Triton for Beginners

Zhan Cheng

March 11, 2026

Contents

1 Basic Memory Operations

In the first stage of learning Triton, the most fundamental operations are `tl.load` and `tl.store`. These two primitives allow kernels to read from and write to GPU memory. Understanding how they work is essential before implementing more complex kernels. Before writing Triton kernels, we first import the Triton library and its language module:

```
import triton
import triton.language as tl
```

1.1 Demo 1: Basic Load with Mask

1.1.1 Goal

The purpose of this example is to understand how Triton reads data from memory using the primitive

`tl.load(ptr, mask, other)`

This operation loads values from memory while optionally masking invalid positions.

1.1.2 Kernel Implementation

```
@triton.jit
def demo1(x_ptr):
    range = tl.arange(0, 8)

    # print works in the interpreter
    print(range)

    x = tl.load(x_ptr + range, range < 5, 0)

    print(x)
```

1.1.3 Driver Code

The following Python function launches the Triton kernel.

```
def run_demo1():
    print("Demo1 Output: ")
    demo1[(1, 1, 1)](torch.ones(4, 3))
    print_end_line()
```

1.1.4 Kernel Launch Semantics

The kernel is launched with

$$\text{demo1}[(1,1,1)](...)$$

This launch configuration means

$$\text{grid} = (1,1,1)$$

Only one program instance executes the kernel. However, we do not need to worry about what these mean at first. The only thing we need to know is that `demo1` is registered with grid size $(1,1,1)$ and it is launched by

$$\text{demo1}[(1,1,1)](\text{tensor\_input})$$

1.1.5 Pointer Arithmetic

The line

```
range = tl.arange(0, 8)
```

creates the vector

$$[0,1,2,3,4,5,6,7],$$

similar to the `arange` function in Python. It can be understood as a set of index offsets. This vector is added to the base pointer

$$\text{x\_ptr} + \text{range},$$

which produces eight memory addresses:

$$[\text{x\_ptr}+0,\ \text{x\_ptr}+1,\ \text{x\_ptr}+2,\ \text{x\_ptr}+3,\ \text{x\_ptr}+4,\ \text{x\_ptr}+5,\ \text{x\_ptr}+6,\ \text{x\_ptr}+7].$$

This operation performs eight parallel memory reads.

1.1.6 Masking Mechanism

The load instruction is

```
x = tl.load(x_ptr + range, range < 5, 0)
```

The mask

$$\text{range} < 5$$

evaluates to

$$[1,1,1,1,1,0,0,0]$$

The first five elements are valid. The last three elements are masked.

Masked elements return the default value 0. This helps us load only part of the elements from a tensor.

1.1.7 Memory Layout

The input tensor is

```
torch.ones(4,3)
```

Even though the tensor is two-dimensional, Triton receives a pointer to the underlying contiguous memory.

PyTorch stores tensors in row-major order:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

which corresponds to the flattened memory layout

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

Therefore the kernel simply reads the first elements of this flattened array.

To be clearer, PyTorch stores tensors in row-major order:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

which corresponds to the flattened memory layout

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

1.1.8 Take Away

This example introduces two key Triton ideas:

- `tl.arange` creates index offsets
- `tl.load` reads elements from addresses given by these offsets and can be masked to prevent invalid memory access

1.2 Demo 2: Loading a 2D Block

1.2.1 Goal

The goal of this example is to understand how Triton constructs a two-dimensional access pattern. By combining two index ranges and using broadcasting, we can generate a 2D grid of memory offsets.

