Hpc final project

컴퓨터공학부

2023-22317 이기훈

성능

```
Model: GPT-2 125M
--------------------------------------------
Validation: ON
Number of Prompts: 6400
Number of Tokens to generate: 8
Input binary path: ./data/input.bin
Model parameter path: /shpc24/project_model_paramet
Answer binary path: ./data/answer.bin
Output binary path: ./data/output.bin
============================================

Initializing input and parameters...Done
Generating tokens...Done!
Elapsed time: 116.649317 (sec)
Throughput: 438.922417 (tokens/sec)
Finalizing...Done
Saving output to ./data/output.bin...Done
Validation...PASS
salloc: Relinquishing job allocation 746174
```

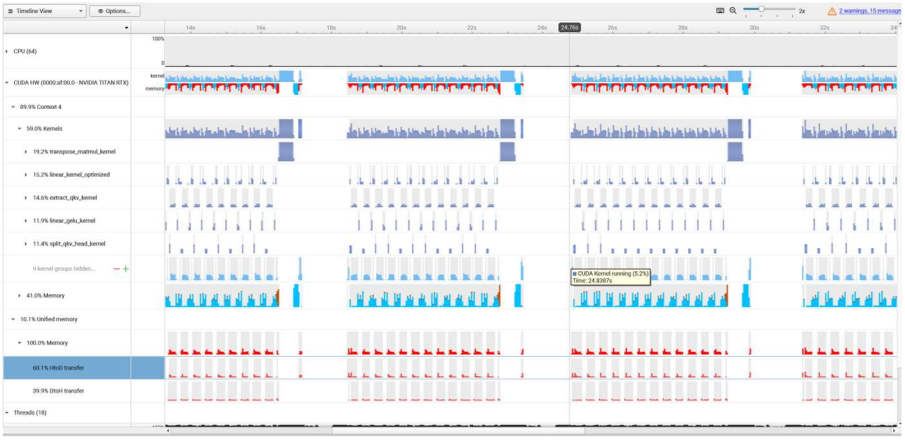
final-project > $ run.sh

```
1  #!/bin/bash
2
3  : ${NODES:=4}
4
5  salloc -N $NODES --partition class1 --exclusive --gres=gpu:4   \
6    mpirun --bind-to none -mca btl ^openib -npernode 4          \
7    numactl --physcpubind 0-31                                  \
8  ./main -v -n 6400 -t 8
9
```

배치 데이터를 처리할 수 있도록 모델을 구현하였고, batch_size 800으로 6400개의 prompts를 조건으로 넣었을 때, 가장 높은 throughput인 438.9가 나왔다. Run.sh에 작성했 듯이 내 프로젝트 제출물은 다음과 같이 실행된다.

Tensor 구조체

과제를 진행하던 초기에, 최적화하여 nsight를 통해 프로파일링한 결과, 트랜스포머 모델 안 연산에서 이상하게 host로의 memory 전송이 많이 발생하는 것을 확인할 수 있었다.



