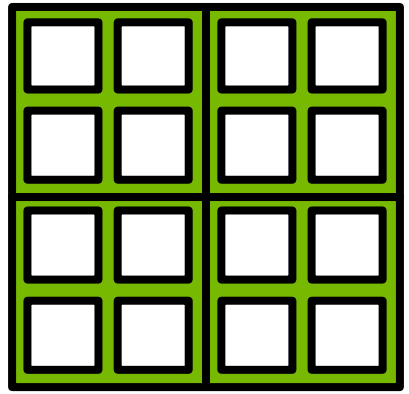
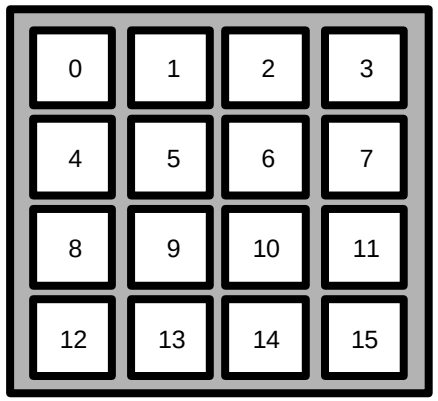
Using Shared Memory to Support Coalesced Memory Access

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

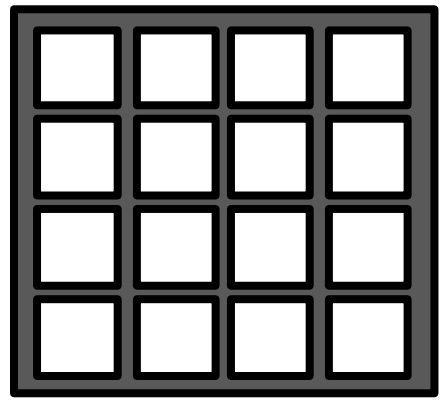
Grid



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

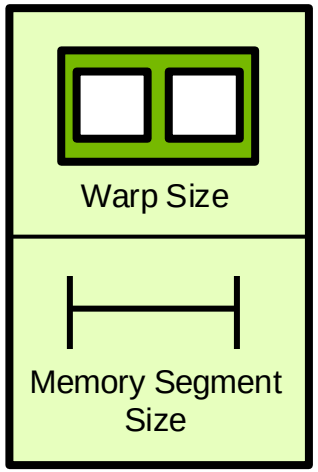


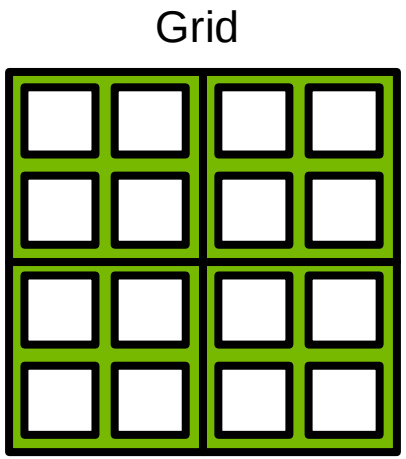
Input



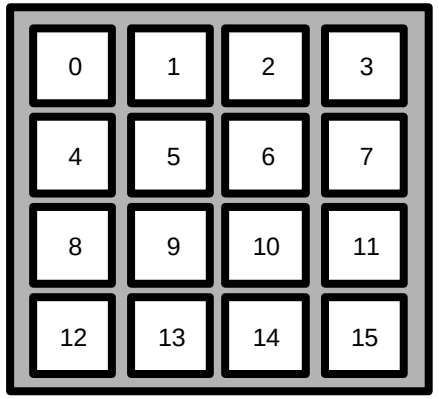
Output

NVIDIA DEEP LEARNING INSTITUTE

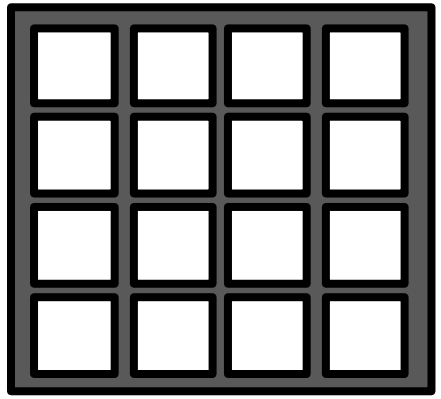




For these slides we will define a warp as 2 threads, and a memory segment as 2 data elements wide

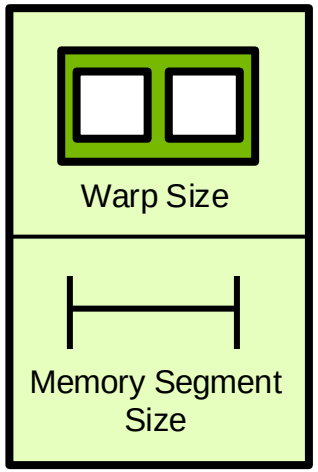


Input

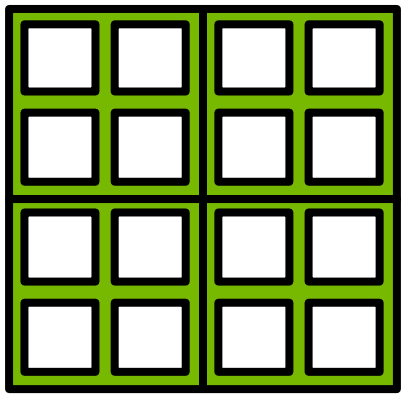


Output

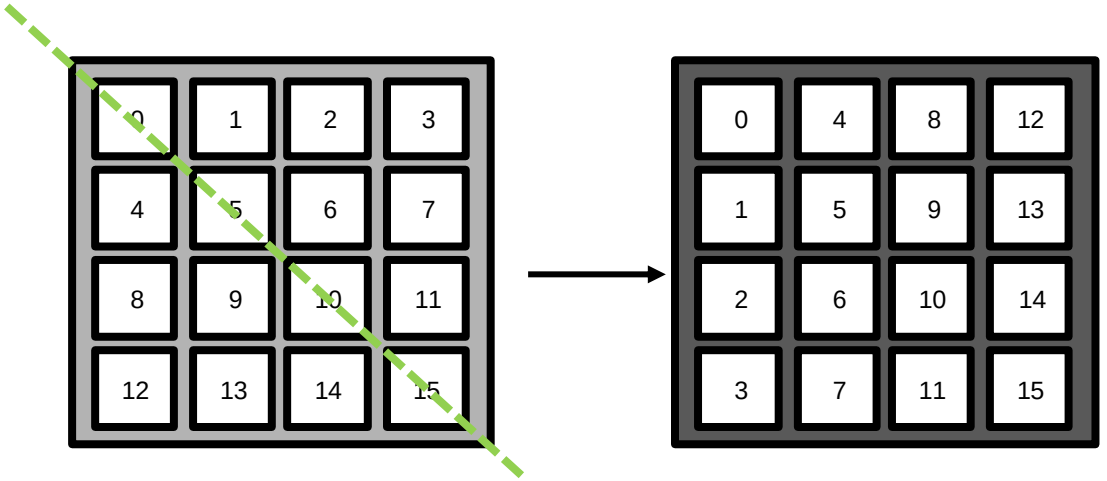
NVIDIA DEEP LEARNING INSTITUTE



Grid



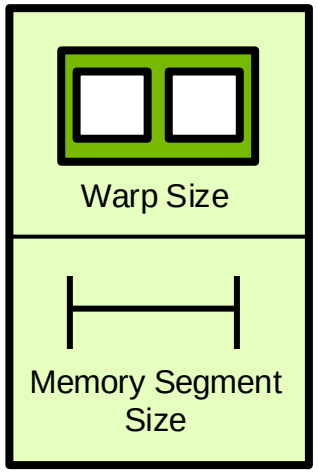
Our goal is to transpose the input by rotating all elements around the diagonal, writing the transposed elements to output



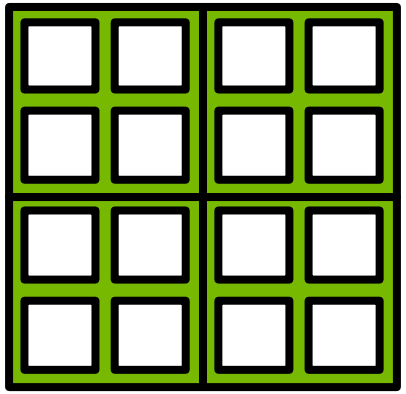
Input

Output

NVIDIA DEEP LEARNING INSTITUTE

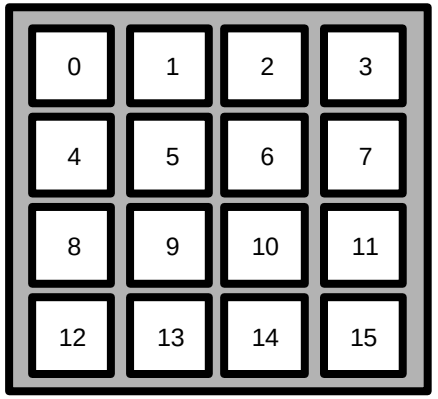


Grid

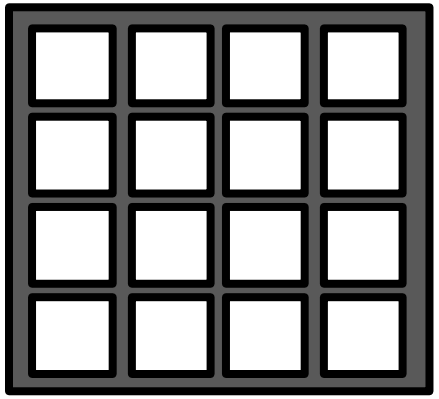


A naïve approach is to launch a grid with threads equal to input elements, and to have each thread read 1 element, then write it to output in the transposed location

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

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

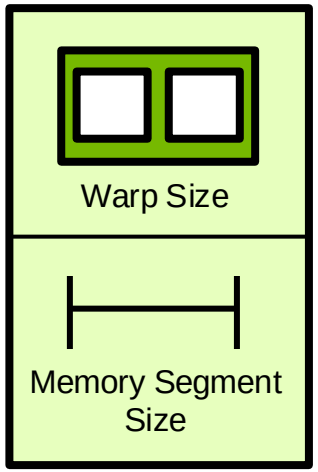


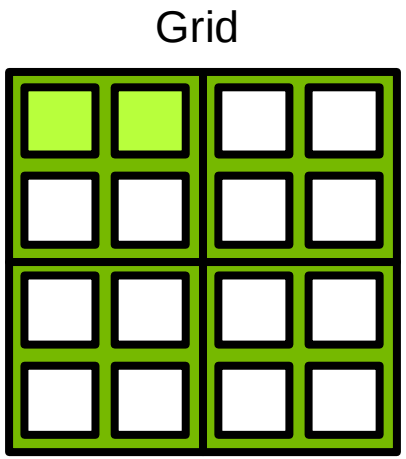
Input



Output

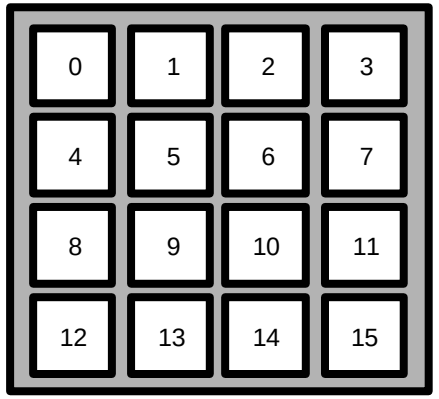
NVIDIA DEEP LEARNING INSTITUTE



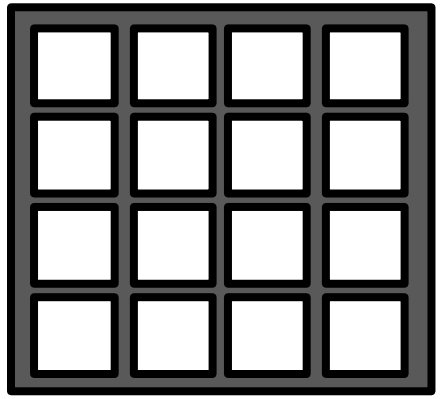


Observing the behavior of a single warp, is it the case that memory reads are coalesced? Let's dig into answering that question

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

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

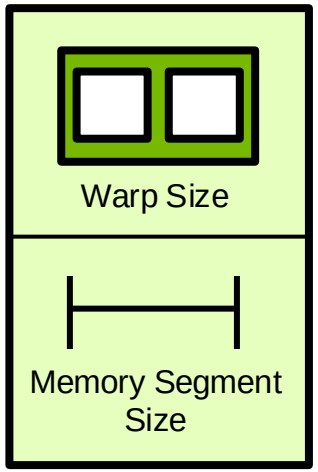


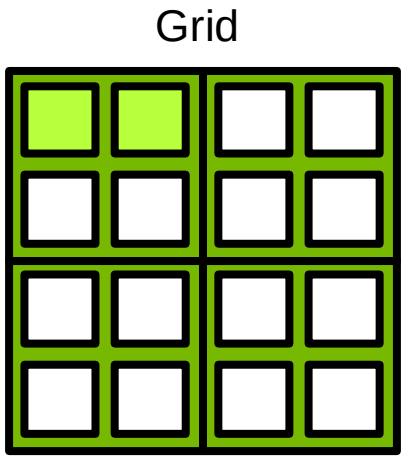
Input



Output

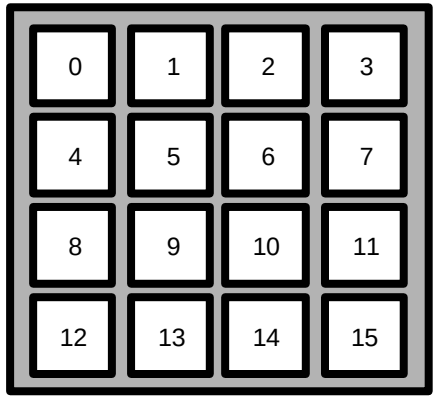
NVIDIA DEEP LEARNING INSTITUTE



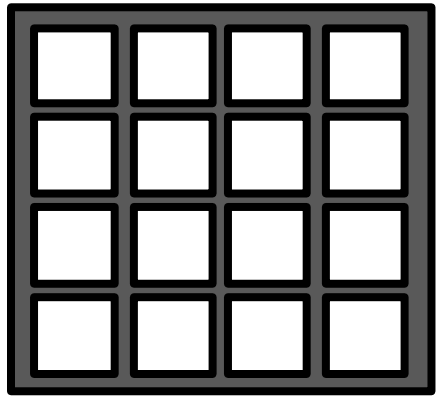


Rewriting the creation of the indexing variables, it is clearer that contiguous threads in the same warp are adjacent along the x axis

