Triton:- ① → Kernel

[Similar to C++ template <int Block_Size> using for loop Unrolling]

@triton.jit ← Convert python func to GPU kernel [Compile GPU kernel]

```
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 karna.
  → when we call this function it will execute on GPU.
  → triton compile this on LLVM IR and then GPU assembly

② → function parameters:-

ⓐ x_ptr, y_ptr, output_ptr:- pointers [memory addresses].

ⓑ n_elements ⇒ Total how much elements have to process.

ⓒ Block_Size: tl.constexpr: Compile-time constant ⇒ no change on run time.

③ **Pid**:- pid = tl.program_id(axis=0)

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 whats 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 → process elements [9984 to 9999]

what is mean by axis = 0?

axis=0 ↓ axis=1 ↙

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

④ Block Start → Starting point:- block_start = pid * Block_size

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 offsets = block_start + tl.arange(0, Block_Size)

tl.arange(0, Block_size) what is that?

→ vectorized format in GPU.

example with blocksize = 4

PID 0 :-

block_start = 0

tl.arange(0,4) = [0,1,2,3]

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

PID 1 :- block_start = 4

tl.arange(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 eles are there and 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 :-

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 = tl.load(x_ptr + offsets, mask = mask)
y = " (y_ptr + offsets, mask = mask)

what is happening?
Pointer Arithematic :- x_ptr + offsets
x_ptr = base add (eg 0x1000), offsets = [0,1,2,3]

(a) Memory addresses:–

x_ptr + 0 → 0x1000
x_ptr + 1 → 0x1004 (assuming 4 bytes per elm)
x_ptr + 2 → 0x1008
x_ptr + 3 → 0x100C

(b) Masked Load :- Mask = [True, True, False, False]

So this tells load only True flag value:–

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

(c) Vectorized :- Load elems in one go. [parallel].

→ Result :- x = [value_at_0, value_at_1, value_at_2, value_at_3]
y = [value_at_0, value_at_2, value_at_2, value_at_3]

8 Computation – Actual work

output = x + y
Vectorized addition :– x = [1.5, 2.3, 4.1, 5.7]
y = [0.5, 1.2, 3.3, 2.1]

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

element wise addition.

all happens on GPU on the go.

9 Store Result – After Compute by (Device) GPU transfer to (Host) CPU.

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: [a0, a1, a3 ------ a9], Block_SIZE = 4

GPU launches 3 workers:-

↳ worker 0 (pid = 0):-

offsets = [0, 1, 2, 3]

processes: a0+b0, a1+b1, a2+b2, a3+b3

↳ worker 1 (pid = 1):-

↳ offsets = [4, 5, 6, 7]

process: a4+b4, a5+b5, a6+b6, a7+b7

↳ worker 2 (pid = 2):-

offsets = [8, 9, 10, 11]

mask = [T, T, F, F]

processes = a8+b8, a9+b9, (skip), (skip)

All workers run simultaneously.

Load → Add → Store

② Helper functions.

Allocate the output tensor

```
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
```

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 (float 32, 64 etc)

→ Same device (GPU ya CPU)

$\boxed{\text{memory allocate hai, but garbage values hain initially (random data)}}$

→ output = tensor([?, ?, ?])

But why empty_like → fast because doesnt initialize values. [value filled by kernel].

ⓑ n_elements = outputs.numel()

→ .numel() ? $\boxed{\text{num of elements — how many ele are there in tensor.}}$

(c) Calculate grid size :-

grid = lambda meta: (triton.cdiv(n_elements, meta['Block_Size']),)

What is Meta here ?

it is dictionary, which contains meta parameters.

Meta['Block size'] → $\boxed{\text{Block\_Size value = 1024}}$

What is torch.cdiv()? Ceiling Division

n_elements = 1000
BLOCK_SIZE = 256

grid_size = cdiv(1000, 256)
= ceil(1000/256)
= ceil(3.906)
= 4 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!

(d) Kernel launch . add_kernel([grid](x, y, output, n_elements, Blocksize)

x_ptr y_ptr out_ptr n_elements

pointer to first elem

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!

                            Block 0 ┐
                            Block 1 |
                            Block 2 |
                            Block 3 ├── All run
                            ...     |   in PARALLEL
                            Block 95|
                            Block 96┘

                            ↓
                            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.