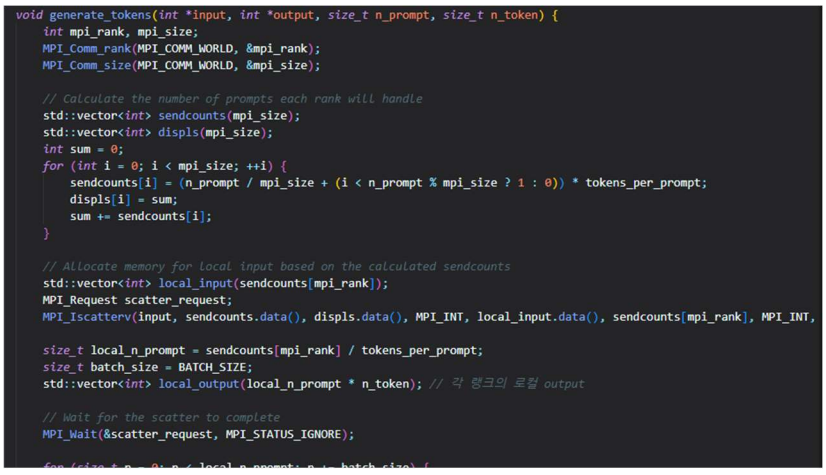
이는 모델의 레이어에 해당하는 함수안에서 쓸데없이 호스트와 디바이스 사이의 전송이 들어가 있다는 의미이다. 배치 데이터가 한번 transformer_block에 들어가면 그 안에선 데이터 전송이 불필요하기 때문에 이를 먼저 개선하고자 하였다. 이 과정에서 알게된 것이 Tensor 구조체의 설계였다.

```cpp
/* Tensor */
Tensor::Tensor(const vector<size_t> &shape_) {
  ndim = shape_.size();
  for (size_t i = 0; i < ndim; i++) { shape[i] = shape_[i]; }
  size_t N_ = num_elem();

  // Allocate unified memory for the device
  cudaError_t err = cudaMallocManaged(&buf, N_ * sizeof(float));
  if (err != cudaSuccess) {
    fprintf(stderr, "Failed to allocate unified memory for Tensor: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }

  // Allocate host memory
  host_buf = (float*)malloc(N_ * sizeof(float));
  if (host_buf == nullptr) {
    fprintf(stderr, "Failed to allocate host memory for Tensor\n");
    cudaFree(buf);
    exit(EXIT_FAILURE);
  }

  memset(buf, 0, N_ * sizeof(float)); // Initialize device memory
  memset(host_buf, 0, N_ * sizeof(float)); // Initialize host memory
}
```

Tensor 구조체의 buf는 device에 올라가 있기 때문에 모델의 레이어 함수들이 서로 인자로 받는 Tensor 변수는 cpu에 있지만 커널 함수에서 그들의 buf를 이용해 연산하기 때문에 호스트와 디바이스 간의 데이터 전송이 불필요하다. 스켈레톤 코드에 대한 이해도가 부족했기 때문에 프로젝트 초반의 많은 시간을 허비한 것이 큰 교훈이 되었다.

MPI 통신

generate_tokens 함수는 MPI를 사용하여 다수의 프로세서 코어에 작업을 분산시키고, 이를 통해 대규모 데이터의 처리와 계산을 병렬화하였다. 이 함수는 입력 데이터를 분산하여 각 프로세서가 병렬로 토큰을 생성하고, 결과를 다시 모아 최종 출력을 생성하도록 구현되었다.



```cpp
void generate_tokens(int *input, int *output, size_t n_prompt, size_t n_token) {
    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    // Calculate the number of prompts each rank will handle
    std::vector<int> sendcounts(mpi_size);
    std::vector<int> displs(mpi_size);
    int sum = 0;
    for (int i = 0; i < mpi_size; ++i) {
        sendcounts[i] = (n_prompt / mpi_size + (i < n_prompt % mpi_size ? 1 : 0)) * tokens_per_prompt;
        displs[i] = sum;
        sum += sendcounts[i];
    }

    // Allocate memory for local input based on the calculated sendcounts
    std::vector<int> local_input(sendcounts[mpi_rank]);
    MPI_Request scatter_request;
    MPI_Iscatterv(input, sendcounts.data(), displs.data(), MPI_INT, local_input.data(), sendcounts[mpi_rank], MPI_INT,

    size_t local_n_prompt = sendcounts[mpi_rank] / tokens_per_prompt;
    size_t batch_size = BATCH_SIZE;
    std::vector<int> local_output(local_n_prompt * n_token); // 각 랭크의 로컬 output

    // Wait for the scatter to complete
    MPI_Wait(&scatter_request, MPI_STATUS_IGNORE);
```

함수는 먼저 MPI 초기화 및 랭크 정보를 얻는다. MPI_Comm_rank와 MPI_Comm_size를 호출하여 현재 프로세스의 랭크와 전체 프로세스 수를 가져온다. 이를 이용해 입력 데이터는 각 프로세스가 처리할 수 있도록 분할된다. 각 프로세스가 처리할 프롬프트 수를 계산하기 위해 sendcounts와 displs 벡터를 사용한다. sendcounts는 각 프로세스가 처리할 데이터의 크기를 저장하고, displs는 각 프로세스가 데이터를 읽어올 시작 인덱스를 저장하였다. 이 인덱스를 이용해, MPI_Iscatterv를 사용하여 입력 데이터를 각 프로세스로 분산한다. 비동기 통신을 사용하여 데이터를 분산시키며, MPI_Wait를 통해 통신이 완료될 때까지 기다린다. 각 프로세스는 로컬 입력 데이터를 배치 단