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

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

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

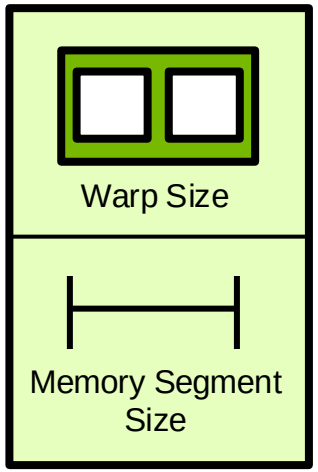


Input

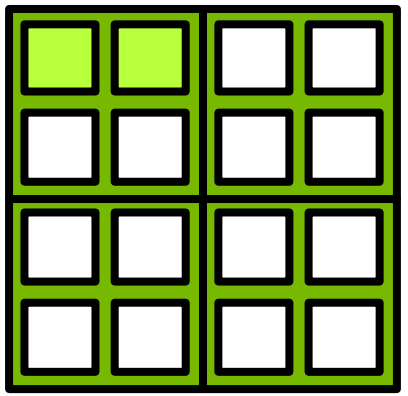


Output

NVIDIA DEEP LEARNING INSTITUTE

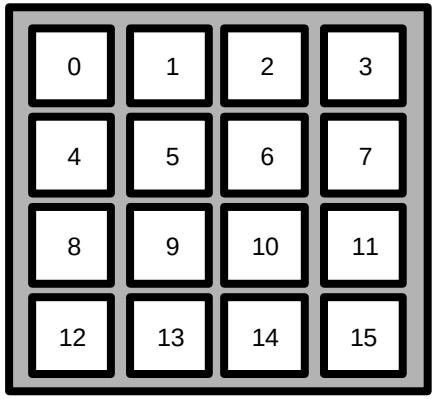


Grid

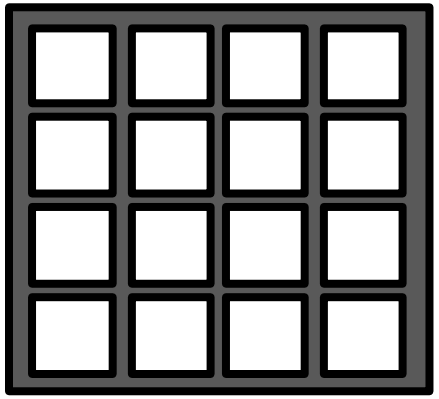


Furthermore, these contiguous threads will read elements from the rows of input where data elements are contiguous

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

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

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

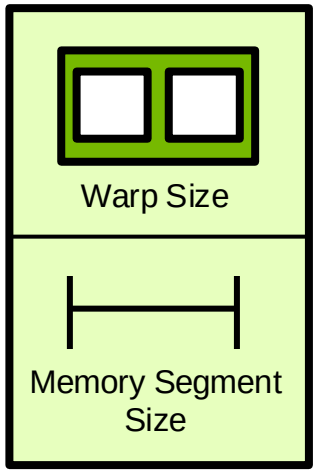


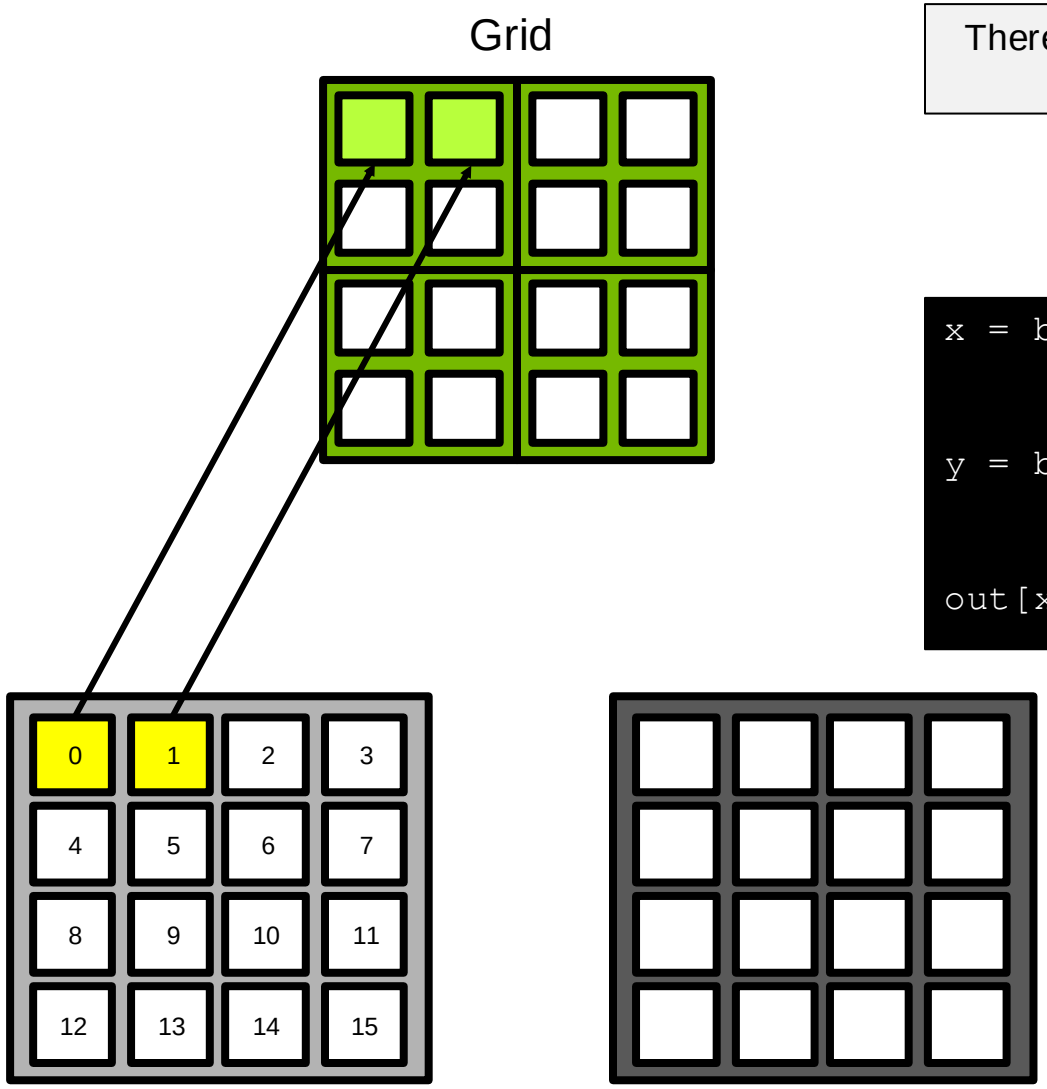
Input



Output

NVIDIA DEEP LEARNING INSTITUTE

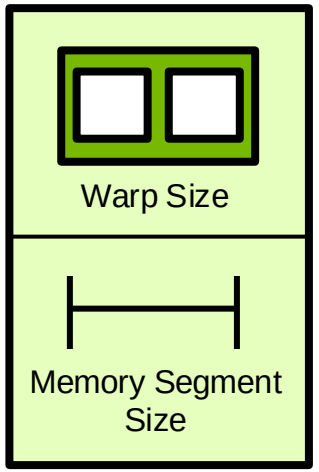




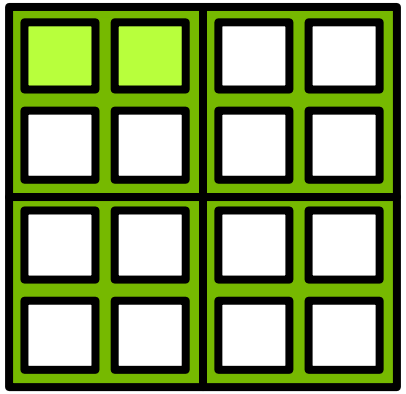
Therefore, it makes sense that reads from input are coalesced

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

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

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

NVIDIA DEEP LEARNING INSTITUTE

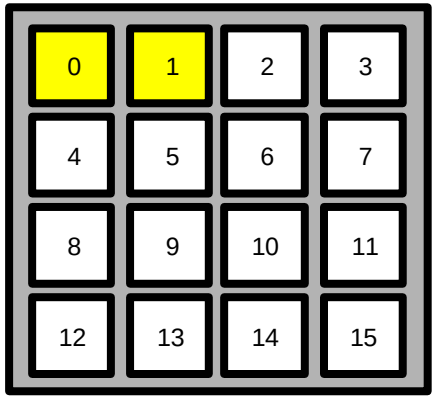


Grid

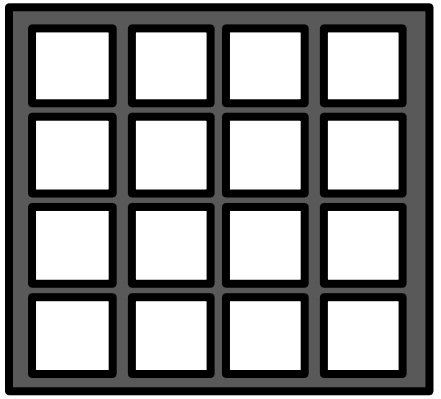


What about this warp's writes to output, will they be coalesced?

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

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

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

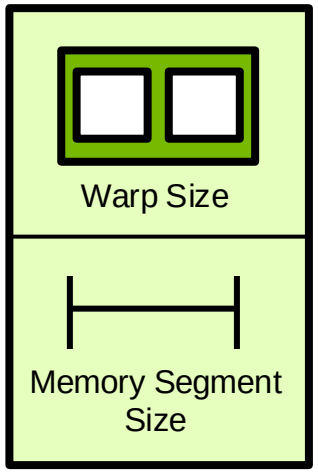


Input

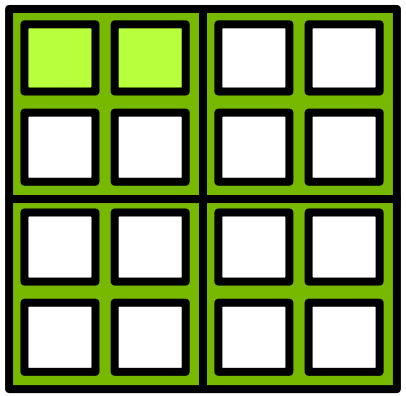


Output

NVIDIA DEEP LEARNING INSTITUTE

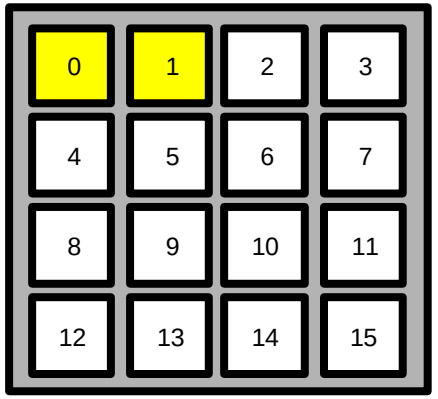


Grid

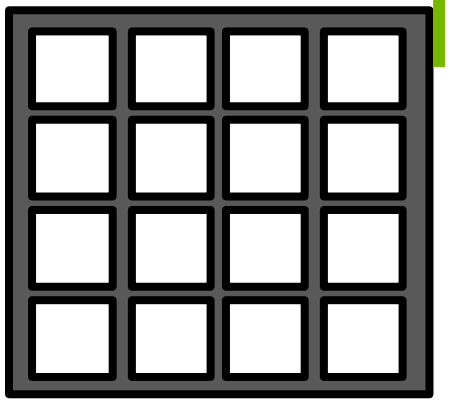


Here we see that contiguous threads in the same warp will be writing along a column in output

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

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

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

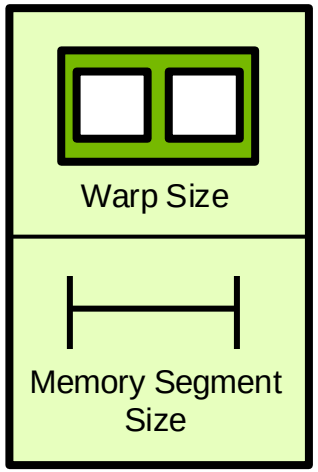


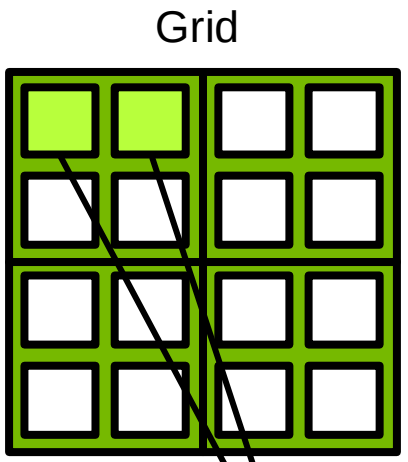
Input



Output

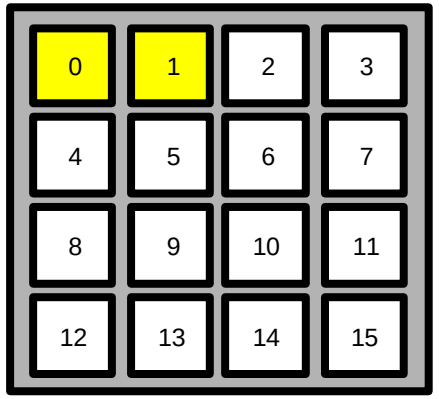
NVIDIA DEEP LEARNING INSTITUTE

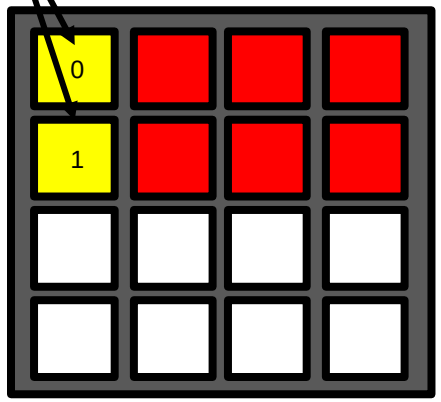




Therefore, the writes will not be coalesced

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

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

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

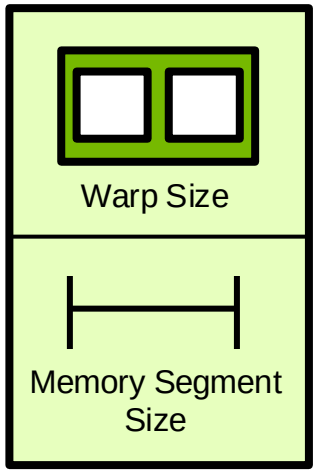


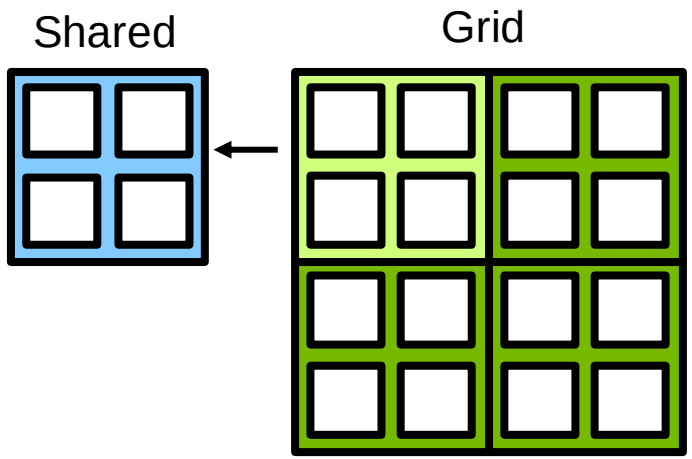


Input

Output

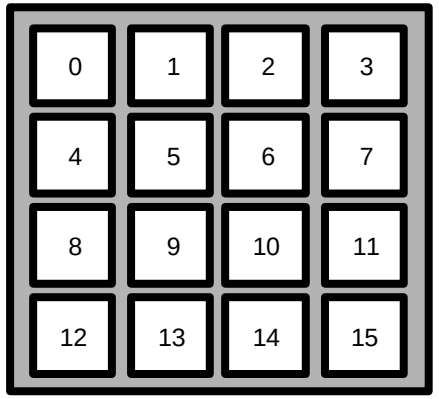
NVIDIA DEEP LEARNING INSTITUTE



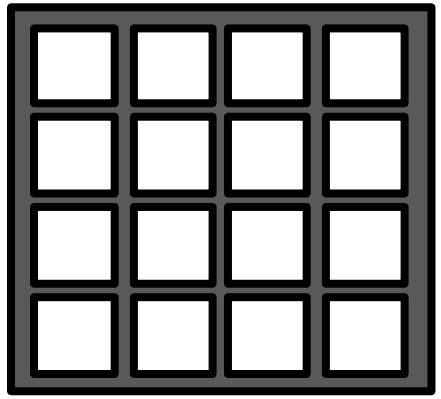


We can use shared memory to make coalesced reads and writes. Here, each block will allocate a (2,2) shared memory tile

