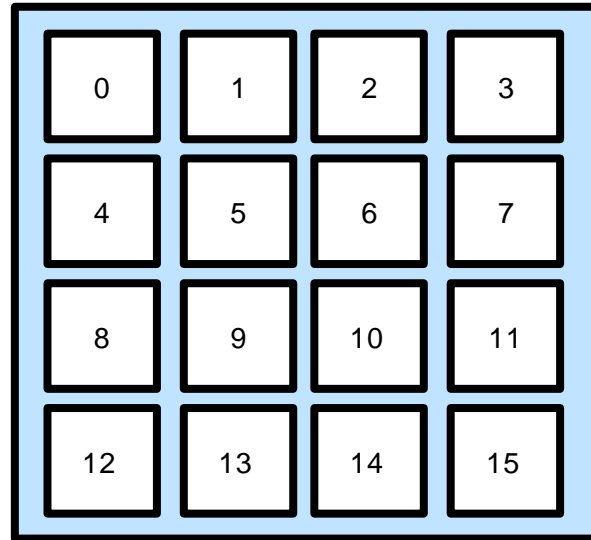
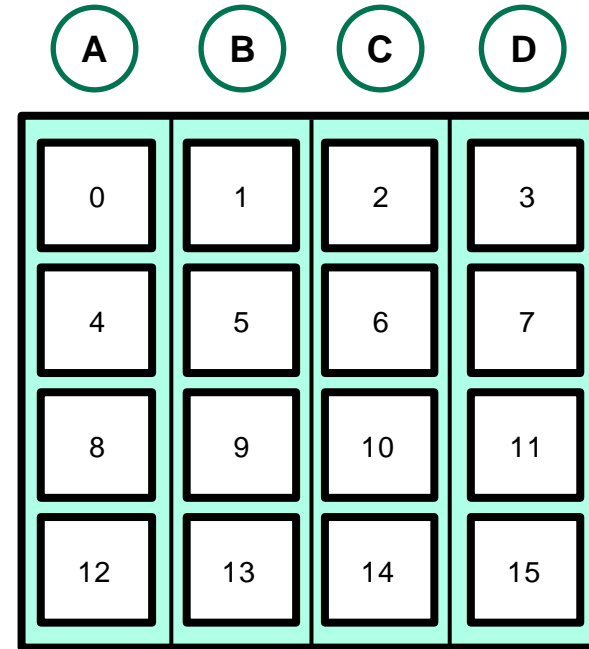


# Shared Memory Bank Conflicts

Shared memory is physically stored in  
**banks**

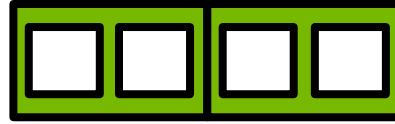


Logical Shared Memory  
`cuda.shared.array(4, 4)`

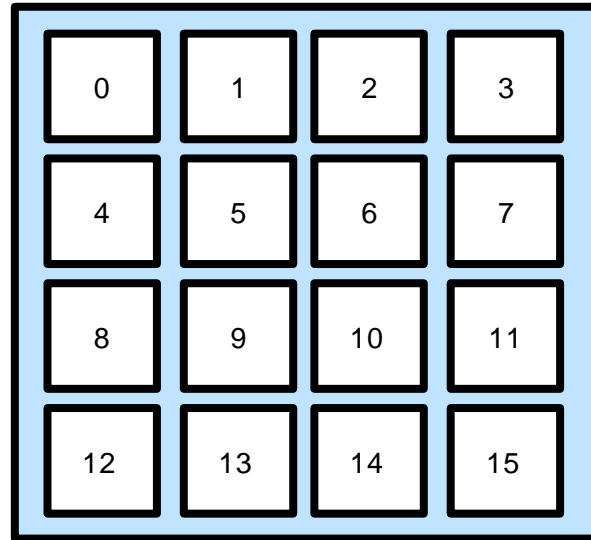


Physical Shared Memory  
in 4 banks

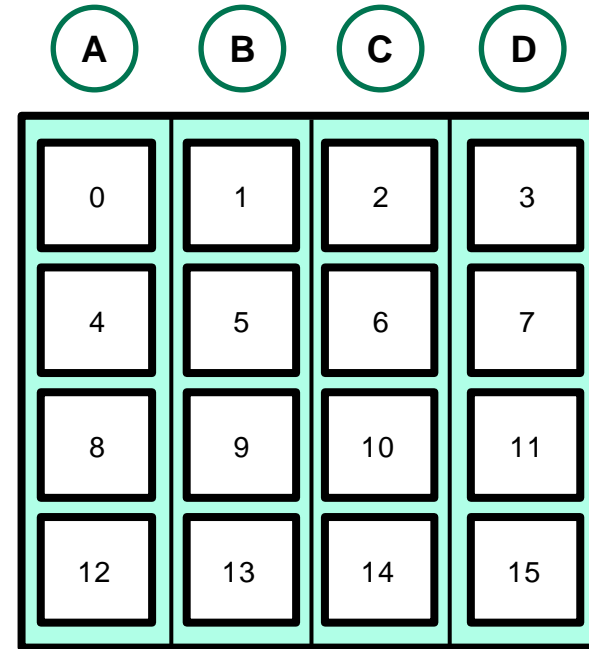
Warp



Actual shared memory is 32 4-byte wide banks. For space in these slides, we will portray shared memory as having 4 banks (**A**, **B**, **C**, **D**) and a warp as having 4 threads

