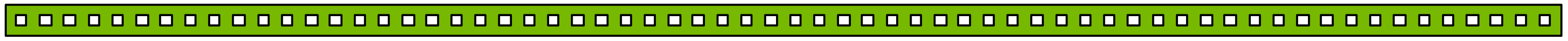
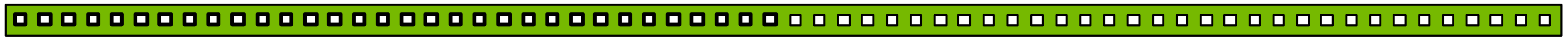
Memory Coalescing

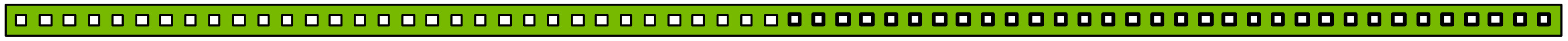
Recall that thread blocks are divided into **warps** of 32 threads



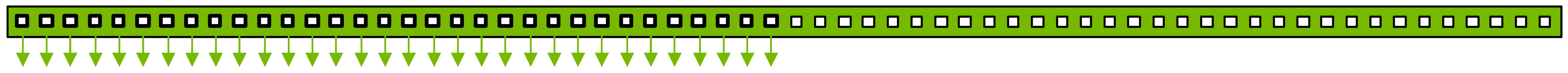
Recall that thread blocks are divided into **warps** of 32 threads



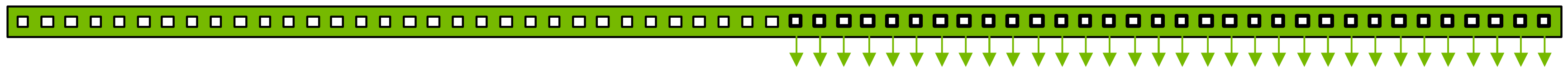
Recall that thread blocks are divided into **warps** of 32 threads



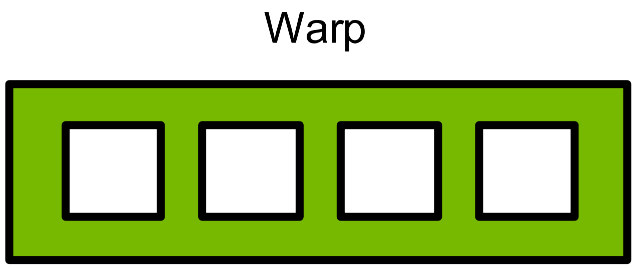
Instructions are issued in parallel at the warp level of 32 threads



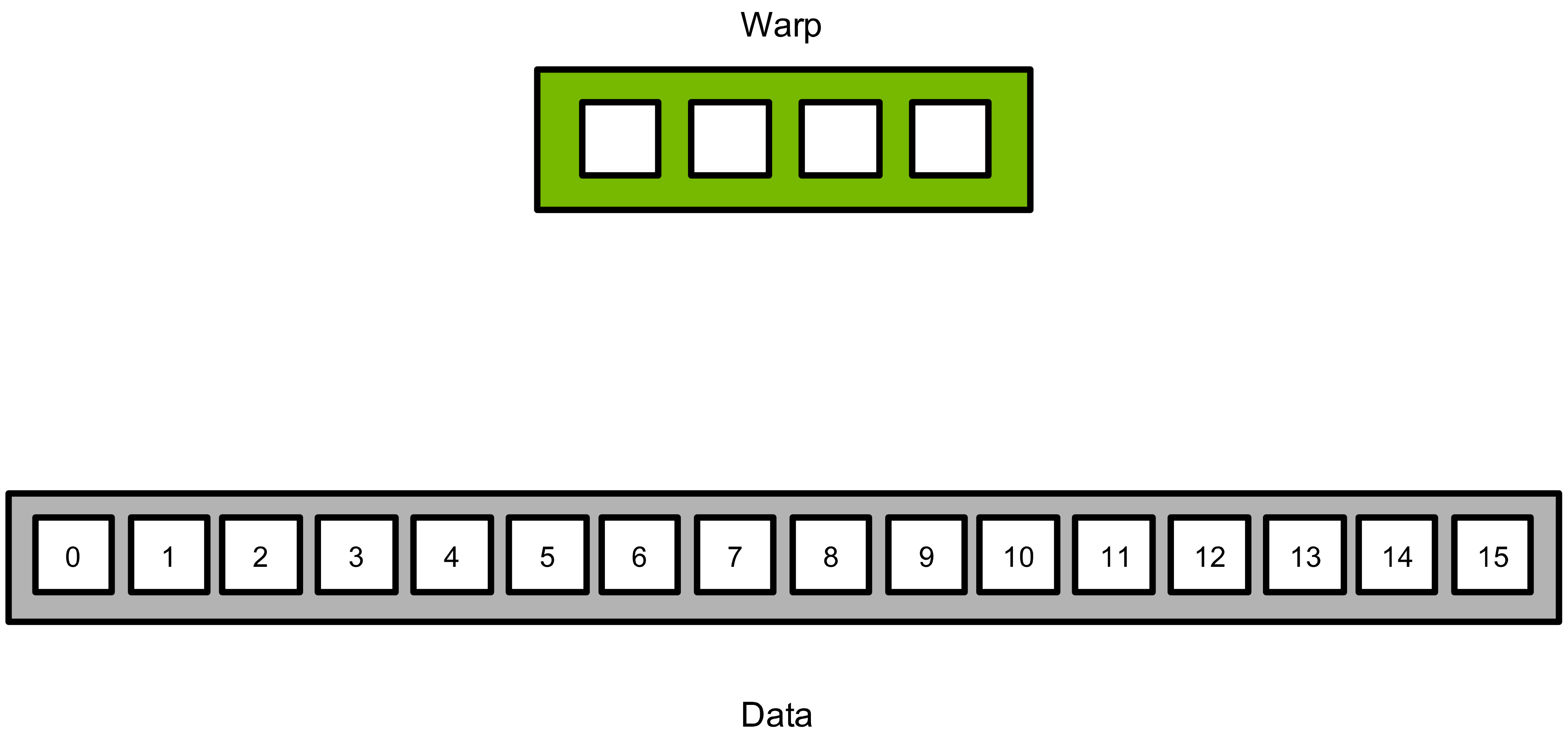
Instructions are issued in parallel at the warp level of 32 threads



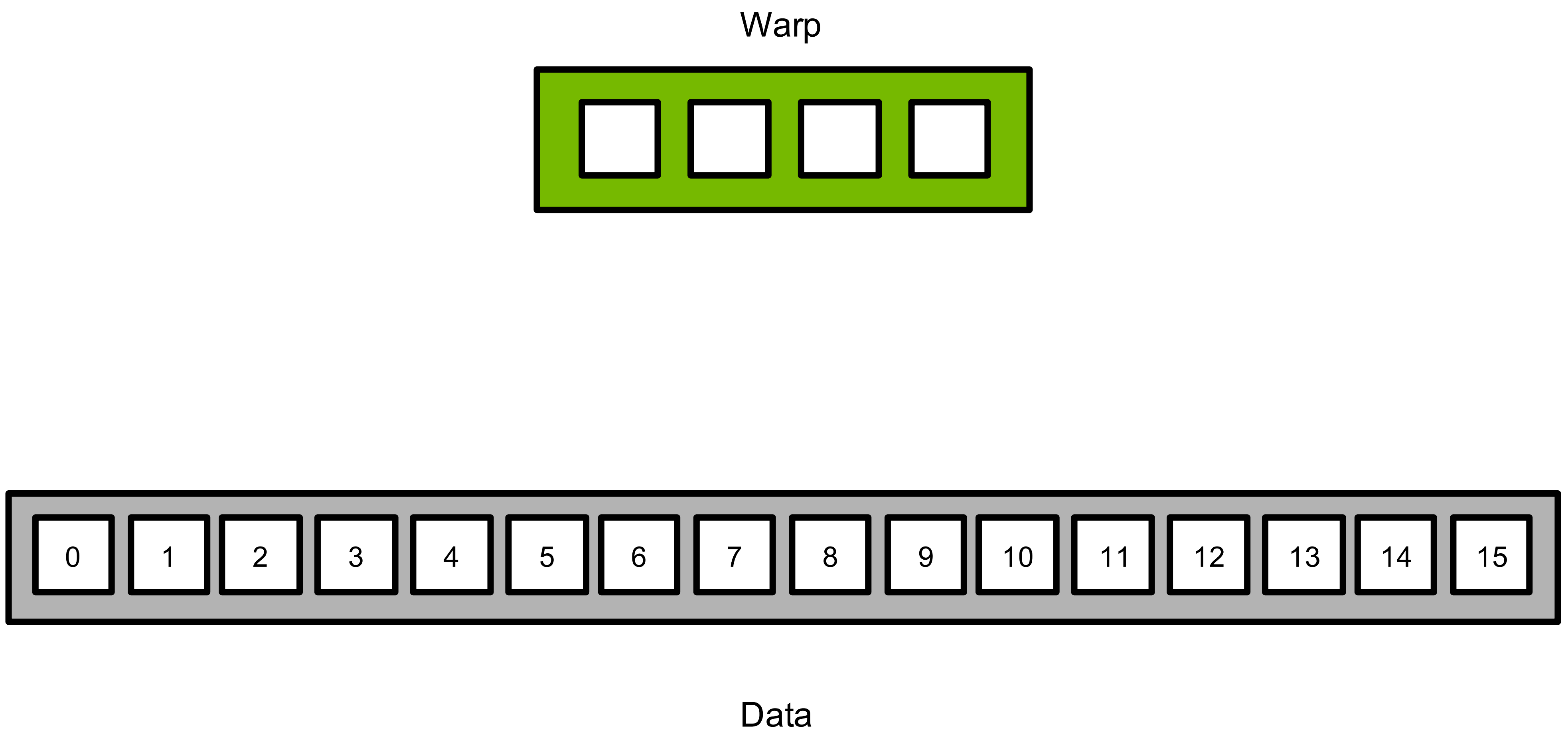
For space on these slides, we will treat just 4 threads as a warp



