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 LightSeq and LightSeq2 papers. We **strongly encourage** you to refer to the papers and the relevant lecture slides while working on this assignment. Before starting to write the CUDA code, make sure you have read through the write-up and understood what each kernel is doing.

Setting up the Code

The starting codebase is provided in the following repository:
https://github.com/llmsystem/llmsys_s25_hw3

You will need to merge it with your implementation from the previous assignments. Here are our suggested steps:

Step 1: Install Requirements

Install the required dependencies and miniTorch by running the following commands:

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

Step 2: Copy and Compile CUDA Kernels

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

```
# From Assignment 2 to the current directory
cp <path_to_assignment_2>/combine.cu src/combine.cu

# Compile the CUDA kernels
bash compile_cuda.sh
```

Step 3: Copy Files from Assignment 1 and Assignment 2

Copy `autodiff.py` from **Assignment 1** to the specified location:

```
cp <path_to_assignment_1>/autodiff.py minitorch/autodiff.py
```

Hints for Copying and Pasting: When copying the `backpropagate()` function, make sure to double-check the implementation you wrote for Assignment 1. It's a good idea to set a default value of 0.0 before accumulating the gradients in `backpropagate()`. We recommend initializing the derivatives map for each unique_id to 0.0 outside the for loop.

Step 4: Incrementally Add Functions

Keep copying several other functions from Assignment 2 as needed to complete this assignment.

Copy `minitorch/nn.py` from **Assignment 2** to the specified location:

```
cp <path_to_assignment_2>/nn.py minitorch/nn.py
```

Copy `minitorch/modules_basic.py` from **Assignment 2** to the specified location:

```
cp <path_to_assignment_2>/modules_basic.py minitorch/modules_basic.py
```

Copy `minitorch/modules_transfomer.py` from **Assignment 2** to the specified location:

```
cp <path_to_assignment_2>/modules_transfomer.py minitorch/modules_transfomer.py
```

Copy `run_machine_translation.py` from **Assignment 2** to the specified location:

```
cp <path_to_assignment_2>/run_machine_translation.py run_machine_translation.py
```

Problem 1.1: Softmax Forward (20 points)

In this part, you will implement a fused kernel of softmax in the attention mechanism.

The softmax function for a vector $\mathbf{x}$ is defined as:

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

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

The kernel also incorporates attention mechanisms, particularly in its handling of attention masks. Attention masks are used to control the model's focus on specific parts of the input.

Instructions

1. **Implement the CUDA Kernel**
   Write the CUDA kernel for 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, ...) {
       ...
   }
   ```

   - The `ker_attn_softmax_lt32` kernel is already implemented for sequences shorter than 32 and does not require block-level parallelism.
   - Review the provided implementation of `ker_attn_softmax_lt32` and use it as a reference to implement `ker_attn_softmax`.
2. **Compile the CUDA File**
   Compile the CUDA file using the following command:

   ```
   nvcc -o minitorch/cuda_kernels/softmax_kernel.so --shared
   src/softmax_kernel.cu -Xcompiler -fPIC
   ```
3. **Bind the Kernel to miniTorch**
   In `minitorch/cuda_kernel_ops.py`, bind the kernel by passing the CUDA stream:

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

   In `minitorch/tensor_functions.py`, define the softmax forward function:

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

