



Triton:- ① → Kernel

@triton.jit

Convert python func to GPU kernel
[Compile GPU kernel]

Similar to C++ template
<int Block Size> using loop
Unrolling

```
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 krna.
→ 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 addresses].

(b) n_elements ⇒ Total how much elements have to process.

(c) Block_Size: tl.constexpr: Come-time constant → no change on run time.

③ Pid :- $\text{pid} = \text{rank} \cdot \text{program_id}(\text{axis} = 20)$

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.

Examples →

worker 0	→ pid = 0	→ processes elements [0 to 255]
worker 1	→ pid = 1	→ processes elements [256 to 511]
⋮		
worker 39	→ pid = 39	→ processes elements [99904 to 99999]

what is mean by $\text{axis} = 20$?

in 2D grid we have to define which axis to follow. — (x or y)

$\text{axis} = 20$

↓

$\text{axis} = 1$

↓

④ Block Start → Starting point! — $\text{block_start} = \text{pid} * \text{Block_Size}$

Pid 0 : $\text{block_start} = 0 * 256 = 0$


Pid 1 : $\text{block_start} = 1 * 256 = 256$

Pid 2 : $\text{block_start} = 2 * 256 = 512$

Pid 3 : $\text{block_start} = 3 * 256 = 768$

→ each worker calculate its own starting point.

⑤ $offsets = block_start + fl.offset(0, BlockSize)$

$fl.offset(0, BlockSize)$ what is that?

 vectorized format in GPU.

Example with $block_size = 4$

PID 0:-

$block_start = 0$

$fl.offset(0, 4) = [0, 1, 2, 3]$

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

PID 1:- $block_start = 4$

$fl.offset(0, 4) = [0, 1, 2, 3]$

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

PID 2:- $block_start = 8$

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

$offsets$ tells to workers which indices we have to process.

⑥ Mask-Boundary protection: $mask = offsets < n_elements$

Example:- if 10 elements are there and $block_size = 4$

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

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

PID 2 $\rightarrow " = [8, 9, 10, 11]$ ✗ 10, 11 Not valid


Mask:-

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

$n_elements = 10$

$mask = [8 < 10, 9 < 10, 10 < 10, 11 < 10]$

$[T, T, F, F]$

 So masks prevent from out of bound mem.

⑦ Data Load - (GPU - CPU)

$x = \text{fl-load}(x_ptr + off_xkb, \text{mask} = \text{mask})$

$y = \text{" (} y_ptr + off_ykb, \text{mask} = \text{mask)}$

what is happening?

Pointer Arithmetic: $x_ptr + off_xkb$

$x_ptr = \text{base add (eg } 0x100), off_xkb = [0, 1, 4, 3]$

⑧ Memory addresses:-

$x_ptr + 0 \rightarrow 0x100$

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

$x_ptr + 2 \rightarrow 0x1008$

$x_ptr + 3 \rightarrow 0x100C$

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

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:- $x = [1.5, 2.3, 4.1, 5.7]$
 $y = [0.5, 1.2, 3.3, 2.1]$

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

} element wise addition.

all happens on GPU on Hego.

⑨ Store Results - After Compute by GPU ^(Device) transfer to CPU ^(Host).

$tl_store(output_ptr + offsets, output, mask = mask)$

what's going on?

- $output_ptr + offsets$ → Calculate on which add it is going to store.
- $output$ → Data to store.
- $mask = mask$ → only valid indices.

Visual Summary [dry run]:-

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

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

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

→ 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 → Add → Store

② Helper functions.

```
def add(x, y):  
    output = torch.empty_like(x)  Allocate the output tensor  
    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
```

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

① `output = torch.empty_like(x)` *why?*

will create empty tensor like X (data).
→ Same shape (size), same dtype (float 32, 64 etc)
→ Same device (GPU or CPU)

memory allocated here, but garbage values here initially (random data)

→ `output = tensor([?, ?, ?])`

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

② `n_elements = output.numel()`

→ `• numel()` ?

num of elements
how many ele are there in tensor.

③ Calculate grid Size:-

$\text{grid} = \text{torch.cuda.device_count}() \times \text{torch.div}(\text{n_elements}, \text{meta}[\text{'Block_Size'}])$

What is meta here?

it is dictionary, which contains meta parameters.

$\text{meta}[\text{'Block_Size'}] \rightarrow \text{Block_Size value} = 256$

What is $\text{torch.div}()$? Ceiling Division

$\text{n_elements} = 1000$

$\text{BLOCK_SIZE} = 256$

$\text{grid_size} = \text{cd}(\text{1000}, \text{256})$

$= \text{ceil}(1000/256)$

$= \text{ceil}(3.906)$

$= 4 \text{ blocks}$

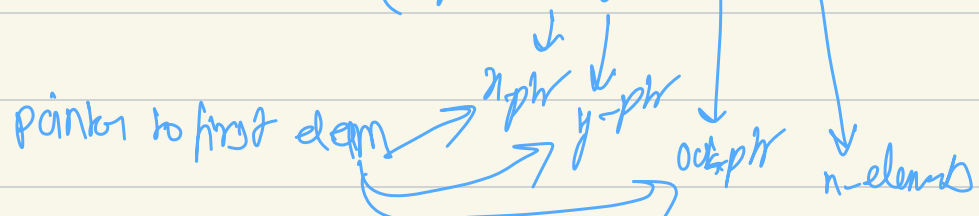
Block 0: elements [0-255] (256 elements)

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

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

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

④ Kernel launch . $\text{add_kernel}(\text{[grid]})(\text{n_y}, \text{output}, \text{n_elements}, \text{Block_size})$



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

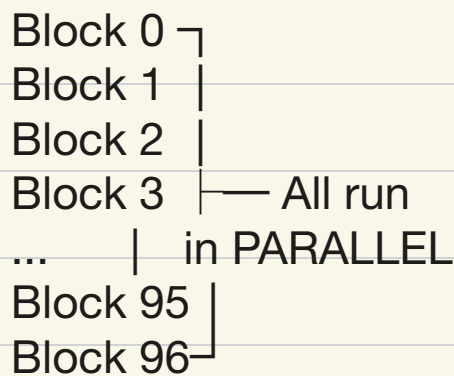
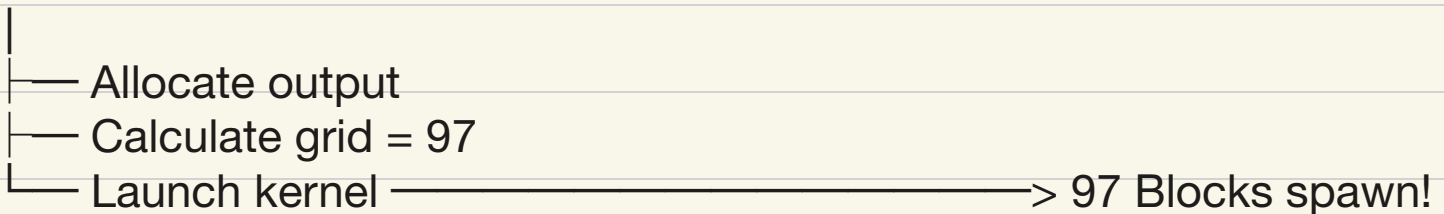
return output # Now filled with $x + y$

Visual Representation:-

CPU (Host)

GPU (Device)

add(x, y) called



- ↓
- 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.