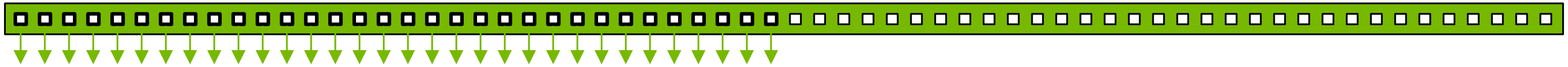
Memory Coalescing

Recall that thread blocks are divided into **warps** of 32 threads

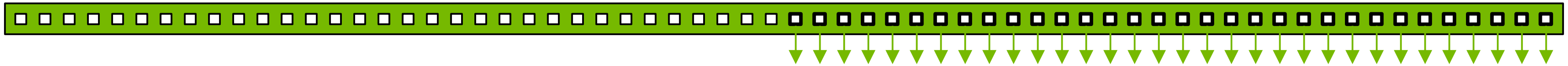
Recall that thread blocks are divided into **warps** of 32 threads

Recall that thread blocks are divided into **warps** of 32 threads

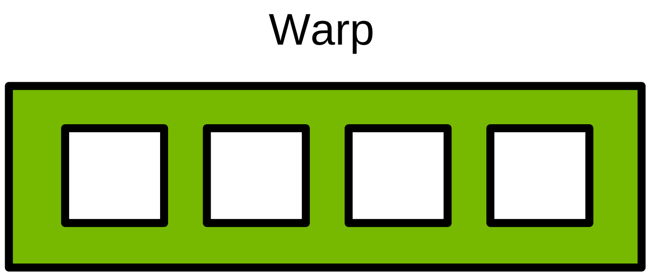
Instructions are issued in parallel at the warp level of 32 threads



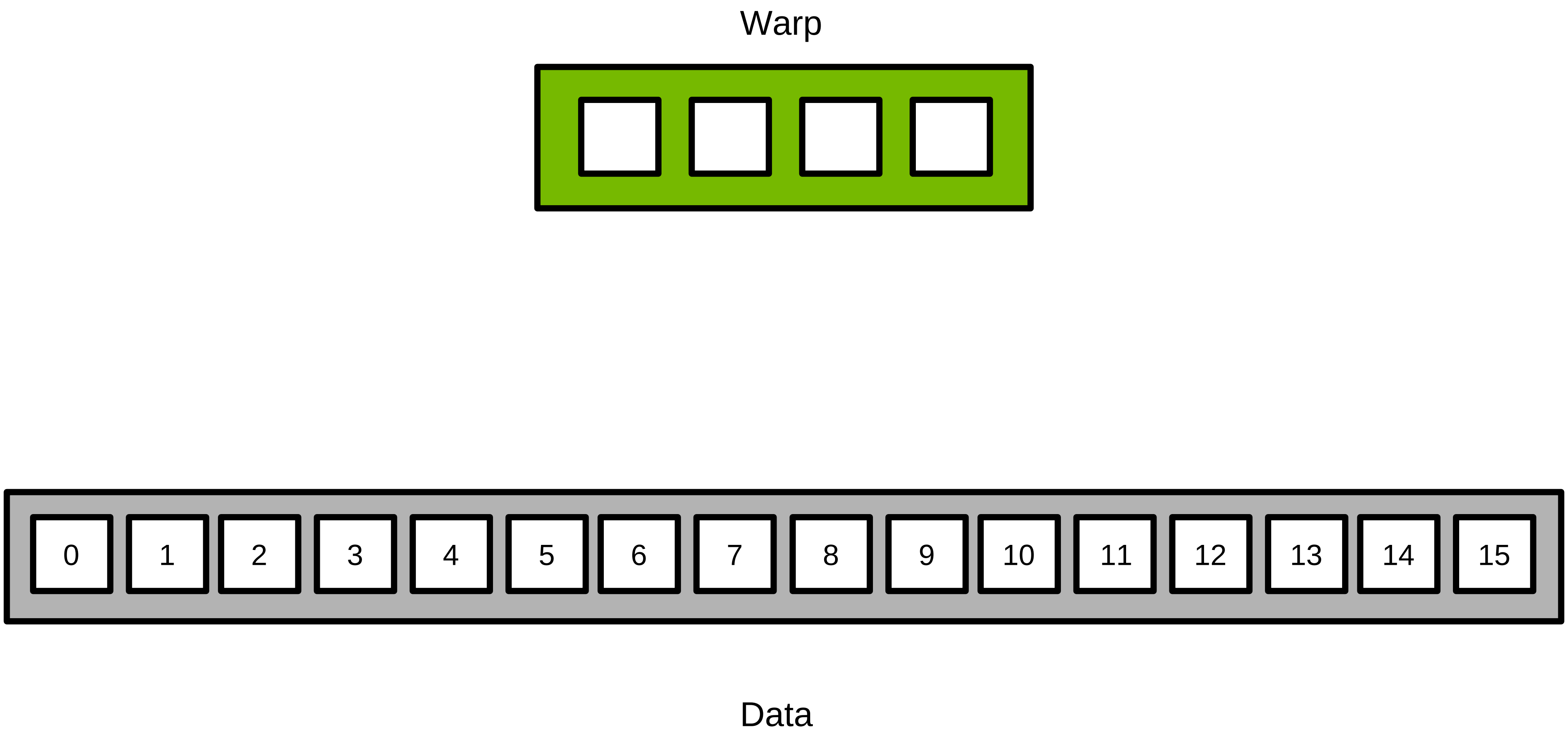
Instructions are issued in parallel at the warp level of 32 threads



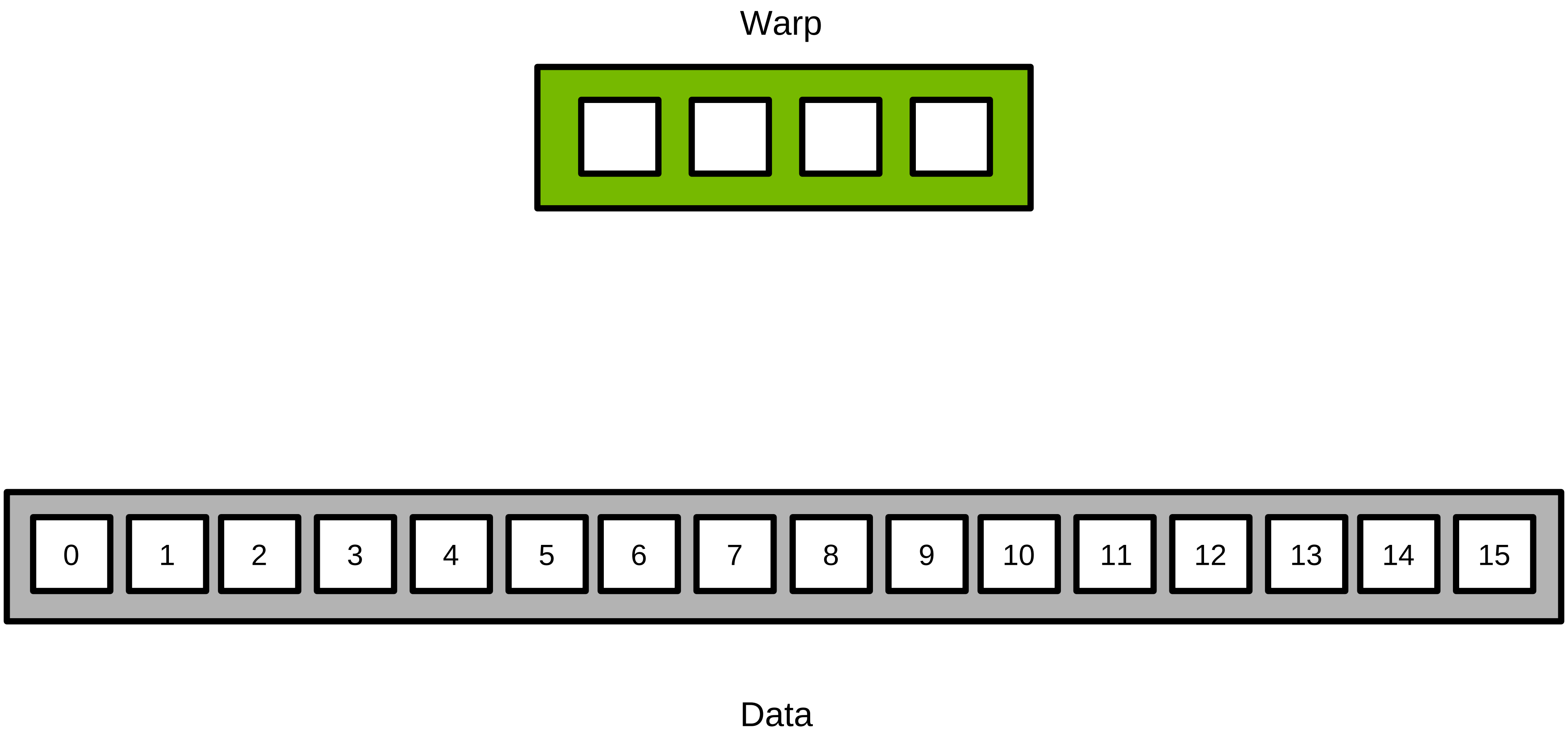
For space on these slides, we will treat just 4 threads as a warp



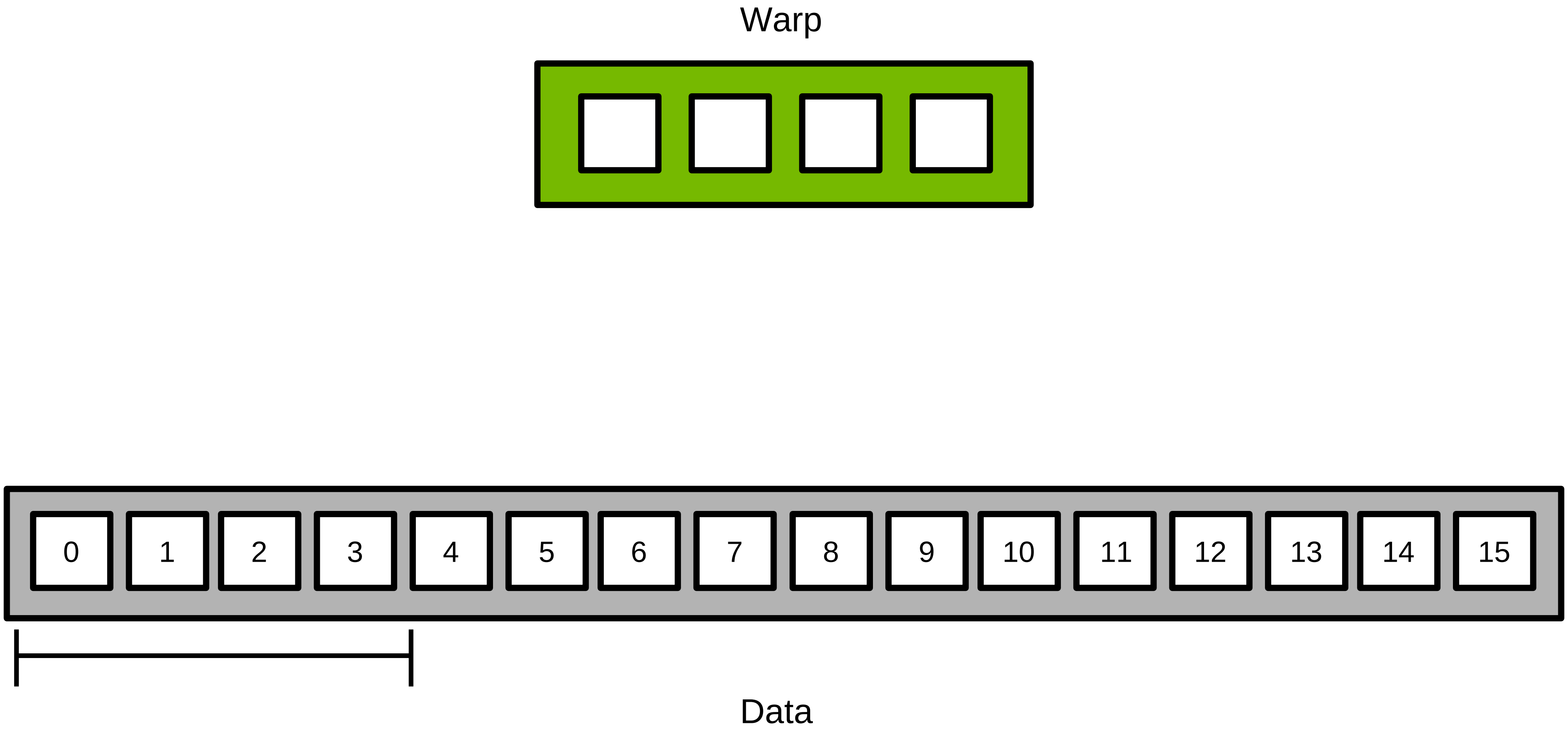
Data is transferred to and from global device memory in 32-byte segments*



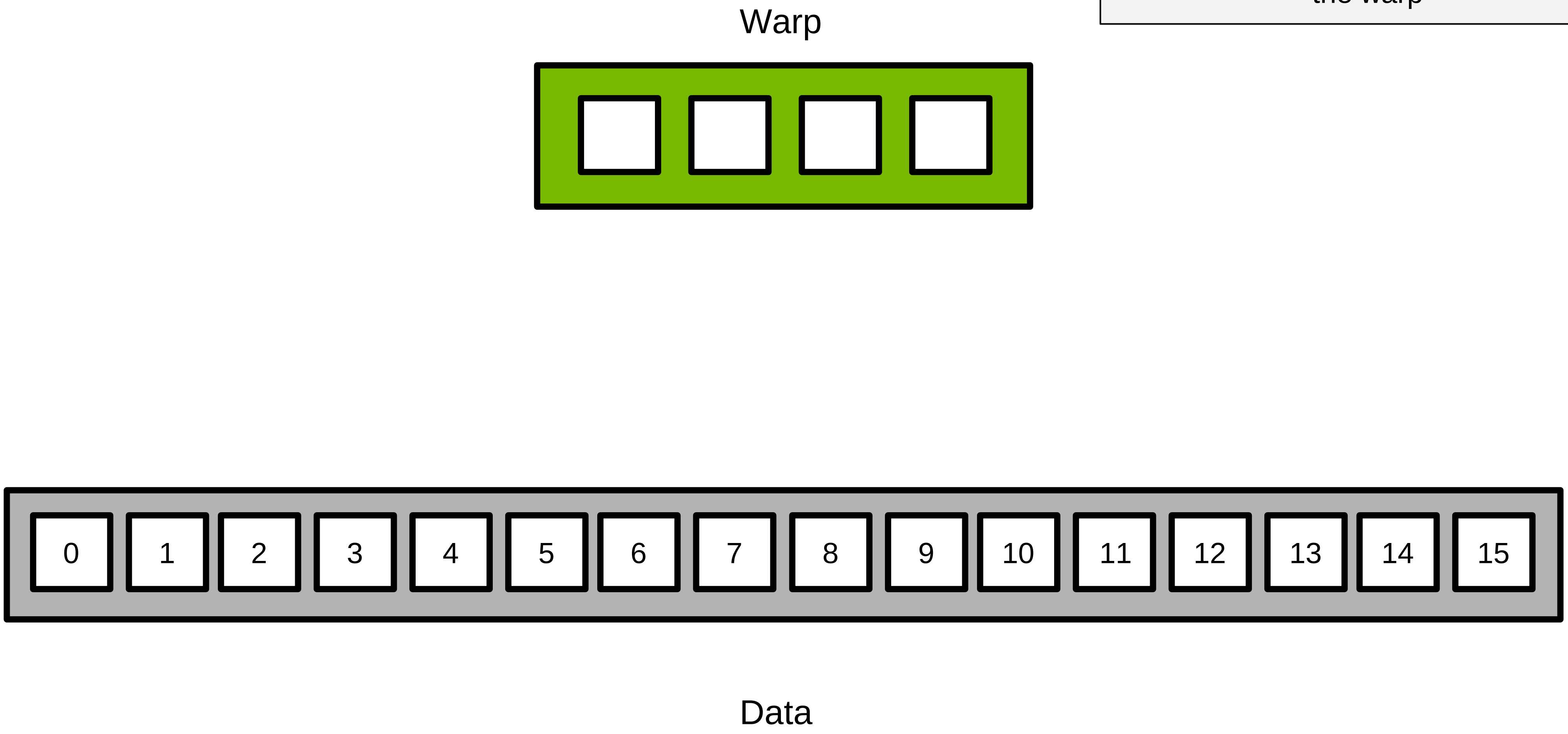
(* If the data is in the L1 cache it will be transferred in 128-byte cache lines – see the notebook for details)



For these slides we will treat 4 data elements as one of these fixed-length lines of contiguous memory

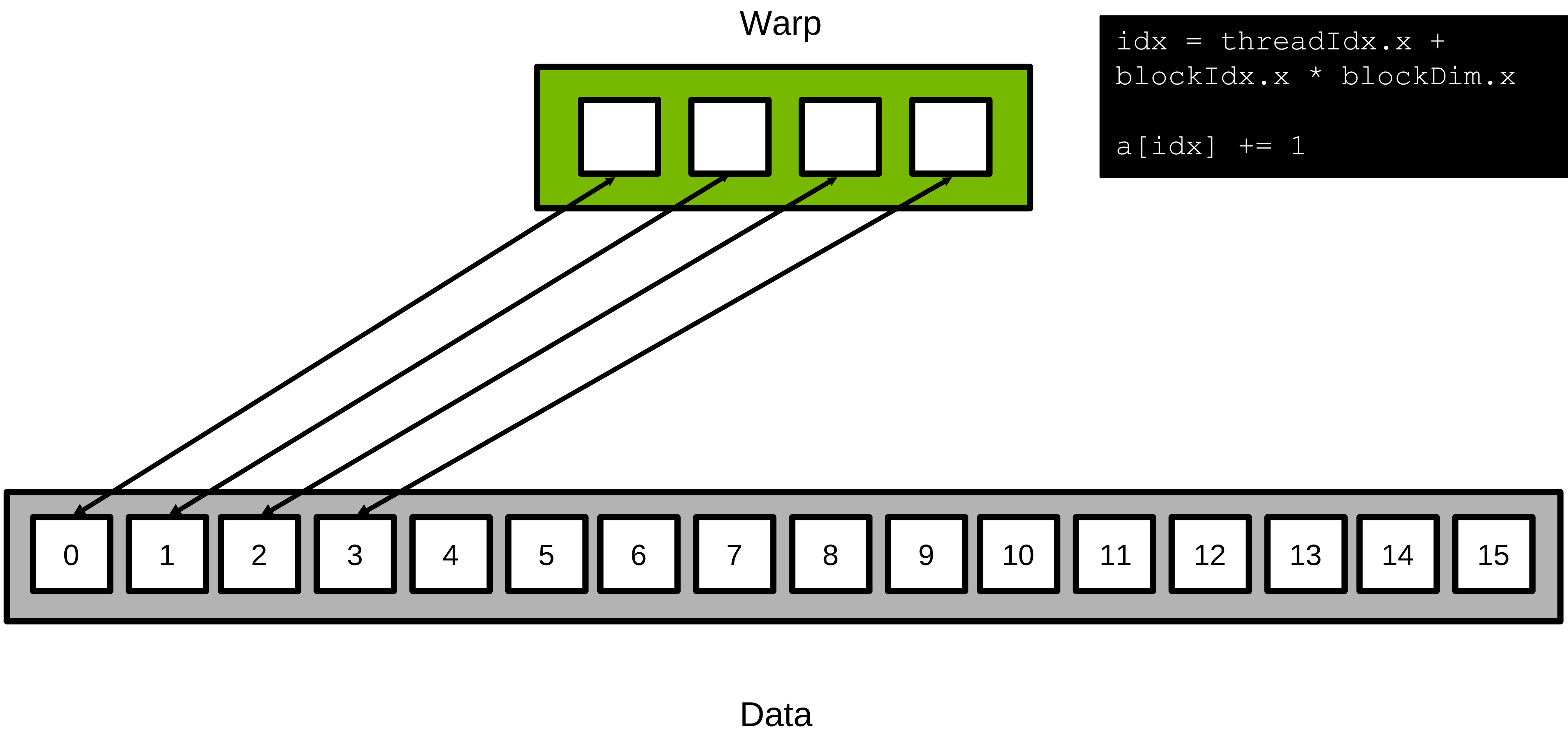


The memory subsystem will attempt to minimize the number of lines required to fulfill the read/write requirements of the warp



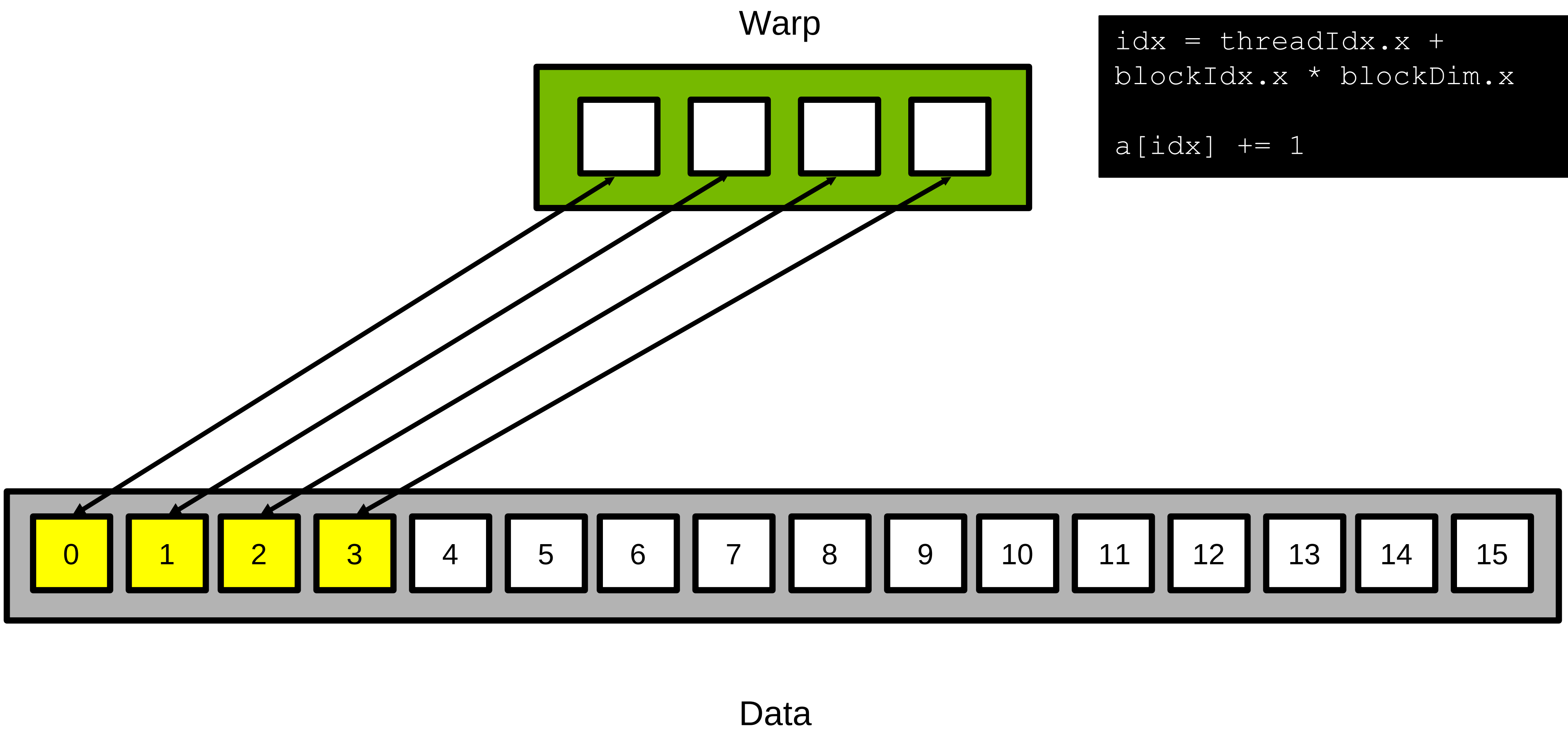
If the addresses requested are contiguous

```
idx = threadIdx.x +
blockIdx.x * blockDim.x

a[idx] += 1
```



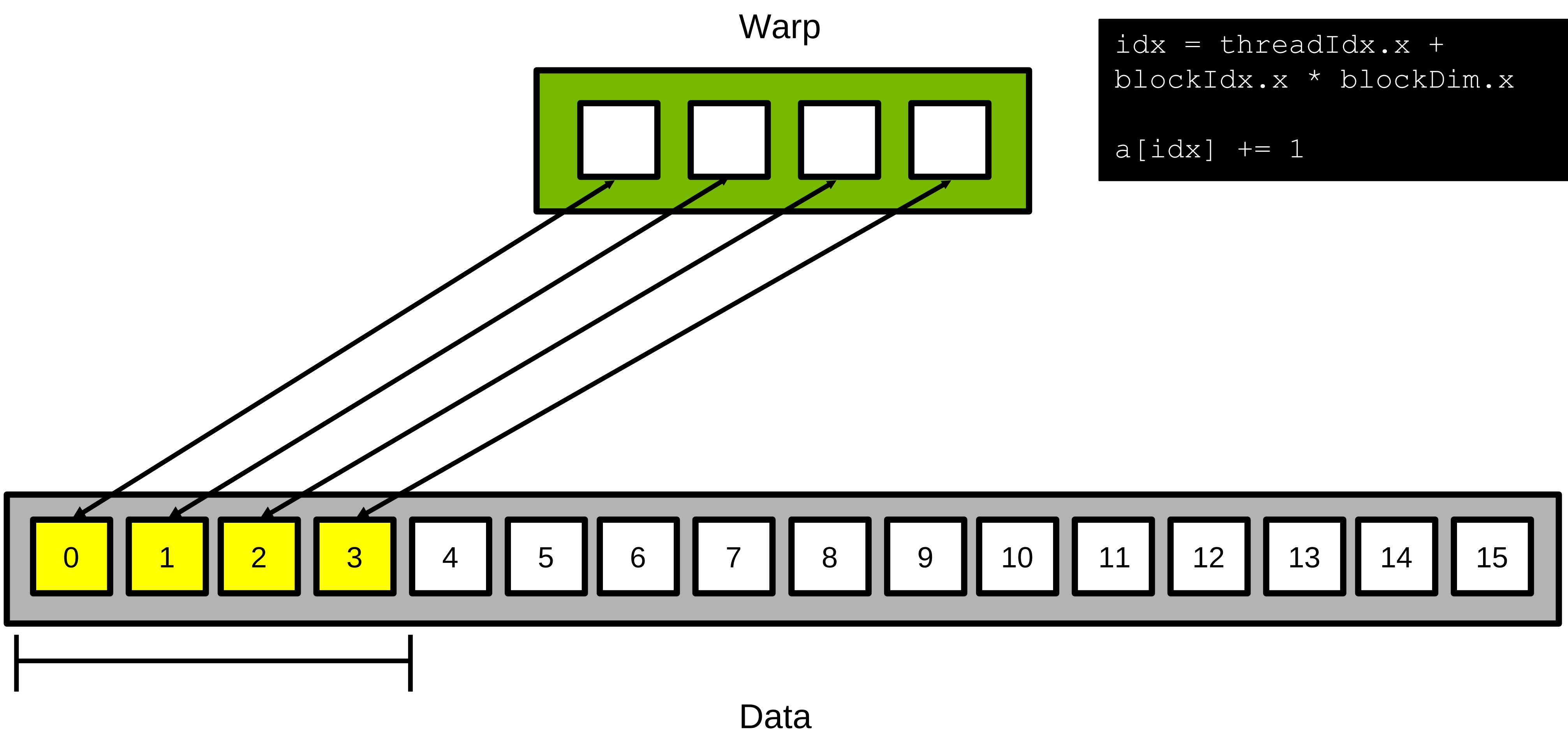
All data in the line will be used

```
idx = threadIdx.x +
blockIdx.x * blockDim.x

a[idx] += 1
```



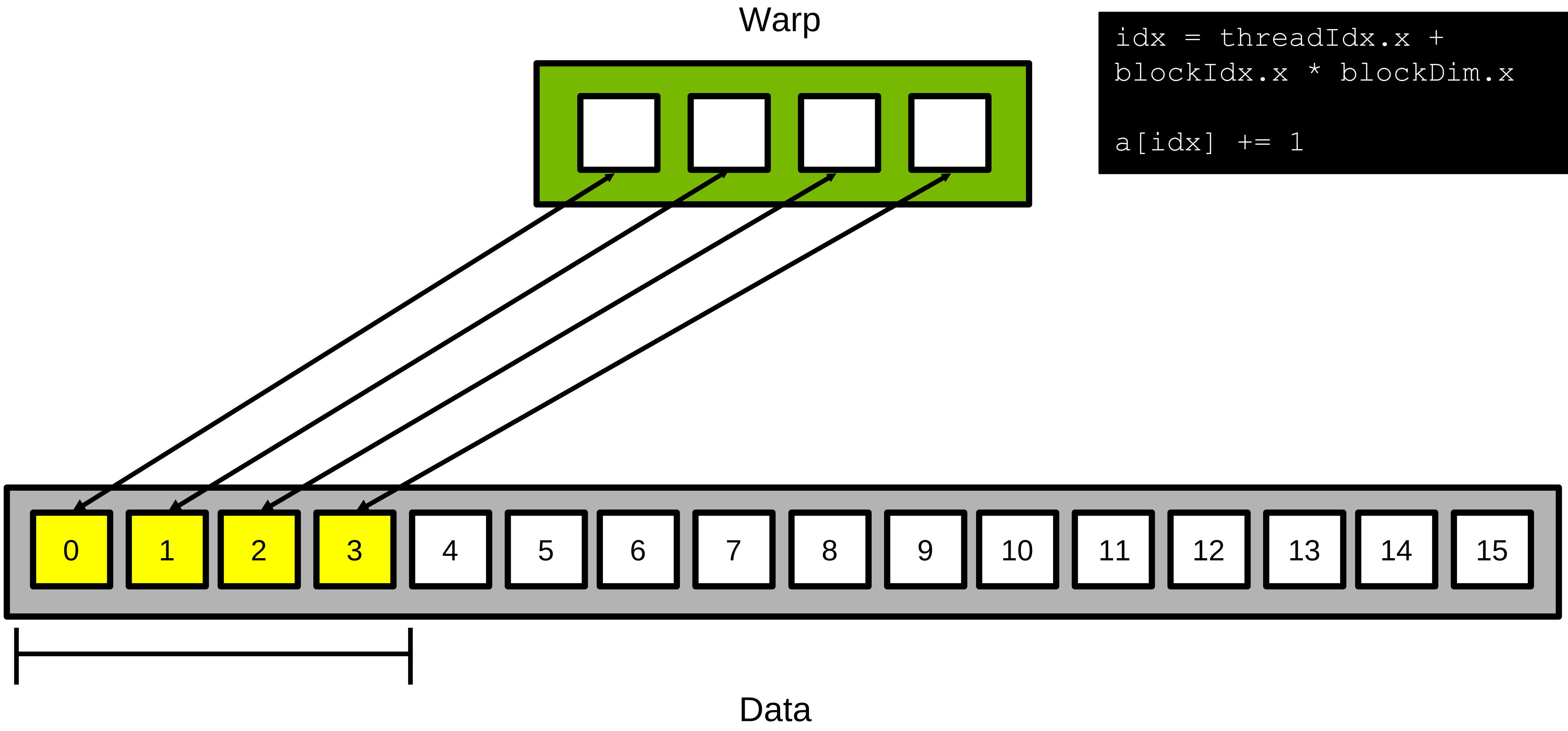
And the transfer will happen in as few lines as possible

```
idx = threadIdx.x +
blockIdx.x * blockDim.x

a[idx] += 1
```



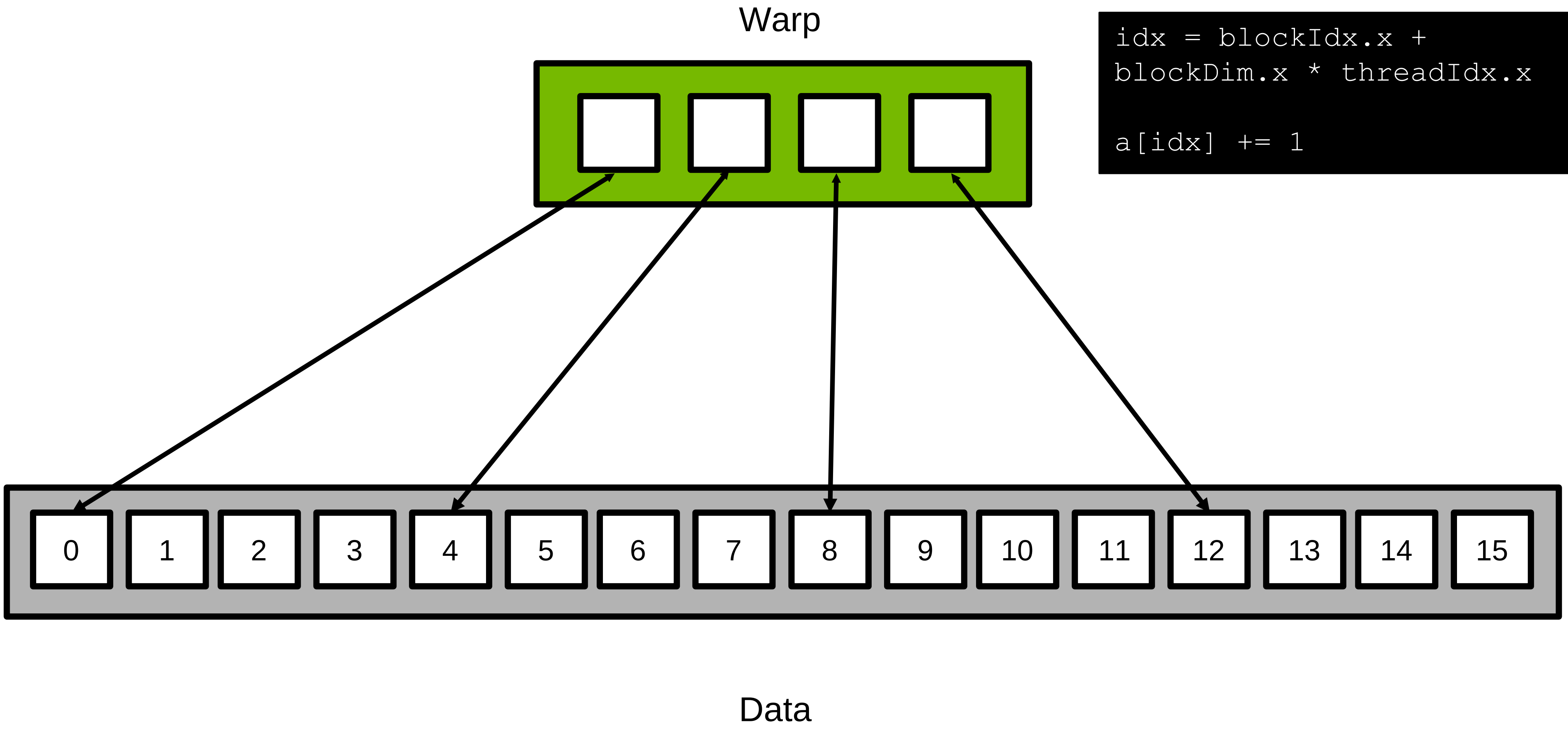
When this occurs, the memory access is fully **coalesced**

```
idx = threadIdx.x +
blockIdx.x * blockDim.x

a[idx] += 1
```



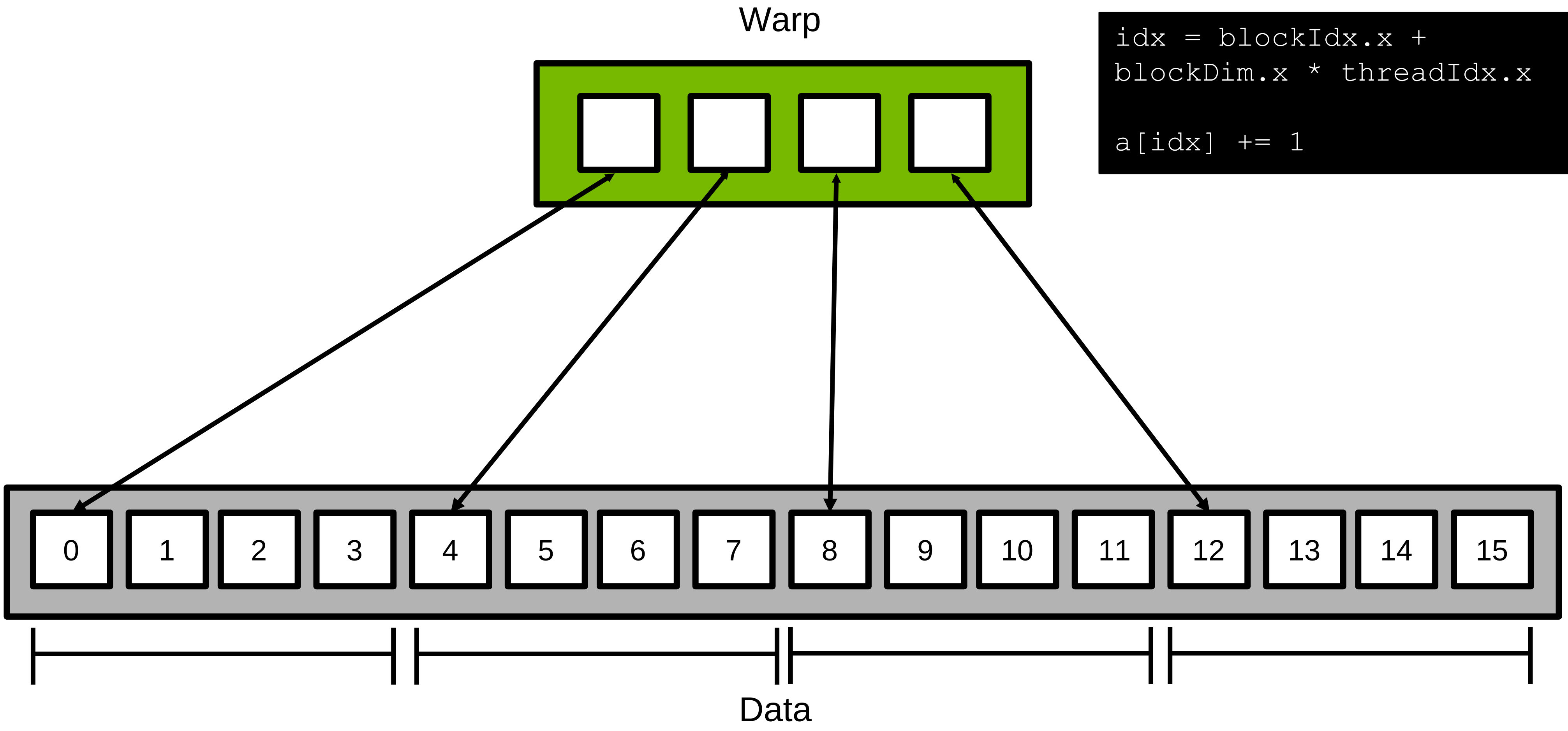
As requested memory becomes less contiguous

```
idx = blockIdx.x +
blockDim.x * threadIdx.x

a[idx] += 1
```

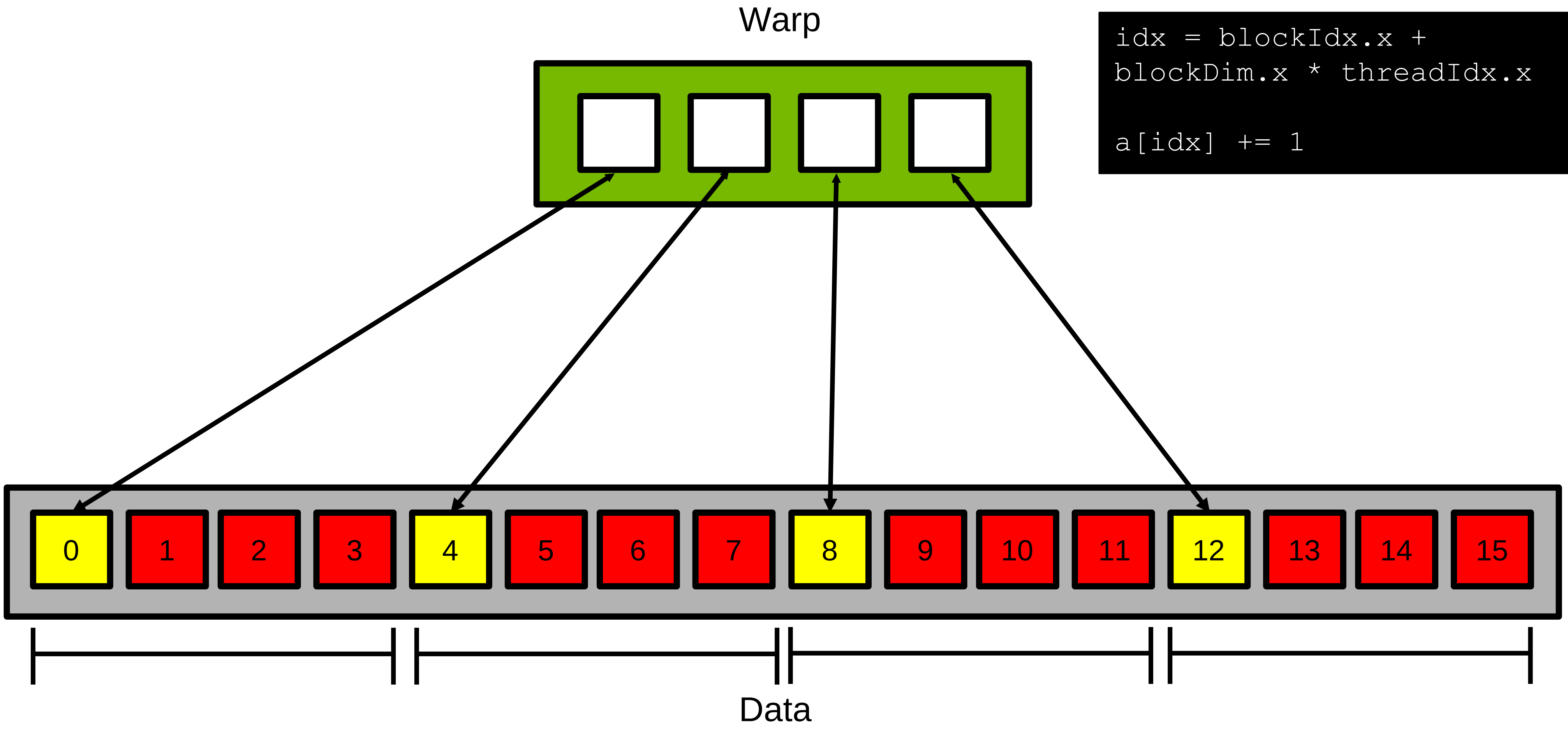


More lines will have to be transferred to fulfil the needs of the warp

```
idx = blockIdx.x +
blockDim.x * threadIdx.x

a[idx] += 1
```



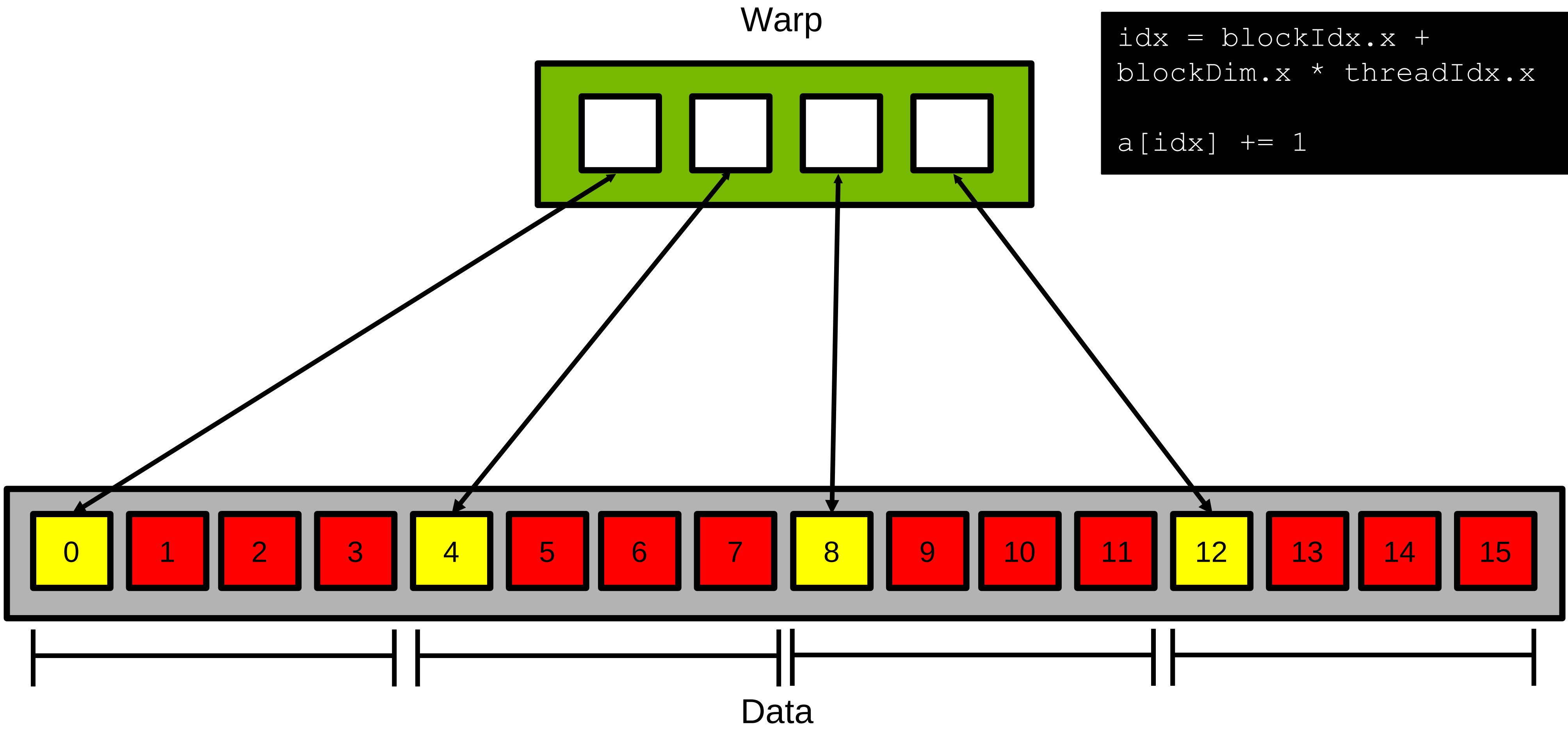
And more of the data being transferred will go unused




```
idx = blockIdx.x +
blockDim.x * threadIdx.x

a[idx] += 1
```