위로 프롬프트와 토큰을 생성한다. memcpy를 사용하여 로컬 입력 데이터를 input_prompts에 복사한다. 이후, 각 프롬프트에 대해 토큰을 생성하고, 이를 다시 local_output에 저장한다.

```
std::vector<int> next_tokens(actual_batch_size);
top1_sampling(logit_a, next_tokens.data(), tokens_per_prompt, actual_batch_size); // 여기서 호스트로 복사.

#pragma omp parallel for
for (size_t b = 0; b < actual_batch_size; b++) {
    input_prompts[b].push_back(next_tokens[b]);

    // Calculate the local index for the output array
    size_t local_output_index = (p + b) * n_token + t;
    local_output[local_output_index] = next_tokens[b];
}

prompt_size += 1;
free_activations();
```

토큰 생성 과정에서 다양한 계산이 이루어지며, 이때 OpenMP를 사용하여 병렬 처리를 수행한다. OpenMP는 각 배치 내의 프롬프트에 대해 병렬로 토큰을 생성하며, 이를 통해 계산 속도를 향상시킨다. 각 프롬프트의 출력은 local_output 배열에 저장되며, 인덱싱을 통해 정확한 위치에 저장된다.

```
// 각 랭크가 계산한 output의 크기
int local_output_size = local_n_prompt * n_token;
std::vector<int> recvcounts(mpi_size);
std::vector<int> displs_output(mpi_size);

// output을 모을 때 사용할 수치들 계산
if (mpi_rank == 0) {
    int output_sum = 0;
    for (int i = 0; i < mpi_size; ++i) {
        recvcounts[i] = (n_prompt / mpi_size + (i < n_prompt % mpi_size ? 1 : 0)) * n_token;
        displs_output[i] = output_sum;
        output_sum += recvcounts[i];
    }
}

// 모든 랭크에서 rank 0으로 output 데이터 모으기 (비동기 통신 사용)
MPI_Request gather_request;
MPI_Igatherv(local_output.data(), local_output_size, MPI_INT, output, recvcounts.data(), displs_output.data(), MPI_INT, 0, MPI_COMM_WORLD

// Wait for the gather to complete
MPI_Wait(&gather_request, MPI_STATUS_IGNORE);
}
```

모든 프로세스가 로컬 출력을 계산한 후, MPI_Igatherv를 사용하여 모든 로컬 출력을 하나의 최종 출력으로 모은다. 이때, recvcounts와 displs_output을 사용하여 각 프로세스의 출력 데이터를 정확한 위치에 모은다. 비동기 통신을 사용하여 출력 데이터를 모으며, MPI_Wait를 통해 통신이 완료될 때까지 기다린다.

Matmul

행렬곱 함수 matmul은 CUDA 스트림과 공유 메모리를 활용하여 GPU에서 효율적인 행렬곱 연산을 수행하도록 설계되었다.

```
__global__ void matmul_kernel(float *A, float *B, float *C, int M, int N, int K) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    const int row = blockIdx.y * blockDim.y + threadIdx.y;
    const int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    for (int t = 0; t < (K + BLOCK_SIZE - 1) / BLOCK_SIZE; t++) {
        if (row < M && t * BLOCK_SIZE + threadIdx.x < K)
            As[threadIdx.y][threadIdx.x] = A[row * K + t * BLOCK_SIZE + threadIdx.x];
        else
            As[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < N && t * BLOCK_SIZE + threadIdx.y < K)
            Bs[threadIdx.y][threadIdx.x] = B[(t * BLOCK_SIZE + threadIdx.y) * N + col];
        else
            Bs[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads();

        for (int i = 0; i < BLOCK_SIZE; ++i)
            sum += As[threadIdx.y][i] * Bs[i][threadIdx.x];

        __syncthreads();
    }

    if (row < M && col < N)
        C[row * N + col] = sum;
}
```

```
void matmul(Tensor *in1, Tensor *in2, Tensor *out, size_t batch_size, bool verbose) {
    size_t M = in1->shape[1];
    size_t K = in1->shape[2];
    size_t N = in2->shape[2];

    float *d_in1 = in1->buf;
    float *d_in2 = in2->buf;
    float *d_out = out->buf;

    size_t size_out = batch_size * M * N * sizeof(float);

    CHECK_CUDA(cudaMemset(d_out, 0, size_out));

    dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (M + blockDim.y - 1) / blockDim.y);

    cudaStream_t streams[NUM_STREAMS];
    for (int i = 0; i < NUM_STREAMS; ++i) {
        cudaStreamCreate(&streams[i]);
    }

    for (size_t b = 0; b < batch_size; b++) {
        int stream_idx = b % NUM_STREAMS;
        matmul_kernel<<<gridDim, blockDim, 0, streams[stream_idx]>>>(d_in1 + b * M * K, d_in2 + b * K * N, d_out + b *
    }

    for (int i = 0; i < NUM_STREAMS; ++i) {
        cudaStreamSynchronize(streams[i]);
        cudaStreamDestroy(streams[i]);
    }
}
```

