MSRA SH Triton Study Group
2. Triton Softmax Kernel

2025/07/11

What is Triton

- A language
  - An open-source Python-like programming language
  - Enables researchers with no CUDA experience to write highly efficient GPU code

- A compiler
  - Compile Triton language code to executable CUDA kernels
  - With higher readability than other existing DSLs.

Triton language and programming model

Triton	**CUDA**	**Hardware**
• Kernel	• Kernel	• GPU
• Program	• Thread Block	• Streaming Multiprocessor
• INVISIBLE	• (32 threads)	• Warp
• INVISIBLE	• Thread	• Lane

Triton language and programming model

- Vectorized and parallel programming
  - Define an array in registers or shared memory across a thread block

    tl.zeros(), tl.arange(), etc.
  - The compiler automatically choose the best parallelization scheme in a program (thread block)

- Torch-like functions for tensor operation
  - Element-wise basic computation: +, -, *, /, tl.exp(), tl.log(), etc.
  - Reduce on an axis: tl.sum(), tl.max(), etc.
  - Matrix multiplication: tl.dot()

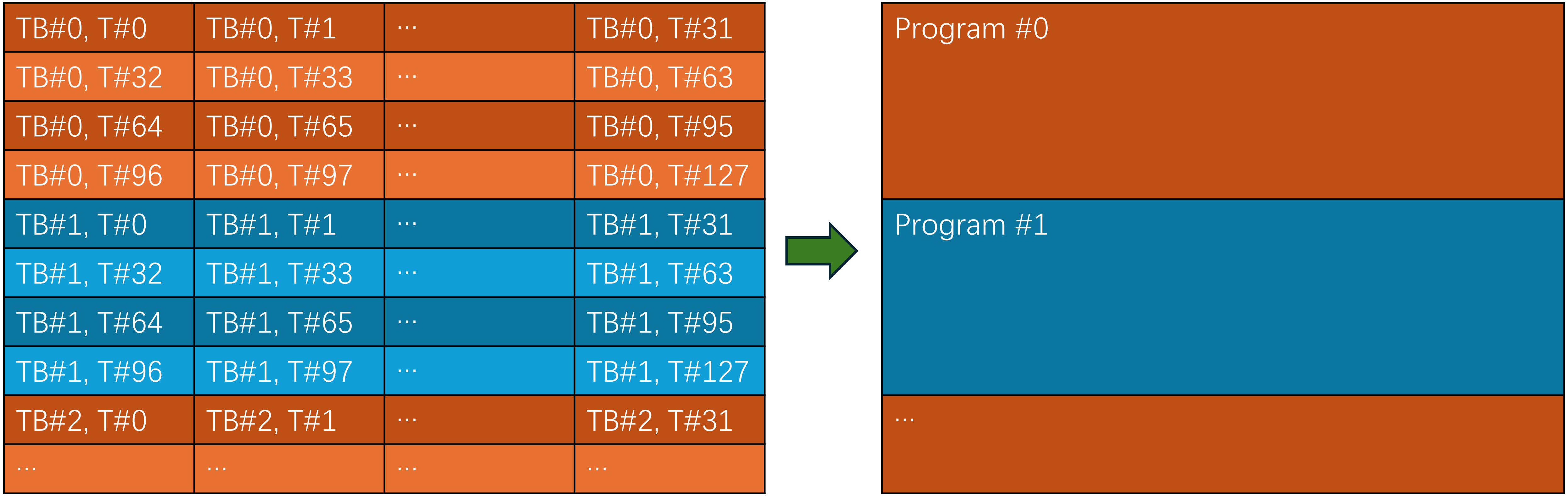
Triton Softmax Kernel

- Equivalent to better softmax kernel in the previous class

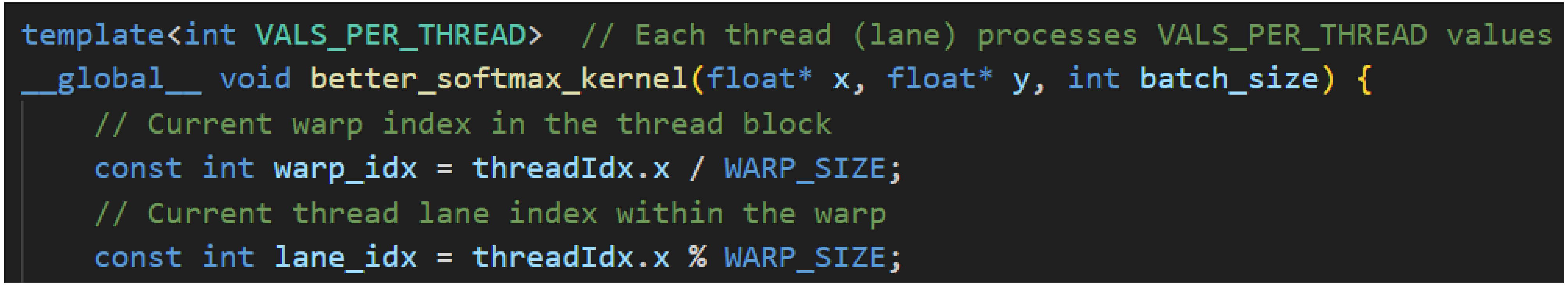
- 1 warp per row
  - 1 program (thread blocks) = 4 warps = 128 threads

