Assignment 3

In this assignment, you'll extend the work from Assignment 1 and 2 to speed up the transformer model for more efficient training and inference. You will focus on optimizing the Softmax and LayerNorm batch reduction operations by writing custom CUDA code.

The CUDA optimizations are based on methods from the lightseq2 paper [1]. We **strongly encourage** you to refer to this paper and the relevant lecture slides while working on this assignment. Before start writing the CUDA code, make sure you have read through the writeup and understood what each kernel is doing.

Setting up the code

The starting code base is provided in `https://github.com/llmsystem/llmsys_s24_hw3`. You will need to merge it with your implementation in the previous assignments. Here are our suggested steps:

1. Install the requirements and miniTorch:

```
pip install -r requirements.extra.txt
pip install -r requirements.txt
pip install -e .
```

2. Copy the CUDA kernels `combine.cu` from Assignment 2 and compile it:

```
combine.cu -> src/combine.cu
bash compile_cuda.sh
```

3. Copy `autodiff.py` from Assignment 1:

```
autodiff.py -> minitorch/autodiff.py
```

4. Keep copying several other functions from Assignment 2 when completing this one.

Problem 1.1: Softmax Forward (20)

In this part, you will implement a fused kernel of softmax in the attention mechanism. The softmax function for a vector $\mathbf{x}$ is given by:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \tag{1}$$

where x_i is the i-th element of $\mathbf{x}$.

The kernel also incorporates implementation of attention mechanisms, particularly in its handling of attention masks. In attention mechanisms, masks are used to control the focus of the model on certain parts of the input.

Assignment 3

1. Implement the CUDA kernel of Softmax in `src/softmax_kernel.cu`.

```
template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax(T *inp, ...) {
    ...
}

template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax_lt32(T *inp, ...) {
    ...
}
```

Note that `ker_attn_softmax_lt32` and `ker_attn_softmax` are almost identical except that `ker_attn_softmax_lt32` is designed for sequence length less than 32, and therefore does not require block-level parallelism. To give you an easy start, we provide you with the implementation of `ker_attn_softmax_lt32`. You should go over its implementation with the explanation below, and then implement `ker_attn_softmax` yourself.

2. Compile the CUDA file.

```
>>> nvcc -o minitorch/cuda_kernels/softmax_kernel.so --shared
↪  src/softmax_kernel.cu -Xcompiler -fPIC
```

3. Bind the kernel with miniTorch in `minitorch/cuda_kernel_ops.py` (this is given as an example for binding).
Hint: you should pass cuda stream to the function, define it with
`stream = torch.cuda.current_stream().cuda_stream`

```
class CudaKernelOps(TensorOps):
    @staticmethod
    def attn_softmax_fw(inp: Tensor, mask: Tensor):
        ...
```

and in `minitorch/tensor_functions.py`:

```
class Attn_Softmax(Function):
    @staticmethod
    def forward(ctx: Context, inp: Tensor, mask: Tensor) -> Tensor:
      ...
```

4. Pass the test and notice an average speedup around 6.5×.

```
>>> python kernel_tests/test_softmax_fw.py
```

Understanding Softmax Forward Kernels

As described in the paper and lectures (slide 26), The `ker_attn_softmax_lt32` and `ker_attn_softmax` kernels are differentiated mainly by their approach to reduction operations:

- **ker_attn_softmax_lt32:** Utilizes warp-level primitives for reduction, suitable for smaller input sizes. This method allows for efficient parallel reduction without the need for block-wide synchronization.
- **ker_attn_softmax:** Employs block-level reduction techniques, making it more suitable for larger input sizes. It involves two phases of reduction (max and sum) followed by a normalization step, with synchronization points to ensure consistency across threads.

Algorithmic Steps

The softmax computation in both kernel can be divided into three main steps:

1. **Compute Max:** Identifying the maximum value for normalization, to avoid numerical overflow in the exponential step.
2. **Compute Exponential and Sum:** Calculating the exponentials of the normalized values and their sum for normalization.
3. **Compute Final Result:** Normalizing the exponentials with the sum to obtain softmax probabilities. Store the results using CUB library's `BlockStore` to minimize memory transactions.