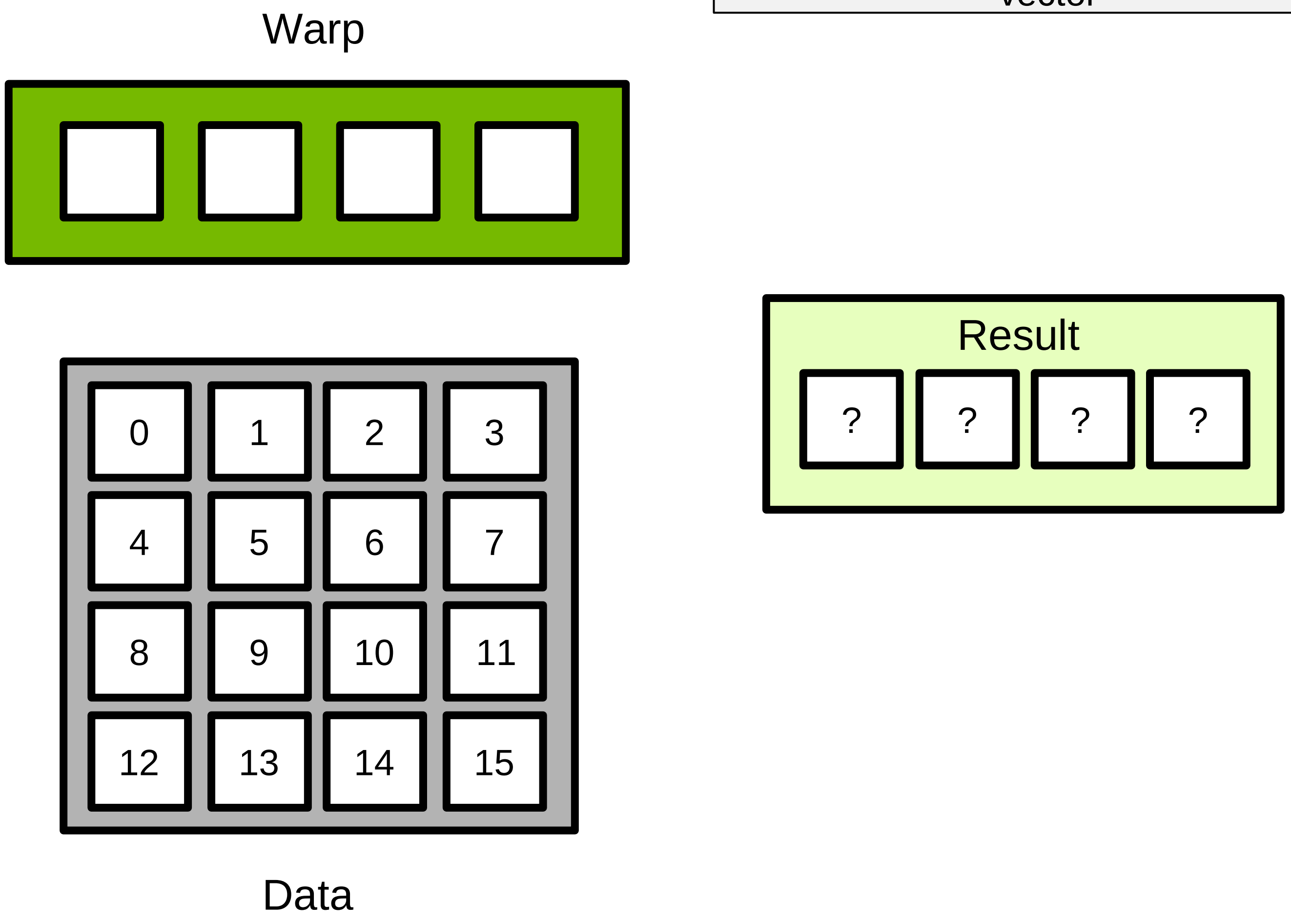
The memory throughput is degraded, and additional time is required: a performance loss

```
idx = blockIdx.x + blockDim.x * threadIdx.x

a[idx] += 1
```

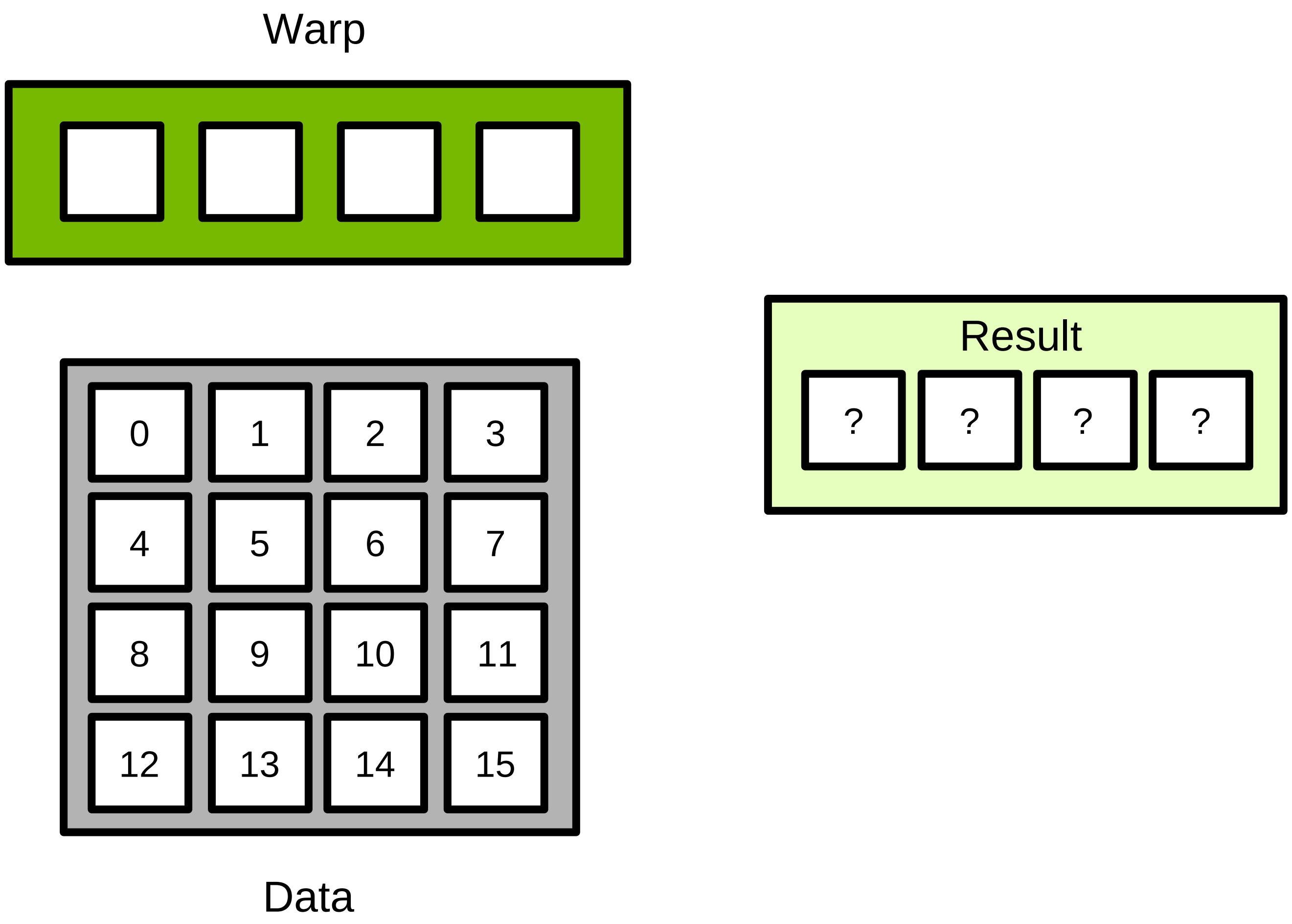


Row and Column Sum Comparison

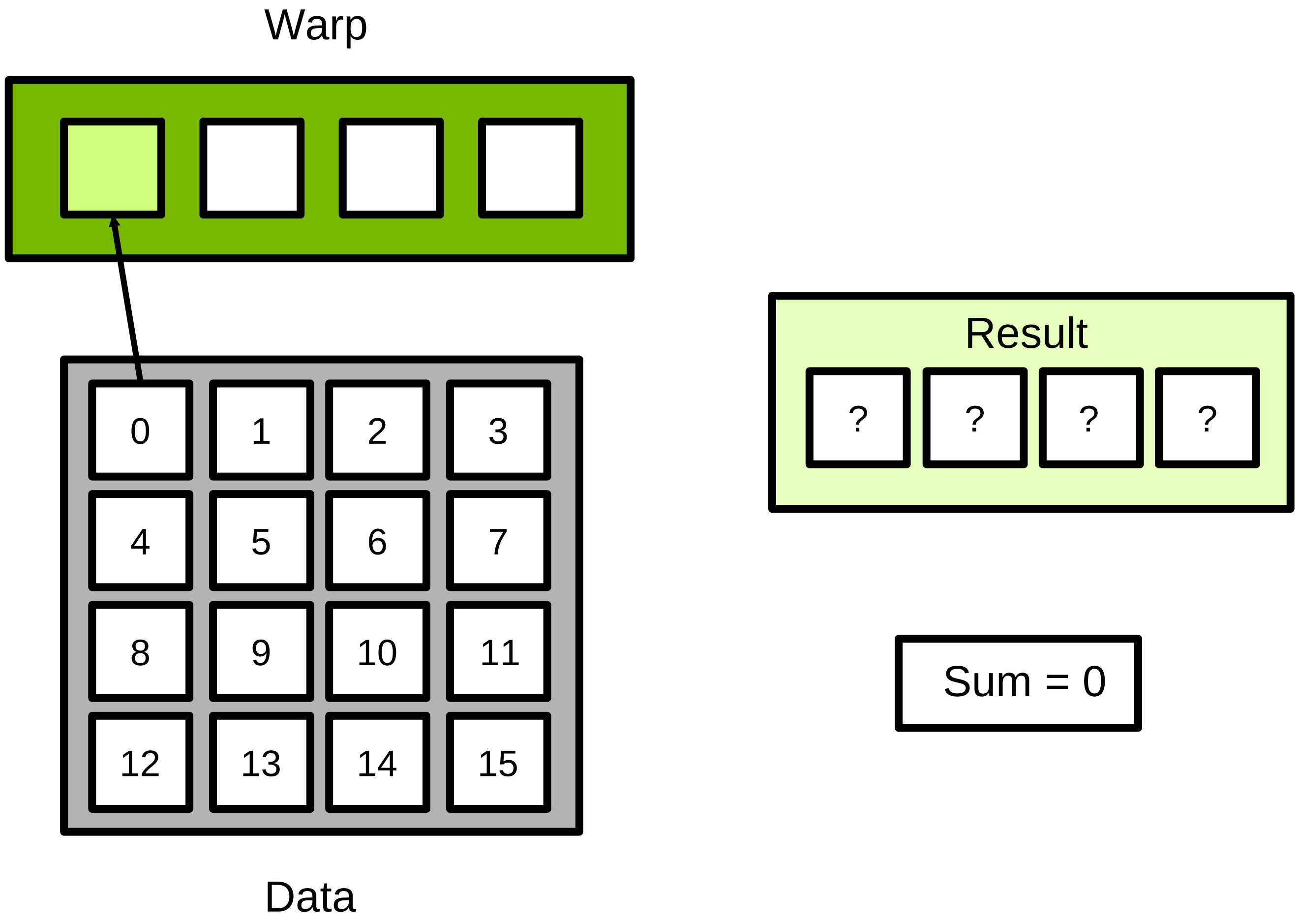
Consider a kernel that stores the sum of each row of a matrix (which here is 4 contiguous data elements) in a result vector



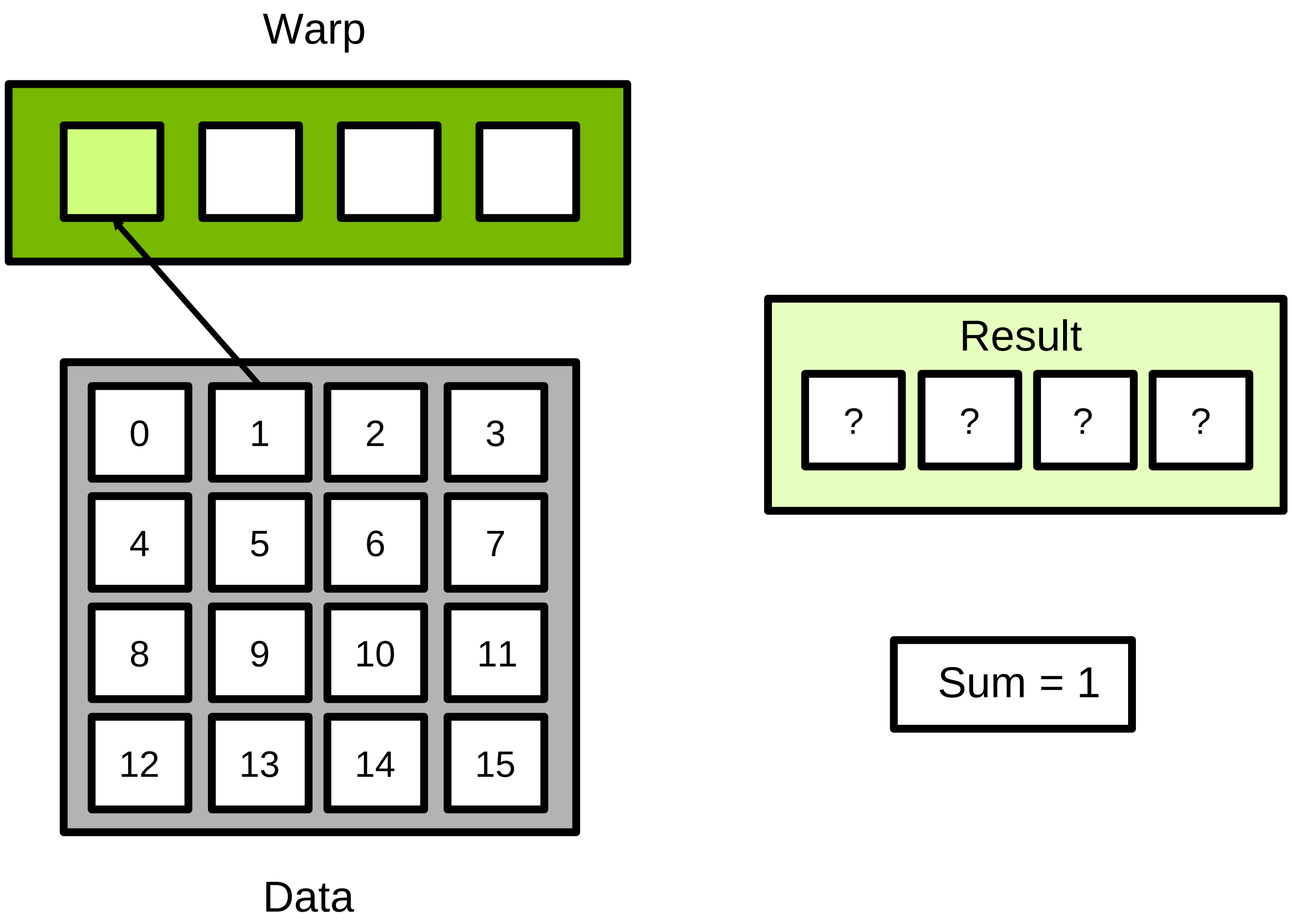
A single thread could iterate over a row, summing it, and then write the result in the solution vector



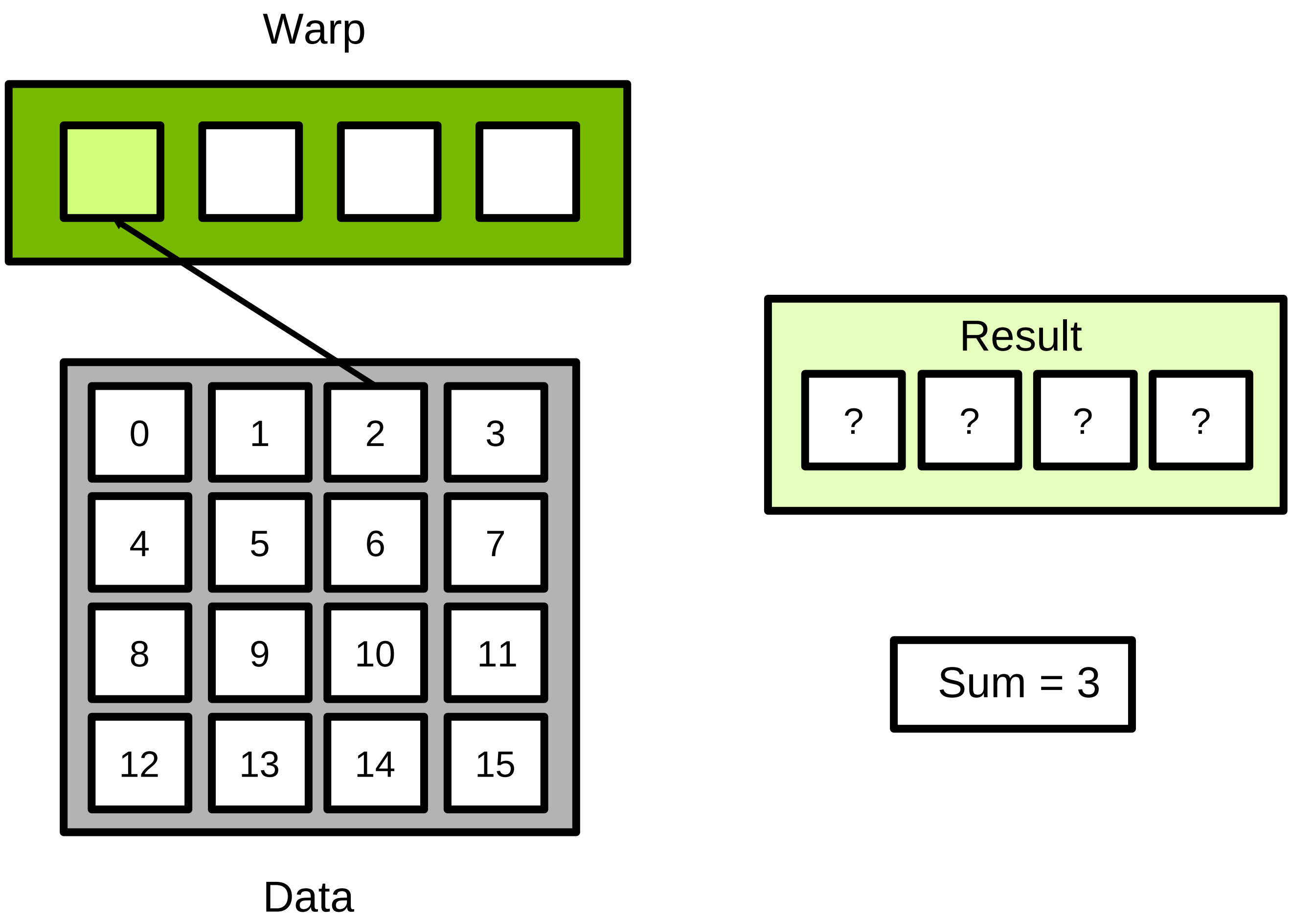
A single thread could iterate over a row, summing it, and then write the result in the solution vector



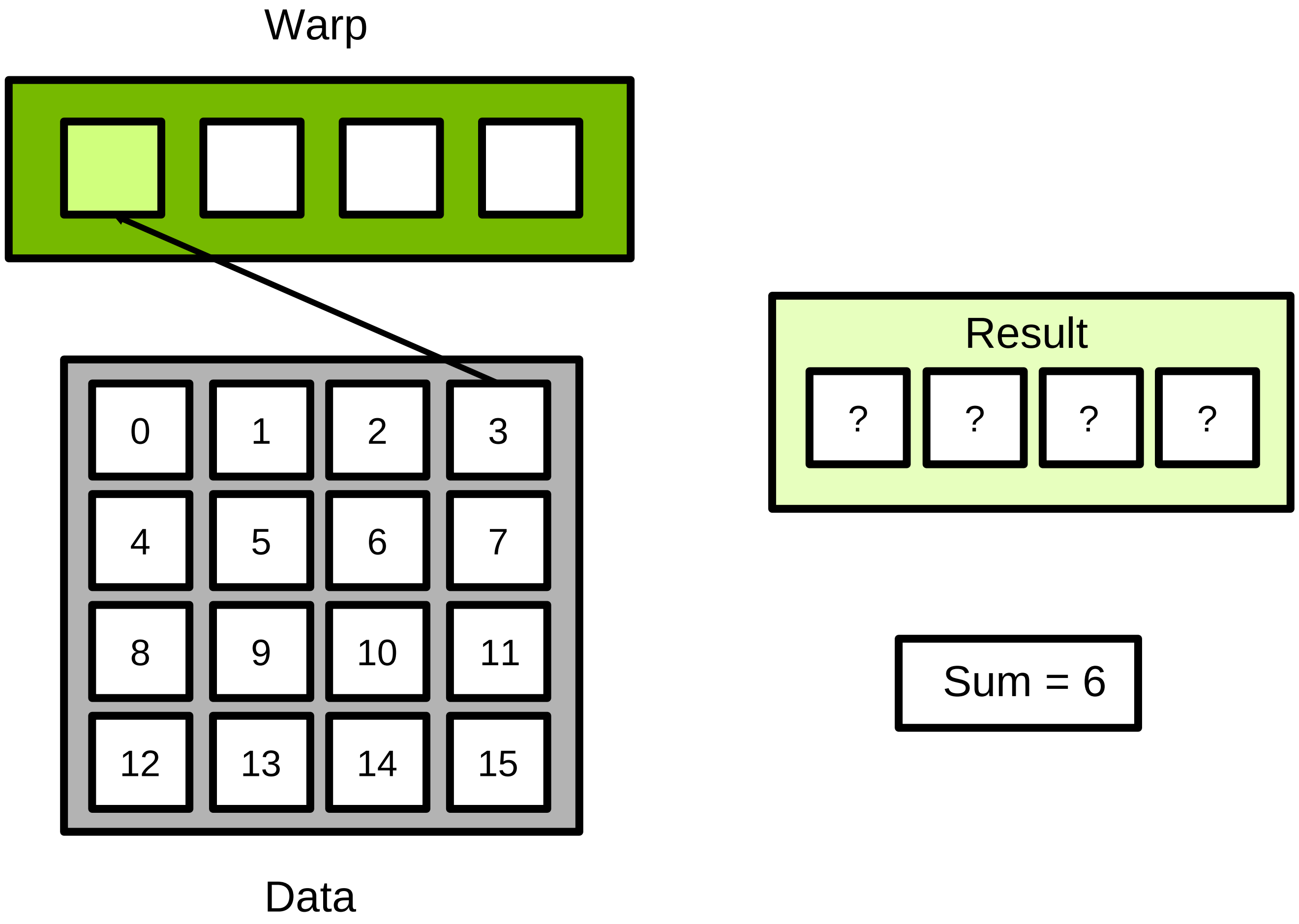
A single thread could iterate over a row, summing it, and then write the result in the solution vector



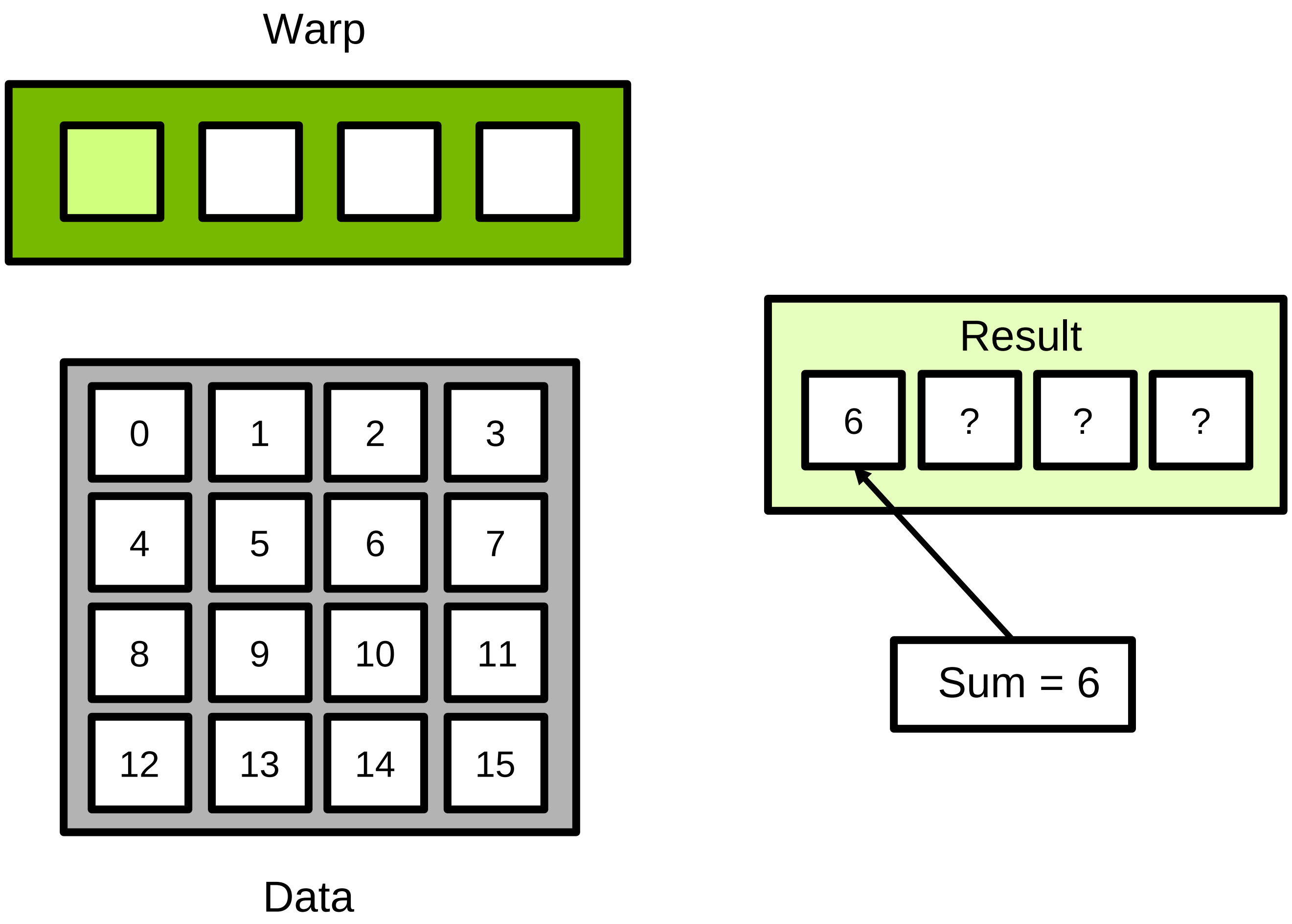
A single thread could iterate over a row, summing it, and then write the result in the solution vector



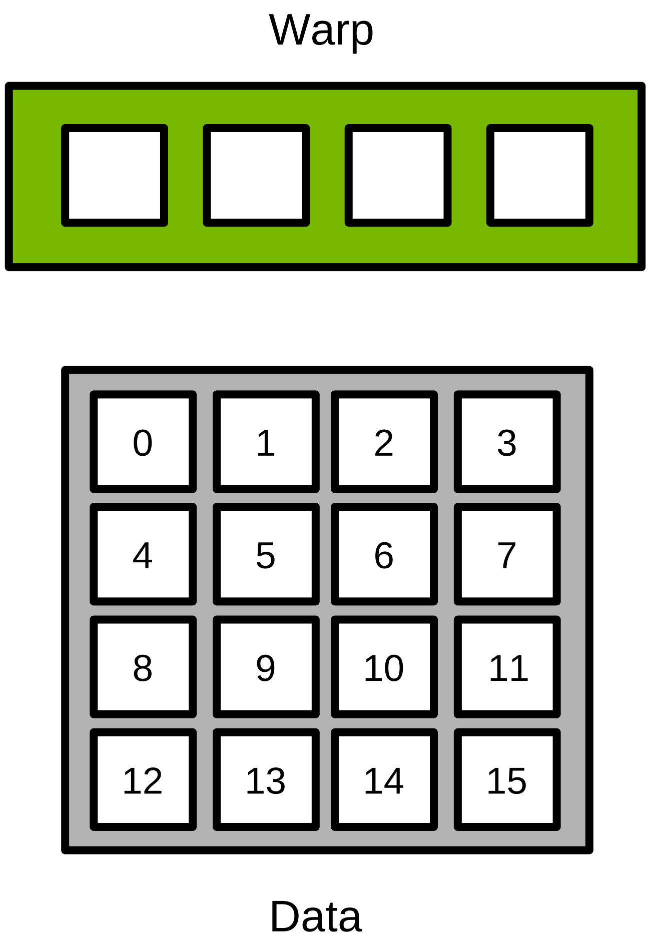
A single thread could iterate over a row, summing it, and then write the result in the solution vector



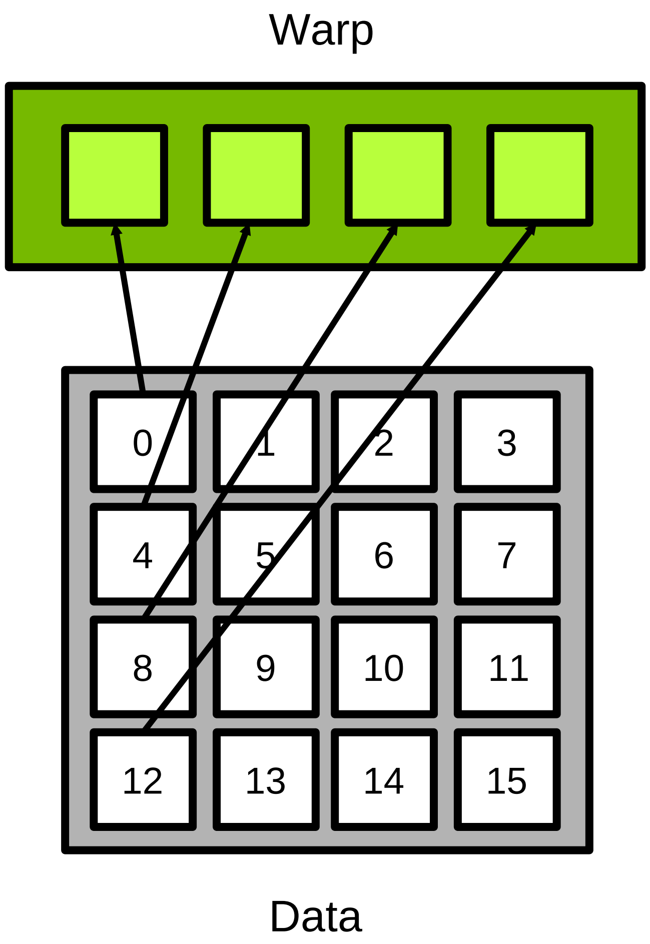
A single thread could iterate over a row, summing it, and then write the result in the solution vector

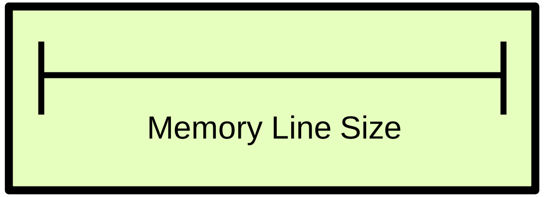


This seems natural, but look at what happens when we consider the parallel execution within the warp

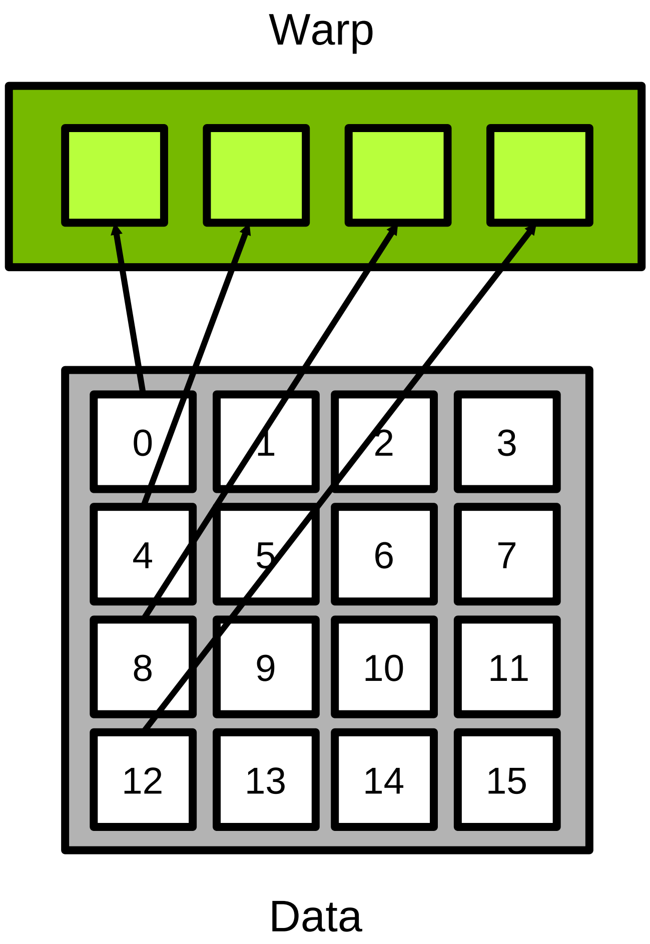


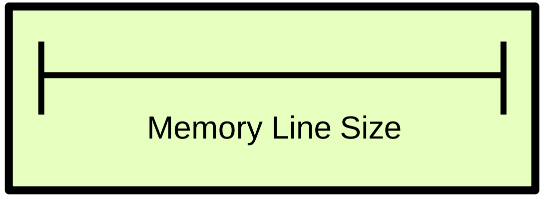
Each thread in the warp is requesting data in a different line of memory



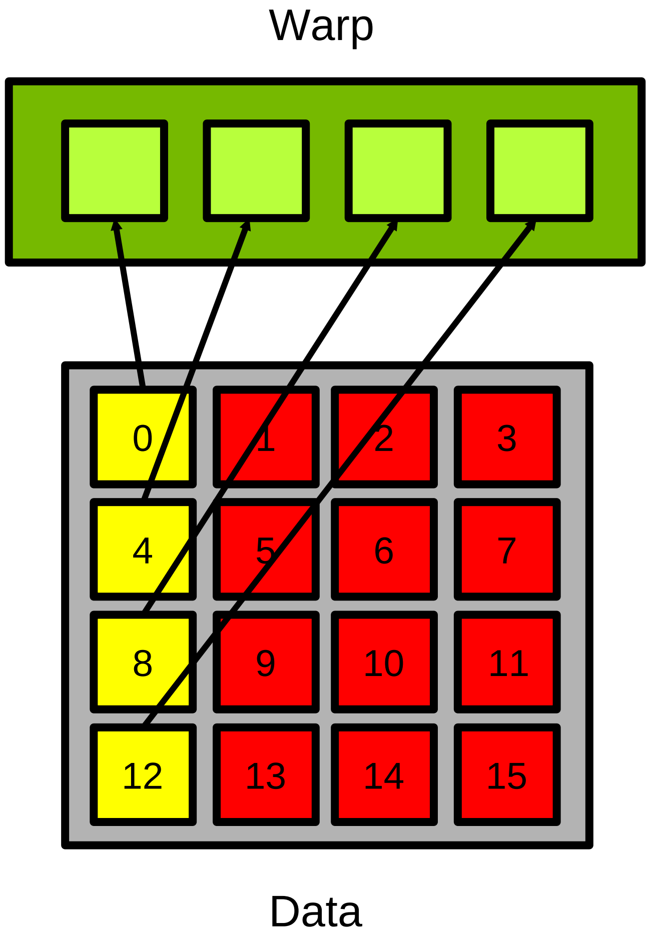


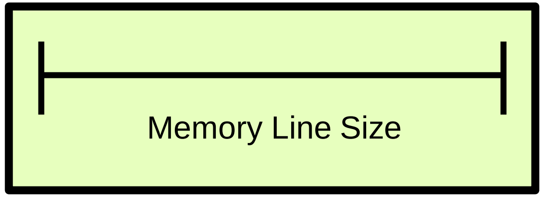
Note that increments to threadIdx.x are mapping to increments in the data along the y axis





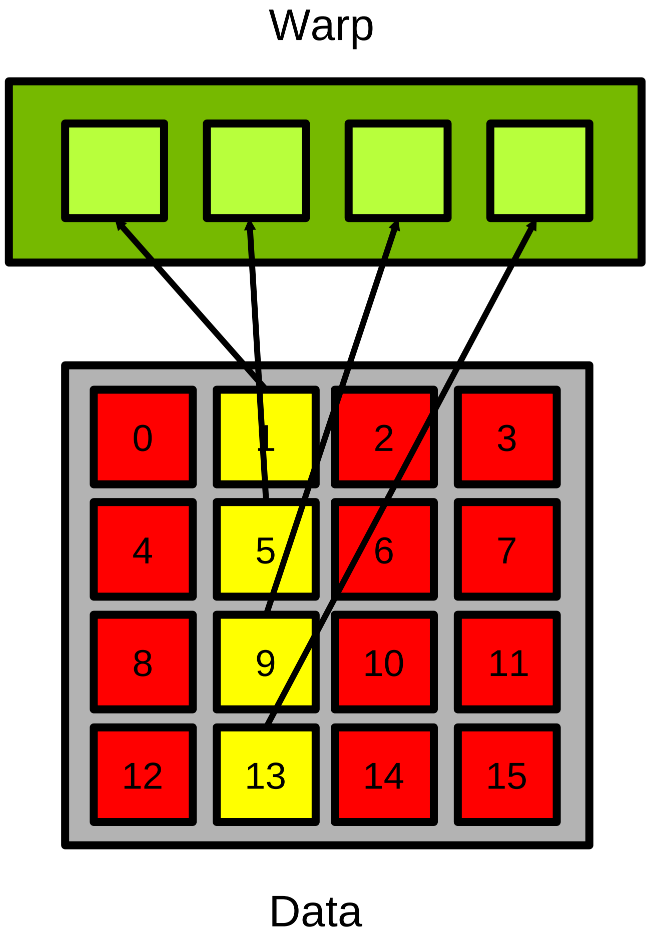
Which means (in our example) 4 lines of data will need to be loaded, and 75% of the data loaded will be unused

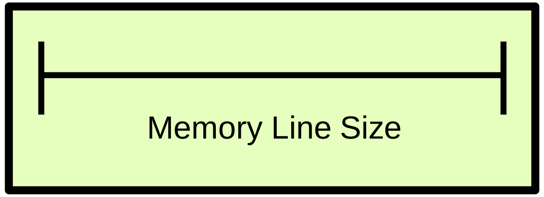




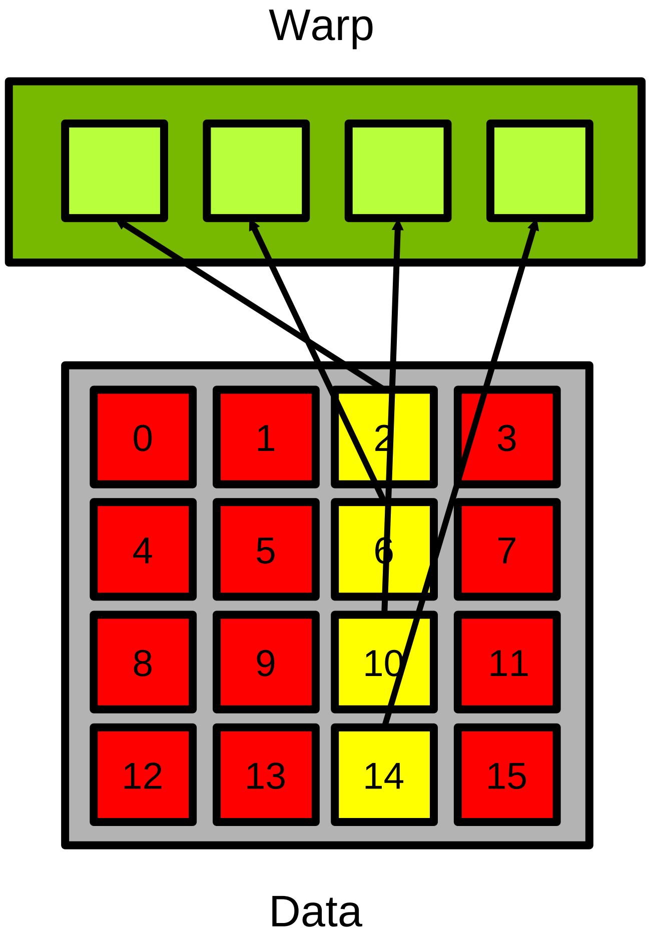
DEEP LEARNING INSTITUTE

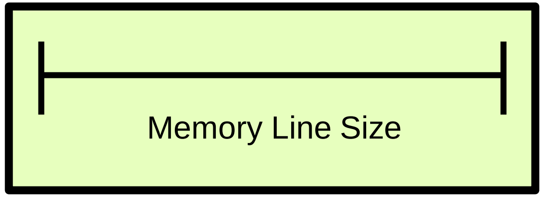
Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues



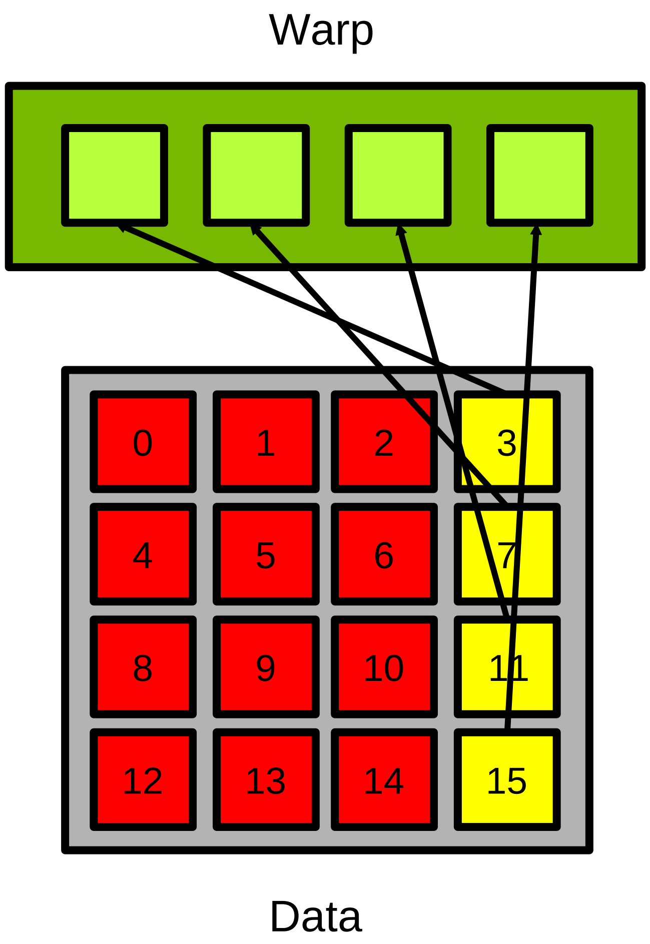


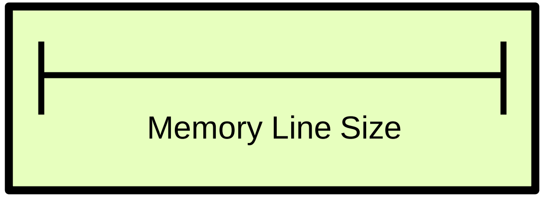
Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues