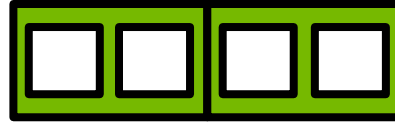


Logical Shared Memory  
`cuda.shared.array(4, 4)`

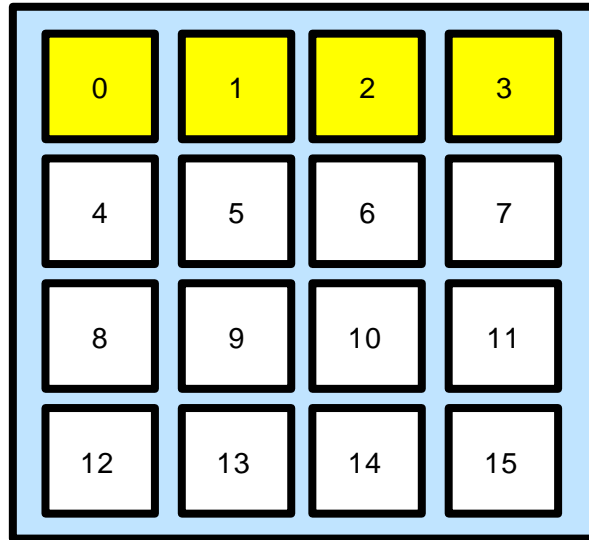


Physical Shared Memory  
in 4 banks

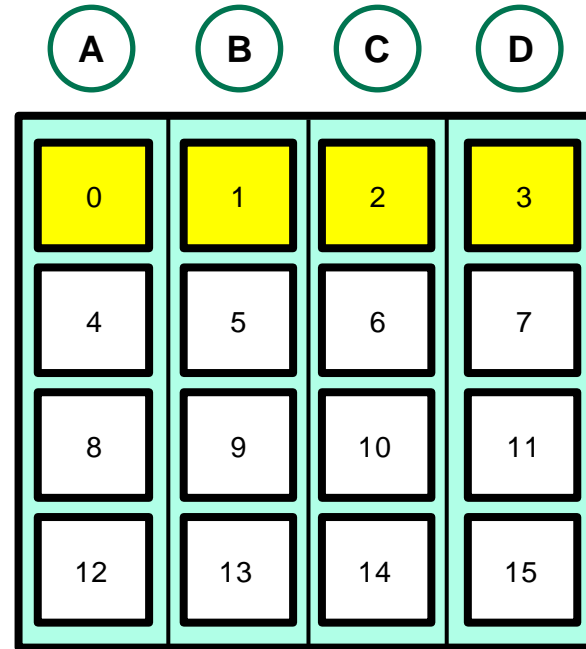
Warp



Successive 4-byte words (1 box in these slides) will belong to different banks



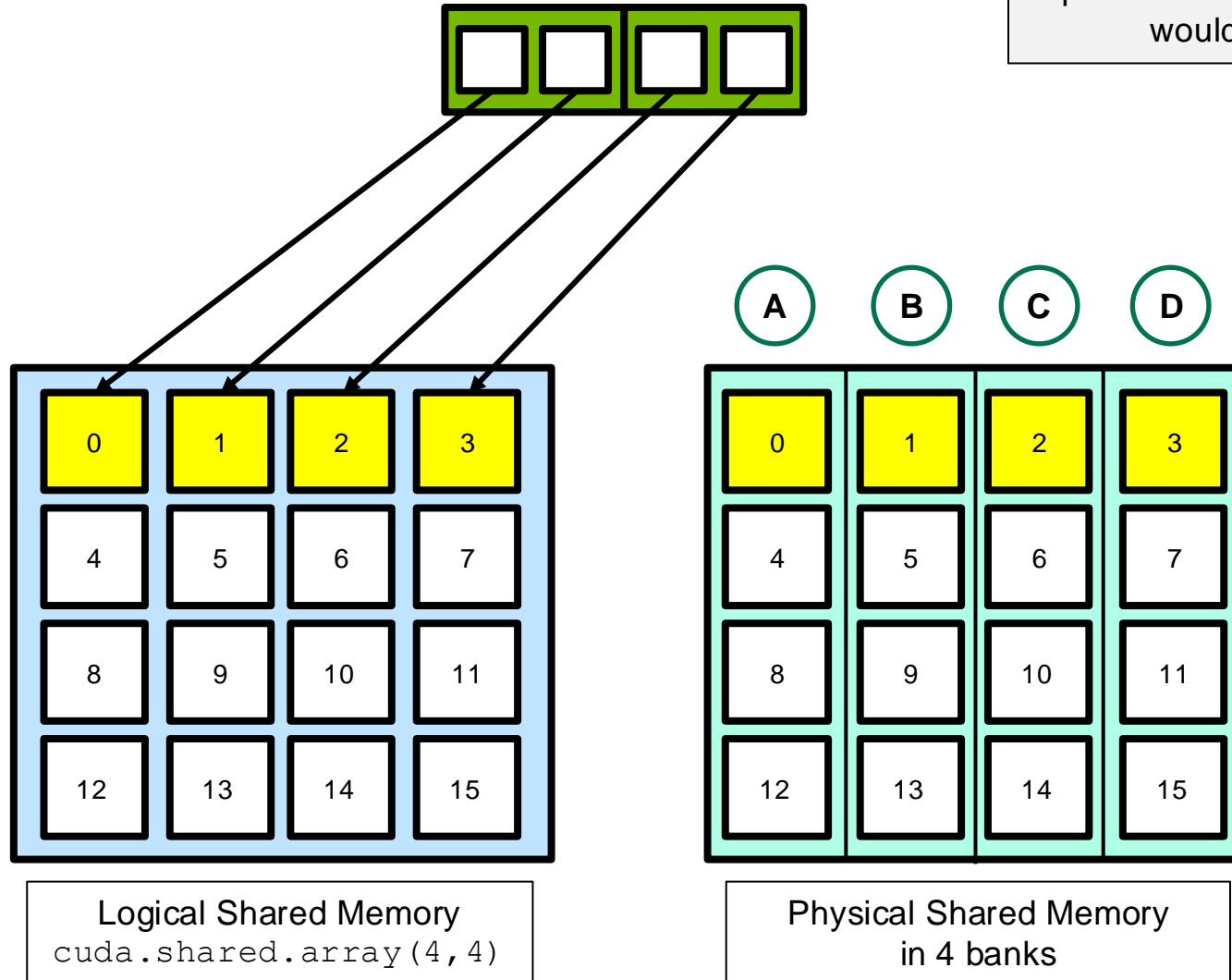
Logical Shared Memory  
`cuda.shared.array(4, 4)`



Physical Shared Memory  
in 4 banks

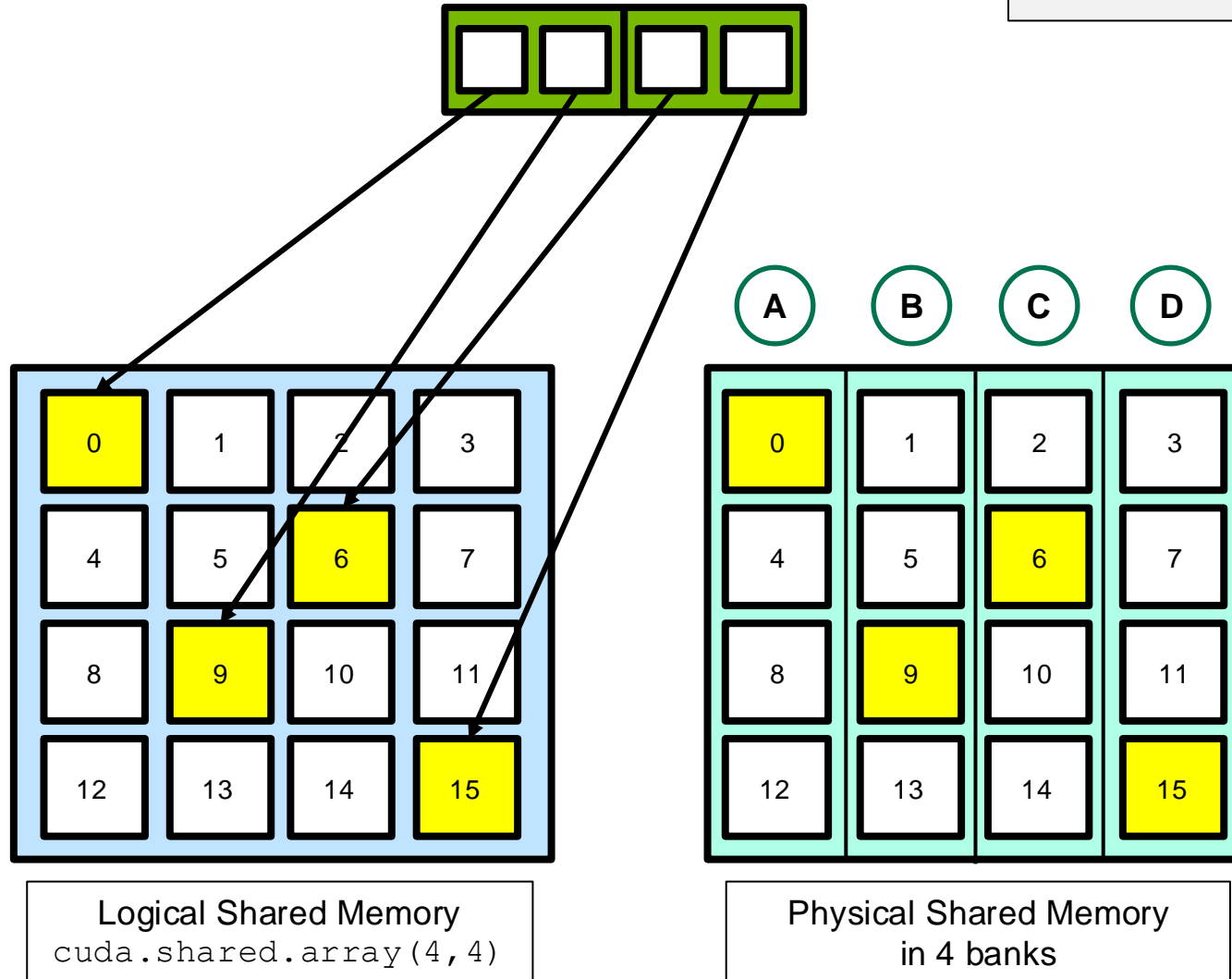
Warp

A warp can access 4 bytes per bank, in parallel. This shared memory access would occur all at once

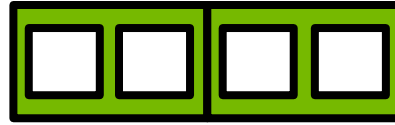


Warp

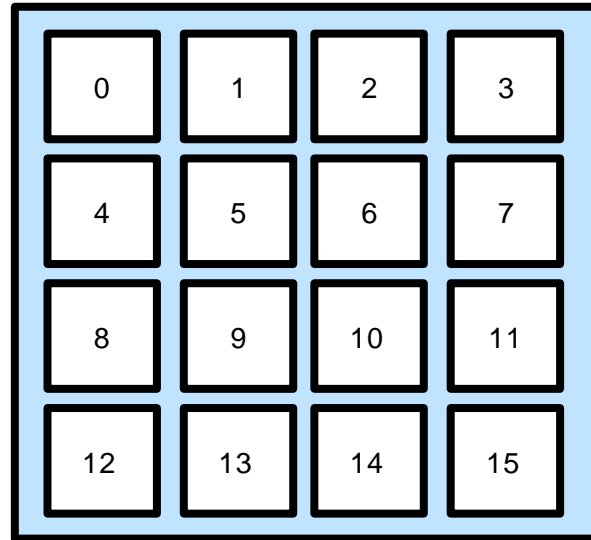
So would this one, since each element is in a different bank



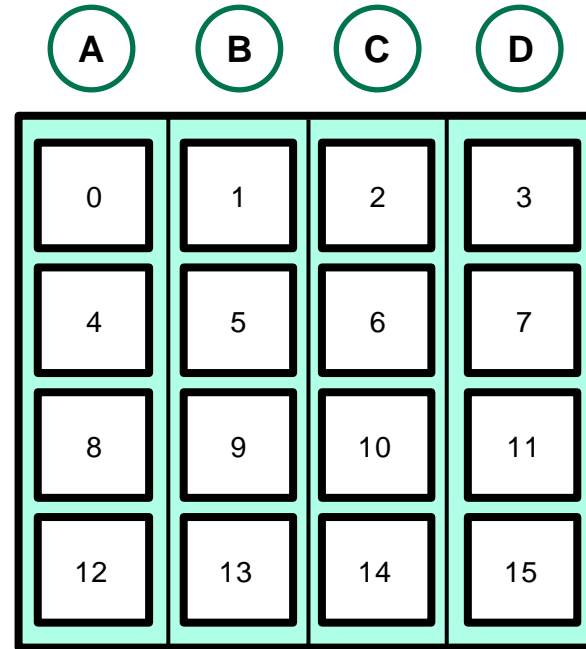
Warp



Memory accesses in the same bank result in the access operations being serialized. We call this a **bank conflict**.