1.2.2 Kernel Implementation

```
@triton.jit
def demo2(x_ptr):
    i_range = tl.arange(0, 8)[:, None]
    j_range = tl.arange(0, 4)[None, :]

    range = i_range * 4 + j_range
    print(range)

    x = tl.load(x_ptr + range, (i_range < 4) & (j_range < 3), 0)
    print(x)
```

1.2.3 Driver Code

```
def run_demo2():
    print("Demo2 Output: ")
    demo2[(1, 1, 1)](torch.ones(4, 4))
    print_end_line()
```

1.2.4 Kernel Launch Semantics

The kernel is launched with

$$\texttt{demo2}[(1,1,1)](...)$$

which means the grid size is

$$(1,1,1)$$

Only one program instance executes the kernel. Similar to Demo 1, we do not need to consider multiple blocks yet.

1.2.5 Pointer Arithmetic

Two index ranges are created:

```
i_range = tl.arange(0, 8)[:, None]
j_range = tl.arange(0, 4)[None, :]
```

Similar to Demo 1, we first obtain the vectors

$$[0,1,2,3,4,5,6,7]$$

and

$$[0,1,2,3].$$

Then by using `[:, None]` and `[None, :]`, these vectors are reshaped for broadcasting:

$$\text{i\_range} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} \quad (8 \times 1)$$

$$\text{j\_range} = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix} \quad (1 \times 4)$$

This transformation is similar to the `unsqueeze()` function in PyTorch.
The flattened memory index is computed using broadcasting, forming an 8×4 grid.

$$\text{range} = i \times 4 + j$$

which produces

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ \vdots & & & \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

Each entry corresponds to a memory offset relative to x_ptr.

1.2.6 Masking Mechanism

The load instruction is

```
x = tl.load(x_ptr + range, (i_range < 4) & (j_range < 3), 0)
```

The mask condition

$$(i < 4) \wedge (j < 3)$$

creates the mask

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ \vdots & & & \end{bmatrix}$$

Only the upper-left 4×3 region is valid.
Masked elements return the default value 0.

1.2.7 Memory Layout

The input tensor is

```
torch.ones(4,4)
```

PyTorch stores tensors in row-major order:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

which corresponds to the flattened memory layout

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

The kernel reads values from this flattened memory according to the computed offsets.

1.2.8 Expected Output

First the index grid is printed:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 ...
 [28 29 30 31]]
```

Then the masked load result:

```
[[1. 1. 1. 0.]
 [1. 1. 1. 0.]
 [1. 1. 1. 0.]
 [1. 1. 1. 0.]
 [0. 0. 0. 0.]
 ...
]
```

1.2.9 Take Away

This example introduces two additional Triton ideas:

- `[:, None]` and `[None, :]` add an additional dimension and enable broadcasting during computation.
- Combining row and column indices to compute flattened memory offsets.

1.3 Demo 3: Writing with tl.store

1.3.1 Goal

The goal of this example is to understand how Triton writes values into memory using the primitive

`tl.store(ptr, value, mask)`.

Similar to `tl.load`, the store operation can also use a mask to avoid writing to invalid memory locations.

1.3.2 Kernel Implementation

```
@triton.jit
def demo3(z_ptr):
    range = tl.arange(0, 8)
    tl.store(z_ptr + range, 10, range < 5)
```

1.3.3 Driver Code

```
def run_demo3():
    print("Demo3 Output: ")
    z = torch.ones(4, 3)
    demo3[(1, 1, 1)](z)
    print(z)
    print_end_line()
```

1.3.4 Kernel Launch Semantics

The kernel is launched with

`demo3`$[(1, 1, 1)](...)$

which means the grid size is

$$(1, 1, 1).$$

Only one program instance executes the kernel.

1.3.5 Pointer Arithmetic

The line

```
range = tl.arange(0, 8)
```

creates the vector

$$[0, 1, 2, 3, 4, 5, 6, 7].$$

These values represent memory offsets relative to the base pointer z_ptr.
The store operation writes values to the addresses

$$[\text{z\_ptr} + 0, \text{z\_ptr} + 1, \ldots, \text{z\_ptr} + 7].$$

1.3.6 Masking Mechanism

The mask condition

$$\text{range} < 5$$

produces

$$[1, 1, 1, 1, 1, 0, 0, 0].$$

Only the first five positions are written with the value 10. The remaining positions remain unchanged.

1.3.7 Expected Output

For a 4×3 tensor initialized with ones:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The first five positions in the flattened memory layout are replaced with 10.