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

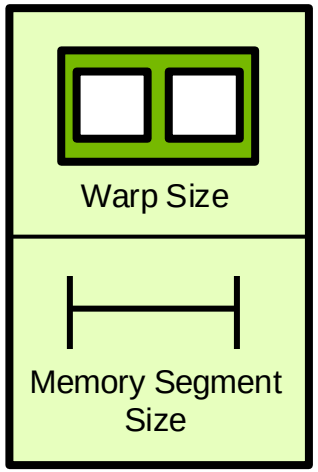


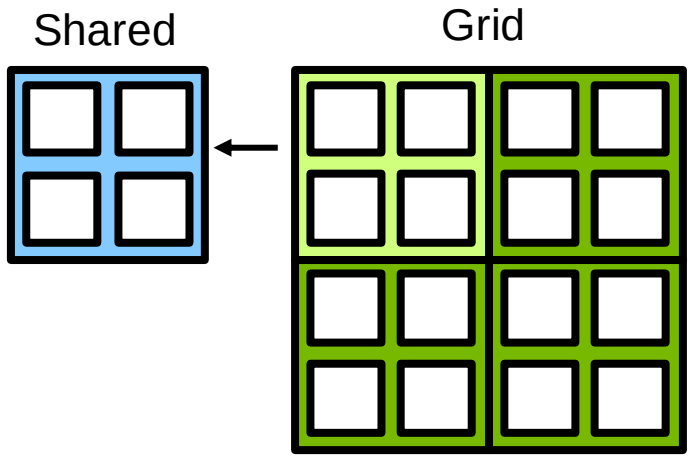
Input



Output

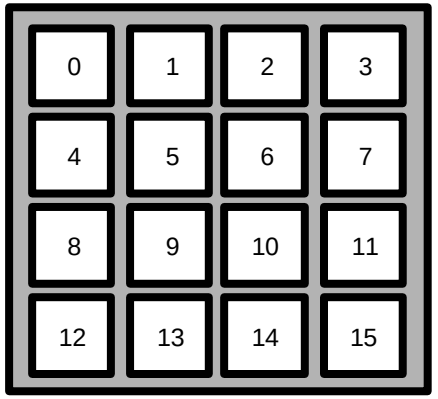
NVIDIA DEEP LEARNING INSTITUTE



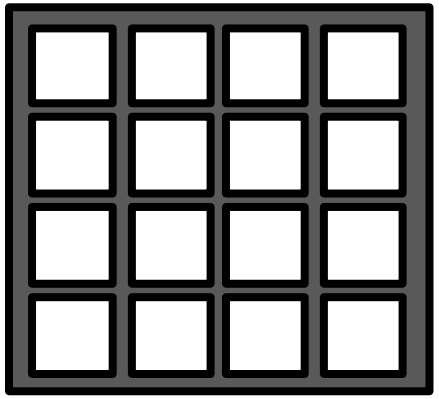


(It is worth reminding that in our slides, to preserve space, 2 threads is a warp length. A real warp is 32 threads)

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

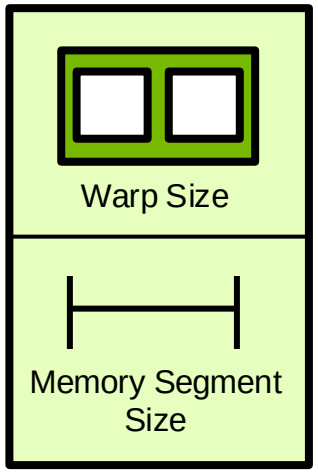


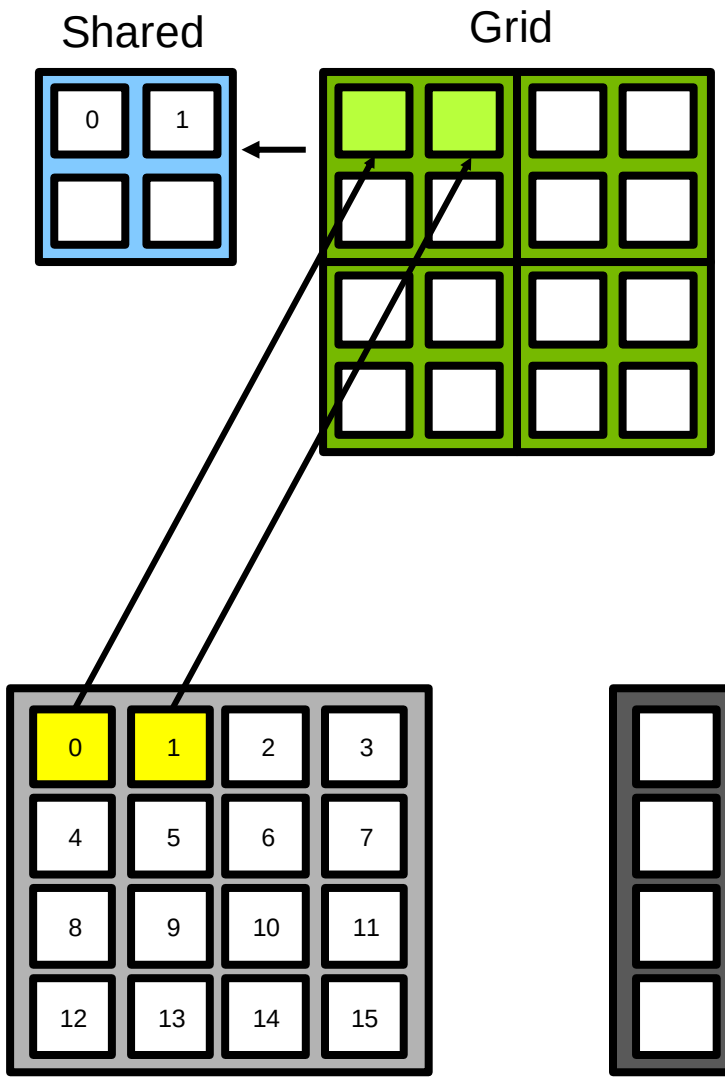
Input



Output

NVIDIA DEEP LEARNING INSTITUTE

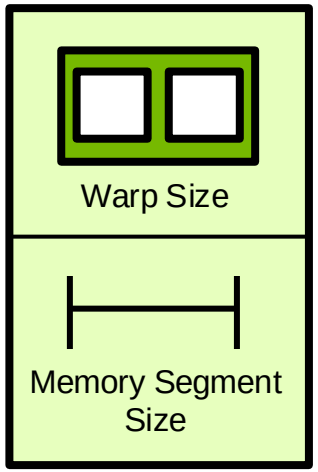


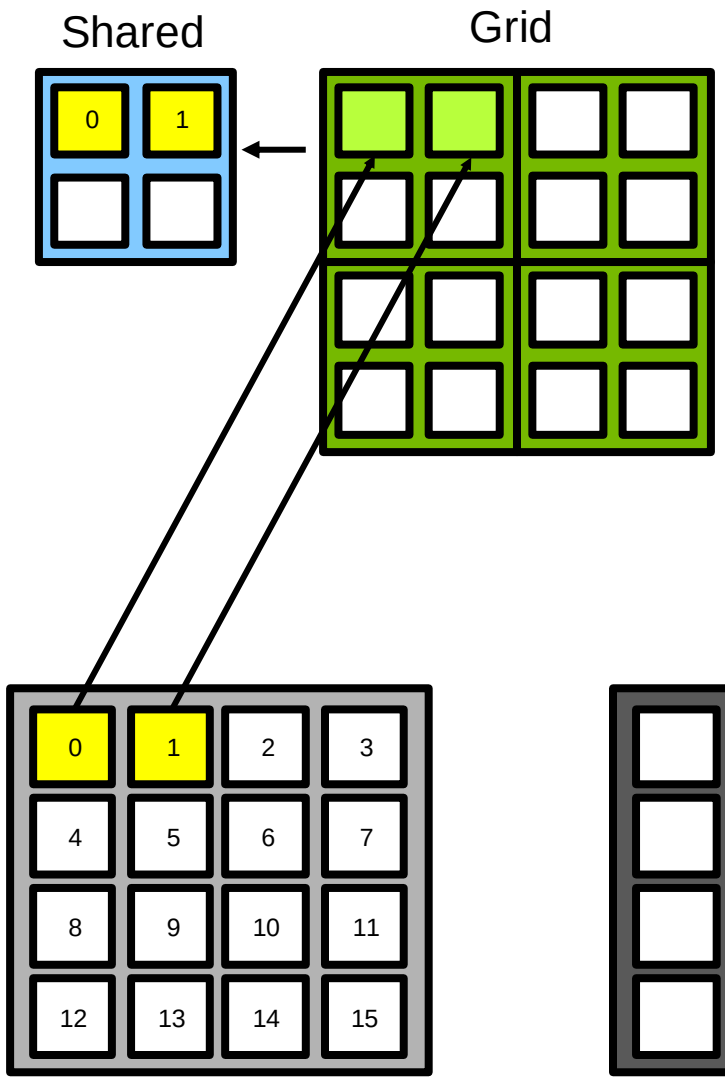


Now we can make coalesced reads from input, and write the values to the block's shared memory tile

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

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

NVIDIA DEEP LEARNING INSTITUTE

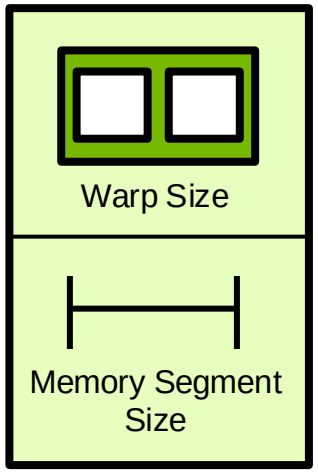


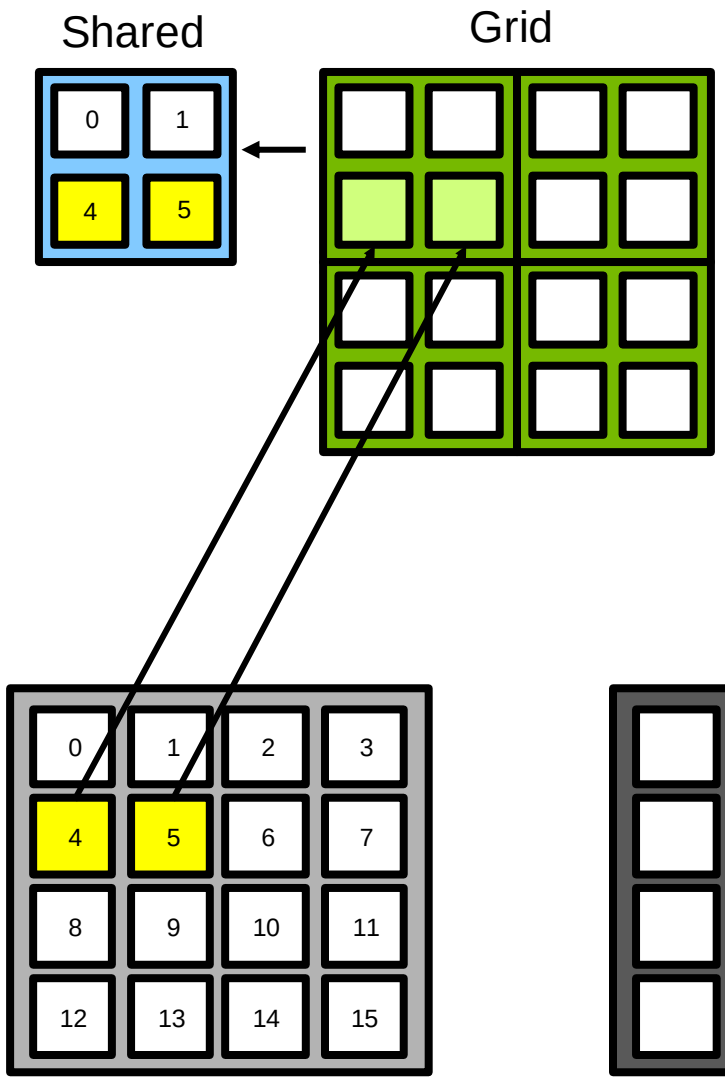


Because each shared memory tile is local to the block (not the grid) we index into it using thread indices, not grid indices