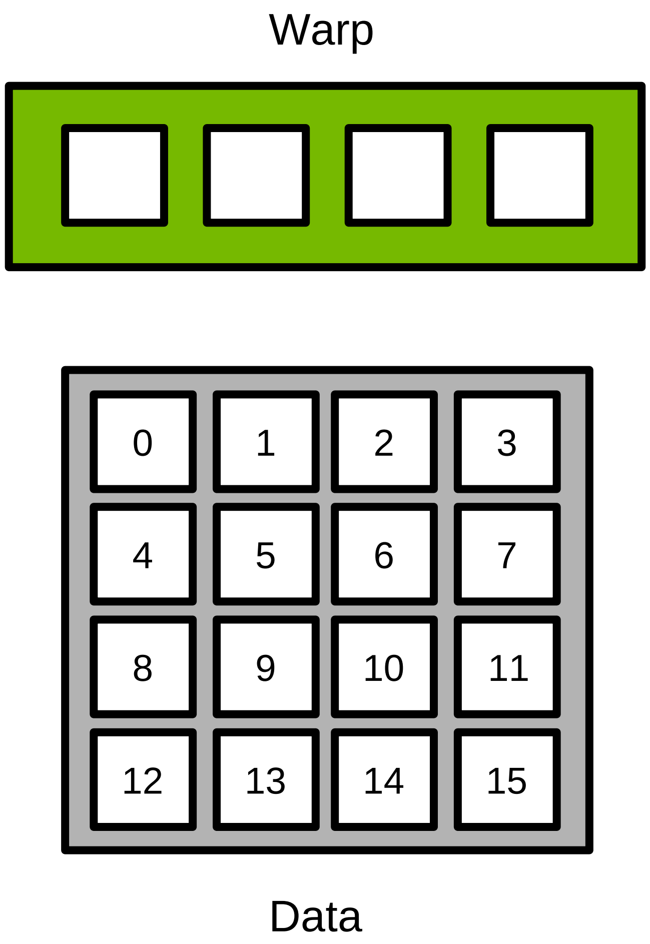


Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues

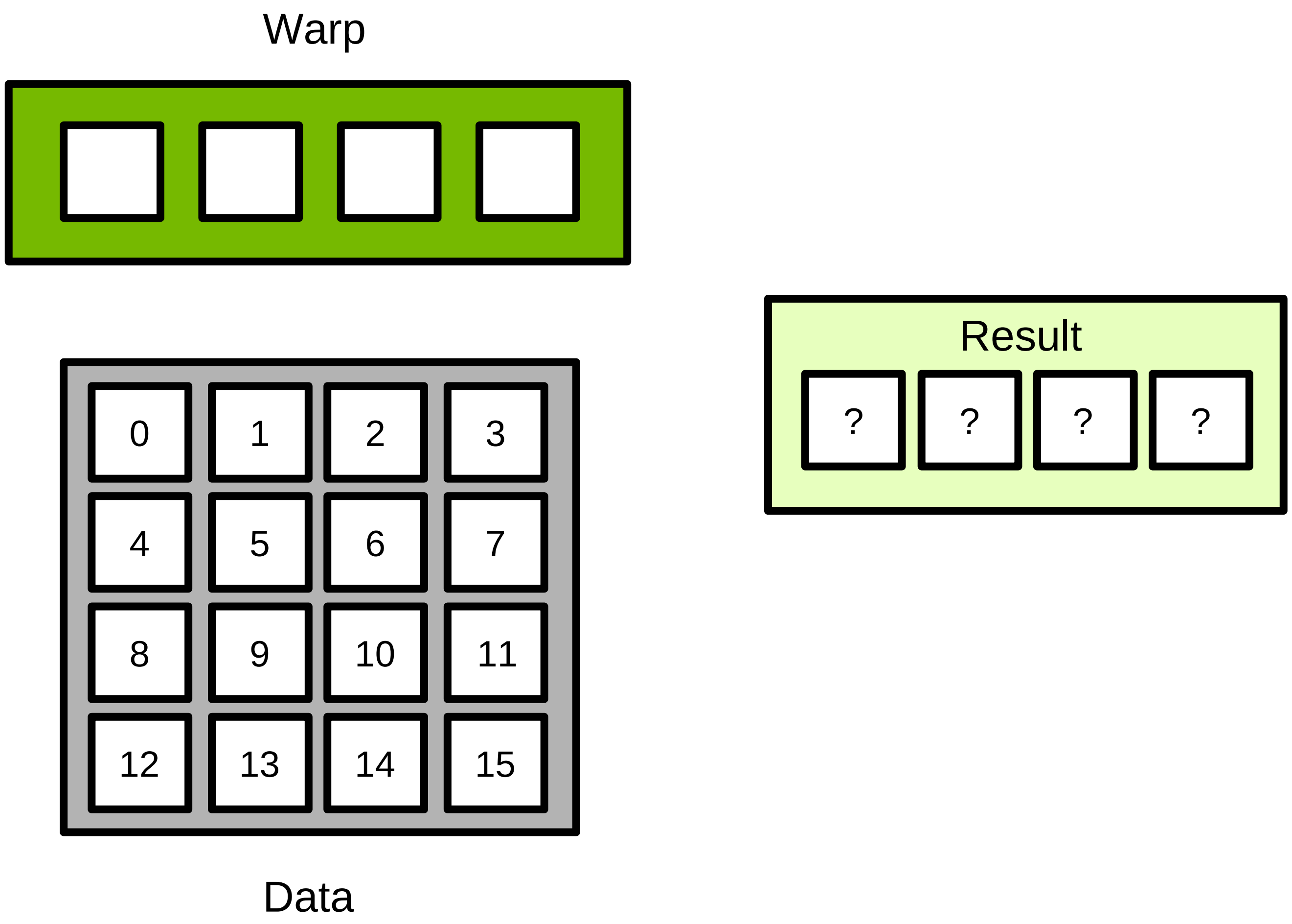




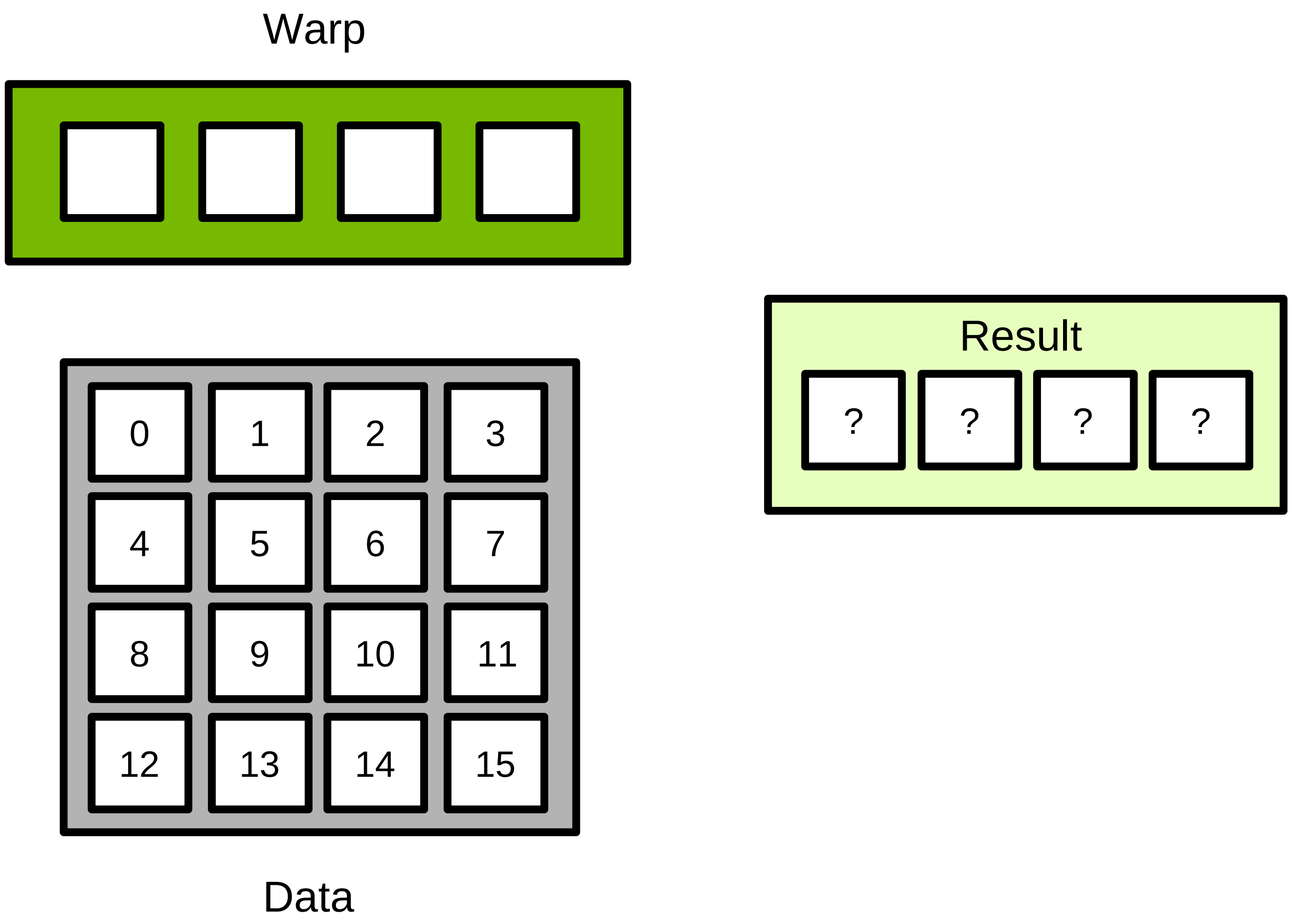
In this example we transferred 16 memory lines, and used 25% of the data for each line transferred



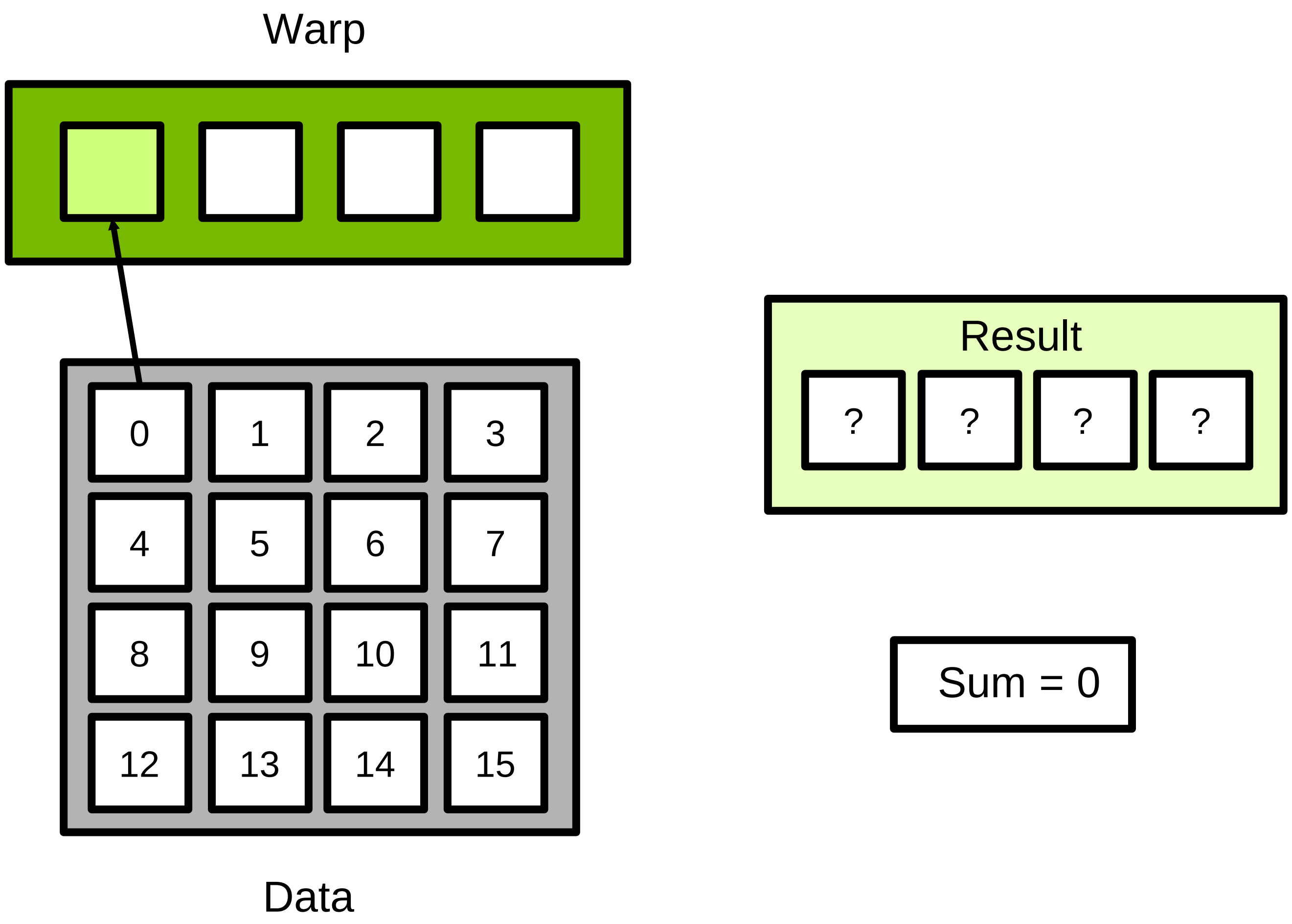
Let’s compare a kernel that stores the sum of each **column** of a matrix in a result vector



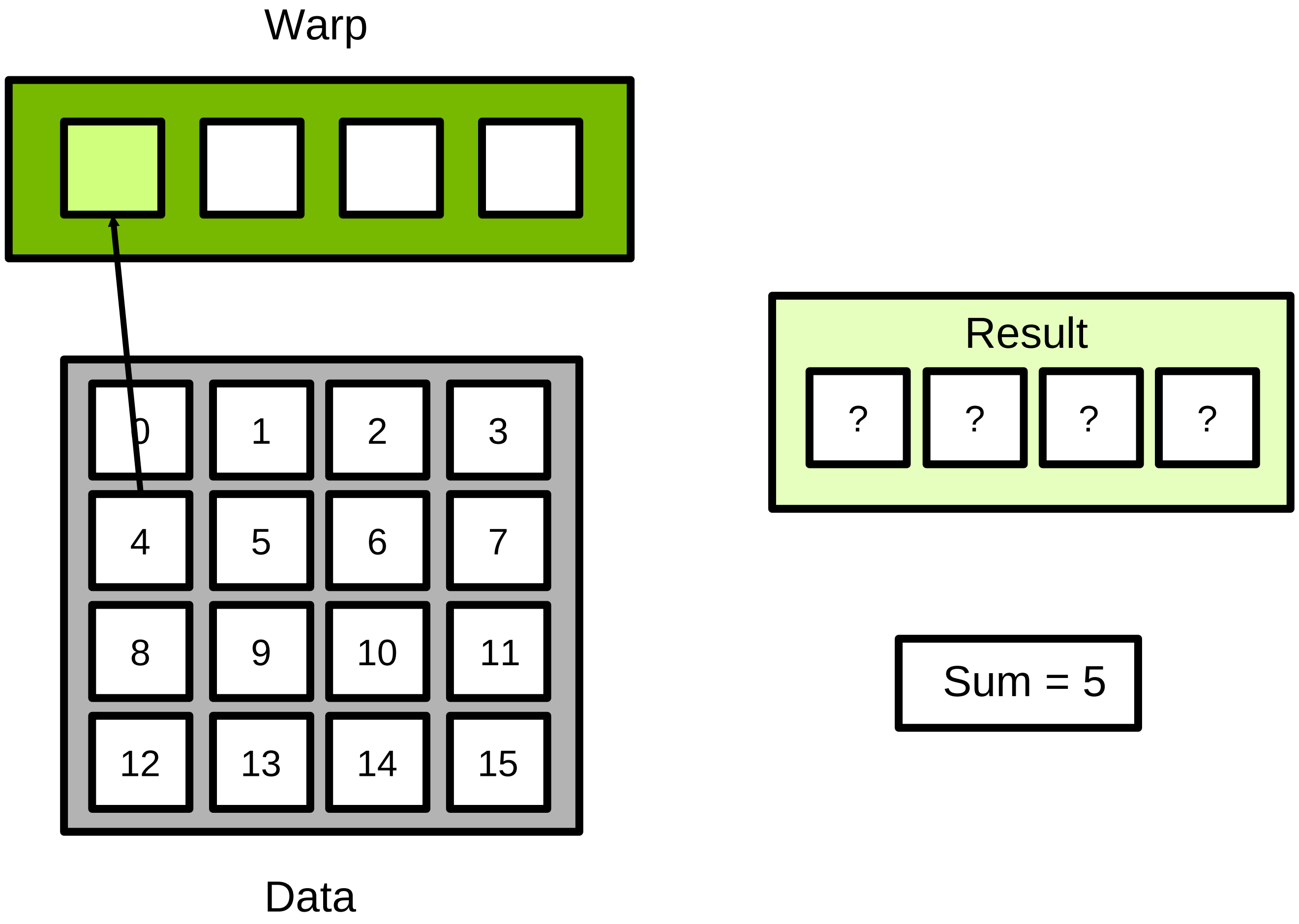
A single thread could iterate over a column, summing it, and then write the result in the solution vector



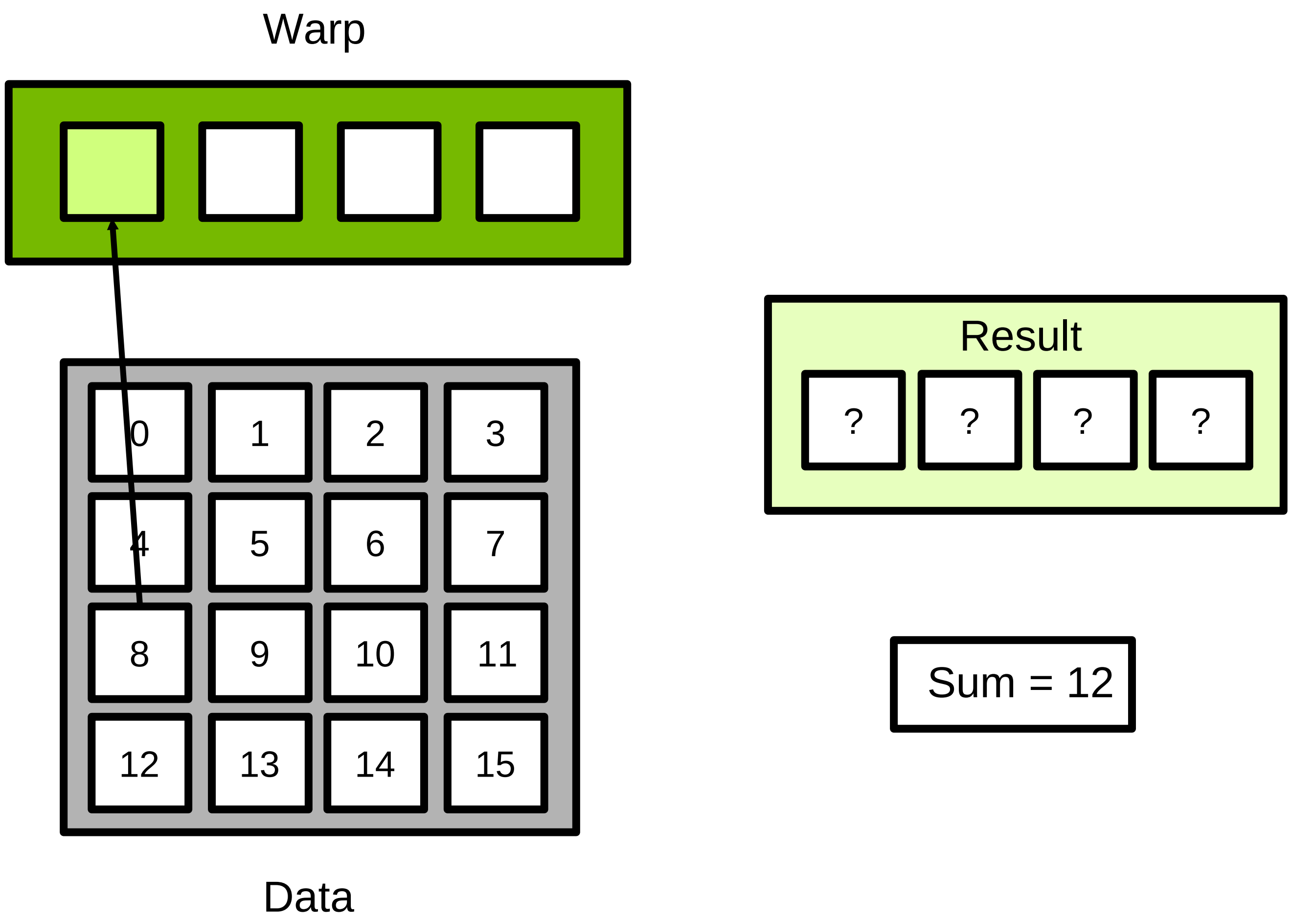
A single thread could iterate over a column, summing it, and then write the result in the solution vector



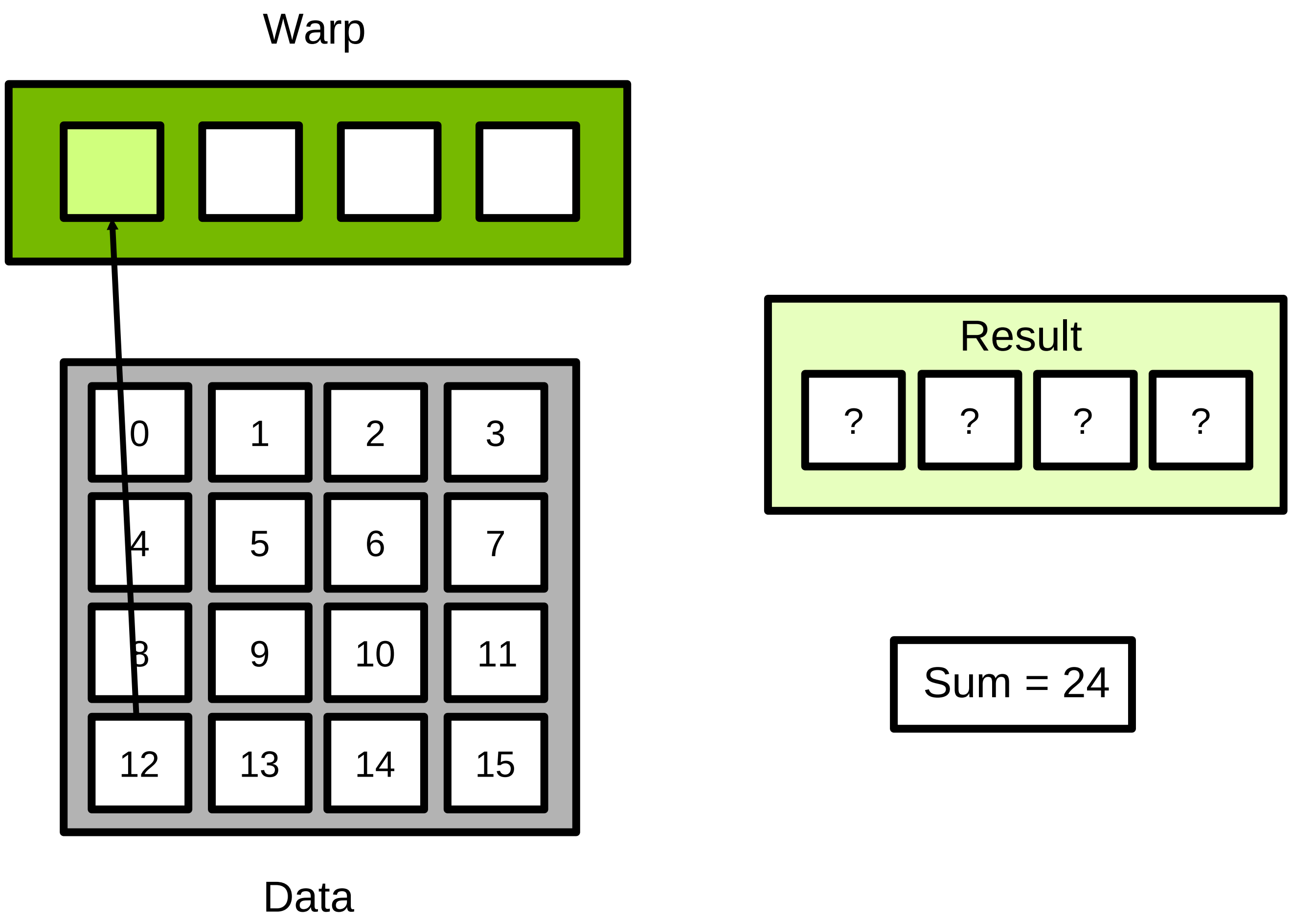
A single thread could iterate over a column, summing it, and then write the result in the solution vector



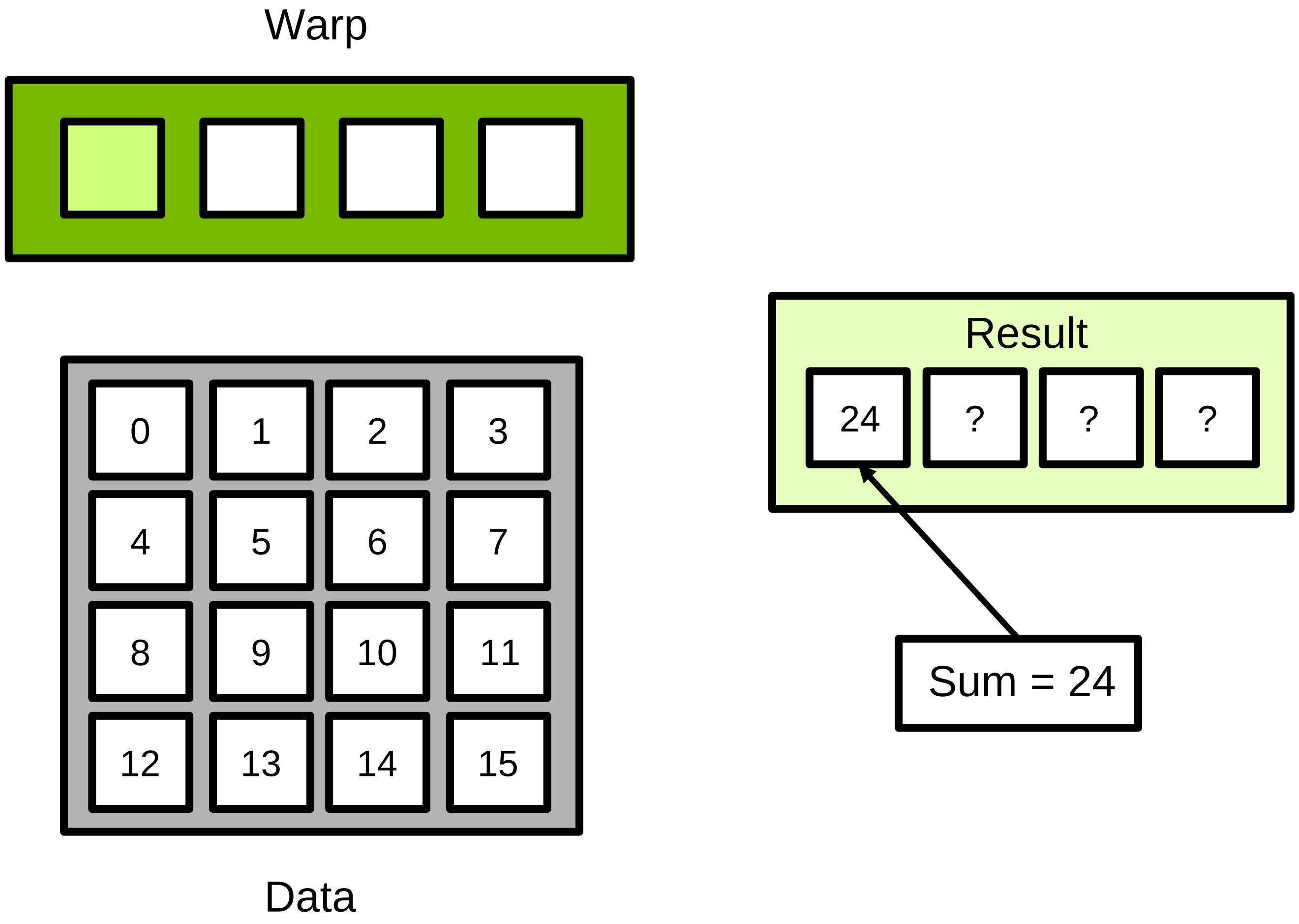
A single thread could iterate over a column, summing it, and then write the result in the solution vector



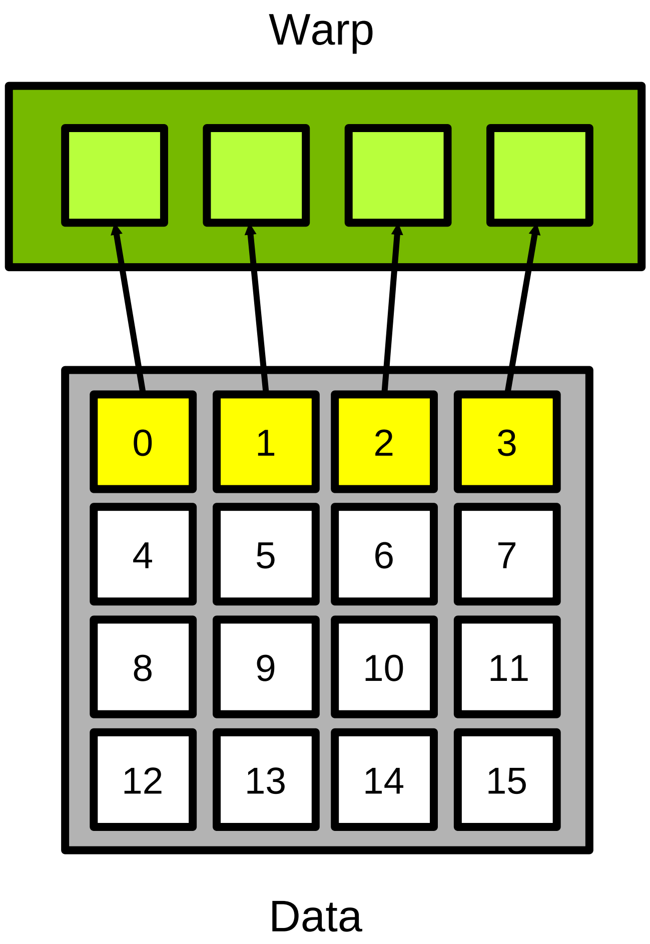
A single thread could iterate over a column, summing it, and then write the result in the solution vector

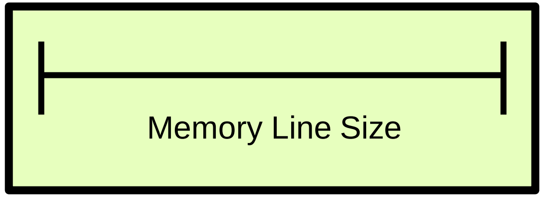


A single thread could iterate over a column, summing it, and then write the result in the solution vector



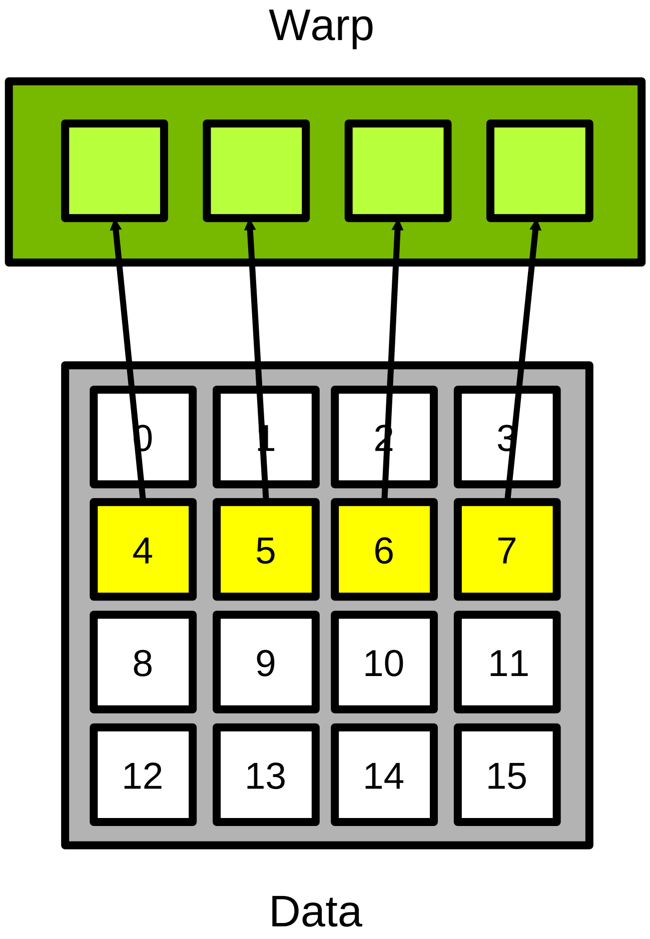
Here when we consider the parallel execution, we see that the warp's memory access is coalesced

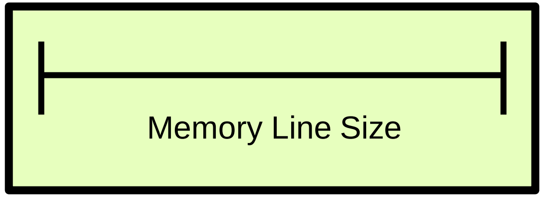




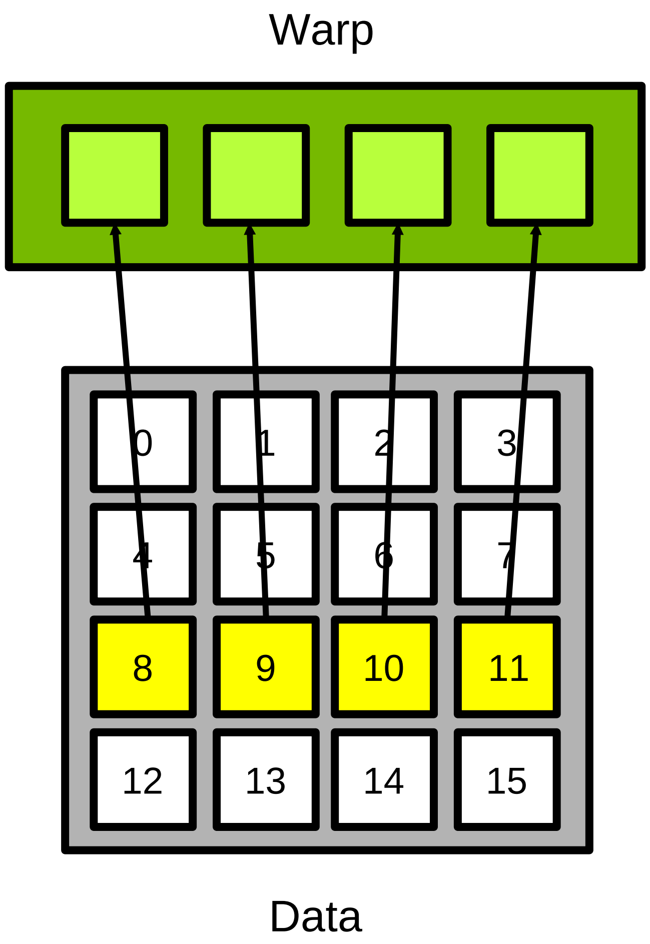
NVIDIA. DEEP LEARNING INSTITUTE

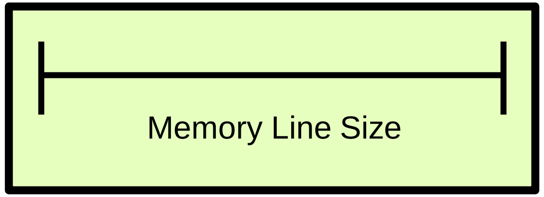
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



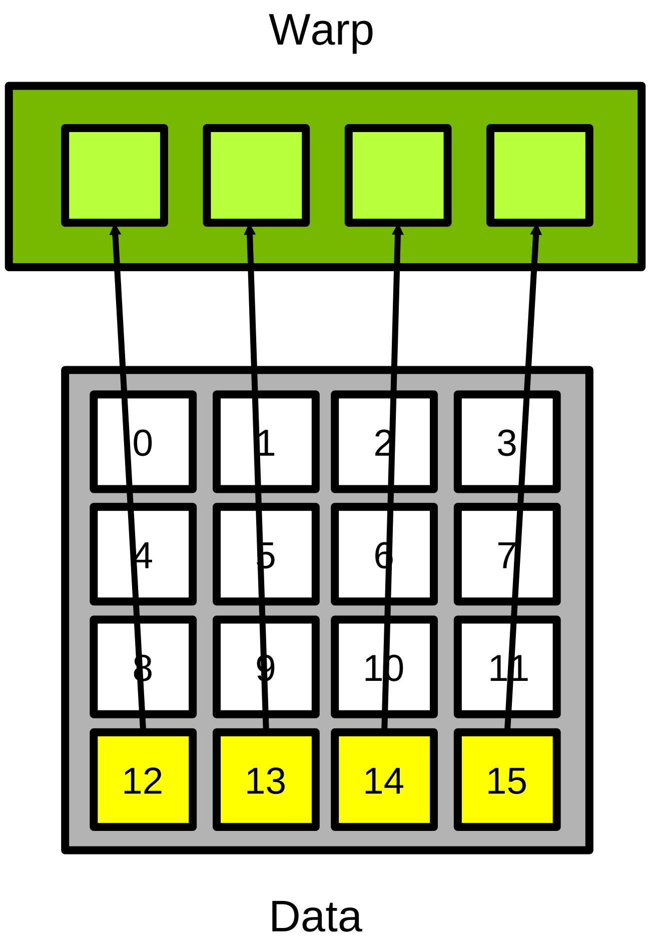


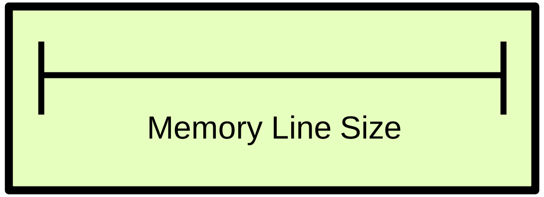
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



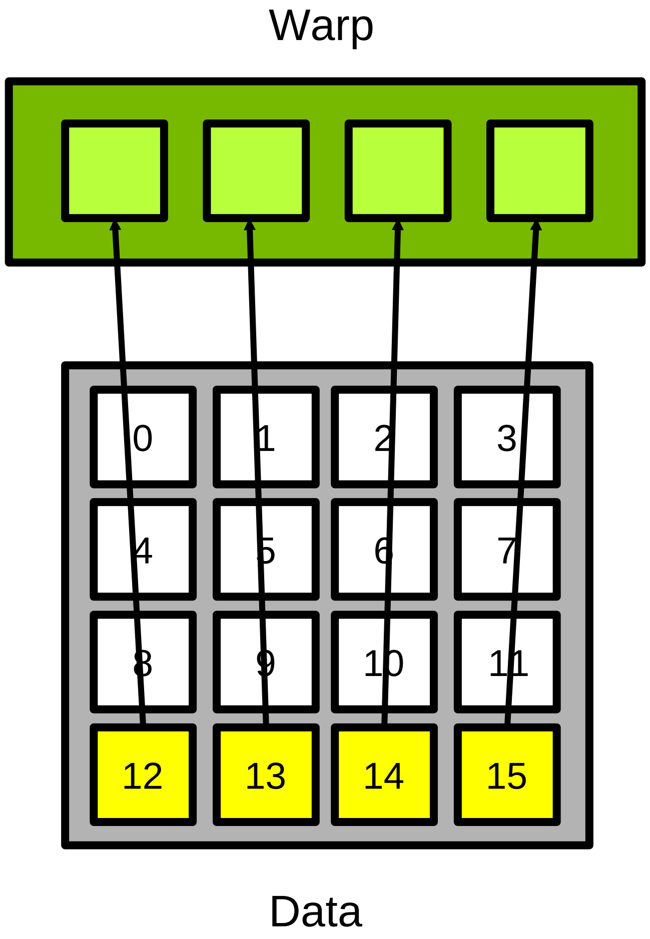


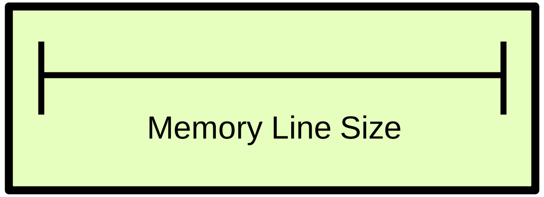
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



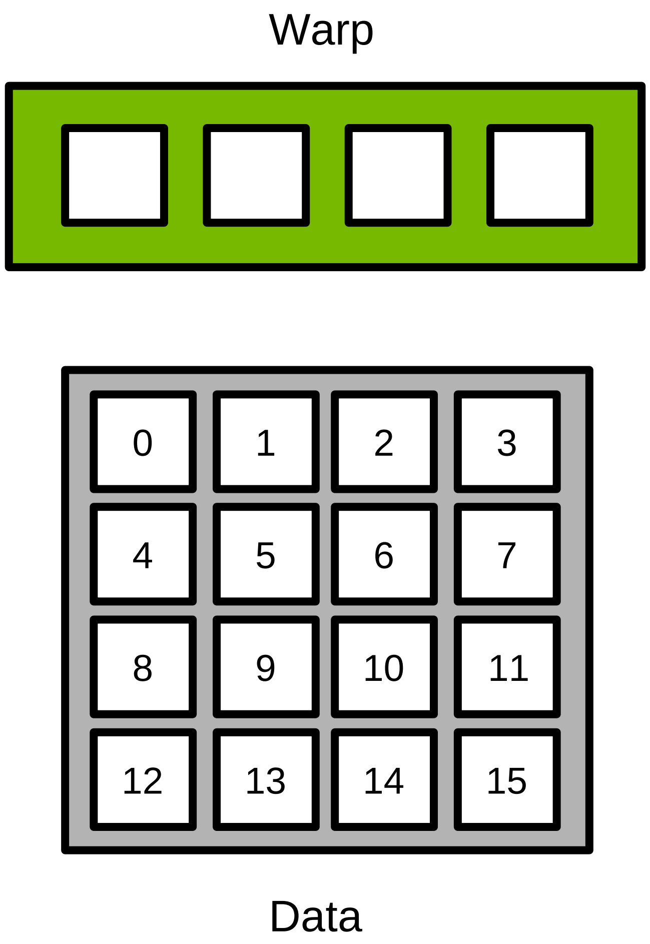


A useful tip to keep in mind is that increments to threadIdx.x should map to increments in data in the direction of fastest changing index – in this case the x axis





In this example we transferred 4 memory lines (compared to 16), and used 100% of the data for each line transferred (compared to 25%)

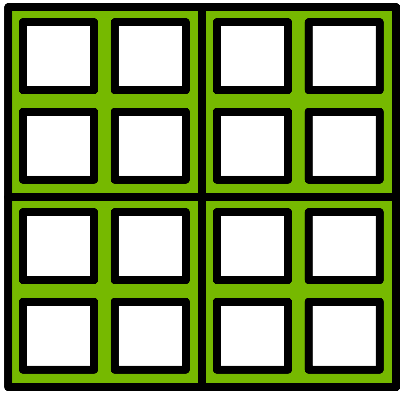


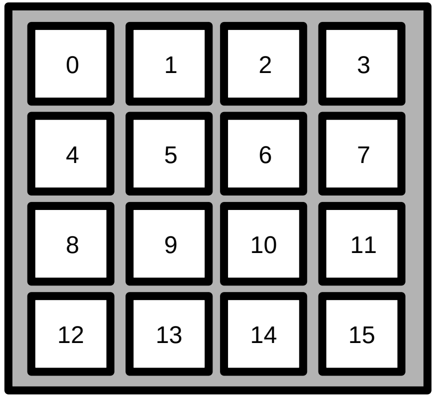
Using Shared Memory to Support Coalesced Memory Access

We will examine a matrix transpose to demonstrate how shared memory can be used to promote coalesced data transfers to and from global memory

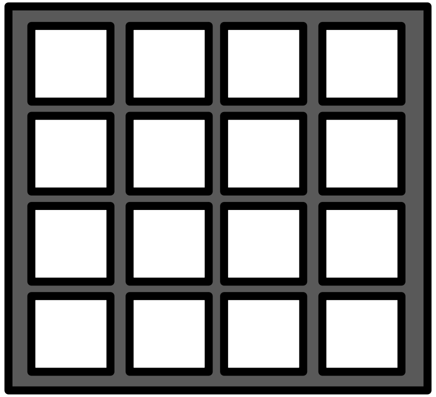
Here we have a (2,2) grid, with each block containing (2,2) threads as well as (4,4) input and output matrices