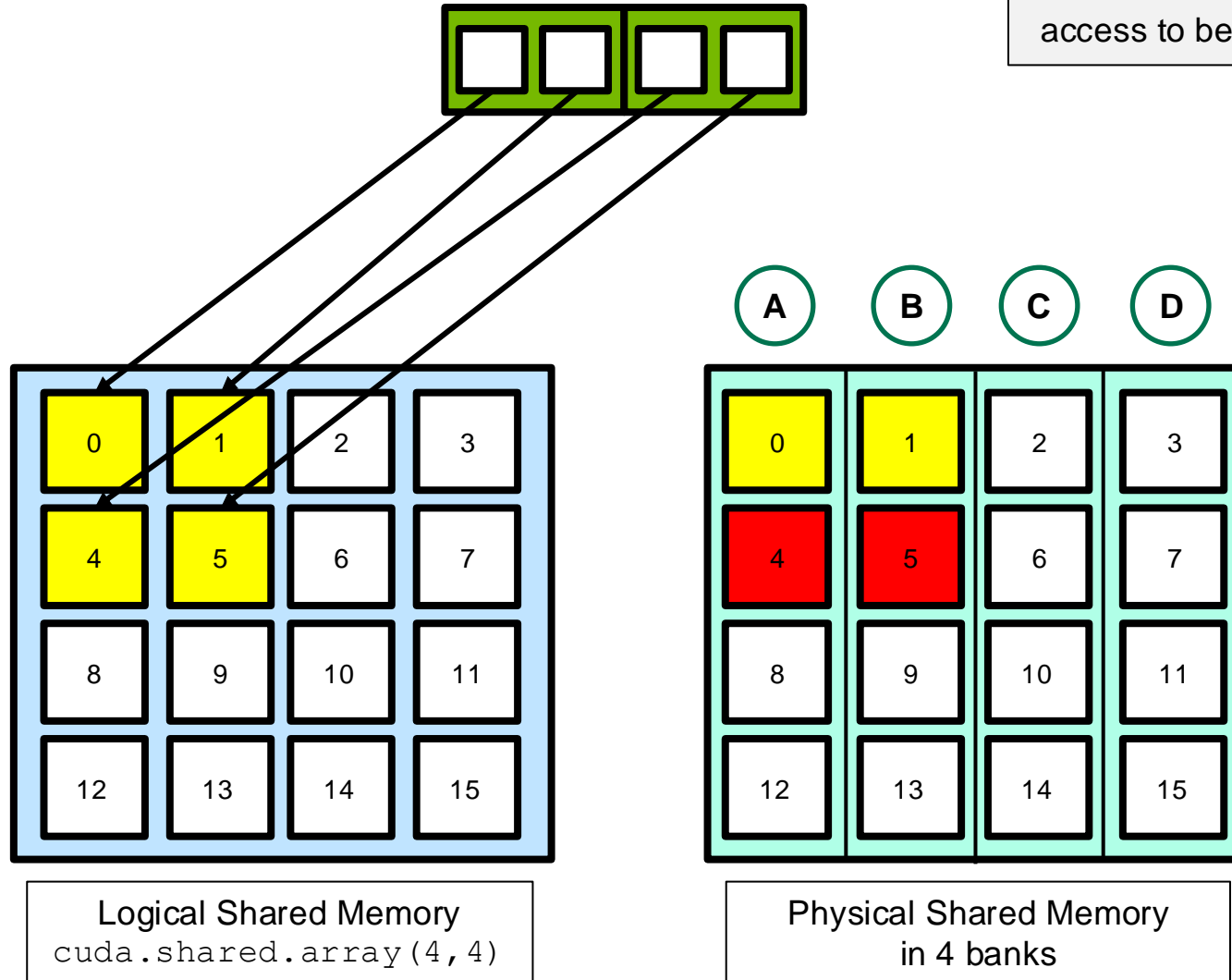
Logical Shared Memory  
`cuda.shared.array(4, 4)`



Physical Shared Memory  
in 4 banks

Warp

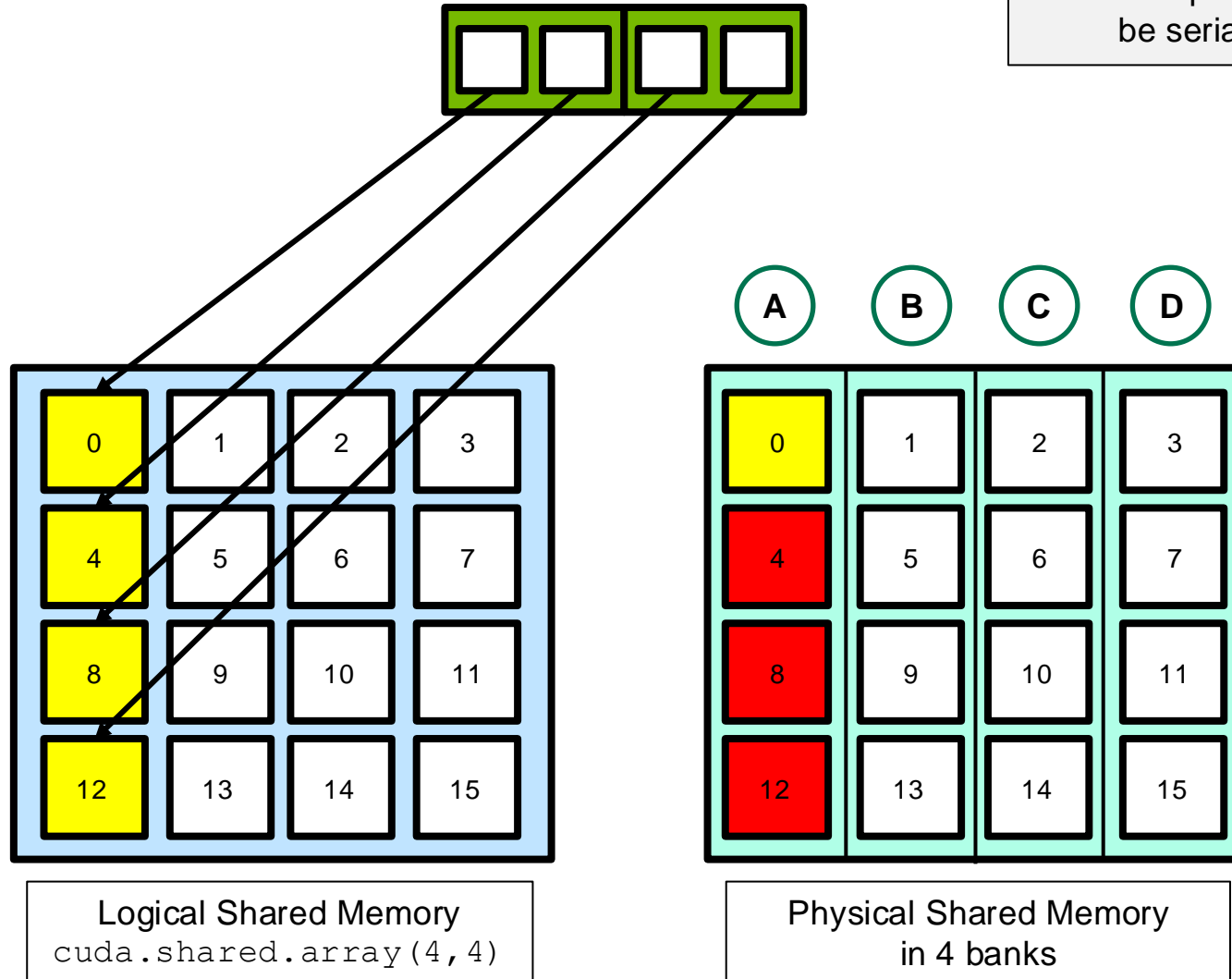
In this scenario, we have a 2-way bank conflict that would require the memory access to be serialized over 2 cycles.