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

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

NVIDIA DEEP LEARNING INSTITUTE

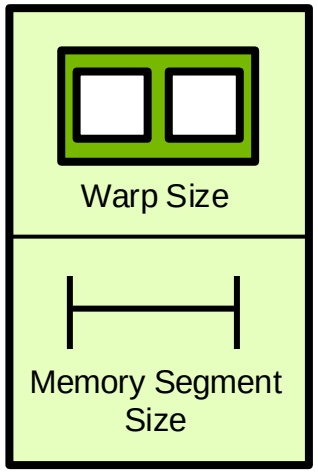


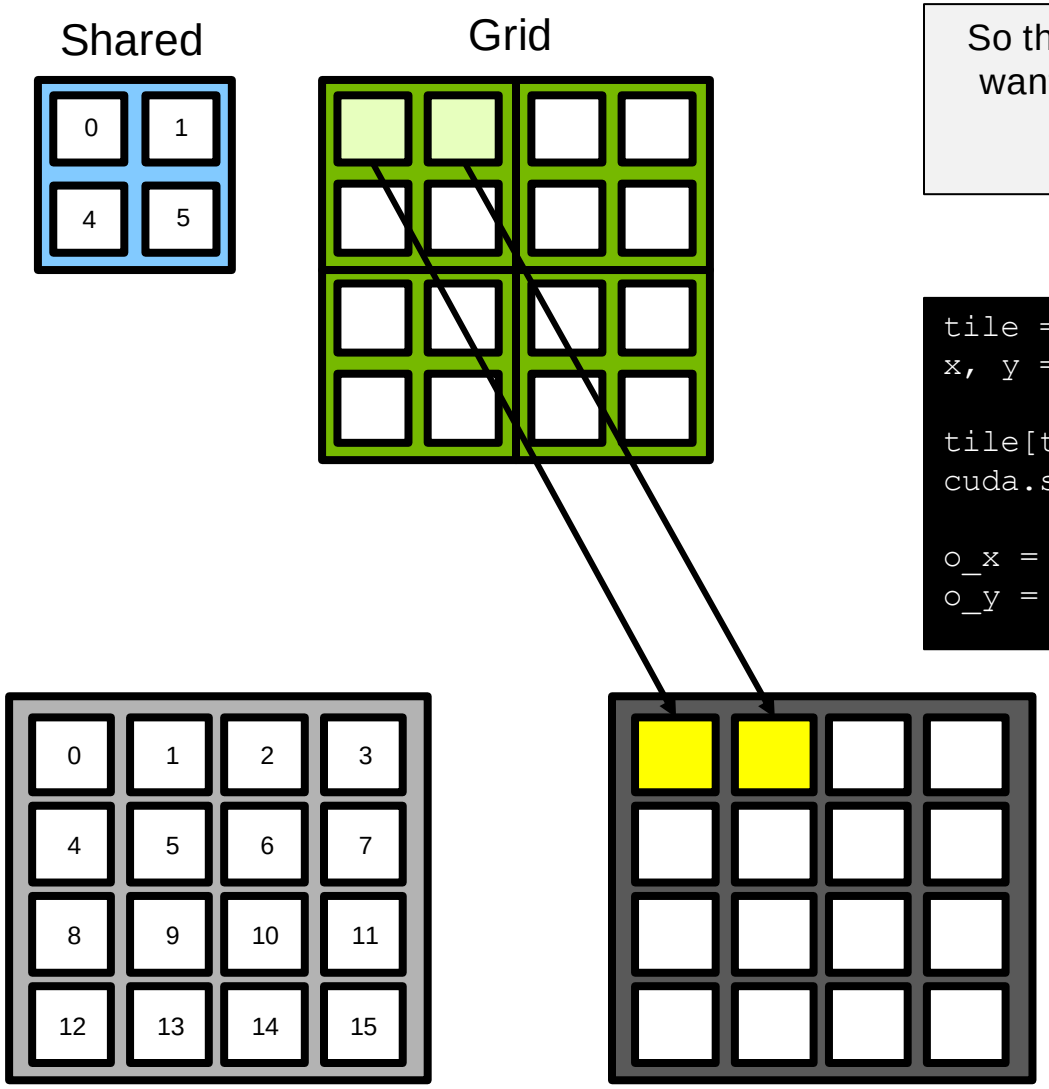


After synchronizing on all threads in the block, the tile will contain all the data this block needs to begin the writes

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

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

NVIDIA DEEP LEARNING INSTITUTE

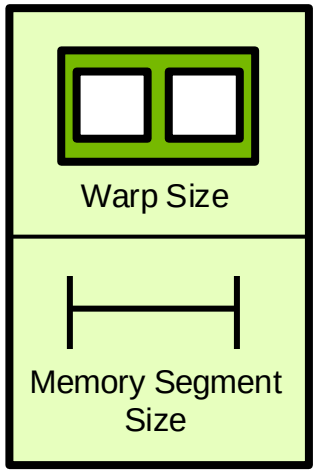


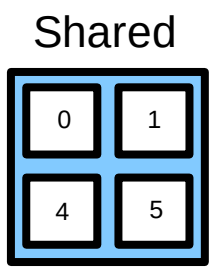


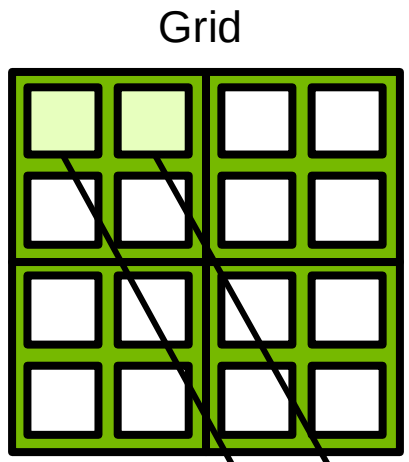
So that the writes are coalesced, we want each warp to write to a row in output

```
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
```

NVIDIA DEEP LEARNING INSTITUTE

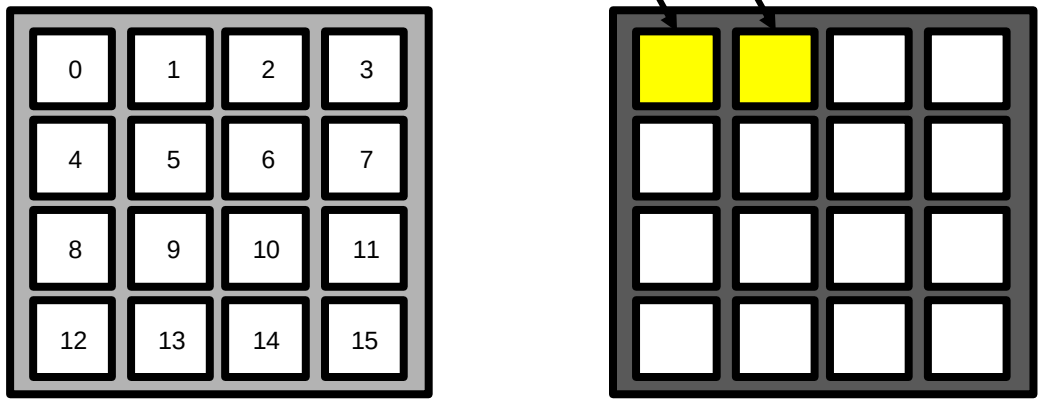




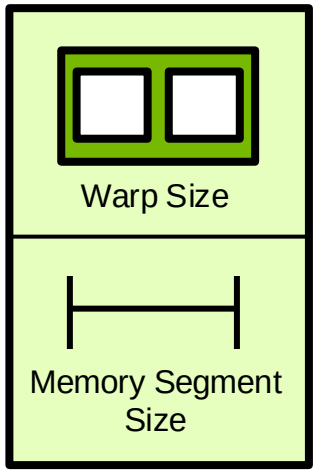


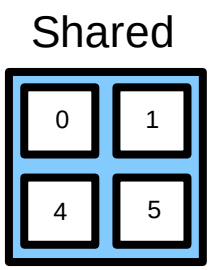
Notice that to write to output at the transposed locations we use blockIdx.y and blockDim.y to calculate the x axis index in output…

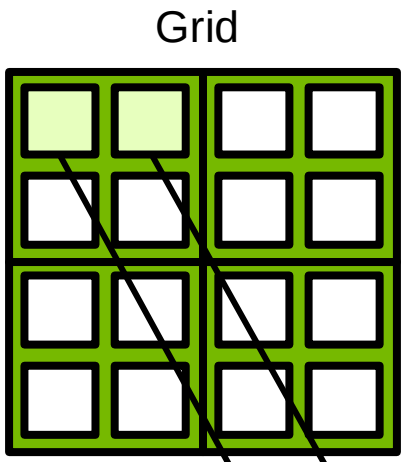
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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
```



NVIDIA DEEP LEARNING INSTITUTE

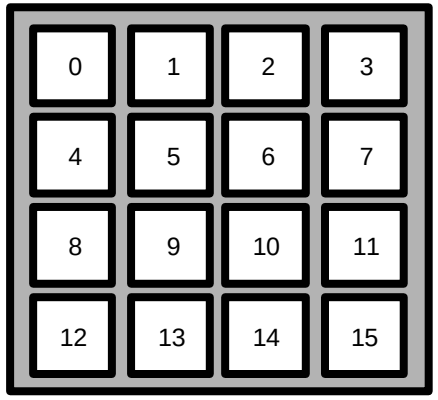


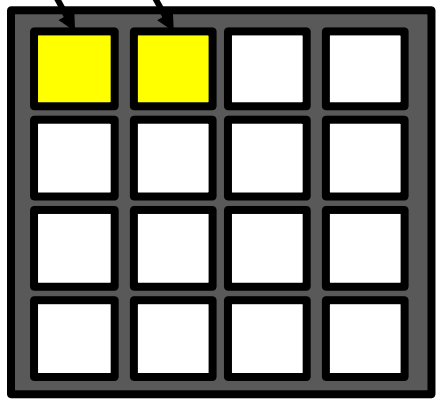




…but to accomplish coalesced writes, we still map increments to threadIdx.x to be along the x output axis

```
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
```

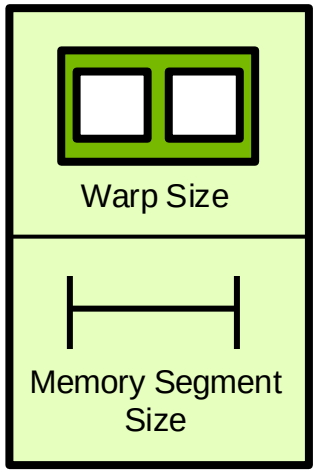


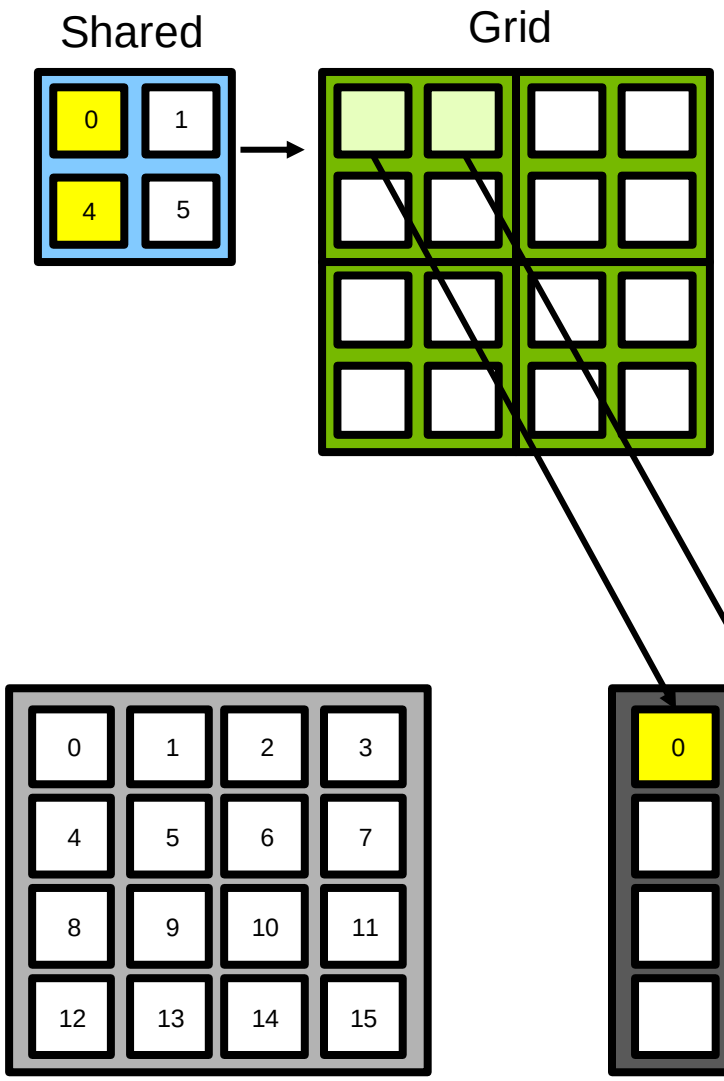


Input

Output

NVIDIA DEEP LEARNING INSTITUTE

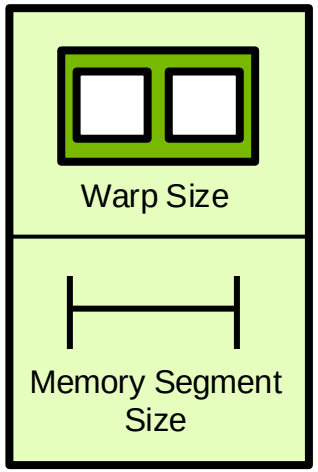


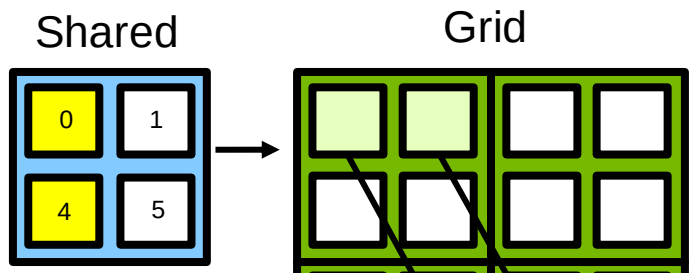


Because of this last detail, each warp will need to read from a column of the shared memory tile in order to perform the transpose

```
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]
```

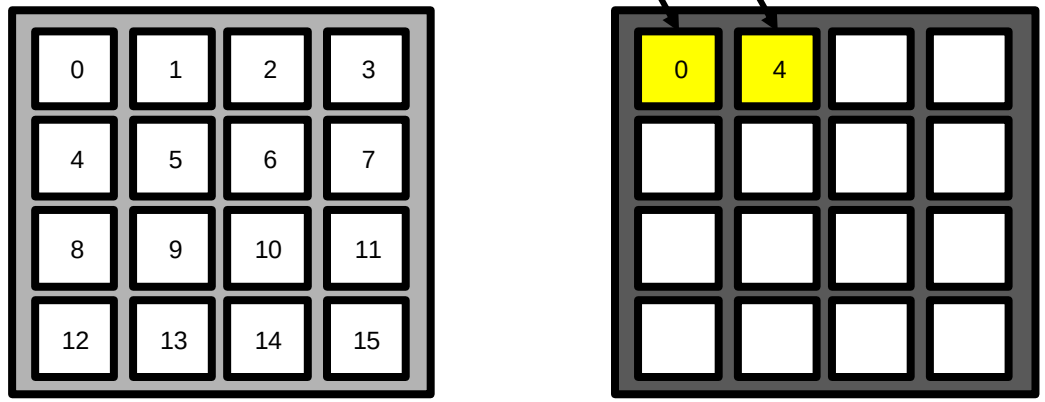
NVIDIA DEEP LEARNING INSTITUTE



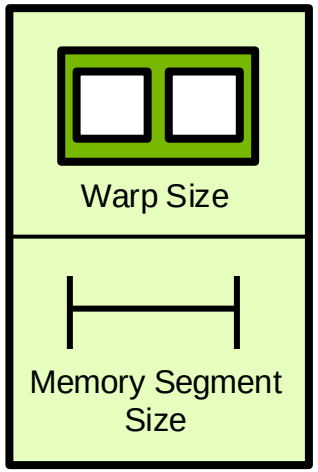


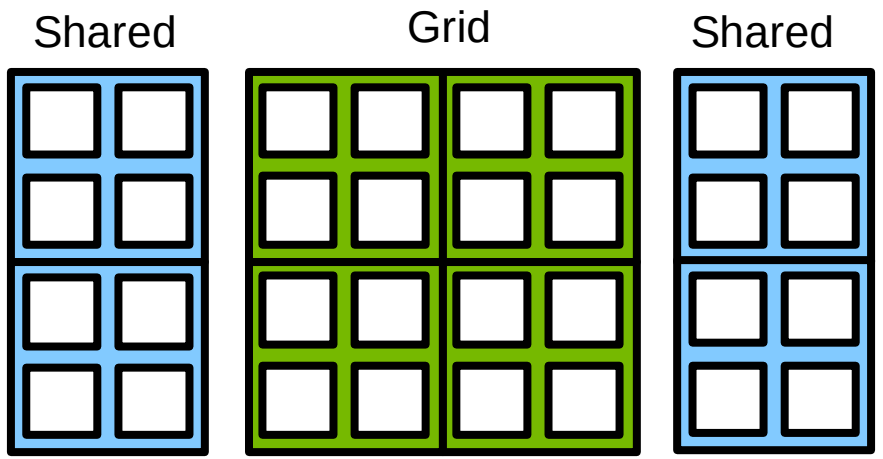
(There's more to come about efficient reads/writes to/from shared memory, but for now know that reading across the column in shared memory has very low impact compared to doing so with global memory)

