

Programming Model

Execution Model

A click through example

Vector add:

c=a+b

$$c[i] = a[i] + b[i]$$

a:

[illegible]

+

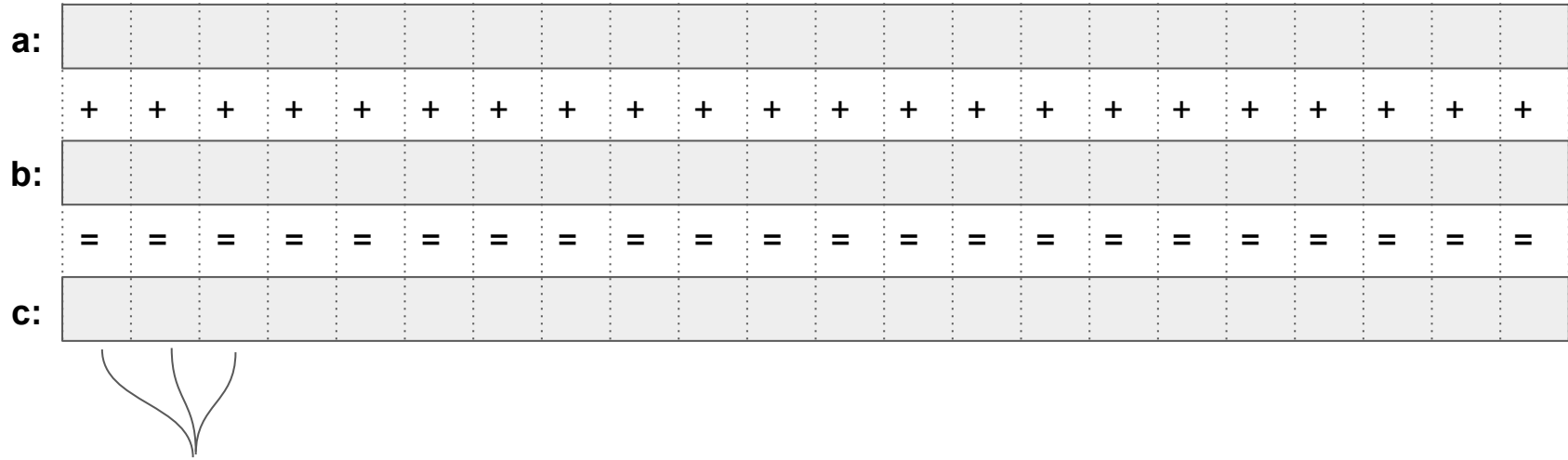
b:

=====

C:

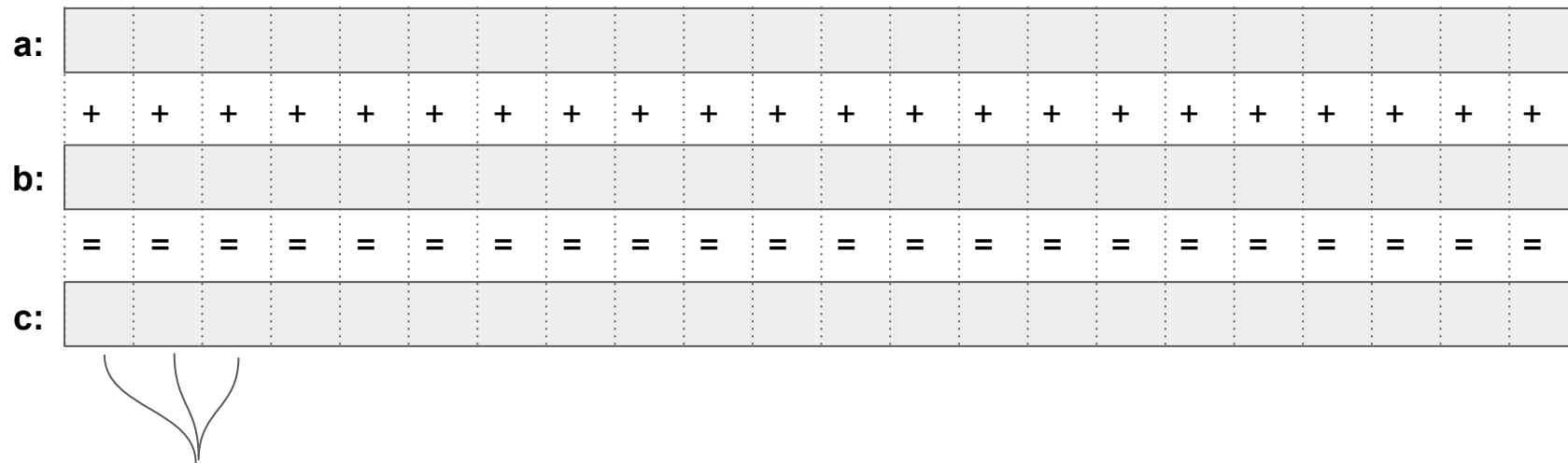
[illegible]

How to split the problem?



