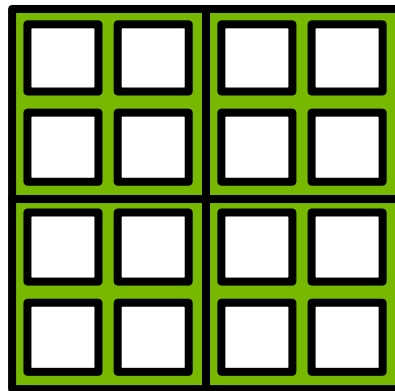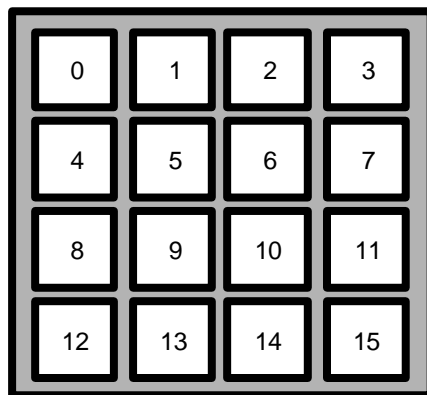# Using Shared Memory to Support Coalesced Memory Access

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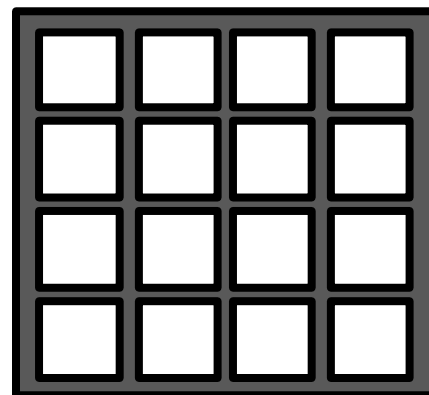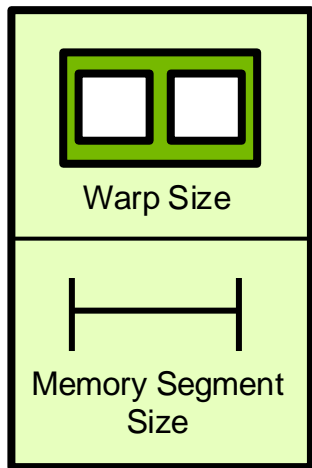
Grid

Input

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

Output

Warp Size

Memory Segment Size

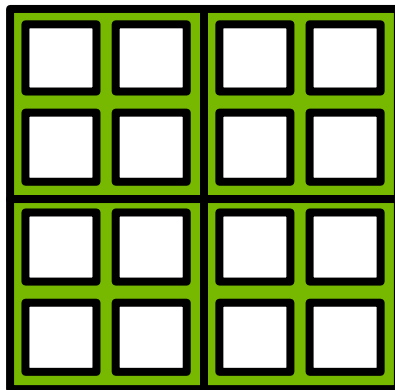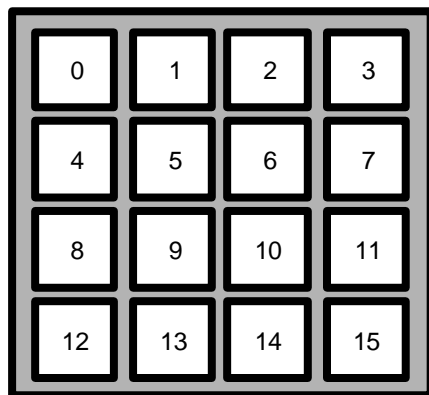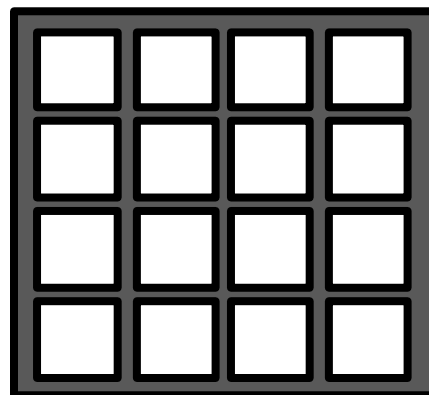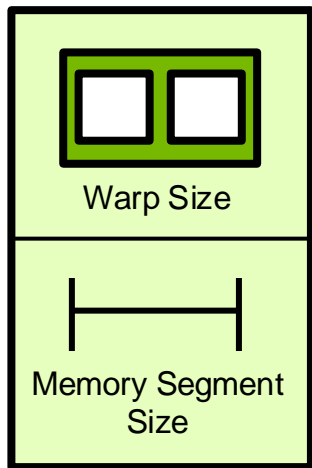## Grid

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

Input

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

Output

Warp Size

Memory Segment Size
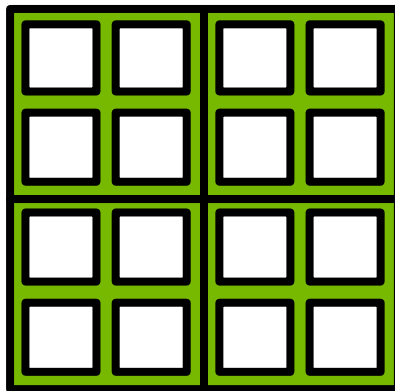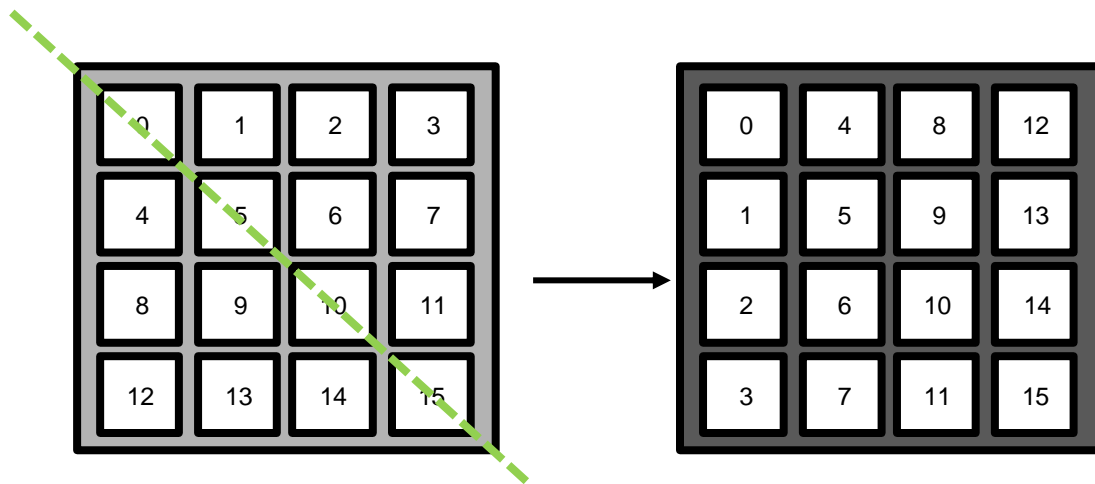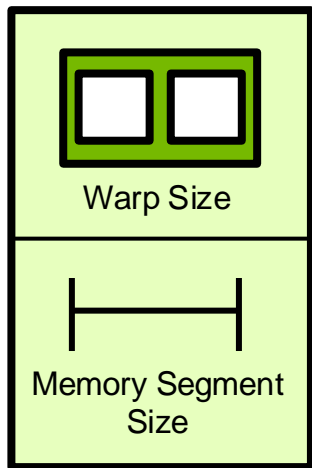
Grid

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

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

Input

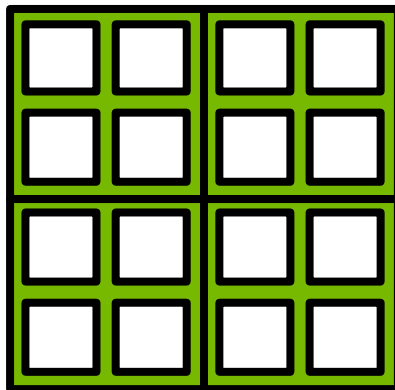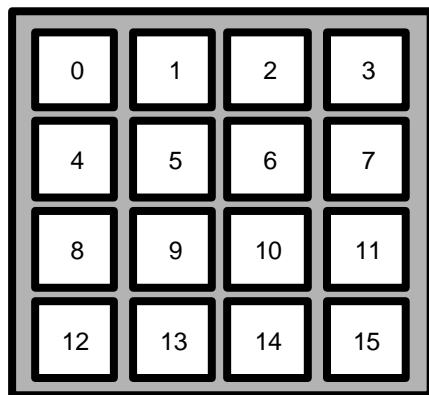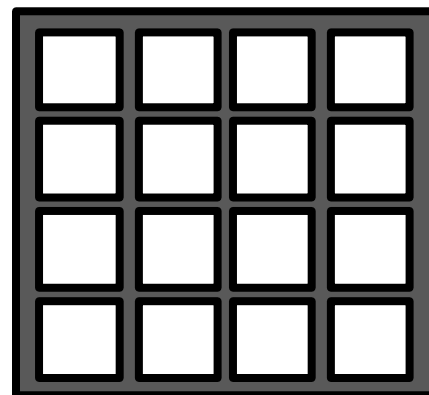| 0 | 4 | 8 | 12 |
1	5	9	13
2	6	10	14
3	7	11	15

Output

Warp Size

Memory Segment Size

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]
```

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

Input

Output

Grid

Warp Size

Memory Segment Size

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]
```

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

Input

Output

Warp Size

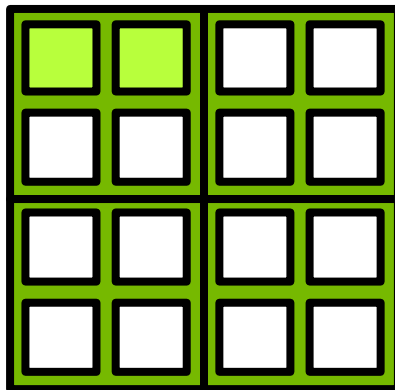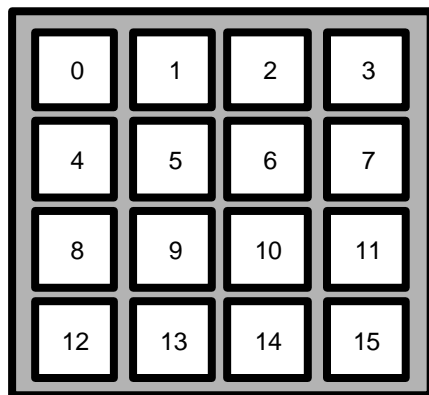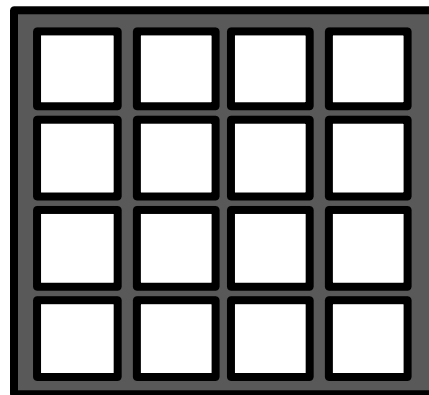Memory Segment Size