```
tensor([[10.,  10.,  10.],
        [10.,  10.,   1.],
        [ 1.,   1.,   1.],
        [ 1.,   1.,   1.]])
```

1.3.8 Take Away

- `tl.store` writes values into memory addresses.
- Masks prevent writing to invalid positions.
- Store operations use the same indexing logic as `tl.load`.

1.4 Demo 4: Multiple Blocks with Program ID

1.4.1 Goal

The goal of this example is to understand how Triton uses multiple program instances to process larger tensors. Each program instance processes a different portion of memory.

1.4.2 Kernel Implementation

```
@triton.jit
def demo4(x_ptr):
    pid = tl.program_id(0)

    range = tl.arange(0, 8) + pid * 8
    x = tl.load(x_ptr + range, range < 20)

    print("Print for each", pid, x)
```

1.4.3 Driver Code

```
def run_demo4():
    print("Demo4 Output: ")
    x = torch.ones(2, 4, 4)
    demo4[(3, 1, 1)](x)
    print_end_line()
```

1.4.4 Kernel Launch Semantics

The kernel is launched with

$$\texttt{demo4}[(3,1,1)](...)$$

which means the grid size is

$$(3,1,1).$$

The grid contains three programs whose coordinates are

$$(0,0,0),\ (1,0,0),\ (2,0,0).$$

Each coordinate represents a program instance.

1.4.5 Program ID

Each program instance obtains its unique identifier using

```
pid = tl.program_id(0)
```

Here `0` indicates the program axis. In this example we use axis 0, which corresponds to the first dimension of the grid.

The value of `pid` determines which portion of memory the program processes. It acts as an offset used to shift the memory pointer.

At the beginning, the first program instance has

$$\text{pid} = 0,$$

which corresponds to the program coordinate

$$(0, 0, 0).$$

1.4.6 Pointer Arithmetic

As shown in the first three demos, each program accesses the same number of addresses. Therefore, to access different parts of the tensor in the second and third programs, we need to compute an offset.

The memory offsets are computed as

$$\text{range} = [0, 1, \ldots, 7] + \text{pid} \times 8.$$

Therefore each program processes a different segment:

$$\text{pid} = 0 \rightarrow [0 \ldots 7]$$

$$\text{pid} = 1 \rightarrow [8 \ldots 15]$$

$$\text{pid} = 2 \rightarrow [16 \ldots 23].$$

1.4.7 Masking Mechanism

The load operation uses the mask

$$\text{range} < 20$$

because we assume that the tensor contains only 20 elements.

The final block attempts to read indices $20, 21, 22, 23$, which are invalid. These elements are masked and replaced with zeros.

1.4.8 Expected Output

```
Print for each 0 [1. 1. 1. 1. 1. 1. 1. 1.]
Print for each 1 [1. 1. 1. 1. 1. 1. 1. 1.]
Print for each 2 [1. 1. 1. 1. 0. 0. 0. 0.]
```

1.4.9 Take Away

- `tl.program_id(x)` identifies each program instance along axis x.
- Multiple program instances allow Triton kernels to process large tensors.
- Each program processes a different memory segment.

2 Triton Puzzles

2.1 Puzzle 1: Constant Add

2.1.1 Goal

In this puzzle, we implement a simple Triton kernel that adds a constant value to a vector.

Mathematically, the operation is

$$z_i = x_i + 10, \quad i = 0, \ldots, N_0 - 1.$$

We assume the block size B_0 is equal to the vector length N_0, so the entire vector can be processed in a single program instance.

2.1.2 Specification

The reference implementation is

