Report 2024-27759 장효형

5페이지 이내로 본인이 적용한 최적화 내용을 간략히 작성

1) Cudamalloc, cudamemcpy of parameters in alloc_perameters(), of activation in alloc_activation

```
linear3_w = new Parameter({2, 512}, param + pos);
pos += 2 * 512;
linear3_b = new Parameter({2}, param + pos);
pos += 2;

initializeGridAndBlockDimensions(&dims);
initGPUContextsParameters(contexts);

if (pos != param_size) {
  fprintf(stderr, "Parameter size mismatched: %zu != %zu\n",
          pos, param_size);
  exit(EXIT_FAILURE);
}
}
```

```
void alloc_activations() {
  emb_a = new Activation({BATCH_SIZE, SEQ_LEN, 4096});
  permute_a = new Activation({BATCH_SIZE, 4096, SEQ_LEN});
  conv0_a = new Activation({BATCH_SIZE, 1024, SEQ_LEN - 2});
  pool0_a = new Activation({BATCH_SIZE, 1024});
  conv1_a = new Activation({BATCH_SIZE, 1024, SEQ_LEN - 4});
  pool1_a = new Activation({BATCH_SIZE, 1024});
  conv2_a = new Activation({BATCH_SIZE, 1024, SEQ_LEN - 6});
  pool2_a = new Activation({BATCH_SIZE, 1024});
  conv3_a = new Activation({BATCH_SIZE, 1024, SEQ_LEN - 8});
  pool3_a = new Activation({BATCH_SIZE, 1024});
  concat_a = new Activation({BATCH_SIZE, 4096});
  linear0_a = new Activation({BATCH_SIZE, 2048});
  linear1_a = new Activation({BATCH_SIZE, 1024});
  linear2_a = new Activation({BATCH_SIZE, 512});
  linear3_a = new Activation({BATCH_SIZE, 2});

  initGPUContextsActivation(contexts);

}
```

2) Split inputs to 4 nodes

```
void predict_sentiment(int *inputs, float *outputs, size_t n_samples) {
  int mpi_rank, mpi_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

  // 총 16개, 노드당 4개, 배치당 2개, 총 2 배치

  size_t samples_per_node = n_samples / mpi_size;
  int *local_inputs = (int *)malloc(samples_per_node * SEQ_LEN * sizeof(int));
  float *local_outputs = (float *)malloc(samples_per_node * M3 * sizeof(float));

  int mpi_status = MPI_Scatter(inputs, samples_per_node * SEQ_LEN, MPI_INT, local_inputs,
                               samples_per_node * SEQ_LEN, MPI_INT, 0, MPI_COMM_WORLD);
```

3) Split inputs, which is splitted to nodes, to 4 gpus

```
// Launch processing on each GPU in parallel
#pragma omp parallel for num_threads(NUM_GPUS_PER_NODE)
for (int gpu_id = 0; gpu_id < NUM_GPUS_PER_NODE; ++gpu_id) {
    cudaSetDevice(contexts[gpu_id].gpu_id);

    for (size_t cur_batch = 0; cur_batch < num_batches; ++cur_batch) {
        int *batchInput = local_inputs + (cur_batch * NUM_GPUS_PER_NODE + gpu_id) * BATCH_SIZE * SEQ_LEN;
        float *batchOutput = local_outputs + (cur_batch * NUM_GPUS_PER_NODE + gpu_id) * BATCH_SIZE * M3;

      // Asynchronous copy input data to GPU
      CUDA_CHECK(cudaMemcpyAsync(contexts[gpu_id].gpu_inputs, batchInput,
                                BATCH_SIZE * SEQ_LEN * sizeof(int), cudaMemcpyHostToDevice, contexts[gpu_id].stream[0]));
```

4) Kernel fusion

Fuse embedding and permute kernel, fuse conv1d and relu kernel

```
// Embedding layer
EmbeddingPermuteKernel<<<dims.embeddingGridDim, dims.embeddingBlockDim, 0, contexts[gpu_id].stream[0]>>>(
  contexts[gpu_id].gpu_inputs, contexts[gpu_id].emb_wg, contexts[gpu_id].permute_ag, BATCH_SIZE, SEQ_LEN, HI

CUDA_CHECK(cudaStreamSynchronize(contexts[gpu_id].stream[0]));
```

```
    // Add bias and apply ReLU
    val += bias[oc];
    val = val > 0.0f ? val : 0.0f;

    // Store the result
    out[b * (OC * os) + oc * os + j] = val;

}
```

(relu joined in convKernel)

```
    }

    // Write the result if in range
    // Only threadIdx.x == 0 corresponds to writing the result since we accu
    if (b < B && m < M && threadIdx.x == 0) {
        out[b * M + m] = val + bias[m] > 0? val + bias[m]: 0;
    }
}
```

(relu joined in linearkernel)