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]
```

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

Input

Output

Warp Size

Memory Segment Size

Grid

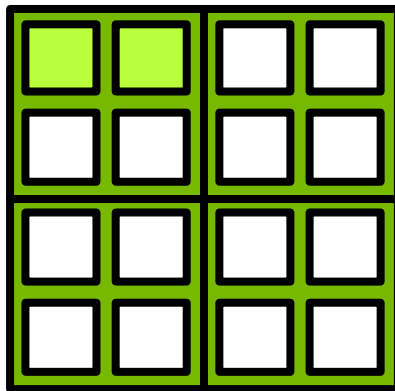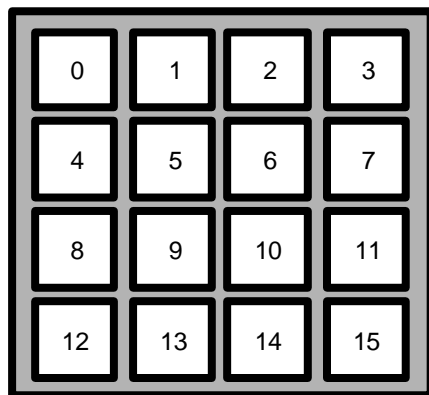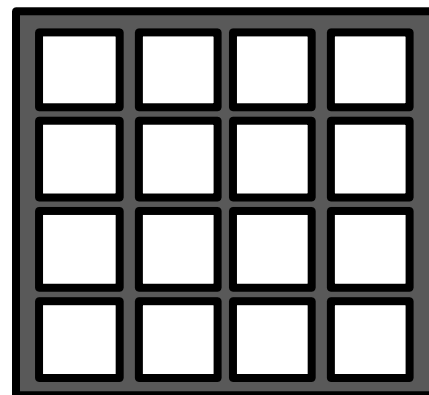Furthermore, these contiguous threads will read elements from the rows of input where data elements are contiguous

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

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

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

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

Input

Output

Warp Size

Memory Segment Size

Grid

Therefore, it makes sense that reads from input are coalesced
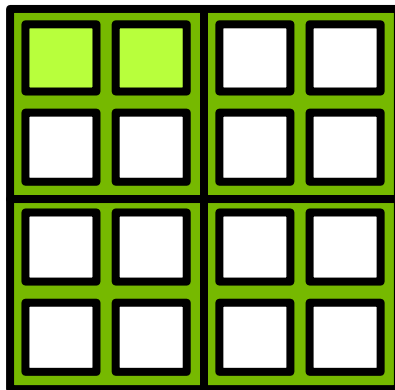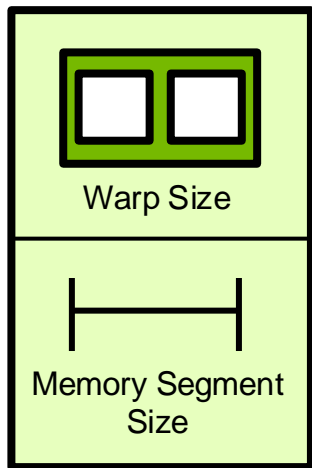
```
x = blockIdx.x * blockDim.x
        + threadIdx.x

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

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

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

Input

Output

Warp Size

Memory Segment Size
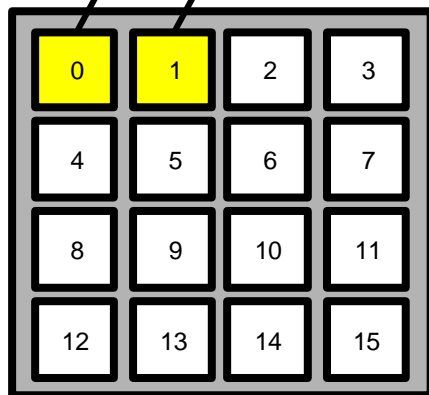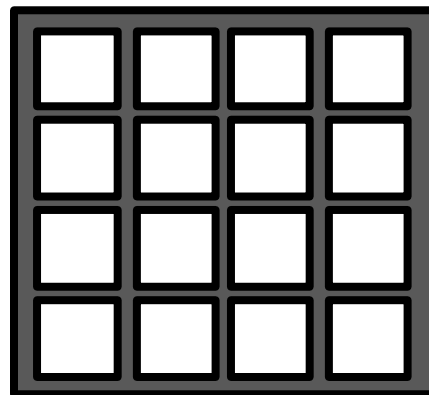
Grid

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]
```

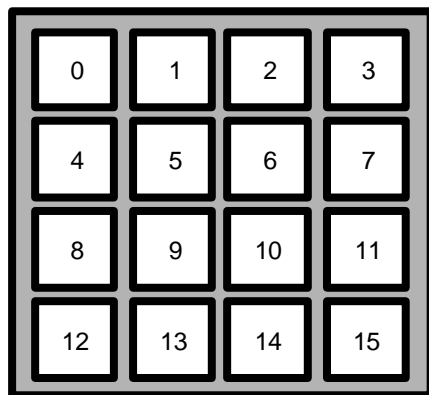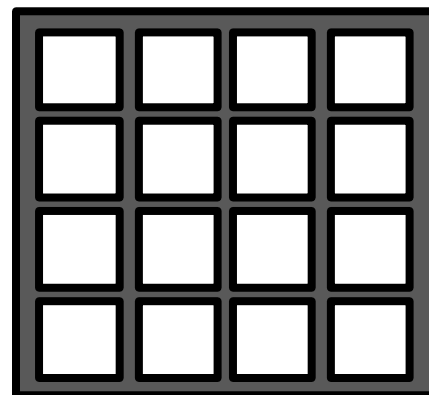| 0 | 1 | 2 | 3 |
4	5	6	7
8	9	10	11
12	13	14	15

Input

Output

Warp Size

Memory Segment Size

Grid

Here we see that contiguous threads in the same warp will be writing along a column in output
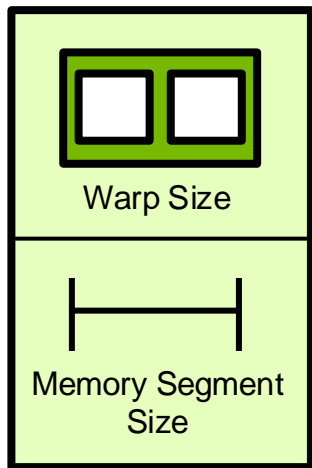
```
x = blockIdx.x * blockDim.x
        + threadIdx.x

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

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

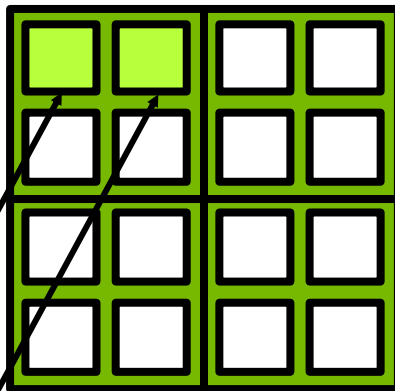| 0 | 1 | 2 | 3 |
4	5	6	7
8	9	10	11
12	13	14	15

Input

Output

Warp Size

Memory Segment Size
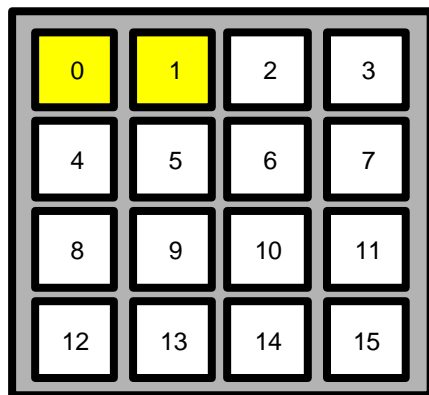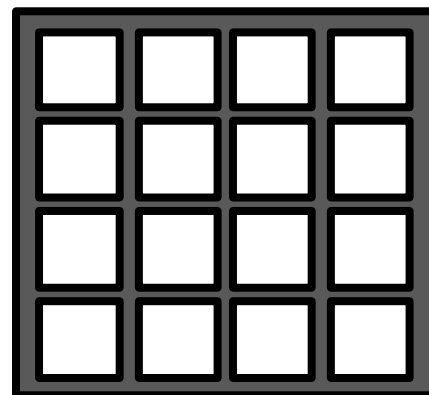
Grid

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

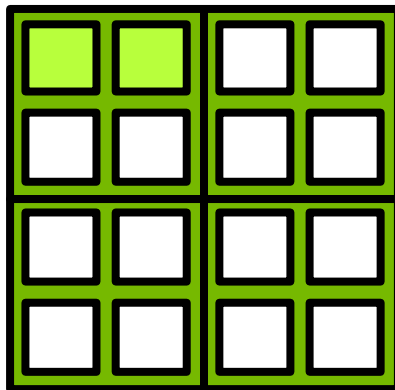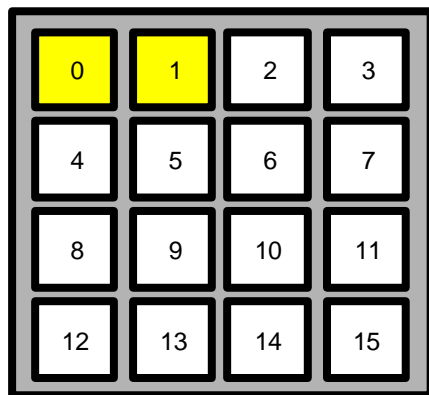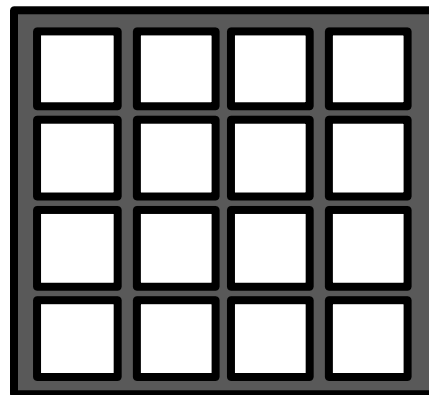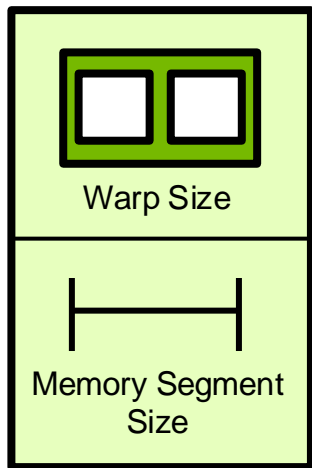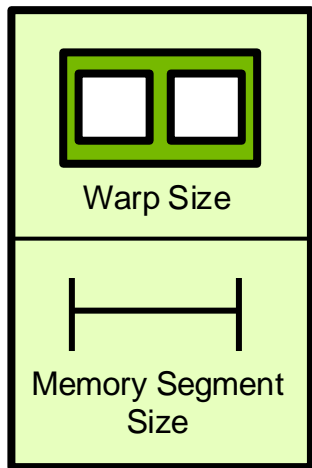## Warp Size

Memory Segment Size

Shared

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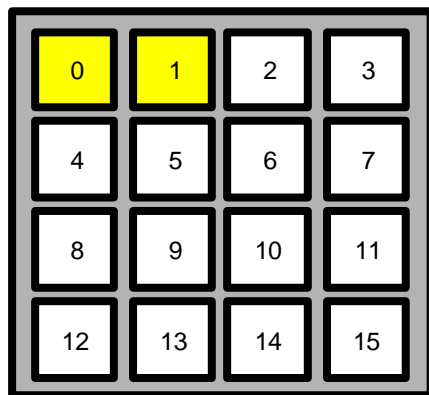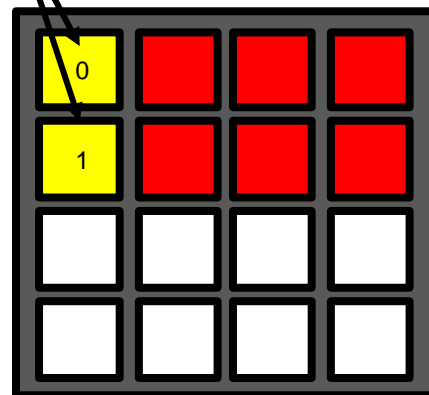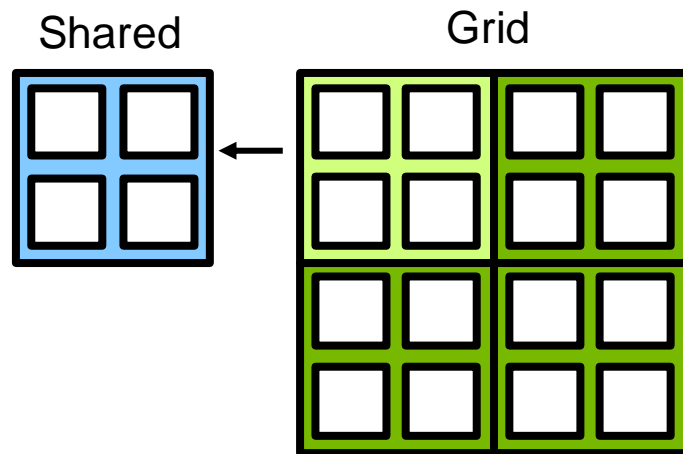
We can use shared memory to make coalesced reads and writes. Here, each block will allocate a (2,2) shared memory tile