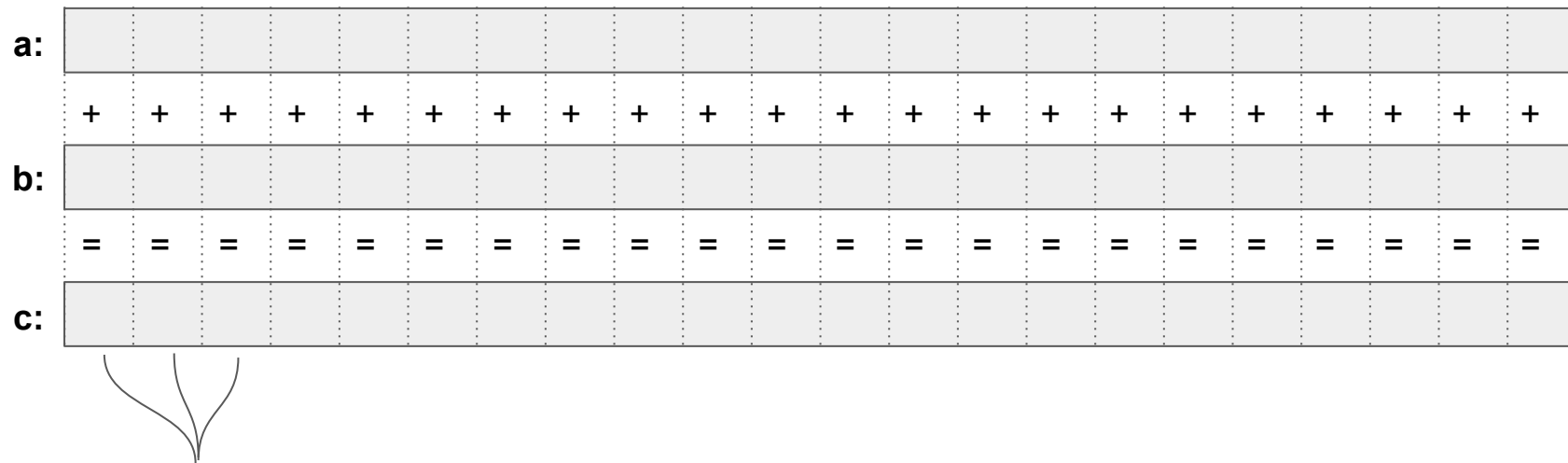
Each core calculates 1 element of c

How to split the problem?



Each core calculates 1 element of c,
But in CUDA code, we do not have the concept of “core”
“Core” is a physical abstraction that the hardware knows about

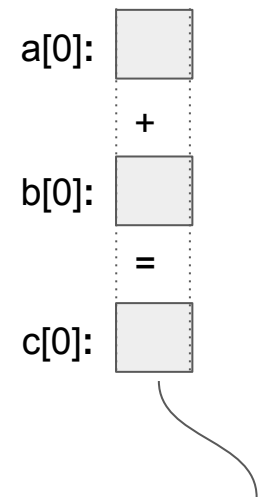
How to split the problem?



Each thread calculates 1 element of c,

“thread” is a logical abstraction that programmer knows about

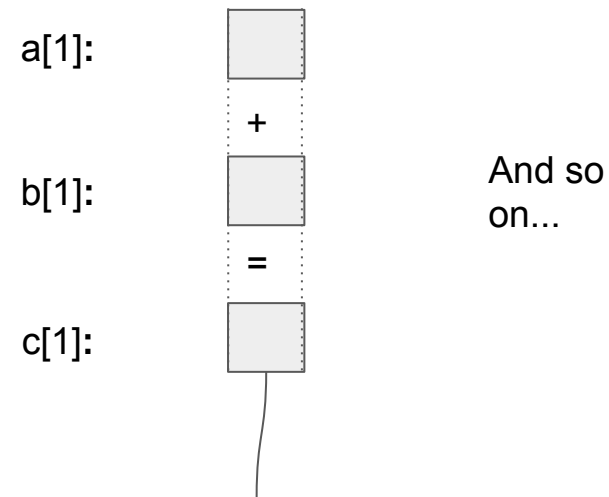
How to split the problem?



Each thread calculates 1 element of c,

“thread” is a logical abstraction that programmer knows about

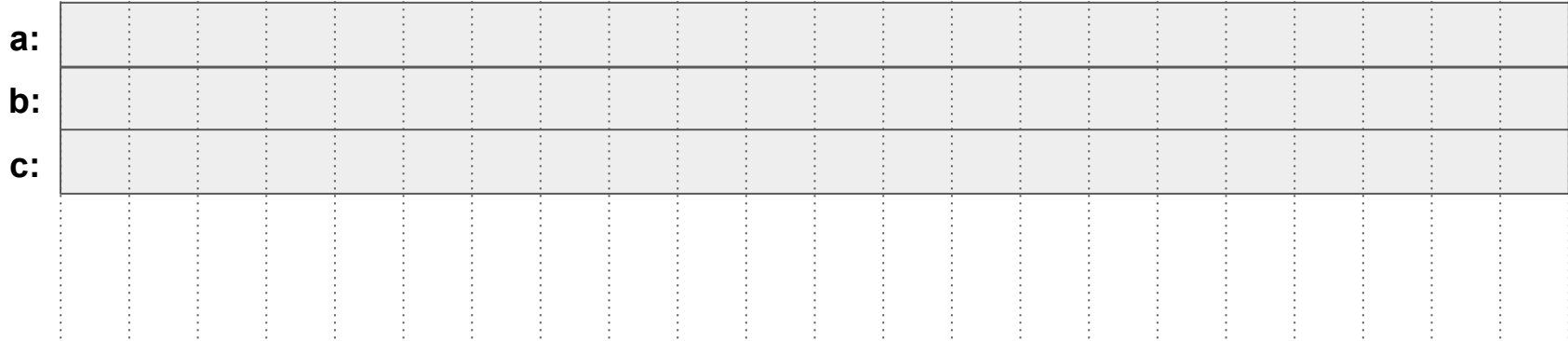
How to split the problem?