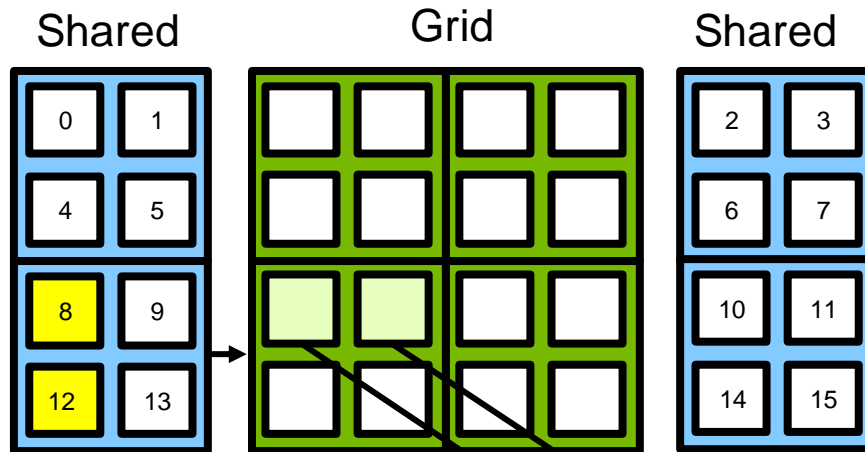
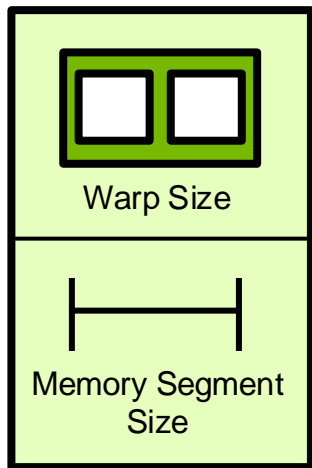




Warp

Here have a 4-way bank conflict that would require the memory access to be serialized over 4 cycles.



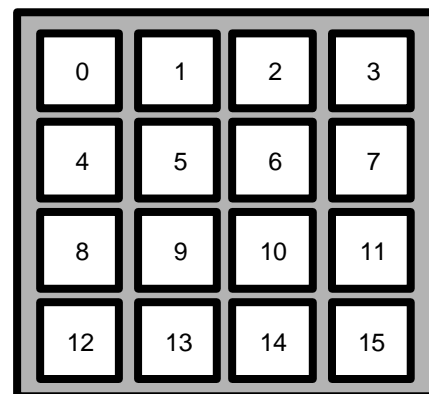


Recall from our earlier matrix transpose example that we were making this very kind of **columnar read** from shared memory, which means we had significant **bank conflicts**

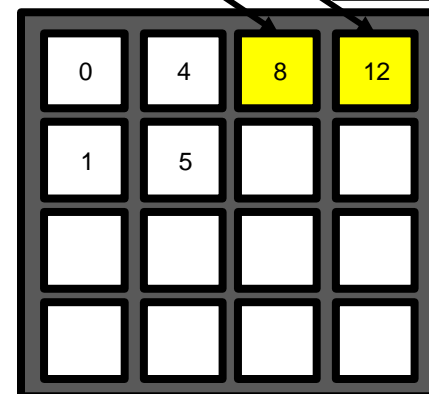
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
cuda.syncthreads()