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

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

Input

Output

Warp Size

Memory Segment Size

Shared

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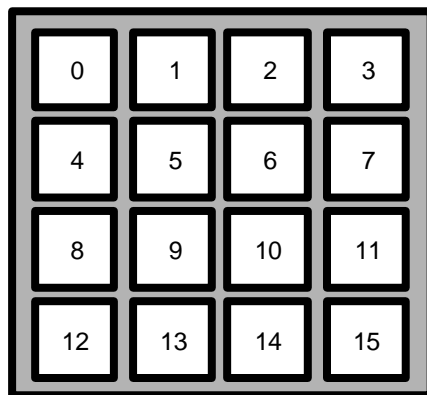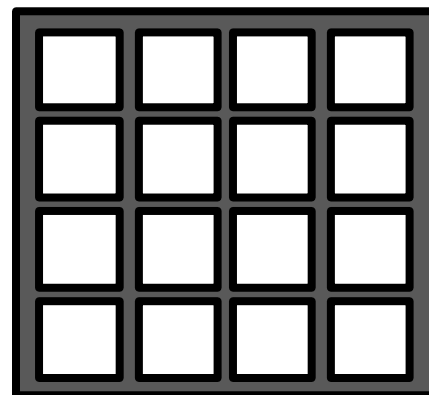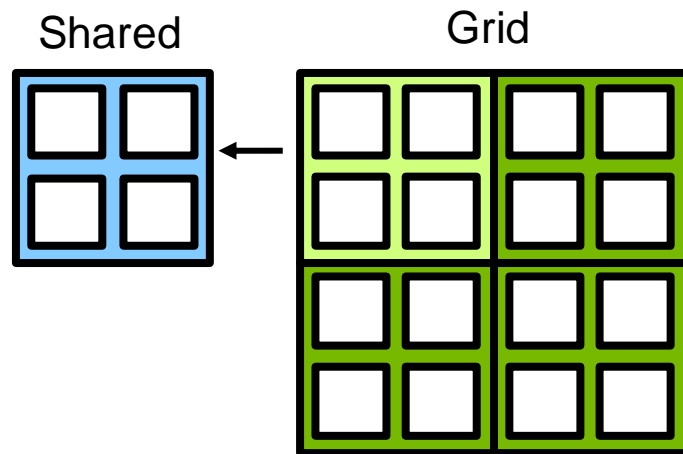
(It is worth reminding that in our slides, to preserve space, 2 threads is a warp length. A real warp is 32 threads)

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

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

Input

Output

Warp Size

Memory Segment Size
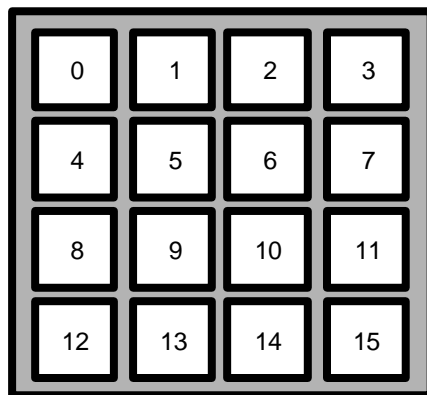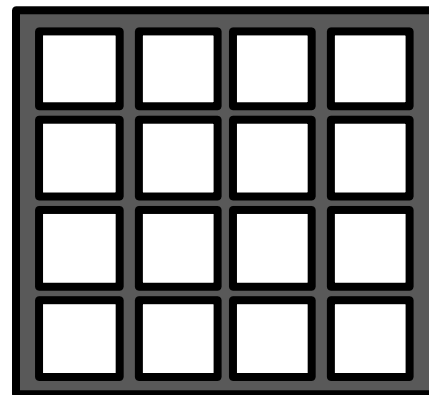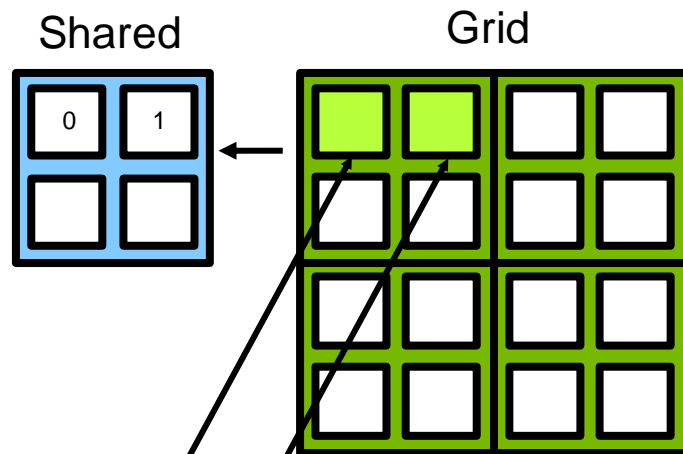
Shared

Grid

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]
```
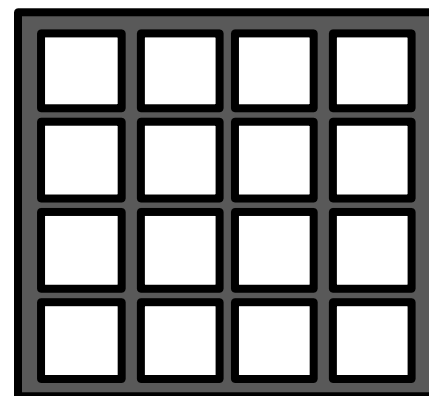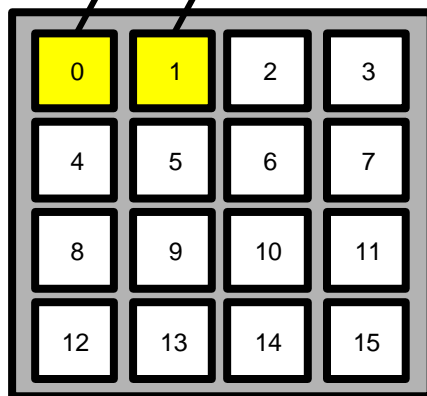
Input

Output

Shared

Grid

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]
```

Warp Size

Memory Segment Size

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

Input

Output

Warp Size

Memory Segment Size

Shared

| 0 | 1 |
| 4 | 5 |
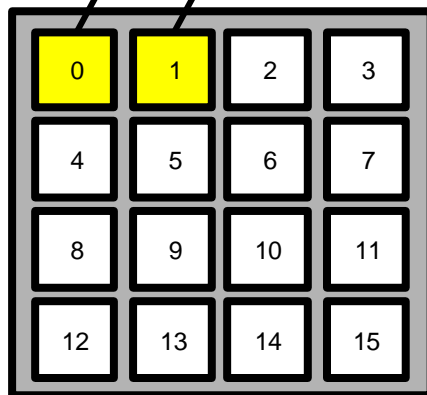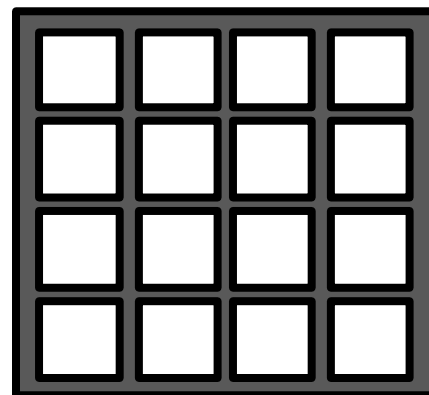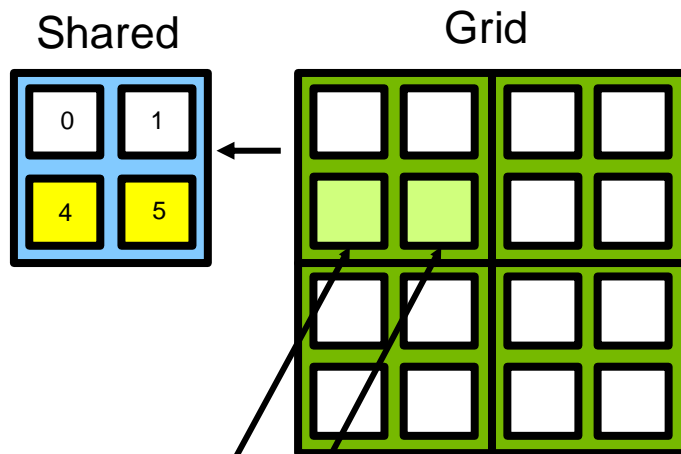
Grid

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()
```

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

Input

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
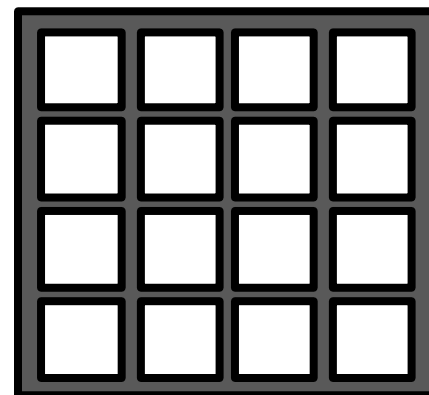
Grid

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
```

Input

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

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
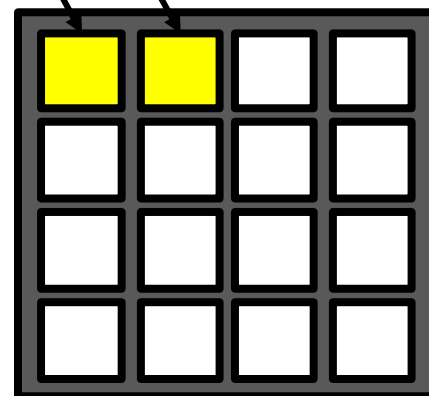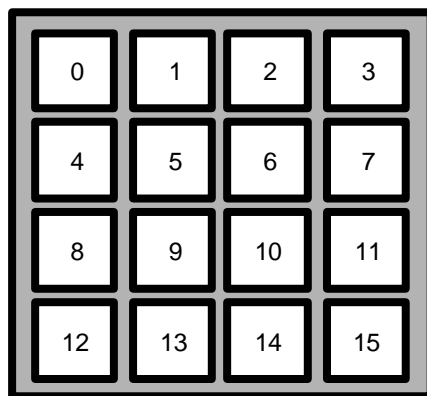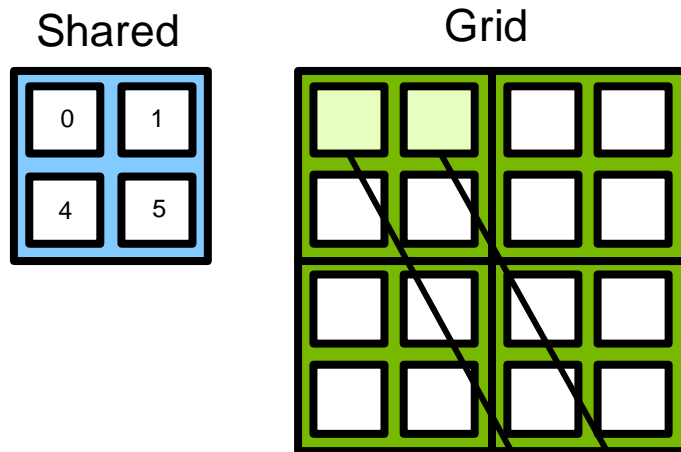
Grid

