



# Triton:-

## ① → Kernel

Convert python func to GPU kernel  
[Compile GPU kernel]

Similar to C++ template  
<int Block\_Size> using for loop  
Unrolling

@triton.jit

```
def add_kernel(x_ptr, y_ptr, output_ptr, n_elements, BLOCK_SIZE:  
tl.constexpr):  
    pid = tl.program_id(axis=0) # Kaun sa program/block chal raha hai  
    block_start = pid * BLOCK_SIZE  
    offsets = block_start + tl.arange(0, BLOCK_SIZE)  
    mask = offsets < n_elements  
  
    x = tl.load(x_ptr + offsets, mask=mask)  
    y = tl.load(y_ptr + offsets, mask=mask)  
    output = x + y  
    tl.store(output_ptr + offsets, output, mask=mask)
```

① → @triton.jit → decorator python ko GPU mein compile karne.  
→ when we call this function it will execute on GPU.  
→ triton compile this on LLVM IR and then GPU assembly

## ② → function parameters:-

- (a) x\_ptr, y\_ptr, output\_ptr:- pointers [memory address].
- (b) n\_elements → Total how much elements have to process.
- (c) Block\_Size: H · constexpr: Come-time constant → no change on run time.

③ Pid :-  $pid = \text{tl} \cdot \text{program\_id}(\text{or} 120)$

Visualize Pid :-

→ GPU works in parallel.

→ Suppose we have 10,000 elements.

→ each worker has to process 256 elements [Block size = 256].

→ So need 40 programs (workers).

So what's the use of pid here?

Pid will tell which worker are you.

Example :-

Worker 0  $\rightarrow pid = 0 \rightarrow$  processes elements [0 to 255]

Worker 1  $\rightarrow pid = 1 \rightarrow$  processes elements [256 to 511]

⋮

Worker 39  $\rightarrow pid = 39 \rightarrow$  processes elements [99914 to 99999]

What is meant by axis = 0?

In 2D grid we have to define which axis to follow - (x or y)

axis<sub>2D</sub>

axis<sub>2D</sub>

↓

↓

④ Block Start  $\rightarrow$  Starting point! :-  $block\_start = pid * \text{Blocksize}$

Pid 0 :  $block\_start = 0 * 256 = 0$

Pid 1 :  $block\_start = 1 * 256 = 256$

Pid 2 :  $block\_start = 2 * 256 = 512$

Pid 3 :  $block\_start = 3 * 256 = 768$

→ each worker calculate its own starting point.

(5)  $\text{offsets} = \text{block\_start} + \text{fl.orange}(0, \text{BlockSize})$

$\text{fl.orange}(0, \text{BlockSize})$  what is that?

→ vectorized format in GPU.

Example with  $\text{block\_size} = 4$

PID 0:-

$\text{block\_start} = 0$   
 $\text{fl.orange}(0, 4) = [0, 1, 2, 3]$

$\text{offsets} = 0 + [0, 1, 2, 3] = [0, 1, 2, 3]$

PID 1:-  $\text{block\_start} = 4$

$\text{fl.orange}(0, 4) = [0, 1, 2, 3]$

$\text{offsets} = 4 + [0, 1, 2, 3] = [4, 5, 6, 7]$

PID 2:-  $\text{block\_start} = 8$

$\text{offsets} = [8, 9, 10, 11]$

offsets tells to workers which indices we have to process.

(6) Mask-Boundary protection - mask =  $\text{offsets} < n\_elements$

Example:- If 10 elems are there and  $\text{block\_size} = 4$

→ PID 0 → offsets =  $[0, 1, 2, 3]$  ✓ valid

PID 1 → offsets =  $[4, 5, 6, 7]$  ✓ valid

PID 2 → " =  $[8, 9, 10, 11]$  ✗ 10, 11 not valid

Mask:-

$\text{offsets} = [8, 9, 10, 11]$

$n\_elements = 10$

$\text{mask} = [8 < 10, 9 < 10, 10 < 10, 11 < 10]$   
[T, T, F, F]

So Masks prevent from out of bound num.

## ⑦ Data Load - (GPU - CPU)

$x = \text{fl\_load}(x\_phr + offset, mark=mark)$

$y = " (y\_phr + offset, mark=mark)$

what is happening?

Pointer Arithmetic :-  $x\_phr + offset$

$x\_phr$  = base add (eg 0x100), offset = [0, 1, 2, 3]

## ⑧ Memory addresses:-

$x\_phr + 0 \rightarrow 0x100$

$x\_phr + 1 \rightarrow 0x1004$  (assuming 4 bytes per elem)

$x\_phr + 2 \rightarrow 0x1008$

$x\_phr + 3 \rightarrow 0x100C$

## ⑨ Masked Load :- $\text{Mark} = [\text{True}, \text{True}, \text{False}, \text{False}]$

So this tells load only True flag value!

$$x = [x[0], x[1], 0, 0]$$

## ⑩ Vectorized :- Load elems in one go. [parallel].

Result:  $x = [\text{value\_at\_0}, \text{value\_at\_1}, \text{value\_at\_2}, \text{value\_at\_3}]$

$y = [\text{value\_at\_0}, \text{value\_at\_2}, \text{value\_at\_2}, \text{value\_at\_3}]$

## ⑪ Computation - Actual work

$$\text{output} = x + y$$

Vectorized addition!  $x = [1.5, 2.3, 4.1, 5.7]$

$y = [0.5, 1.2, 3.5, 2.1]$

$$\text{output} = [2.0, 3.5, 7.4, 7.8]$$

} element wise addition.

all happens on GPU on Rego.

