MSRA SH Triton Study Group
1. Naive CUDA Softmax Kernel

2025/07/04

Problem Definition

- Shape: batch size $M \geq 4096$, hidden dim $N = 4096$

- Mathemetical

$$X \in \mathbb{R}^{M \times N}$$
$$Y_{ij} = \mathrm{softmax}(X_i)_j$$
$$= \frac{\exp\left(X_{ij}\right)}{\sum_{j=1}^{N} \exp(X_{ij})}$$

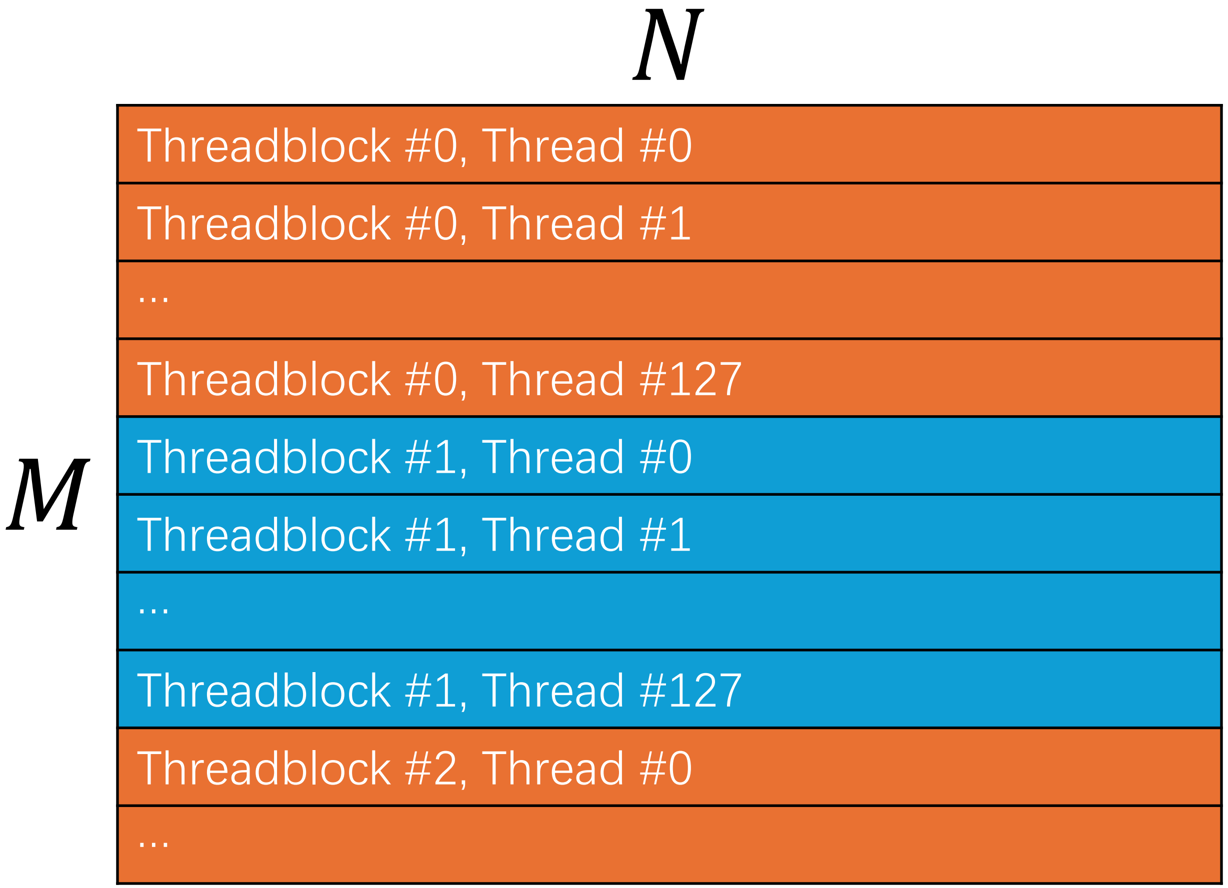
Standard Softmax Operation

- Input is a 32-bit float tensor $X \in \mathbb{R}^{M \times N}_{\mathbf{float32}}$
- Find maximum value for each row $M_i \leftarrow \max_{j=1}^{N} X_{ij}$
- Overflow-free exponential calculation $Z_{ij} \leftarrow \exp\left(X_{ij} - M_i\right)$
- Sum of exponentials for each row $S_i \leftarrow \sum_{j=1}^{N} Z_{ij}$
- Output $Y_{ij} \leftarrow \frac{Z_{ij}}{S_i}$