```
def add_spec(x: Float32[32,]) -> Float32[32,]:
    return x + 10.0
```

2.1.3 Kernel Skeleton

```
@triton.jit
def add_kernel(x_ptr, z_ptr, N0, B0: tl.constexpr):
    # offsets
    off_x = tl.arange(0, B0)

    # load input
    x = tl.load(x_ptr + off_x)

    # TODO: finish the kernel
```

2.1.4 Solution

Solution

```
@triton.jit
def add_kernel(x_ptr, z_ptr, N0, B0: tl.constexpr):

    off_x = tl.arange(0, B0)

    # load
    x = tl.load(x_ptr + off_x)

    # computation
    z = x + 10

    # store
    tl.store(z_ptr + off_x, z)
```

2.1.5 Explanation

The kernel performs three steps:

- `tl.arange(0,B0)` creates vector offsets.
- `tl.load` reads elements from memory.
- The constant 10 is added element-wise.
- `tl.store` writes the result to the output pointer.

The element-wise addition means that the constant 10 is added to each element of the vector independently. If the loaded vector is

$$[x_0, x_1, x_2, \ldots, x_{B_0-1}],$$

then the result of the computation

$$z = x + 10$$

produces

$$[z_0, z_1, z_2, \ldots, z_{B_0-1}],$$

where

$$z_i = x_i + 10.$$

Since Triton operations are vectorized, this addition is performed simultaneously for all elements in the block rather than sequentially.

2.2 Puzzle 2: Constant Add with Blocks

2.2.1 Goal

In this puzzle, we again add a constant value to a vector. However, unlike Puzzle 1, the block size B_0 is smaller than the vector length N_0. Therefore multiple program instances are required to process the entire vector.

The computation is

$$z_i = x_i + 10, \quad i = 0, \ldots, N_0 - 1.$$

Each program processes a block of size B_0.

2.2.2 Specification

The reference implementation is

```
def add2_spec(x: Float32[200,]) -> Float32[200,]:
    return x + 10.0
```

2.2.3 Kernel Skeleton

```
@triton.jit
def add_mask2_kernel(x_ptr, z_ptr, N0, B0: tl.constexpr):

    pid = tl.program_id(0)

    off = pid * B0 + tl.arange(0, B0)

    x = tl.load(x_ptr + off, mask=off < N0)

    # TODO: finish the kernel
```

2.2.4 Solution

Solution

```
@triton.jit
def add_mask2_kernel(x_ptr, z_ptr, N0, B0: tl.constexpr):

    pid = tl.program_id(0)

    off = pid * B0 + tl.arange(0, B0)

    x = tl.load(x_ptr + off, mask=off < N0)

    z = x + 10

    tl.store(z_ptr + off, z, mask=off < N0)
```

2.2.5 Explanation

The kernel first obtains the program identifier

```
pid = tl.program_id(0)
```

Each program processes a block of size B_0. The offsets of the elements processed by the current program are

$$\text{off} = \text{pid} \times B_0 + [0, 1, \ldots, B_0 - 1].$$

This means that different programs process different segments of the vector.

Since the vector length N_0 may not be an exact multiple of the block size B_0, some offsets may exceed the valid range. Therefore we apply the mask

$$\text{off} < N_0$$

to prevent invalid memory access.

Finally, the constant 10 is added element-wise to each loaded value and the results are written back to memory using `tl.store`.

2.3 Puzzle 3: Outer Vector Add

2.3.1 Goal

In this puzzle, we compute the outer addition of two vectors. The result is a matrix where each element is the sum of an element from x and an element from y.

Mathematically, the operation is

$$z_{j,i} = x_i + y_j, \quad i = 0, \ldots, B_0 - 1, \quad j = 0, \ldots, B_1 - 1.$$

The output tensor therefore has shape

$$B_1 \times B_0.$$

2.3.2 Specification

The reference implementation is