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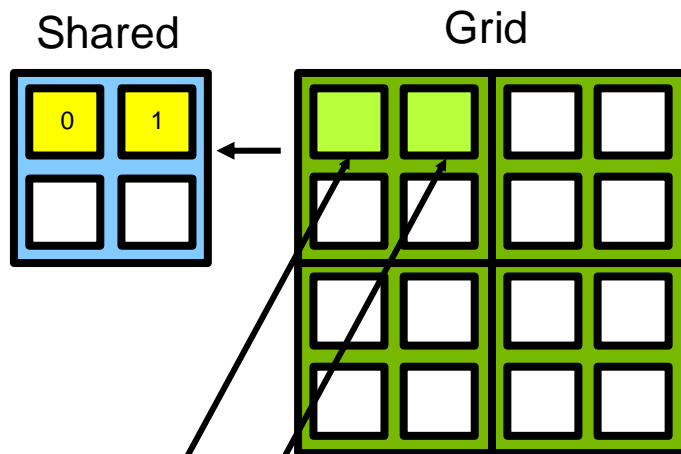
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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
```

Input

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

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
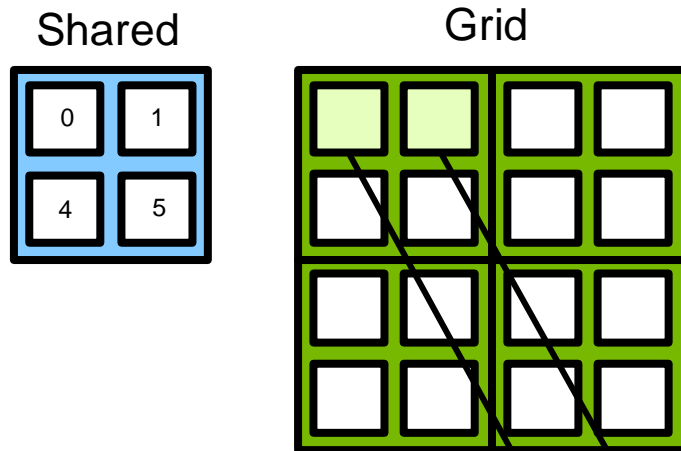
Grid

...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

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

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5

Grid

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]
```
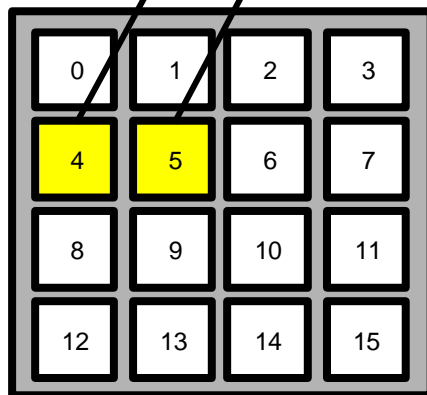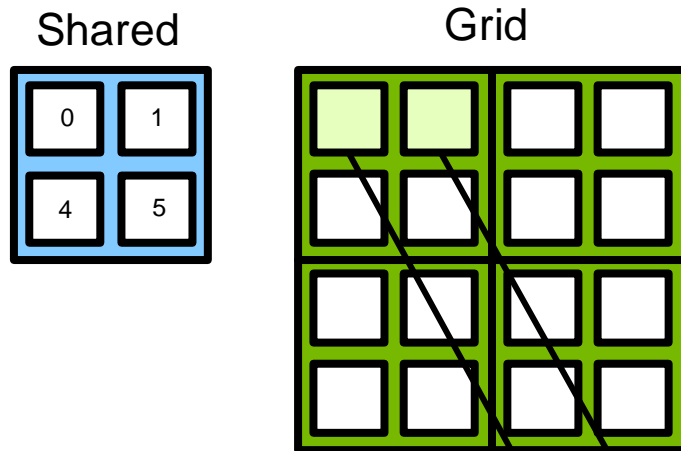
Input

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

Output

0	4		

Warp Size

Memory Segment Size

Shared

0	1
4	5

Grid

Input

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

Output

0	4		
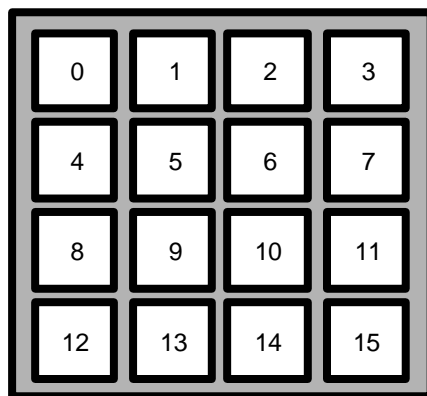
(There's more to come about efficient reads/writes to/from shared memory, but for now know that reading across the column in shared memory has very low impact compared to doing so with global memory)

```
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]
```
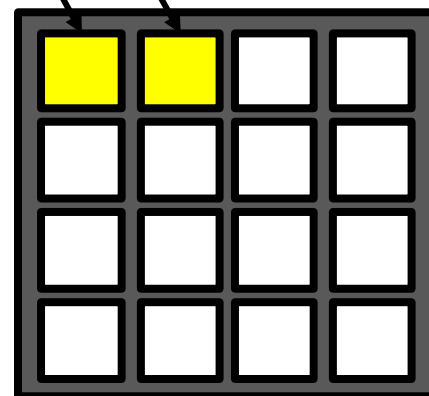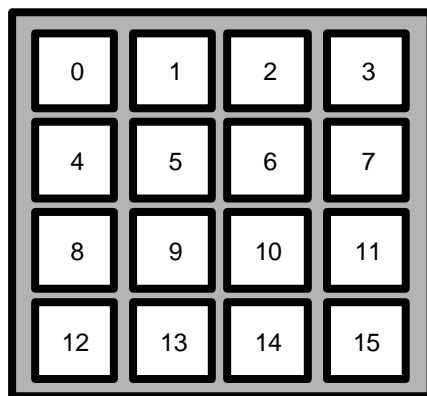
Warp Size

Memory Segment Size

Shared   Grid   Shared

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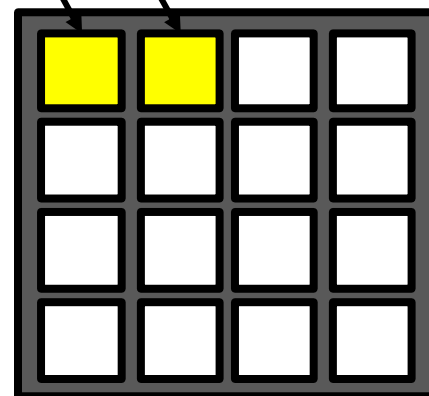
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Input

Output

Warp Size

Memory Segment Size

Shared

Grid

Shared

Input

Output

In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory
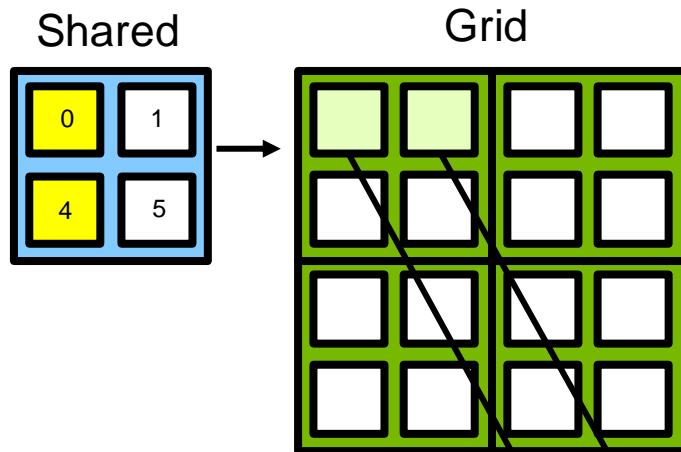
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```
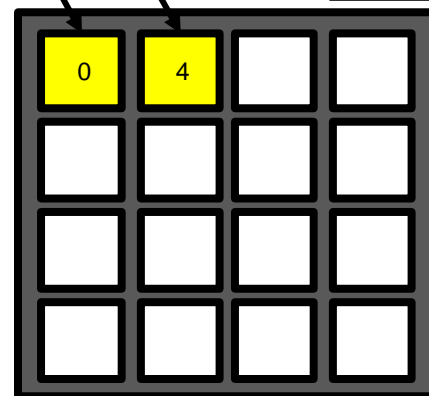
Warp Size

Memory Segment Size

Shared

Grid

Shared

| 0 | 1 |
| 4 | 5 |

In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory
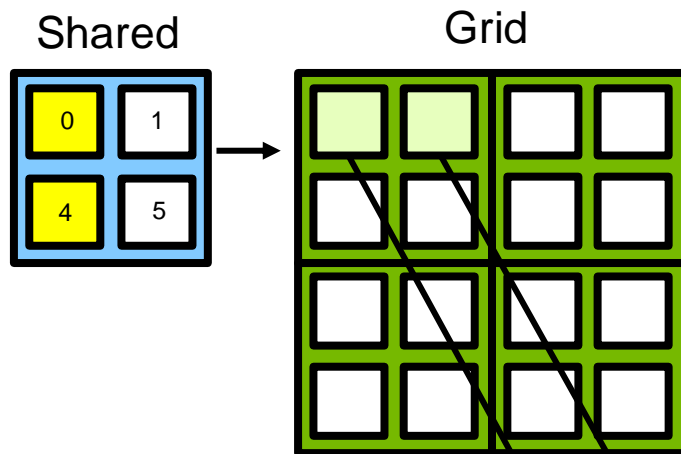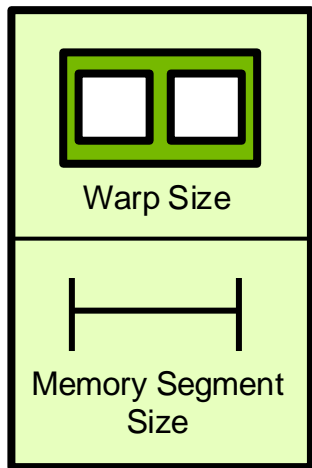
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