Each thread calculates 1 element of c,

“thread” is a logical abstraction that programmer knows about

Allocate the data and send to the device



The arrays (a,b,c) are allocated in the device
The arrays (a,b) are transfered to the device

Allocate the data and send to the device



The arrays (a,b,c) are allocated in the device
The arrays (a,b) are transfered to the device

Kernel call: `vAdd <<<???, ???>>>> (a, b, c, N)`

Kernel call: `vAdd <<<???, ???>>> (a, b, c, N)`



Grid dimension

`gridDim`

Number of blocks in the grid

Kernel call: vAdd <<<???, ???>>>> (a, b, c, N)



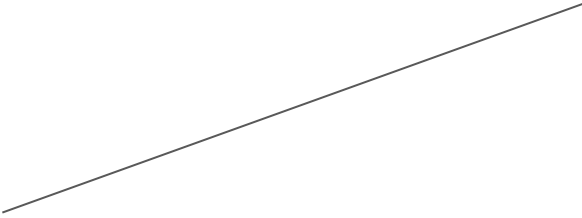
Grid dimension

gridDim

Number of blocks in the grid

Number of block in the kernel

Kernel call: `vAdd <<<???, ???>>> (a, b, c, N)`



Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

Number of blocks in the grid

Number of block in the kernel

Kernel call: `vAdd <<<???, ???>>>> (a, b, c, N)`



Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

Number of blocks in the grid

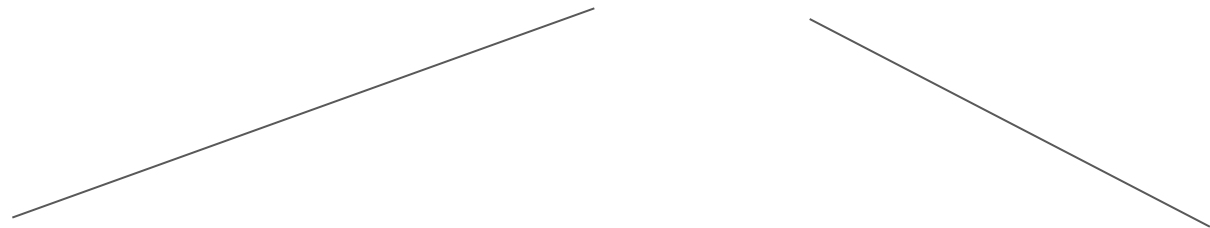
Number of block in the kernel

Block dimension

`blockDim`

Number of threads in 1 block

Kernel call: `vAdd <<<???, ???>>>> (a, b, c, N)`



Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

Number of blocks in the grid

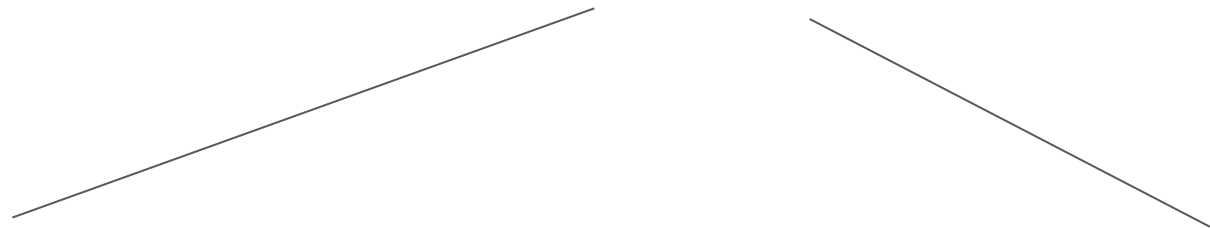
Number of block in the kernel

Block dimension

`blockDim.x, blockDim.y, blockDim.z`

Number of threads in 1 block

Kernel call: vAdd <<<4, 64>>>> (a, b, c, N)



Grid dimension

gridDim.x, gridDim.y, gridDim.z

Number of blocks in the grid

Number of block in the kernel

Block dimension

blockDim.x, blockDim.y, blockDim.z

Number of threads in 1 block

This is part of the programming model

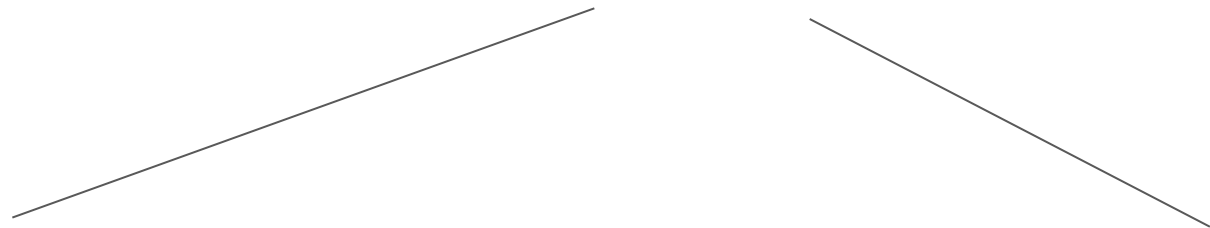
You, as a **programmer**, are aware of these concepts:

- Grid, Blocks, Threads

The **hardware** is also aware of these concepts

- So it is also part of the execution model

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`



Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

Number of blocks in the grid

Number of block in the kernel

Block dimension

`blockDim.x, blockDim.y, blockDim.z`

Number of threads in 1 block

If I have to calculate 256 element-wise vector add,
I could do it with 4 blocks and 64 threads per block
 $4 * 64 = 256$