- For each row i:
  - Load all X_{ij} in once
  - Reduce to find M_i
  - Reduce to calc S_i
  - Store Y_{ij}

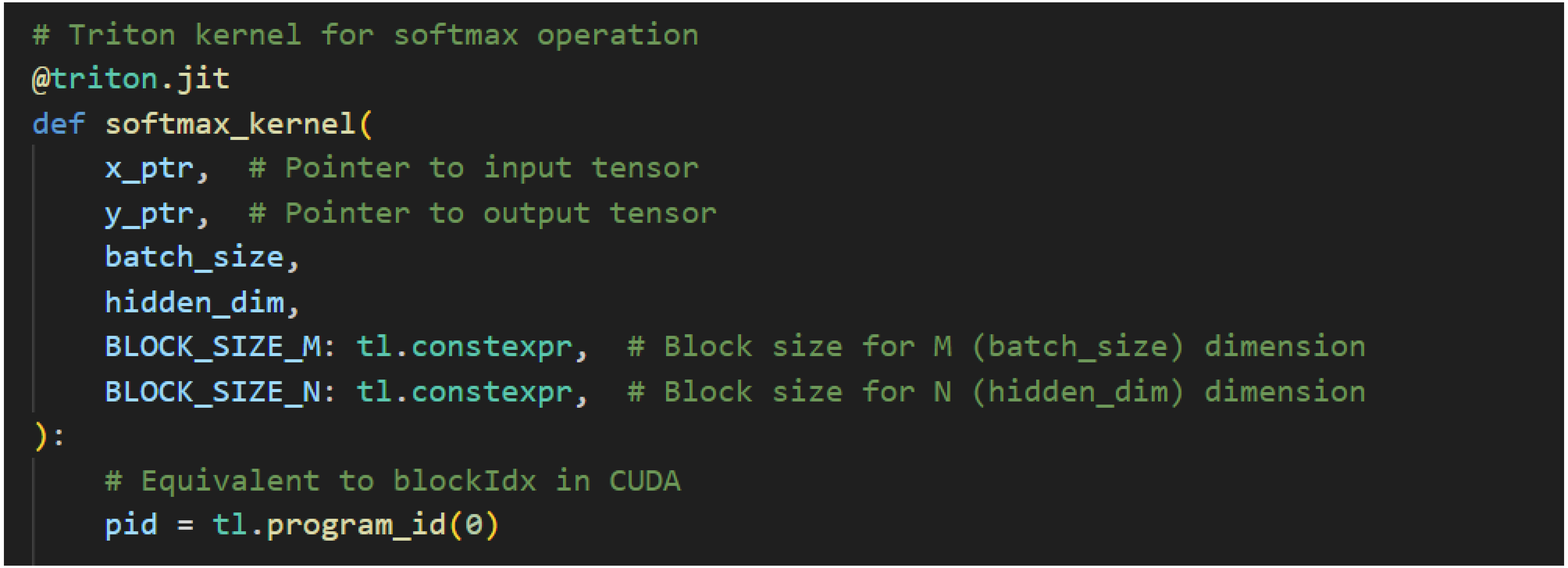
Triton Softmax Kernel



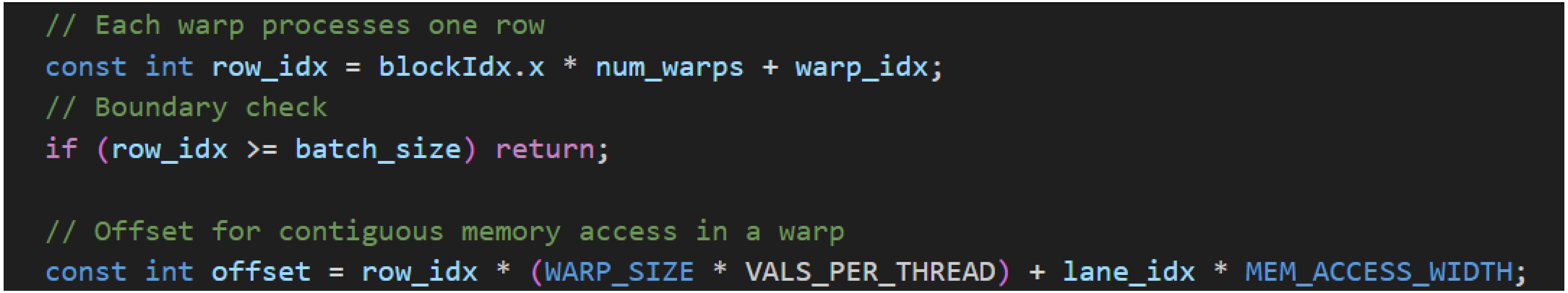
TB#0, T#0	TB#0, T#1	…	TB#0, T#31
TB#0, T#32	TB#0, T#33	…	TB#0, T#63
TB#0, T#64	TB#0, T#65	…	TB#0, T#95
TB#0, T#96	TB#0, T#97	…	TB#0, T#127
TB#1, T#0	TB#1, T#1	…	TB#1, T#31
TB#1, T#32	TB#1, T#33	…	TB#1, T#63
TB#1, T#64	TB#1, T#65	…	TB#1, T#95
TB#1, T#96	TB#1, T#97	…	TB#1, T#127
TB#2, T#0	TB#2, T#1	…	TB#2,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