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

Input

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9

Grid

Shared

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]
```
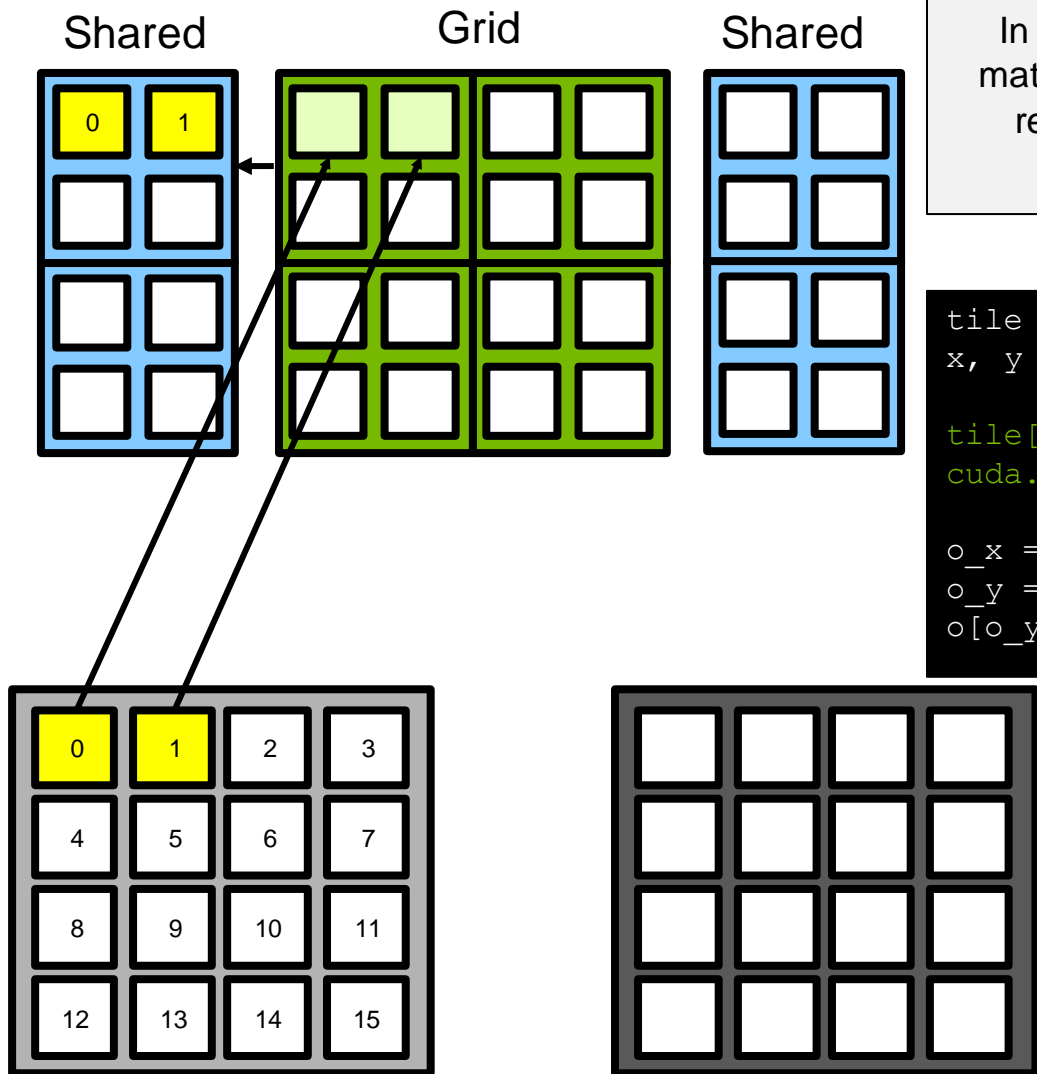
Input

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

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9
12	13

Grid

Shared

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]
```
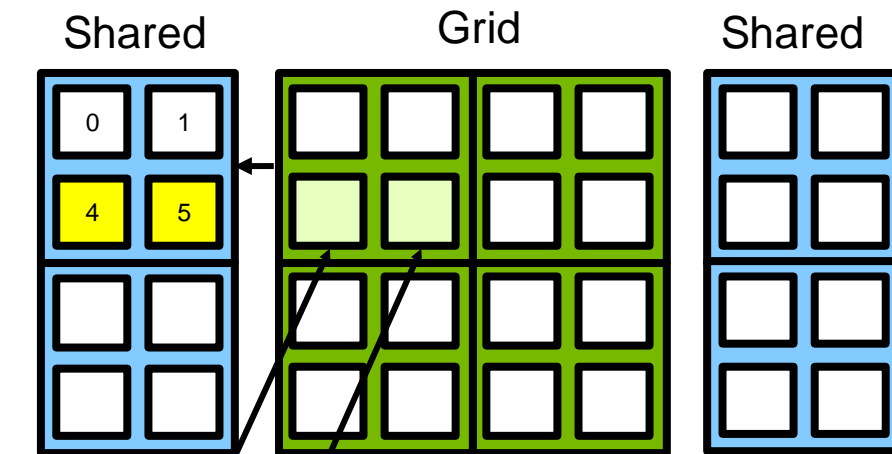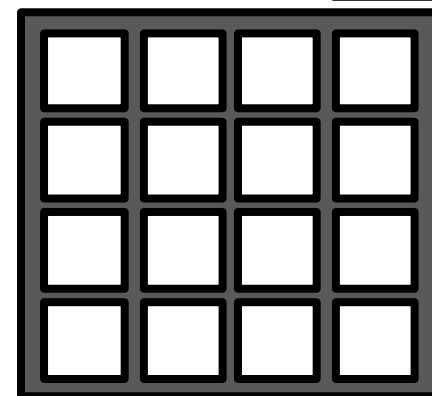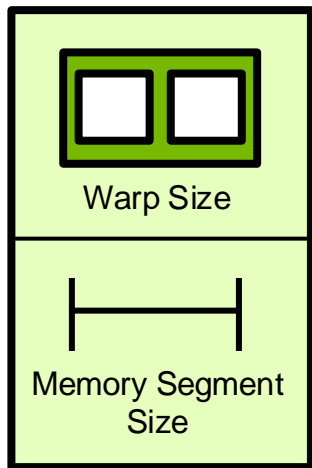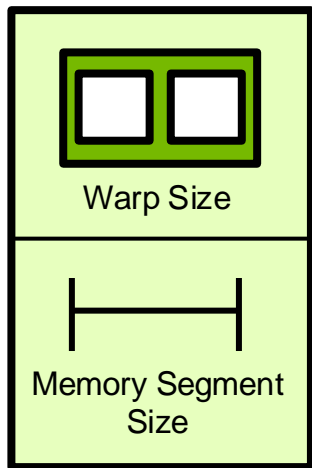
Input

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

Output

Warp Size

Memory Segment Size

Shared

Grid

Shared

In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory
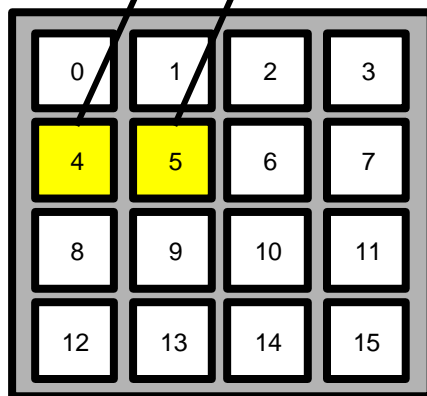
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```
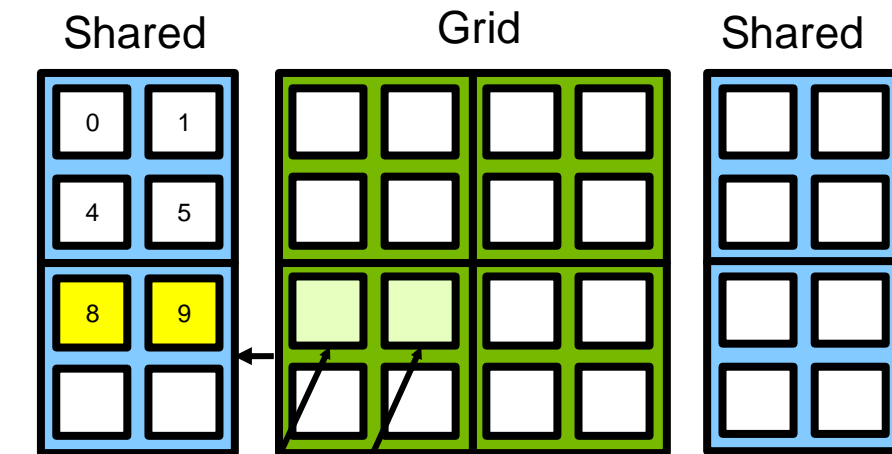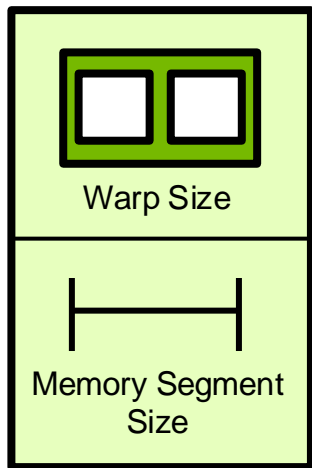
Input