Computing the Maximum Value for Softmax Normalization

The implementation of this part are identical in the two kernels. You should go over the code in `ker_attn_softmax_lt32` and understand how this is implemented.
First, compute max on each thread (thread local max):

1. **Initialization:** Two arrays are declared:
   - 'val[token_per_reduce][ele_per_thread]' for storing intermediate values, including any adjustments from the attention mask.
   - 'l_max[token_per_reduce]' for recording the maximum value found for each token, initialized with 'REDUCE_FLOAT_INF_NEG' to ensure that any actual input value will be larger.
2. **Iterative Computation:** iterates over each token and its elements in two nested loops. For each element:
   (a) **Future Token Masking:** If future tokens are to be masked ('mask_future' is 'true'), and the element index suggests it is a future token, 'temp_val' is set to 'REDUCE_FLOAT_INF_NEG', excluding it from the max computation.

(b) **Attention Mask Adjustment:** If an attention mask is provided, its corresponding value is added to the input value, adjusting it based on the mask.

(c) **Intermediate Value Storage:** The adjusted value ('temp_val') is stored in the 'val' array for subsequent steps.

(d) **Maximum Value Update:** The 'fmaxf' function updates 'l_max' to the maximum value between its current value and 'temp_val', ensuring that after all iterations, 'l_max' holds the maximum value for each token.

After getting thread local max, both kernels reduce the results for block max:

- The `ker_attn_softmax_lt32` kernel performs warp-level reduction using a custom `warpReduce` function.
- The `ker_attn_softmax` kernel conducts a block-wide reduction using CUB library's `BlockLoad` primitive and uses shared memory to distribute the max value among threads.

Problem 1.2: Softmax Backward (20)

The gradient of the softmax function for a vector $\mathbf{x}$ is given by:

$$\frac{\partial \text{softmax}(\mathbf{x})_i}{\partial x_j} = \text{softmax}(\mathbf{x})_i(\delta_{ij} - \text{softmax}(\mathbf{x})_j) \tag{2}$$

where δ_{ij} is the Kronecker delta.
1. Implement `launch_attn_softmax_bw` in `src/softmax_kernel.cu`.

```
void launch_attn_softmax_bw(float *out_grad,
                            const float *soft_inp, int rows,
                            int softmax_len,
                            cudaStream_t stream)
```

In lectures we described the use of templates for tuning kernel parameters. When implementing `launch_attn_softmax_bw`, you should compute the ITERATIONS parameter of `ker_attn_softmax_bw` depending on different max sequence lengths in `{32, 64, 128, 256, 384, 512, 768, 1024, 2048}`.
Hint: refer to the way templates are used in `launch_attn_softmax`.

```
template <typename T, int ITERATIONS>
__global__ void ker_attn_softmax_bw(T *grad, ...) {
    ...
}
```

2. Compile the CUDA file.

```
>>> nvcc -o minitorch/cuda_kernels/softmax_kernel.so --shared
↪  src/softmax_kernel.cu -Xcompiler -fPIC
```

3. Bind the kernel with miniTorch in `minitorch/cuda_kernel_ops.py`.
Hint: you should pass cuda stream to the function, define it with
`stream_1 = torch.cuda.current_stream().cuda_stream`

```
class CudaKernelOps(TensorOps):
    @staticmethod
    def attn_softmax_bw(out_grad: Tensor, soft_inp: Tensor):
        ...
```

and in `minitorch/tensor_functions.py`:

```
class Attn_Softmax(Function):
    @staticmethod
    def backward(ctx: Context, out_grad: Tensor) -> Tensor:
      ...
```

4. Pass the test and notice an average speedup around 0.5 with our given default max lengths `{32, 64, 128, 256, 384, 512, 768, 1024, 2048}`. You can try other setups of max length and achieve a higher speedup, but it will not be graded.

```
>>> python kernel_tests/test_softmax_bw.py
```

Understanding Softmax Backward Kernel

The `ker_attn_softmax_bw` function is a CUDA kernel for computing the backward pass of the softmax function in self-attention mechanisms. Here are the steps:

Initialization

- The function calculates the backward pass for each element in the gradient and the output of the softmax forward pass.
- The grid and block dimensions are configured based on the batch size, number of heads, and sequence length.

Gradient Calculation

- The function iterates over the softmax length, with each thread handling a portion of the data.
- It loads the gradient and input (output of softmax forward) into registers.

- A local sum is computed for each thread, which is a key part of the gradient calculation for softmax.

Gradient Computation

1. The sum is shared across the warp using warp shuffle operations.
2. The final gradient for each element is computed by modifying the forward pass output with the computed sum.

Problem 2.1: LayerNorm Forward (20)

LayerNorm normalizes the input $\boldsymbol{x}$ by:

$$\boldsymbol{y}_i = \boldsymbol{\gamma}_i \cdot \frac{\boldsymbol{x}_i - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})} + \boldsymbol{\beta}_i, \tag{3}$$

where $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ are the mean and the standard deviation of $\boldsymbol{x}$ respectively, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are the learnable affine transform parameters in LayerNorm. Noting that Equation 3 takes two reduction operations since typical computation of the standard deviation requires the mean, meaning that they can not be computed in parallel.