Grid

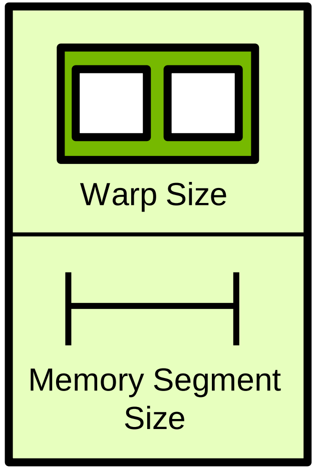


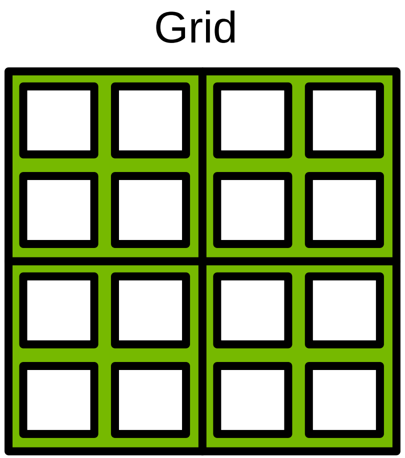


Input

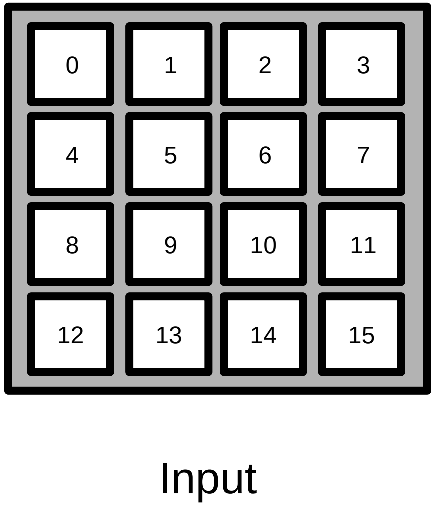


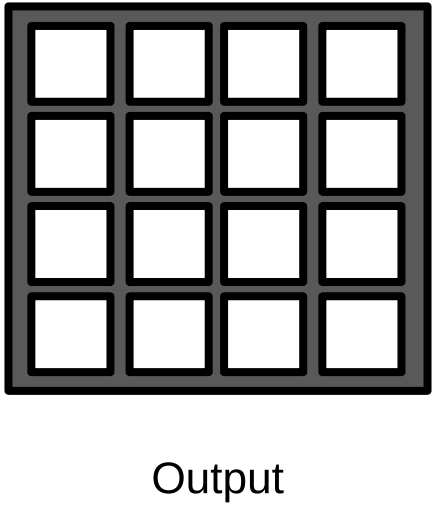
Output

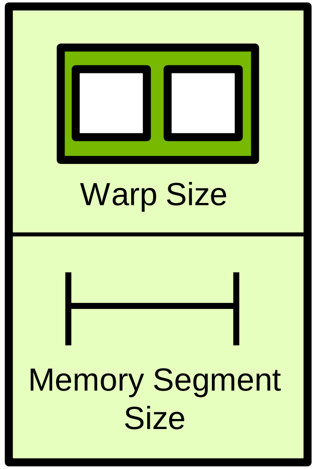


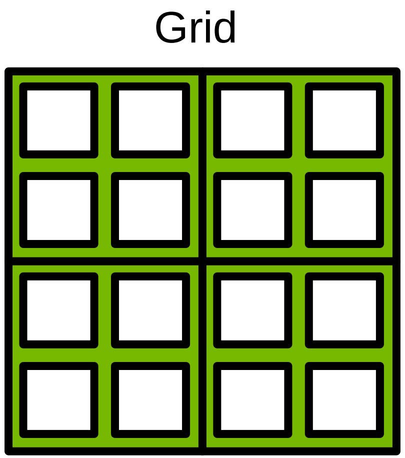


For these slides we will define a warp as 2 threads, and a memory segment as 2 data elements wide

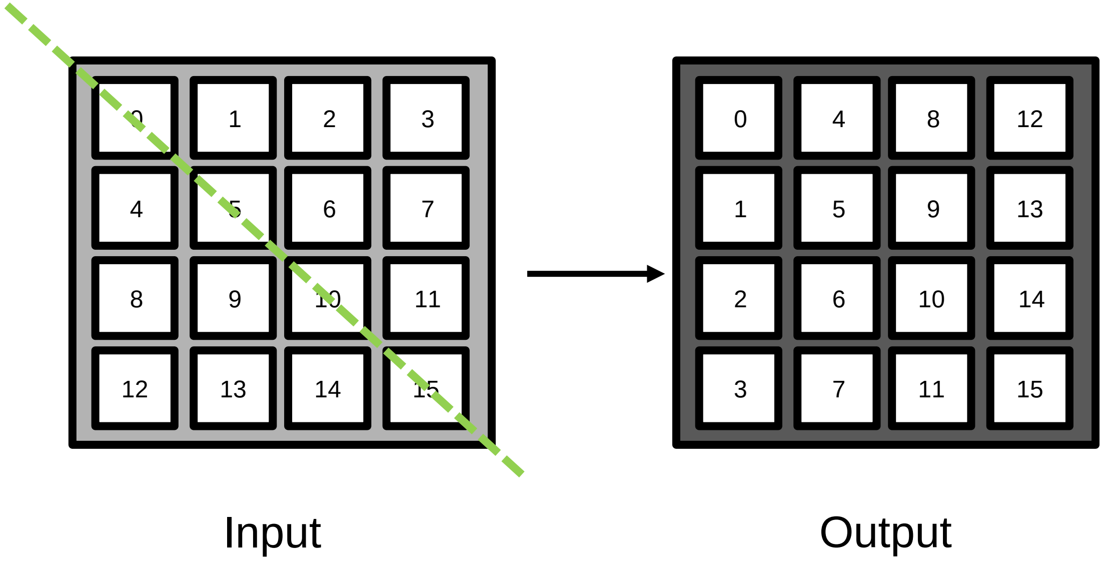


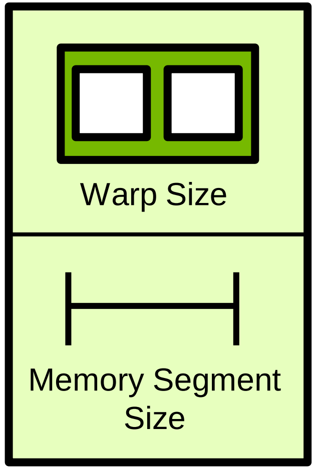




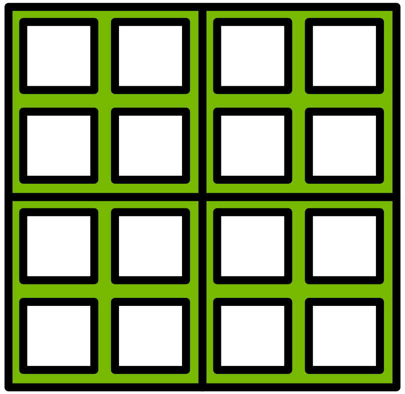


Our goal is to transpose the input by rotating all elements around the diagonal, writing the transposed elements to output



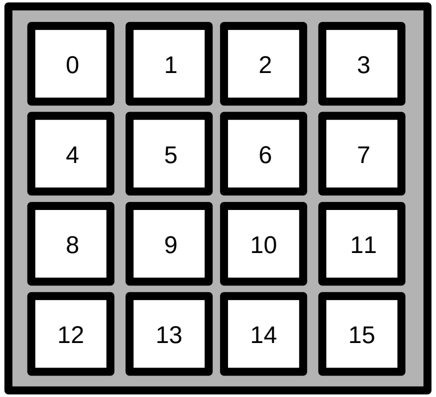


Grid

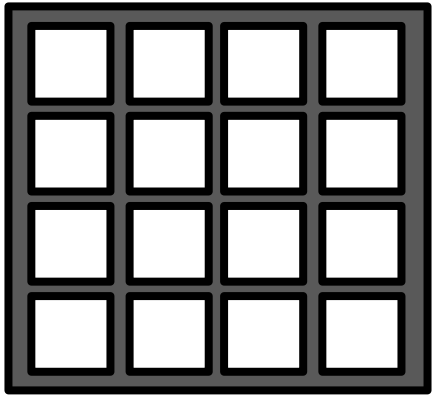


A naïve approach is to launch a grid with threads equal to input elements, and to have each thread read 1 element, then write it to output in the transposed location

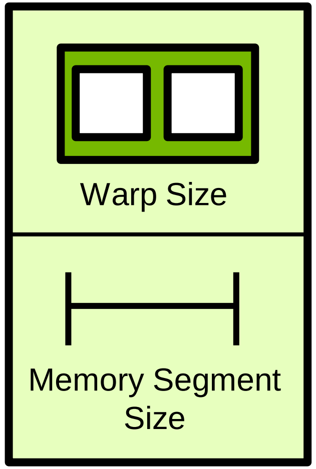
```
x, y = cuda.grid(2)

out[x][y] = in[y][x]
```



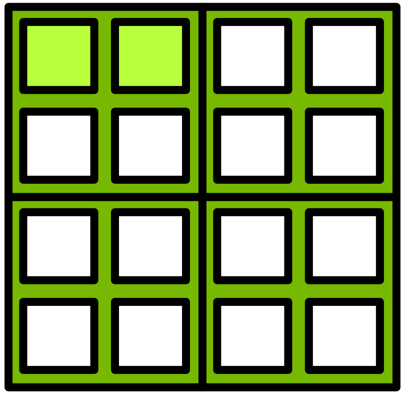
Input



Output

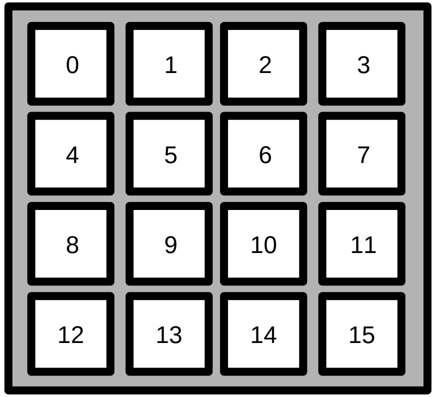


Grid

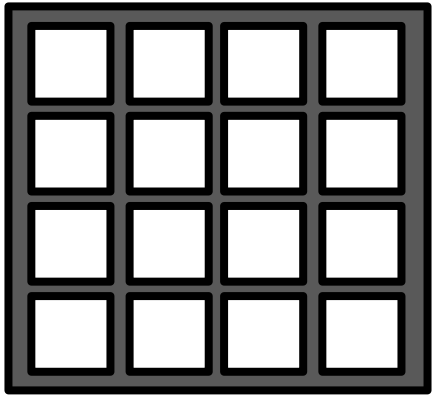


Observing the behavior of a single warp, is it the case that memory reads are coalesced? Let's dig into answering that question

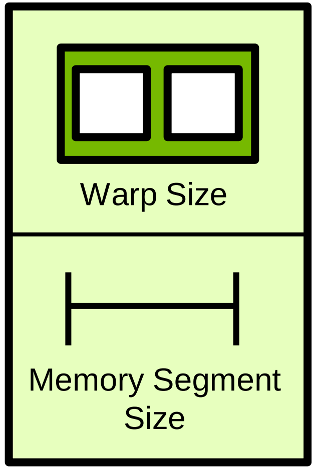
```
x, y = cuda.grid(2)

out[x][y] = in[y][x]
```



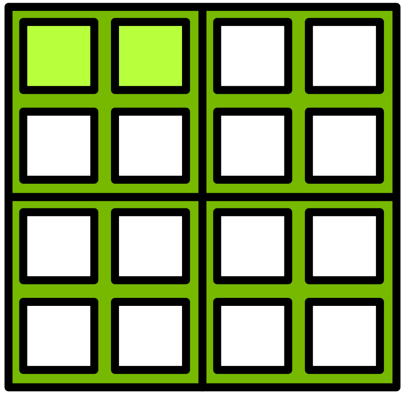
Input



Output

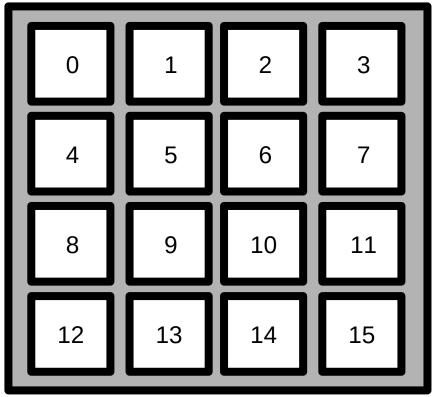


Grid

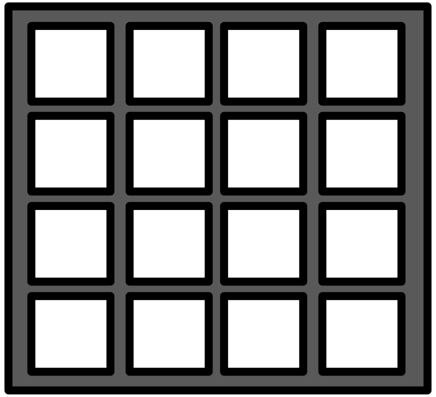


Rewriting the creation of the indexing variables, it is clearer that contiguous threads in the same warp are adjacent along the x axis

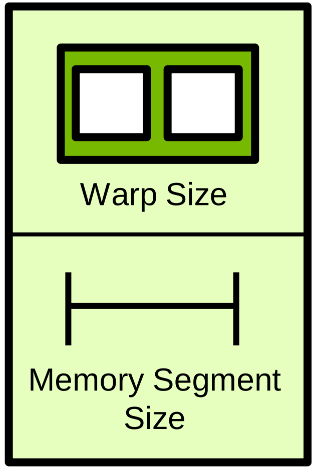
```
x = blockIdx.x * blockDim.x
        + threadIdx.x

y = blockIdx.y * blockDim.y
        + threadIdx.y

out[x][y] = in[y][x]
```



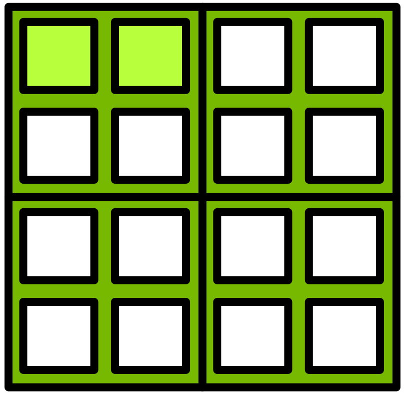
Input



Output

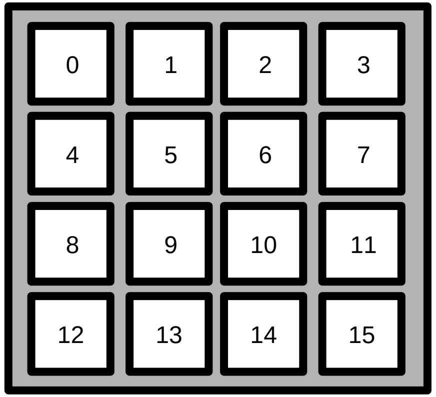


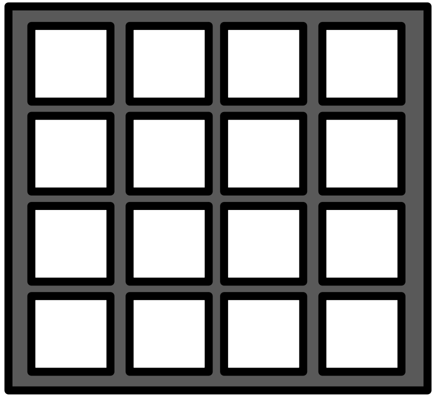
Grid



Furthermore, these contiguous threads will read elements from the rows of input where data elements are contiguous

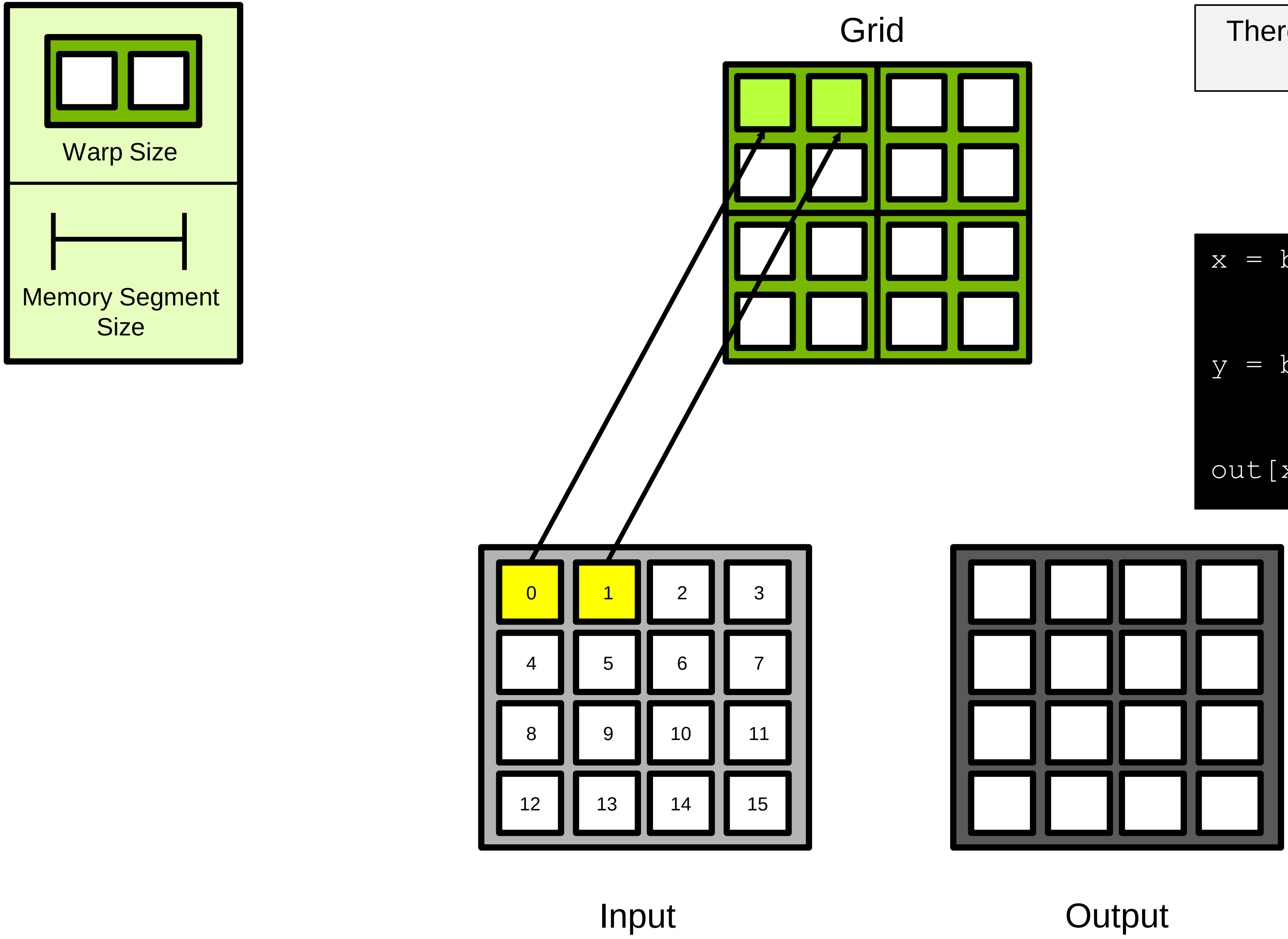
```
x = blockIdx.x * blockDim.x
          + threadIdx.x

y = blockIdx.y * blockDim.y
          + threadIdx.y

out[x][y] = in[y][x]
```





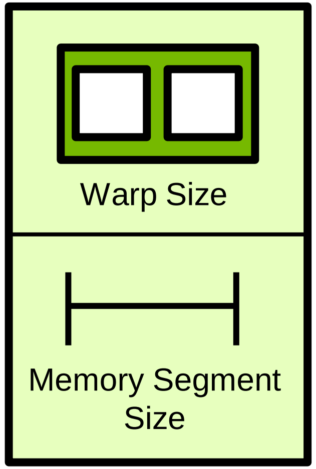
Input

Output

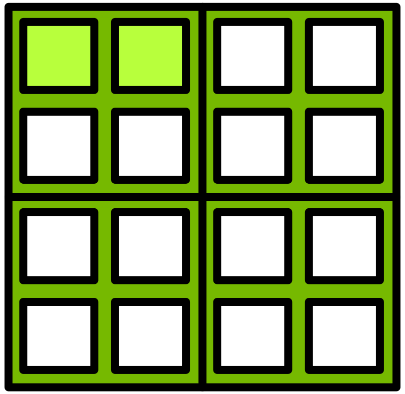


Therefore, it makes sense that reads from input are coalesced

```
x = blockIdx.x * blockDim.x
        + threadIdx.x

y = blockIdx.y * blockDim.y
        + threadIdx.y

out[x][y] = in[y][x]
```

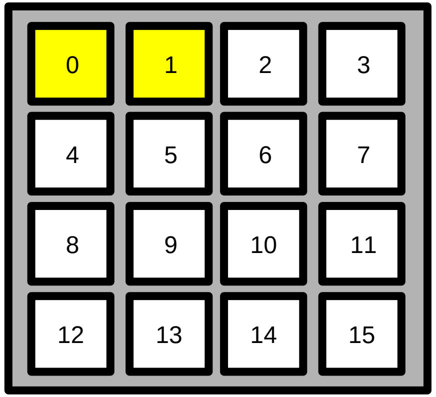


Grid

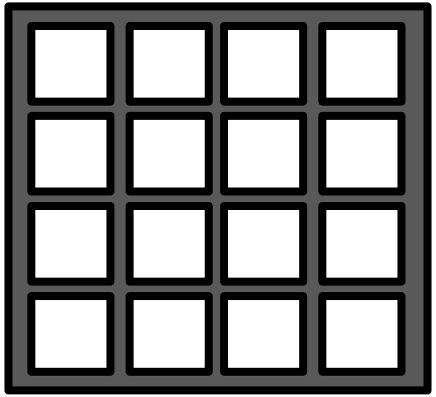


What about this warp's writes to output, will they be coalesced?

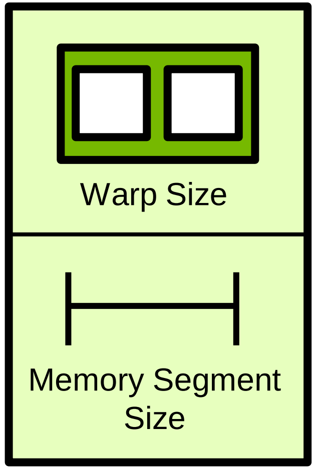
```
x = blockIdx.x * blockDim.x
        + threadIdx.x

y = blockIdx.y * blockDim.y
        + threadIdx.y

out[x][y] = in[y][x]
```



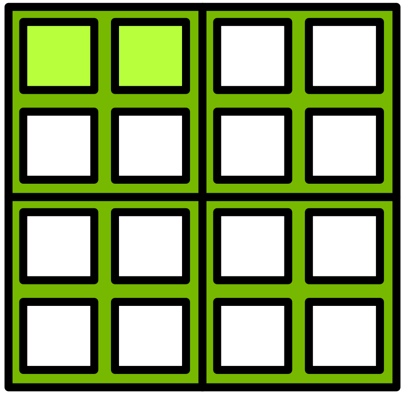
Input



Output

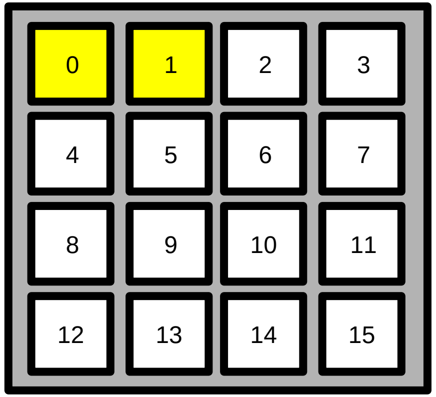


Grid

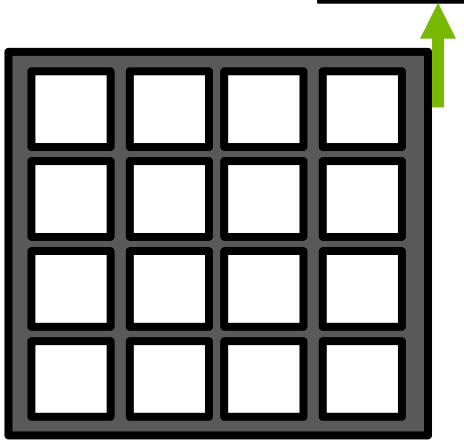


Here we see that contiguous threads in the same warp will be writing along a column in output

```
x = blockIdx.x * blockDim.x
        + threadIdx.x

y = blockIdx.y * blockDim.y
        + threadIdx.y

out[x][y] = in[y][x]
```

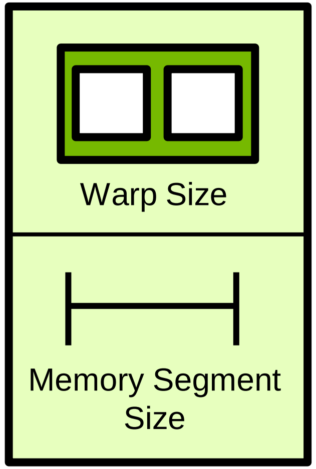


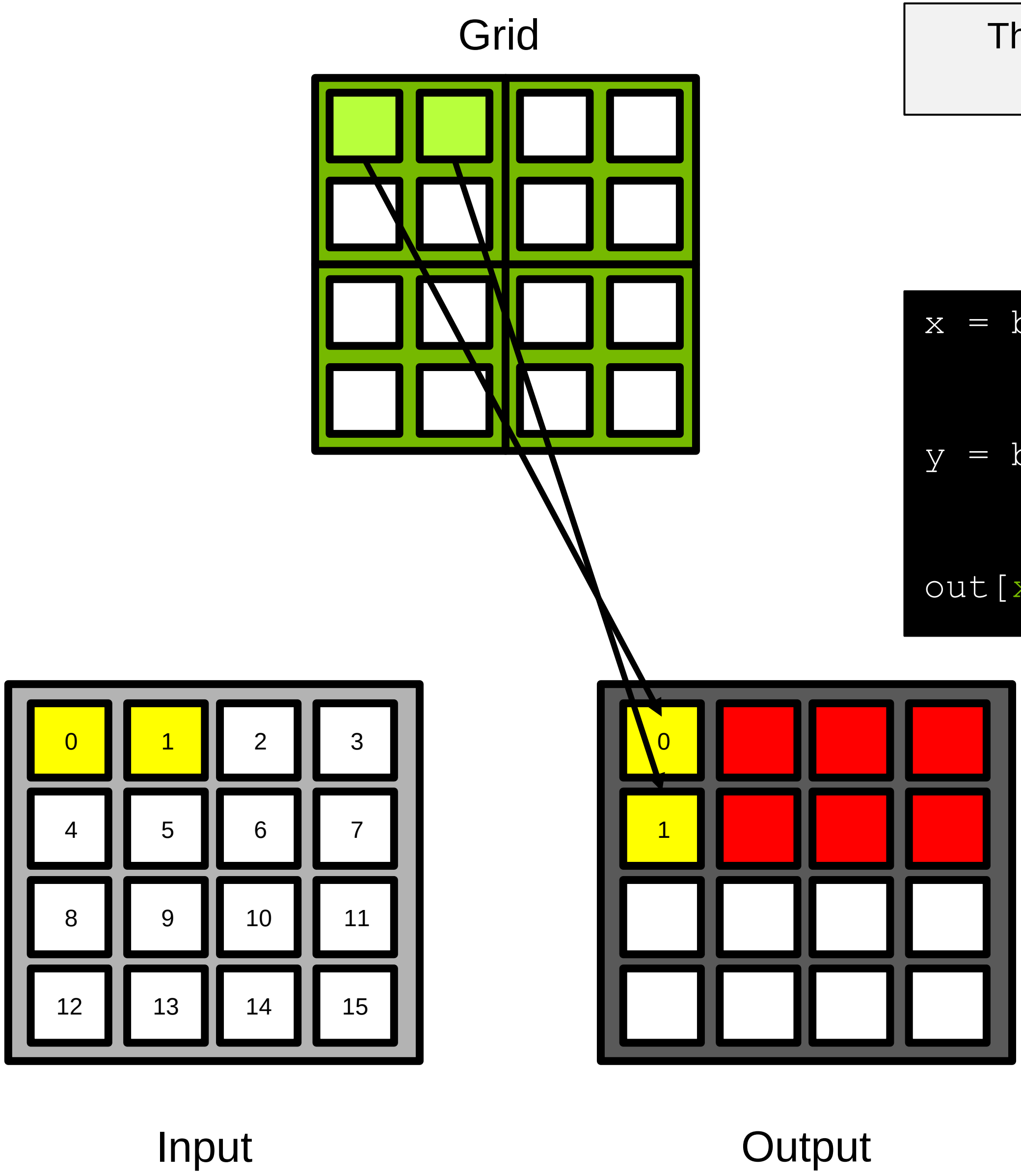
Input



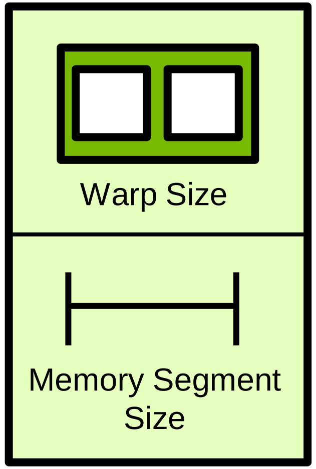
Output

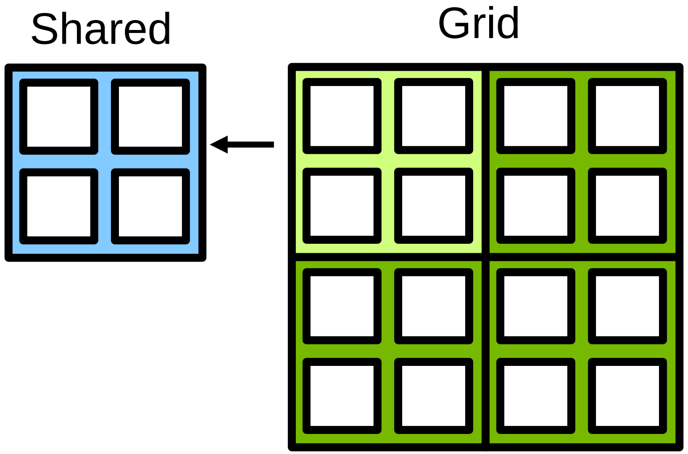
Therefore, the writes will not be coalesced





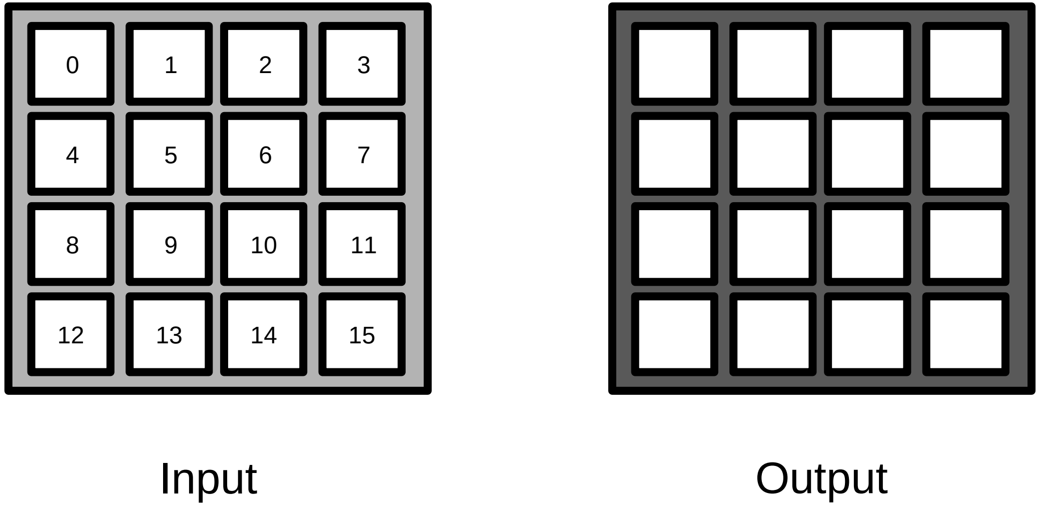

```
x = blockIdx.x * blockDim.x
         + threadIdx.x

y = blockIdx.y * blockDim.y
         + threadIdx.y

out[x][y] = in[y][x]
```

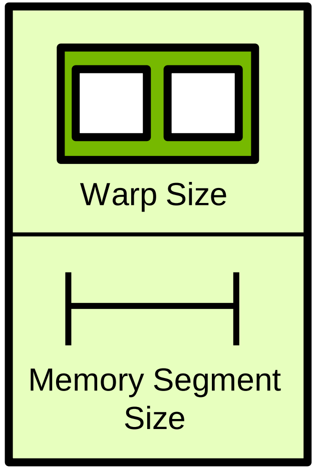


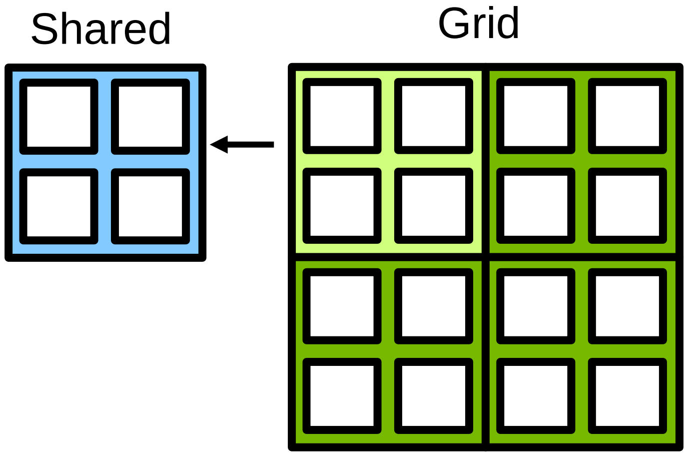


We can use shared memory to make coalesced reads and writes. Here, each block will allocate a (2,2) shared memory tile

```
tile = cuda.shared.array(2,2)
```

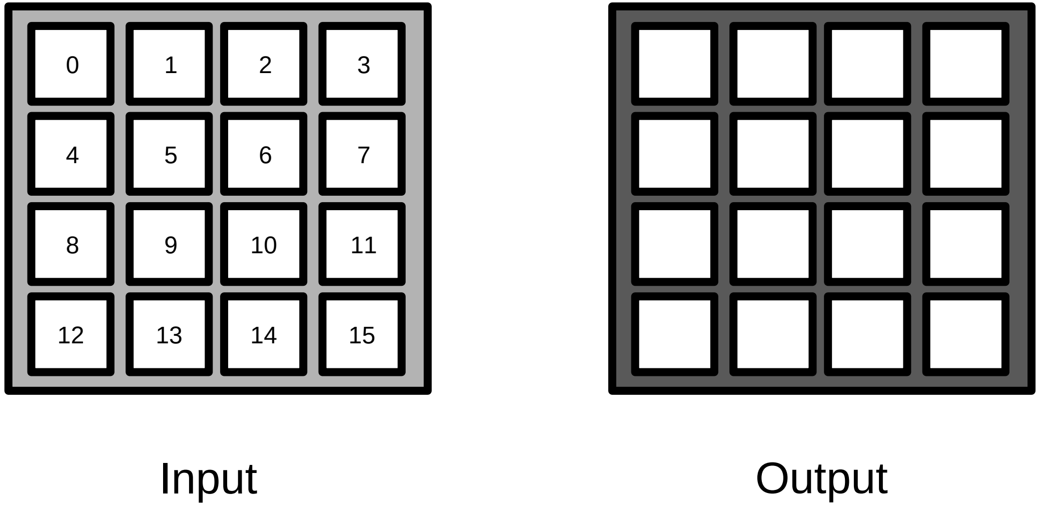


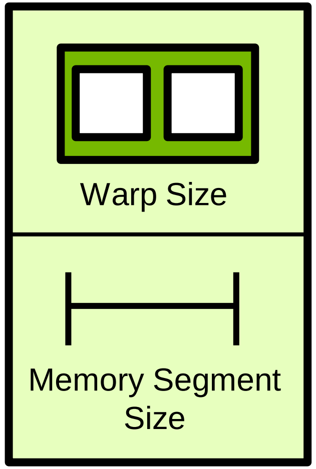


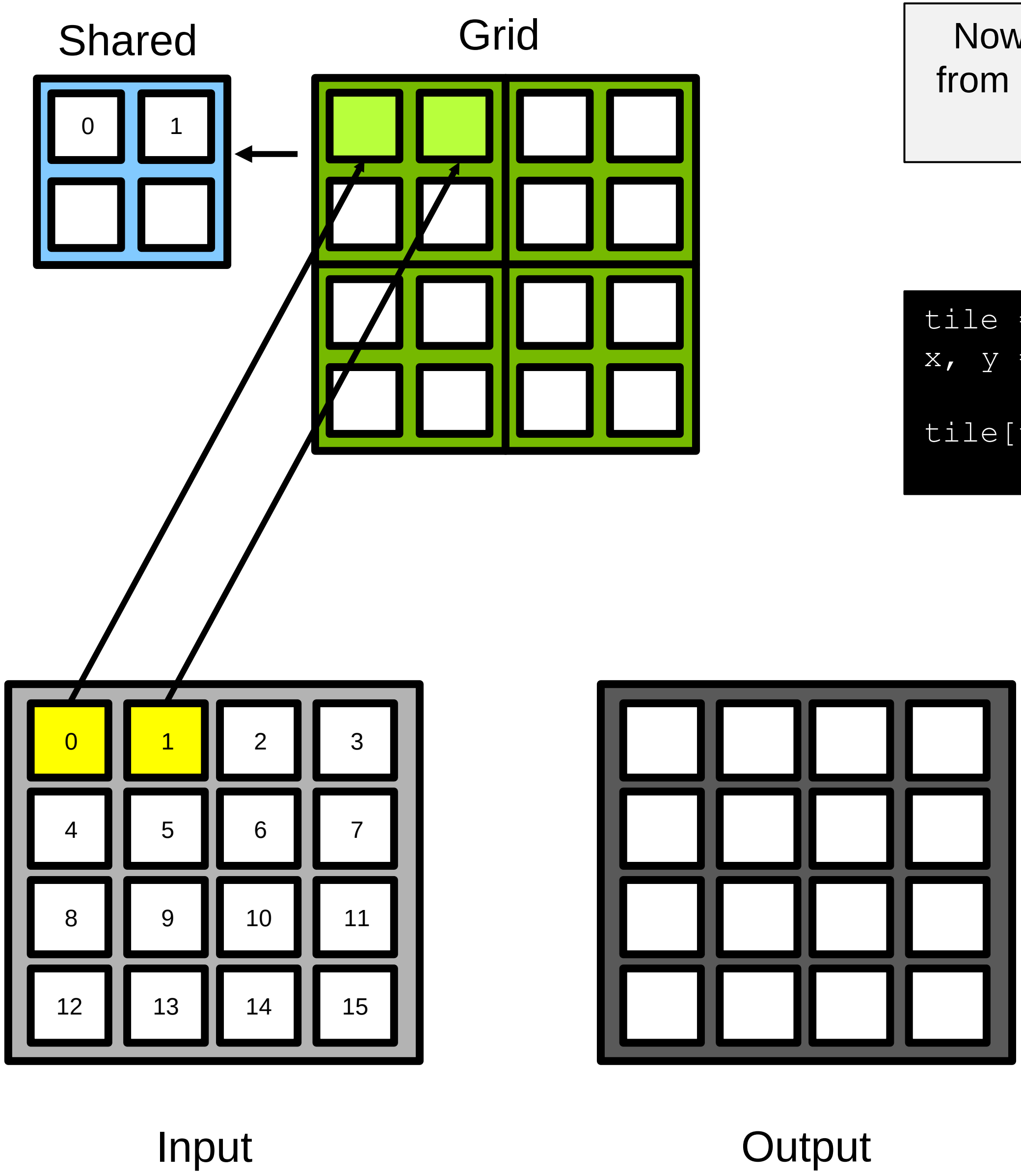


(It is worth reminding that in our slides, to preserve space, 2 threads is a warp length. A real warp is 32 threads)

```
tile = cuda.shared.array(2,2)
```

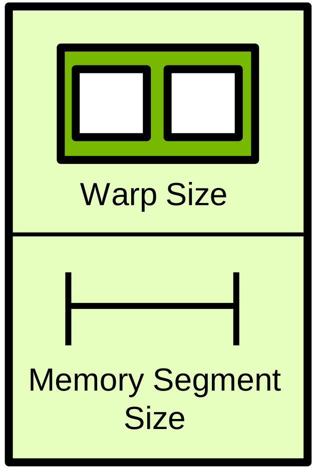


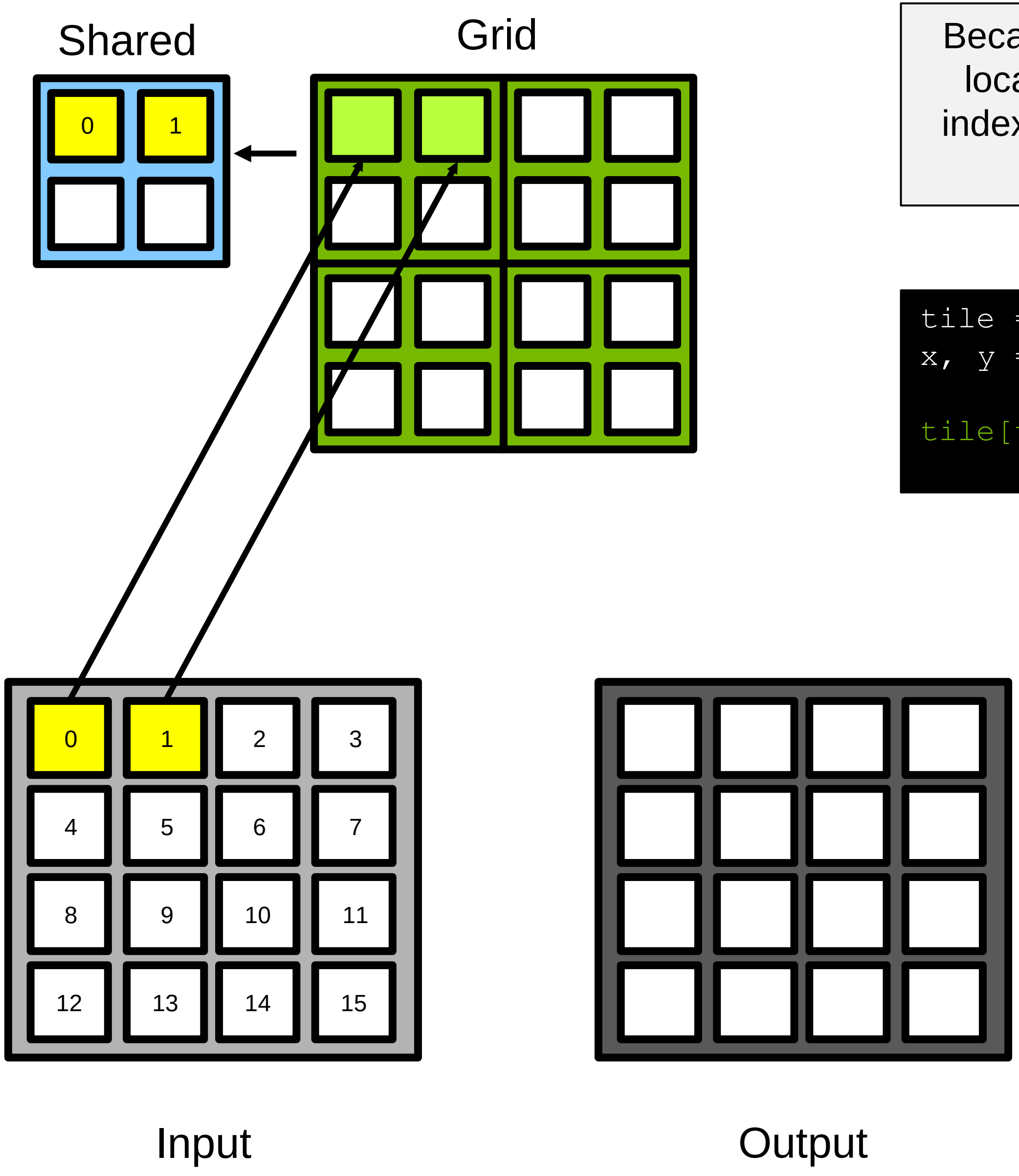




Now we can make coalesced reads from input, and write the values to the block's shared memory tile