```
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]
```



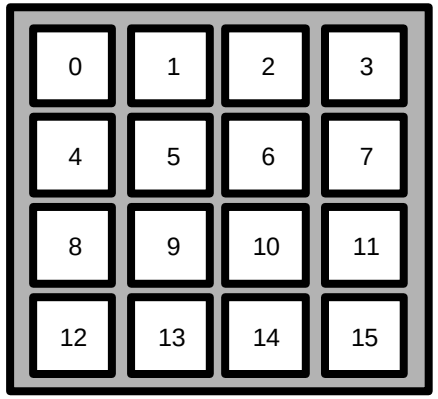
NVIDIA DEEP LEARNING INSTITUTE



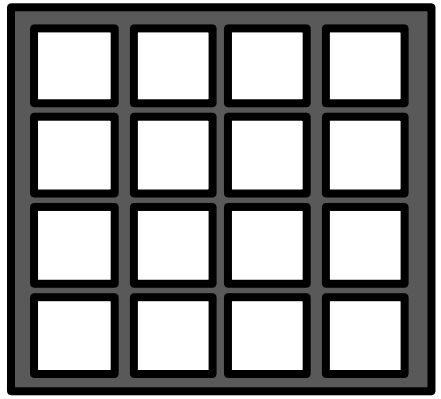


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

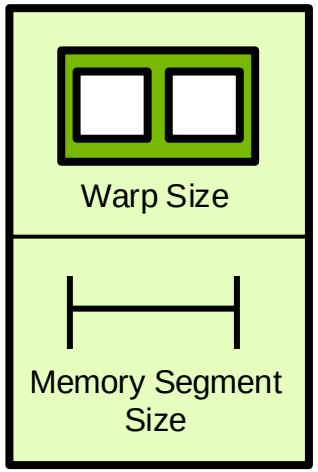


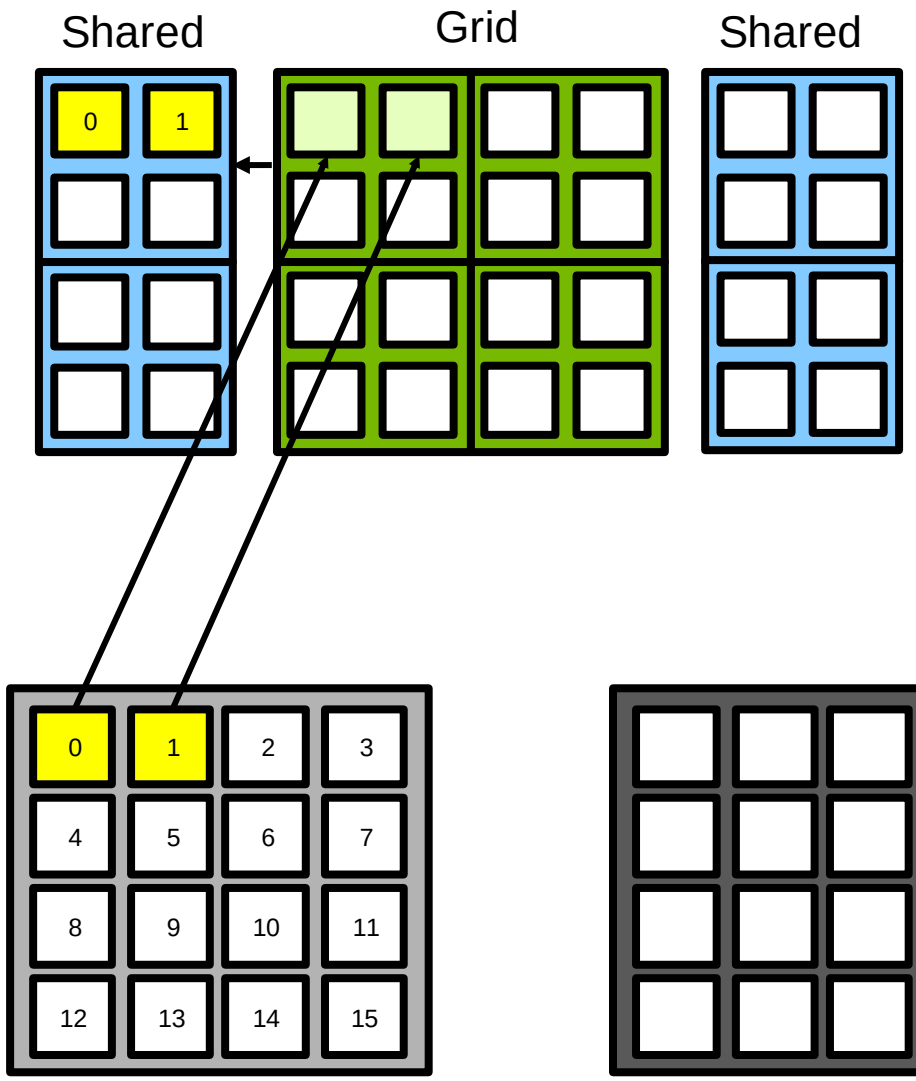
Input



Output

NVIDIA DEEP LEARNING INSTITUTE

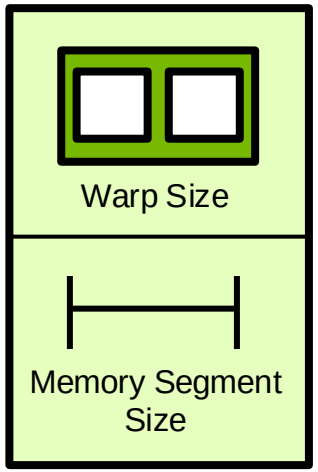


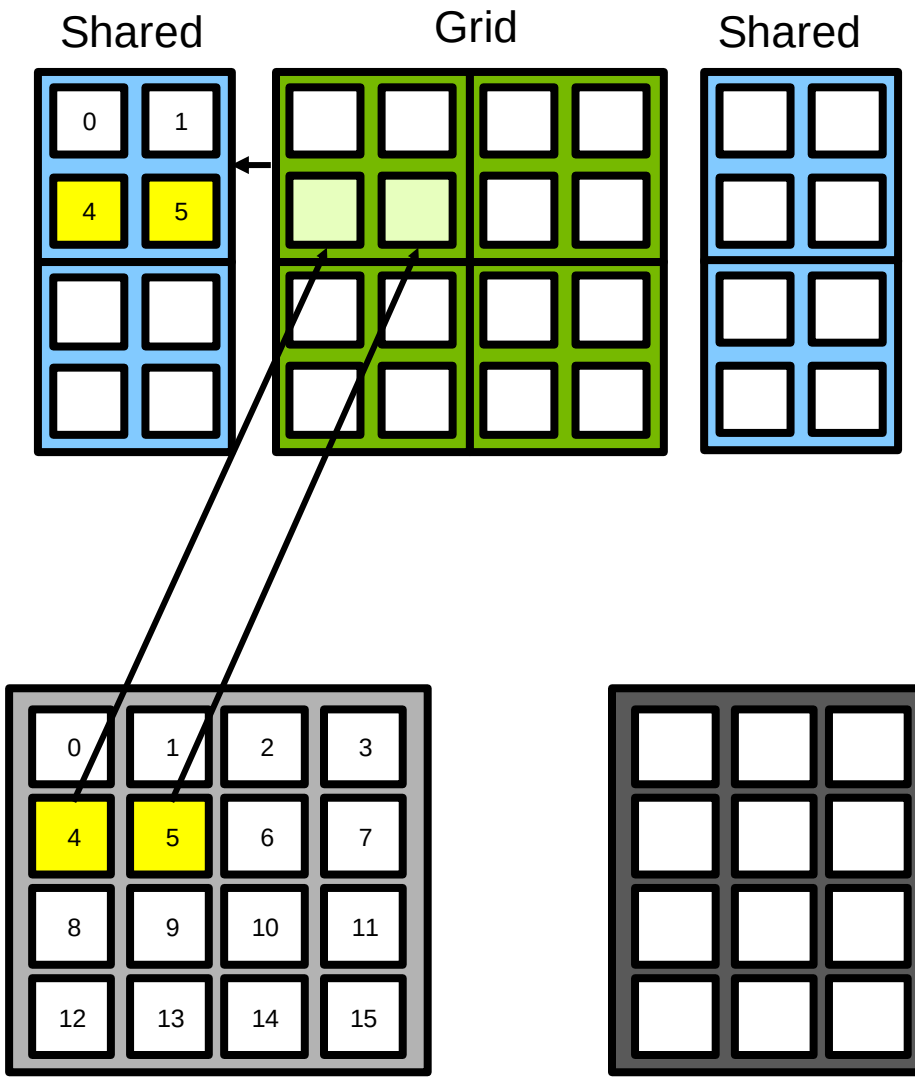


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

NVIDIA DEEP LEARNING INSTITUTE

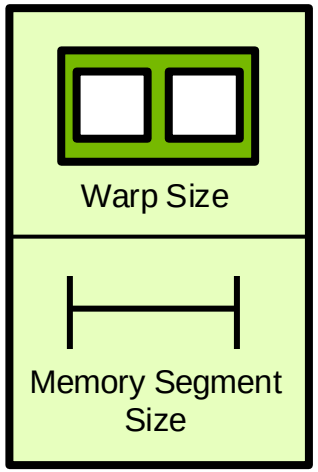


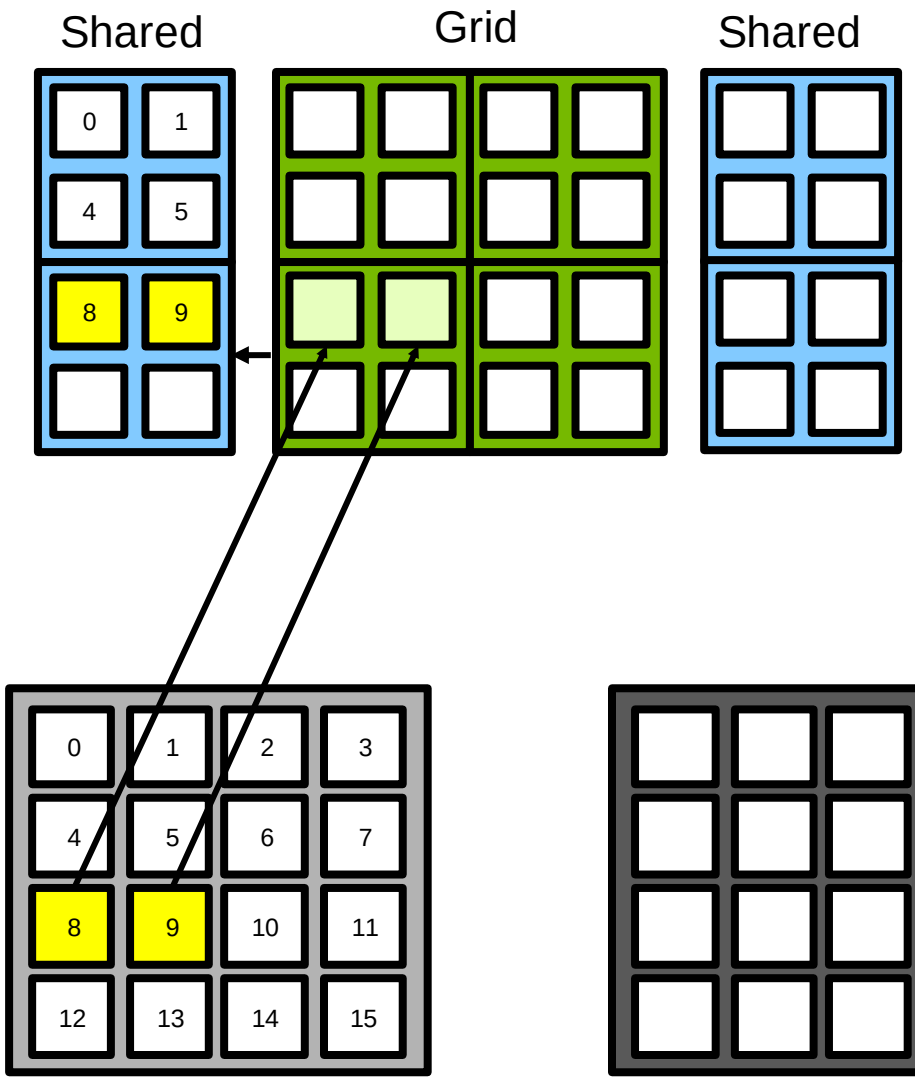


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

NVIDIA DEEP LEARNING INSTITUTE

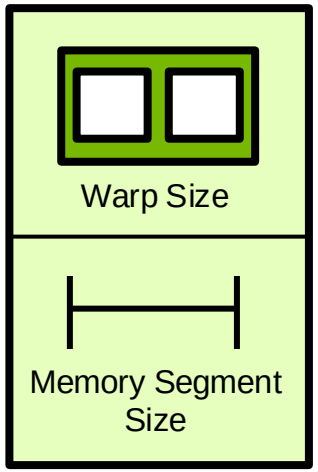