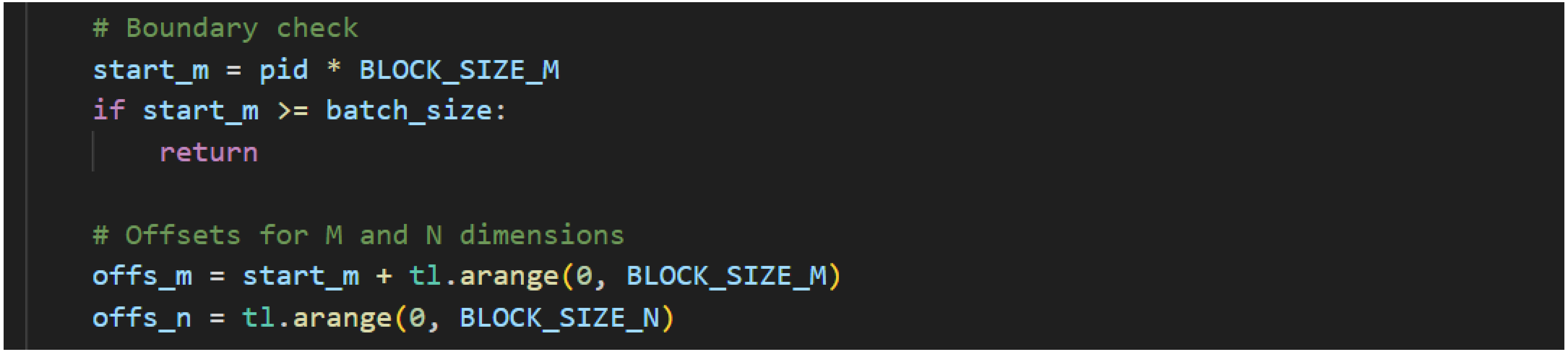
```
template<int VALS_PER_THREAD>  // Each thread (lane) processes VALS_PER_THREAD values
__global__ void better_softmax_kernel(float* x, float* y, int batch_size) {
    // Current warp index in the thread block
    const int warp_idx = threadIdx.x / WARP_SIZE;
    // Current thread lane index within the warp
    const int lane_idx = threadIdx.x % WARP_SIZE;
```