- Row-wise data dependency

Naïve Softmax Kernel

- 1 thread per row
- For each row i:
  - Loop to find M_i
  - Loop to calc S_i
  - Loop to calc & store Y_{ij}



```
// CUDA kernel for naive softmax implementation
__global__ void naive_softmax_kernel(float* x, float* y, int batch_size, int hidden_dim) {
    // Each thread processes one row of the input matrix x
    int row_idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Boundary check
    if (row_idx >= batch_size) return;

    // Calculate the maximum value in the row
    float max_val = -FLT_MAX;
    for (int i = 0; i < hidden_dim; i++) {
        float tmp_val = x[row_idx * hidden_dim + i];  // Read from global memory
        max_val = max(max_val, tmp_val);
    }

    // Calculate the sum of exponentials
    float sum_exp = 0.0f;
    for (int i = 0; i < hidden_dim; i++) {
        float tmp_val = x[row_idx * hidden_dim + i];  // Read from global memory
        sum_exp += expf(tmp_val - max_val);
    }

    // Write the softmax output
    for (int i = 0; i < hidden_dim; i++) {
        float tmp_val = x[row_idx * hidden_dim + i];  // Read from global memory
        y[row_idx * hidden_dim + i] = expf(tmp_val - max_val) / sum_exp;  // Write to global memory
    }
}
```

```cpp

// C++ function to call the naive softmax kernel
torch::Tensor naive_softmax(torch::Tensor X) {
    cudaSetDevice(X.get_device());

    int batch_size = X.size(0);
    int hidden_dim = X.size(1);
    torch::Tensor Y = torch::empty_like(X, X.options());

    // Thread block size
    const int num_threads = 128;
    // Grid size (= number of thread blocks)
    int num_blocks = (batch_size + num_threads - 1) / num_threads;

    // Launch the kernel
    const dim3 dimBlock(num_threads);
    const dim3 dimGrid(num_blocks);
    naive_softmax_kernel<<<dimGrid, dimBlock>>>(
        X.data_ptr<float>(),  // Pointer to input data
        Y.data_ptr<float>(),  // Pointer to output data
        batch_size, hidden_dim
    );

    return Y;
}

```

Naïve Softmax Kernel

- Obviously slow

```
Batch size: 8765, Hidden dim: 4096
Torch Softmax Latency: 0.20 ms
Naive Softmax Latency: 2.16 ms
```

- Low GPU utility
  - Thread block number: 4096 / 128 = 32 < physical SM number
- Redundant memory access
  - Register number: 255 < hidden dim
  - Repeat load input values from global memory (L2 cache)
  - Non-vectorized memory access: 32 discrete address for 32 threads

Better Softmax Kernel

- 1 warp per row
  - shfl_sync() for warp-level reduce
- For each row i:
  - Load all X_{ij} in once
  - Reduce to find M_i
  - Reduce to calc S_i
  - Store Y_{ij}

N

M

TB#0, T#0	TB#0, T#1	…	TB#0, T#31
TB#0, T#32	TB#0, T#33	…	TB#0, T#63
TB#0, T#64	TB#0, T#65	…	TB#0, T#95
TB#0, T#96	TB#0, T#97	…	TB#0, T#127
TB#1, T#0	TB#1, T#1	…	TB#1, T#31
TB#1, T#32	TB#1, T#33	…	TB#1, T#63
TB#1, T#64	TB#1, T#65	…	TB#1, T#95
TB#1, T#96	TB#1, T#97	…	TB#1, T#127
TB#2, T#0	TB#2, T#1	…	TB#2, T#31
…	…	…	…

Vectorized Memory Access

- GPU memory bus width: 128 bits = 4x float32 values

```
const int MEM_ACCESS_WIDTH = 4;
const int VALS_PER_THREAD = 128;
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i += MEM_ACCESS_WIDTH) {
    reinterpret_cast<float4*>(&tmp_val[i])[0] =
        reinterpret_cast<float4*>(&x[row_idx * N + lane_idx * VALS_PER_THREAD + i])[0];
}
```

- Warp-level access consecutive 128 Byte global memory in a cycle

```
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i += MEM_ACCESS_WIDTH) {
    reinterpret_cast<float4*>(&tmp_val[i])[0] =
        reinterpret_cast<float4*>(&x[row_idx * N + i * WARP_SIZE + lane_idx * MEM_ACCESS_WIDTH])[0];
}
```