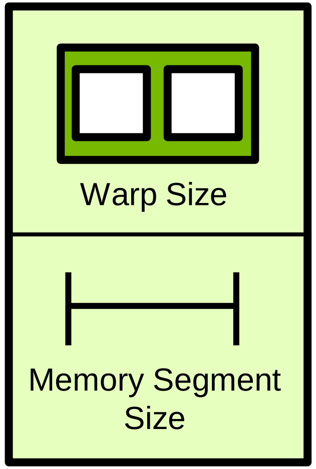
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
```

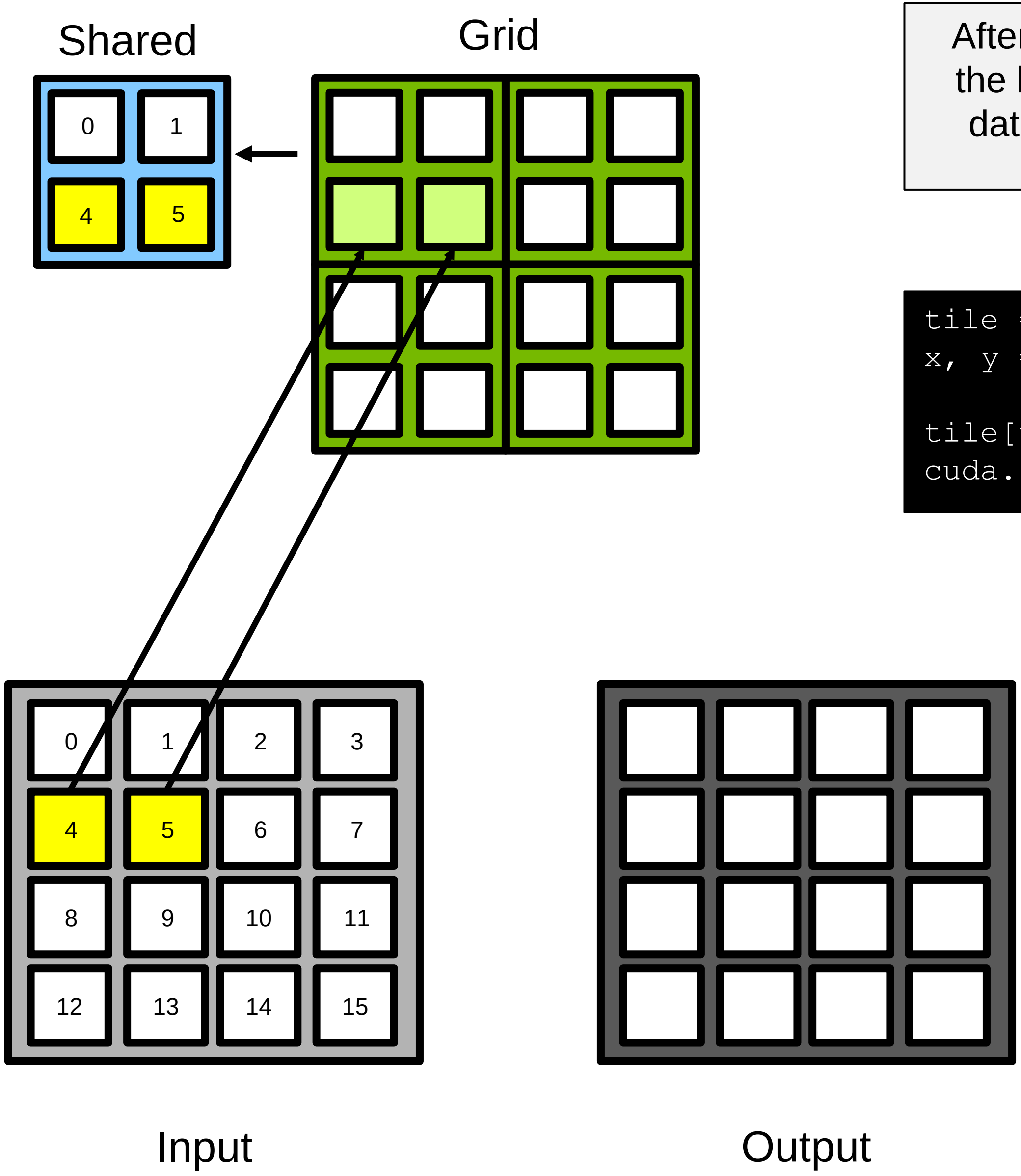




Because each shared memory tile is local to the block (not the grid) we index into it using thread indices, not grid indices

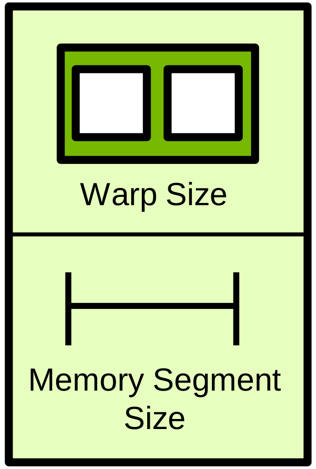
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
```

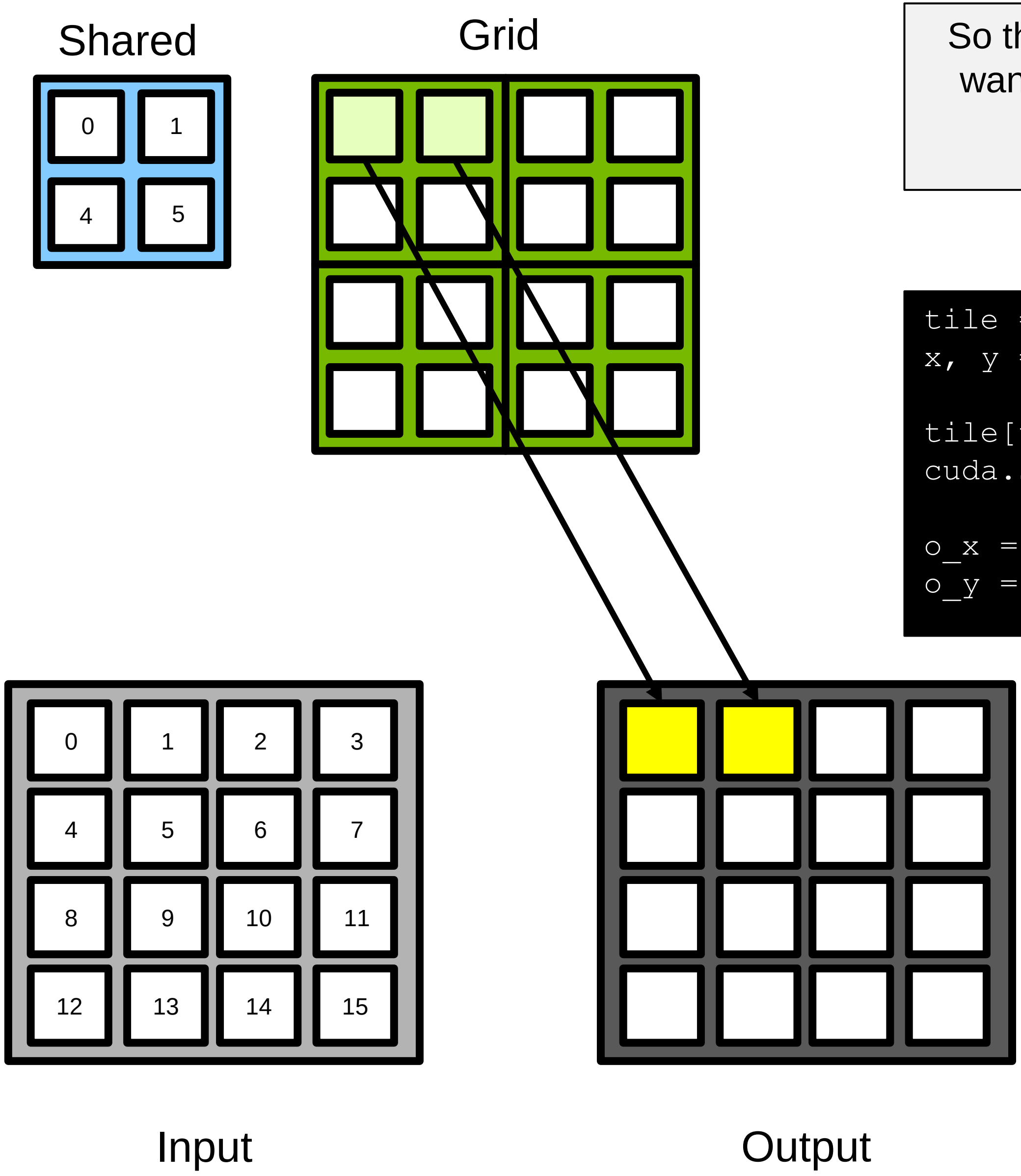




After synchronizing on all threads in the block, the tile will contain all the data this block needs to begin the writes

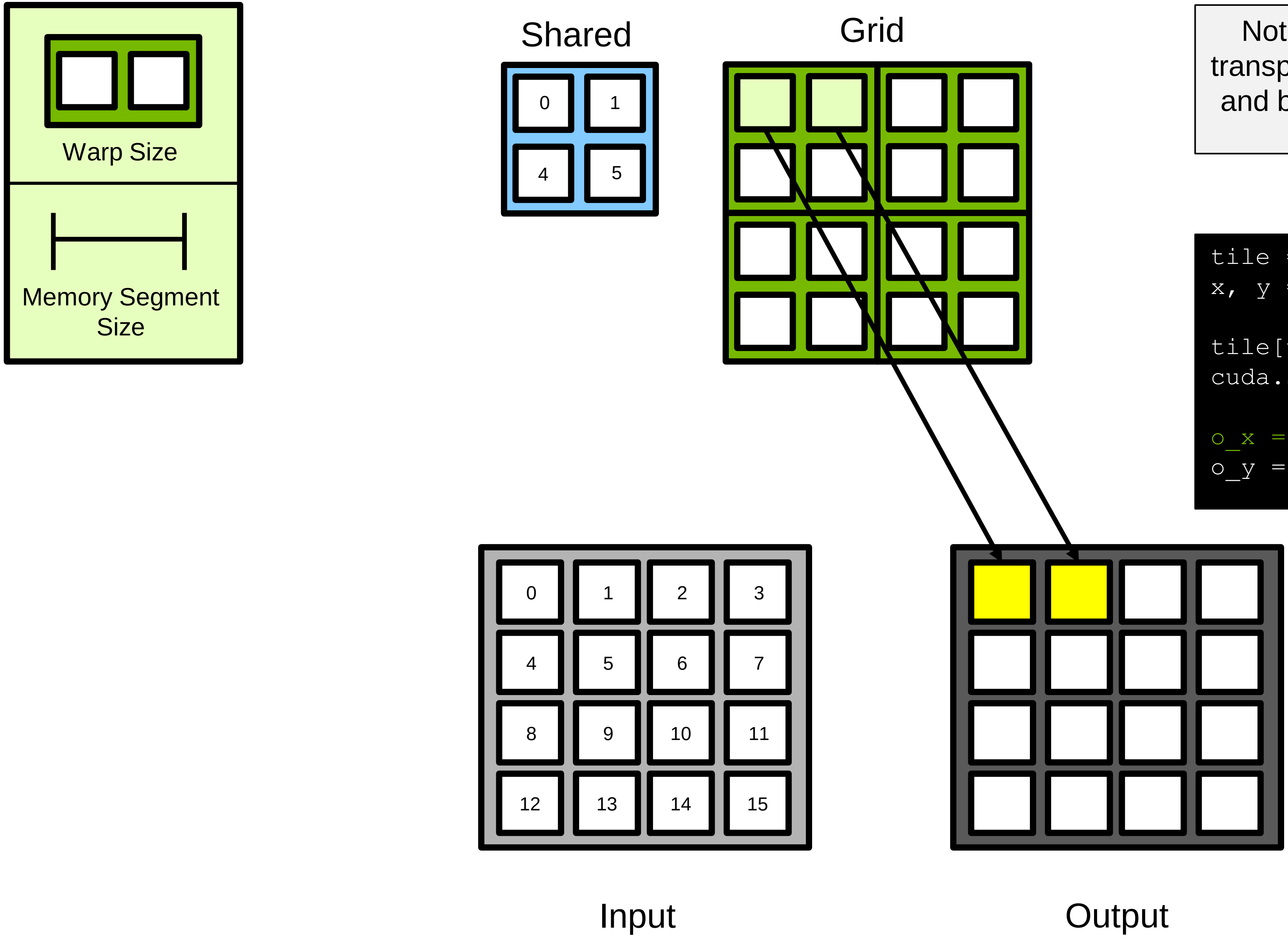
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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





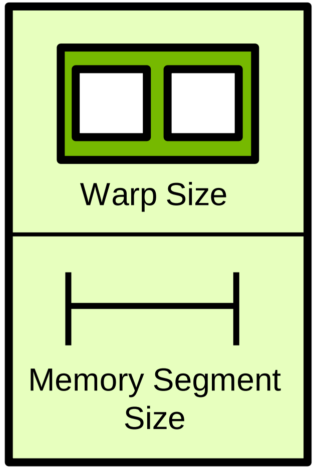
So that the writes are coalesced, we want each warp to write to a row in output

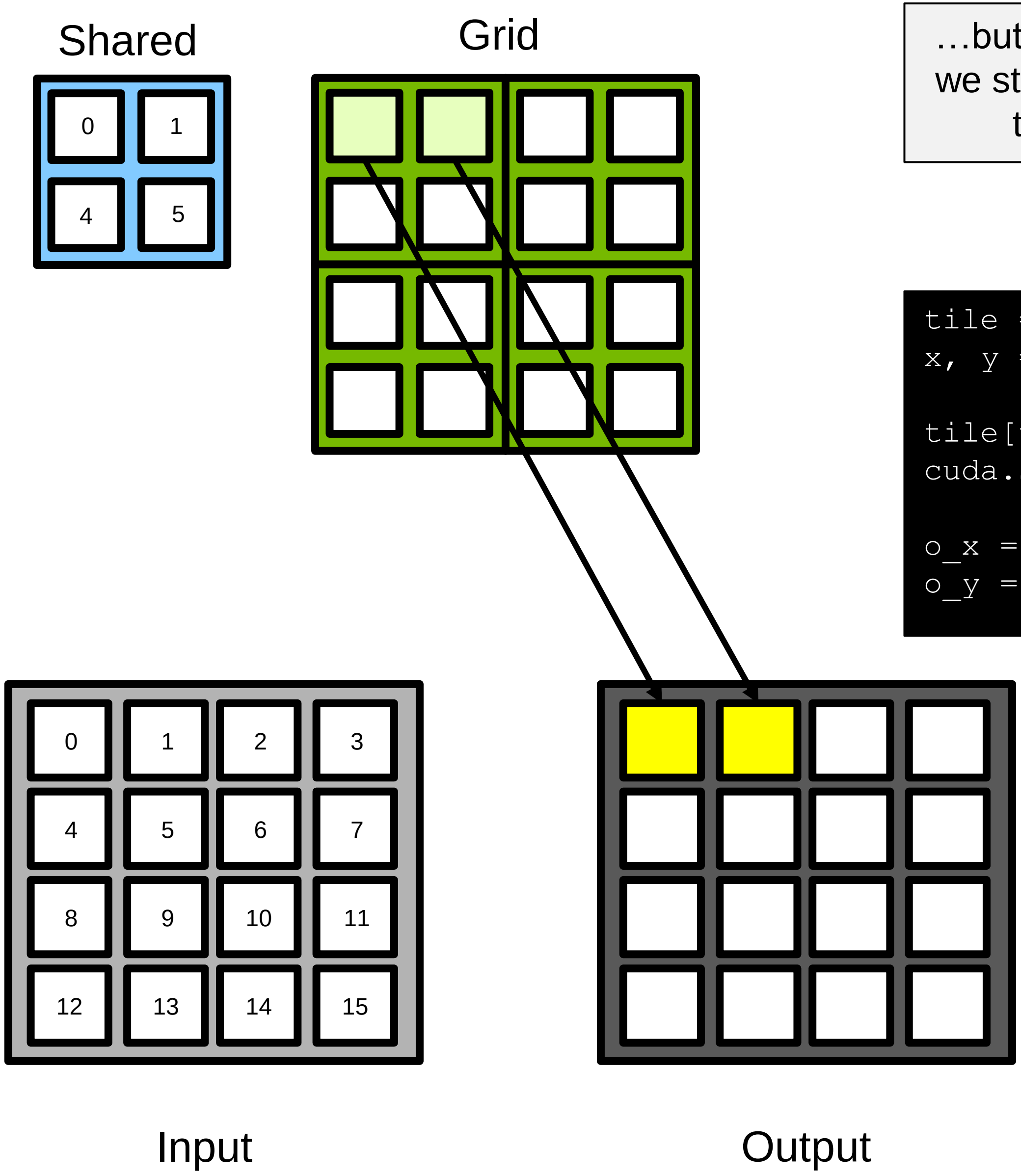
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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
```



Notice that to write to output at the transposed locations we use blockIdx.y and blockDim.y to calculate the x axis index in output…

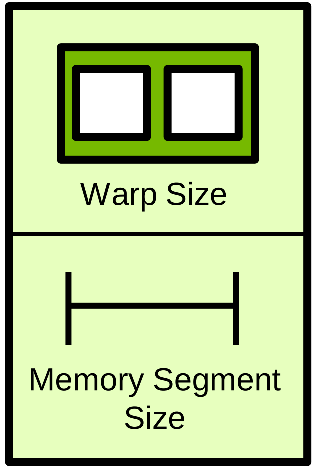
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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
```

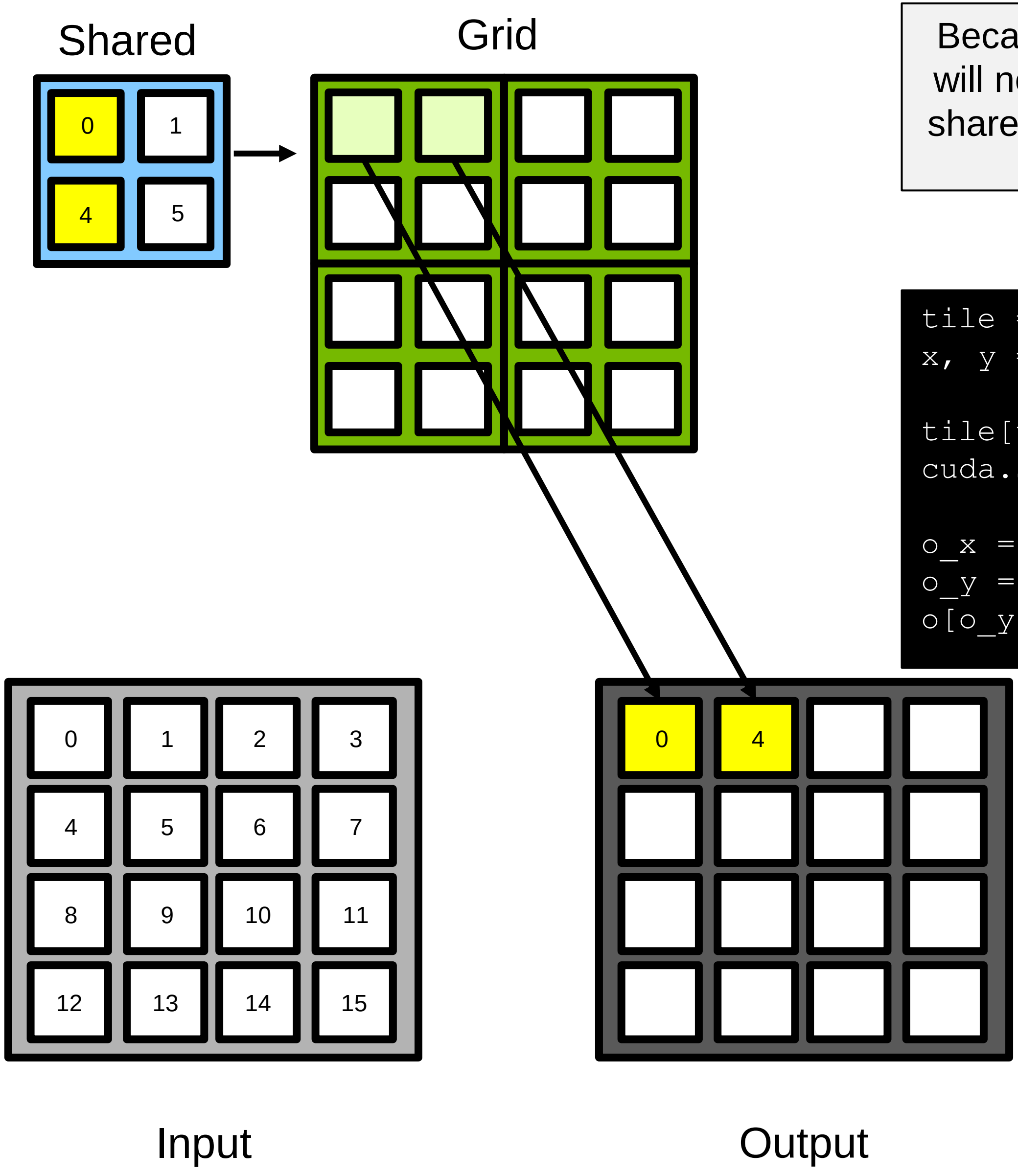




…but to accomplish coalesced writes, we still map increments to threadIdx.x to be along the x output axis