Output

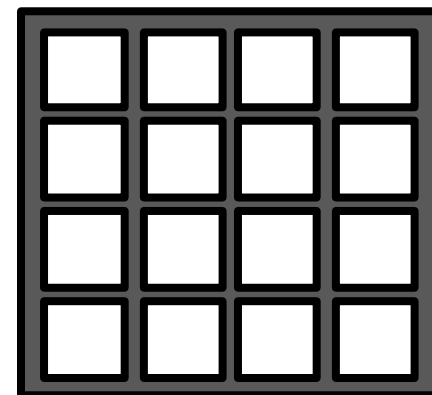Warp Size

Memory Segment Size
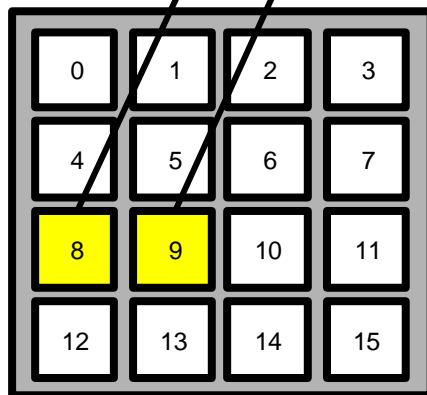
Shared

Grid

Shared

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]
```
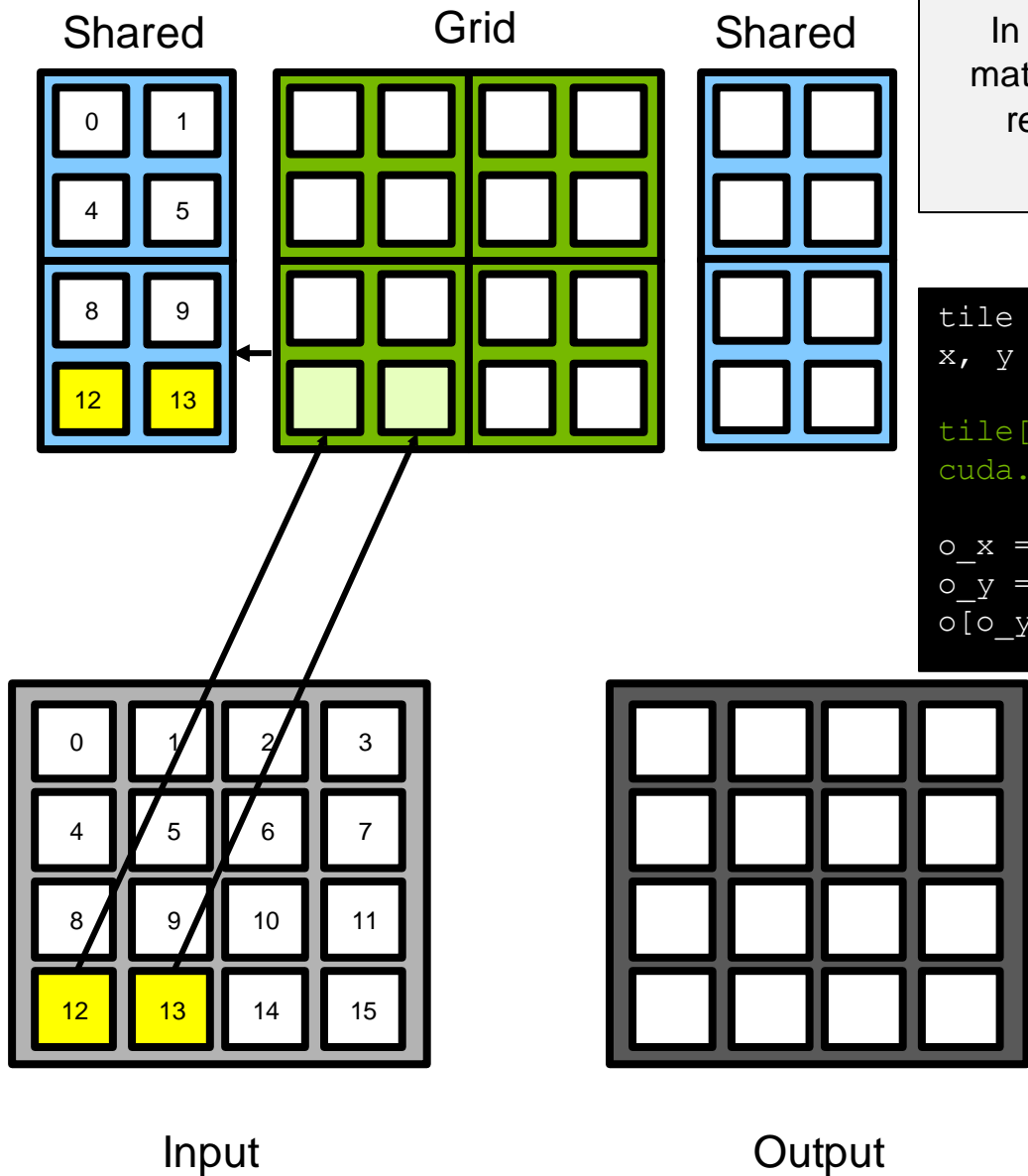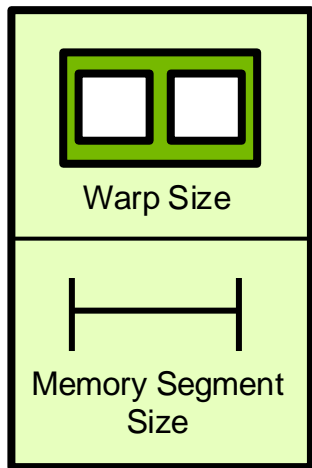
Input

Output

Warp Size

Memory Segment Size

Shared

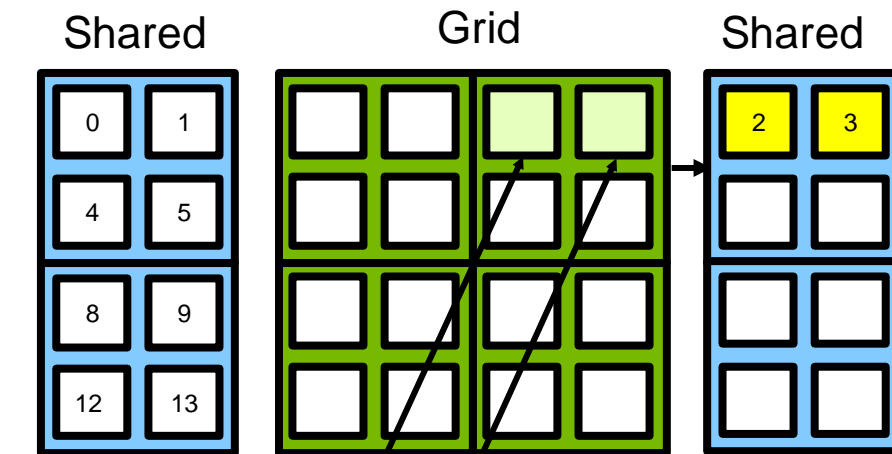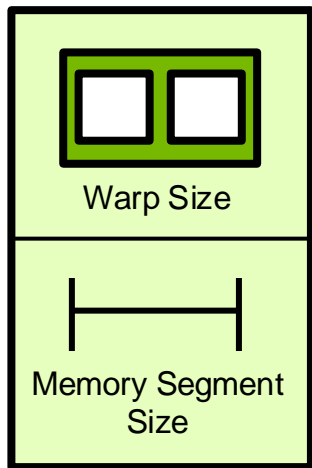| 0 | 1 |
4	5
8	9
12	13

Grid

Shared

2	3
6	7
10	11

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

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

Output

Input

Output

Warp Size

Memory Segment Size

Shared
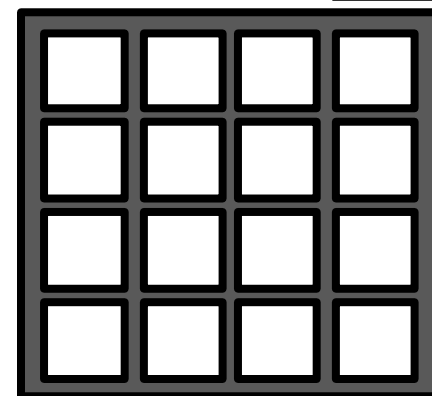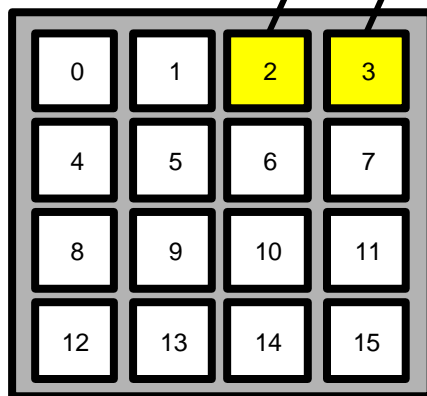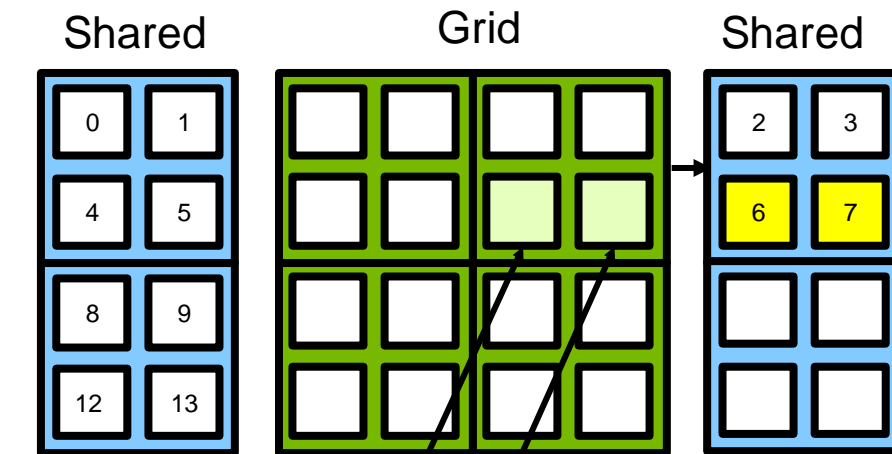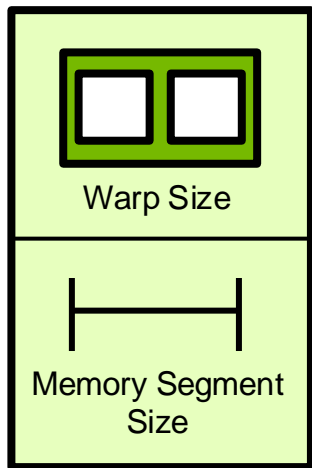
Grid

Shared

Input

Output

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]
```

Warp Size

Memory Segment Size

Shared

Grid

Shared

Input

Output

In this way we can transpose the matrix while making fully coalesced reads from and writes to global memory
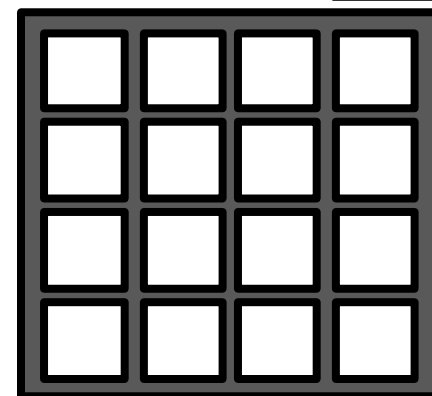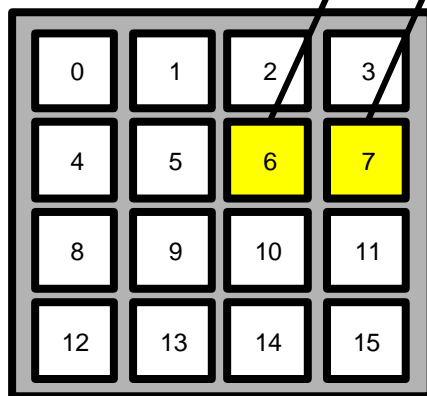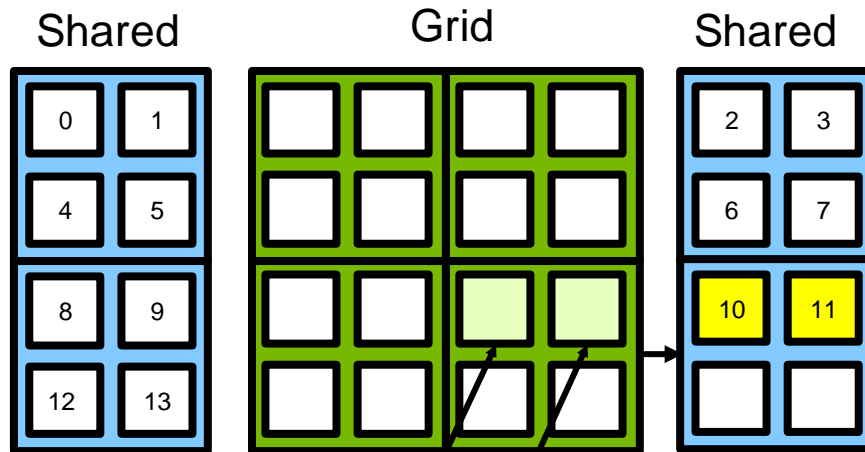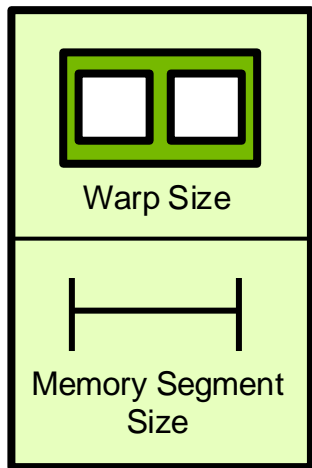
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

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]
```

Warp Size

Memory Segment Size

Shared

Grid

Shared

0	1
4	5

8	9
12	13

2	3
6	7

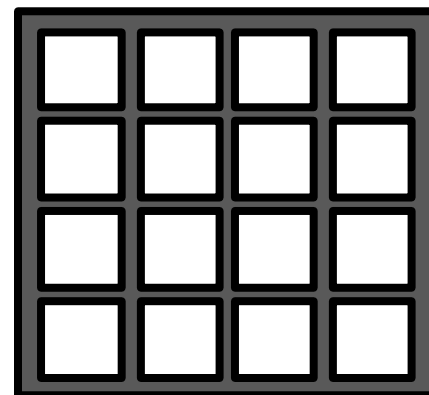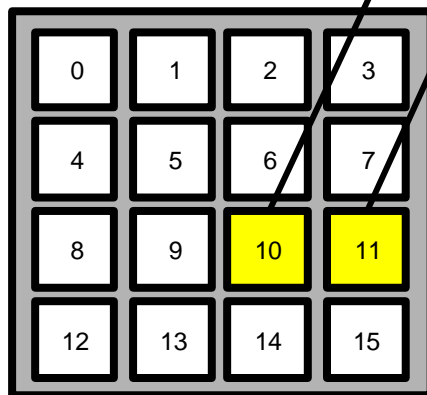|	
10	11
14	15

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]
```
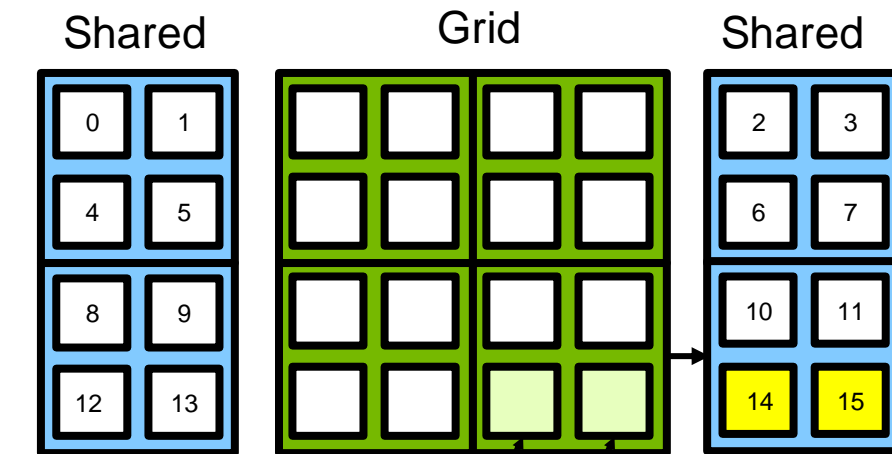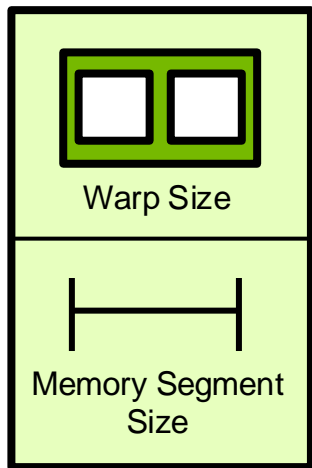
Input