CUDA 커널 matmul_kernel에서는 행렬 A와 B의 블록을 각각 공유 메모리인 As와 Bs에 로드한다. 공유 메모리는 GPU의 전역 메모리보다 접근 속도가 훨씬 빠르기 때문에, 이를 활용하여 메모리 접근 시간을 줄인다. 각 블록의 스레드들은 반복적으로 행렬 A와 B의 블록을 공유 메모리에 로드하고, 이를 이용하여 부분 행렬곱을 계산한다. 공유 메모리 내에서의 데이터 재사용을 통해 글로벌 메모리 접근 횟수를 줄이고, 캐시 히트율을 높인다.

그리고 CUDA 스트림을 사용한 비동기적으로 수행되도록 최적화하였다. 호스트 코드에서는 여러 CUDA 스트림을 생성하여 각 배치를 병렬로 처리한다. 스트림은 GPU에서 비동기적으로 실행되기 때문에, 여러 스트림을 사용하면 데이터 전송과 커널 실행을 겹쳐서 수행할 수 있다. 이를 통해 GPU 자원의 활용도를 극대화하고, 전체 연산 시간을 줄이도록 하였다. matmul 함수에서는 NUM_STREAMS = 16 개의 스트림을 생성하고, 각 배치가 고유의 스트림에서 실행된다. 스트림의 개수는 실험을 통해 최적의 값을 설정하였다.

커널 실행 시, 각 블록은 BLOCK_SIZE x BLOCK_SIZE 크기의 스레드로 구성되며, 그리드는 행렬 C의 크기에 맞게 설정된다. 이를 통해 모든 행렬 요소가 병렬로 계산될 수 있도록 한다. matmul에서의 최적의 BLOCK_SIZE는 실험 결과 8이었다.

또한, 동기화 지점을 최소화하여 성능을 높였다. 커널 내에서는 공유 메모리에 데이터를 로드한 후, 동기화를 통해 모든 스레드가 데이터를 사용할 준비가 되었는지 확인한다. 이후 행렬곱 계산을 수행하고, 다시 동기화하여 모든 스레드가 다음 블록의 데이터를 로드할 준비가 되었는지 확인한다. 이러한 동기화 지점을 최소화하여, 불필요한 대기 시간을 줄이고, 계산의 효율성을 높였다.

Matmul types

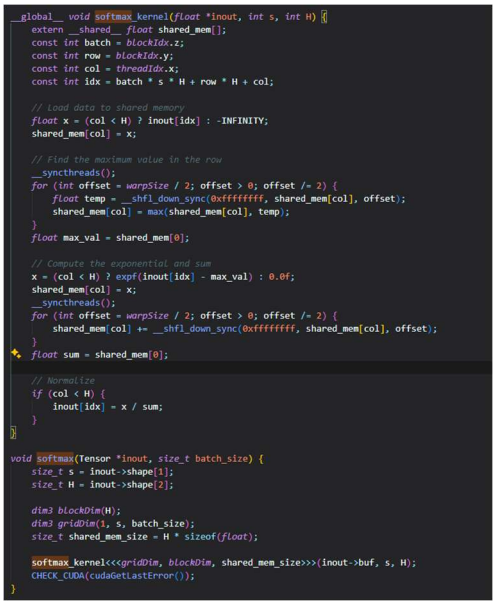
Skeleton code의 model.cu에서 matmul은 여러 상황에서 호출되었기 때문에 그 앞뒤로 호출되는 scaling이나 transpose 작업과 따로 호출된다. 이를 한꺼번에 처리하면 성능을 높일 수 있을 것이라고 생각하여 행렬곱을 3개의 type으로 나눠 구현하였다.