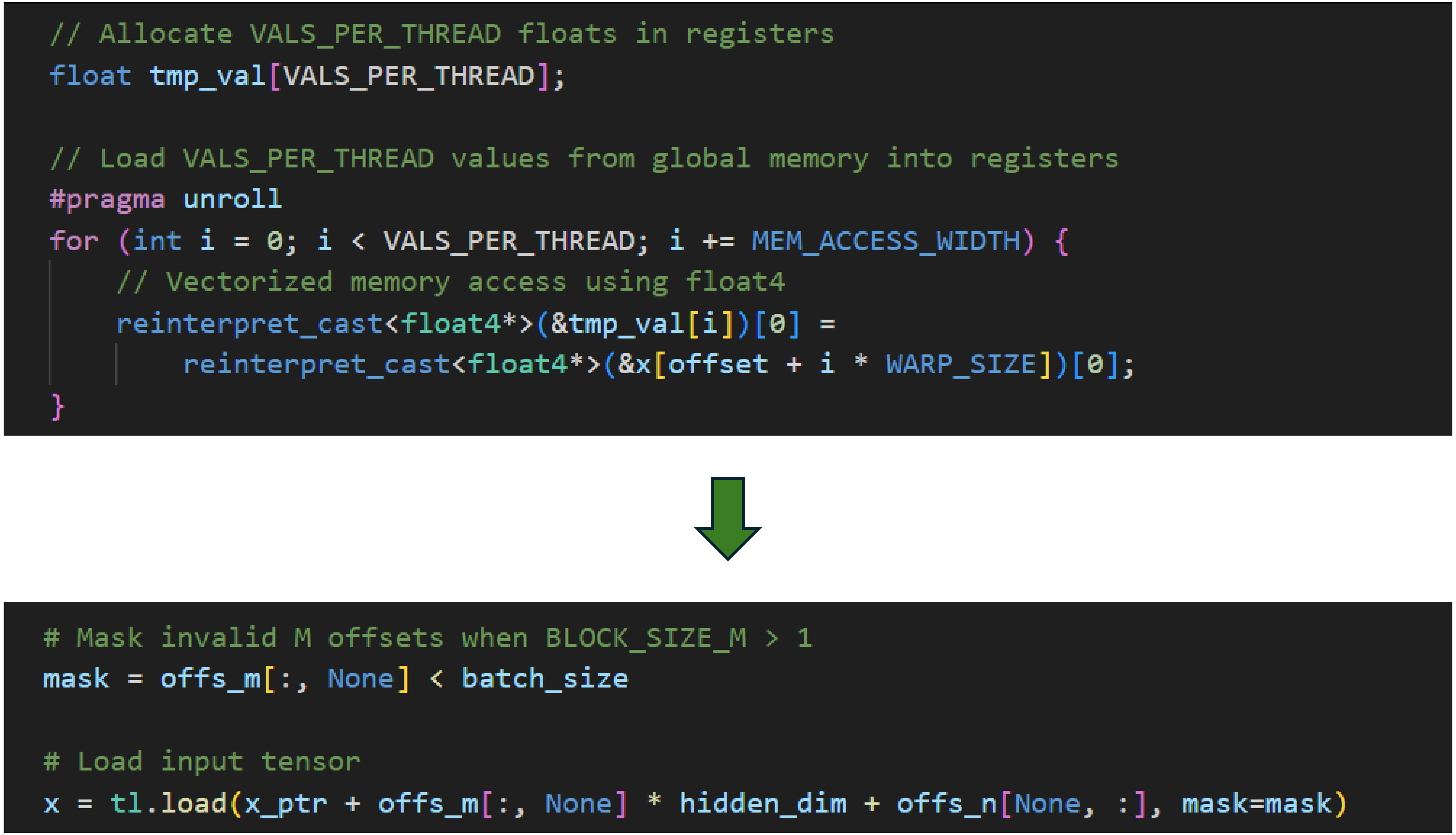
```
# Triton kernel for softmax operation
@triton.jit
def softmax_kernel(
    x_ptr,  # Pointer to input tensor
    y_ptr,  # Pointer to output tensor
    batch_size,
    hidden_dim,
    BLOCK_SIZE_M: tl.constexpr,  # Block size for M (batch_size) dimension
    BLOCK_SIZE_N: tl.constexpr,  # Block size for N (hidden_dim) dimension
):
    # Equivalent to blockIdx in CUDA
    pid = tl.program_id(0)
```



```
// Each warp processes one row
const int row_idx = blockIdx.x * num_warps + warp_idx;
// Boundary check
if (row_idx >= batch_size) return;

// Offset for contiguous memory access in a warp
const int offset = row_idx * (WARP_SIZE * VALS_PER_THREAD) + lane_idx * MEM_ACCESS_WIDTH;
```

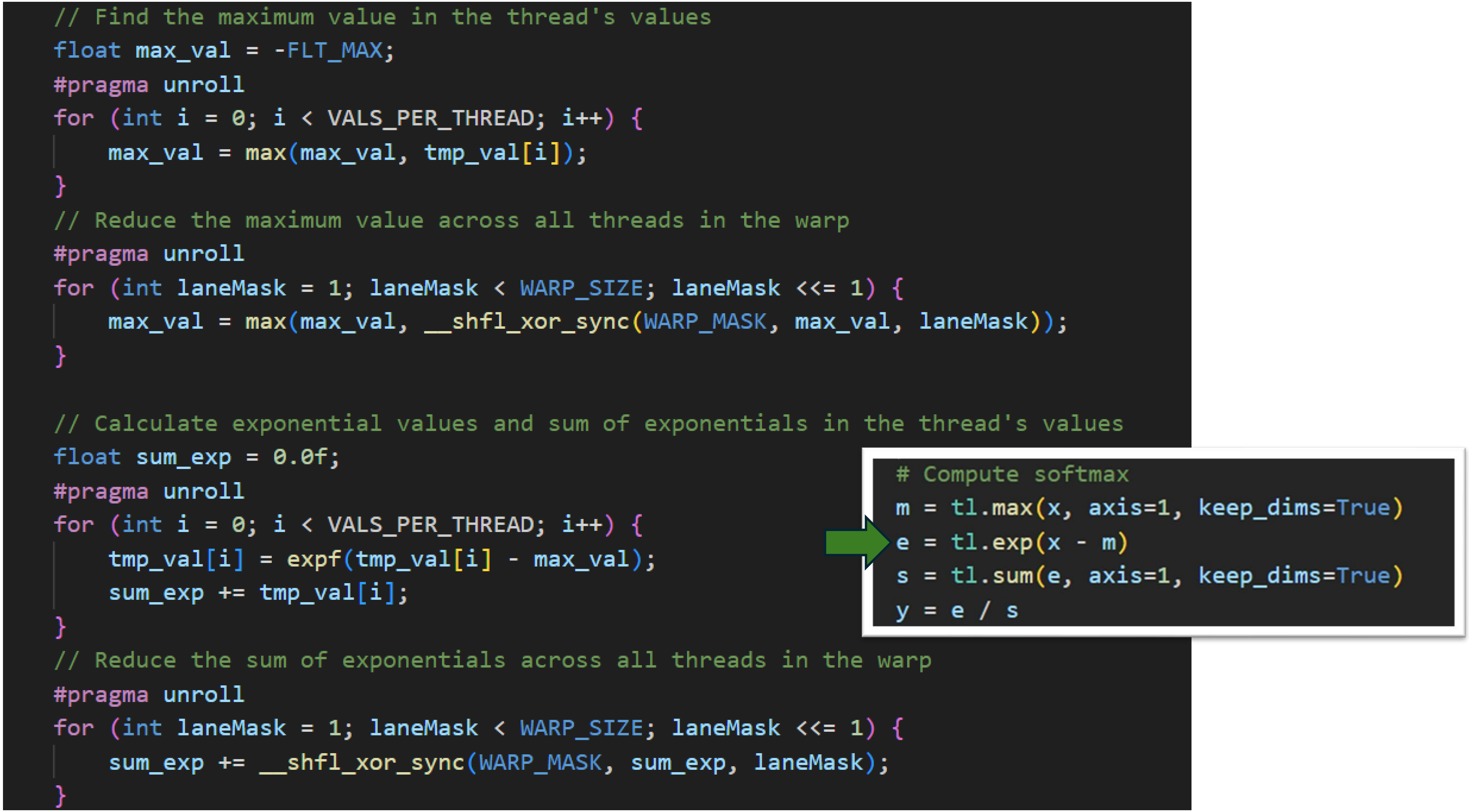


```
    # Boundary check
    start_m = pid * BLOCK_SIZE_M
    if start_m >= batch_size:
        return

    # Offsets for M and N dimensions
    offs_m = start_m + tl.arange(0, BLOCK_SIZE_M)
    offs_n = tl.arange(0, BLOCK_SIZE_N)
```