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

Output

0	4		
1	5		

Warp Size

Memory Segment Size
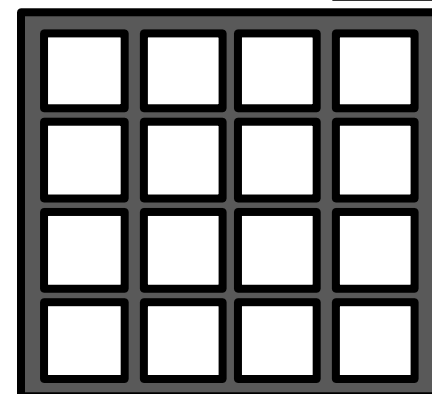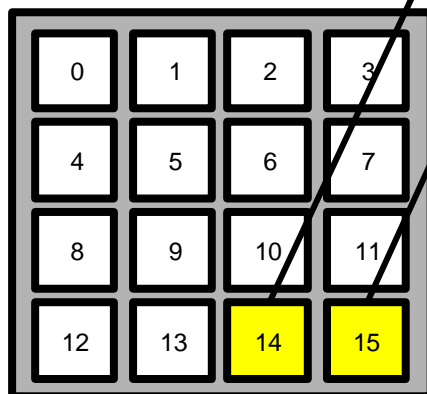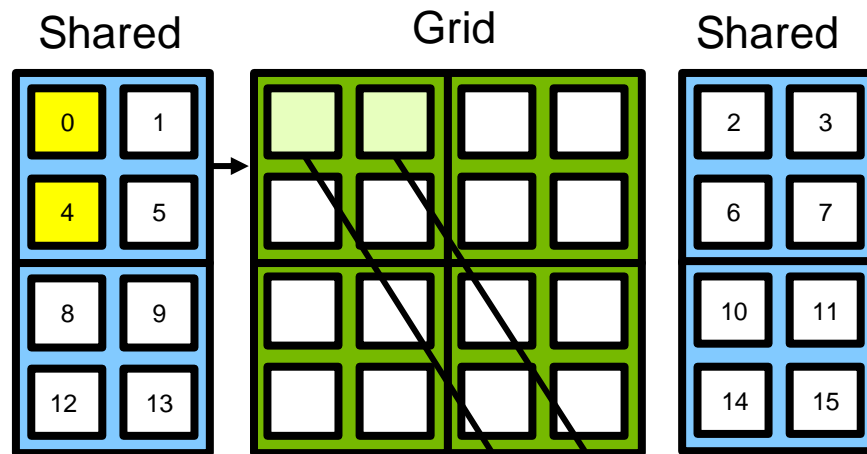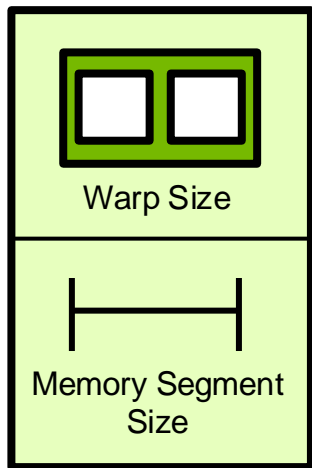
Shared

Grid

Shared

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

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	**9**
12	**13**

Grid

Shared

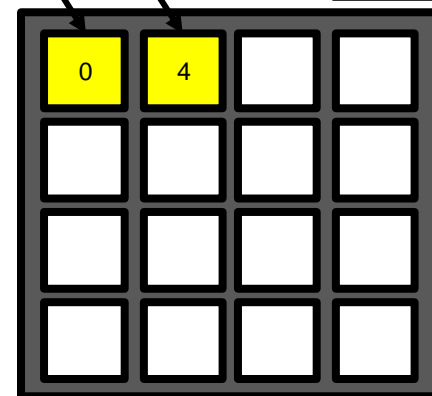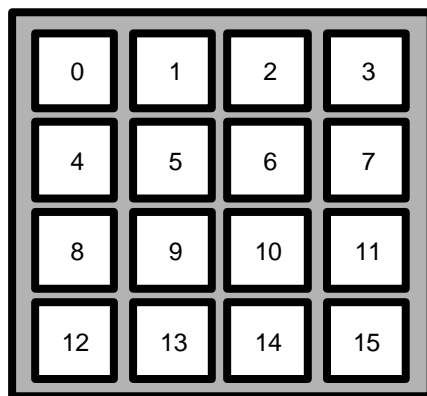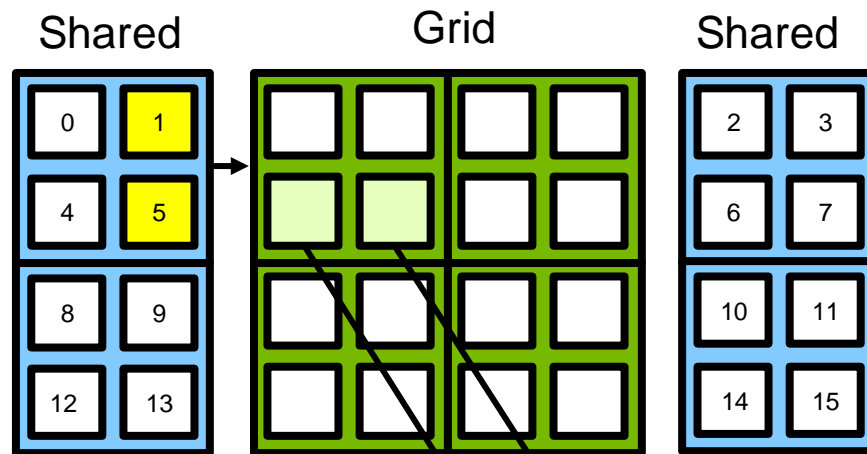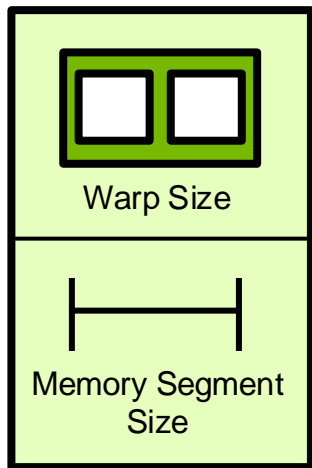| 2	3
6	7
10	11
14	15

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

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

Output

0	4	8	12
1	5	**9**	**13**

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9
12	13

Grid

Shared

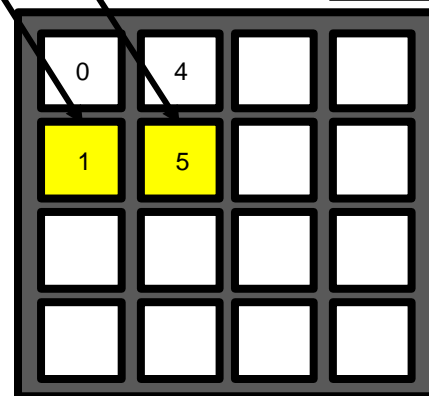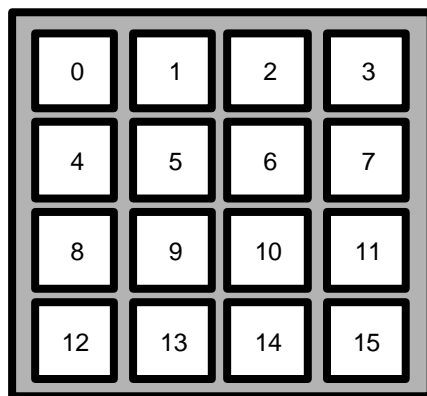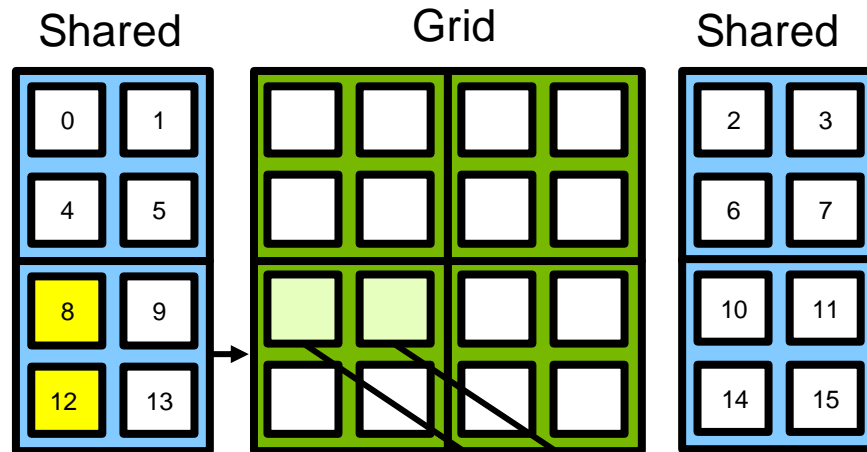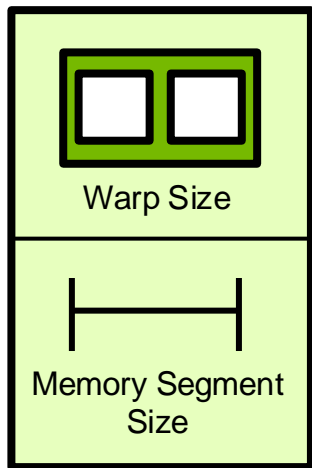|	
2	3
6	7
10	11
14	15

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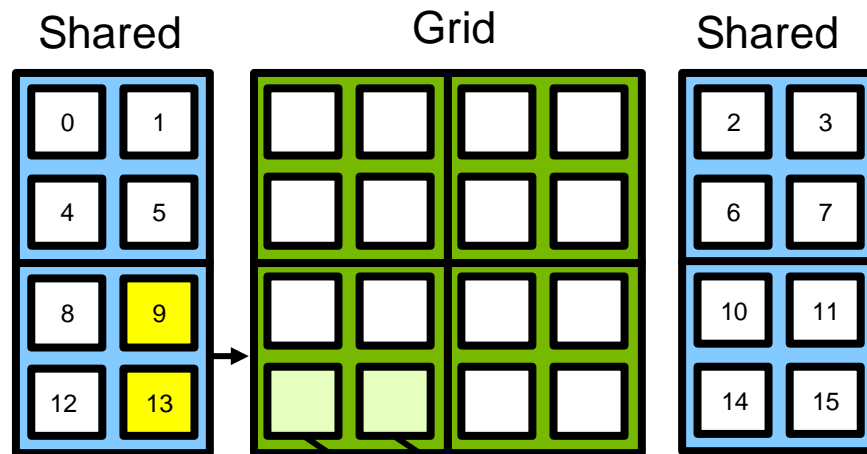
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Input