Shuffle Sync

t = __shfl_xor_sync(m, laneMask=1)

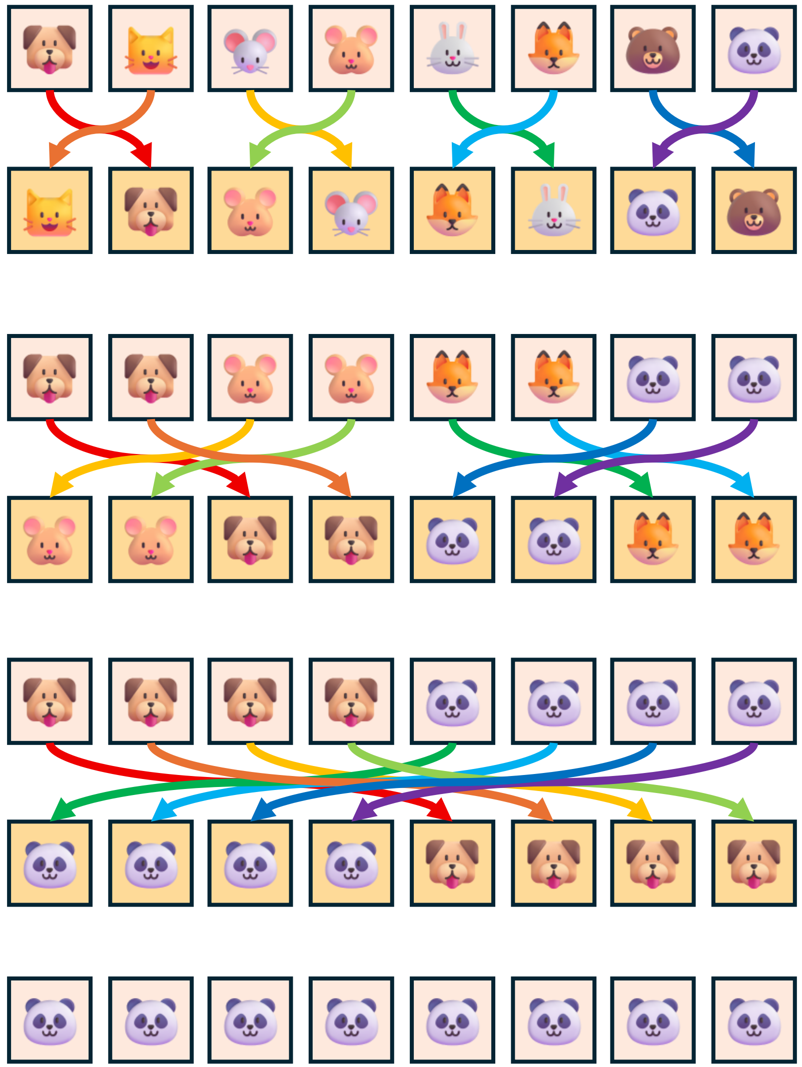
m = max(m, t)

t = __shfl_xor_sync(m, laneMask=2)

m = max(m, t)

t = __shfl_xor_sync(m, laneMask=4)

m = max(m, t)



Better Softmax Kernel

- Reasonable fast

```
Batch size: 8765, Hidden dim: 4096
Torch Softmax Latency: 0.20 ms
Naive Softmax Latency: 2.16 ms
Better Softmax Latency: 0.19 ms
```

- High GPU utility
  - Thread block number: 4096 / 4 = 1024 >> physical SM number
- Efficient memory access
  - Register number: 255 × 32 > hidden dim
  - Vectorized memory access
  - Contiguous global memory address in a warp

Homework

- Play with [TritonStudyGroup/1_CUDA_Softmax at main · Starmys/TritonStudyGroup]()

- Questions:
  - Calculate the theoretical minimum latency for input size [8765, 4096] on single A100 GPU
  - Can we load the entire row into L1 cache or shared memory in naïve softmax?
  - How much acceleration does warp-level contiguous global memory access bring? Why?
  - How does the best softmax implementation changes for inputs of different shapes?

Reading Materials

- How to Implement an Efficient Softmax CUDA Kernel— OneFlow Performance Optimization | by OneFlow | Medium
- Using CUDA Warp-Level Primitives | NVIDIA Technical Blog