5) Run 4 kernel in different streams(but this does not affect to the performance much, because major bottleneck is in linear layer)
6) Make unrolling on conv1Dkernel(only tiled3 are shown)

```
__global__ void Conv1DKernelTiled3(
    const float * in,
    const float * __restrict__ w,
    const float * __restrict__ bias,
    float * __restrict__ out,
    size_t B, size_t C, size_t s, size_t OC, size_t K)
{
    // Output sequence length
    size_t os = s - K + 1;

    // Identify current batch index
    size_t b = blockIdx.x;
    // Output channel index
    size_t oc = blockIdx.y * blockDim.y + threadIdx.y;
    // Sequence position index
    size_t j = blockIdx.z * blockDim.x + threadIdx.x;

    // Bounds check
    if (b >= B || oc >= OC || j >= os) return;

    // Accumulator
    float val = 0.0f;

    // Unroll K since K=3.
    // We'll process C in chunks of 4.
    // We'll process C in chunks of 4.
    for (size_t c4 = 0; c4 < C; c4 += 4) {
        // Instead of float4 vector loads, we do scalar loads for each element.
        // Pre-load the input values for k=0, k=1, and k=2 for these 4 channels.

        float x_k0[4], x_k1[4], x_k2[4];
        for (int idx = 0; idx < 4; idx++) {
            size_t c_index = c4 + idx;
            if (c_index < C) {
                x_k0[idx] = in[b * (C * s) + c_index * s + (j + 0)];
                x_k1[idx] = in[b * (C * s) + c_index * s + (j + 1)];
                x_k2[idx] = in[b * (C * s) + c_index * s + (j + 2)];
            } else {
                // If c_index is out of range, set to 0.0f
                x_k0[idx] = 0.0f;
                x_k1[idx] = 0.0f;
                x_k2[idx] = 0.0f;
            }
        }

        // Now perform the accumulation with the corresponding weights
        #pragma unroll
        for (int idx = 0; idx < 4; idx++) {
            size_t c_index = c4 + idx;
            if (c_index < C) {
                // Load weights for each k
                float w_k0 = w[oc * (C * K) + c_index * K + 0];
                float w_k1 = w[oc * (C * K) + c_index * K + 1];
                float w_k2 = w[oc * (C * K) + c_index * K + 2];

                float val_k0 = x_k0[idx];
                float val_k1 = x_k1[idx];
                float val_k2 = x_k2[idx];
```

7) send D2H data transfer with different streams

```
    LinearKernelTiledWithRelu<<<dims.grid2D2, dims.block1D2, 0, contexts[gpu_id].stream[0]>>>(
      contexts[gpu_id].linear1_ag, contexts[gpu_id].linear2_wg, contexts[gpu_id].linear2_bg, context
    // ReLUKernel<<<dims.reluGridDimLin3, dims.reluBlockDimLin3, 0, contexts[gpu_id].stream[0]>>>(
    //   contexts[gpu_id].linear2_ag, linear2_a->num_elem());
    // Linear layers 4

    // convertFloatToHalf(contexts[gpu_id].linear2_ag, contexts[gpu_id].linear2_agh, linear2_a->num_
    // TensorCoreLinearReluKernel<<<dims.grid2D3, dims.block1D3, 0, contexts[gpu_id].stream[0]>>>(
    //   contexts[gpu_id].linear2_ag, contexts[gpu_id].linear3_wg, contexts[gpu_id].linear3_bg, cont

    LinearKernelTiled2<<<dims.grid2D3, dims.block1D3, 0, contexts[gpu_id].stream[0]>>>(
      contexts[gpu_id].linear2_ag, contexts[gpu_id].linear3_wg, contexts[gpu_id].linear3_bg, context

    // Copy the final output back to host asynchronously
    cudaStreamSynchronize(contexts[gpu_id].stream[0]);
    cudaMemcpyAsync(batchOutput, contexts[gpu_id].linear3_ag, BATCH_SIZE * M3 * sizeof(float),
                    cudaMemcpyDeviceToHost, contexts[gpu_id].stream[4]);
}
```