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

Number of blocks in the grid

Number of block in the kernel

Block dimension

`blockDim.x, blockDim.y, blockDim.z`

Number of threads in 1 block

Grid: share the kernel



Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

`blockIdx`

Number of blocks in the grid

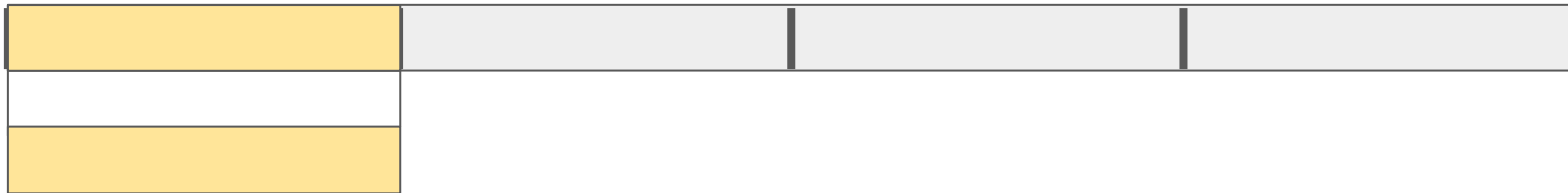
Number of block in the kernel

Block dimension

`blockDim.x, blockDim.y, blockDim.z`

Number of threads in 1 block

Grid: share the kernel



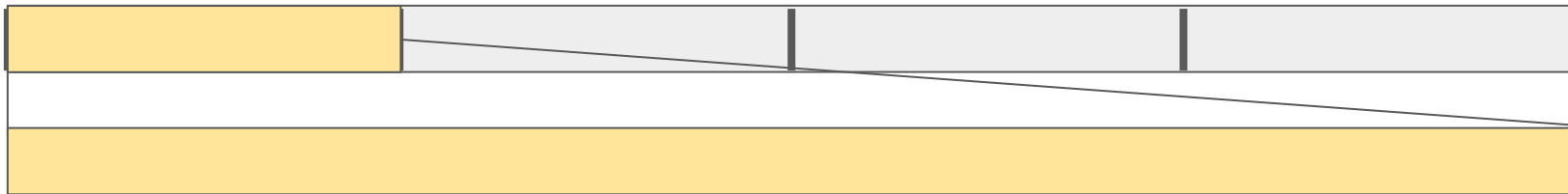
Block: share fast memory, share number of registers

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid dimension
`gridDim.x, gridDim.y, gridDim.z`
`blockIdx`
Number of blocks in the grid
Number of block in the kernel

Block dimension
`blockDim.x, blockDim.y, blockDim.z`
Number of threads in 1 block