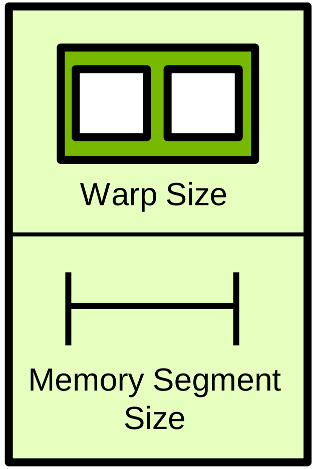
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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
```

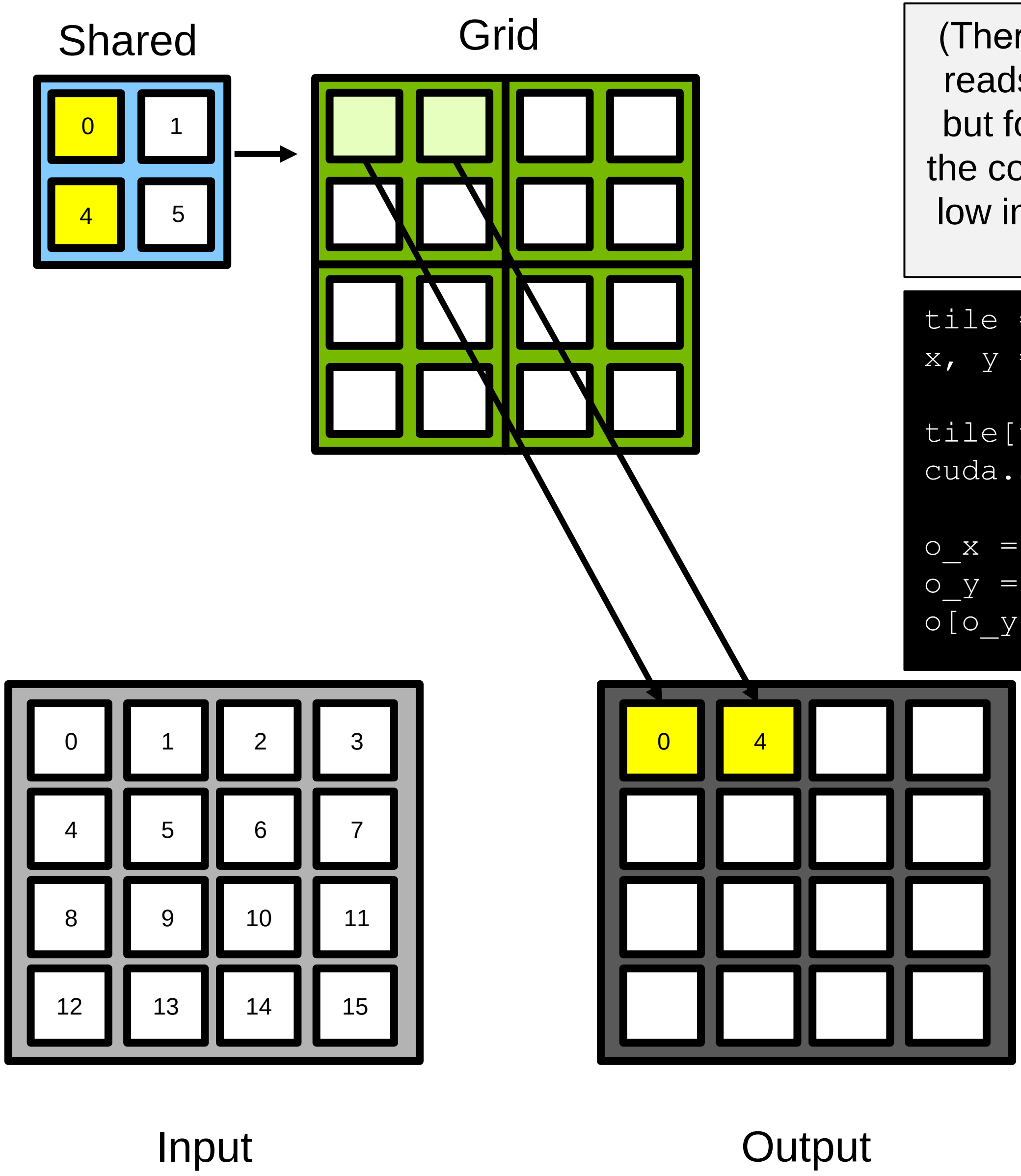




Because of this last detail, each warp will need to read from a column of the shared memory tile in order to perform the transpose

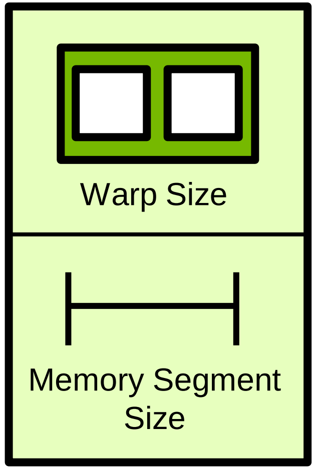
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

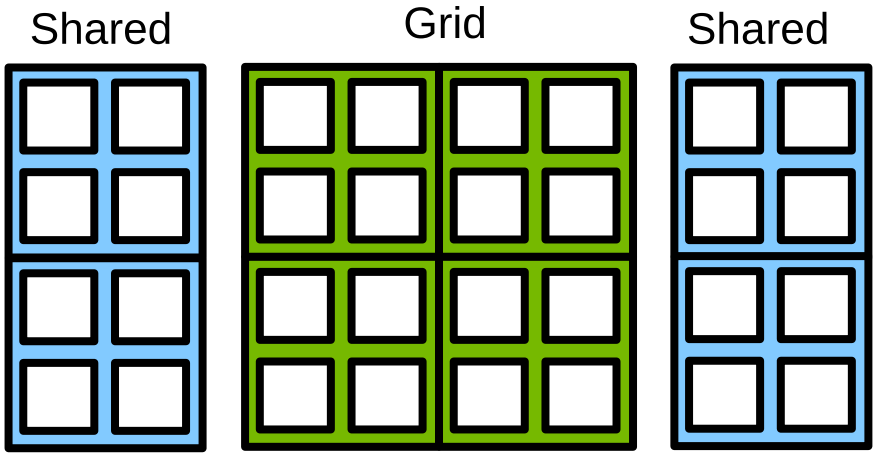




(There's more to come about efficient reads/writes to/from shared memory, but for now know that reading across the column in shared memory has very low impact compared to doing so with global memory)

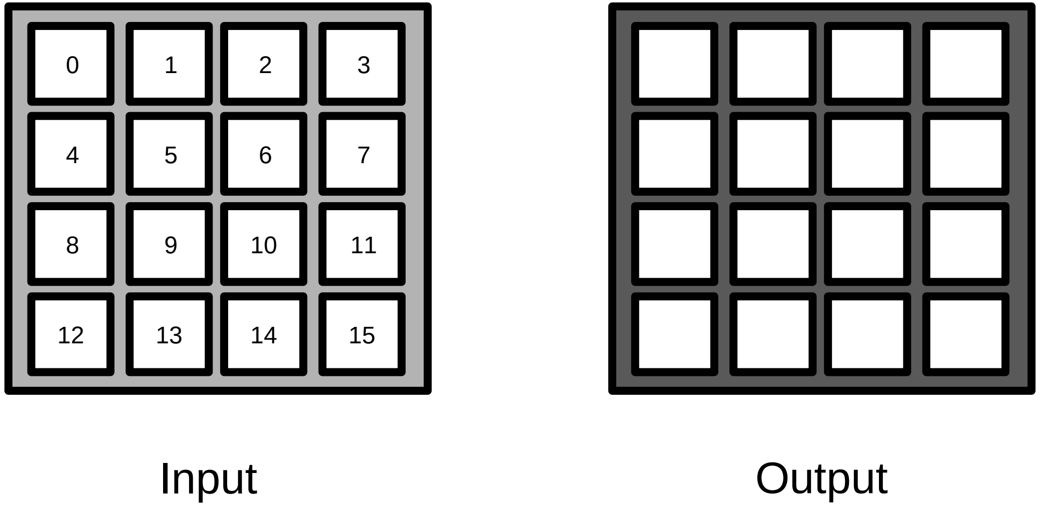
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

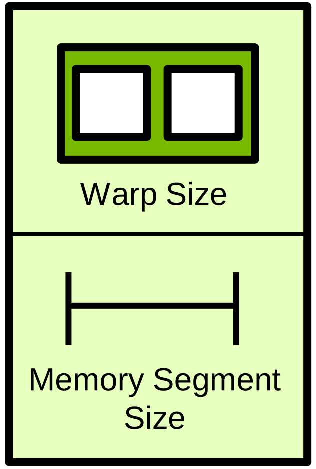


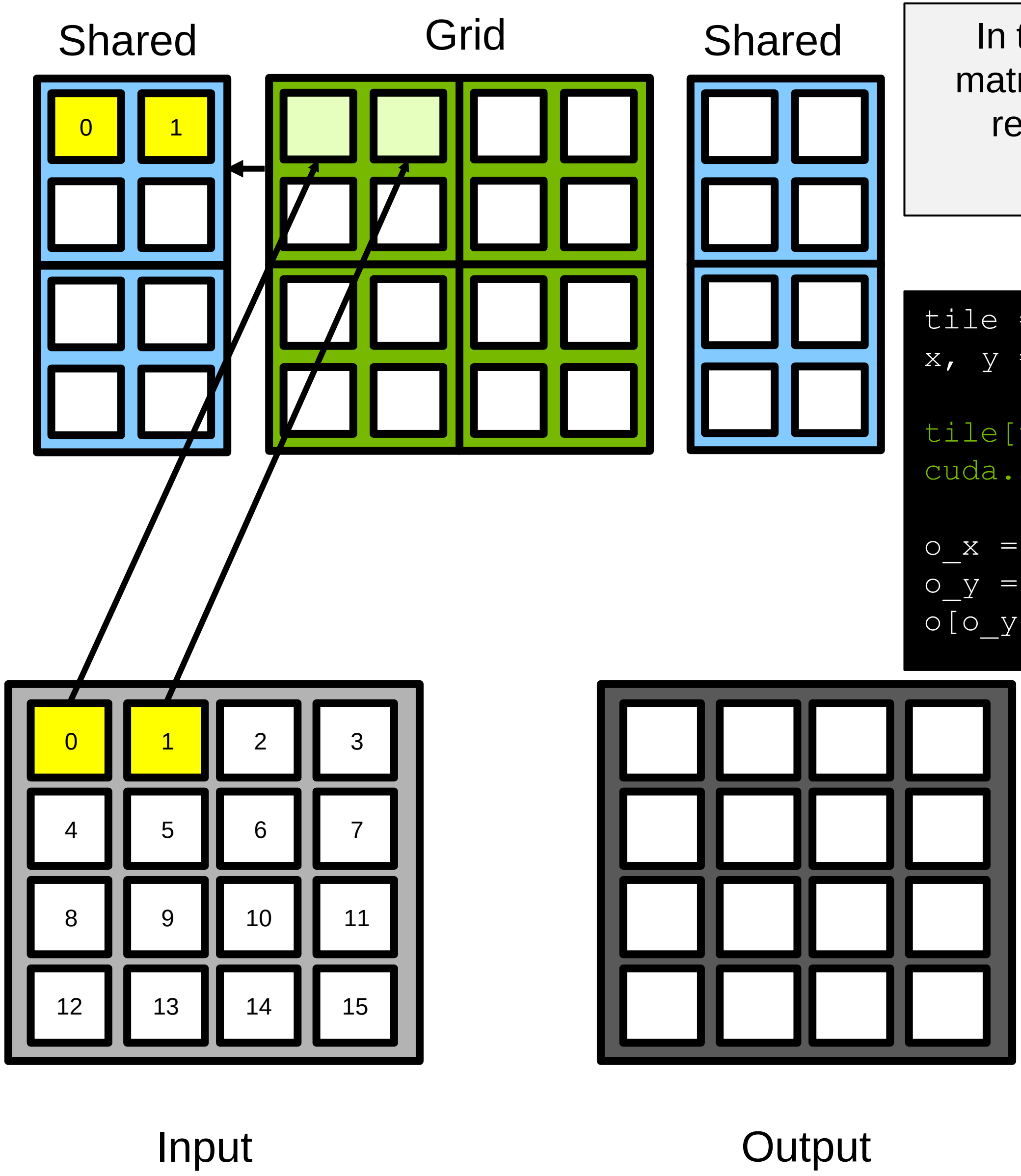


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

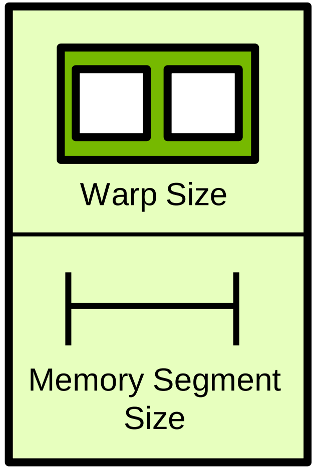


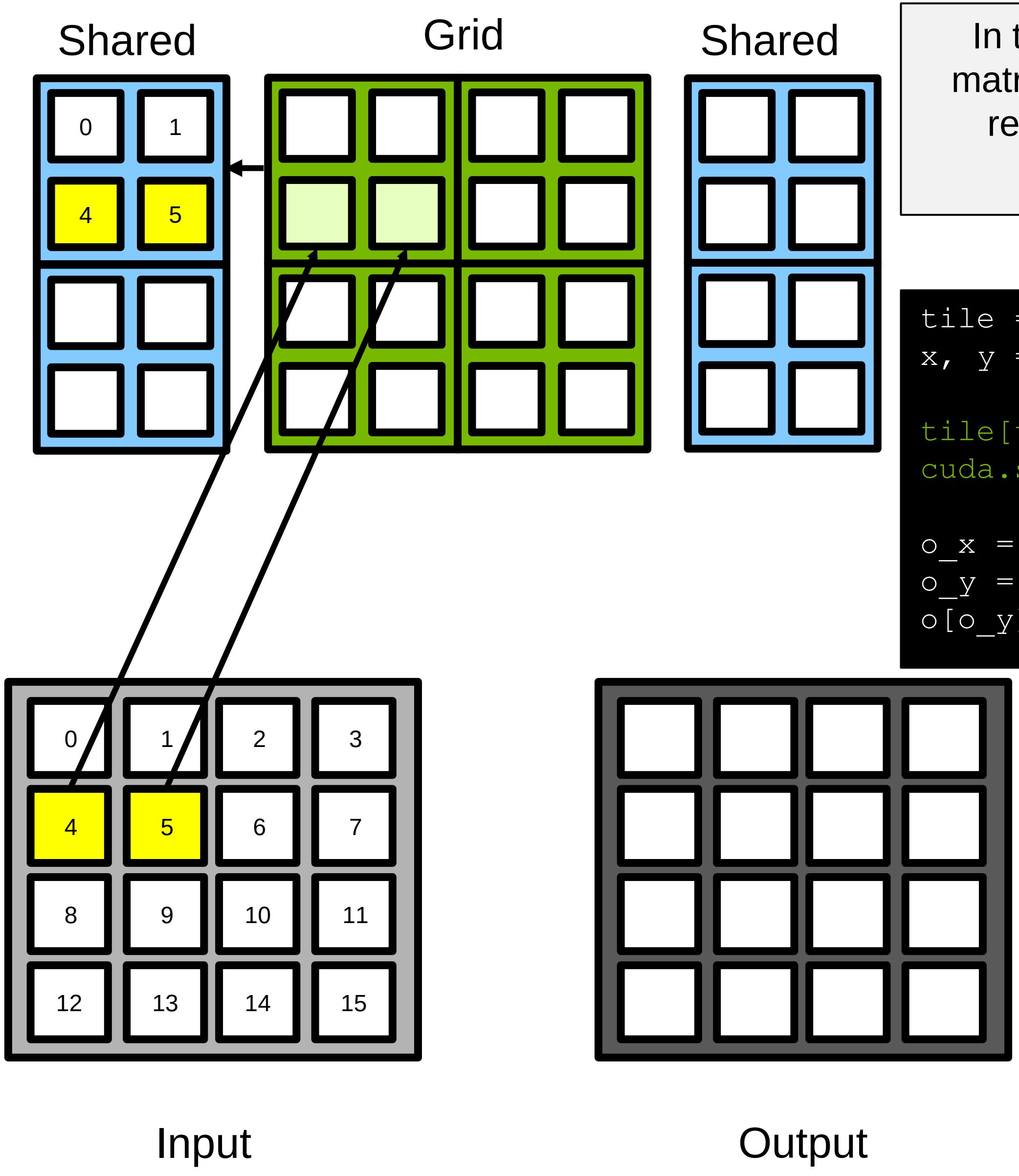




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

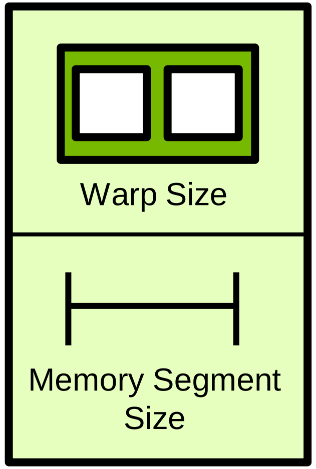
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

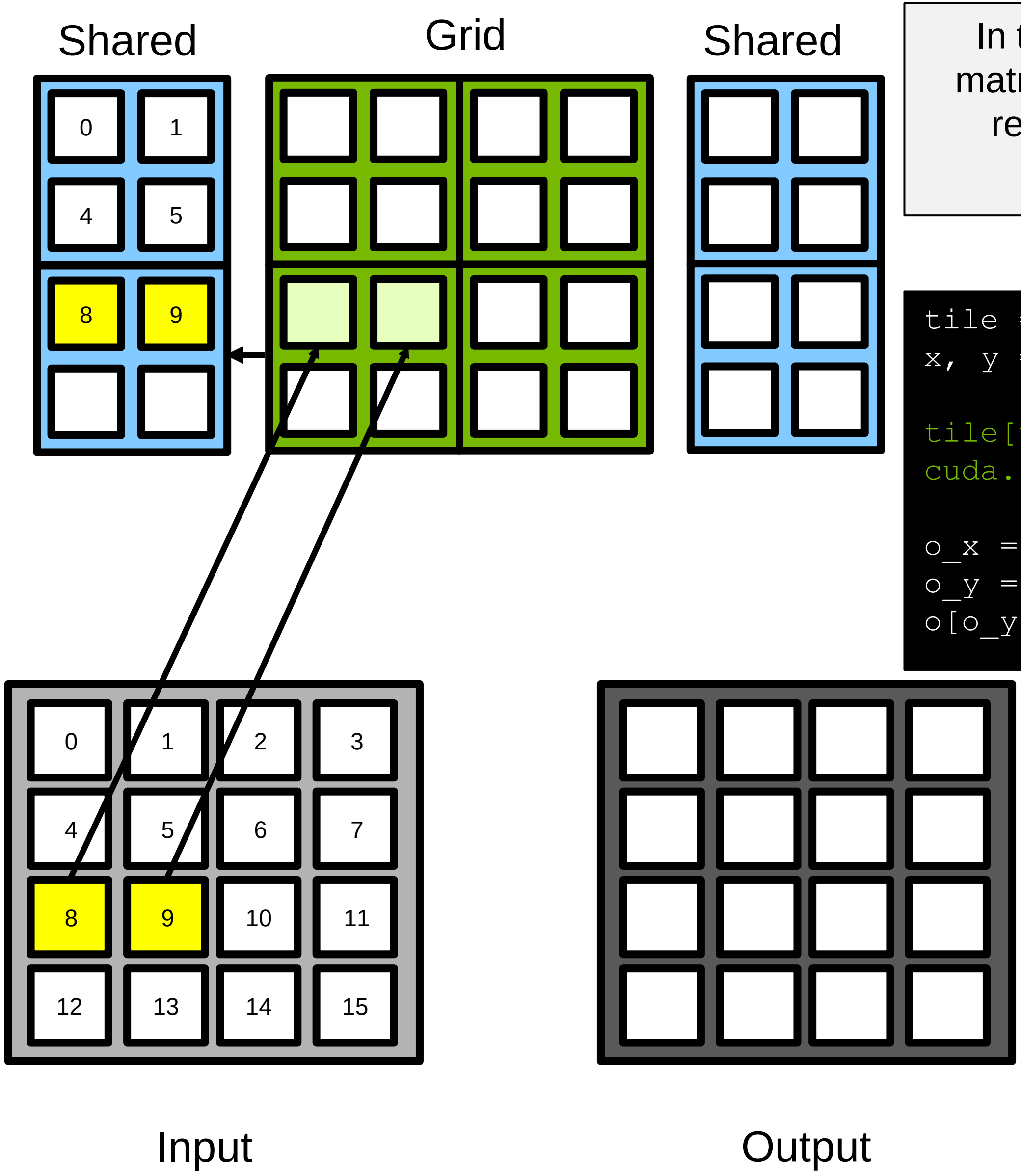




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

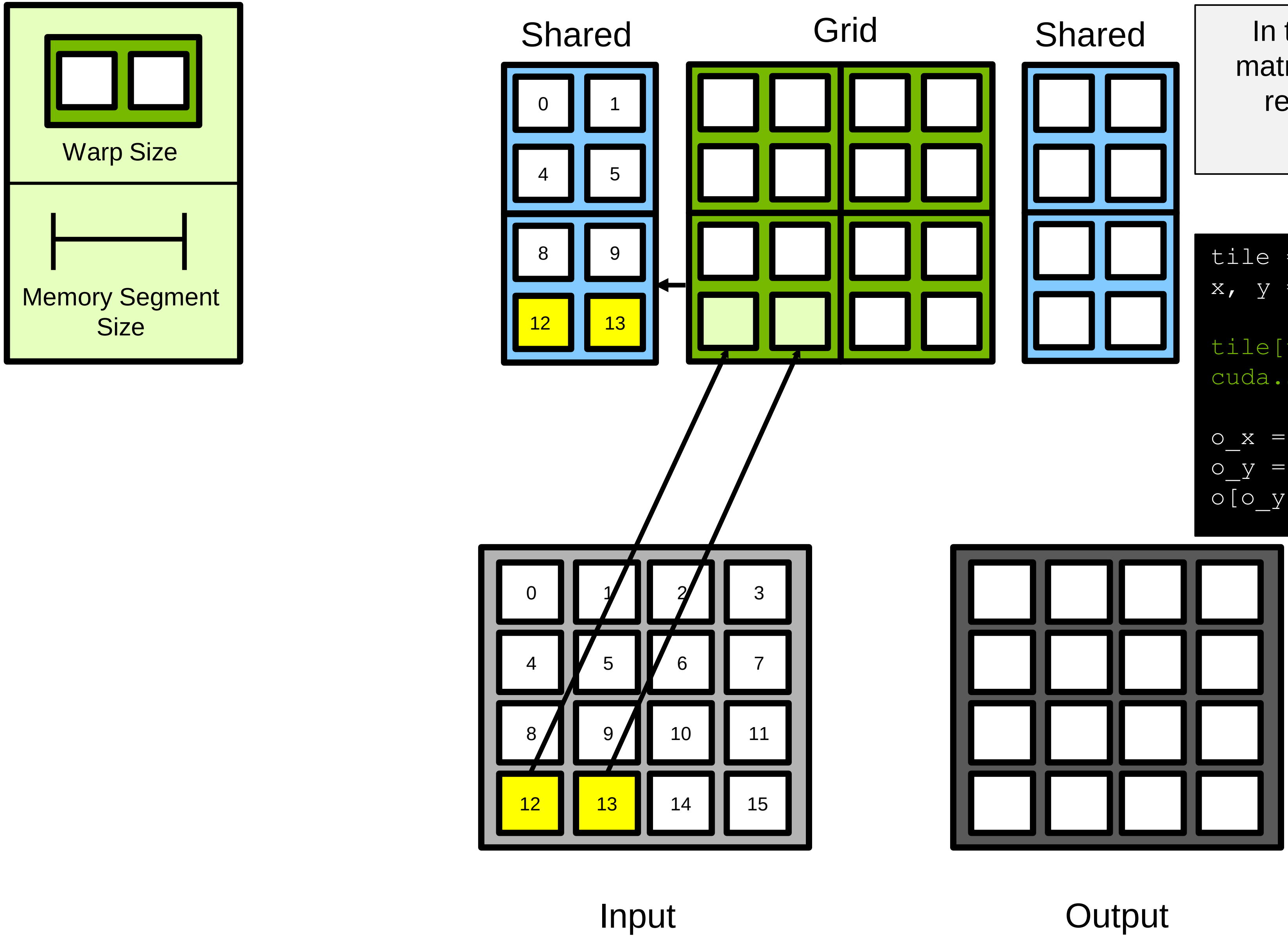
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```





In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

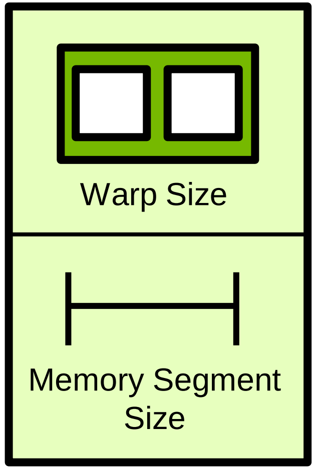
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

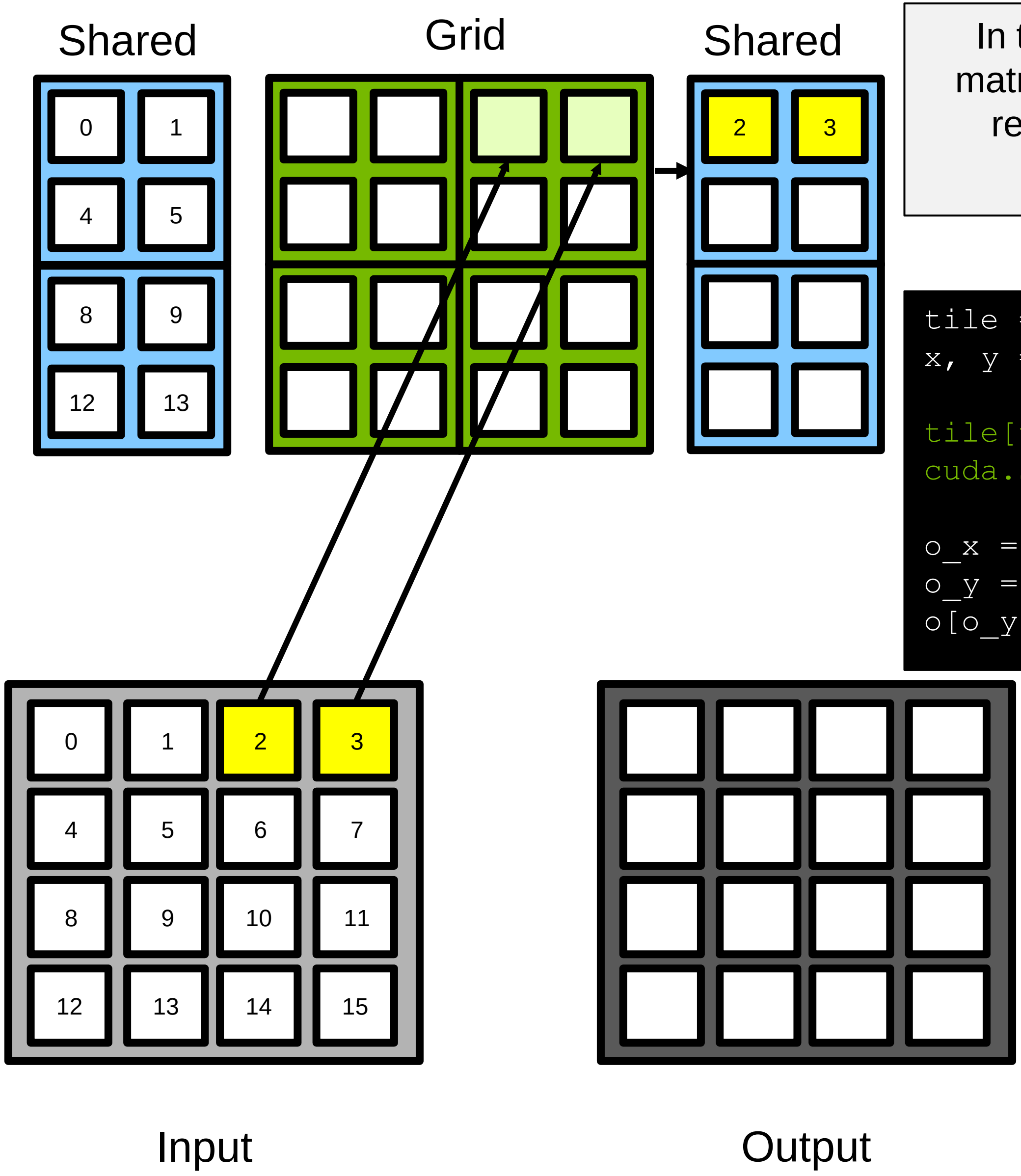


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

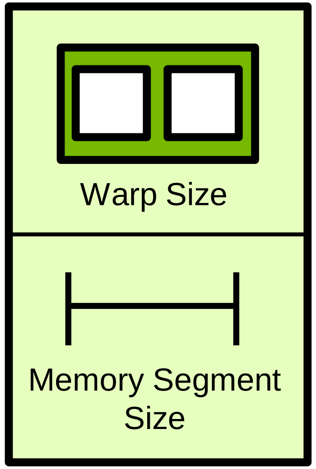
DEEP LEARNING INSTITUTE

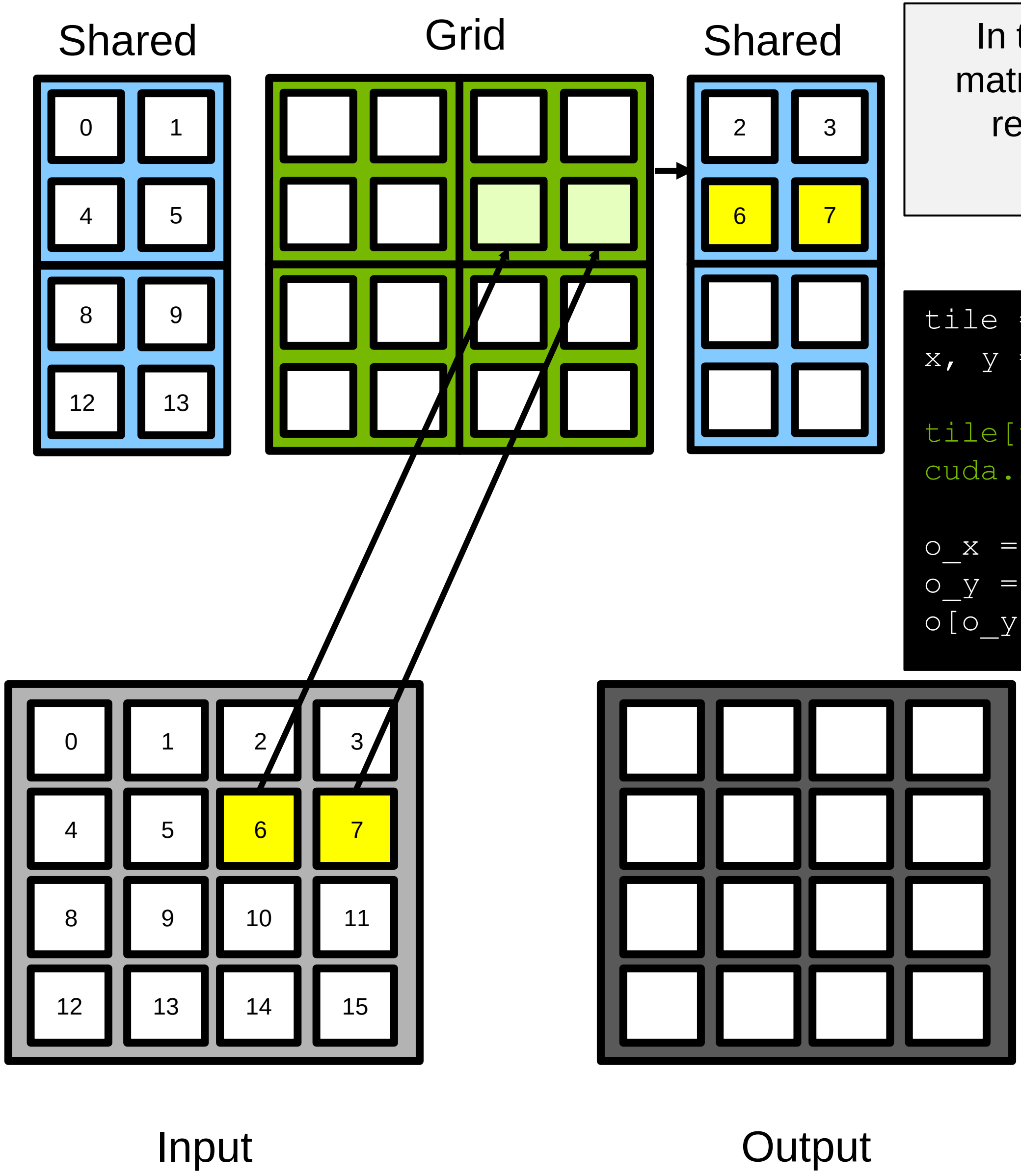




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

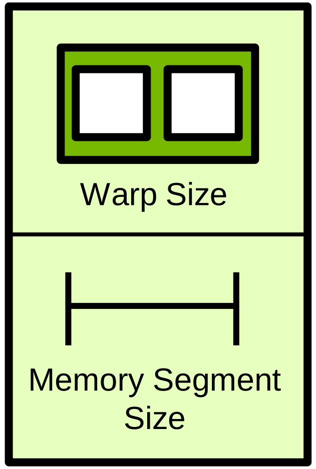
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

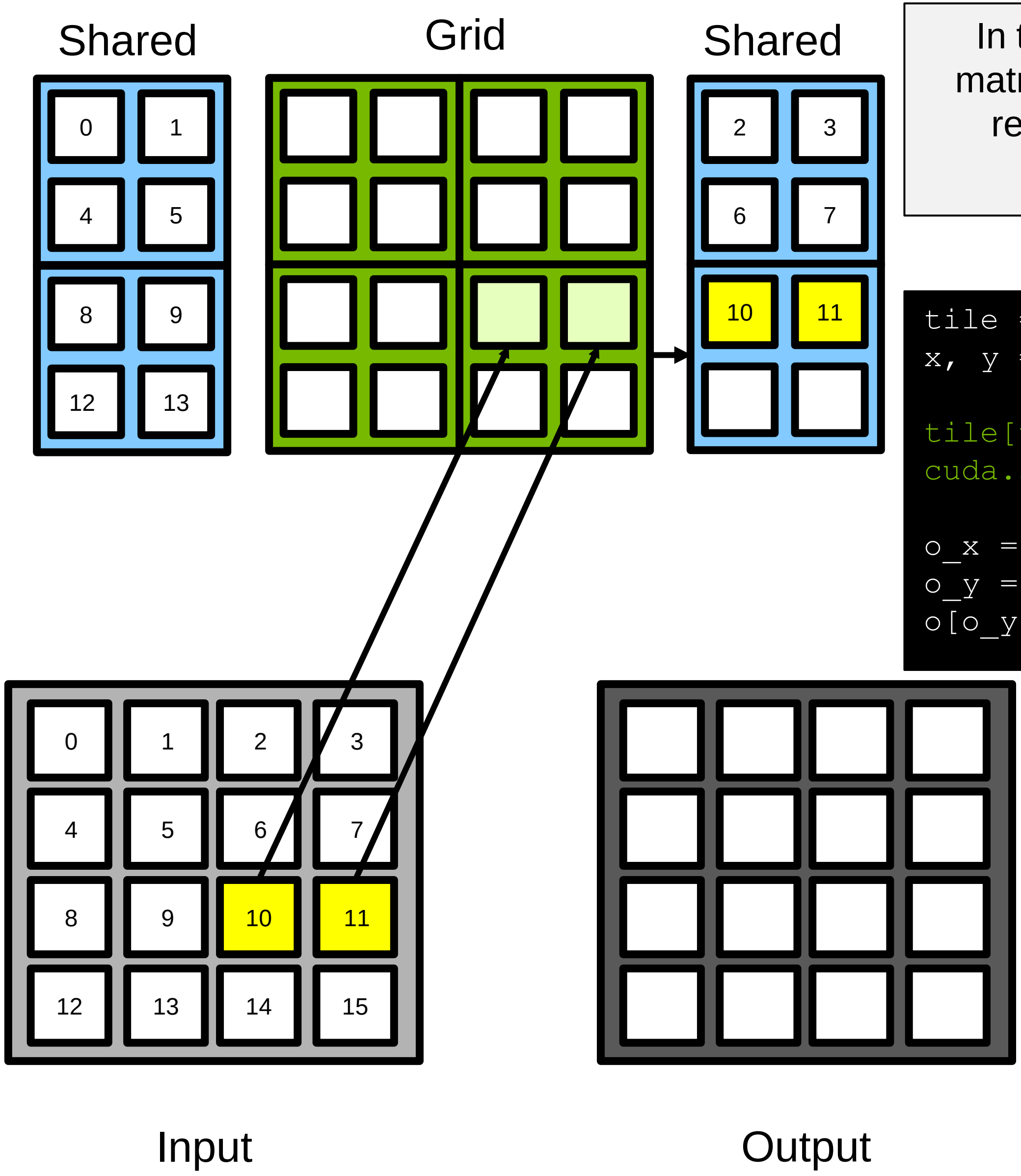




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

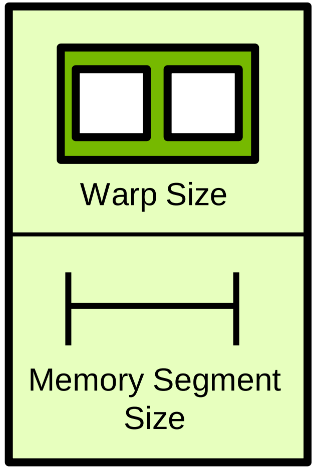
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

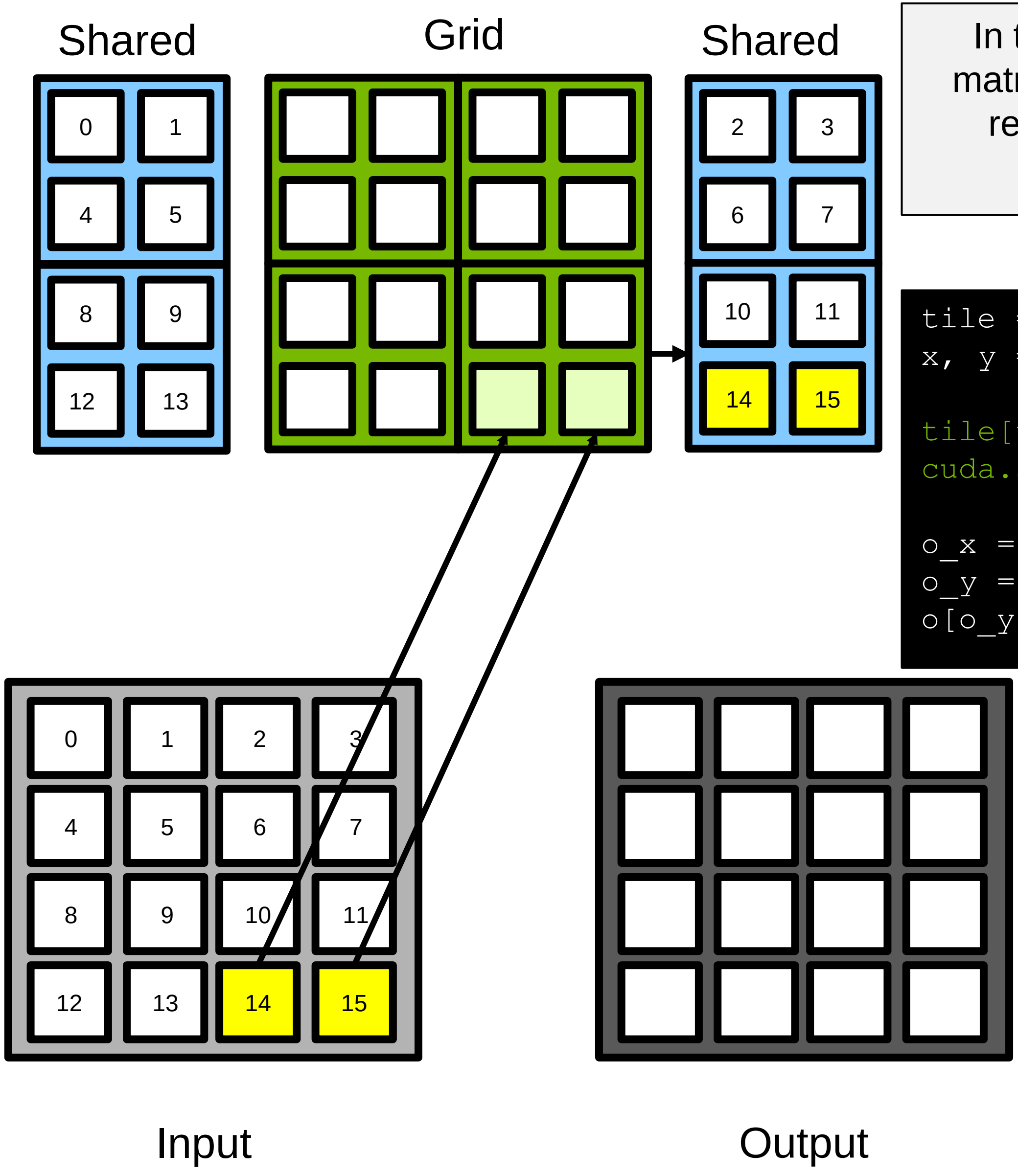




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

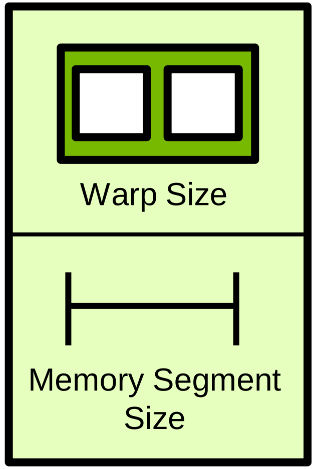
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

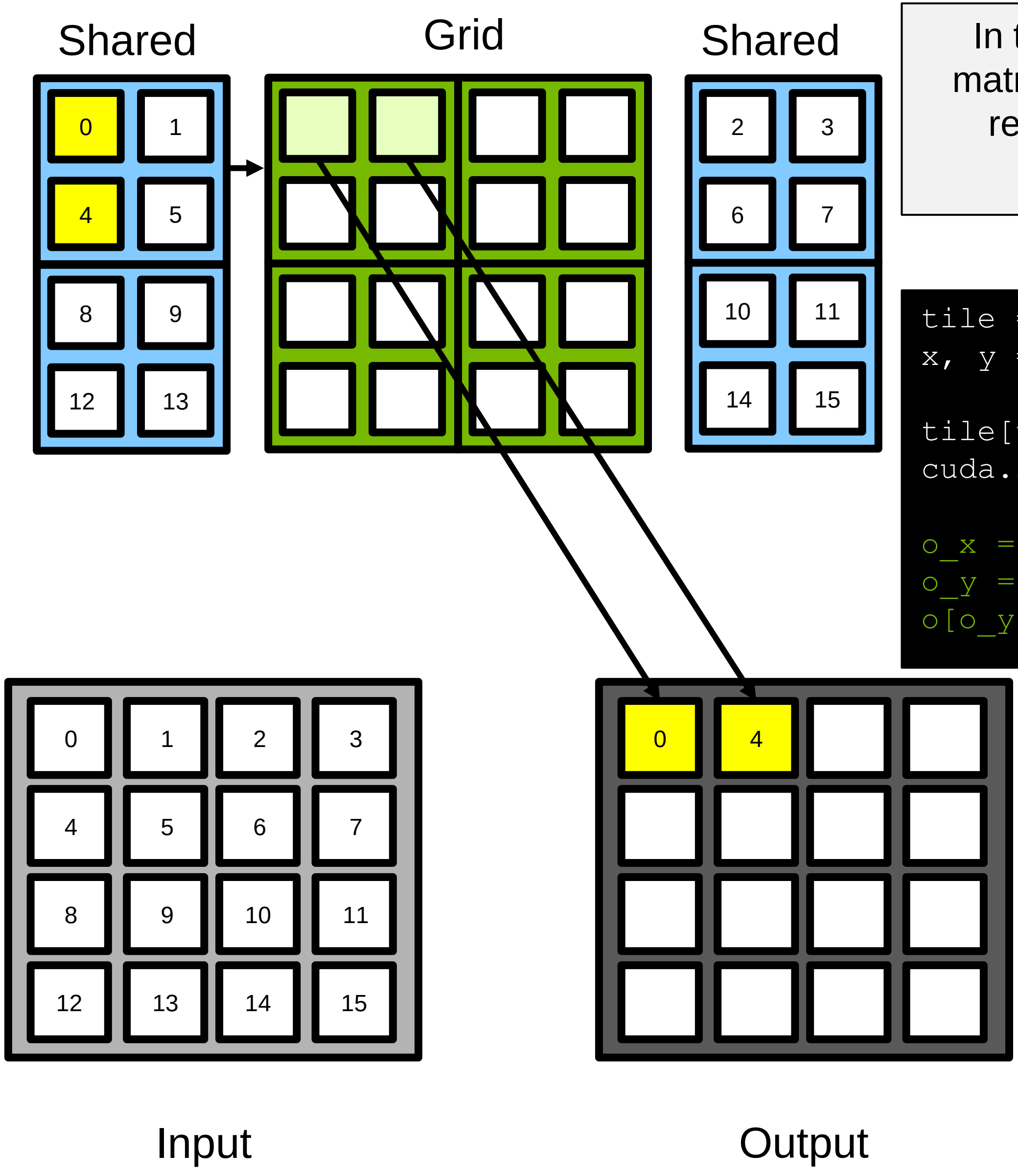




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

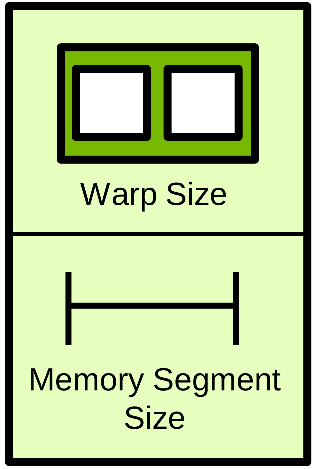
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

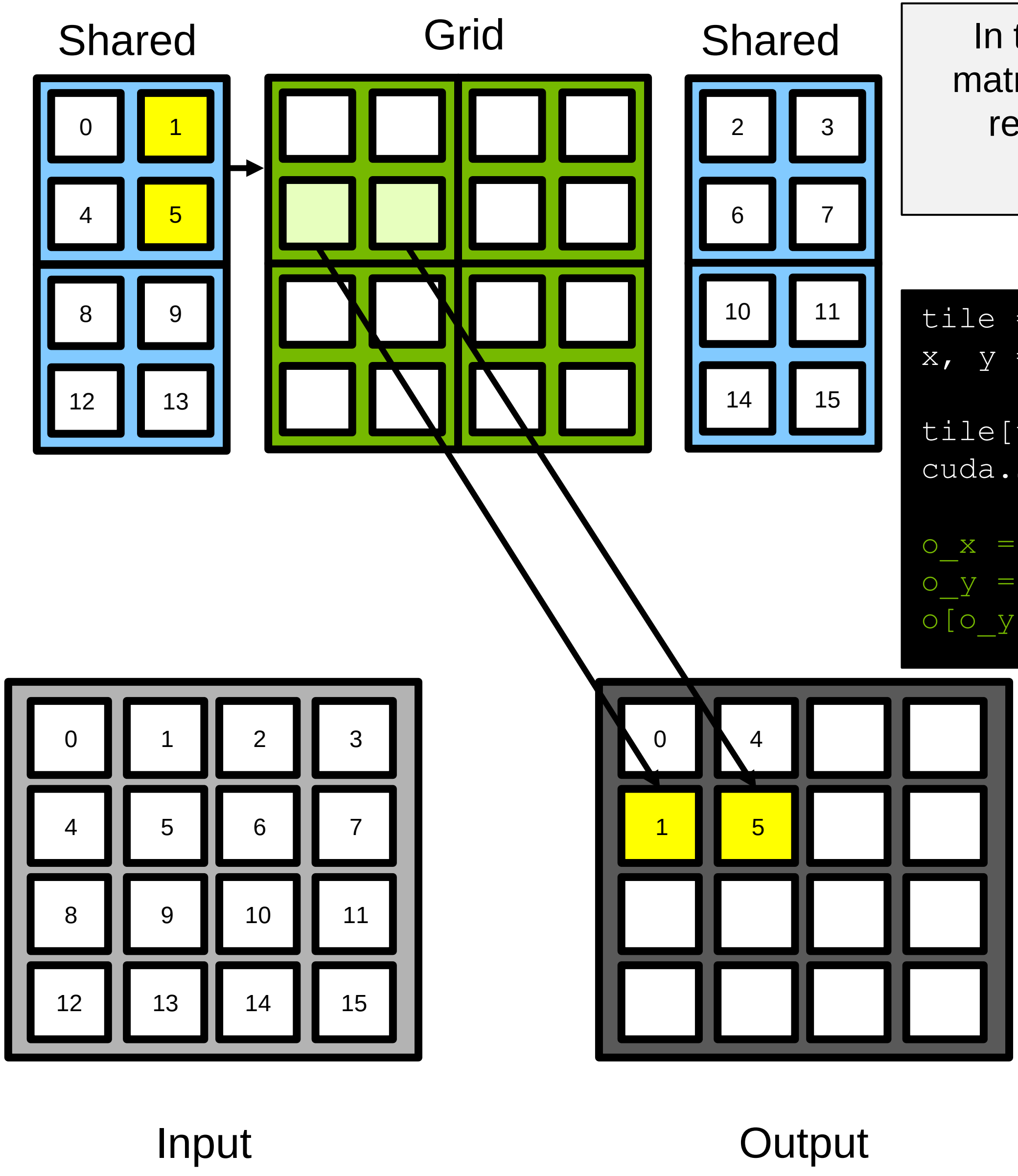




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

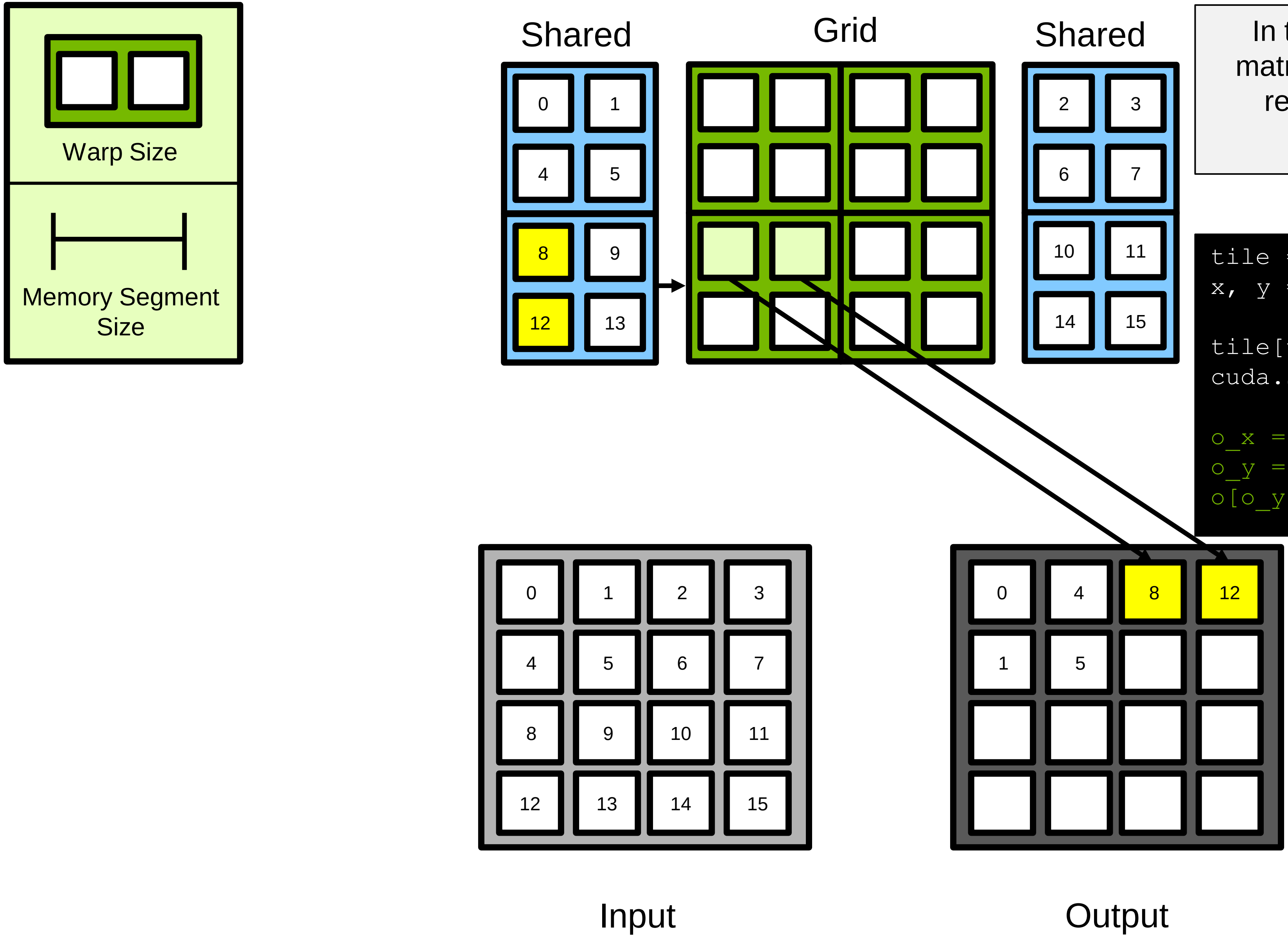
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```





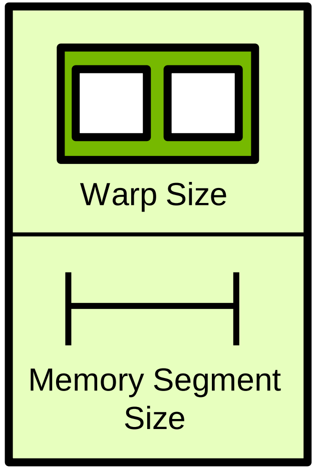
In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

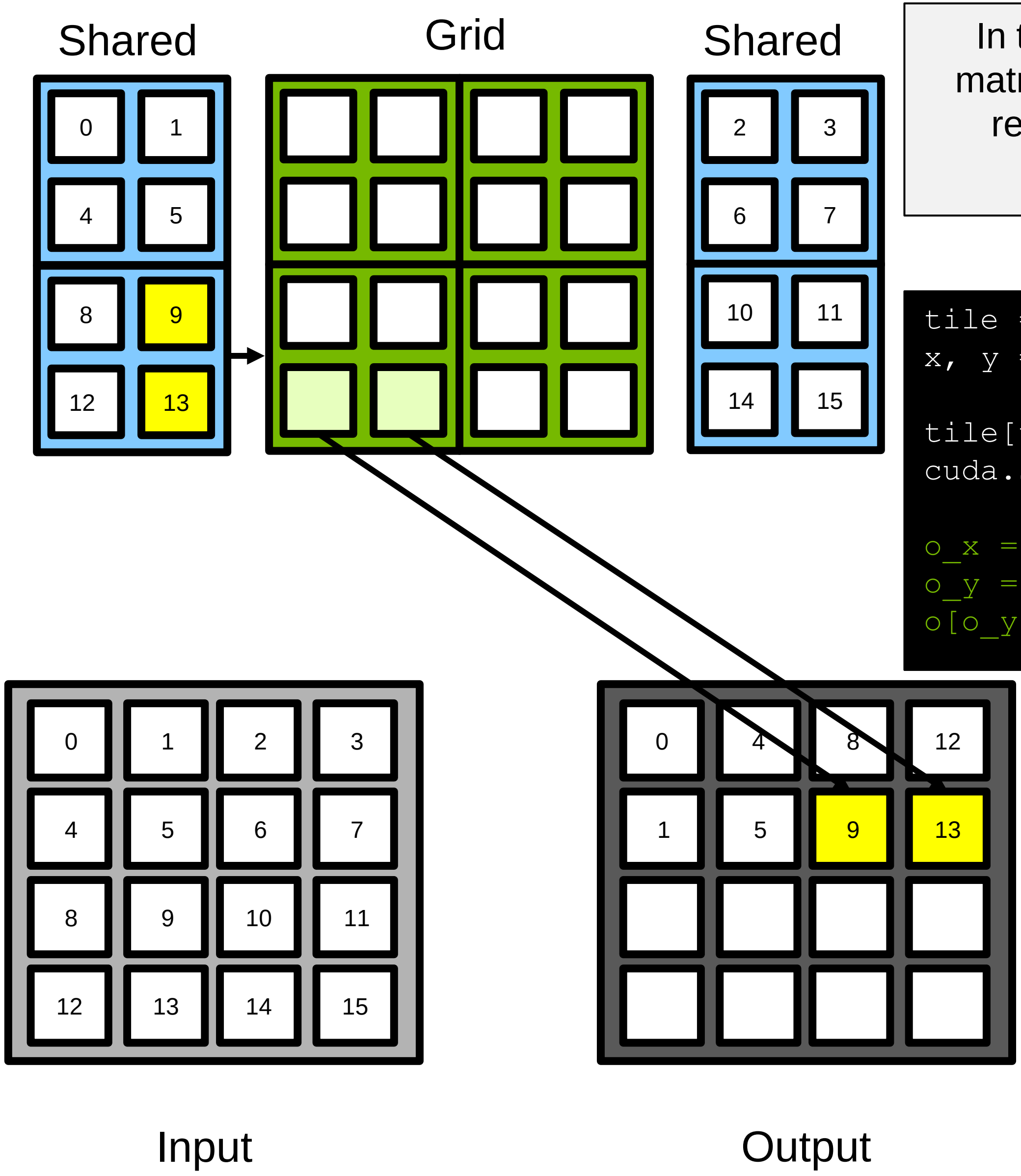
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```



In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

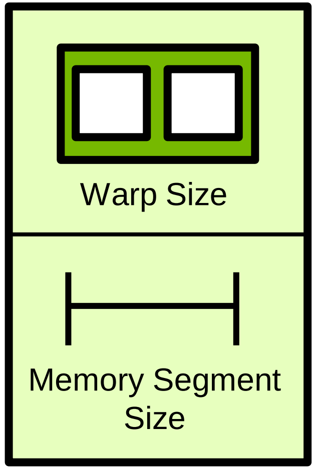
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

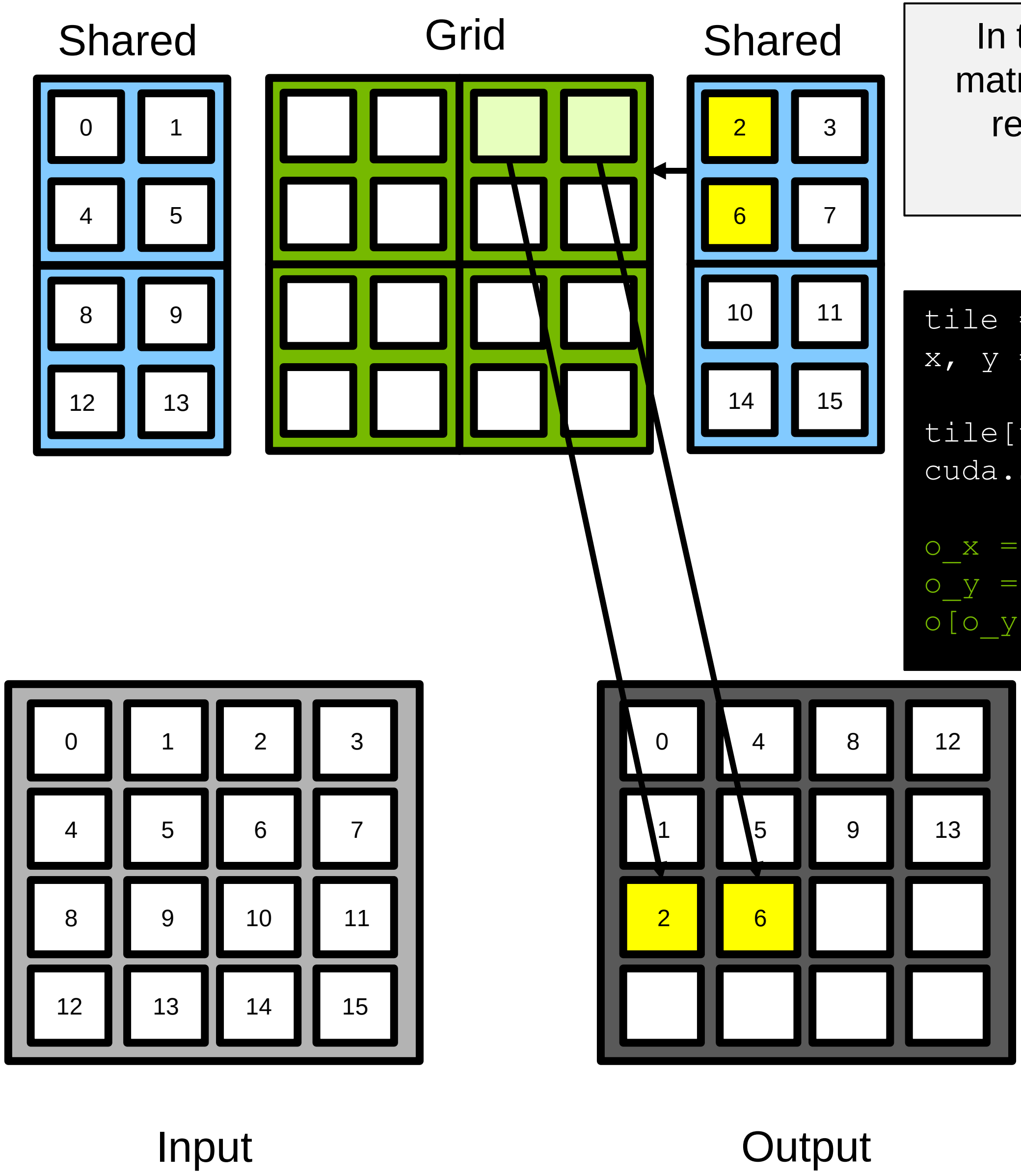