8) transfer cpu function to CUDA kernel

```
__global__ void EmbeddingPermuteKernel(int *in, float *w, float *out, size_t B, size_t S, size_t H) {
    // Calculate the global thread index
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Total number of elements across all batches
    size_t total_elements = B * S * H;

    // Return if the thread index exceeds the total number of elements
    if (idx >= total_elements) return;

    // Calculate the batch index, sequence index, and hidden dimension index
    size_t batch_idx = idx / (S * H);           // Batch index
    size_t seq_hidden_idx = idx % (S * H);      // Position within the sequence and hidden dims
    size_t seq_idx = seq_hidden_idx / H;        // Sequence index
    size_t hidden_idx = seq_hidden_idx % H;     // Hidden dimension index

    // Retrieve the vocabulary index for this input
    int vocab_idx = in[batch_idx * S + seq_idx];
    if (vocab_idx < 0 || vocab_idx >= 21635) return; // Check bounds for vocabulary size

    // Perform embedding lookup and permutation in one step
    out[batch_idx * (S * H) + hidden_idx * S + seq_idx] = w[vocab_idx * H + hidden_idx];
}
```

```
__global__ void ConcatKernelOneN(const float *in1, const float *in2, const float *in3, const float *in4,
                                 float *out, size_t B, size_t N) {
    // Shared memory for inputs
    __shared__ float shared_in1[ELEMENTS_PER_THREAD * BLOCK_SIZE];
    __shared__ float shared_in2[ELEMENTS_PER_THREAD * BLOCK_SIZE];
    __shared__ float shared_in3[ELEMENTS_PER_THREAD * BLOCK_SIZE];
    __shared__ float shared_in4[ELEMENTS_PER_THREAD * BLOCK_SIZE];

    // Calculate the total number of elements per batch
    size_t total_size_per_batch = 4 * N;  // Since N1, N2, N3, N4 are equal

    // Global thread index, adjusted to process multiple elements per thread
    size_t idx = (blockIdx.x * blockDim.x + threadIdx.x) * ELEMENTS_PER_THREAD;

    // Ensure we are within bounds
    if (idx >= B * total_size_per_batch) return;

    // Determine the batch index
    size_t b = idx / total_size_per_batch;
    size_t offset_within_batch = idx % total_size_per_batch;

    // Load data into shared memory
    #pragma unroll
    for (int i = 0; i < ELEMENTS_PER_THREAD; i++) {
        if (offset_within_batch + i < N) {
            shared_in1[threadIdx.x * ELEMENTS_PER_THREAD + i] = in1[b * N + offset_within_batch + i];
            shared_in2[threadIdx.x * ELEMENTS_PER_THREAD + i] = in2[b * N + offset_within_batch + i];
            shared_in3[threadIdx.x * ELEMENTS_PER_THREAD + i] = in3[b * N + offset_within_batch + i];
            shared_in4[threadIdx.x * ELEMENTS_PER_THREAD + i] = in4[b * N + offset_within_batch + i];
        }
    }

    __syncthreads();

    // Copy data from shared memory to the output
    #pragma unroll
    for (int i = 0; i < ELEMENTS_PER_THREAD; i++) {
        if (offset_within_batch + i < N) {
            out[b * total_size_per_batch + offset_within_batch + i] = shared_in1[threadIdx.x * ELEMENTS_PE
            out[b * total_size_per_batch + offset_within_batch + N + i] = shared_in2[threadIdx.x * ELEMENT
            out[b * total_size_per_batch + offset_within_batch + 2 * N + i] = shared_in3[threadIdx.x * ELE
            out[b * total_size_per_batch + offset_within_batch + 3 * N + i] = shared_in4[threadIdx.x * ELE
        }
    }
}
```

9) apply float4 operation to linearKernel

```
__global__ void LinearKernelTiled2(const float * __restrict__ in,
                                   const float * __restrict__ w,
                                   const float * __restrict__ bias,
                                   float * __restrict__ out,
                                   size_t B, size_t N, size_t M)
{
    // Each block processes one batch element (z-dimension)
    size_t b = blockIdx.z;

    // m: output feature index
    size_t m = blockIdx.y * BLOCK_SIZE + threadIdx.y;

    // We only have one block in x-dim, since we loop over N tiles inside the
    // threadIdx.x indexes a chunk of 4 floats along N dimension
    // nBase is the starting N index for this thread in float terms
    size_t nBase = threadIdx.x * 4;

    // Shared memory for tiles
    __shared__ float4 tileIn[2][BLOCK_SIZE][BLOCK_SIZE/4];
    __shared__ float4 tileW[2][BLOCK_SIZE][BLOCK_SIZE/4];

    float val = 0.0f;
    int bufIdx = 0;

    // Number of N-tiles
    size_t numTiles = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;

    for (size_t t = 0; t < numTiles; t++) {
        // Compute the N offset for this tile
        size_t nOffset = t * BLOCK_SIZE;

        // Load W tile:
        float4 wVal = make_float4(0.f, 0.f, 0.f, 0.f);
        size_t wN = nOffset + nBase; // wN is the global N index for w
        if (m < M && (wN + 3) < N) {
            wVal = *(const float4*)(&w[m * N + wN]);
        }
        tileW[bufIdx][threadIdx.y][threadIdx.x] = wVal;

        // Load In tile:
        // For 'in', we don't have M dimension. We just have B and N.
        // Each thread will load one float4 from the input vector for the give
        float4 iVal = make_float4(0.f, 0.f, 0.f, 0.f);
        if (b < B && (wN + 3) < N) {
```

출력값(성능 및 성능 화면 캡쳐 이미지)

성능: 4290.546629 (sentences/sec)

캡쳐 이미지:

```
shpc125@login2:~/final$ ./run.sh -v
salloc: Pending job allocation 1180998
salloc: job 1180998 queued and waiting for resources
salloc: job 1180998 has been allocated resources
salloc: Granted job allocation 1180998

=============================================
 Model: Sentiment Analysis
---------------------------------------------
 Validation: ON
 Warm-up: OFF
 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!
Predicting sentiment...Done!
Elapsed time: 3.818628 (sec)
Throughput: 4290.546629 (sentences/sec)
Finalizing...Done!
Saving outputs to ./data/outputs.bin...Done!
Validating...PASSED!
salloc: Relinquishing job allocation 1180998
```