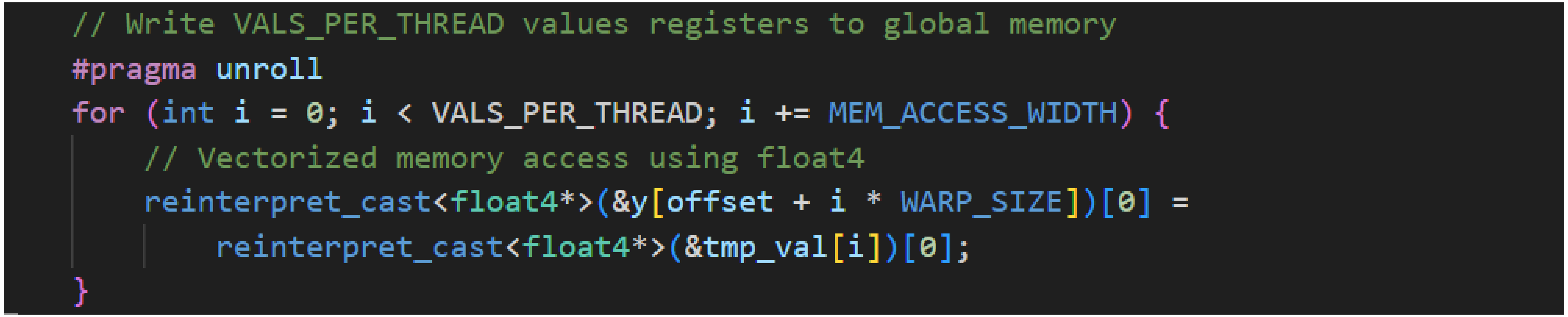
```
// Allocate VALS_PER_THREAD floats in registers
float tmp_val[VALS_PER_THREAD];

// Load VALS_PER_THREAD values from global memory into registers
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i += MEM_ACCESS_WIDTH) {
    // Vectorized memory access using float4
    reinterpret_cast<float4*>(&tmp_val[i])[0] =
        reinterpret_cast<float4*>(&x[offset + i * WARP_SIZE])[0];
}
```

```
# Mask invalid M offsets when BLOCK_SIZE_M > 1
mask = offs_m[:, None] < batch_size

# Load input tensor
x = tl.load(x_ptr + offs_m[:, None] * hidden_dim + offs_n[None, :], mask=mask)
```

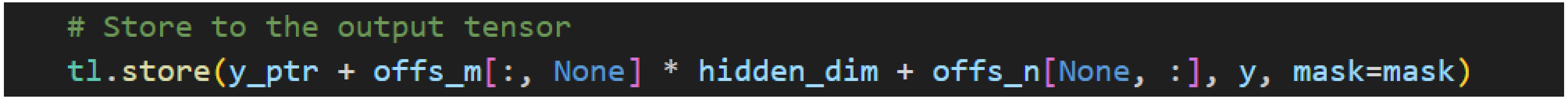


```
// Find the maximum value in the thread's values
float max_val = -FLT_MAX;
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i++) {
    max_val = max(max_val, tmp_val[i]);
}
// Reduce the maximum value across all threads in the warp
#pragma unroll
for (int laneMask = 1; laneMask < WARP_SIZE; laneMask <<= 1) {
    max_val = max(max_val, __shfl_xor_sync(WARP_MASK, max_val, laneMask));
}

// Calculate exponential values and sum of exponentials in the thread's values
float sum_exp = 0.0f;
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i++) {
    tmp_val[i] = expf(tmp_val[i] - max_val);
    sum_exp += tmp_val[i];
}
// Reduce the sum of exponentials across all threads in the warp
#pragma unroll
for (int laneMask = 1; laneMask < WARP_SIZE; laneMask <<= 1) {
    sum_exp += __shfl_xor_sync(WARP_MASK, sum_exp, laneMask);
}
```