o_x = bId.y*bDim.y + tId.x
o_y = bId.x*bDim.x + tId.y
o[o_y][o_x] = tile[tIdx.x][tIdx.y]
```



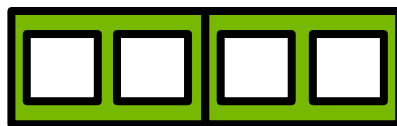
Input



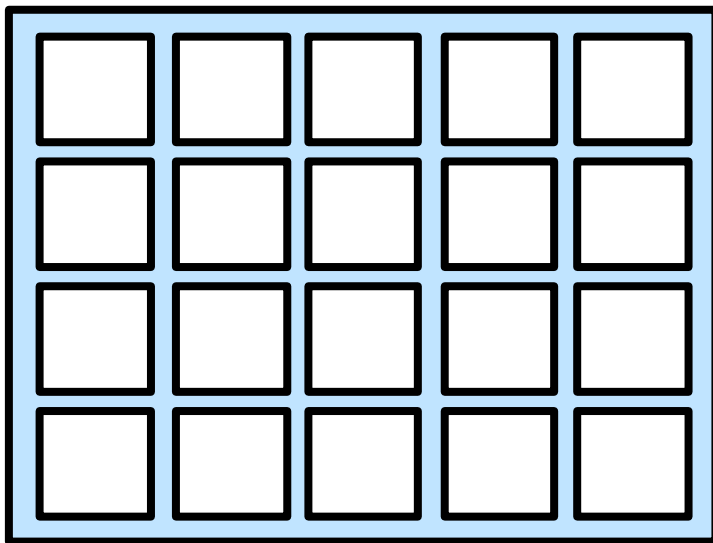
Output

Here is a technique we can use to avoid bank conflicts when we know we need to make columnar access to shared memory

Warp

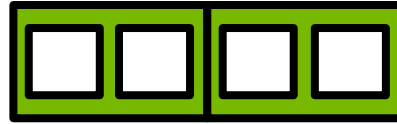


First, when we allocate our shared memory tile, we will pad it with an extra column

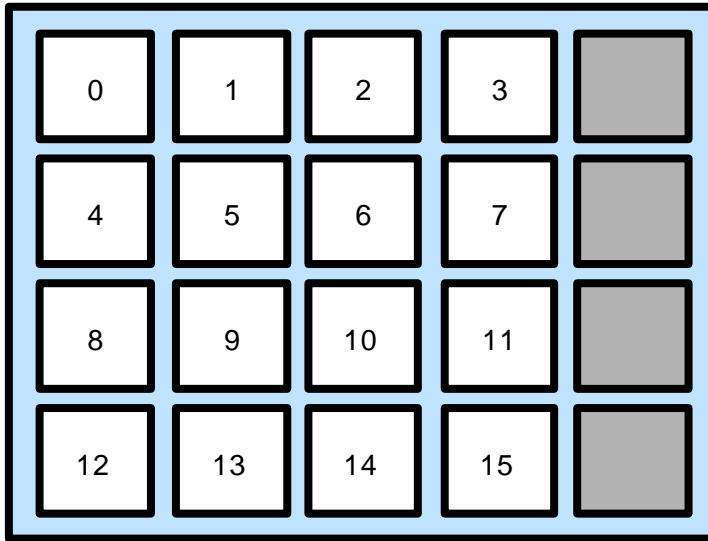


Logical Shared Memory  
`cuda.shared.array(4, 5)`

Warp

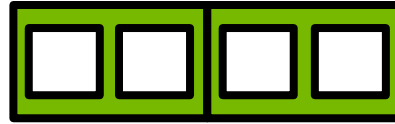


Next, when we write to the tile, we act as if the tile is (4,4) and only write to addresses in the range [0:4][0:4]



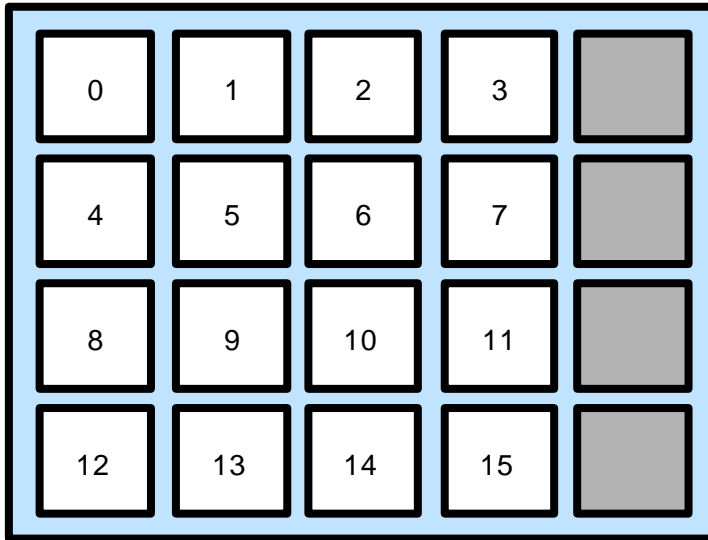
Logical Shared Memory  
`cuda.shared.array(4, 5)`

Warp

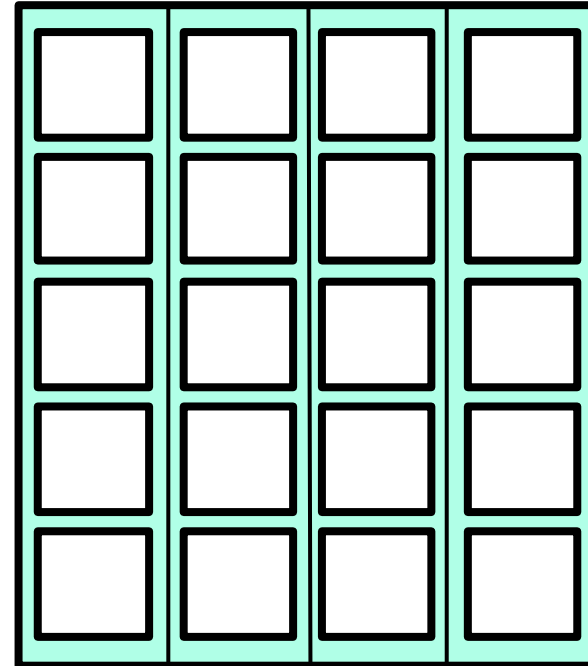


The physical shared memory has a fixed size of 32 banks (4 banks in our slides to save space), so our padding of the shared memory array does not affect the number of memory banks

A B C D