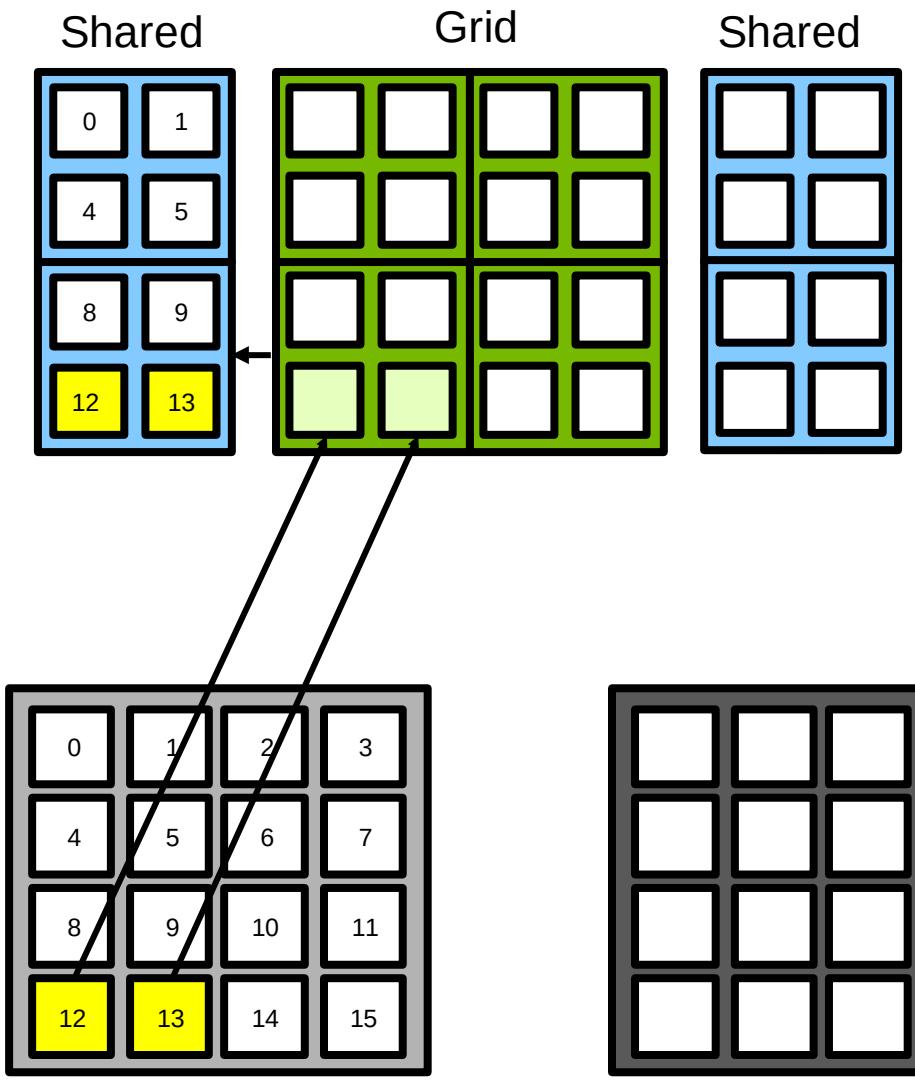


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

NVIDIA DEEP LEARNING INSTITUTE

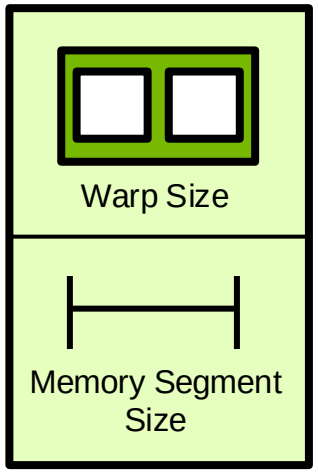


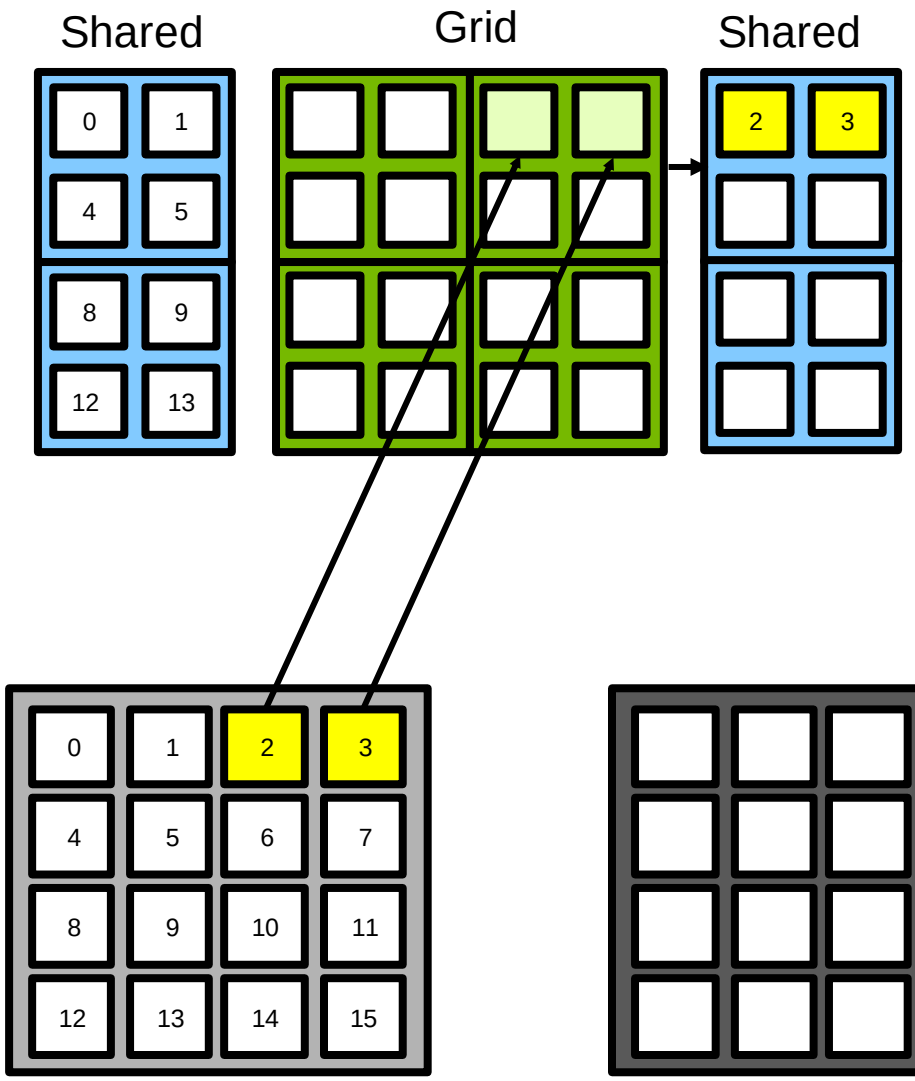


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

NVIDIA DEEP LEARNING INSTITUTE

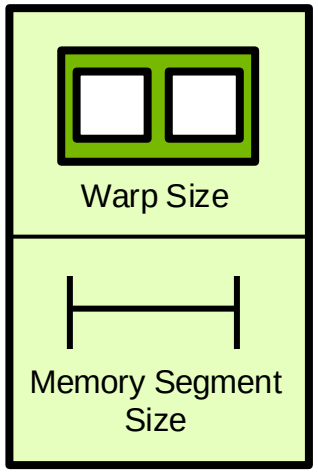


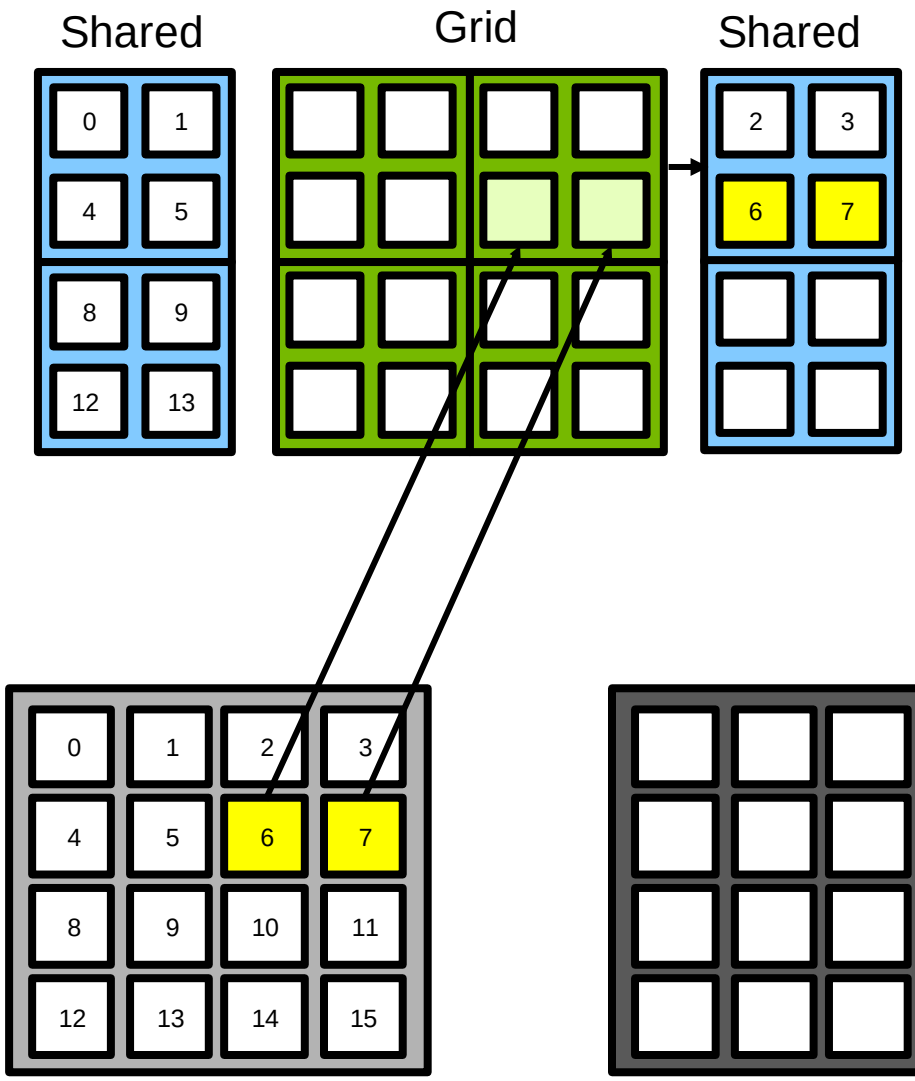


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

NVIDIA DEEP LEARNING INSTITUTE

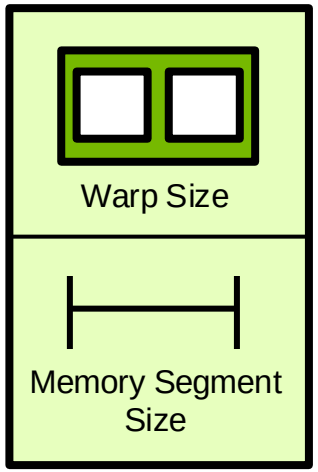


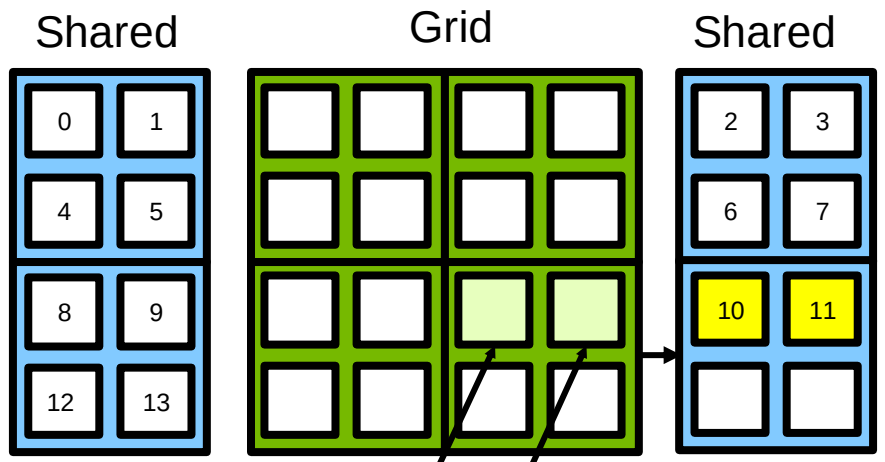


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

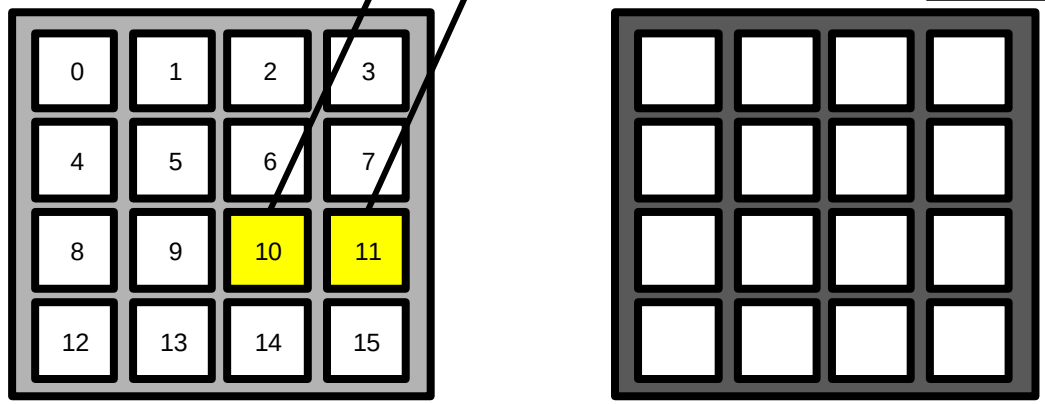
NVIDIA DEEP LEARNING INSTITUTE





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

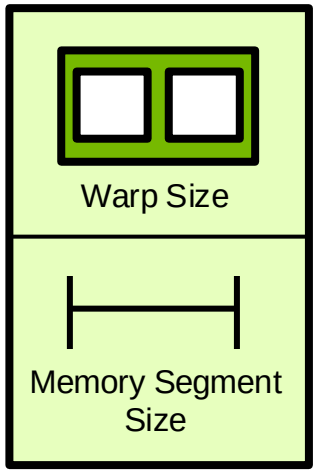