```
def add_vec_spec(x: Float32[32,], y: Float32[32,]) -> Float32[32, 32]:
    return x[None, :] + y[:, None]
```

Here broadcasting is used to construct a 32×32 matrix.

2.3.3 Kernel Skeleton

```
@triton.jit
def add_vec_kernel(x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr, B1:
   tl.constexpr):

    off_x = tl.arange(0, B0)
    off_y = tl.arange(0, B1)

    # TODO: load vectors and compute outer addition
```

2.3.4 Solution

Solution

```
@triton.jit
def add_vec_kernel(x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr,
   B1: tl.constexpr):

    off_x = tl.arange(0, B0)
    off_y = tl.arange(0, B1)

    x = tl.load(x_ptr + off_x)
    y = tl.load(y_ptr + off_y)

    z = x[None, :] + y[:, None]

    off_z = off_y[:, None] * B0 + off_x[None, :]

    tl.store(z_ptr + off_z, z)
```

2.3.5 Explanation

First, the kernel loads the two input vectors:

```
x = tl.load(x_ptr + off_x)
y = tl.load(y_ptr + off_y)
```

The outer addition is computed using broadcasting:

$$z = x[None, :] + y[:, None].$$

Here

$$x[None, :] \to (1, B_0)$$

and

$$y[:, None] \to (B_1, 1).$$

Broadcasting produces a matrix of shape

$$(B_1, B_0).$$

Finally, the 2D result is written back to memory. Since Triton memory is flattened, the offsets are computed as

$$\text{off\_z} = \text{off\_y} \times B_0 + \text{off\_x}.$$

This converts the two-dimensional indices into flattened memory addresses.

2.3.6 Take Away

- Broadcasting allows Triton to compute outer operations efficiently.
- `[:, None]` and `[None, :]` introduce additional dimensions.
- Two-dimensional indices must be converted into flattened memory offsets before writing to memory.

2.4 Puzzle 4: Outer Vector Add with Blocks

2.4.1 Goal

In this puzzle we compute the outer addition of two vectors, but now both vectors may be larger than the block sizes. Therefore multiple program instances are required along two axes. Mathematically, the operation is

$$z_{j,i} = x_i + y_j, \quad i = 0, \ldots, N_0 - 1, \quad j = 0, \ldots, N_1 - 1.$$

The output tensor has shape

$$N_1 \times N_0.$$

Each program instance processes a block of size $B_1 \times B_0$.

2.4.2 Specification

The reference implementation is

```
def add_vec_block_spec(x: Float32[100,], y: Float32[90,]) -> Float32
    [90, 100]:
    return x[None, :] + y[:, None]
```

2.4.3 Kernel Skeleton

```
@triton.jit
def add_vec_block_kernel(
    x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr, B1: tl.constexpr
):
    block_id_x = tl.program_id(0)
    block_id_y = tl.program_id(1)

    # TODO: compute block offsets and perform outer addition
```

2.4.4 Solution

Solution

```
@triton.jit
def add_vec_block_kernel(
    x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr, B1: tl.
        constexpr
):

    block_id_x = tl.program_id(0)
    block_id_y = tl.program_id(1)

    off_x = block_id_x * B0 + tl.arange(0, B0)
    off_y = block_id_y * B1 + tl.arange(0, B1)

    x = tl.load(x_ptr + off_x, mask=off_x < N0)
    y = tl.load(y_ptr + off_y, mask=off_y < N1)

    z = x[None, :] + y[:, None]

    off_z = off_y[:, None] * N0 + off_x[None, :]

    mask = (off_y[:, None] < N1) & (off_x[None, :] < N0)

    tl.store(z_ptr + off_z, z, mask=mask)
```

2.4.5 Explanation

The kernel uses two program axes:

```
block_id_x = tl.program_id(0)
block_id_y = tl.program_id(1)
```

These identify which block of the output matrix the current program processes.
The offsets for the current block are computed as

$$\text{off\_x} = \text{block\_id\_x} \times B_0 + [0, \ldots, B_0 - 1]$$

$$\text{off\_y} = \text{block\_id\_y} \times B_1 + [0, \ldots, B_1 - 1].$$

The vectors x and y are loaded from memory using these offsets.
The outer addition

$$z = x[None, :] + y[:, None]$$

produces a block of shape

$$B_1 \times B_0.$$

Since Triton memory is flattened, the offsets for storing results are

$$\text{off\_z} = \text{off\_y} \times N_0 + \text{off\_x}.$$

Finally, a mask is applied to avoid writing outside the valid region when N_0 or N_1 are not multiples of the block sizes.

2.5 Puzzle 5: Fused Outer Multiplication

2.5.1 Goal

In this puzzle we compute the outer multiplication of two vectors and then apply the ReLU activation function.

Mathematically, the operation is

$$z_{j,i} = \text{relu}(x_i \times y_j), \quad i = 0, \ldots, N_0 - 1, \quad j = 0, \ldots, N_1 - 1.$$

The output tensor has shape

$$N_1 \times N_0.$$

Each program instance processes a block of size $B_1 \times B_0$.

2.5.2 Specification

The reference implementation is

```
def mul_relu_block_spec(x: Float32[100,], y: Float32[90,]) -> Float32
    [90, 100]:
    return torch.relu(x[None, :] * y[:, None])
```

2.5.3 Kernel Skeleton

```
@triton.jit
def mul_relu_block_kernel(
    x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr, B1: tl.constexpr
):
    block_id_x = tl.program_id(0)
    block_id_y = tl.program_id(1)

    # TODO: compute block offsets, multiply vectors, and apply relu
```

2.5.4 Solution

Solution

```
@triton.jit
def mul_relu_block_kernel(
    x_ptr, y_ptr, z_ptr, N0, N1, B0: tl.constexpr, B1: tl.
        constexpr
):

    block_id_x = tl.program_id(0)
    block_id_y = tl.program_id(1)

    off_x = block_id_x * B0 + tl.arange(0, B0)
    off_y = block_id_y * B1 + tl.arange(0, B1)

    x = tl.load(x_ptr + off_x, mask=off_x < N0)
    y = tl.load(y_ptr + off_y, mask=off_y < N1)

    z = x[None, :] * y[:, None]

    z = tl.maximum(z, 0)

    off_z = off_y[:, None] * N0 + off_x[None, :]

    mask = (off_y[:, None] < N1) & (off_x[None, :] < N0)

    tl.store(z_ptr + off_z, z, mask=mask)
```

2.5.5 Explanation

The kernel first determines which block of the output matrix the current program instance processes:

```
block_id_x = tl.program_id(0)
block_id_y = tl.program_id(1)
```

The offsets of the elements processed by this program are

$$\text{off\_x} = \text{block\_id\_x} \times B_0 + [0, \ldots, B_0 - 1]$$

$$\text{off\_y} = \text{block\_id\_y} \times B_1 + [0, \ldots, B_1 - 1].$$

The vectors are loaded from memory using these offsets.
The outer multiplication

$$z = x[\text{None}, :] \times y[:, \text{None}]$$

produces a block of size

$$B_1 \times B_0.$$

The ReLU activation is applied element-wise:

$$z = \max(z, 0).$$

Finally, the results are written back to memory using flattened offsets.

2.6 Puzzle 6: Fused Outer Multiplication — Backward

2.6.1 Goal

In this puzzle we implement the backward pass of a fused operation

$$z_{j,i} = \text{relu}(x_{j,i} \times y_j), \quad i = 0, \ldots, N_0 - 1, \quad j = 0, \ldots, N_1 - 1.$$

Here x is a matrix of shape $N_1 \times N_0$, y is a vector of shape N_1, and dz is the upstream gradient of shape $N_1 \times N_0$.

The goal is to compute the gradient with respect to x:

$$dx_{j,i} = \frac{\partial z_{j,i}}{\partial x_{j,i}} \times dz_{j,i}.$$

Each program instance processes a block of shape $B_1 \times B_0$.

2.6.2 Specification

The reference implementation is