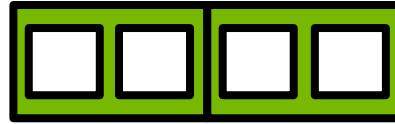


Logical Shared Memory  
`cuda.shared.array(4, 5)`

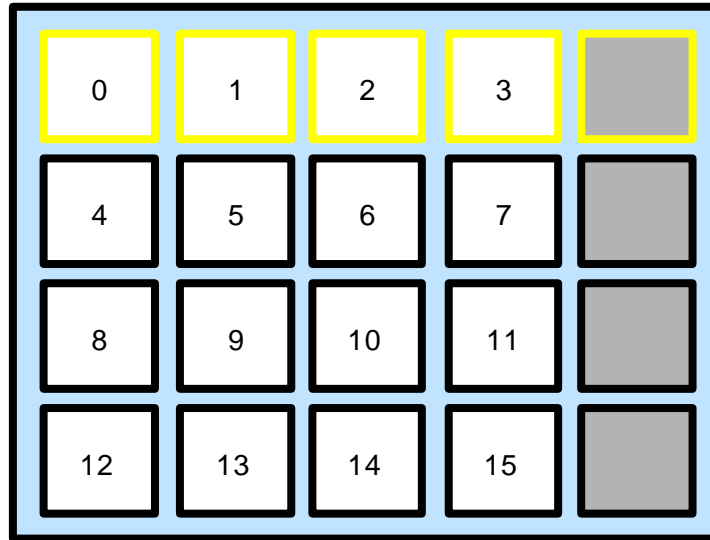


Physical Shared Memory  
in 4 banks

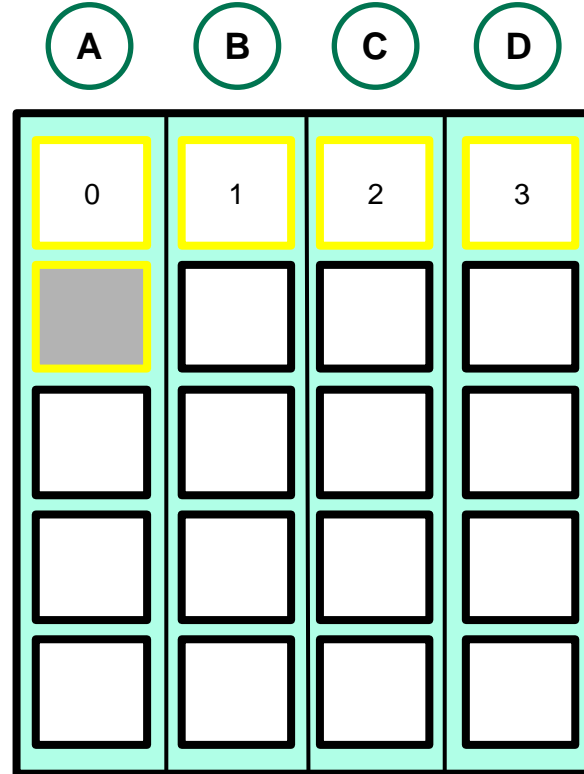
Warp



So if we consider how the array is laid out within the memory banks, we see the following:

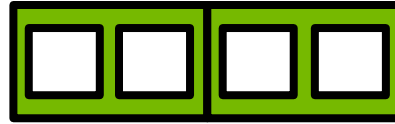


Logical Shared Memory  
`cuda.shared.array(4, 5)`

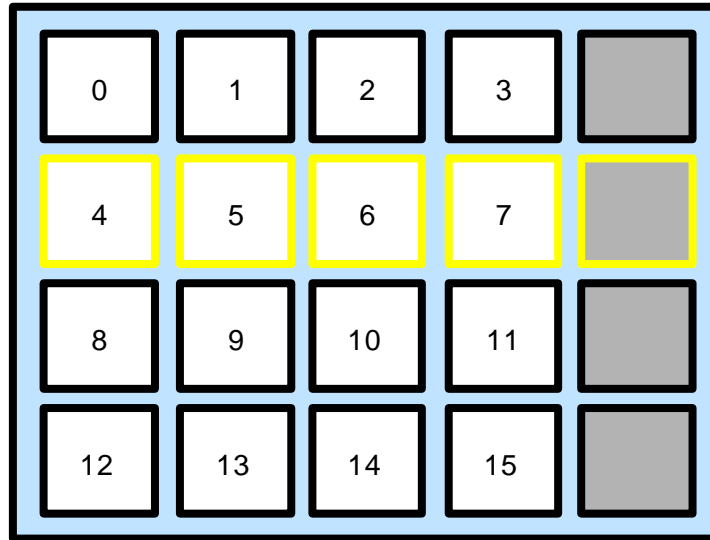


Physical Shared Memory  
in 4 banks

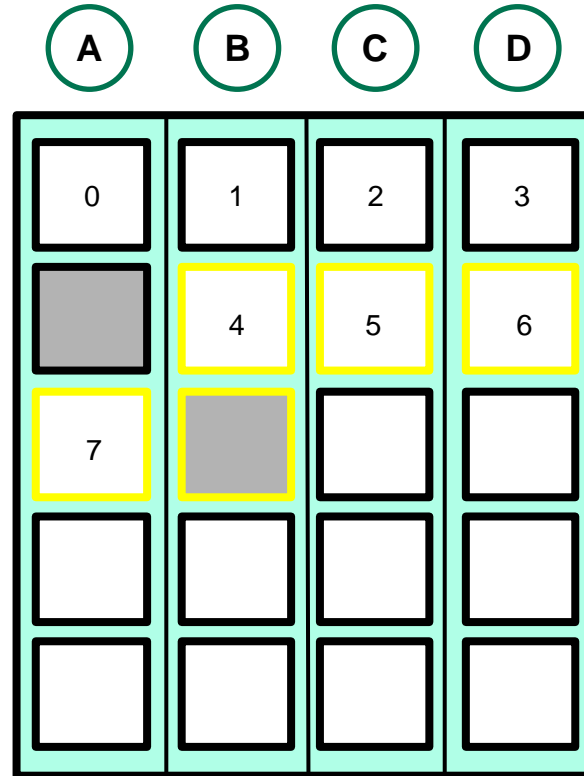
Warp



So if we consider how the array is laid out within the memory banks, we see the following:



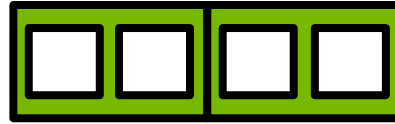
Logical Shared Memory  
`cuda.shared.array(4, 5)`



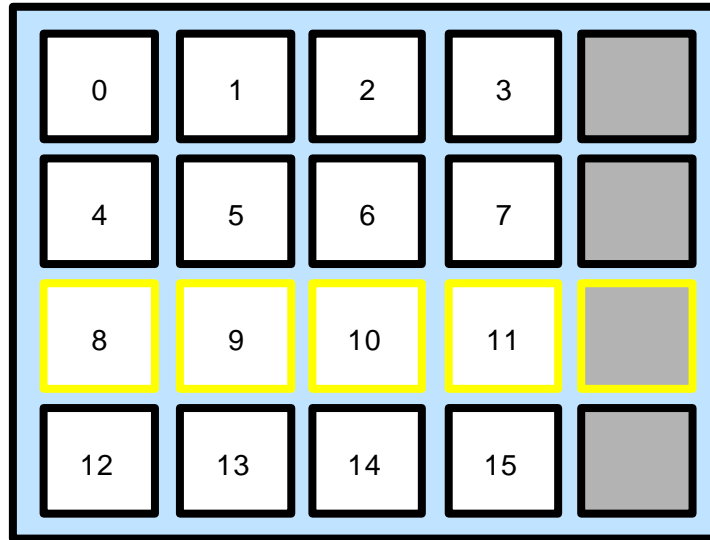
Physical Shared Memory  
in 4 banks



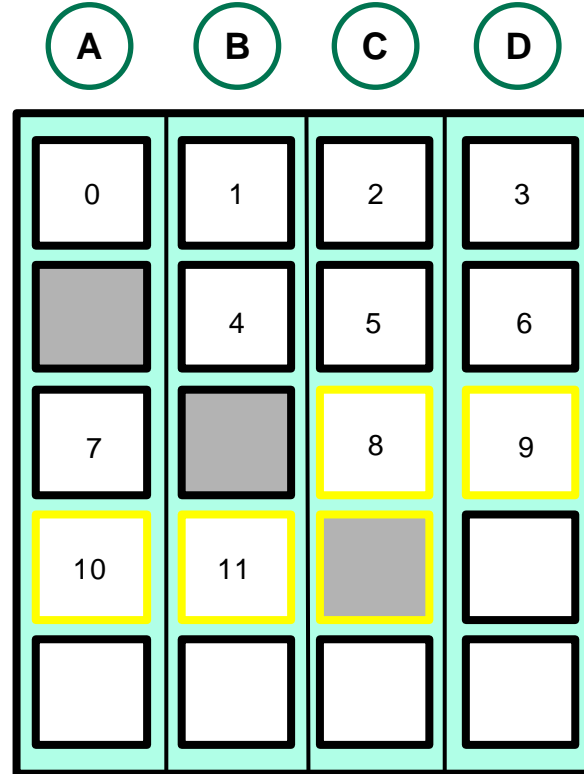
Warp



So if we consider how the array is laid out within the memory banks, we see the following:

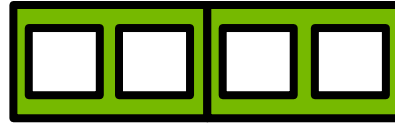


Logical Shared Memory  
`cuda.shared.array(4, 5)`

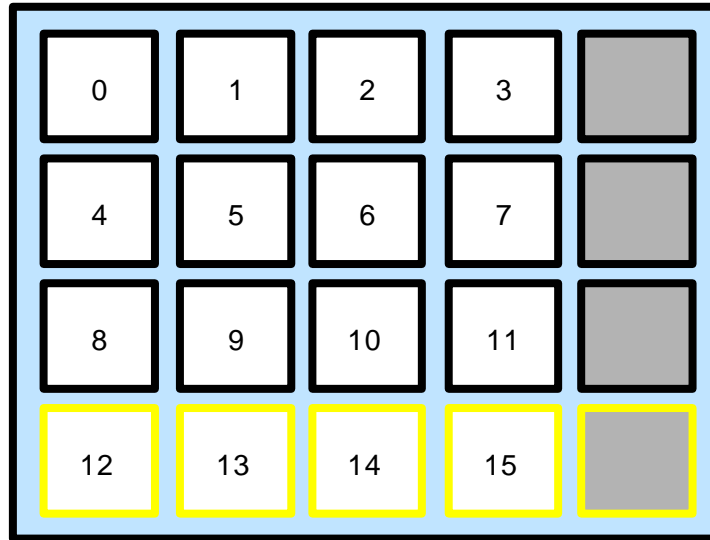