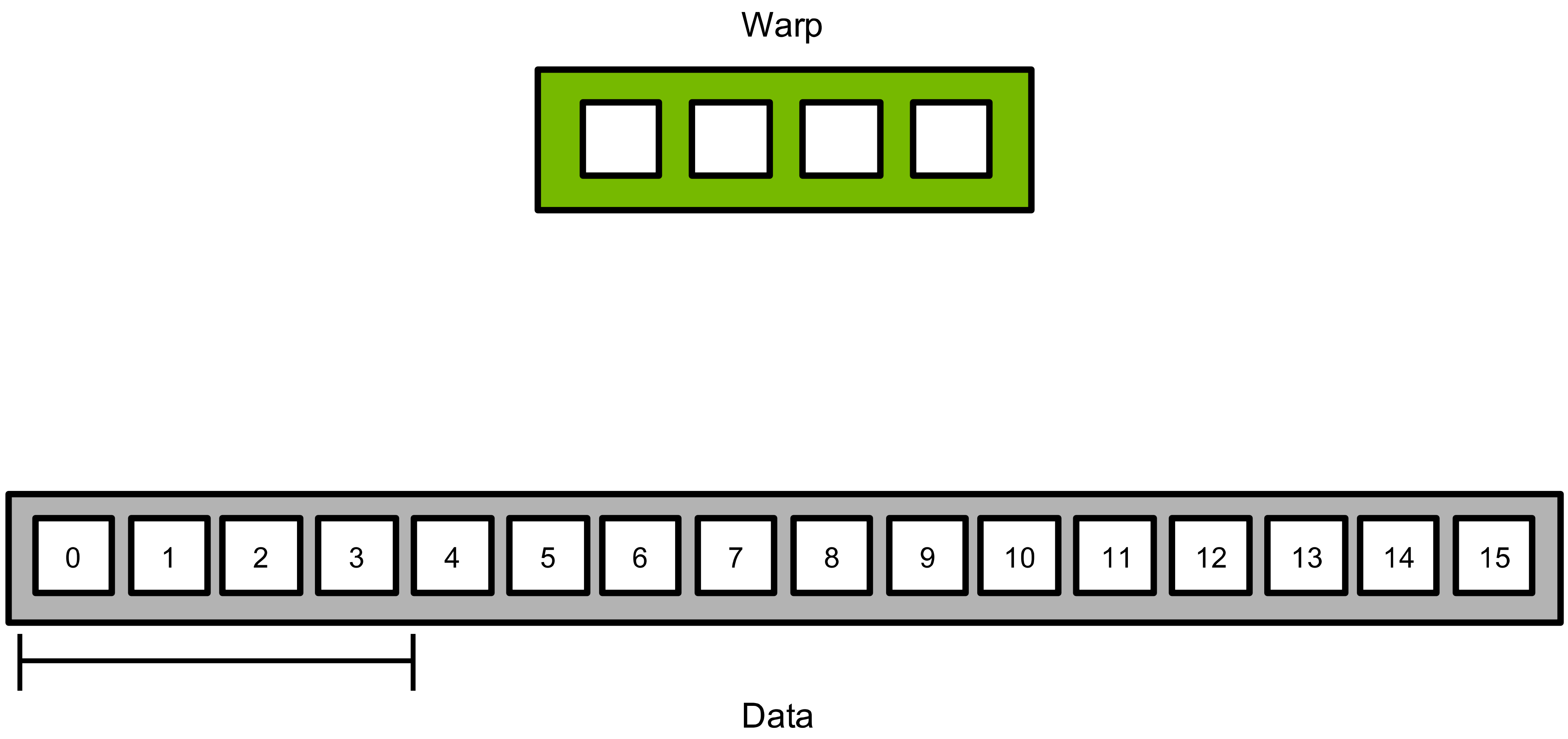
Data is transferred to and from global device memory in 32-byte segments*



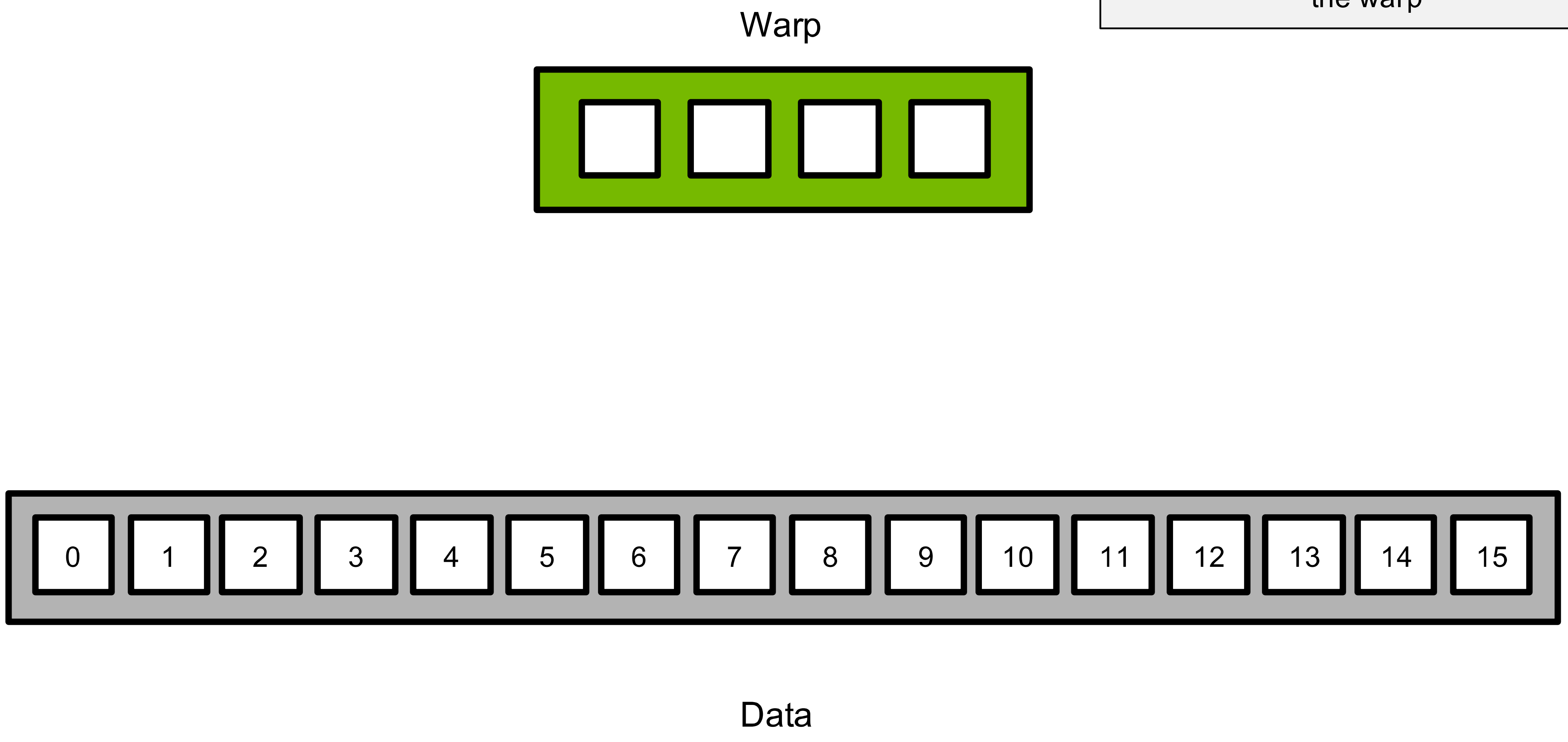
(* If the data is in the L1 cache it will be transferred in 128-byte cache lines – see the notebook for details)



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

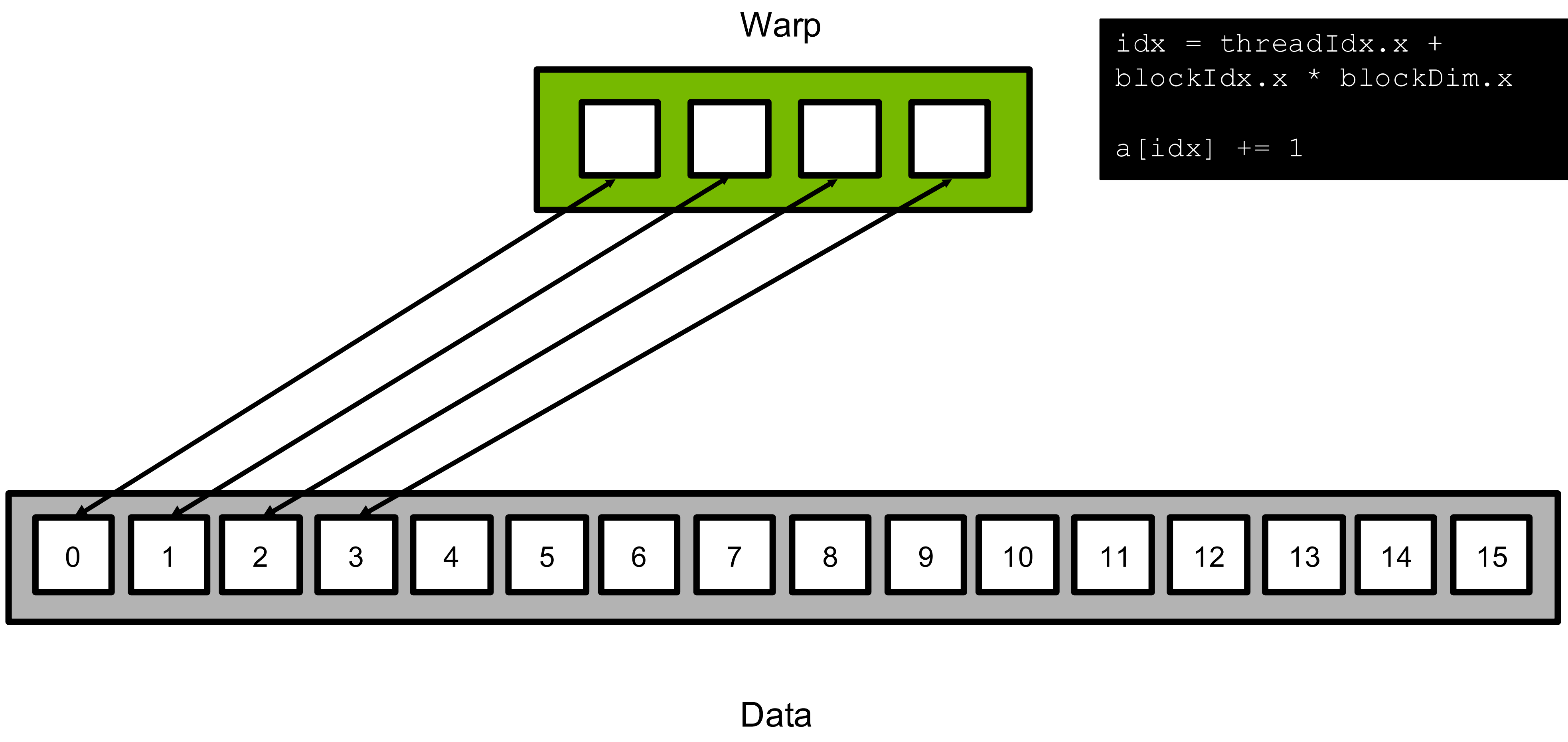


The memory subsystem will attempt to minimize the number of lines required to fulfill the read/write requirements of the warp