Grid: share the kernel



Zoom on Block: share fast memory, share number of registers

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid dimension

`gridDim.x, gridDim.y, gridDim.z`

`blockIdx`

Number of blocks in the grid

Number of block in the kernel

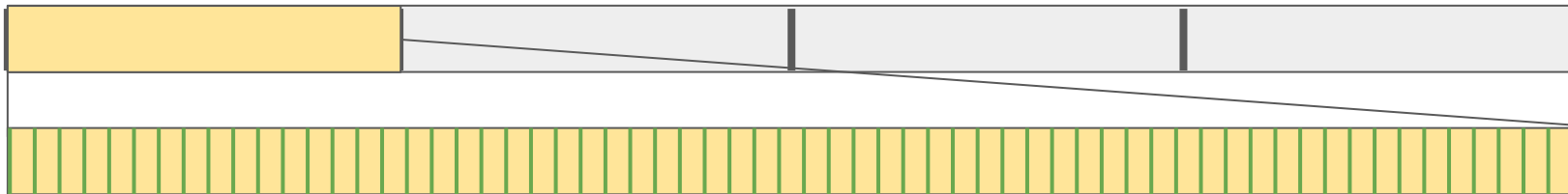
Block dimension

`blockDim.x, blockDim.y, blockDim.z`

`threadIdx`

Number of threads in 1 block

Grid: share the kernel



Zoom on Block: share fast memory, share number of registers

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

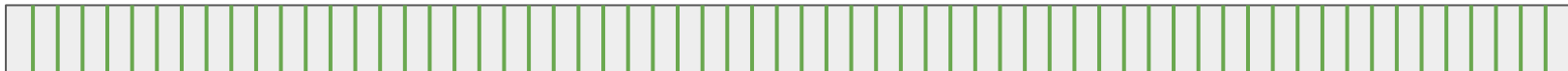
Grid



`blockIdx.x=0`



`blockIdx.x=1`



`blockIdx.x=2`

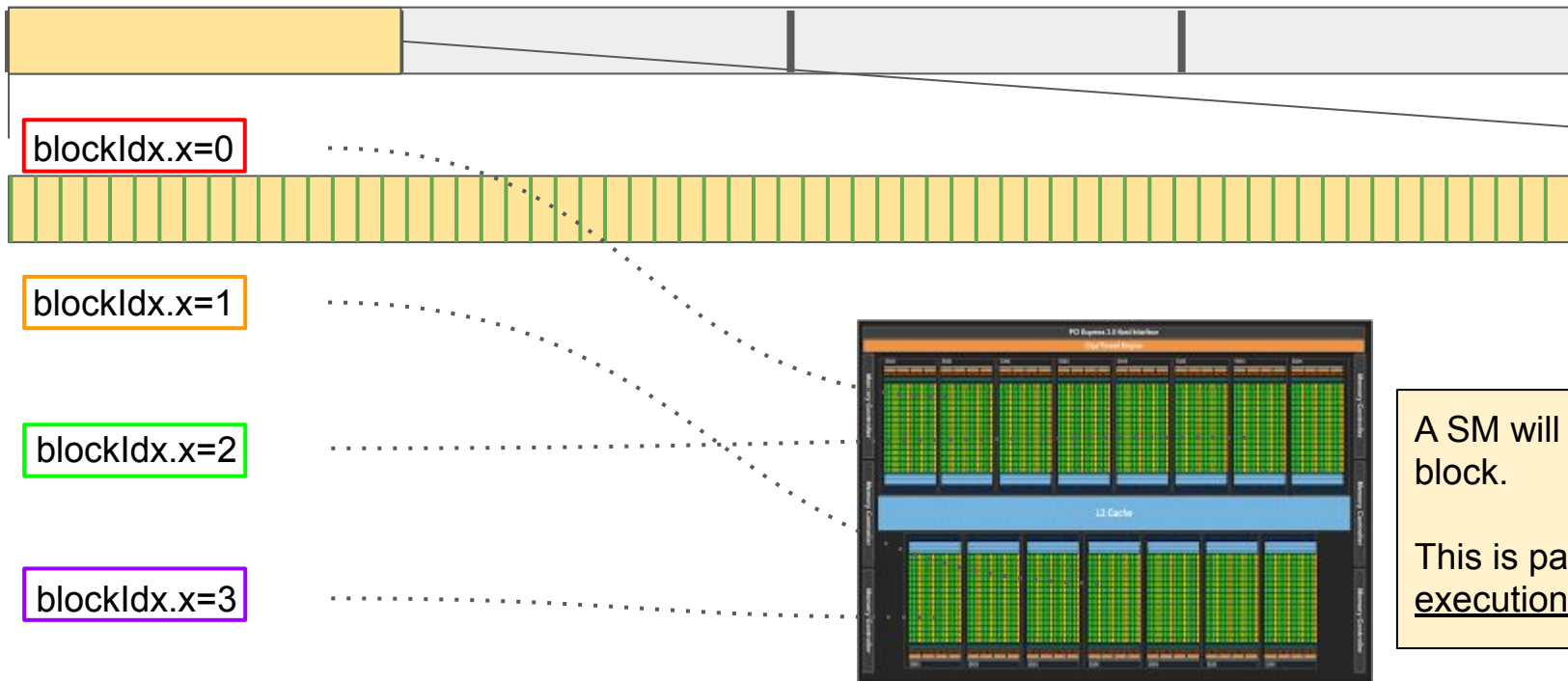


`blockIdx.x=3`



Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid

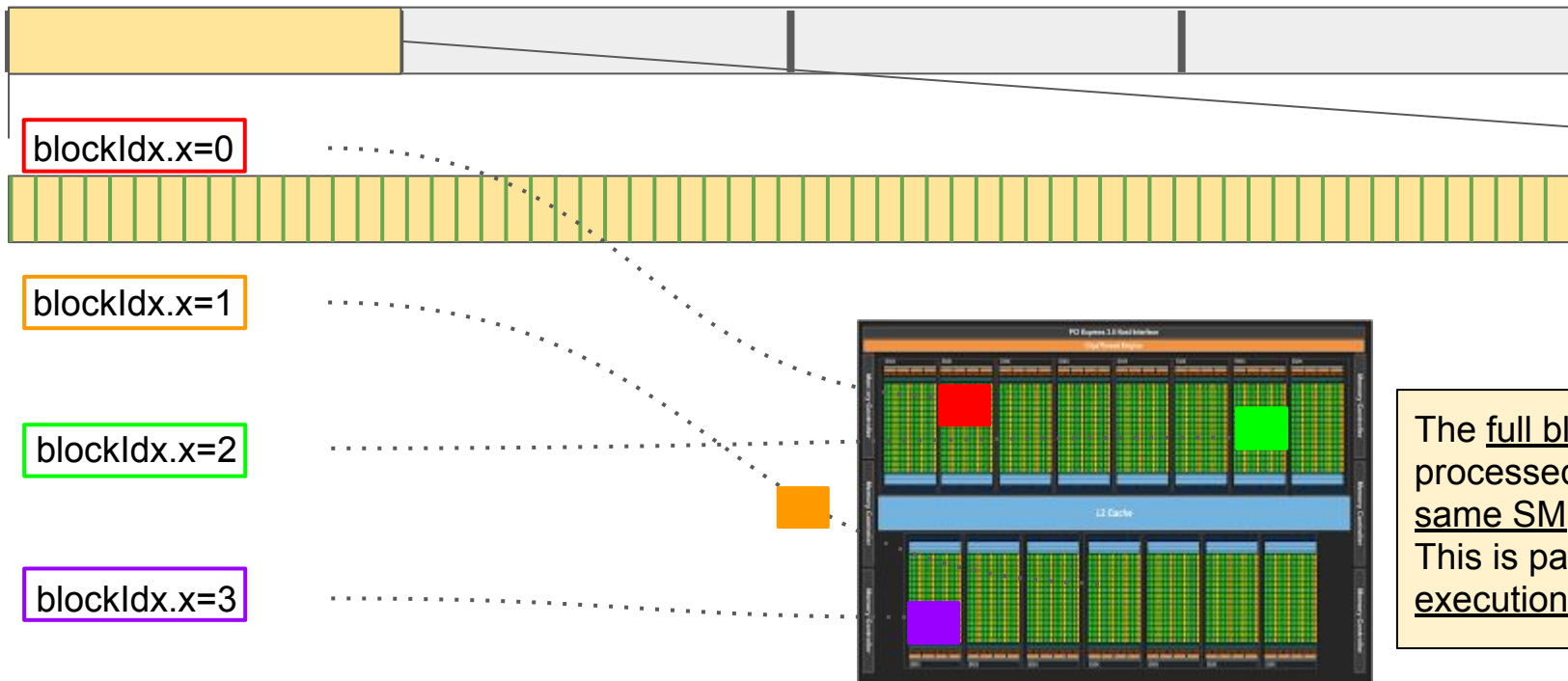


A SM will work on a block.