Physical Shared Memory  
in 4 banks

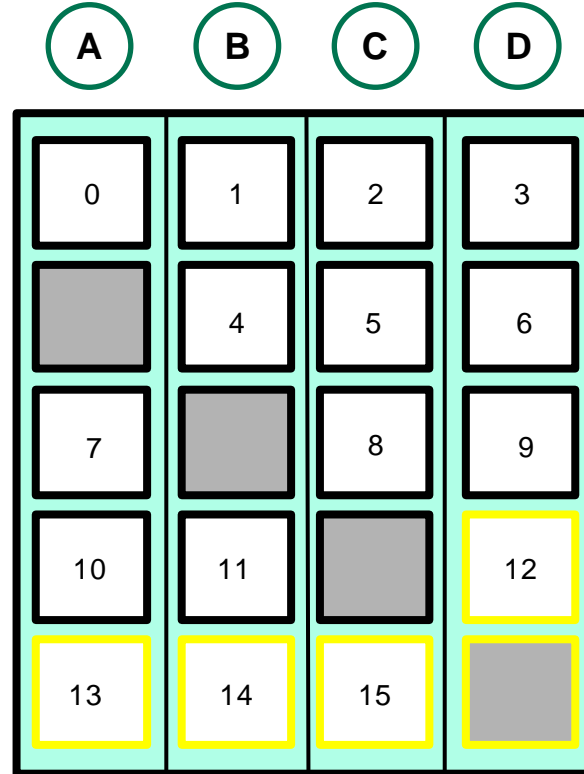
Warp



So if we consider how the array is laid out within the memory banks, we see the following:

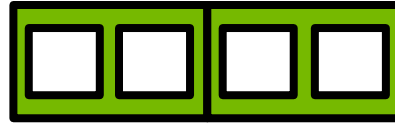


Logical Shared Memory  
`cuda.shared.array(4, 5)`

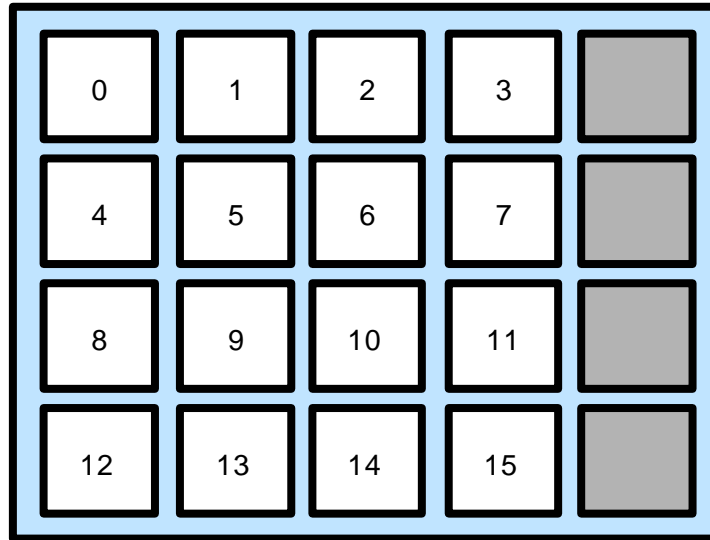


Physical Shared Memory  
in 4 banks

Warp

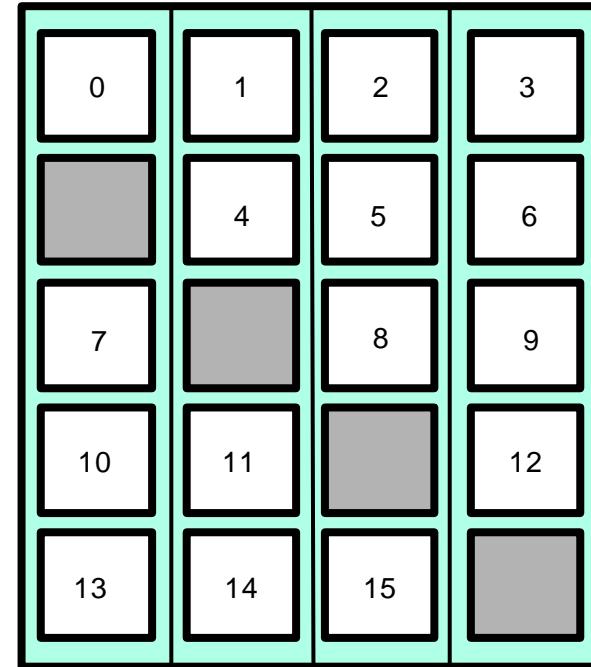


Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



Logical Shared Memory  
`cuda.shared.array(4, 5)`

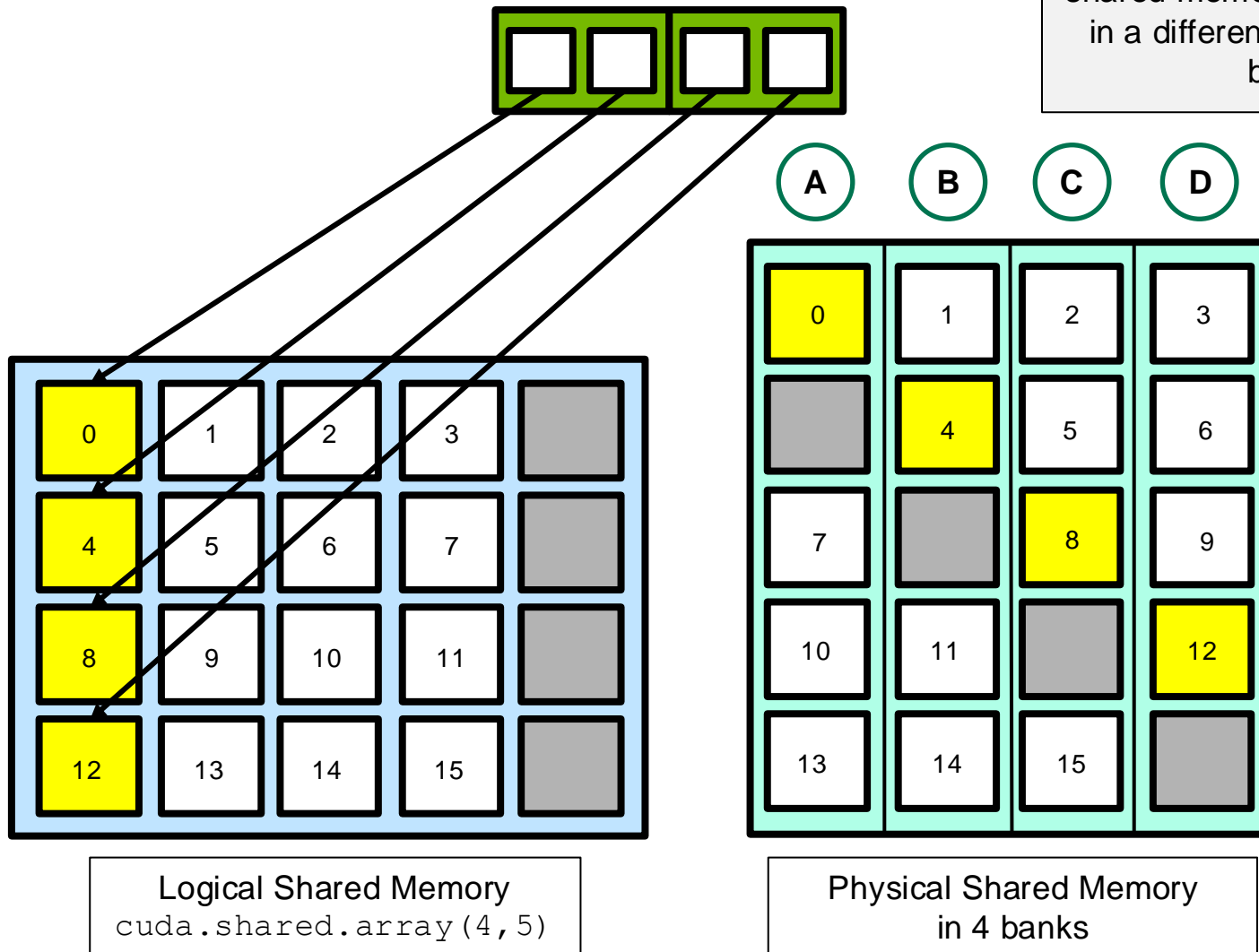
A B C D



Physical Shared Memory  
in 4 banks

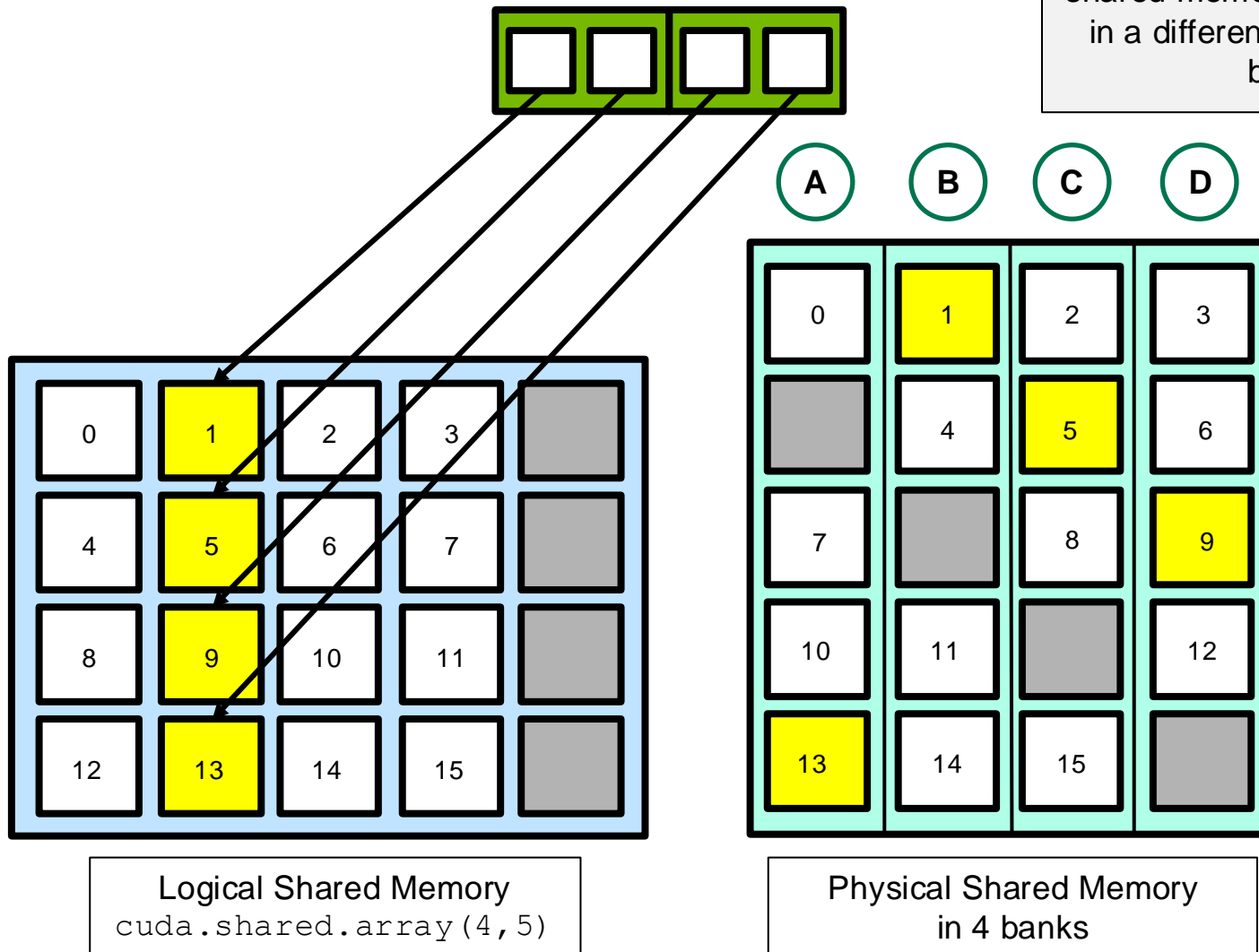
Warp

Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



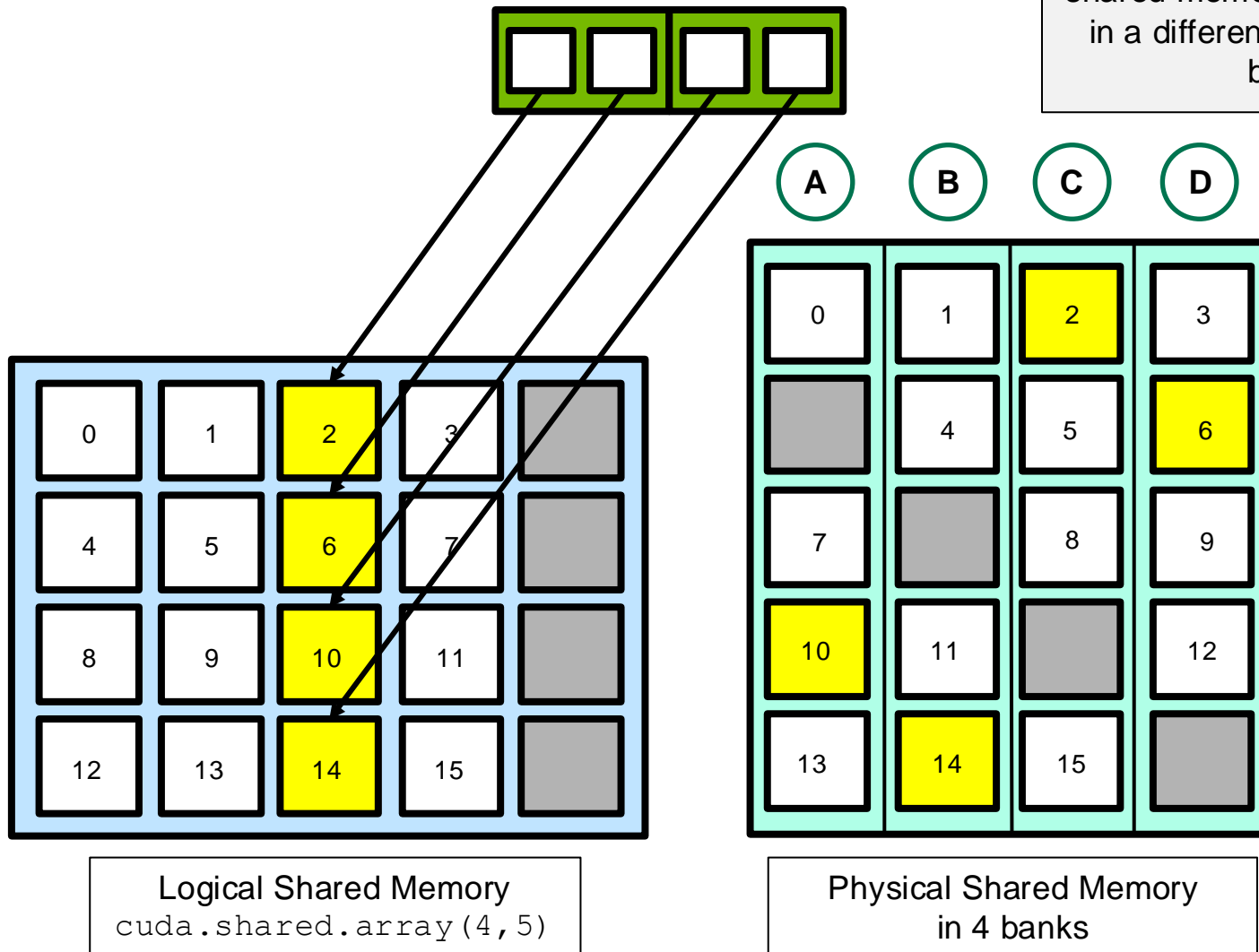
Warp

Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



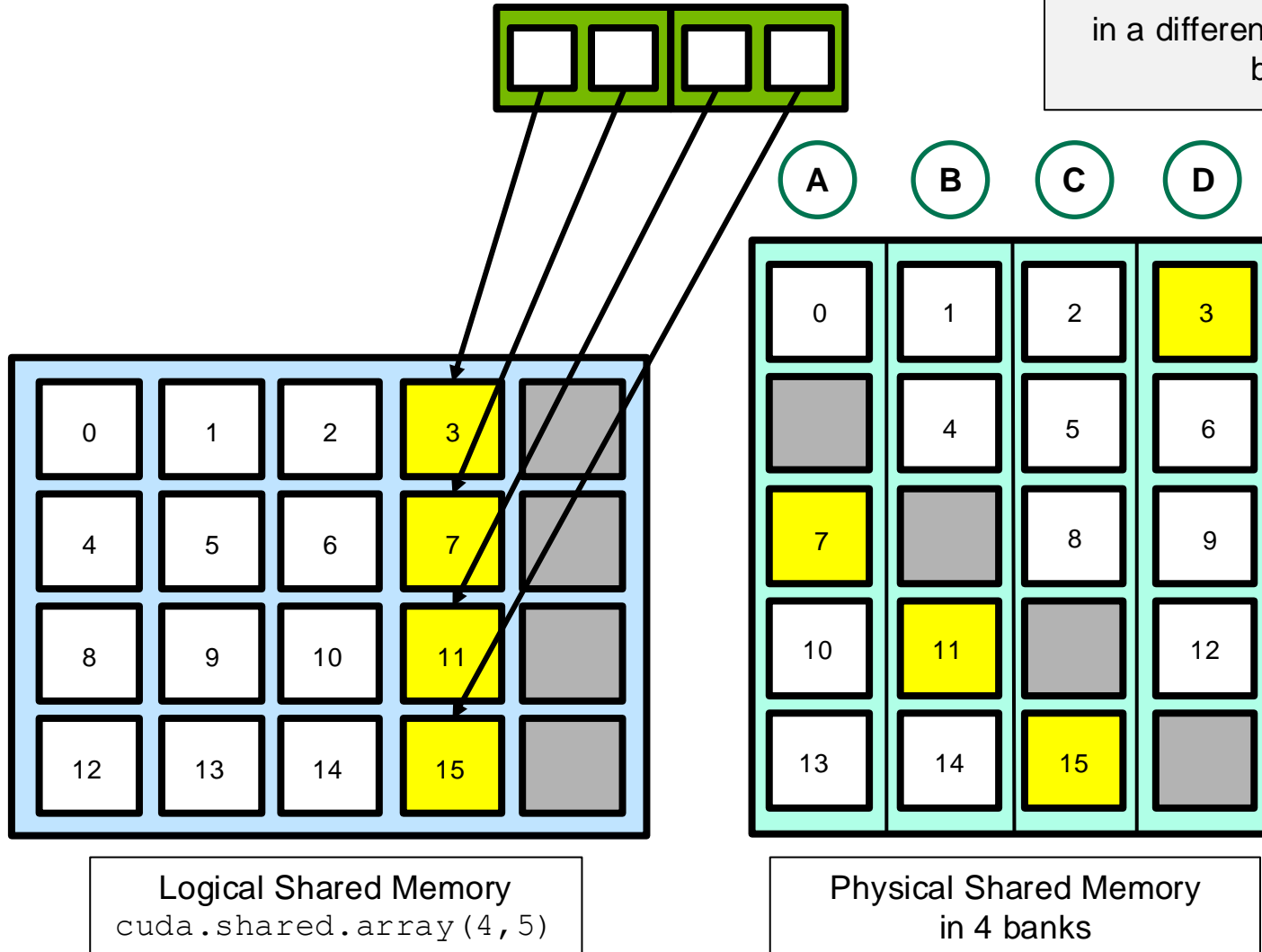
Warp

Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



Warp

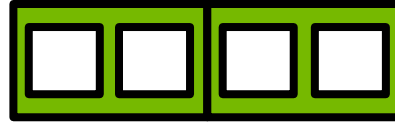
Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