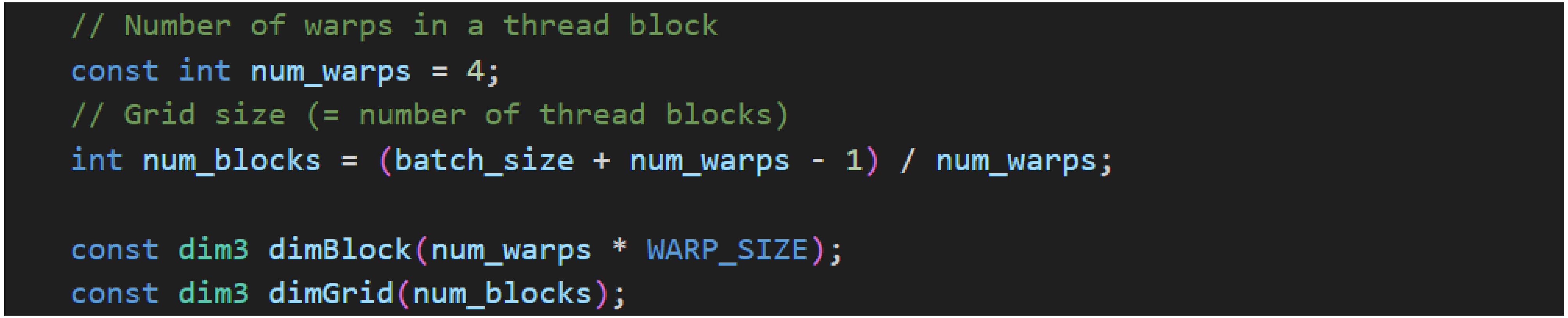
```
# Compute softmax
m = tl.max(x, axis=1, keep_dims=True)
e = tl.exp(x - m)
s = tl.sum(e, axis=1, keep_dims=True)
y = e / s
```



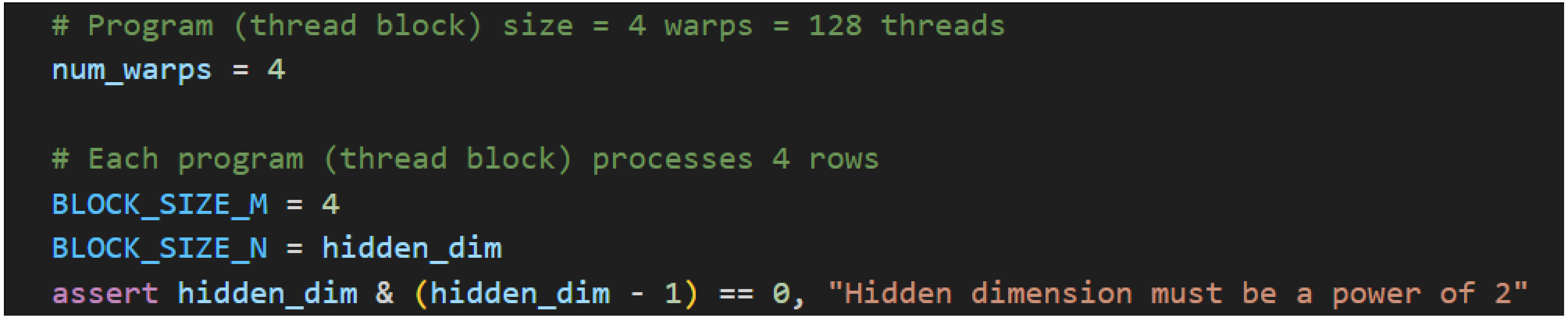
```
// Write VALS_PER_THREAD values registers to global memory
#pragma unroll
for (int i = 0; i < VALS_PER_THREAD; i += MEM_ACCESS_WIDTH) {
    // Vectorized memory access using float4
    reinterpret_cast<float4*>(&y[offset + i * WARP_SIZE])[0] =
        reinterpret_cast<float4*>(&tmp_val[i])[0];
}
```