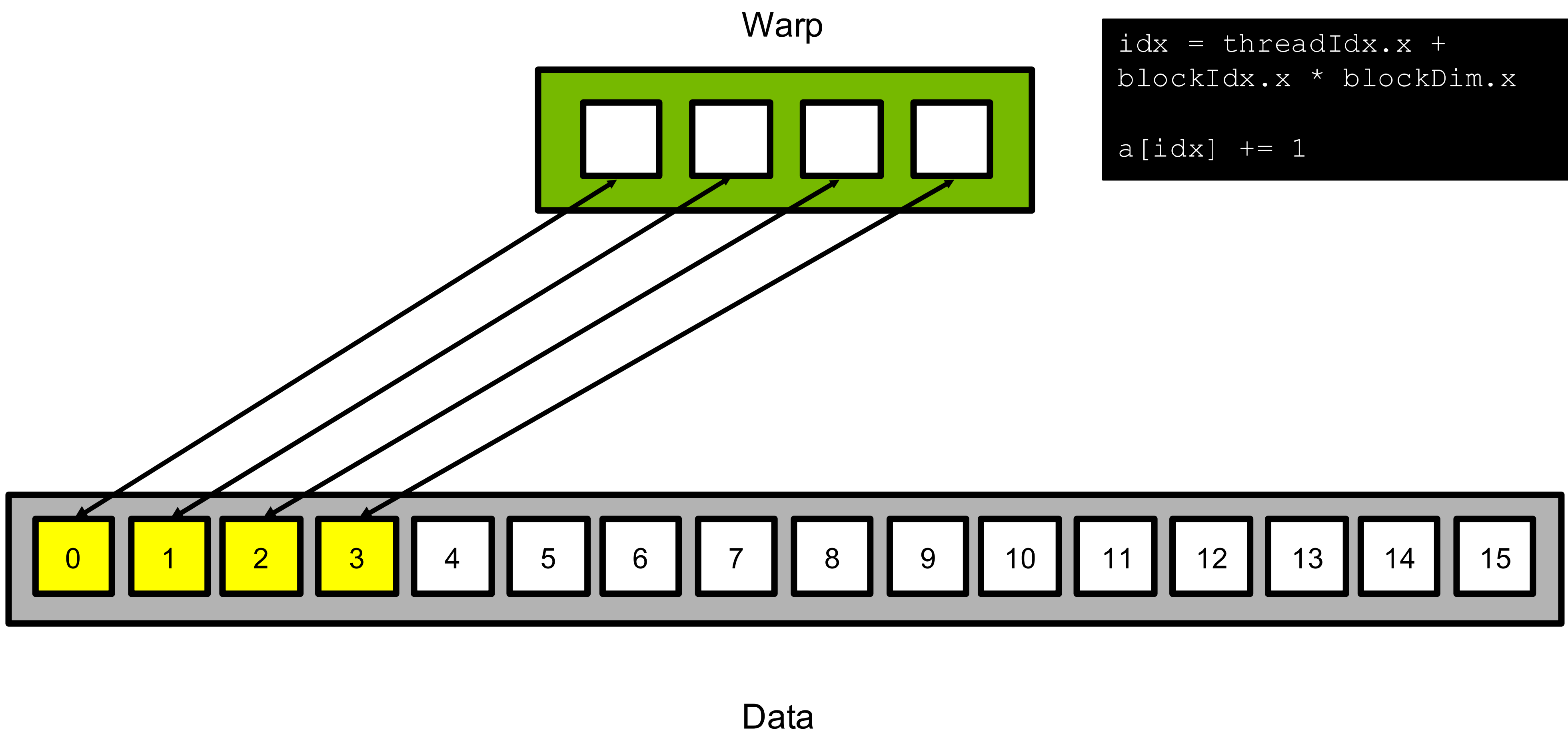
If the addresses requested are contiguous

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

a[idx] += 1
```



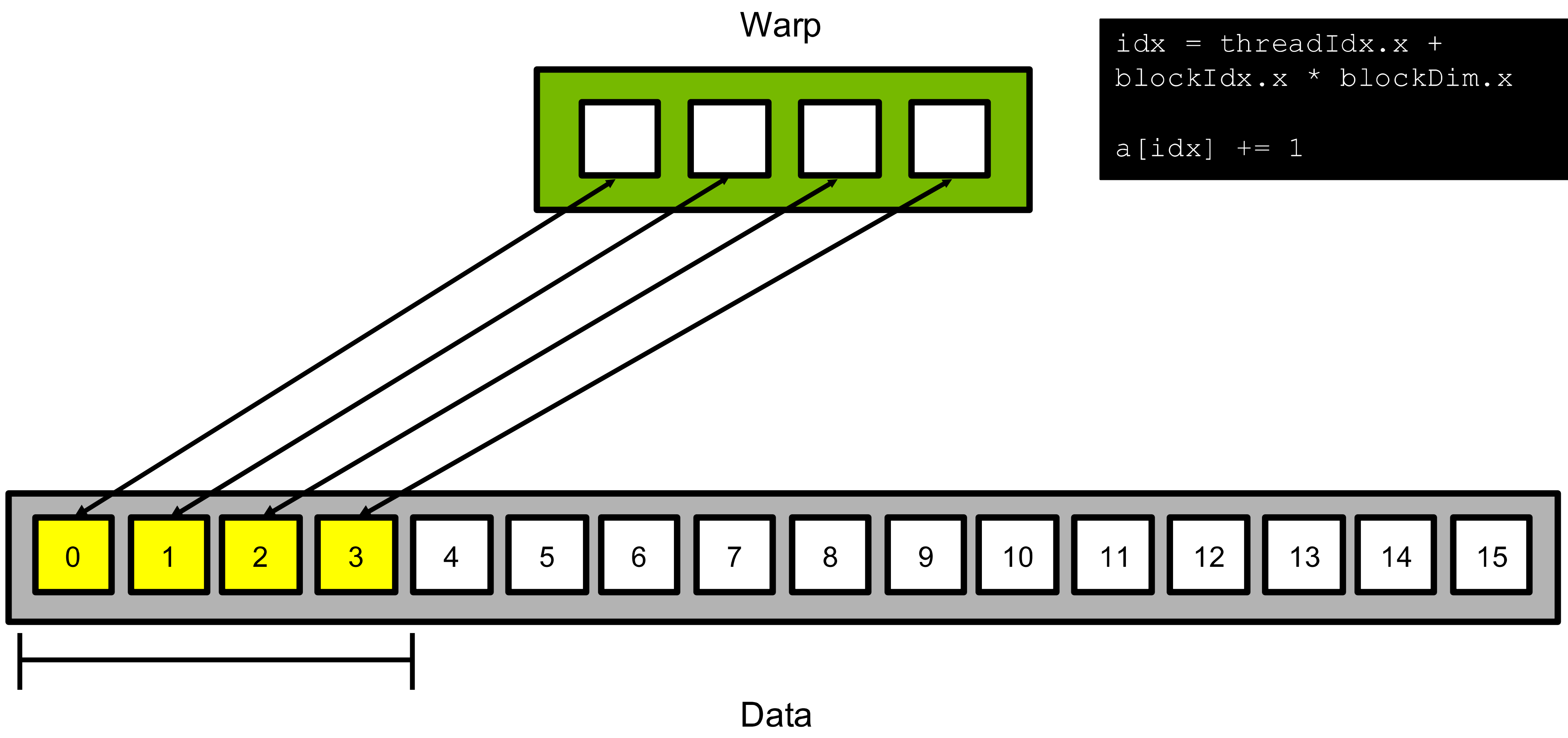
All data in the line will be used

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

a[idx] += 1
```