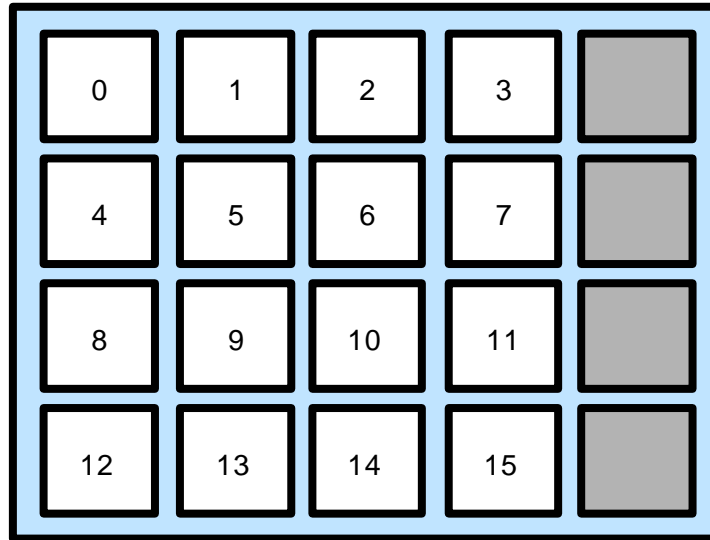
Logical Shared Memory  
`cuda.shared.array(4, 5)`

Physical Shared Memory  
in 4 banks

Warp

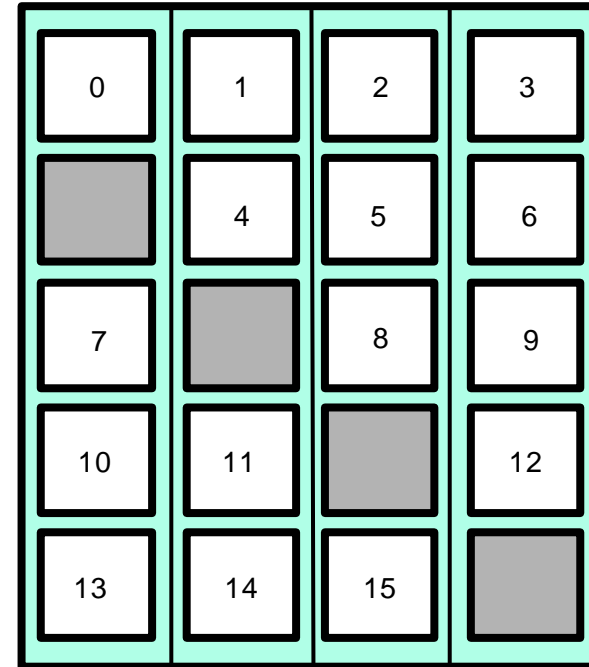


Worth mentioning that to use this technique for this example, the only change we had to make to our code was add one extra column to our shared memory allocation



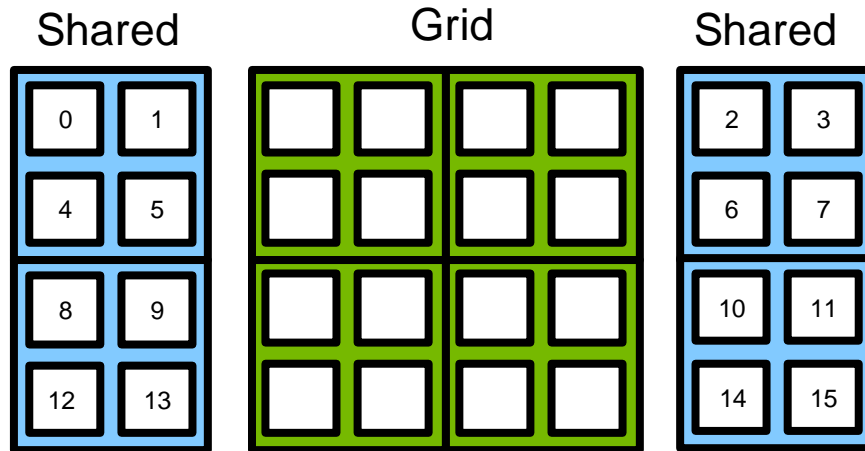
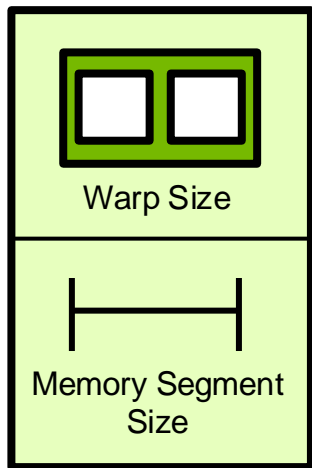
Logical Shared Memory  
`cuda.shared.array(4, 5)`

A B C D



Physical Shared Memory  
in 4 banks





From our earlier matrix transpose example, the single change in **green** below would suffice to avoid bank conflicts while retaining correctness

```
tile = cuda.shared.array(2,3)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
cuda.syncthreads()

o_x = bId.y*bDim.y + tId.x
o_y = bId.x*bDim.x + tId.y
o[o_y][o_x] = tile[tIdx.x][tIdx.y]
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Input

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Output



DEEP  
LEARNING  
INSTITUTE

[www.nvidia.com/dli](http://www.nvidia.com/dli)