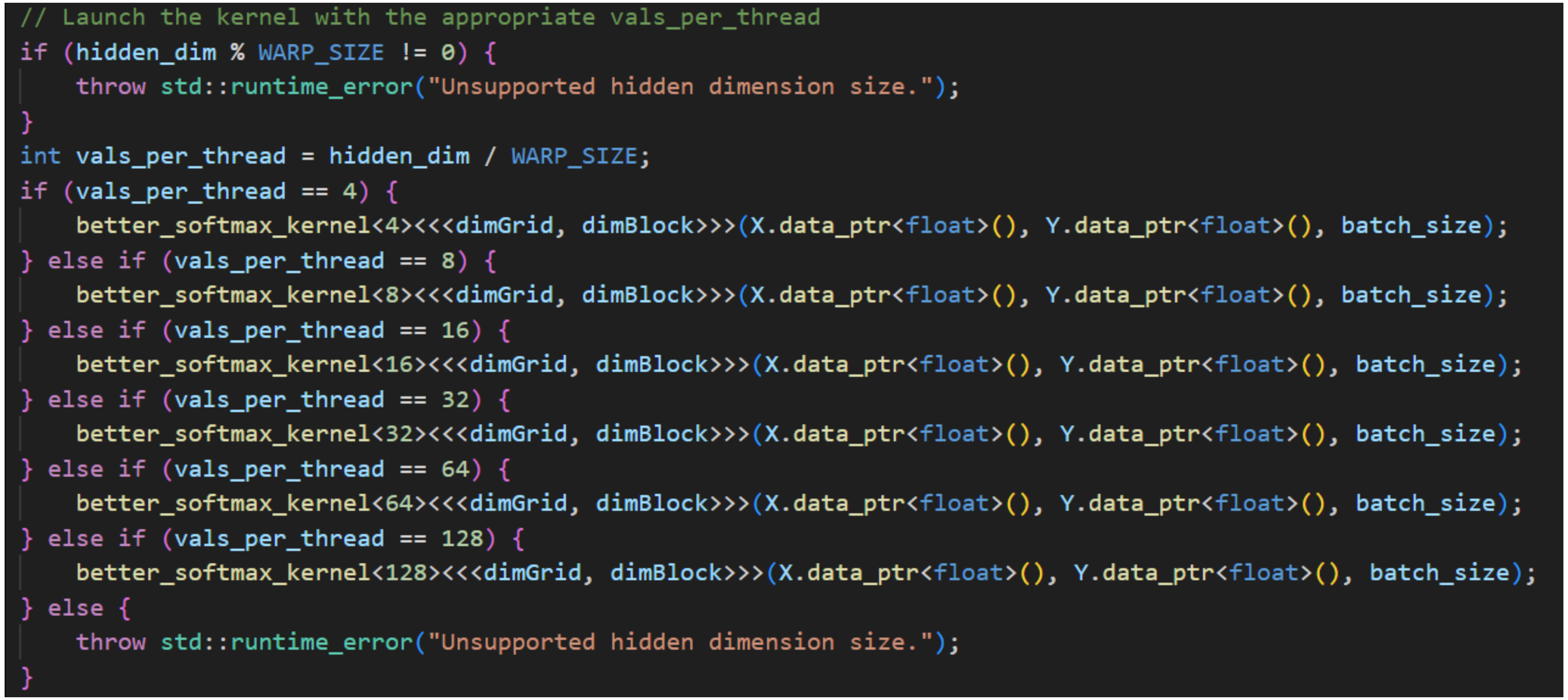
```
# Store to the output tensor
tl.store(y_ptr + offs_m[:, None] * hidden_dim + offs_n[None, :], y, mask=mask)
```



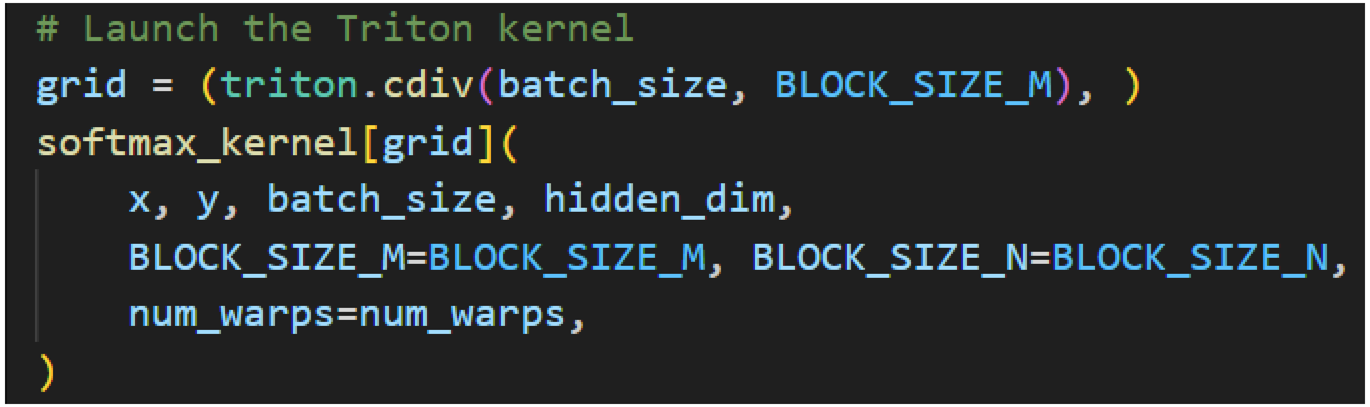
```
// Number of warps in a thread block
const int num_warps = 4;
// Grid size (= number of thread blocks)
int num_blocks = (batch_size + num_warps - 1) / num_warps;

const dim3 dimBlock(num_warps * WARP_SIZE);
const dim3 dimGrid(num_blocks);
```



```
# Program (thread block) size = 4 warps = 128 threads
num_warps = 4

# Each program (thread block) processes 4 rows
BLOCK_SIZE_M = 4
BLOCK_SIZE_N = hidden_dim
assert hidden_dim & (hidden_dim - 1) == 0, "Hidden dimension must be a power of 2"
```