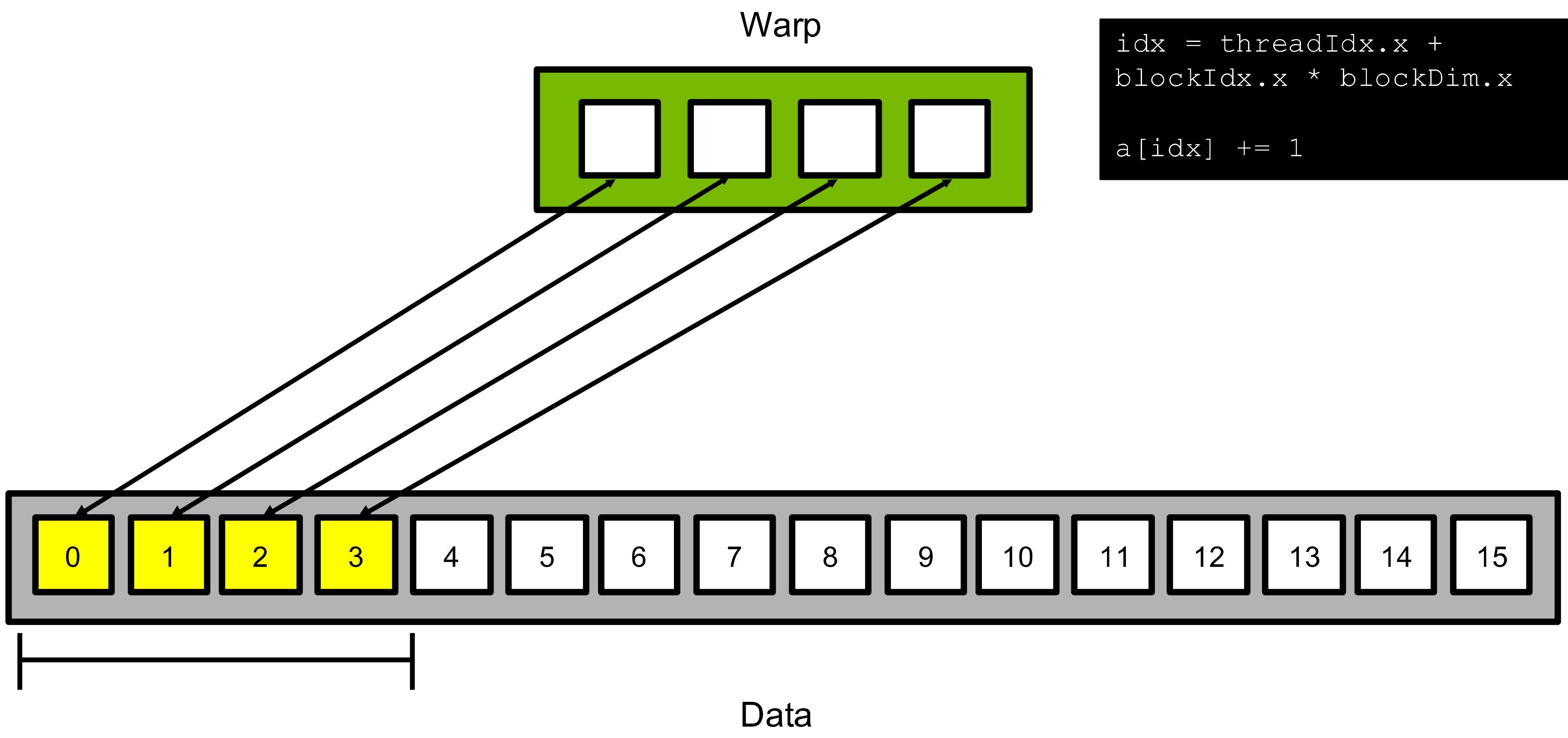
And the transfer will happen in as few lines as possible

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

a[idx] += 1
```



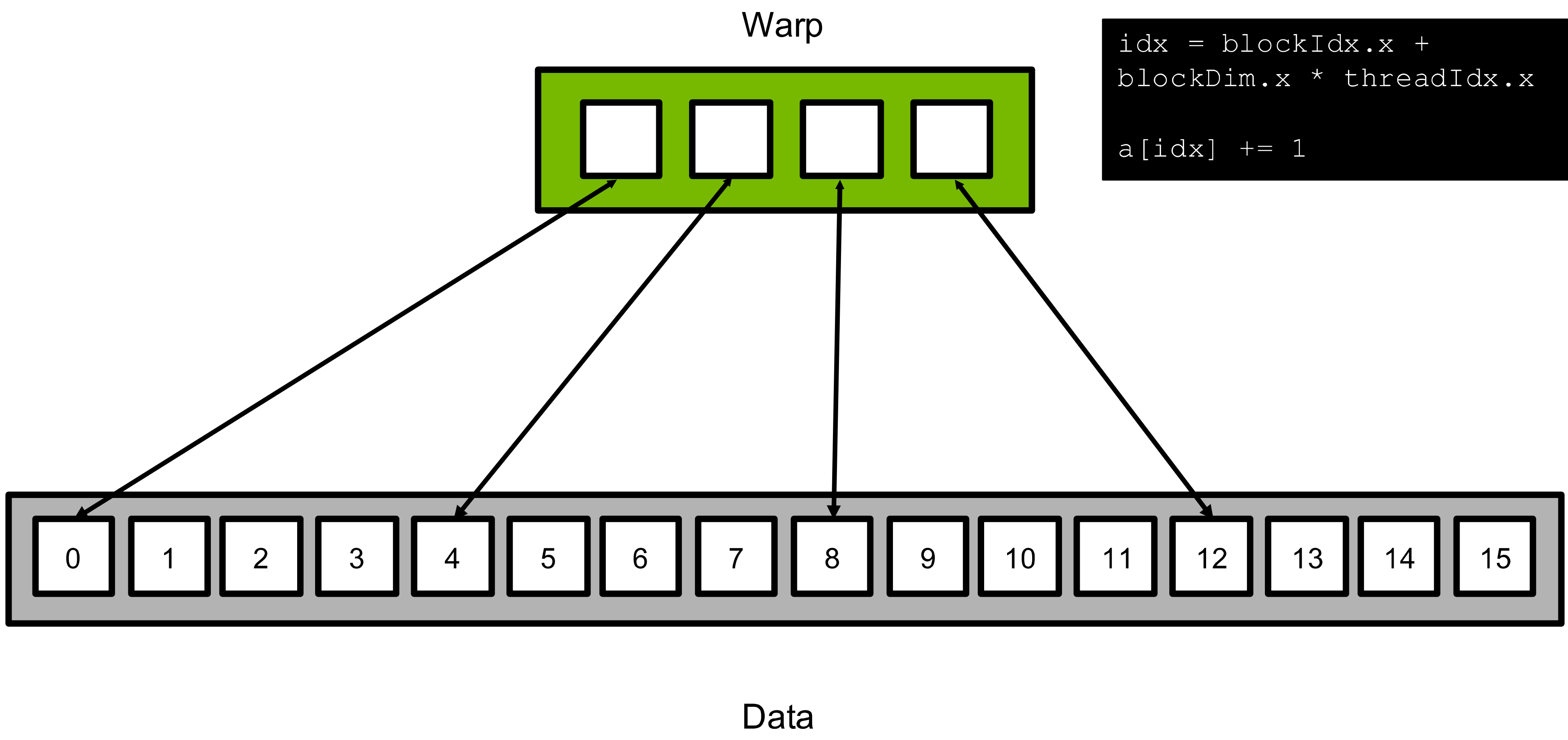
When this occurs, the memory access is fully **coalesced**

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

a[idx] += 1
```



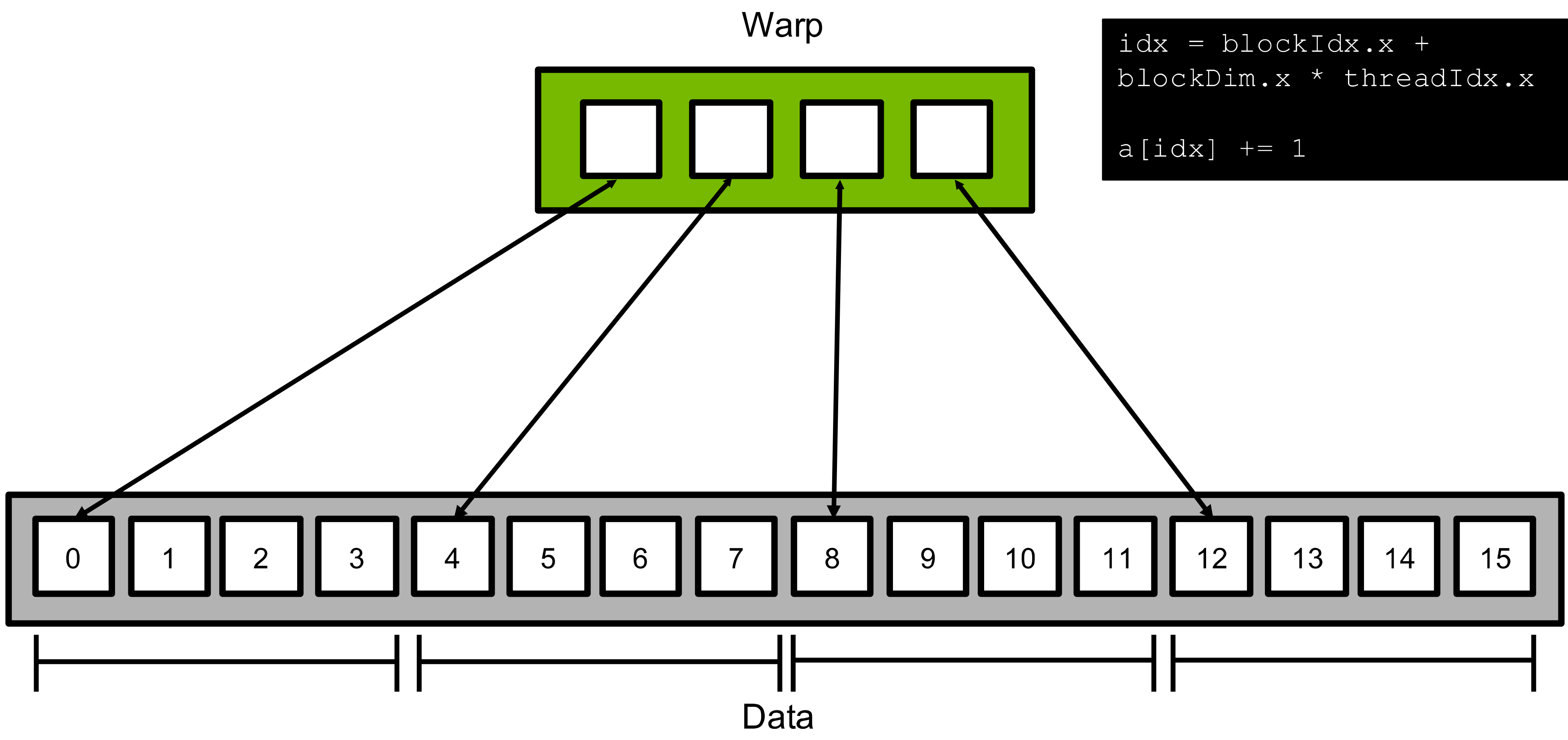
As requested memory becomes less contiguous