The speedup can be achieved by computing the standard deviation with:

$$\sigma(\boldsymbol{x}) = \sqrt{\mu(\boldsymbol{x}^2) - \mu(\boldsymbol{x})^2 + \epsilon}, \tag{4}$$

where $\epsilon = 1e^{-8}$ is a small value added to the variance for numerical stability. By doing so, the means of $\boldsymbol{x}$ and $\boldsymbol{x}^2$, i.e. two batch reduction operations, can then be concurrently executed.

1. Implement the CUDA kernel of LayerNorm forward in `src/layernorm_kernel.cu`.

```
template <typename T>
__global__ void ker_layer_norm(T *ln_res, ...) {
    ...
}
```

2. Compile the CUDA file.

```
>>> nvcc -o minitorch/cuda_kernels/layernorm_kernel.so --shared
 ↪  src/layernorm_kernel.cu -Xcompiler -fPIC
```

3. Bind the kernel with miniTorch in `minitorch/cuda_kernel_ops.py`:
Hint: you should pass cuda stream to the function, define it with
`stream_1 = torch.cuda.current_stream().cuda_stream`

```
class CudaKernelOps(TensorOps):
    @staticmethod
    def layernorm_fw(inp: Tensor, gamma: Tensor, beta: Tensor):
        ...
```

and in `minitorch/tensor_functions.py`:

```
class LayerNorm(Function):
    @staticmethod
    def forward(ctx: Context, ...) -> Tensor:
      ...
      return out
```

4. Pass the test and notice an average speedup around 15.8×.

```
>>> python kernel_tests/test_layernorm_fw.py
```

Understanding LayerNorm Forward Kernels

In this kernel, we are going to use `float4` to speed up adding numbers. It can lead to improved performance when dealing with large datasets. The main advantage comes from the ability to process multiple data elements simultaneously, taking advantage of the SIMD (Single Instruction, Multiple Data) parallelism inherent in GPUs.

When working with CUDA programming and `float4`, we need to use `reinterpret_cast` to convert between types. In `src/layernorm_kernel.cu`, we give an example of how to compute the sum of $\boldsymbol{x}$ in step 1. In this example, `reinterpret_cast` is used to convert a float array `inp` to a `float4` vector `inp_f4`. Each thread within a block calculates `l_sum` for its assigned elements in `inp_f4`.

Algorithmic Steps

1. Compute the sums of $\boldsymbol{x}$ and $\boldsymbol{x}^2$ with `reinterpret_cast` by casting to `float4` for speedup
2. Compute reduce sum with `blockReduce` and add epsilon with `LN_EPSILON`
3. Compute layernorm result with `reinterpret_cast` by casting to `float4` for speedup

Problem 2.2: LayerNorm Backward (20)

Let $\hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}$, then the final gradient of $\boldsymbol{x}_i$ can be re-written into:

$$\nabla \boldsymbol{x}_i = \frac{\nabla \boldsymbol{y}_i \boldsymbol{\gamma}_i}{\sigma(\boldsymbol{x})} - \frac{1}{m\sigma(\boldsymbol{x})} \left(\sum_j \nabla \boldsymbol{y}_j \boldsymbol{\gamma}_j + \hat{\boldsymbol{x}}_i \sum_j \nabla \boldsymbol{y}_j \boldsymbol{\gamma}_j \hat{\boldsymbol{x}}_j \right), \tag{5}$$

where m is the dimension of $\boldsymbol{x}$, and $\nabla \boldsymbol{x}$ and $\nabla \boldsymbol{y}$ are the input and output gradients. The speedup can then be achieved by concurrently executing two batch reduction operations in the parentheses in Equation 5.