This is part of the execution model

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

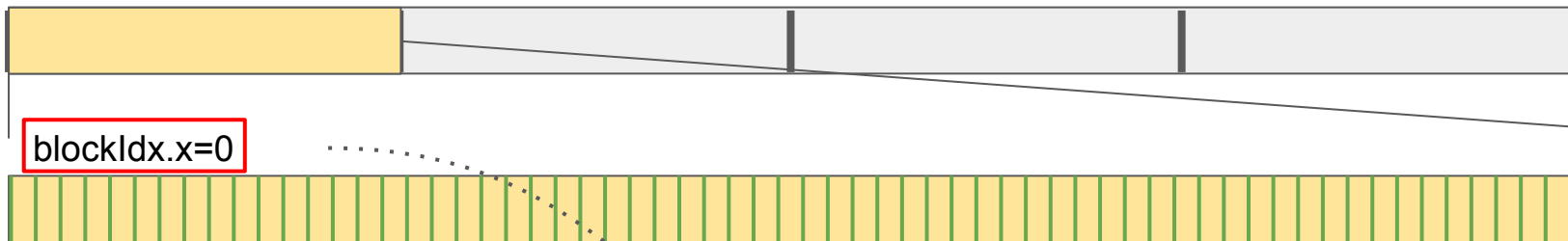
Grid



The full block will be processed on the same SM
This is part of the execution model

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid



blockIdx.x=1

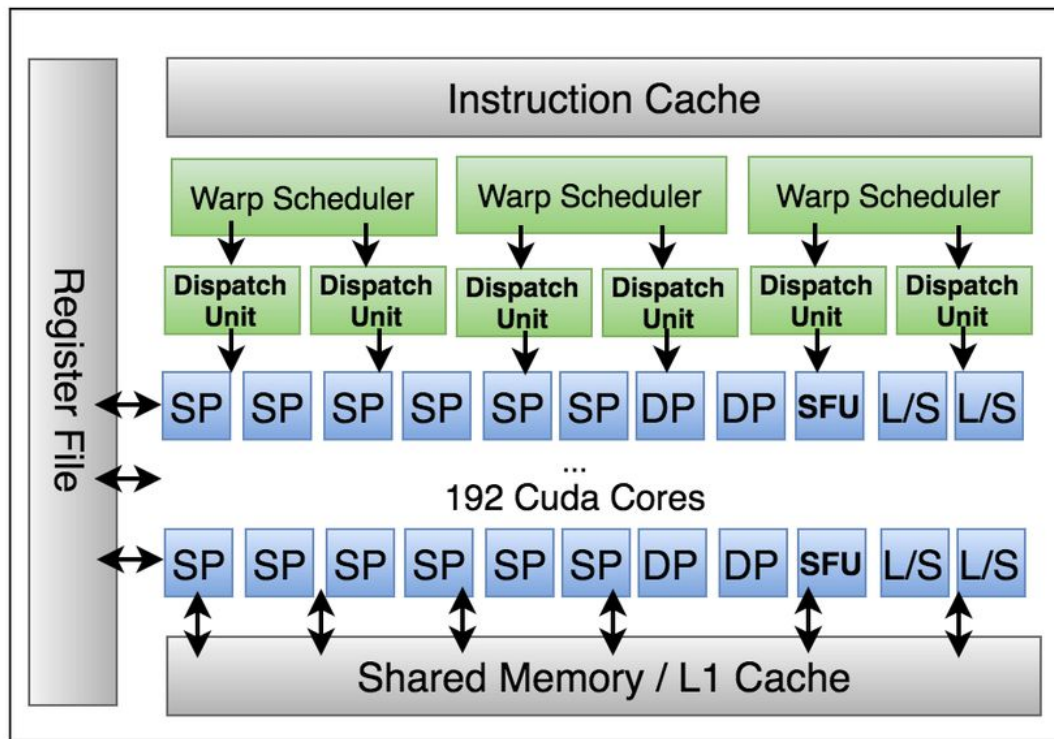
blockIdx.x=2

blockIdx.x=3



A block does NOT get split into multiple SMs. To use multiple SMs you must have multiple blocks

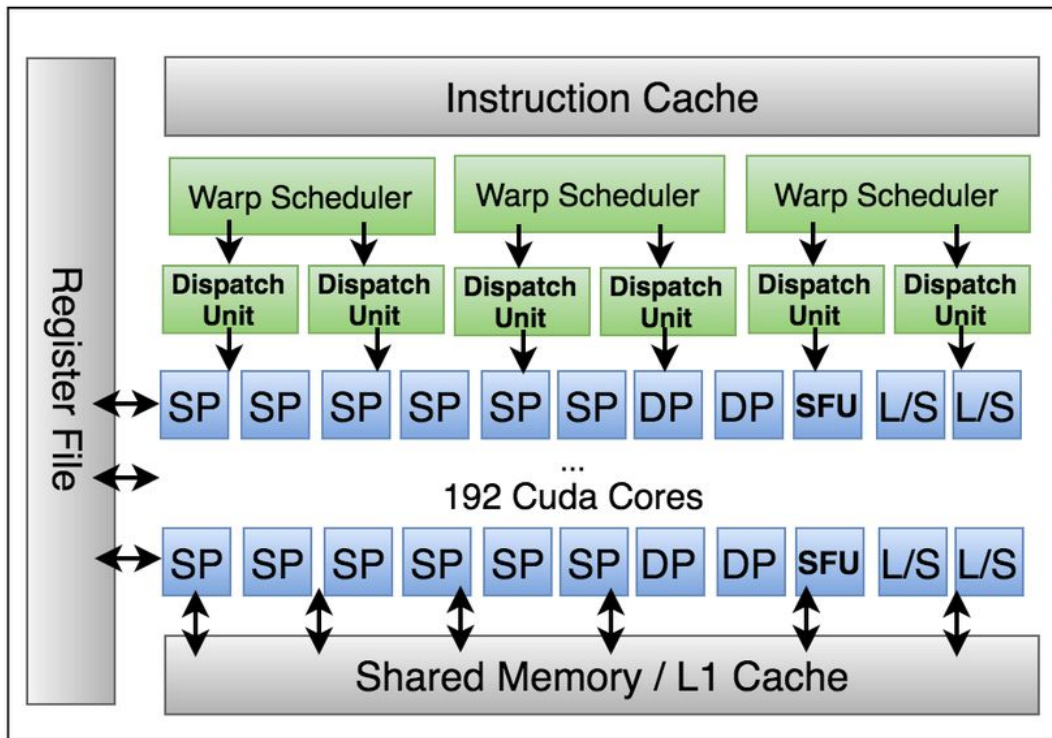
Kepler SM



But the streaming multiprocessors do not “execute in blocks”