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

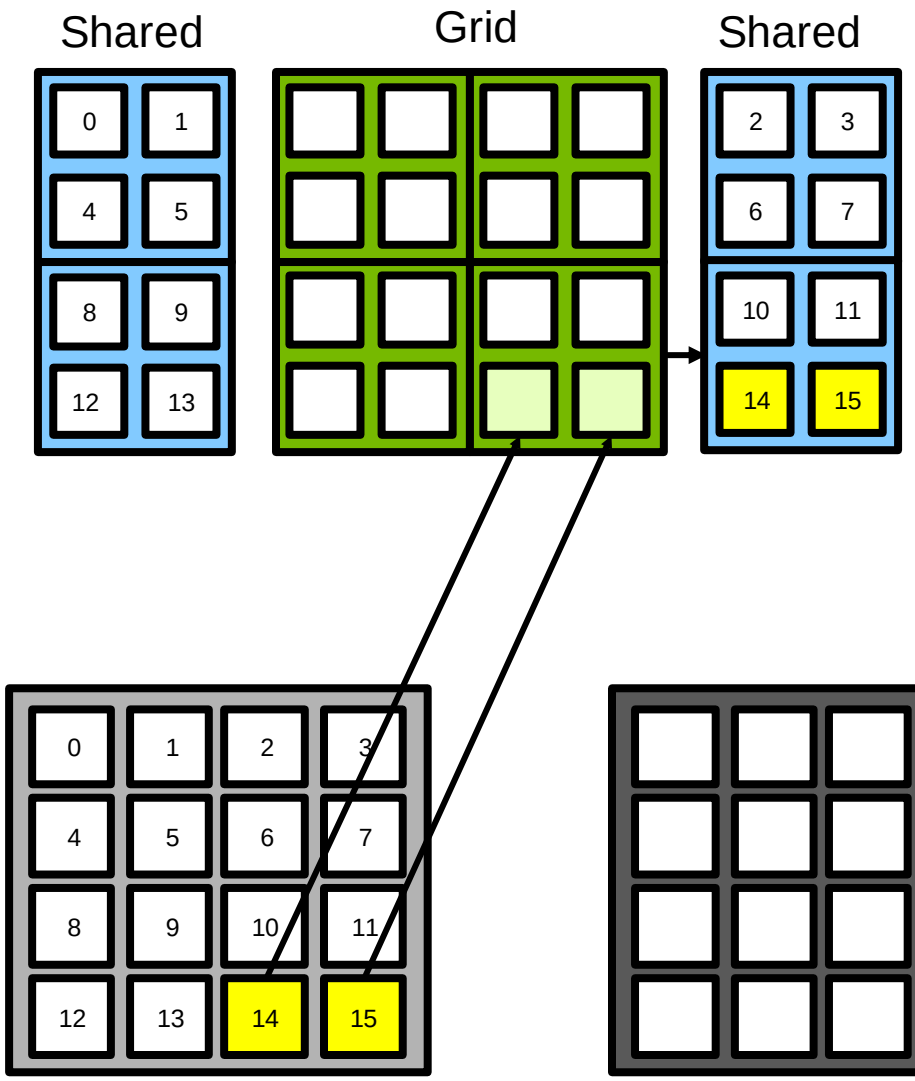


Input

Output

NVIDIA DEEP LEARNING INSTITUTE

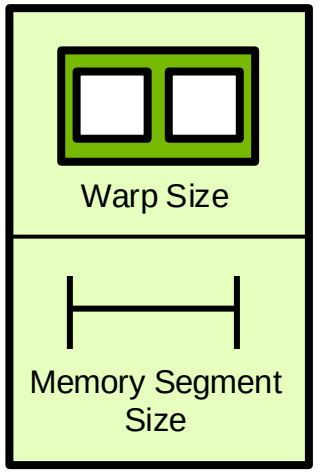


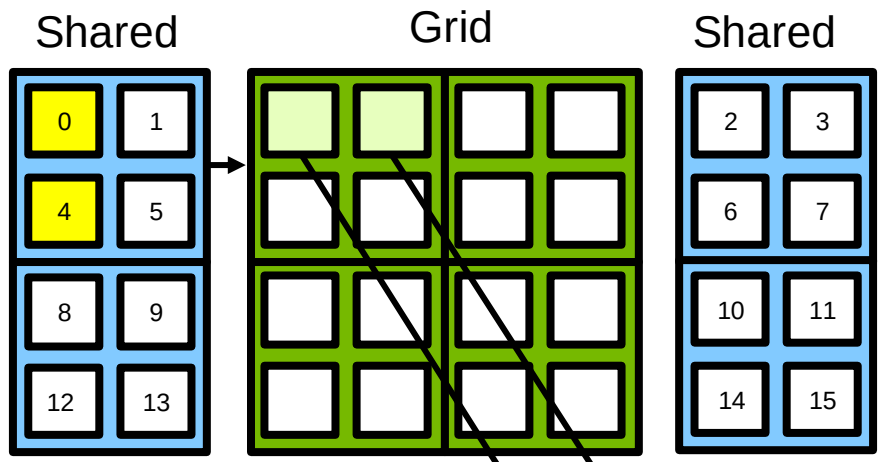


In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```

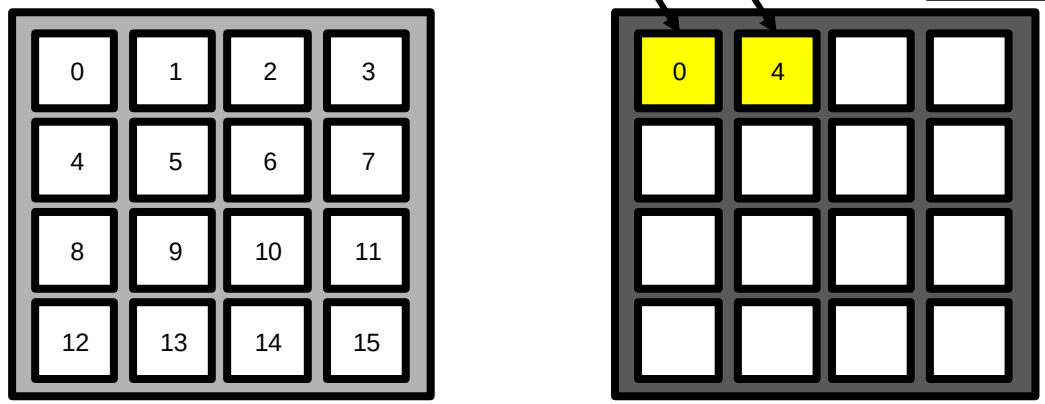
NVIDIA DEEP LEARNING INSTITUTE





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

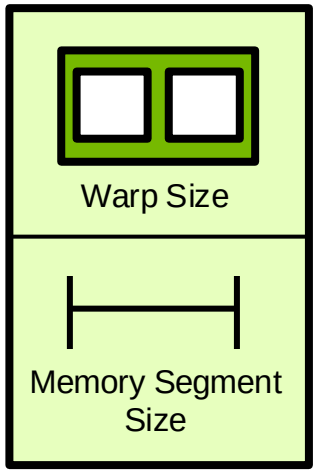
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

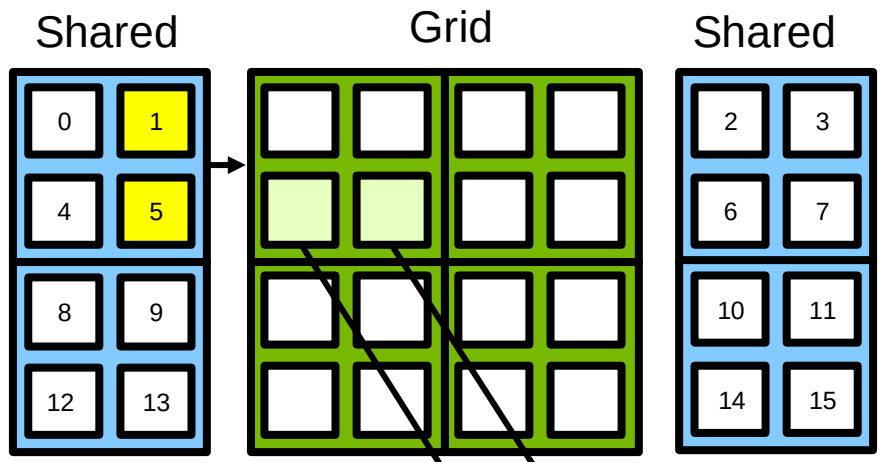


Input

Output

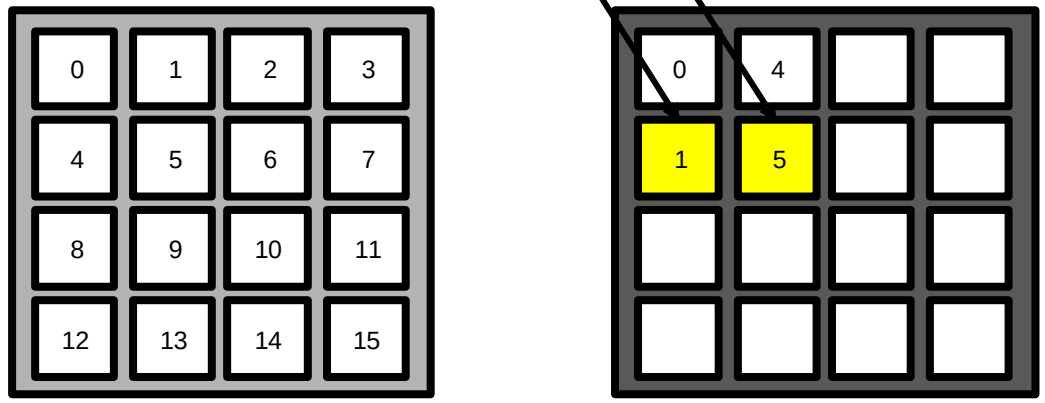
NVIDIA DEEP LEARNING INSTITUTE



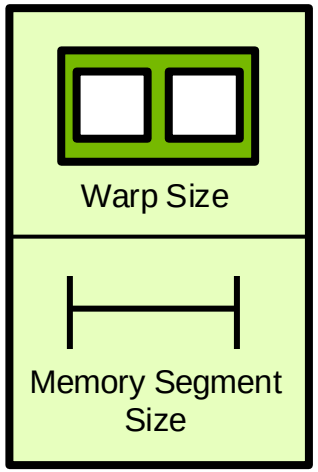


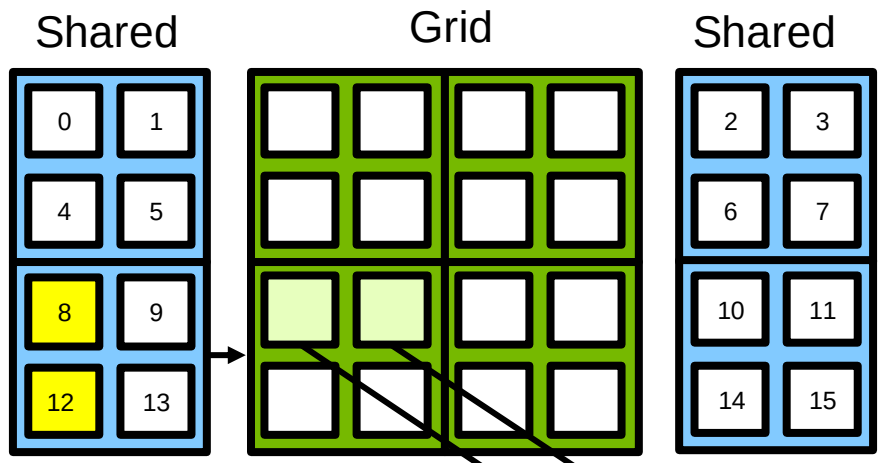
In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory