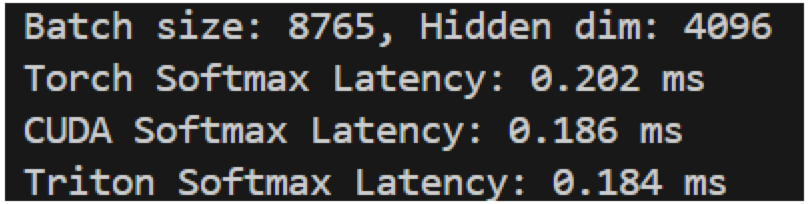
```
// Launch the kernel with the appropriate vals_per_thread
if (hidden_dim % WARP_SIZE != 0) {
    throw std::runtime_error("Unsupported hidden dimension size.");
}
int vals_per_thread = hidden_dim / WARP_SIZE;
if (vals_per_thread == 4) {
    better_softmax_kernel<4><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else if (vals_per_thread == 8) {
    better_softmax_kernel<8><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else if (vals_per_thread == 16) {
    better_softmax_kernel<16><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else if (vals_per_thread == 32) {
    better_softmax_kernel<32><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else if (vals_per_thread == 64) {
    better_softmax_kernel<64><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else if (vals_per_thread == 128) {
    better_softmax_kernel<128><<<dimGrid, dimBlock>>>(X.data_ptr<float>(), Y.data_ptr<float>(), batch_size);
} else {
    throw std::runtime_error("Unsupported hidden dimension size.");
}
```



```
# Launch the Triton kernel
grid = (triton.cdiv(batch_size, BLOCK_SIZE_M), )
softmax_kernel[grid](
    x, y, batch_size, hidden_dim,
    BLOCK_SIZE_M=BLOCK_SIZE_M, BLOCK_SIZE_N=BLOCK_SIZE_N,
    num_warps=num_warps,
)
```

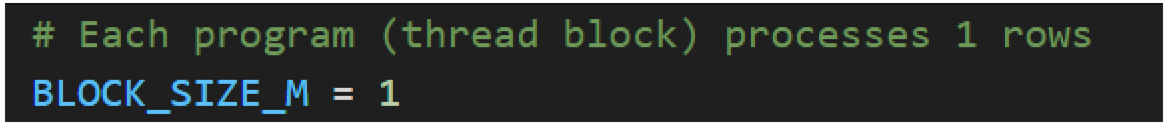
Triton Softmax Kernel

- Almost the same speed with CUDA better_softmax_kernel()

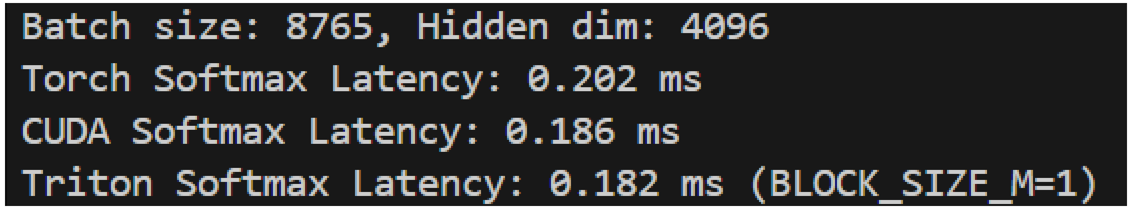


```
Batch size: 8765, Hidden dim: 4096
Torch Softmax Latency: 0.202 ms
CUDA Softmax Latency: 0.186 ms
Triton Softmax Latency: 0.184 ms
```

- Try different tile with low cost



```
# Each program (thread block) processes 1 rows
BLOCK_SIZE_M = 1
```



```
Batch size: 8765, Hidden dim: 4096
Torch Softmax Latency: 0.202 ms
CUDA Softmax Latency: 0.186 ms
Triton Softmax Latency: 0.182 ms (BLOCK_SIZE_M=1)
```

Homework

- Play with TritonStudyGroup/2_Triton_Softmax at main · Starmys/TritonStudyGroup

- Questions:
  - Implement a Triton softmax kernel for arbitrary hidden dimension
  - Which things have changed when we set BLOCK_SIZE_M=1?

Reading Materials

- Triton introduction and tutorials
  - Triton-lang/triton: Development repository for the Triton language and compiler
  - Introducing Triton: Open-source GPU programming for neural networks | OpenAI
  - Tutorials — Triton documentation

- Other AI kernel compilers
  - apache/tvm: Open deep learning compiler stack for cpu, gpu and specialized accelerators
  - ThunderKittens: Simple, Fast, and Adorable AI Kernels
  - TileLang: A Composable Tiled Programming Model for AI Systems