0	4	8	12
1	5	9	13
2	6		

Output

Warp Size

Memory Segment Size

Shared

Grid

Shared

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]
```
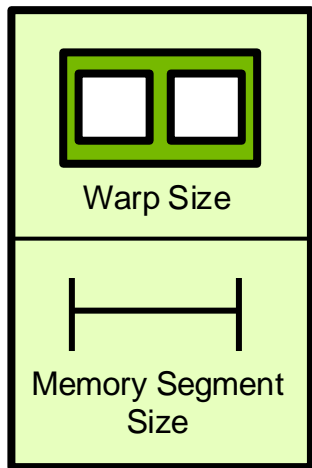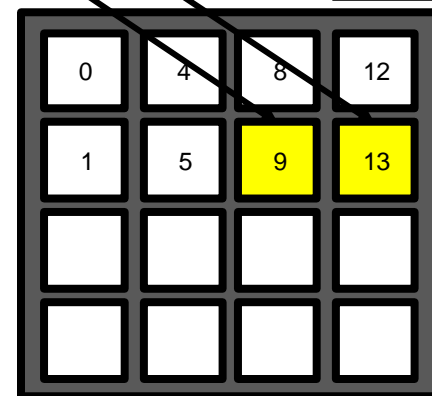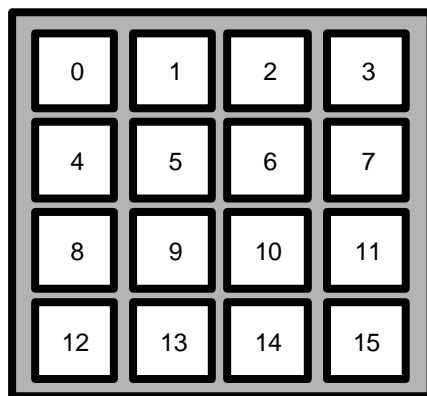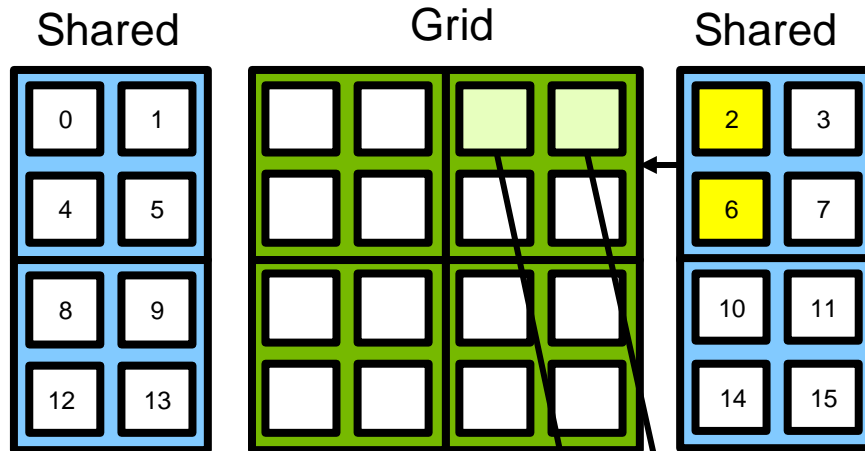
Input

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9
12	13

Grid

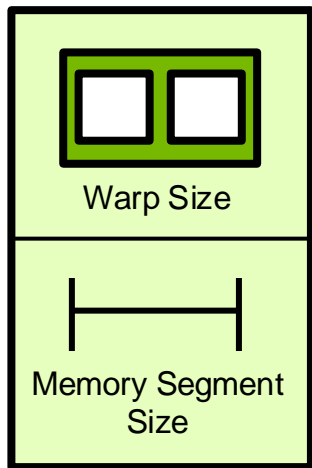Shared

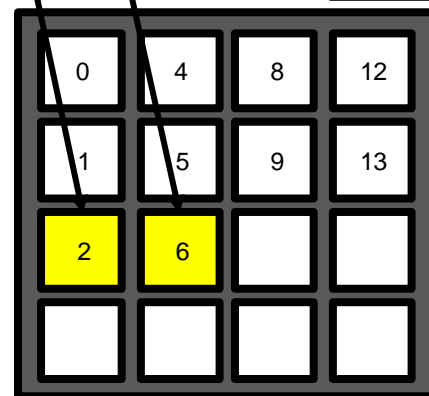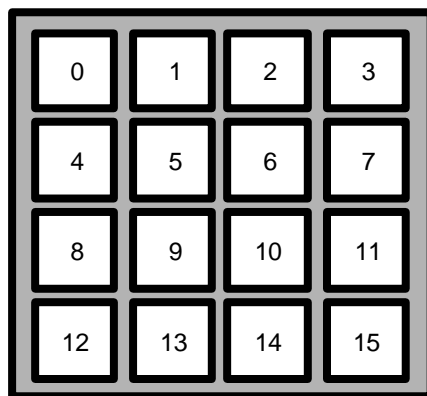| 2 | 3 |
6	7
10	11
14	15

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]
```
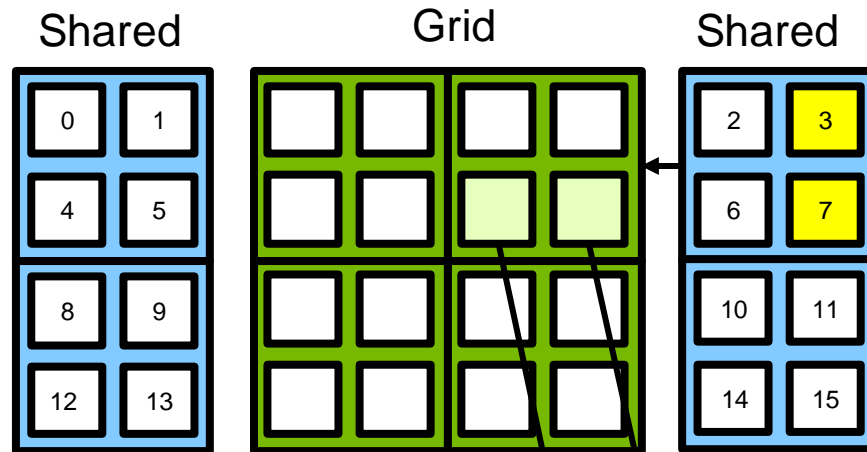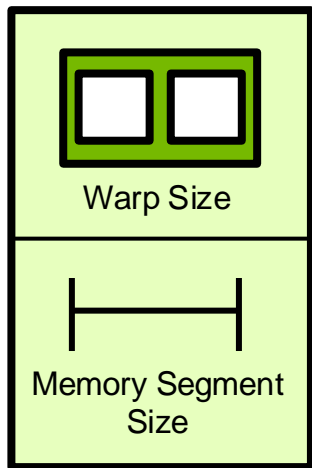
Input

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

Output

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

Input

Output

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9
12	13

Grid

Shared

2	3
6	7
10	11
14	15

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]
```
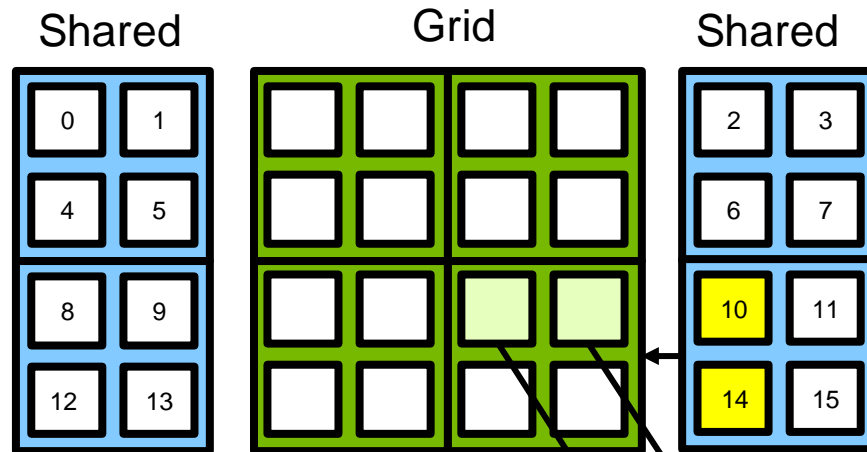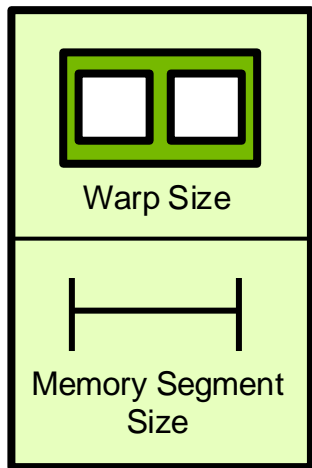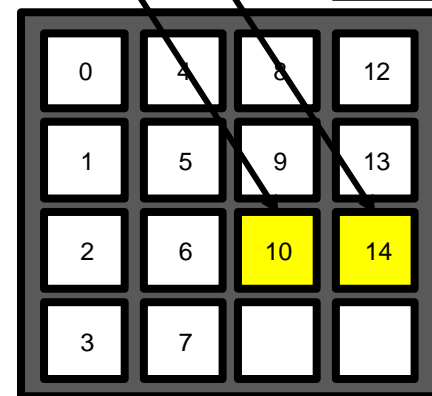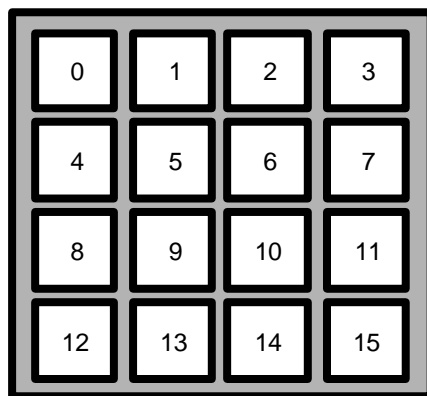
Input

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

Output

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

Warp Size

Memory Segment Size

Shared

0	1
4	5
8	9
12	13

Grid

Shared

2	3
6	7
10	11
14	15

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]
```
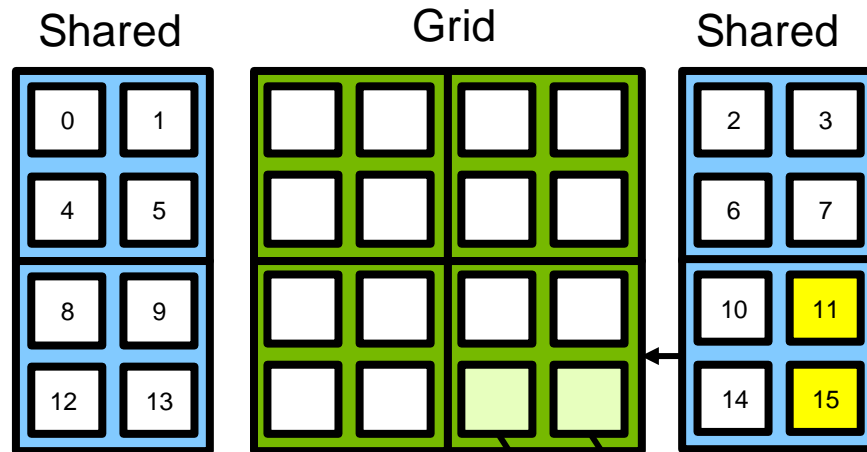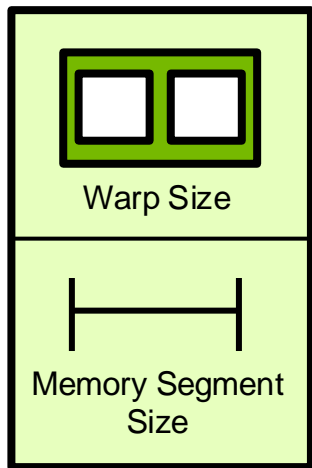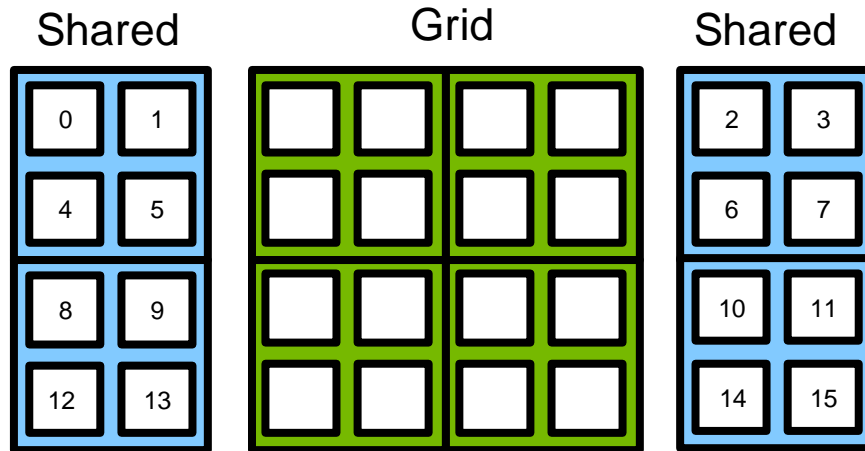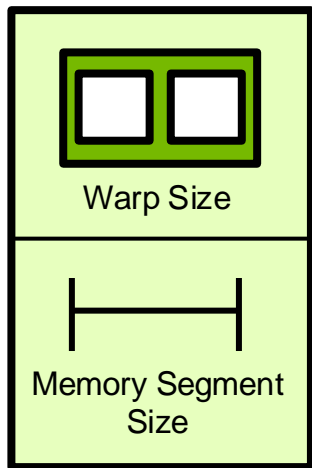
Input

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

Output

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