```
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]
```



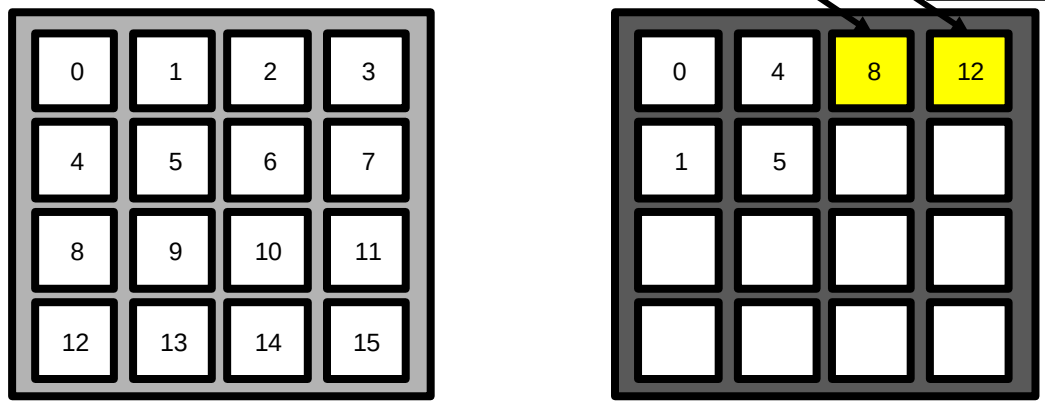
NVIDIA DEEP LEARNING INSTITUTE





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

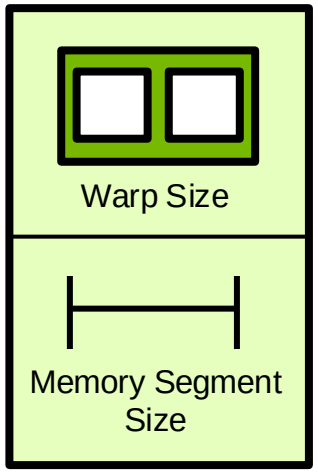
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

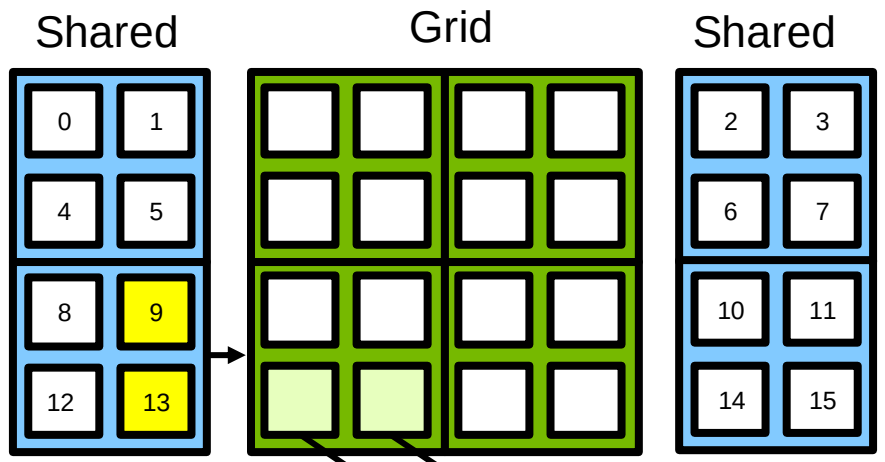


Input

Output

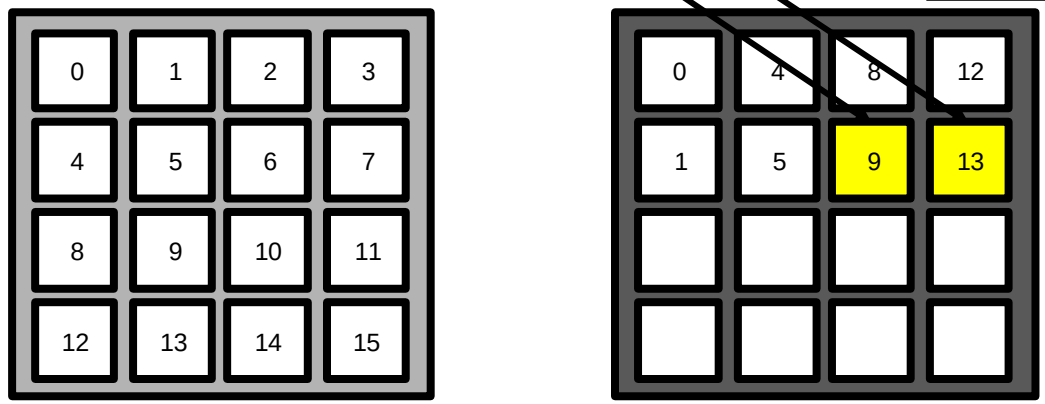
NVIDIA DEEP LEARNING INSTITUTE





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

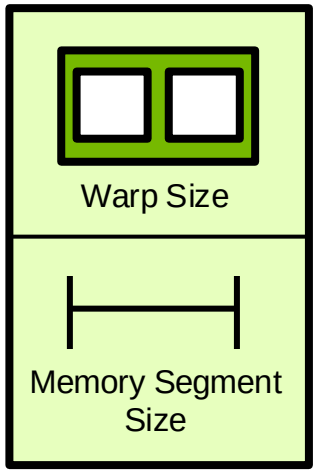
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

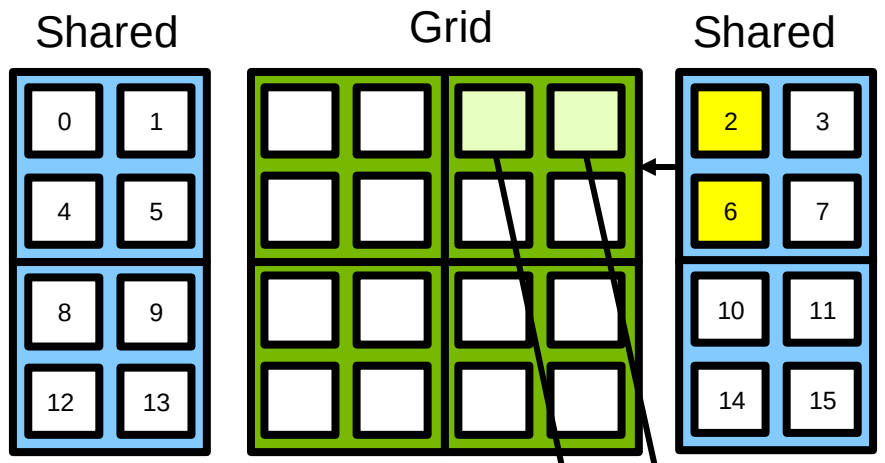


Input

Output

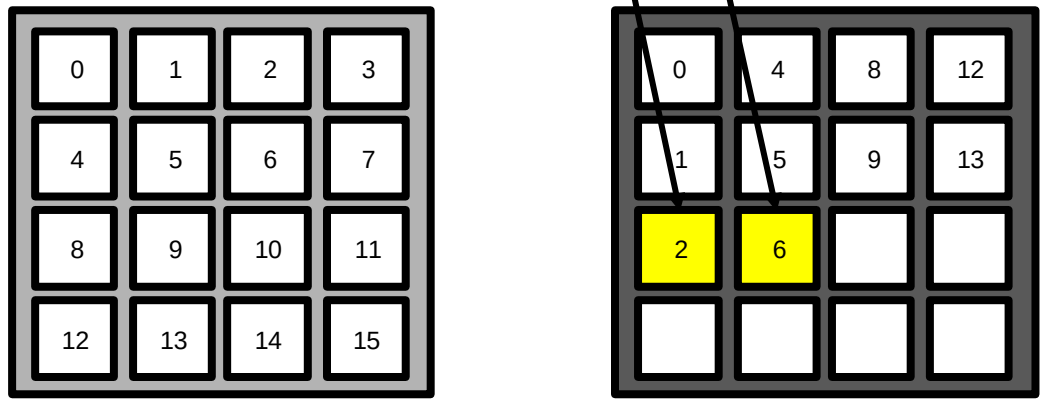
NVIDIA DEEP LEARNING INSTITUTE





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

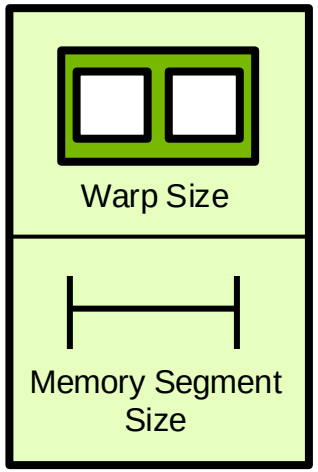
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

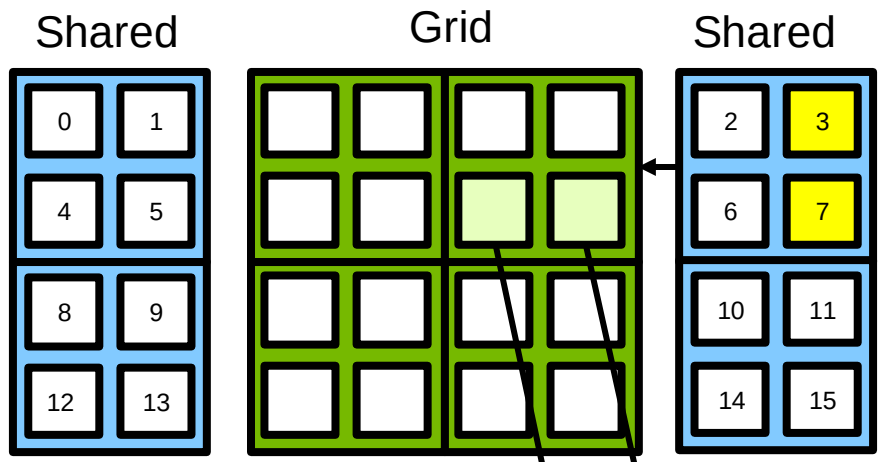


Input

Output

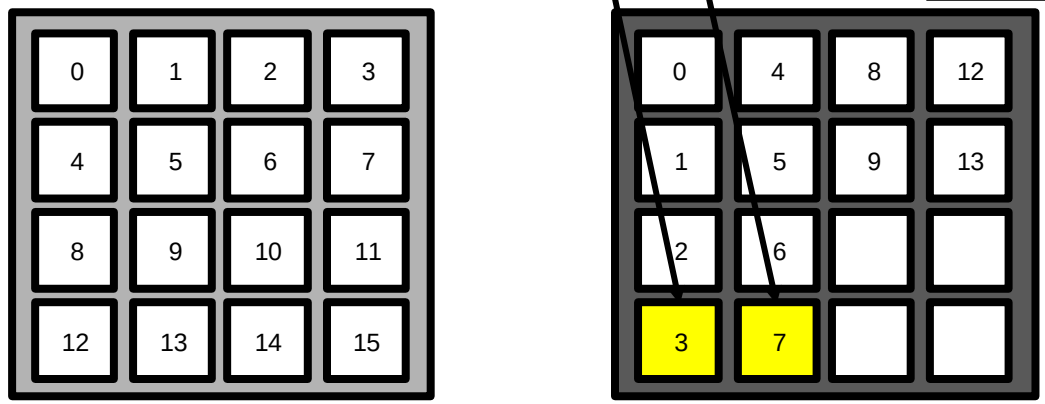
NVIDIA DEEP LEARNING INSTITUTE





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

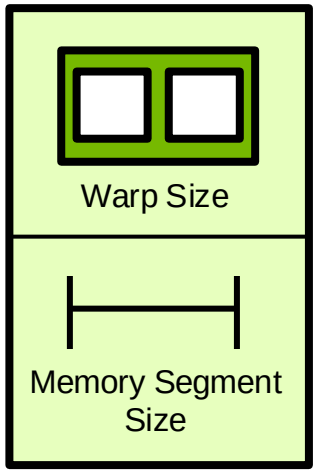
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

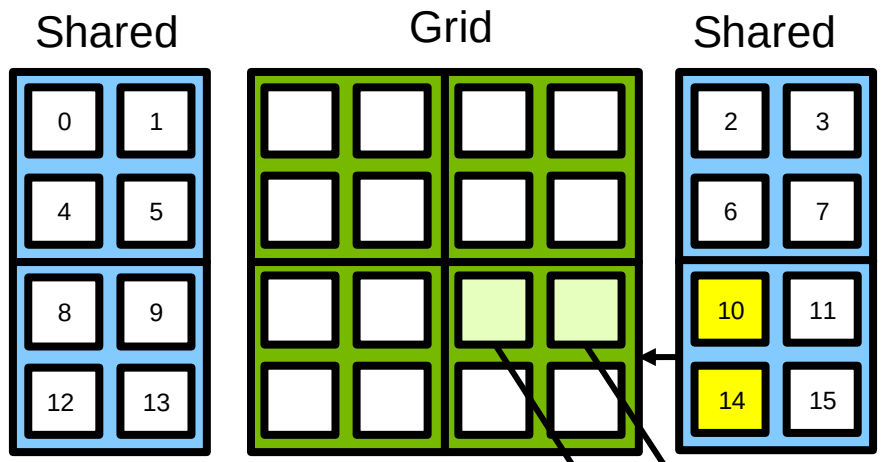


Input

Output

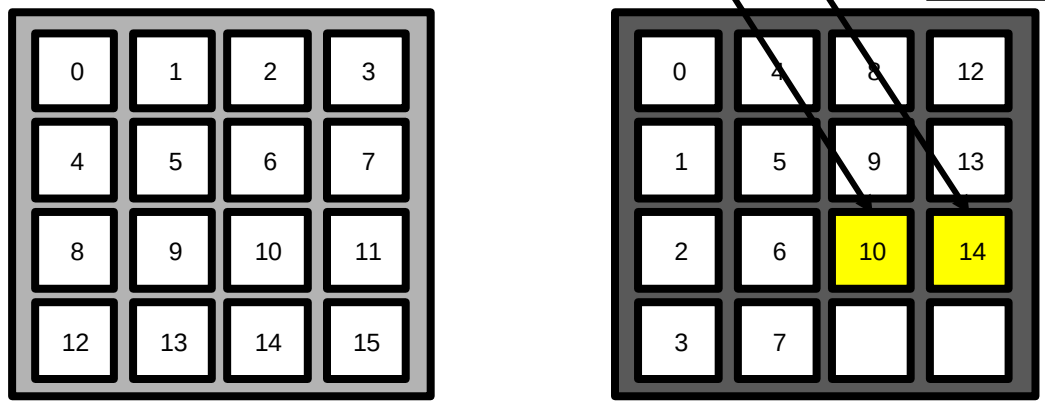
NVIDIA DEEP LEARNING INSTITUTE





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

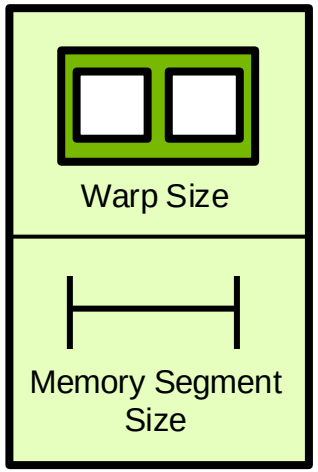
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

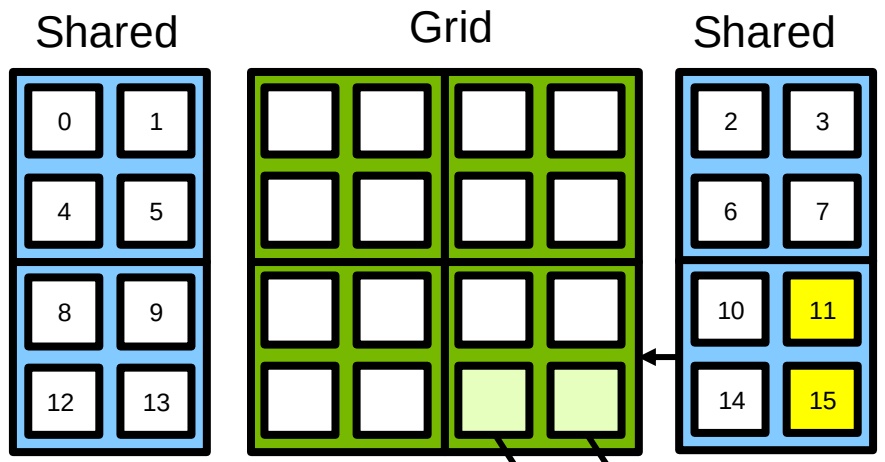


Input

Output

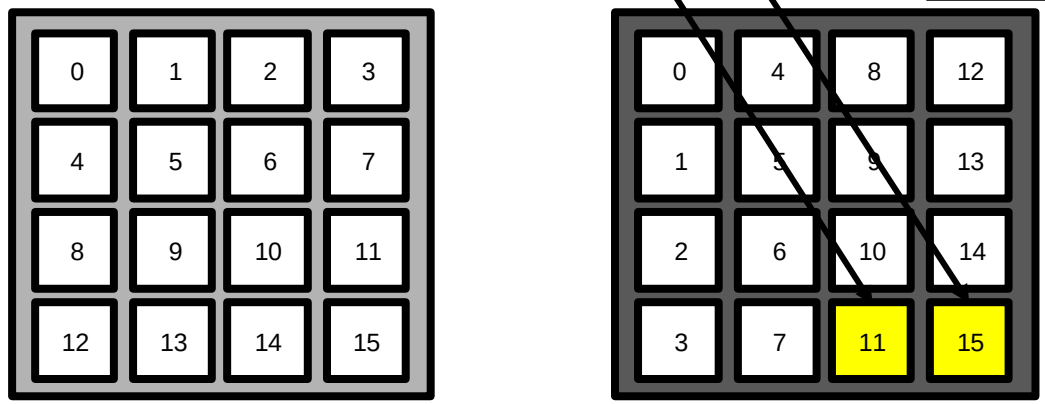
NVIDIA DEEP LEARNING INSTITUTE





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

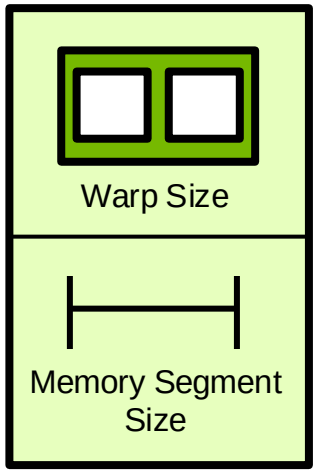
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

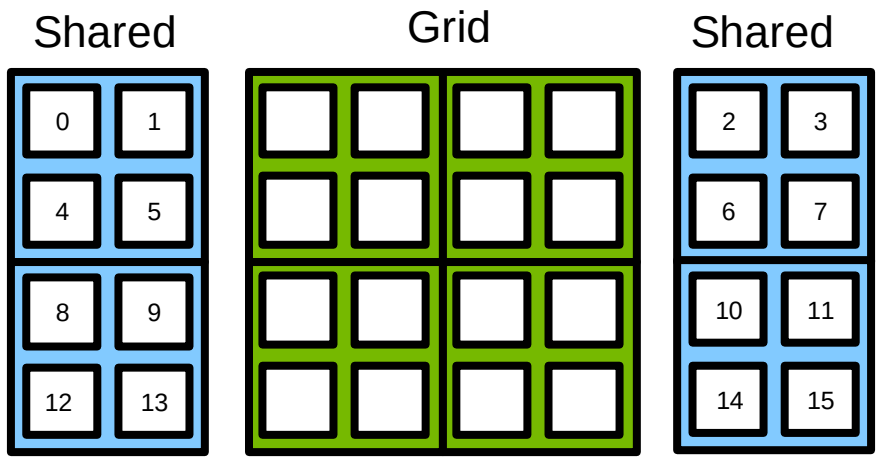


Input

Output

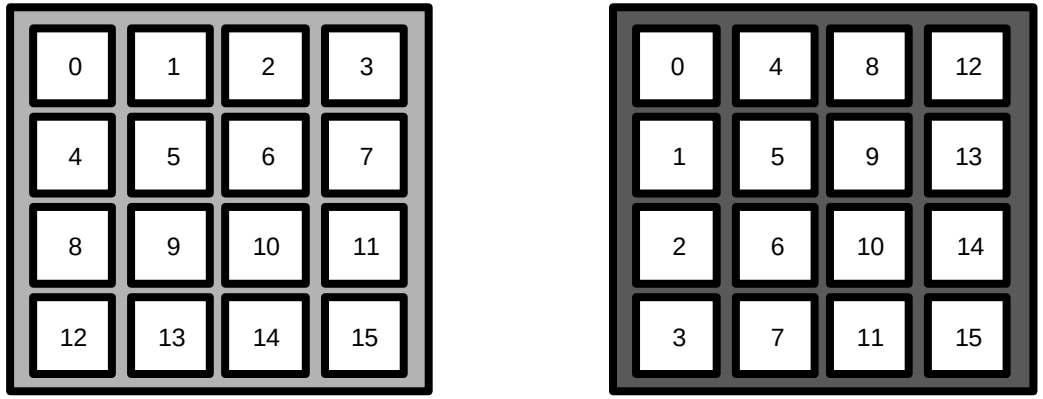
NVIDIA DEEP LEARNING INSTITUTE





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]
```



www.nvidia.com/dli