Matmul은 일반적인 배치 처리된 데이터에 대한 행렬곱 함수이고, matmul_2는 행렬 B를 열 기준으로 분할하여 전치(transpose) 연산을 동시에 수행함으로써 메모리 접근 패턴을 최적화하였다. 이는 메모리 접근 시의 캐시 적중률을 높여 성능을 향상시킨다. 또한, 결과 행렬 C에 대해 마스크(mask)와 스케일링(scaling)을 적용하여, 특정 조건(row < col)에 따라 값을 설정하거나, 계산된 합(sum)을 스케일 값으로 곱해 저장한다. 이러한 최적화는 추가적인 연산을 줄이고, 필요한 경우에만 연산을 수행하도록 하여 효율성을 높인다

그리고 matmul_3는 두 번째 구현을 더욱 개선한 형태로, 입력 행렬 B를 전치하는 과정을 지속적으로 최적화하였다. 이를 통해, 행렬 B의 요소를 불필요하게 여러 번 읽지 않고, 한 번의 메모리 접근으로 다수의 연산을 수행할 수 있도록 하였다. 이 구현에서는 각 배치(batch)에 대해 별도의

CUDA 스트림을 사용하여 비동기적으로 계산을 수행한다. 이는 GPU의 여러 연산 유닛을 동시에 활용하여, 연산과 데이터 전송이 중첩될 수 있도록 하여 성능을 극대화한다. 스트림 간의 동기화는 계산 완료 후에 이루어지며, 이를 통해 계산이 끝난 스트림은 즉시 다음 배치를 처리할 수 있도록 한다.

Softmax 함수 최적화



```
__global__ void softmax_kernel(float *inout, int s, int H) {
    extern __shared__ float shared_mem[];
    const int batch = blockIdx.z;
    const int row = blockIdx.y;
    const int col = threadIdx.x;
    const int idx = batch * s * H + row * H + col;

    // Load data to shared memory
    float x = (col < H) ? inout[idx] : -INFINITY;
    shared_mem[col] = x;

    // Find the maximum value in the row
    __syncthreads();
    for (int offset = warpSize / 2; offset > 0; offset /= 2) {
        float temp = __shfl_down_sync(0xffffffff, shared_mem[col], offset);
        shared_mem[col] = max(shared_mem[col], temp);
    }
    float max_val = shared_mem[0];

    // Compute the exponential and sum
    x = (col < H) ? expf(inout[idx] - max_val) : 0.0f;
    shared_mem[col] = x;
    __syncthreads();
    for (int offset = warpSize / 2; offset > 0; offset /= 2) {
        shared_mem[col] += __shfl_down_sync(0xffffffff, shared_mem[col], offset);
    }
    float sum = shared_mem[0];

    // Normalize
    if (col < H) {
        inout[idx] = x / sum;
    }
}

void softmax(Tensor *inout, size_t batch_size) {
    size_t s = inout->shape[1];
    size_t H = inout->shape[2];

    dim3 blockDim(H);
    dim3 gridDim(1, s, batch_size);
    size_t shared_mem_size = H * sizeof(float);

    softmax_kernel<<<gridDim, blockDim, shared_mem_size>>>(inout->buf, s, H);
    CHECK_CUDA(cudaGetLastError());
}
```

소프트맥스 함수에선 공유 메모리와 워프 수준의 함수들을 적절히 사용해 메모리 접근 시간을 줄이고, 병렬 연산의 효율성을 높였다. 그리고 __shfl_down_sync 함수를 이용하여 워프 내 스레드 간의 데이터를 효율적으로 교환하고, 병렬 연산을 수행하도록 하였다. 이 함수는 하드웨어 수준에서 스레드 간 데이터를 교환하므로, 전통적인 메모리 접근보다 훨씬 빠르다. 또한 동기화 지점을 최소화하여, 스레드 간의 불필요한 대기 시간을 줄이도록 하였다. __syncthreads를 적절히 사용하여, 필요한 경우에만 스레드 간 동기화를 수행하게 하였다.