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

a[idx] += 1
```



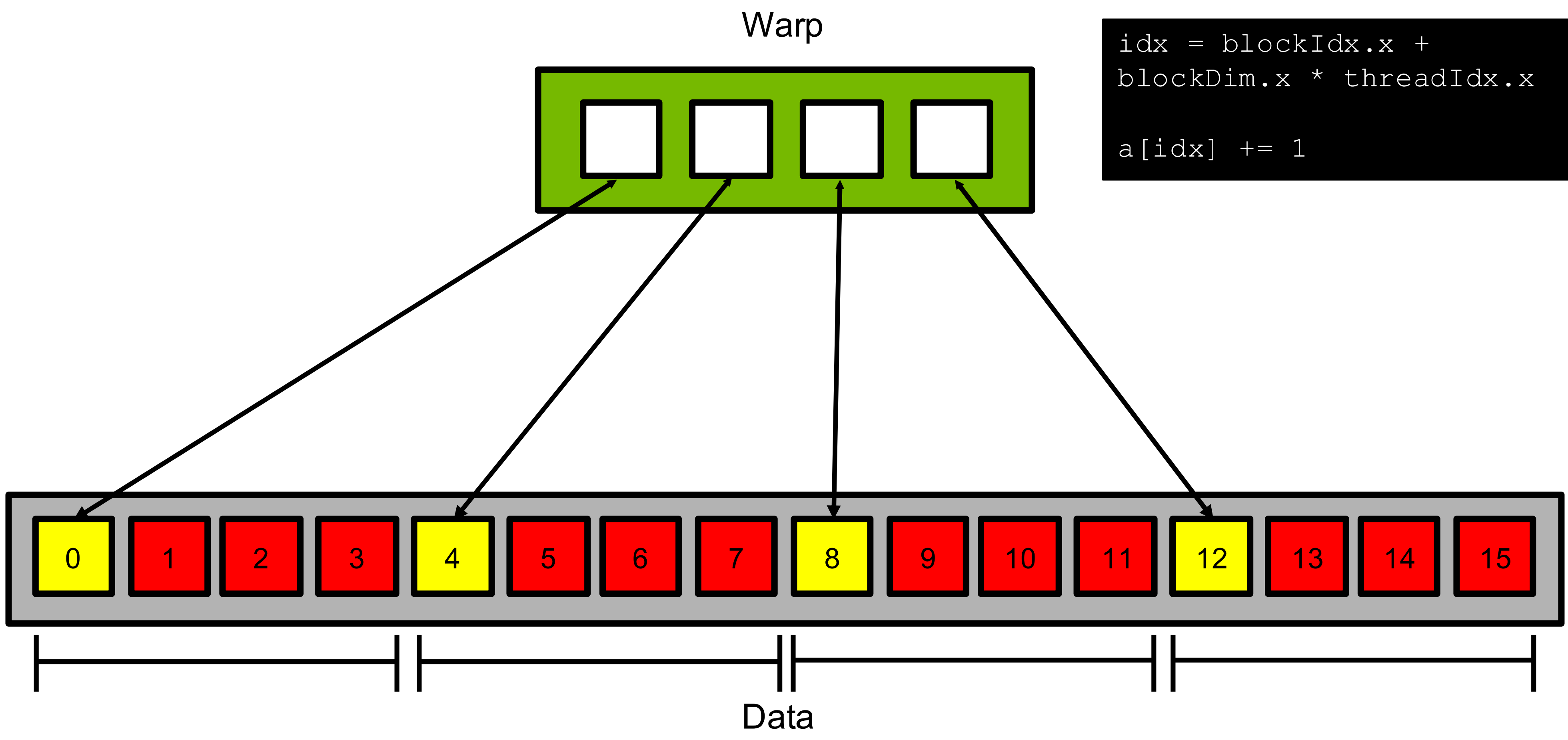
More lines will have to be transferred to fulfil the needs of the warp

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

a[idx] += 1
```



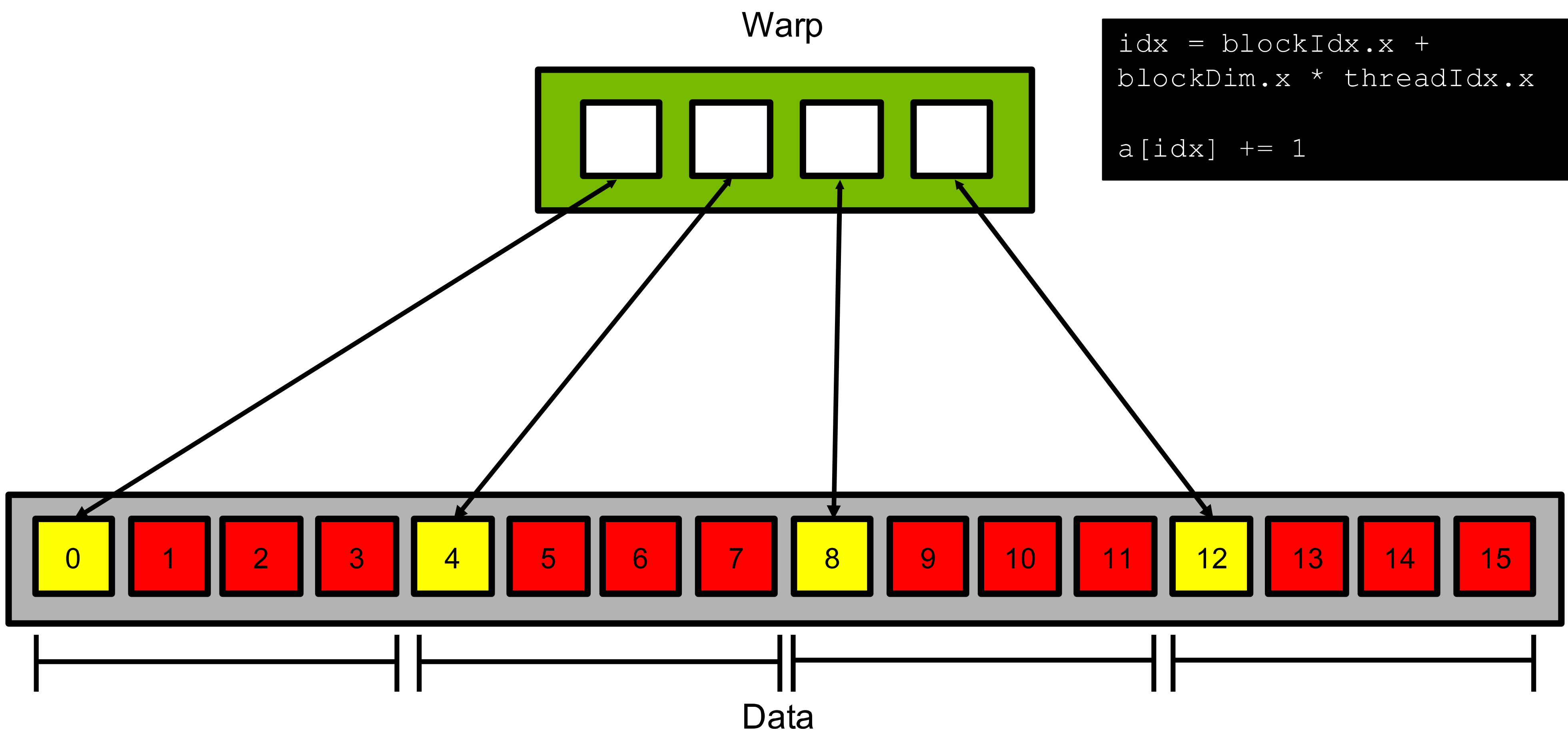
And more of the data being transferred will go unused

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

a[idx] += 1
```



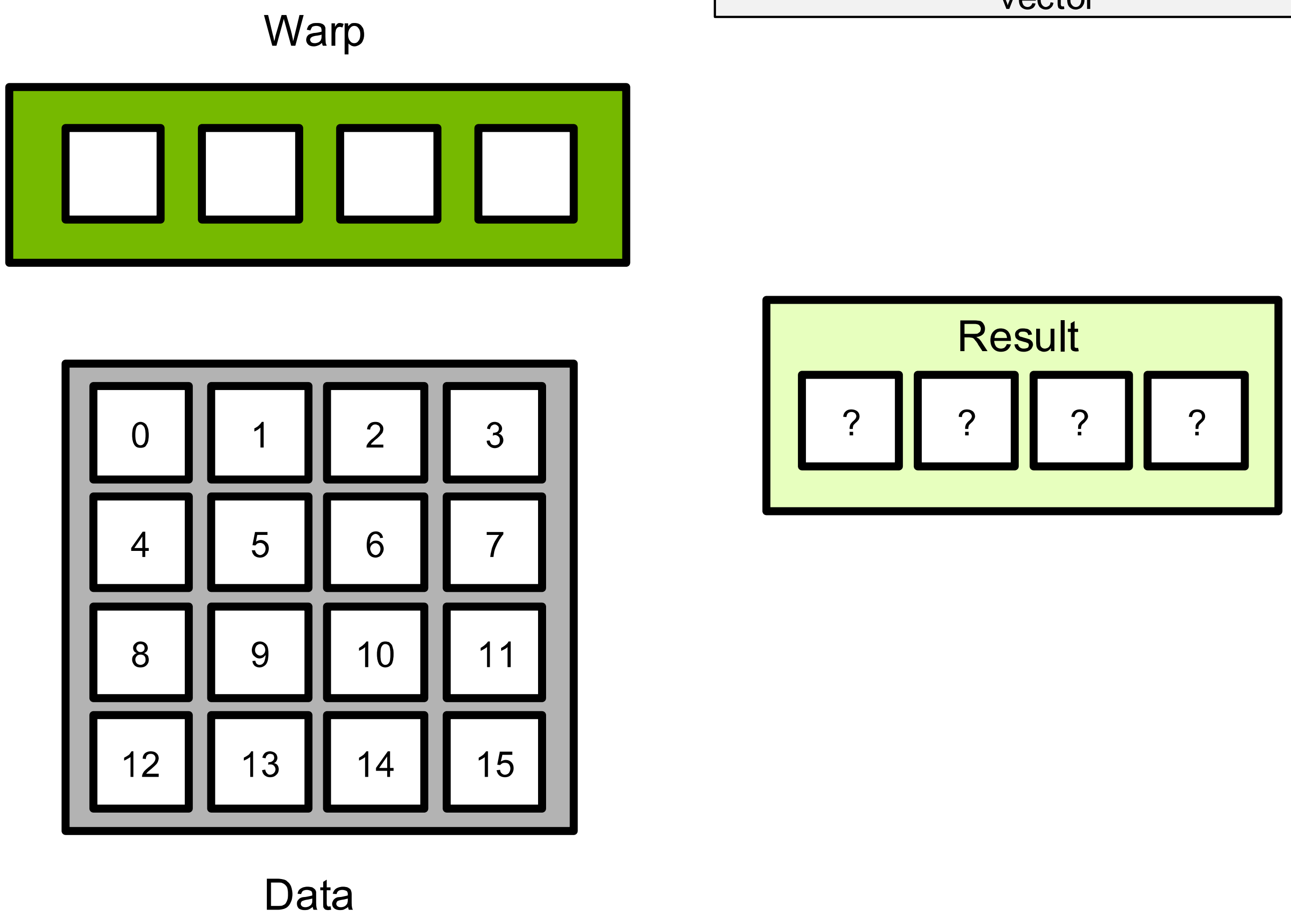
The memory throughput is degraded, and additional time is required: a performance loss

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

a[idx] += 1
```