```
def mul_relu_block_back_spec(
    x: Float32[90, 100], y: Float32[90,], dz: Float32[90, 100]
) -> Float32[90, 100]:
    x = x.clone().detach().requires_grad_(True)
    y = y.clone().detach().requires_grad_(True)
    z = torch.relu(x * y[:, None])
    z.backward(dz)
    dx = x.grad
    return dx
```

2.6.3 Kernel Skeleton

```
@triton.jit
def mul_relu_block_back_kernel(
    x_ptr, y_ptr, dz_ptr, dx_ptr, N0, N1, B0: tl.constexpr, B1: tl.
        constexpr
):
    block_id_i = tl.program_id(0)
    block_id_j = tl.program_id(1)

    # TODO: load x, y, dz and compute dx
```

2.6.4 Solution

Solution

```
@triton.jit
def mul_relu_block_back_kernel(
    x_ptr, y_ptr, dz_ptr, dx_ptr, N0, N1, B0: tl.constexpr, B1: tl
        .constexpr
):

    block_id_i = tl.program_id(0)
    block_id_j = tl.program_id(1)

    off_i = block_id_i * B0 + tl.arange(0, B0)
    off_j = block_id_j * B1 + tl.arange(0, B1)

    off_x = off_j[:, None] * N0 + off_i[None, :]
    mask = (off_j[:, None] < N1) & (off_i[None, :] < N0)

    x = tl.load(x_ptr + off_x, mask=mask, other=0)
    dz = tl.load(dz_ptr + off_x, mask=mask, other=0)
    y = tl.load(y_ptr + off_j, mask=off_j < N1, other=0)

    prod = x * y[:, None]
    relu_grad = prod > 0

    dx = relu_grad * y[:, None] * dz

    tl.store(dx_ptr + off_x, dx, mask=mask)
```

2.6.5 Explanation

The current program instance is identified by

```
block_id_i = tl.program_id(0)
block_id_j = tl.program_id(1)
```

These determine which block of the matrix is processed.
The offsets for the current block are

$$\text{off\_i} = \text{block\_id\_i} \times B_0 + [0, \ldots, B_0 - 1]$$

$$\text{off\_j} = \text{block\_id\_j} \times B_1 + [0, \ldots, B_1 - 1].$$

Since x and dz are matrices of shape $N_1 \times N_0$, their flattened offsets are

$$\text{off\_x} = \text{off\_j}[:, None] \times N_0 + \text{off\_i}[None, :].$$

A mask is applied to avoid invalid memory accesses near the block boundaries.
The forward function is

$$z_{j,i} = \text{relu}(x_{j,i} y_j).$$

To compute the backward pass, we apply the chain rule:

$$dx_{j,i} = \frac{\partial \, \text{relu}(x_{j,i} y_j)}{\partial x_{j,i}} \cdot dz_{j,i}.$$

Since the derivative of ReLU is

$$\text{relu}'(u) = \begin{cases} 1, & u > 0 \\ 0, & u \leq 0 \end{cases},$$

we have

$$\frac{\partial \,\text{relu}(x_{j,i} y_j)}{\partial x_{j,i}} = \mathbf{1}(x_{j,i} y_j > 0) \cdot y_j.$$

Therefore

$$dx_{j,i} = \mathbf{1}(x_{j,i} y_j > 0) \cdot y_j \cdot dz_{j,i}.$$

In the kernel, this is implemented by first computing

```
prod = x * y[:, None]
relu_grad = prod > 0
```

and then

```
dx = relu_grad * y[:, None] * dz
```

Finally, the result is written back to memory using `tl.store`.