The gradients of $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$ are:

$$\nabla \boldsymbol{\gamma}_i = \sum_j \nabla \boldsymbol{y}_j \hat{\boldsymbol{x}}_j, \nabla \boldsymbol{\beta}_i = \sum_j \nabla \boldsymbol{y}_j. \tag{6}$$

1. Implement the CUDA kernel of LayerNorm backward in `src/layernorm_kernel.cu`.

```
template <typename T>
__global__ void ker_ln_bw_dinp(T *inp_grad, ...) {
    ...
}

template <typename T>
__global__ void ker_ln_bw_dgamma_dbetta(T *gamma_grad, ...){
    ...
}
```

2. Compile the CUDA file.

```
>>> nvcc -o minitorch/cuda_kernels/layernorm_kernel.so --shared
↪  src/layernorm_kernel.cu -Xcompiler -fPIC
```

3. Bind the kernel with miniTorch in `minitorch/cuda_kernel_ops.py`:
Hint: you should pass cuda stream to the function, define it with
`stream_1 = torch.cuda.current_stream().cuda_stream`

```
class CudaKernelOps(TensorOps):
    @staticmethod
    def layernorm_bw(out_grad: Tensor, ...):
        ...
```

and in `minitorch/tensor_functions.py`:

```
class LayerNorm(Function):
    @staticmethod
    def backward(ctx: Context, out_grad: Tensor) -> Tensor:
      ...
```

4. Pass the test and notice an average speedup around 3.7×.

```
>>> python kernel_tests/test_layernorm_bw.py
```

Understanding LayerNorm Backward Kernels

In this kernel, you are going to implement the backward function for the input $\boldsymbol{x}$ in function `ker_ln_bw_dinp`, and the one for the learnable parameters of LayerNorm $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$ in function `ker_ln_bw_dgamma_dbetta`.

The goal is to take advantage of CUDA's thread cooperation features, such as shared memory and shuffle operations, to efficiently perform the reduction within a thread block, so as to improve memory access patterns and reduce the overall computation time.

Input Gradients:

Initialization
Each thread is responsible for a specific element in the `inp_grad` array.

Algorithmic Steps

1. Compute $\nabla \boldsymbol{y}_i \boldsymbol{\gamma}_i$ with `reinterpret_cast` by casting to `float4` for speedup
2. Compute $\hat{\boldsymbol{x}}_i$ with `reinterpret_cast` by casting to `float4` for speedup
3. Compute reduce sum for $\nabla \boldsymbol{y}_i \boldsymbol{\gamma}_i$ and $\nabla \boldsymbol{y}_i \boldsymbol{\gamma}_i \hat{\boldsymbol{x}}_i$ with `blockReduce`
4. Compute final gradient

Gamma and Beta Gradients:

Initialization
Shared memory arrays `betta_buffer` and `gamma_buffer` are declared to store intermediate results within the thread block. CUDA thread blocks `cg::thread_block` and thread block tiles `cg::thread_block_tile` are used to organize threads.

Loop Over Rows
Threads in the y-dimension loop over rows, calculating partial gradients for each row based on the given inputs (`out_grad`, `inp`, `means`, `vars`).

Shared Memory Storage
The computed partial gradients values are stored in shared memory arrays `betta_buffer` and `gamma_buffer` in a tiled manner.

Reduction within Thread Block
Threads cooperate to perform a reduction operation on `betta_buffer` and `gamma_buffer` using `g.shfl_down` (shuffle down) operations along `threadIdx.y`. `g.shfl_down(value, delta)` is used to perform a warp-level reduction, where the value is a variable holding a partial result, and delta is the warp-wide offset specifying the distance by which the threads

should shuffle down. This helps avoid bank conflicts and improves warp-level parallelism, making it a powerful tool for efficient parallel reduction operations within a thread block.

Final Result Assignment
The final reduction result is assigned to the appropriate positions in the global output arrays (`gamma_grad` and `betta_grad`).

Algorithmic Steps

1. Compute the partial gradients by looping across inp rows
2. Store the partial gradients in the shared memory arrays
3. Compute the reduce sum of the shared memory arrays with `g.shfl_down`
4. Assign the final result to the correct position in the global output

Problem 3: Adopt Fused Kernels in Transformer (20)

The improved CUDA kernels are now bound with miniTorch library. Now integrate the improved CUDA kernels into the transformer in Assignment 2.

1. Replace the softmax and layernorm operations in MultiHeadAttention, TransformerLayer, and DecoderLM with your accelerated kernels in `minitorch/modules_transfomer.py`.

2. Train the transformer for one epoch, with and without using fused kernel, and record the running time.

```
>>> python project/run_machine_translation.py --use-fused-kernel False
>>> python project/run_machine_translation.py --use-fused-kernel True
```

3. According to Amdahl's law, the improvement should not be significant since only softmax and layernorm are improved, but you should still notice an average speedup around 1.1×.

Submission

Please submit the whole `llmsys_s24_hw3` as a zip on canvas.

References

[1] Xiaohui Wang, Yang Wei, Ying Xiong, Guyue Huang, Xian Qian, Yufei Ding, Mingxuan Wang, and Lei Li. LightSeq2: Accelerated training for transformer-based models on GPUs. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*, November 2022.