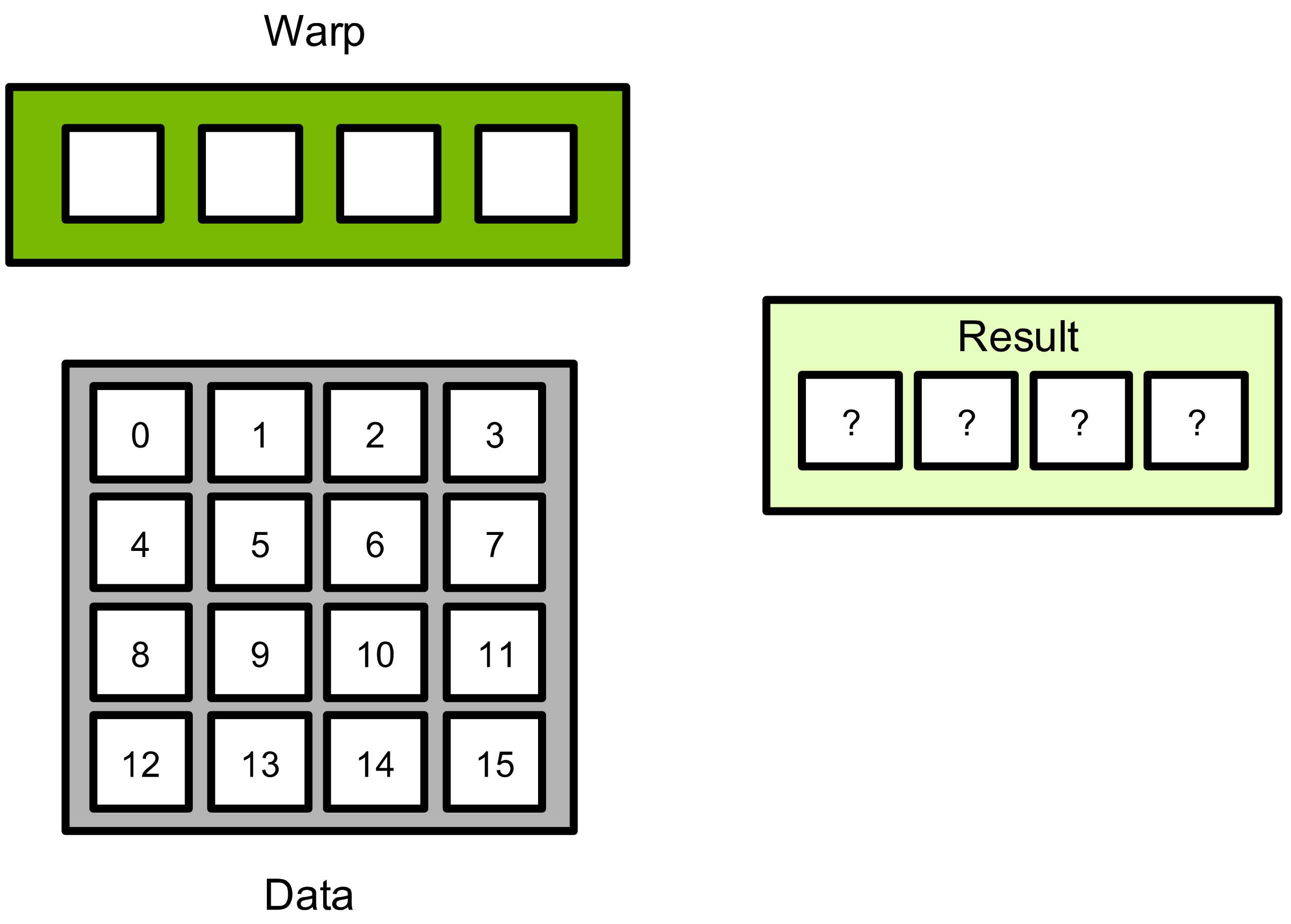


Row and Column Sum Comparison

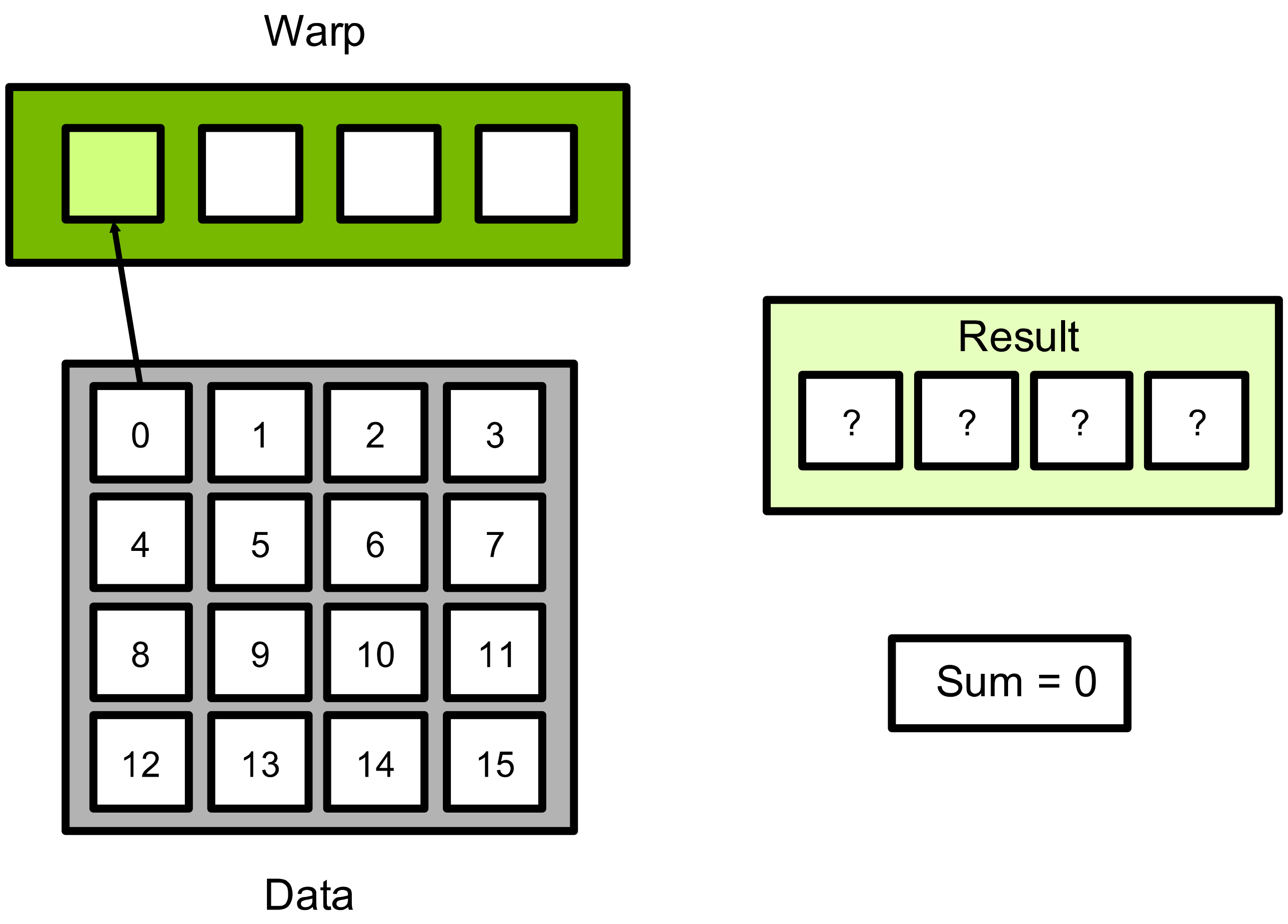
Consider a kernel that stores the sum of each row of a matrix (which here is 4 contiguous data elements) in a result vector