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

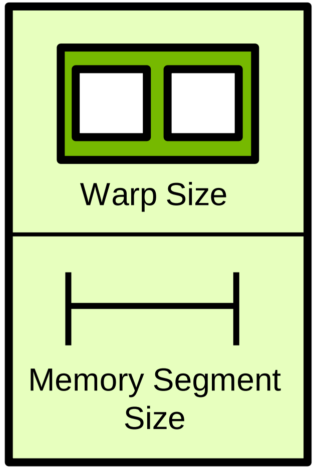
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

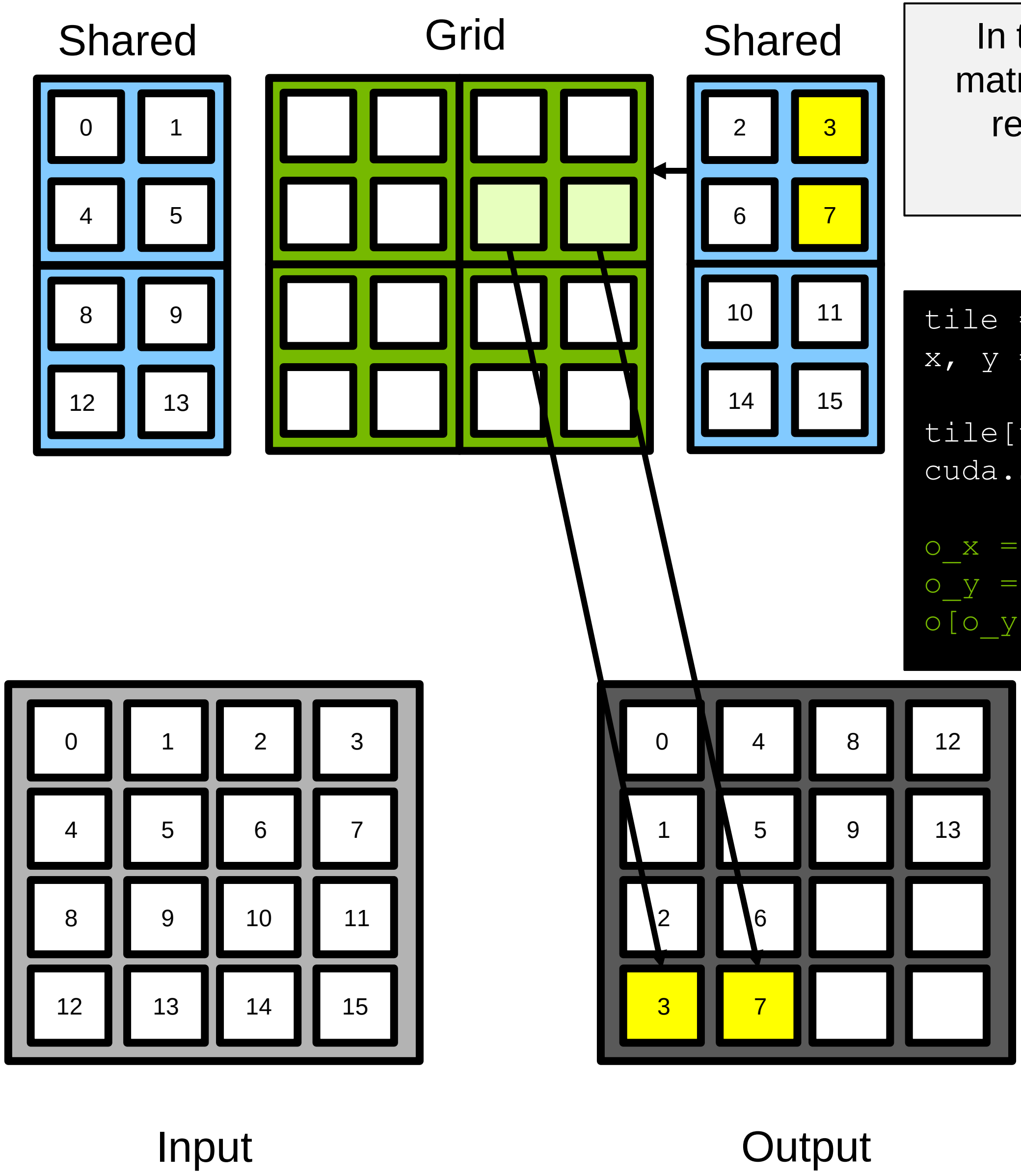




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

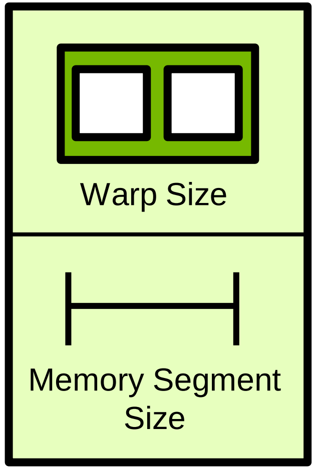
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

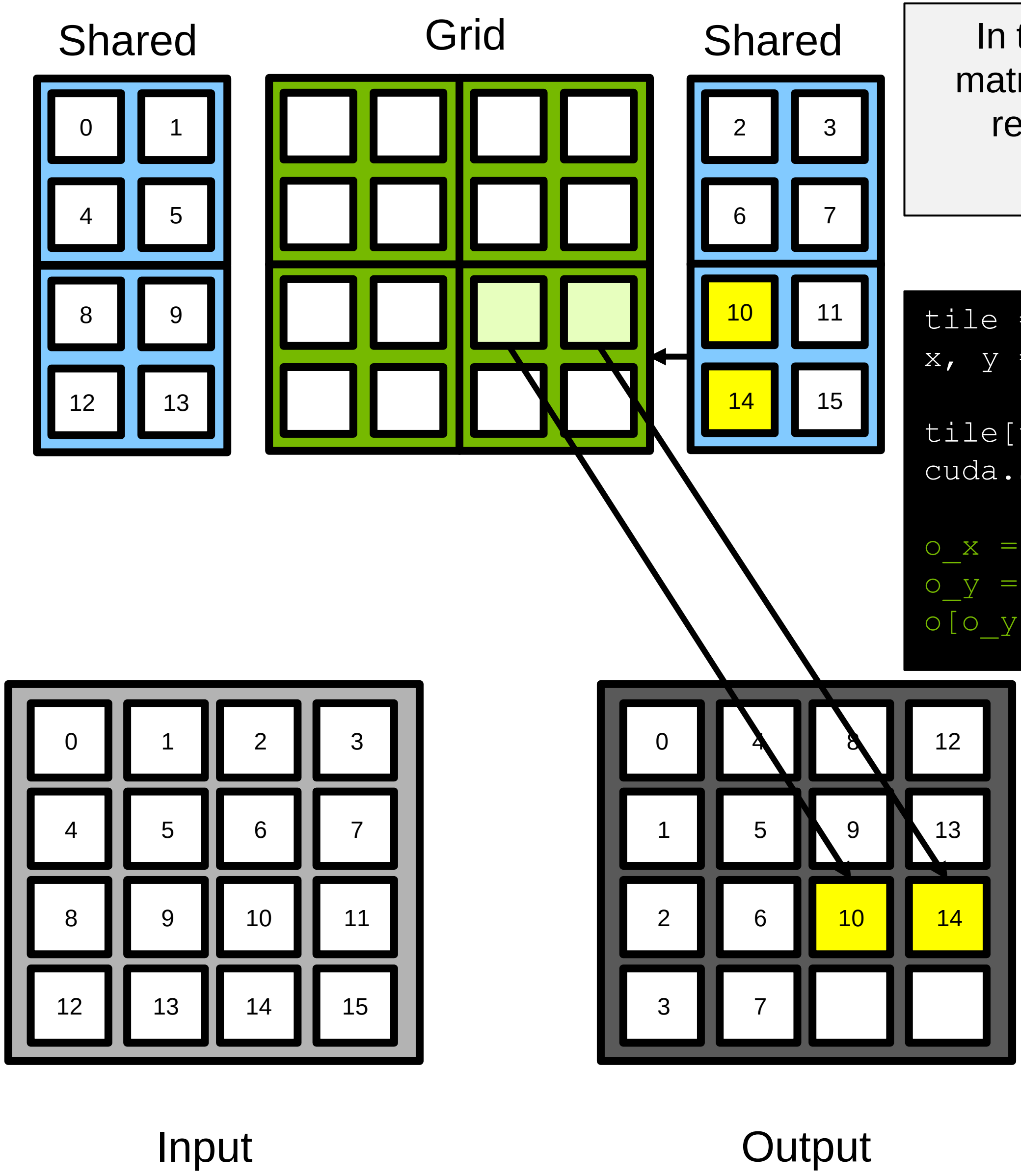




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

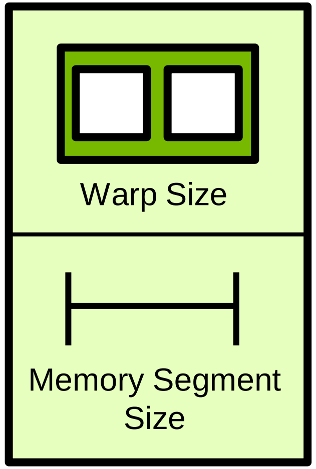
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

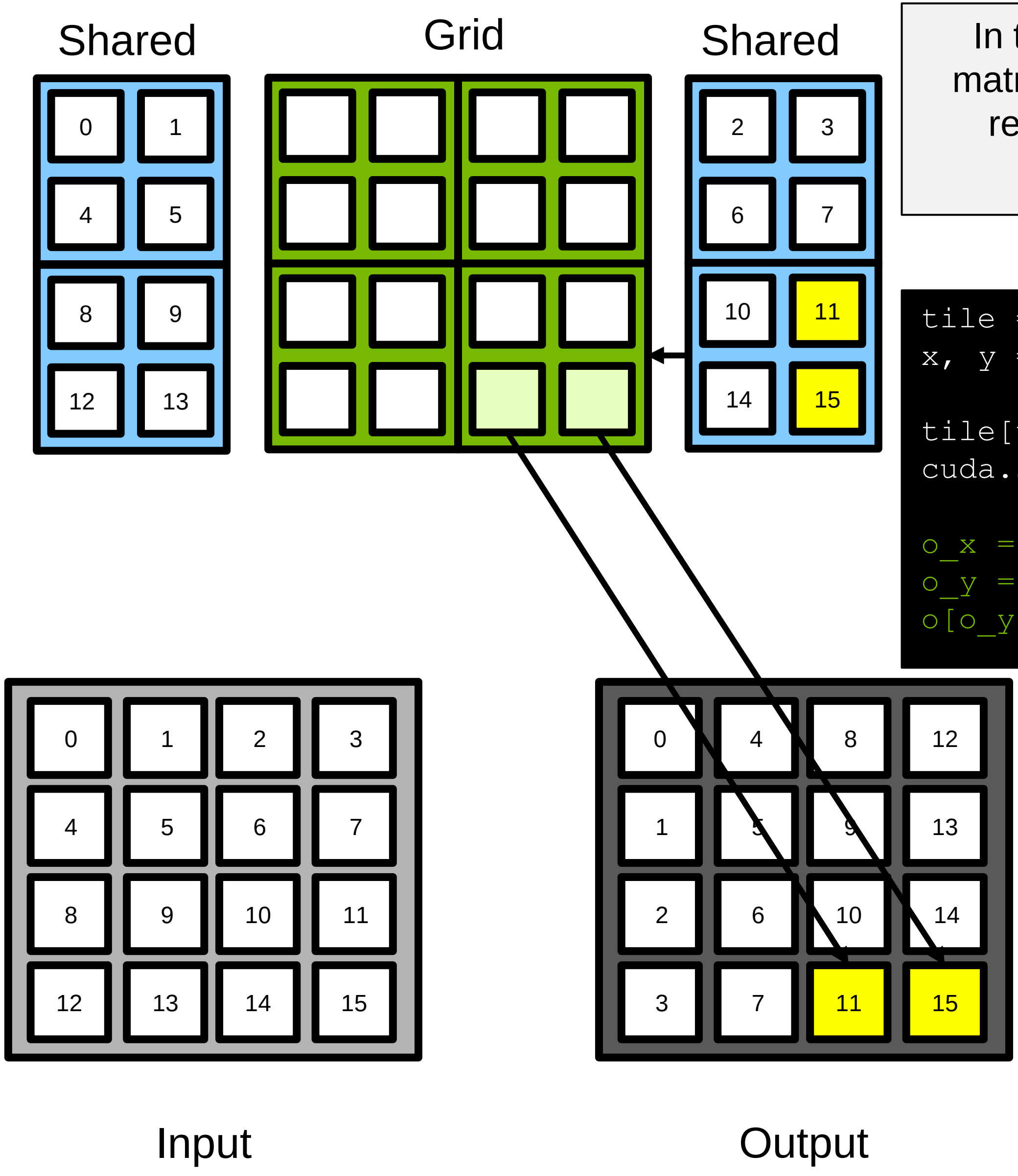




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

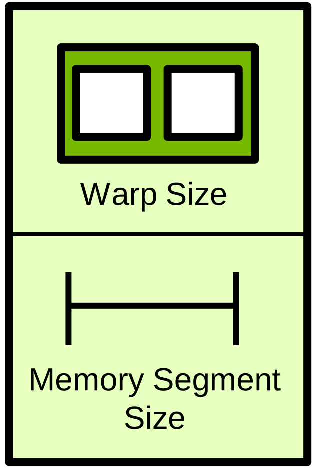
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

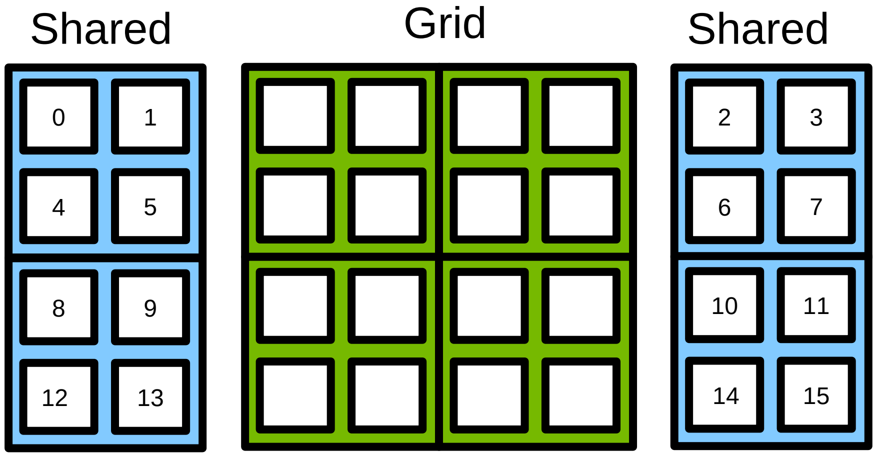




In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

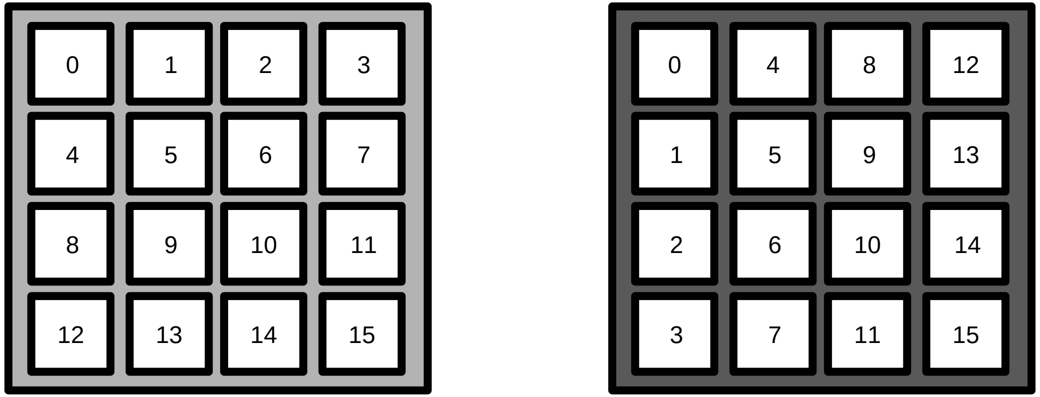
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```





In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

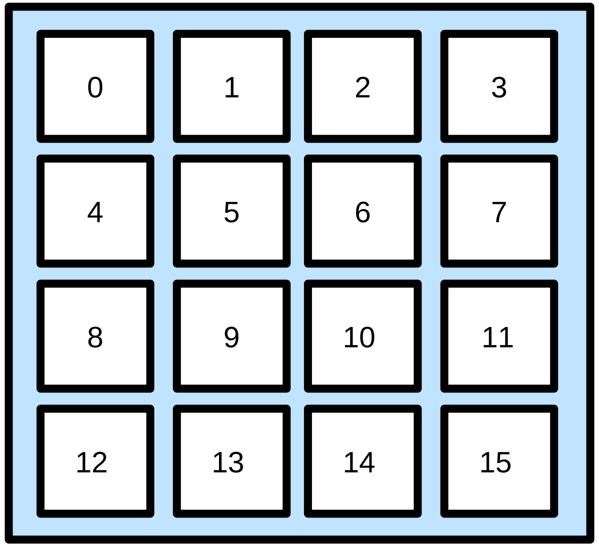


Input

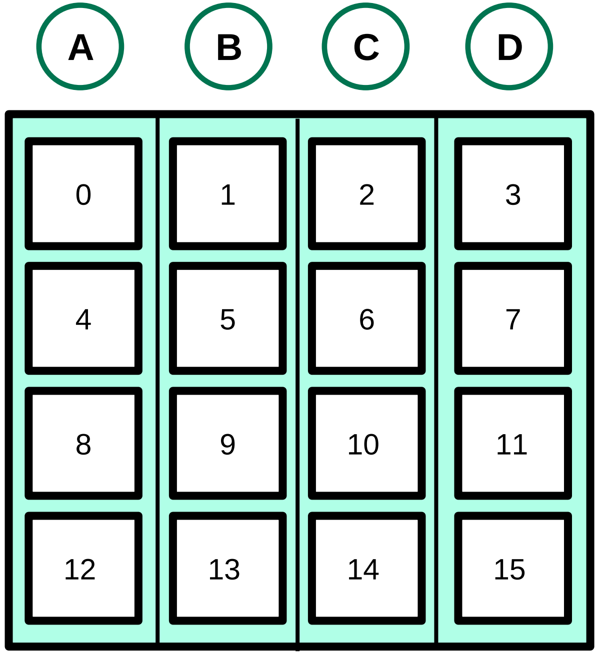
Output

Shared Memory Bank Conflicts

Shared memory is physically stored in **banks**

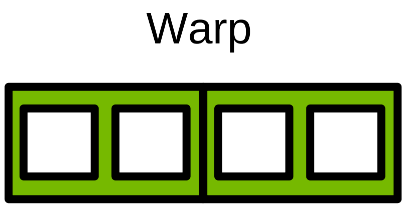


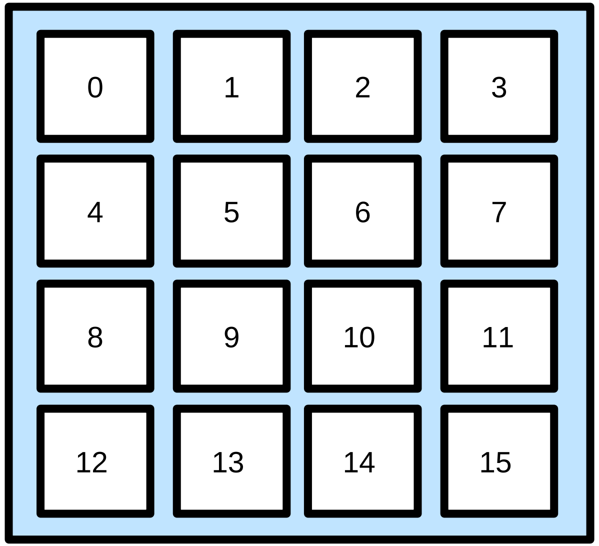
Logical Shared Memory
`cuda.shared.array(4,4)`



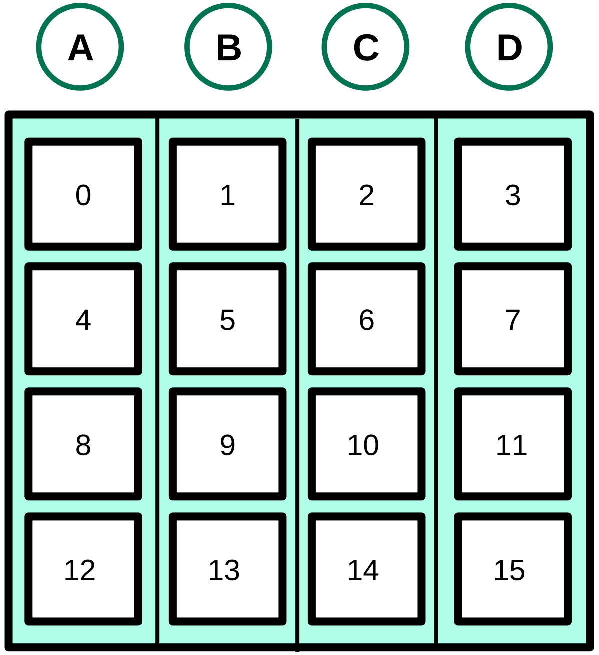
Physical Shared Memory
in 4 banks

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

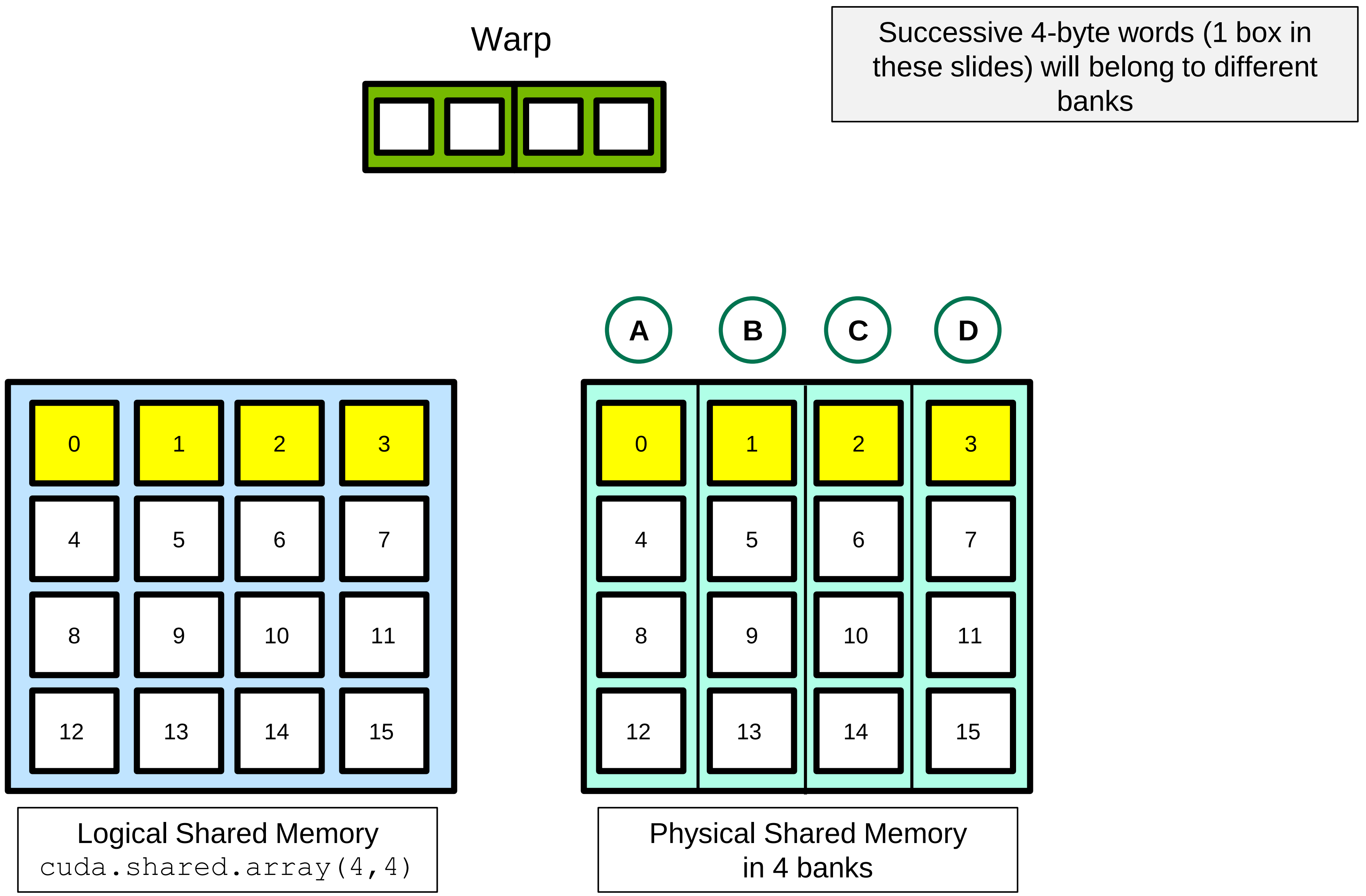


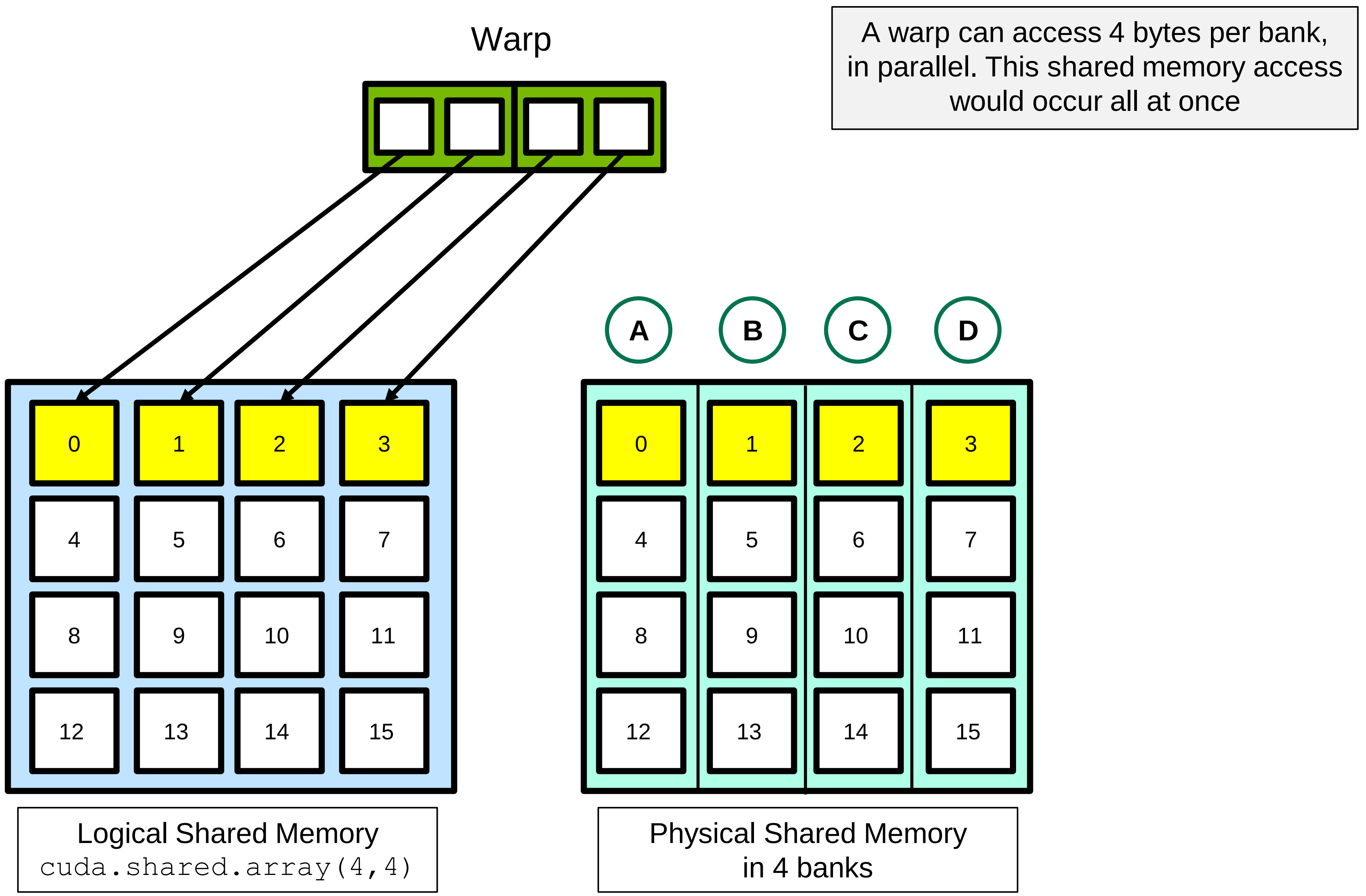


Logical Shared Memory
`cuda.shared.array(4,4)`

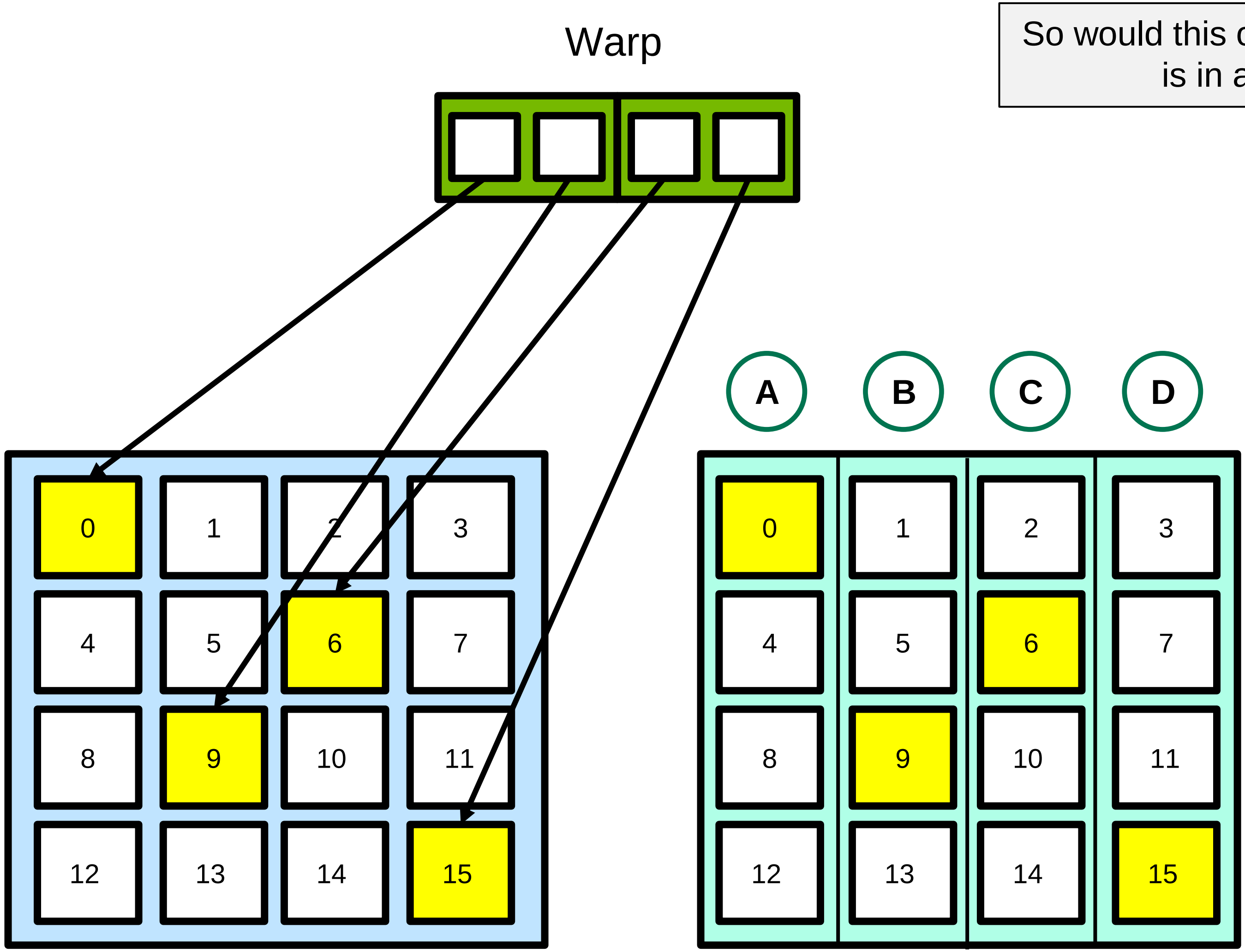


Physical Shared Memory
in 4 banks


Warp
Successive 4-byte words (1 box in these slides) will belong to different banks
A
B
C
D
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
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
A
B
C
D
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
Logical Shared Memory
cuda.shared.array(4,4)
Physical Shared Memory in 4 banks

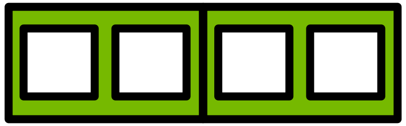
So would this one, since each element is in a different bank



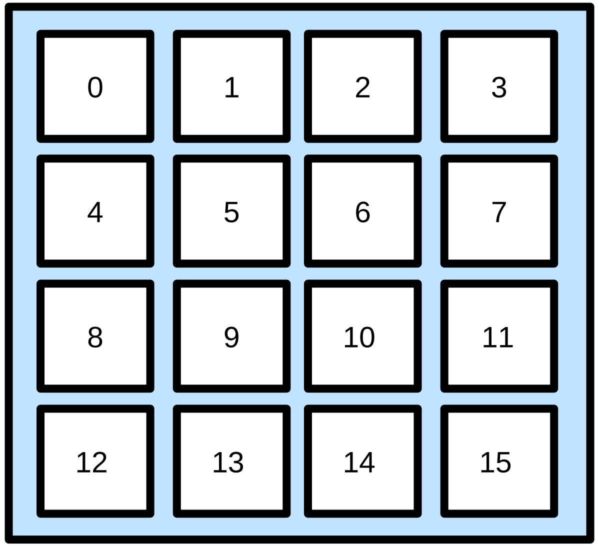
Logical Shared Memory
`cuda.shared.array(4,4)`

Physical Shared Memory
in 4 banks

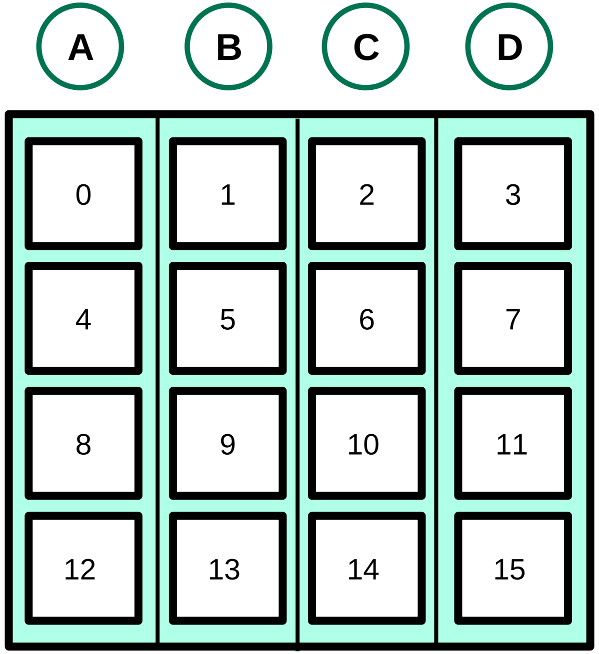
Warp



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

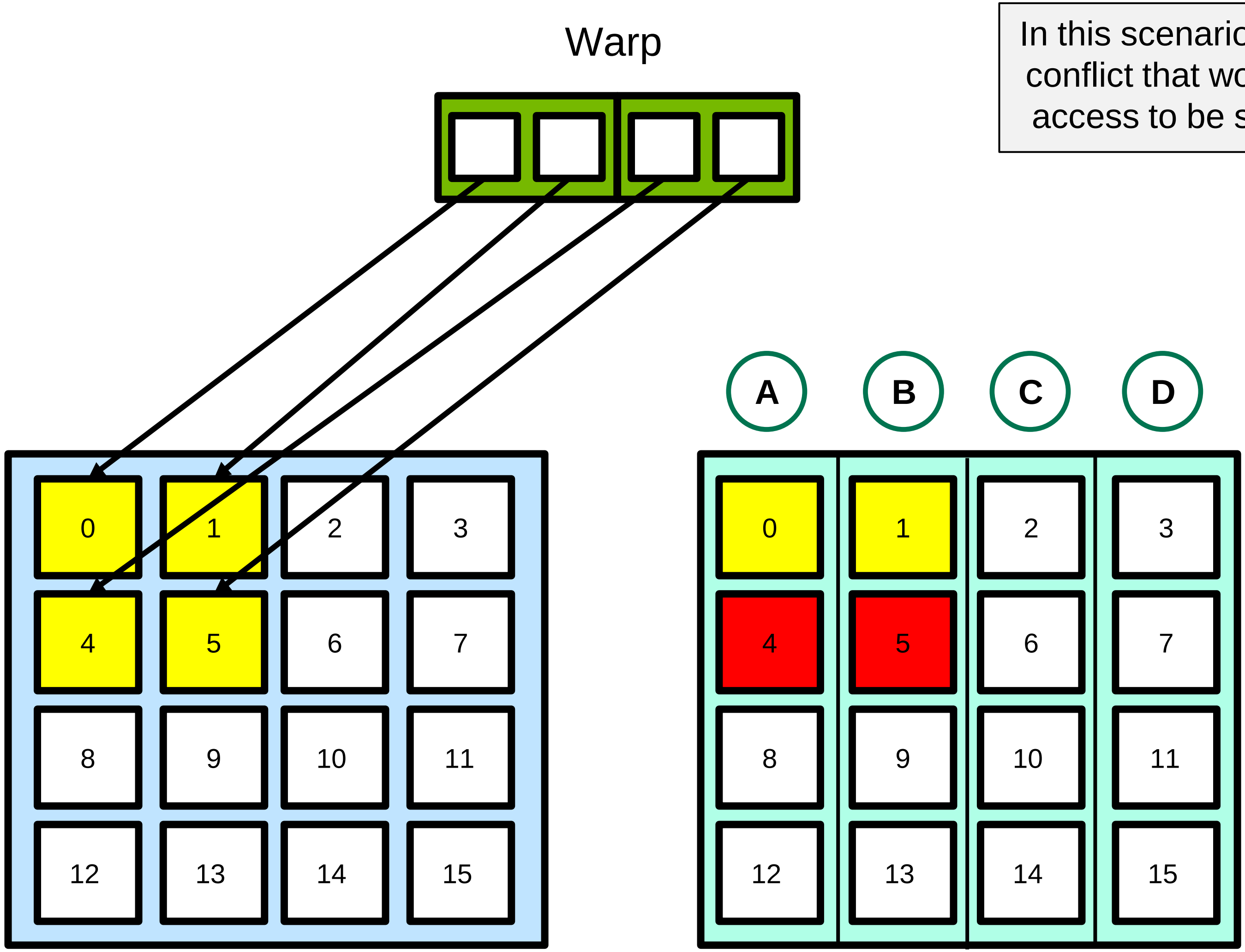


Logical Shared Memory
`cuda.shared.array(4,4)`



Physical Shared Memory
in 4 banks

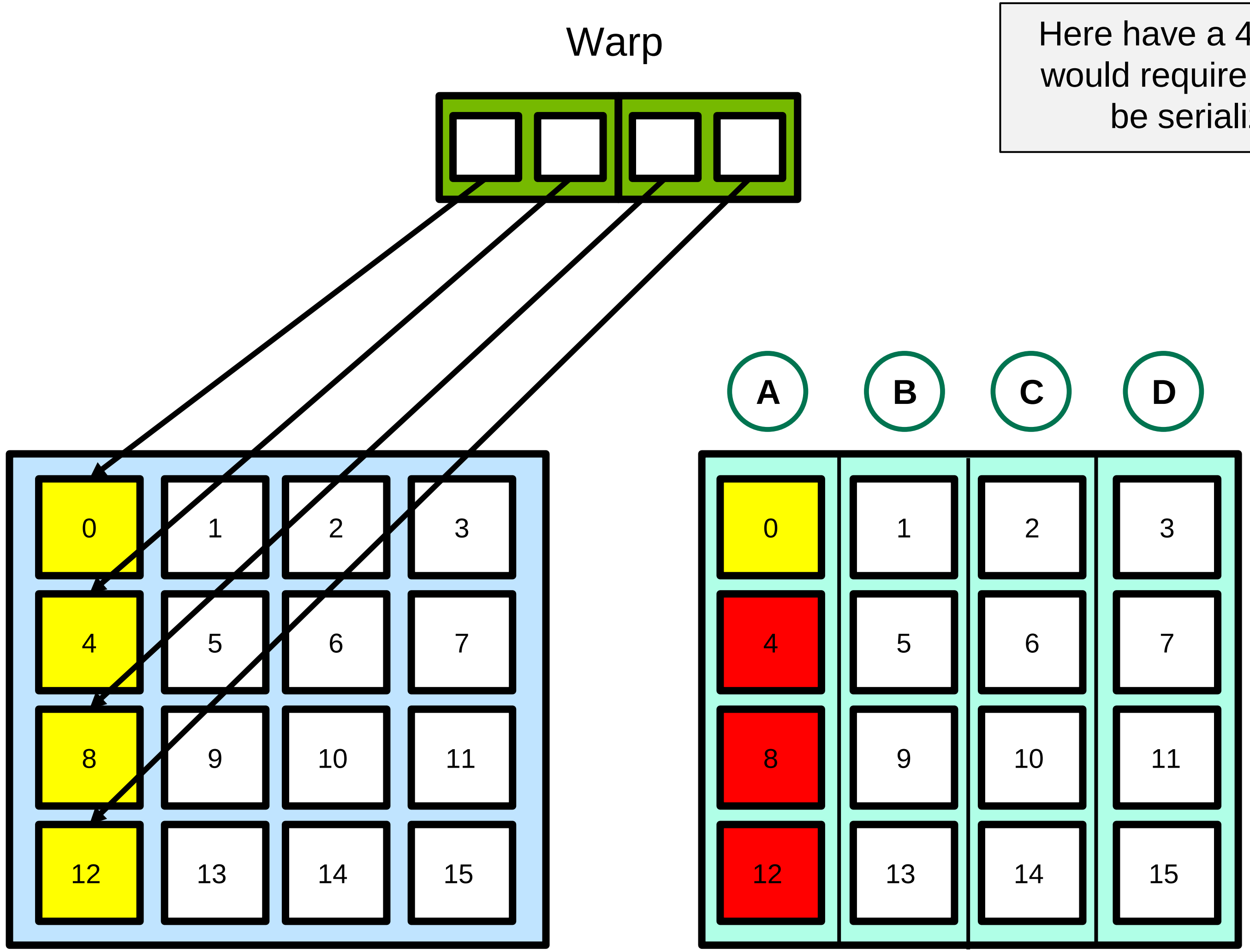
In this scenario, we have a 2-way bank conflict that would require the memory access to be serialized over 2 cycles.



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

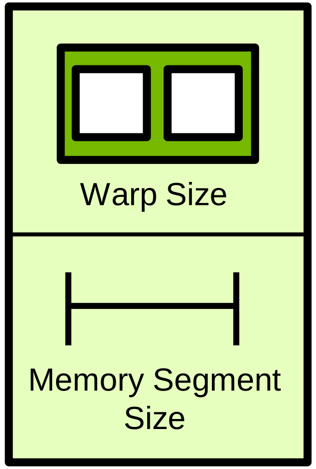
Physical Shared Memory
in 4 banks

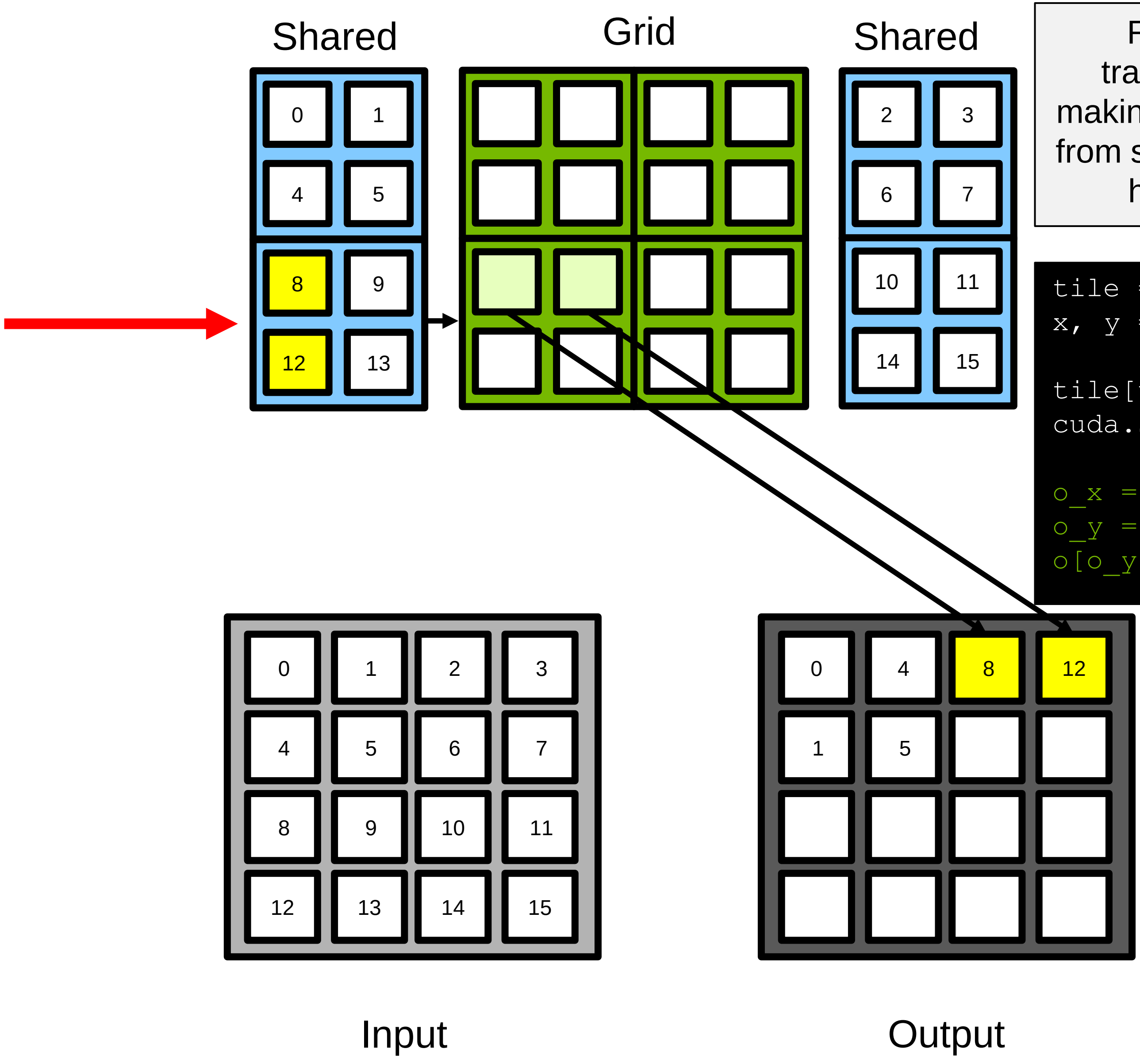
Here have a 4-way bank conflict that would require the memory access to be serialized over 4 cycles.



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

Physical Shared Memory
in 4 banks





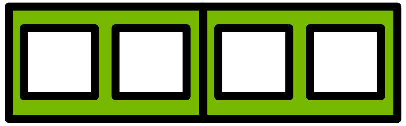