```
        ...
```

4. **Test the Implementation**
   Run the provided test script and ensure your implementation achieves an average speedup of approximately **6.5×**:

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

Understanding Softmax Forward Kernels

The `ker_attn_softmax_lt32` and `ker_attn_softmax` kernels are optimized for different input sizes:

- **`ker_attn_softmax_lt32`:**
  - Utilizes warp-level primitives for reduction.
  - Suitable for smaller input sizes (<32 sequence length).
  - Efficient parallel reduction without block-wide synchronization.
- **`ker_attn_softmax`:**
  - Employs block-level reduction techniques.
  - Suitable for larger input sizes.
  - Includes two phases of reduction (max and sum) followed by a normalization step with synchronization.

Algorithmic Steps

The softmax computation in both kernels consists of three main steps:

1. **Compute Max**
   - Identify the maximum value for normalization to avoid numerical overflow during exponentiation.
2. **Compute Exponentials and Sum**
   - Calculate the exponentials of normalized values and their sum for final normalization.
3. **Compute Final Result**
   - Normalize the exponentials by dividing by the sum to obtain softmax probabilities.

- Use CUB library's `BlockStore` to minimize memory transactions.

Computing the Maximum Value for Normalization

Both kernels implement the maximum value computation in the same way. Study the implementation in `ker_attn_softmax_lt32`.

Steps to Compute the Maximum Value:

1. **Thread Local Max:**
   Each thread computes a local maximum:
   - Intermediate values and attention mask adjustments are stored in `val[token_per_reduce][ele_per_thread]`.
   - The thread-local maximum is recorded in `l_max[token_per_reduce]`, initialized to `REDUCE_FLOAT_INF_NEG`.
2. **Future Token Masking:**
   If future tokens are masked, their values are excluded from the maximum computation by setting them to `REDUCE_FLOAT_INF_NEG`.
3. **Attention Mask Adjustment:**
   Adjust the input value by adding the corresponding attention mask value.
4. **Iterative Updates:**
   Update the thread-local maximum using `fmaxf`.

Block-Level Reduction for Global Max:

- **`ker_attn_softmax_lt32`:**
  - Uses a warp-level reduction with a custom `warpReduce` function.
- **`ker_attn_softmax`:**
  - Uses block-wide reduction with the CUB library's `BlockLoad` and shared memory for synchronization.

Problem 1.2: Softmax Backward (20)

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

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

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 the 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

- The sum is shared across the warp using warp shuffle operations.
- 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:

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

where $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ are the mean and the standard deviation of $\boldsymbol{x}$ respectively, and γ and $\boldsymbol{\beta}$ are the learnable affine transform parameters in LayerNorm.

Noting that the equation above requires two reduction operations (mean and standard deviation), these cannot be computed in parallel. Speedup can be achieved by computing the standard deviation as:

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

where $\epsilon = 1 \times 10^{-8}$ is a small value added to the variance for numerical stability. This approach allows concurrent computation of the means of $\boldsymbol{x}$ and $\boldsymbol{x}^2$.

Steps

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 the CUDA stream to the function, defining it as:

```
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 $15.8\times$:**

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

Understanding LayerNorm Forward Kernels

In this kernel, we use `float4` to speed up computations. This approach enhances performance when handling large datasets by processing multiple data elements simultaneously, leveraging the SIMD (Single Instruction, Multiple Data) parallelism inherent in GPUs.

When using CUDA programming and `float4`, `reinterpret_cast` is required to convert between types. For example, in `src/layernorm_kernel.cu`, the sum of \boldsymbol{x} in step 1 is computed as follows:

- `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} :**
    - Use `reinterpret_cast` by casting to `float4` for speedup.
2. **Perform reduction:**
    - Compute the reduce sum with `blockReduce` and add epsilon (`LN_EPSILON`).
3. **Compute the LayerNorm result:**
    - Use `reinterpret_cast` to cast 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 as:

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

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 be achieved by concurrently executing two batch reduction operations in the parentheses above.

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

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

Steps to Implement

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: Pass the CUDA stream to the function, defining it with:

```
stream_1 = torch.cuda.current_stream().cuda_stream
```

Example implementation:

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

Then integrate it 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 of approximately **3.7×**:

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

Understanding LayerNorm Backward Kernels

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 gradient 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`.
This approach avoids bank conflicts and improves warp-level parallelism.

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.

More hints about `g.shfl_down`: Read

```
https://developer.nvidia.com/blog/cooperative-
groups/#:~:text=Using%20thread_block_tile%3A%3Ashfl_down()%20to%20simplify%20our%20w
```

The highlighted line gives you a conceptual understanding of what the `g.shfl_down` is doing. Usually, the threads inside a block need to load everything to shared memory and work together to reduce the result (like what you have implemented in the hw1 for reduce function). Now `g.shfl_down` helps you do so without consuming any shared memory. `g.shfl_down` makes it more efficient.

Problem 3: Adopt Fused Kernels in Transformer (20)

The improved CUDA kernels are now bound with the miniTorch library.
Integrate the improved CUDA kernels into the transformer from 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 the 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. However, you should still notice an average speedup of approximately **1.1×**.

Submission

Please submit the entire `llmsys_s24_hw3` folder as a zip file on Canvas.