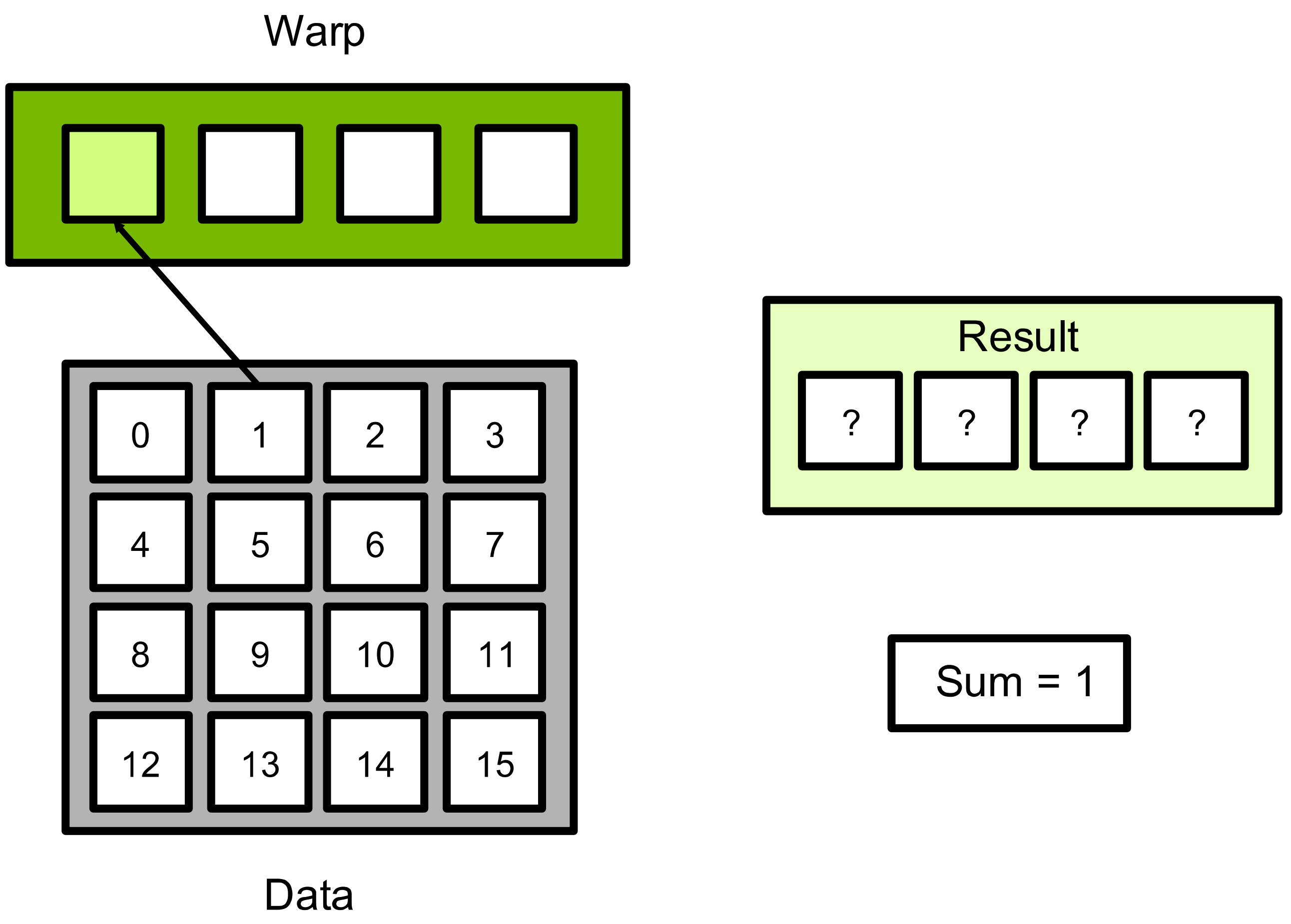
A single thread could iterate over a row, summing it, and then write the result in the solution vector



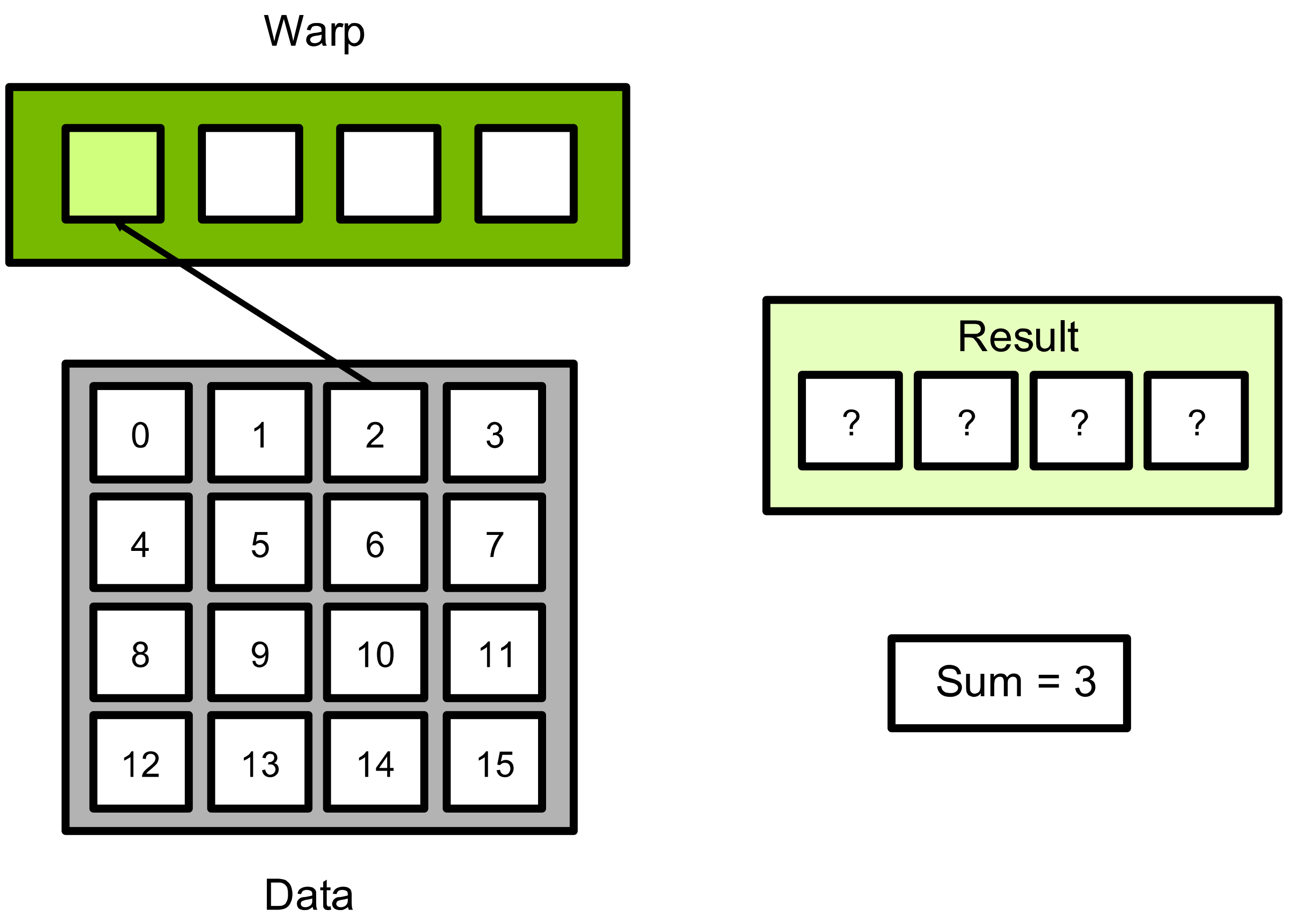
A single thread could iterate over a row, summing it, and then write the result in the solution vector



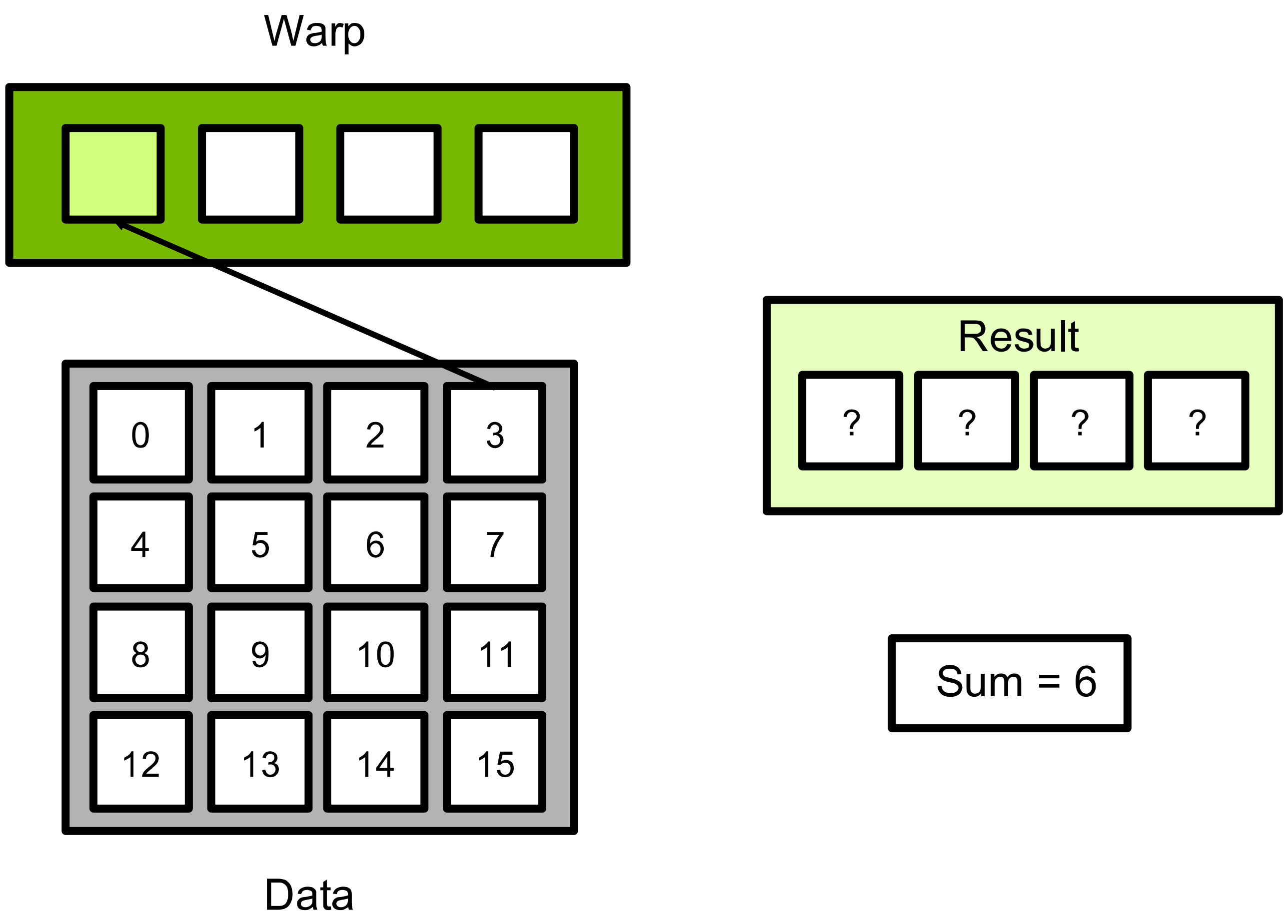
A single thread could iterate over a row, summing it, and then write the result in the solution vector