Recall from our earlier matrix transpose example that we were making this very kind of columnar read from shared memory, which means we had significant bank conflicts

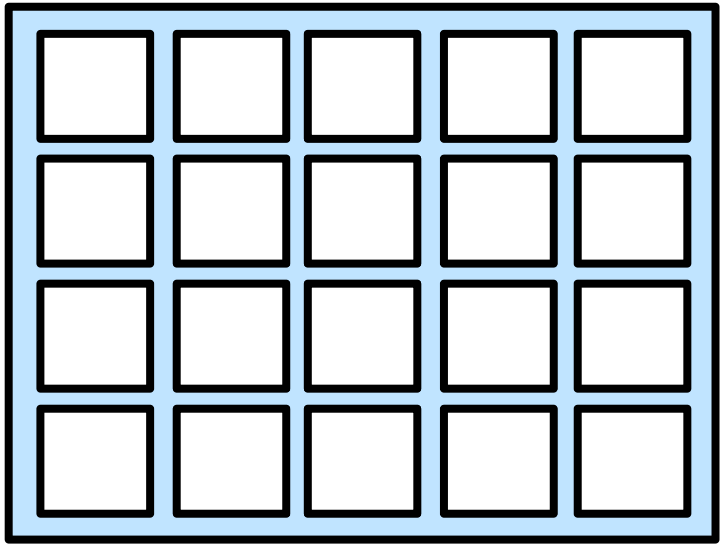
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

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

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

Warp

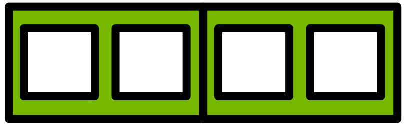


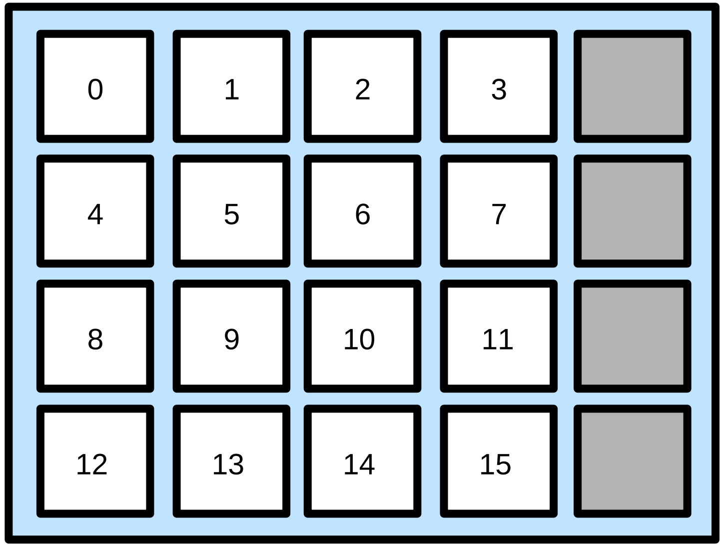


Logical Shared Memory
`cuda.shared.array(4,5)`

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]

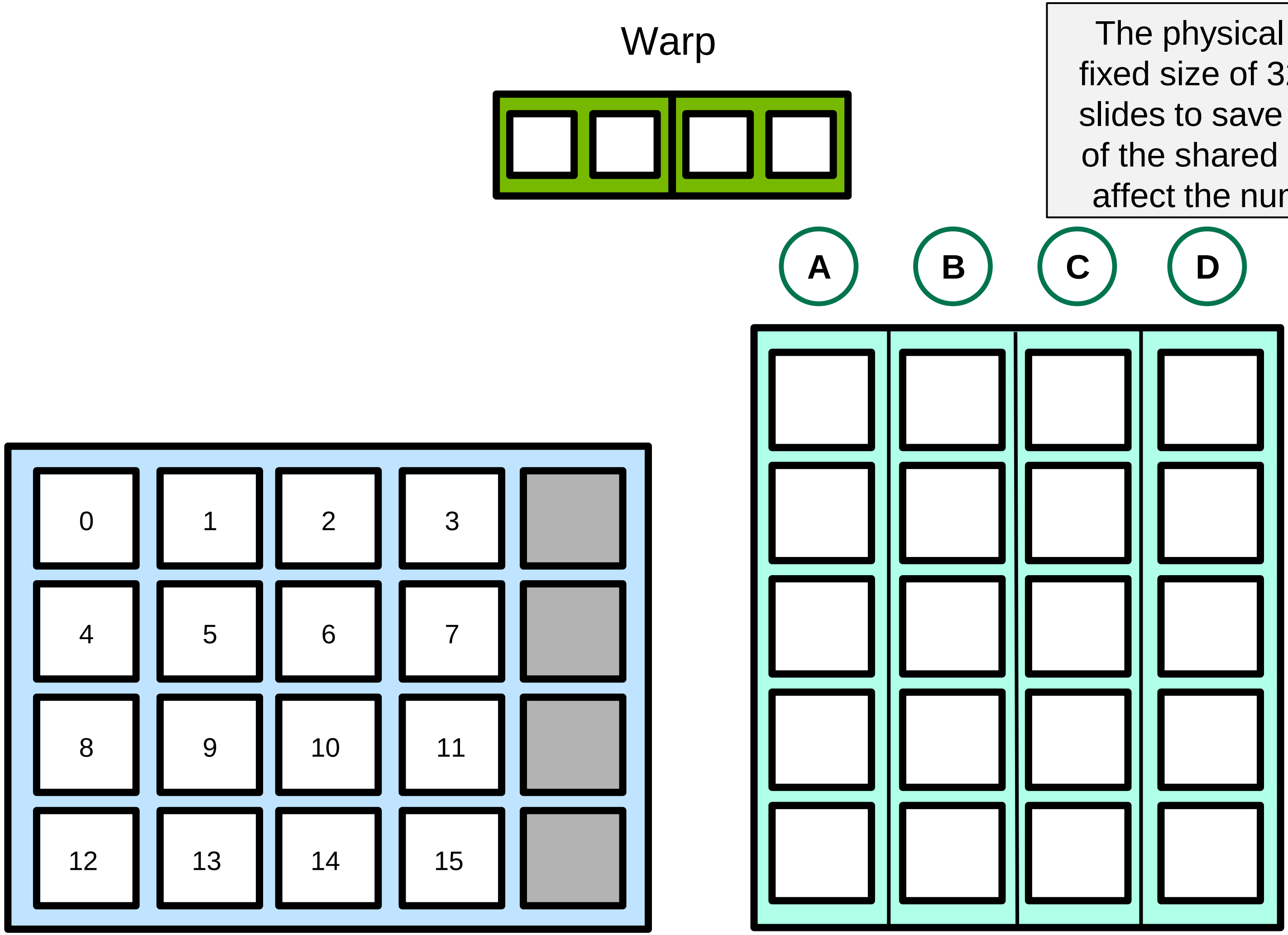
Warp





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

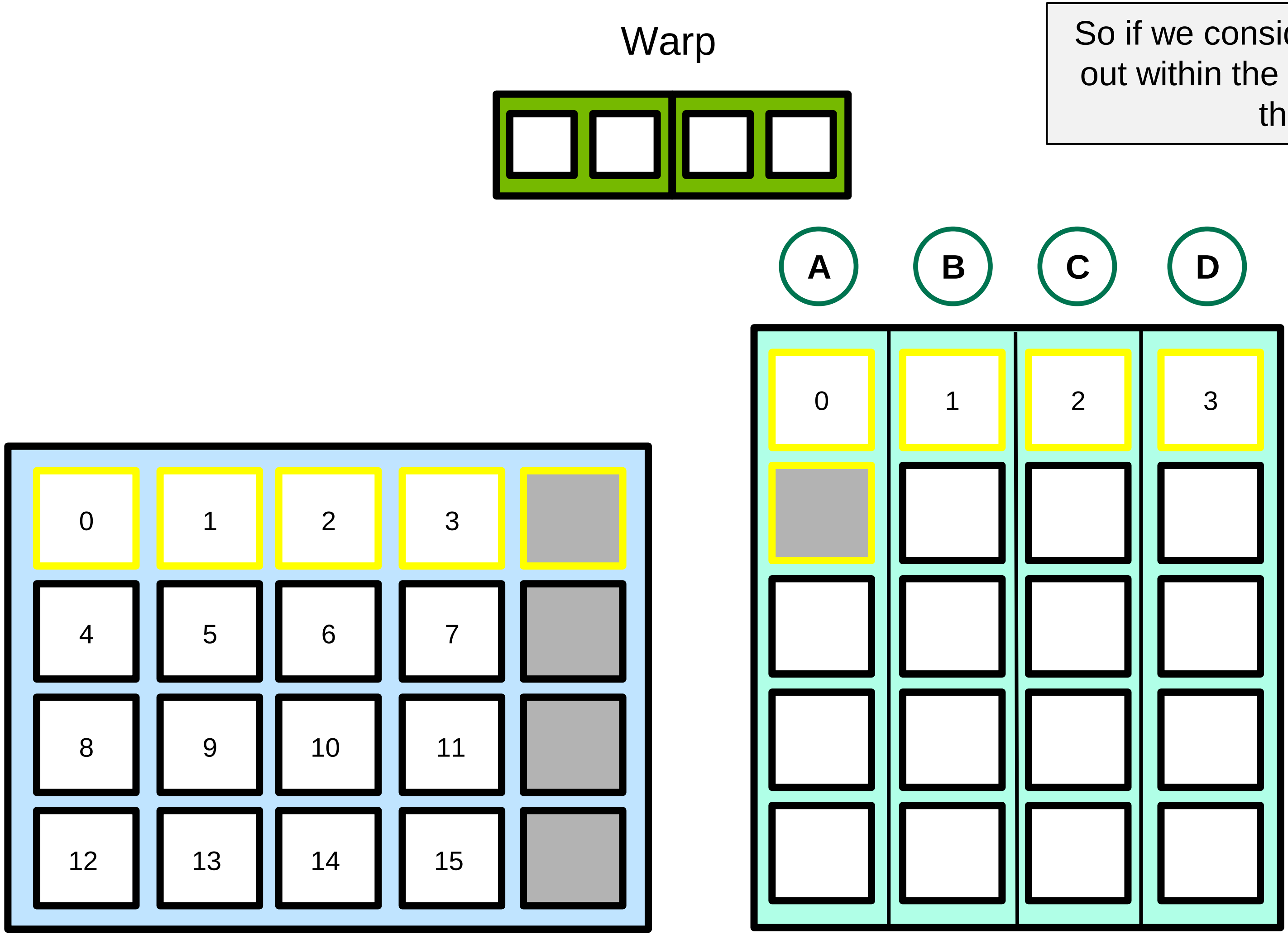
The physical shared memory has a fixed size of 32 banks (4 banks in our slides to save space), so our padding of the shared memory array does not affect the number of memory banks



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

Physical Shared Memory
in 4 banks

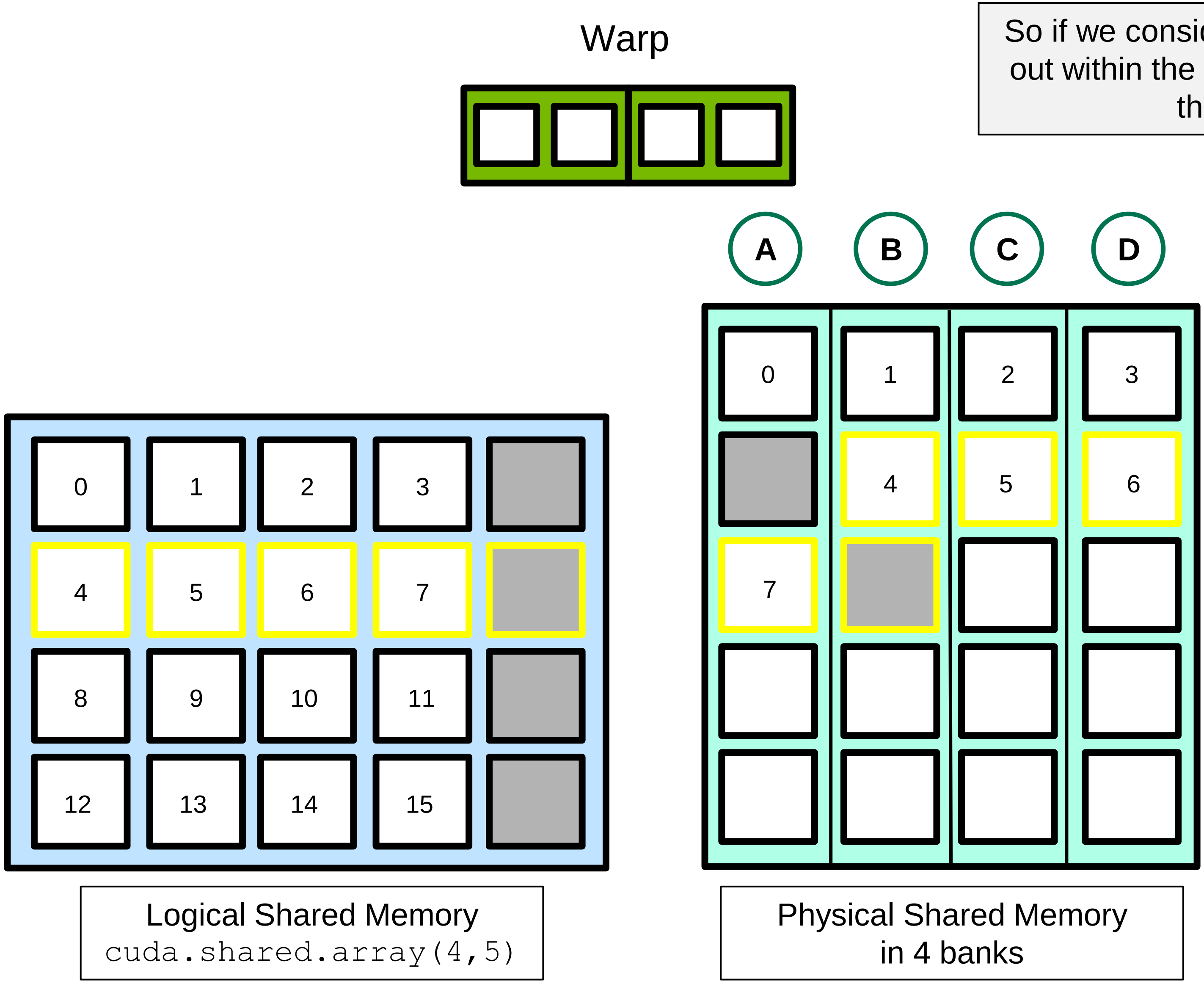
So if we consider how the array is laid out within the memory banks, we see the following:



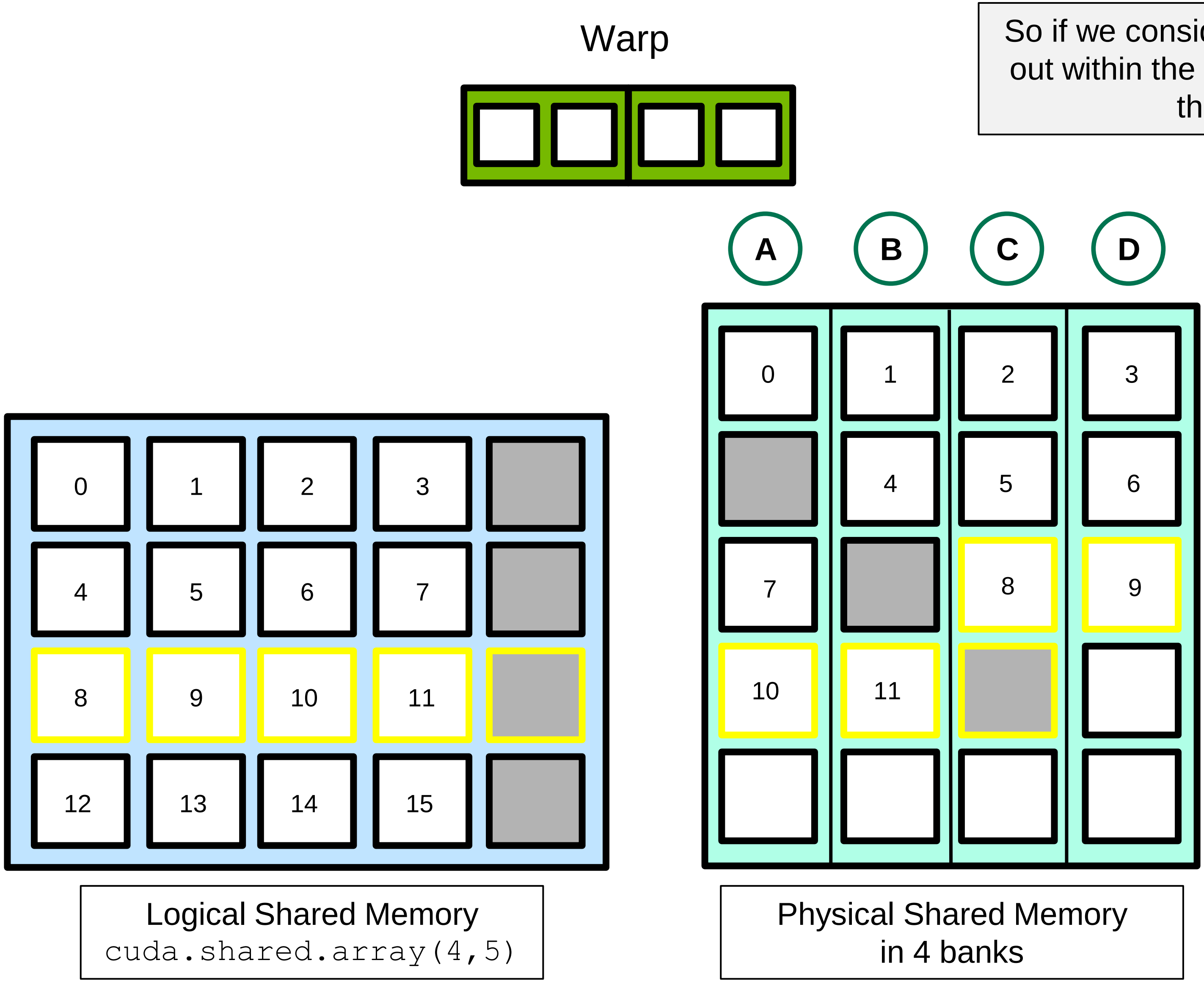
Logical Shared Memory
`cuda.shared.array(4,5)`

Physical Shared Memory
in 4 banks

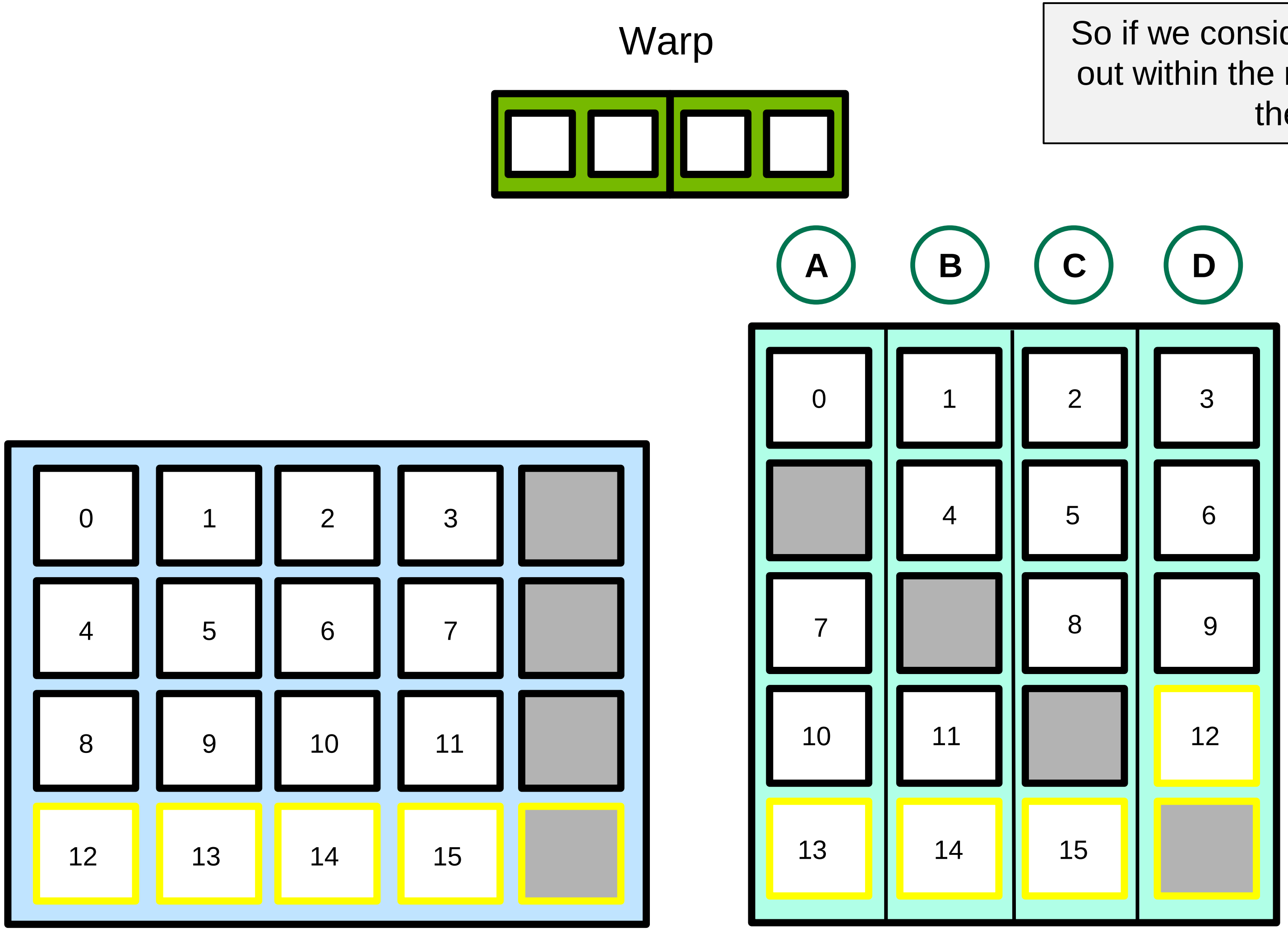
So if we consider how the array is laid out within the memory banks, we see the following:



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



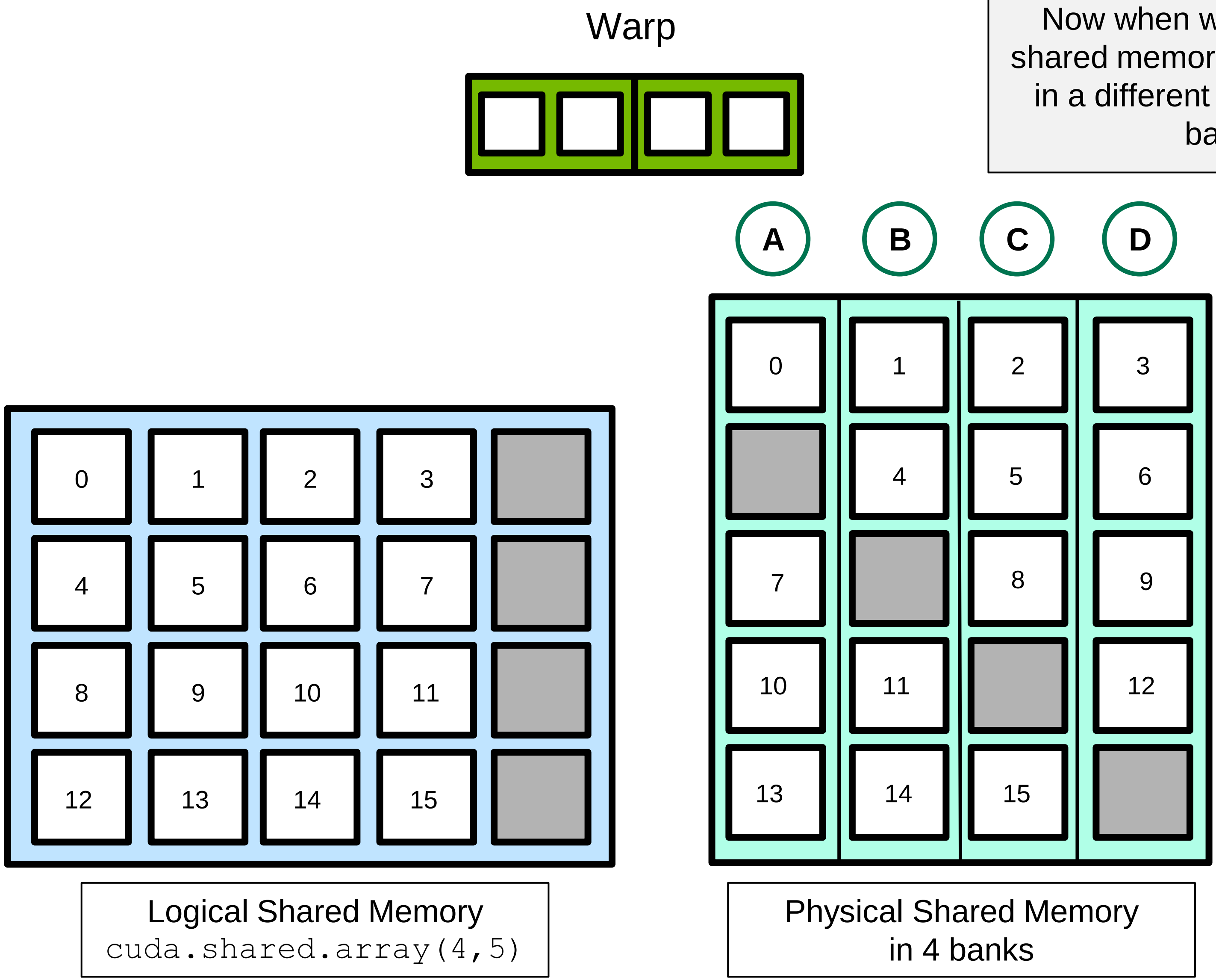
So if we consider how the array is laid out within the memory banks, we see the following:



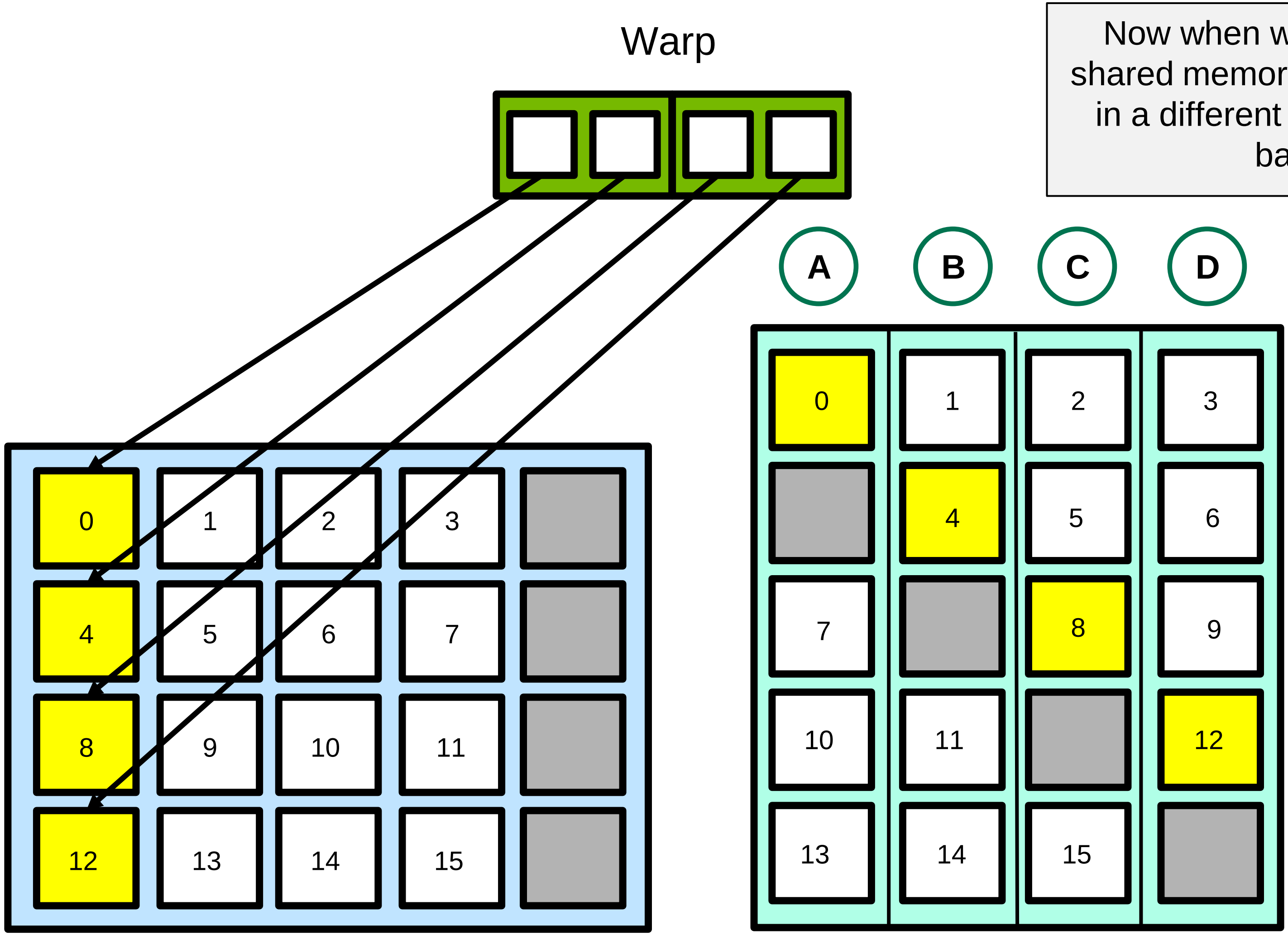
Logical Shared Memory
`cuda.shared.array(4,5)`

Physical Shared Memory
in 4 banks

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



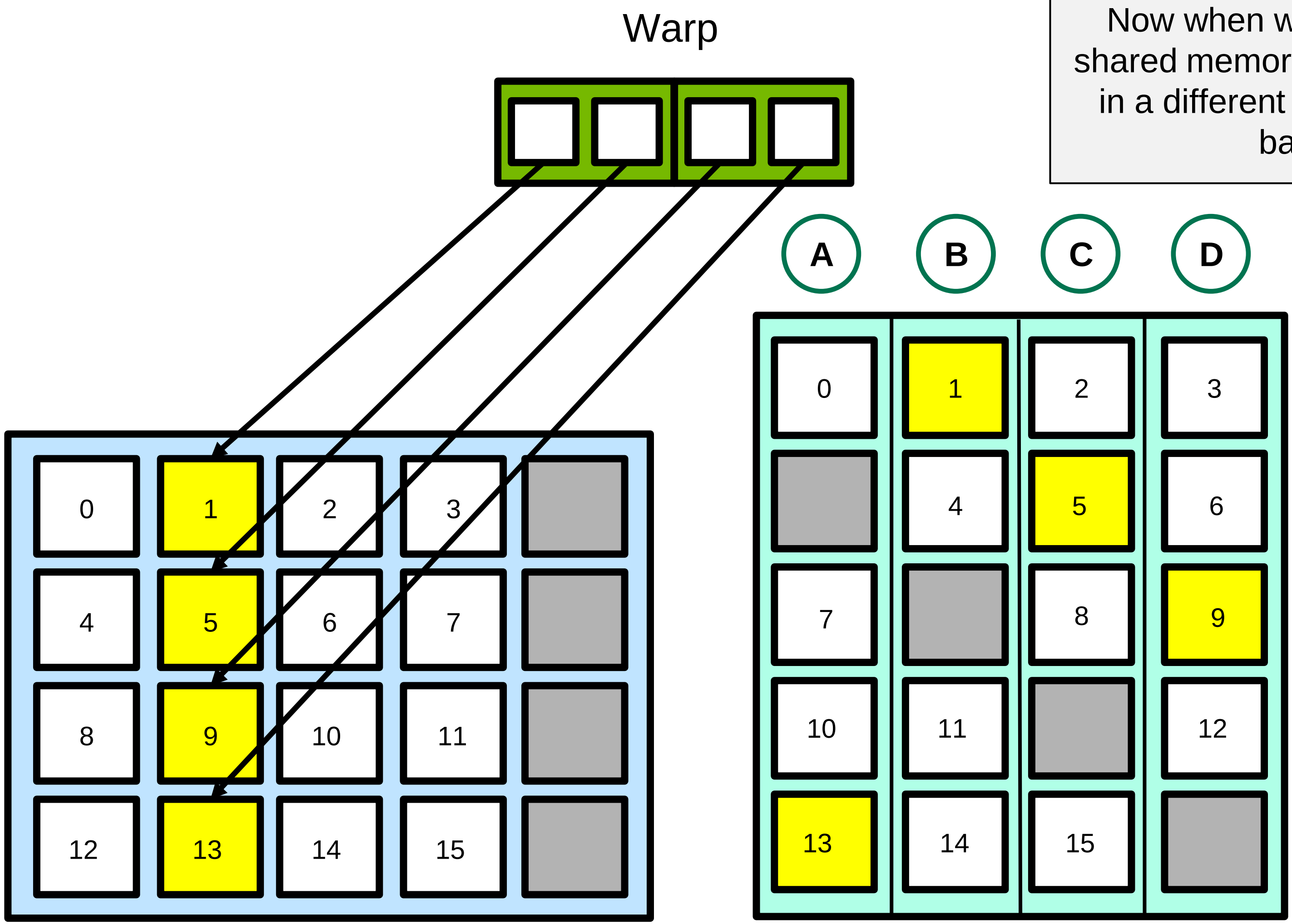
Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



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

Physical Shared Memory
in 4 banks

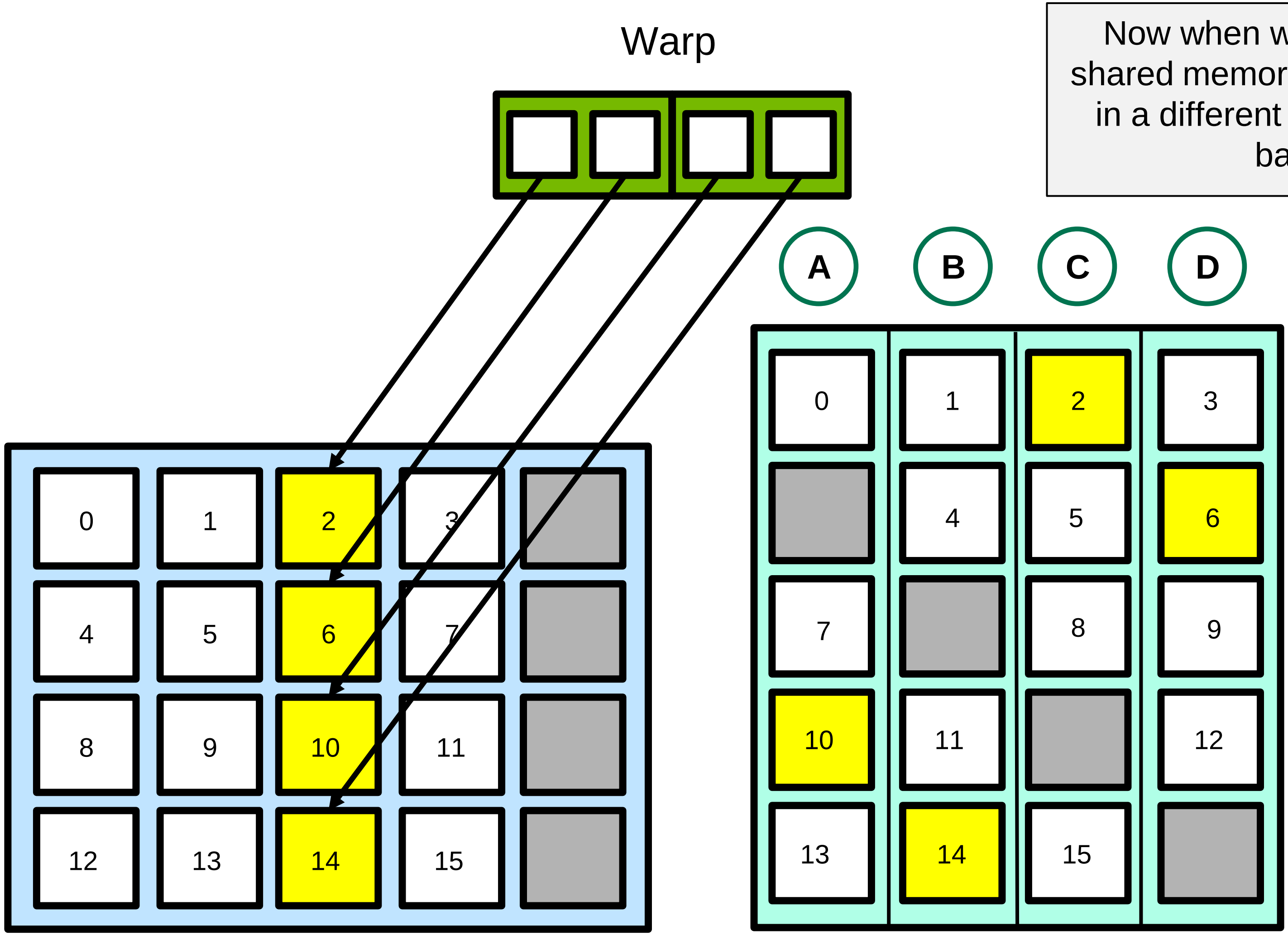
Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts



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

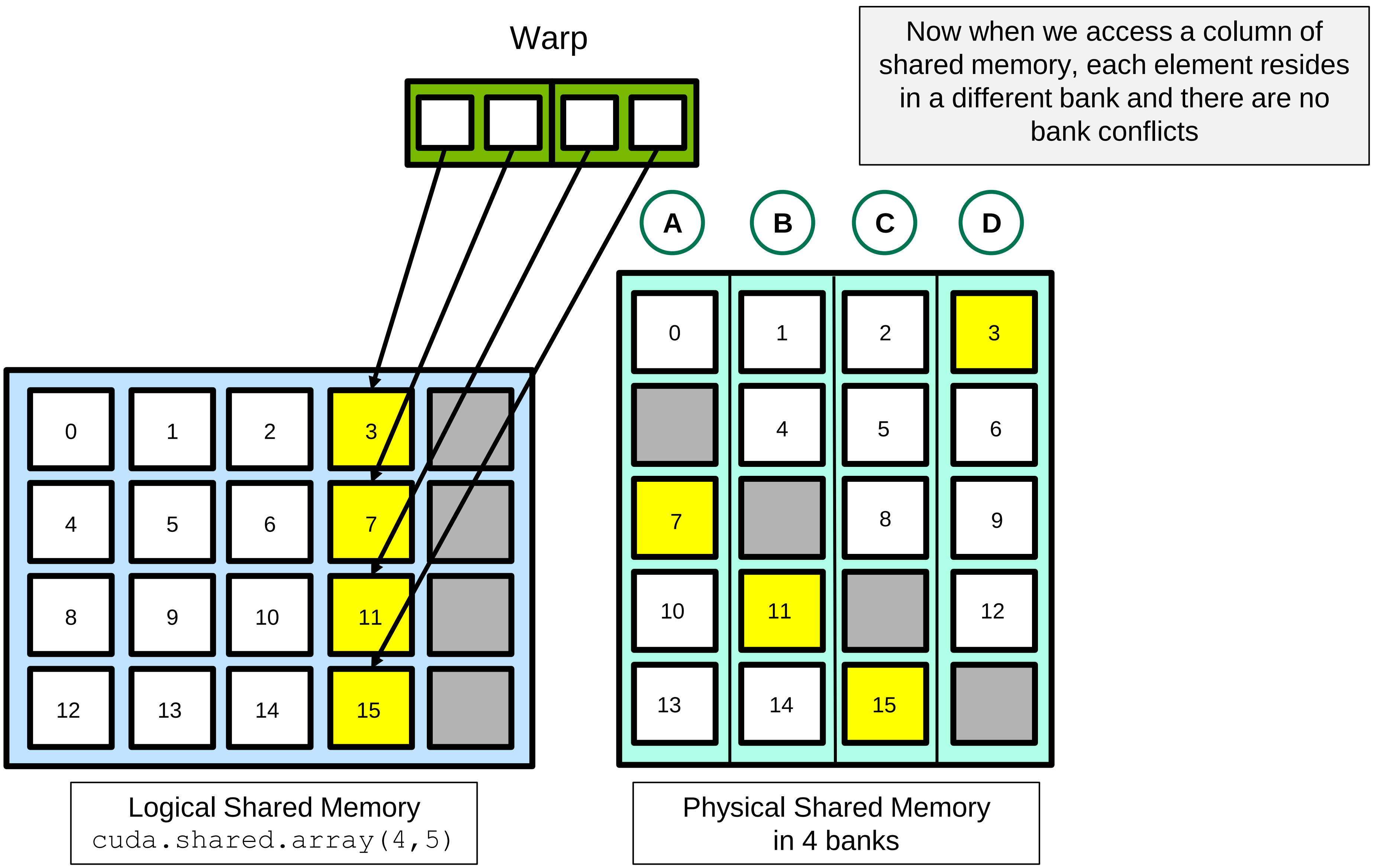
Physical Shared Memory
in 4 banks

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

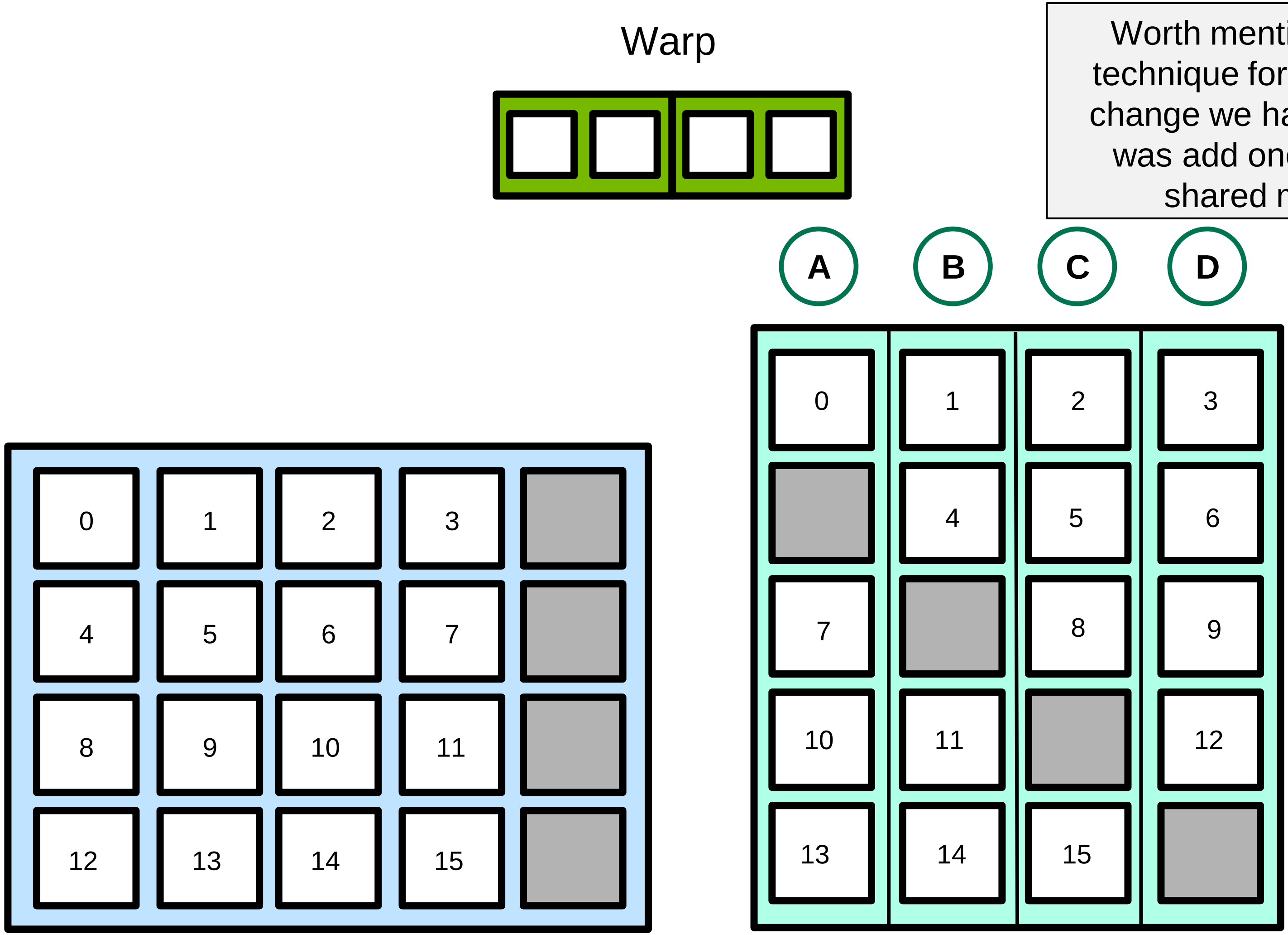


Logical Shared Memory
`cuda.shared.array(4,5)`

Physical Shared Memory
in 4 banks

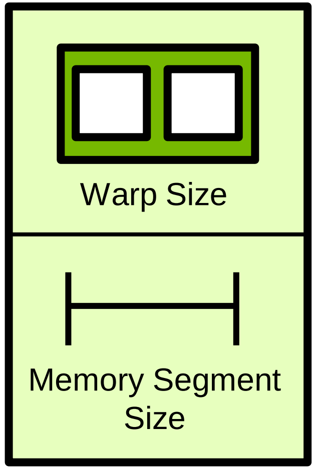

Warp
Now when we access a column of shared memory, each element resides in a different bank and there are no bank conflicts
A
B
C
D
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Logical Shared Memory
cuda.shared.array(4,5)
Physical Shared Memory
in 4 banks

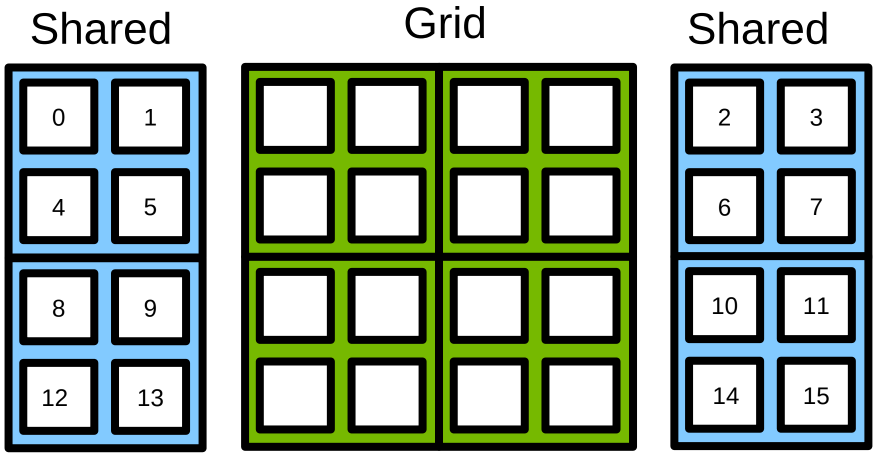
Worth mentioning that to use this technique for this example, the only change we had to make to our code was add one extra column to our shared memory allocation



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

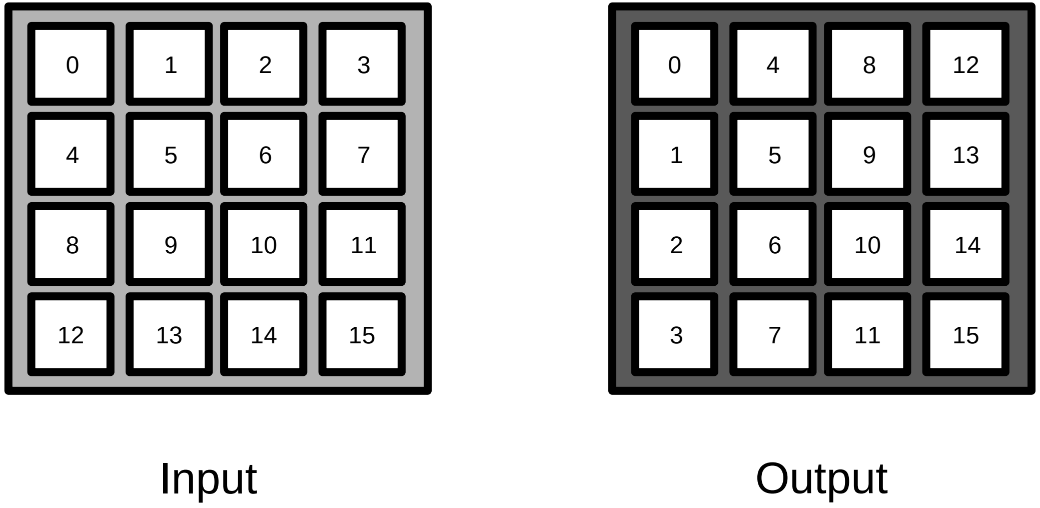
Physical Shared Memory
in 4 banks





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]
```



DEEP
LEARNING
INSTITUTE

www.nvidia.com/dli