (batch_size, n_samples)	880

마지막으로 n_sample과 batch_size의 크기를 조절해가면서 가장 성능이 잘 나오는 조합을 찾은 결과는 다음과 같았다. (n_sample=16384, BATCH_SIZE=2048)

[성능 결과 사진]

```
shpc106@login2:~/skeleton/final-project_original$ ./run.sh -n 16384 -w -v
salloc: Pending job allocation 1178080
salloc: job 1178080 queued and waiting for resources
salloc: job 1178080 has been allocated resources
salloc: Granted job allocation 1178080

================================================
 Model: Sentiment Analysis
------------------------------------------------
 Validation: ON
 Warm-up: ON
 Number of sentences: 16384
 Input binary path: ./data/inputs.bin
 Model parameter path: /home/s0/shpc_data/params.bin
 Answer binary path: ./data/answers.bin
 Output binary path: ./data/outputs.bin
================================================

Initializing inputs and parameters...Done!
Warming up...Done!
Predicting sentiment...Done!
Elapsed time: 18.617908 (sec)
Throughput: 880.012932 (sentences/sec)
Finalizing...Done!
Saving outputs to ./data/outputs.bin...Done!
Validating...PASSED!
salloc: Relinquishing job allocation 1178080
```