They execute in warps

Kepler SM

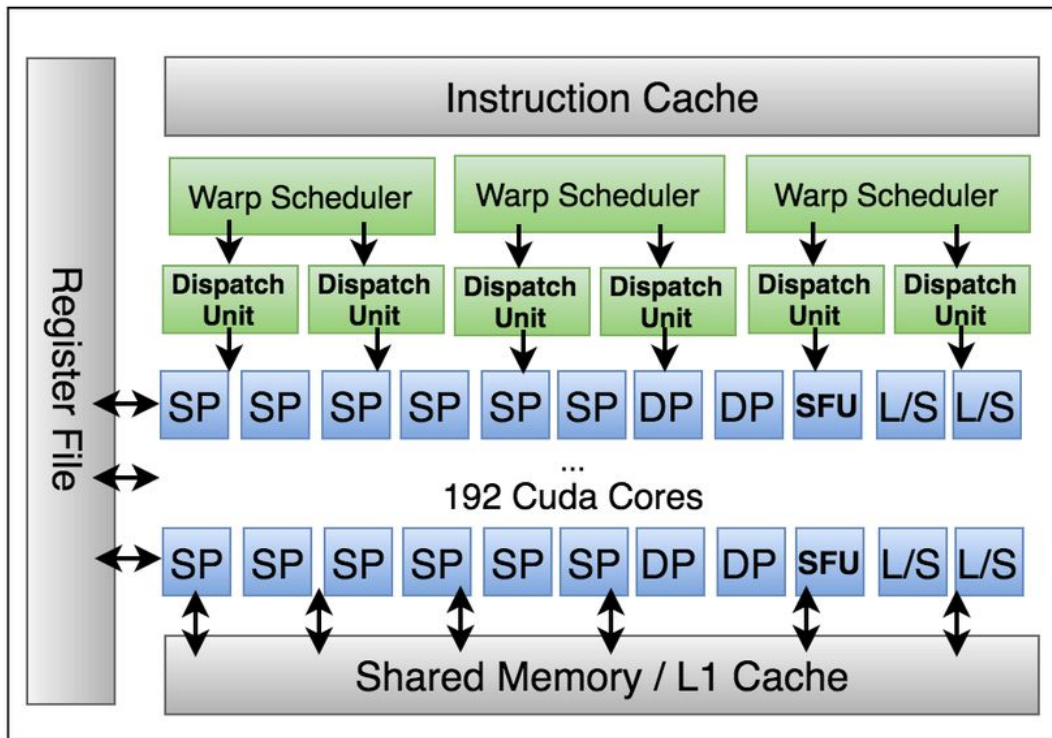


But the streaming multiprocessors do not “execute in blocks”

They execute in warps

Each warp is a group of 32 threads that will execute in 32 cores

Kepler SM



But the streaming multiprocessors do not “execute in blocks”

They execute in warps

Each warp is a group of 32 threads that will execute in 32 cores

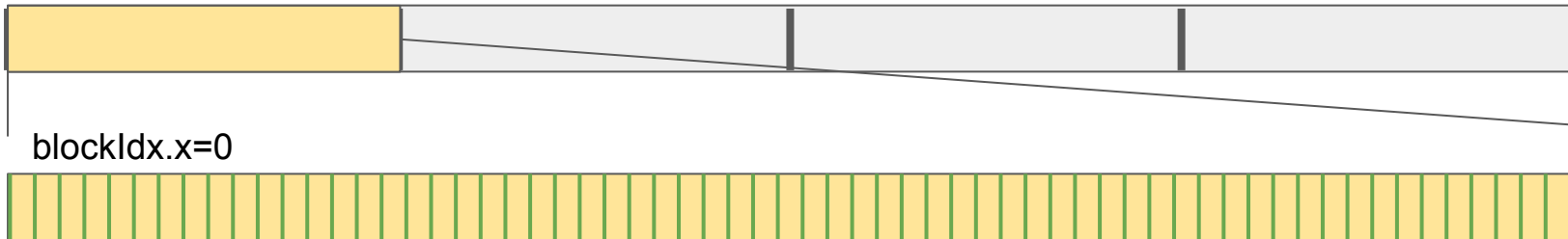
Warps are NOT part of the programming model

The **hardware** handles warps

- Warps are part of the execution model

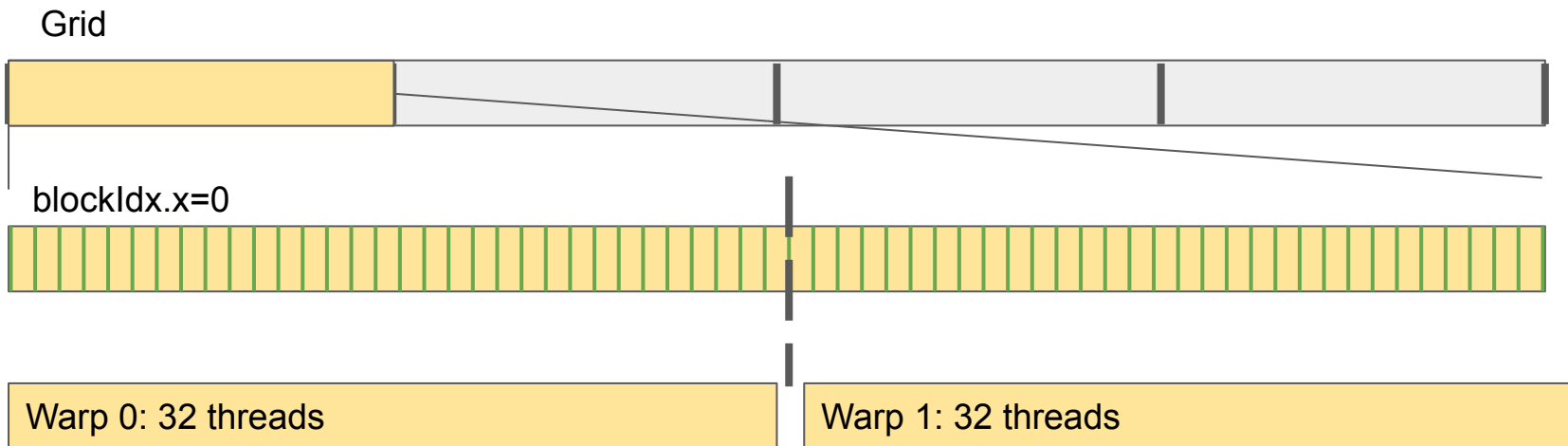
Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`