⑨ Store Result - After Compute by <sup>(Device)</sup> GPU transfer to <sup>(Host)</sup> CPU.

`tl::store(output_ptr + offsets, output, mask == mark)`

What's going on?

$\rightarrow$  `output_ptr + offsets`  $\rightarrow$  calculate on which add it is going to store.

$\rightarrow$  `output`  $\rightarrow$  Data to store.

$\rightarrow$  `mask == mark`  $\rightarrow$  only valid indices.

Visual Summary [dry run]:-

Array:  $[a_0, a_1, a_3 \dots a_9]$ , Block Size = 4

GPU launches 3 workers:-

Worker 0 (pid = 0):-

$$offsets = [0, 1, 2, 3]$$

$$processes: a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3$$

Worker 1 (pid = 1):-

$$offsets = [4, 5, 6, 7]$$

$$processes: a_4+b_4, a_5+b_5, a_6+b_6, a_7+b_7$$

Worker 2 (pid = 2):-

$$offsets = [8, 9, 10, 11]$$

$$mask = [T, T, F, F]$$

$$processes = a_8+b_8, a_9+b_9, (\text{skip}), (\text{skip})$$

All workers run simultaneously.

Load  $\rightarrow$  Add  $\rightarrow$  Store

②

## Helper functions.

```
def add(x, y):  
    output = torch.empty_like(x)  
    n_elements = output.numel()  
    grid = lambda meta:  
        (triton.cdiv(n_elements,  
                     meta['BLOCK_SIZE']), )  
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)  
    return output
```

Allocate the output tensor

This func [wrapper function] → This will run on CPU and launch the GPU kernel.

ⓐ output = torch.empty\_like(x) why?

will create empty tensor like X(data).

→ Same shape(size), Same dtype (f32, b4 etc)

→ Same device (GPU or CPU)

memory allocate here, but garbage values have initially (random data)

→ output = tensor([?, ?, ?])

But why empty\_like → fast because doesn't initialize values.  
[value filled by kernel].

ⓑ n\_elements = output.numel()

→ • numel() ?

num of elements  
how many ele are there in tensor.

(c) Calculate grid size:-

grid = lambda meta: (m1on.cdiv(n\_elements, meta['Block Size']),)

What is Meta here?

It is dictionary, which contains meta parameters.

Meta['Block Size'] → Block\_Size value = 1024

What is torch.cdiv()? Ceiling Division

n\_elements = 1000

BLOCK\_SIZE = 256

$$\begin{aligned} \text{grid\_size} &= \text{cdiv}(1000, 256) \\ &= \text{ceil}(1000/256) \\ &= \text{ceil}(3.906) \\ &= 4 \text{ blocks} \end{aligned}$$

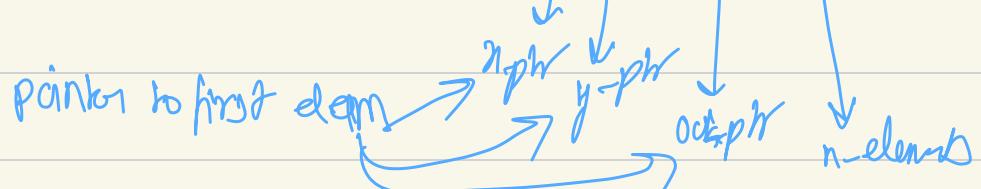
Block 0: elements [0-255] (256 elements)

Block 1: elements [256-511] (256 elements)

Block 2: elements [512-767] (256 elements)

Block 3: elements [768-999] (232 elements) ← Mask handle karega!

(d) Kernel Launch . add\_kernel([grid]) (x,y, output, n\_elements, BlockSize)



# full Execution flow! -

```
x = torch.rand(98432, device='cuda')  
y = torch.rand(98432, device='cuda')  
result = add(x,y)
```

## Step-By-Step Guide

Step 1: Allocate output

```
output = torch.empty(98432, device='cuda')
```

Step 2: Count elements

```
n_elements = 98432
```

Step 3: Calculate grid size

```
BLOCK_SIZE = 1024
```

```
grid_size = cdiv(98432, 1024) = 97
```

Step 4: Launch kernel

GPU launches 97 blocks:

Block 0 (pid=0): processes elements [0 - 1023]

Block 1 (pid=1): processes elements [1024 - 2047]

Block 2 (pid=2): processes elements [2048 - 3071]

...

Block 95 (pid=95): processes elements [97280 - 98303]

Block 96 (pid=96): processes elements [98304 - 99327]

but mask prevents [98432 - 99327] 

Step 5: GPU executes all 97 blocks

SIMULTANEOUSLY! 

Step 6: Return result

```
return output # Now filled with x + y
```

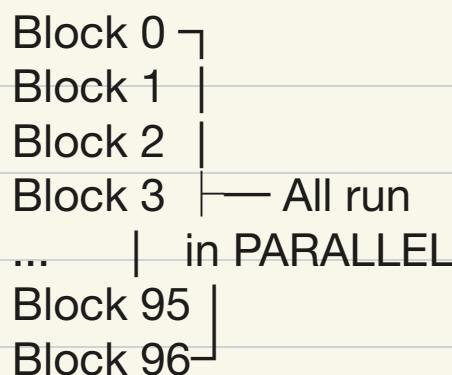
# Visual Representation :-

CPU (Host)

GPU (Device)

add(x, y) called

- Allocate output
- Calculate grid = 97
- Launch kernel ——————> 97 Blocks spawn!



Each block:

- Gets its pid
- Calculates offsets
- Loads data
- Adds
- Stores result

Return output ←

All blocks done!

Note:-

After launching kernel CPU doesn't wait [unless sync called]

inbuilt **Asynchronous** behaviour.