Grid



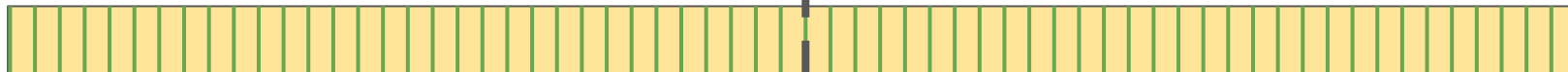
**Each warp is a group of
32 threads that will
execute in 32 cores**

Kernel call: `vAdd <<<4, 64>>>> (a, b, c, N)`



Each warp is a group of
32 threads that will
execute in 32 cores

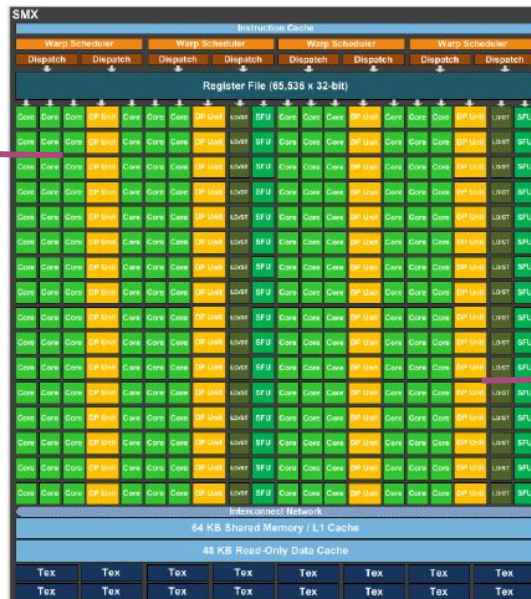
blockIdx.x=0



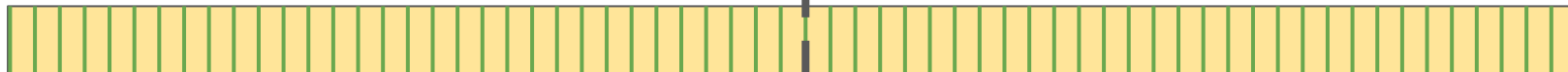
Warp 0: 32 threads

Warp 1: 32 threads

Each warp executes all its threads together on a 32 cores. It happens independently of other warps.



blockIdx.x=0



Warp 0: 32 threads

Warp 1: 32 threads

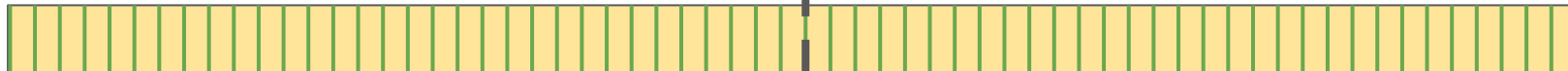


Each warp executes
all its threads
together on 32 cores.

W1

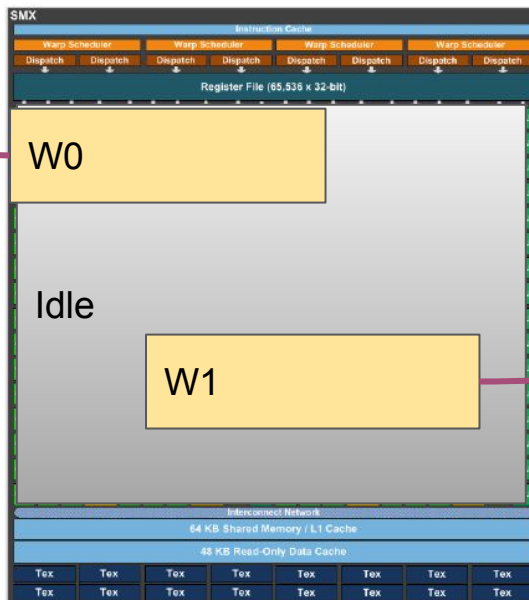
It happens
independently
of other
warps.

blockIdx.x=0



Warp 0: 32 threads

Warp 1: 32 threads



All the other cores
that are not
processing warps
will be idle.

What is best?

vAdd <<<8, 128>>>> (a, b, c, N)

vAdd <<<4, 256>>>> (a, b, c, N)

vAdd <<<2, 512>>>> (a, b, c, N)

vAdd <<<16, 64>>>> (a, b, c, N)

What is best?

vAdd <<<8, 128>>>> (a, b, c, N)

vAdd <<<4, 256>>>> (a, b, c, N)

These are definitely worse, why?

vAdd <<<2, 512>>>> (a, b, c, N)

vAdd <<<16, 64>>>> (a, b, c, N)