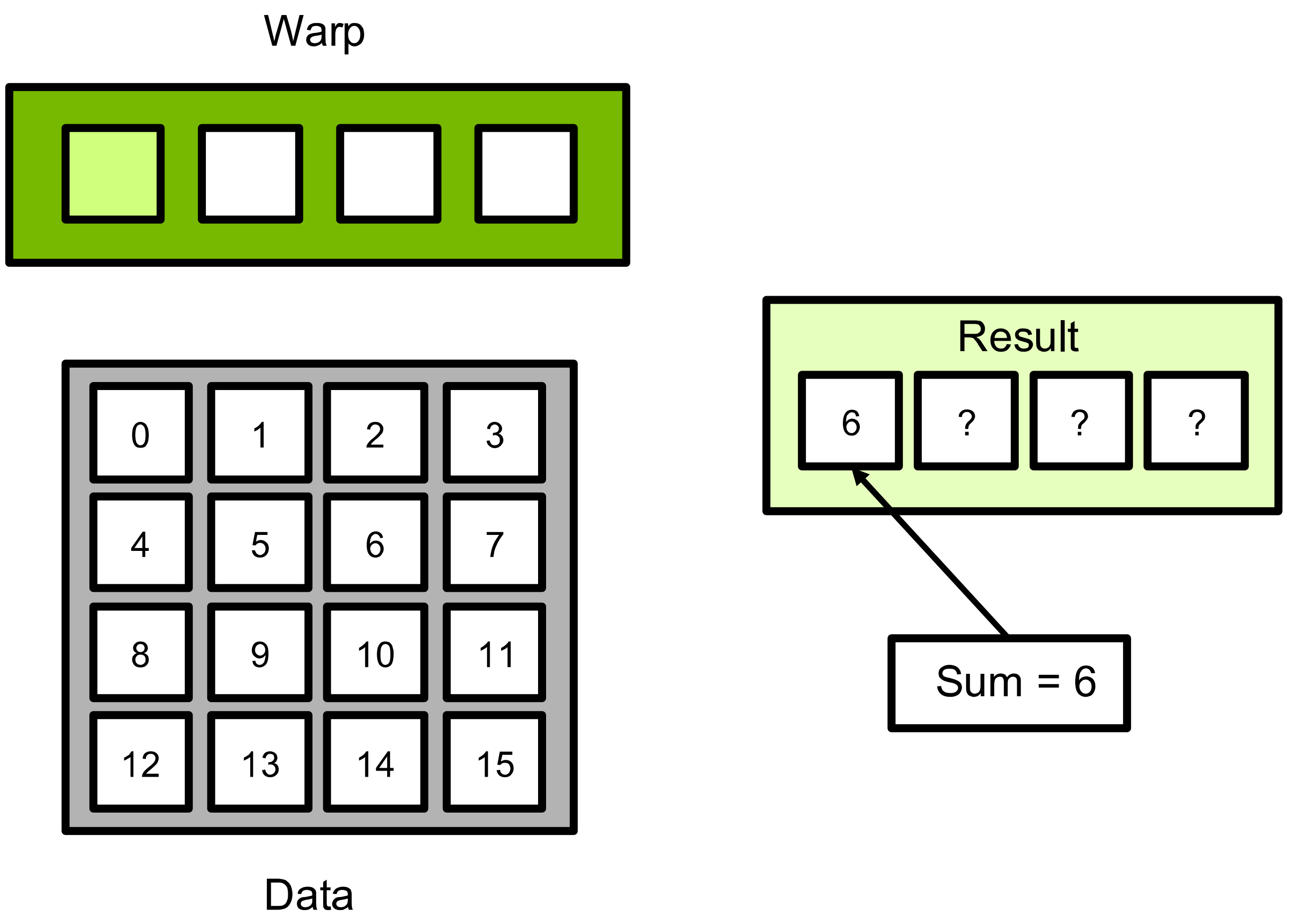
A single thread could iterate over a row, summing it, and then write the result in the solution vector



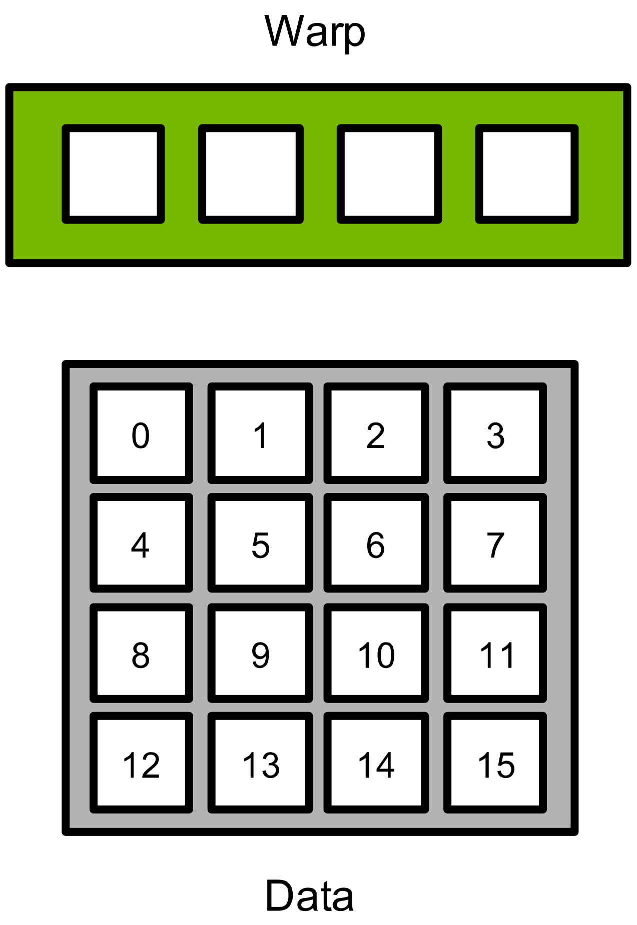
A single thread could iterate over a row, summing it, and then write the result in the solution vector



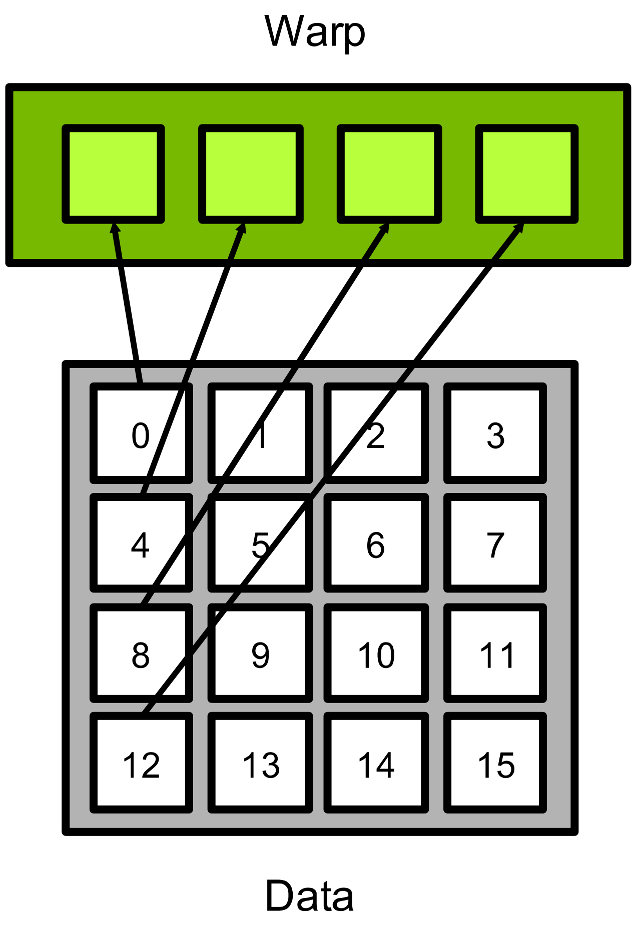
A single thread could iterate over a row, summing it, and then write the result in the solution vector

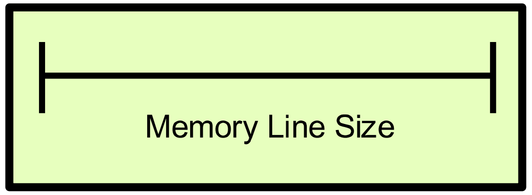


This seems natural, but look at what happens when we consider the parallel execution within the warp

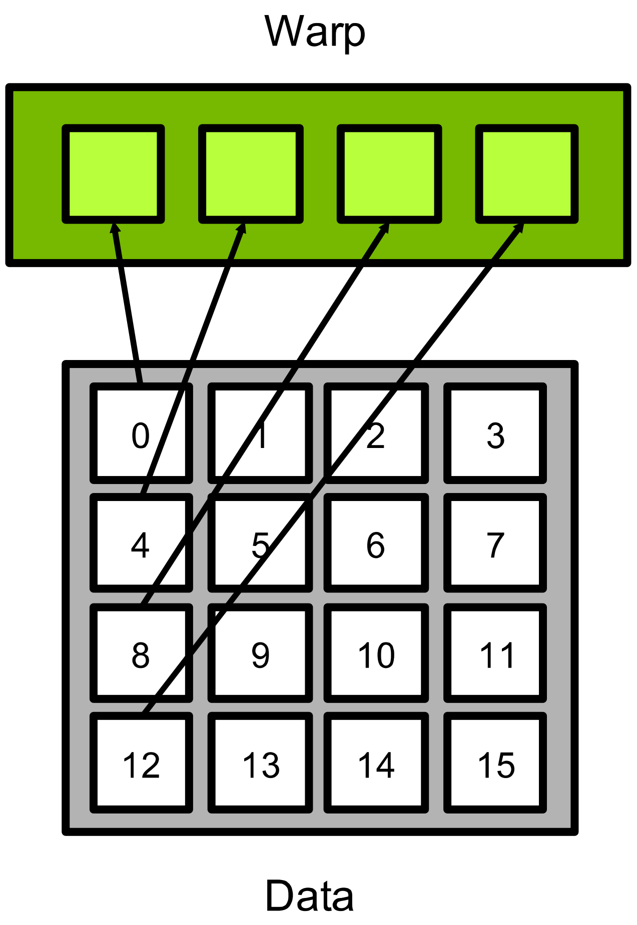


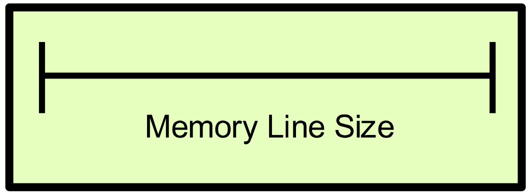
Each thread in the warp is requesting data in a different line of memory



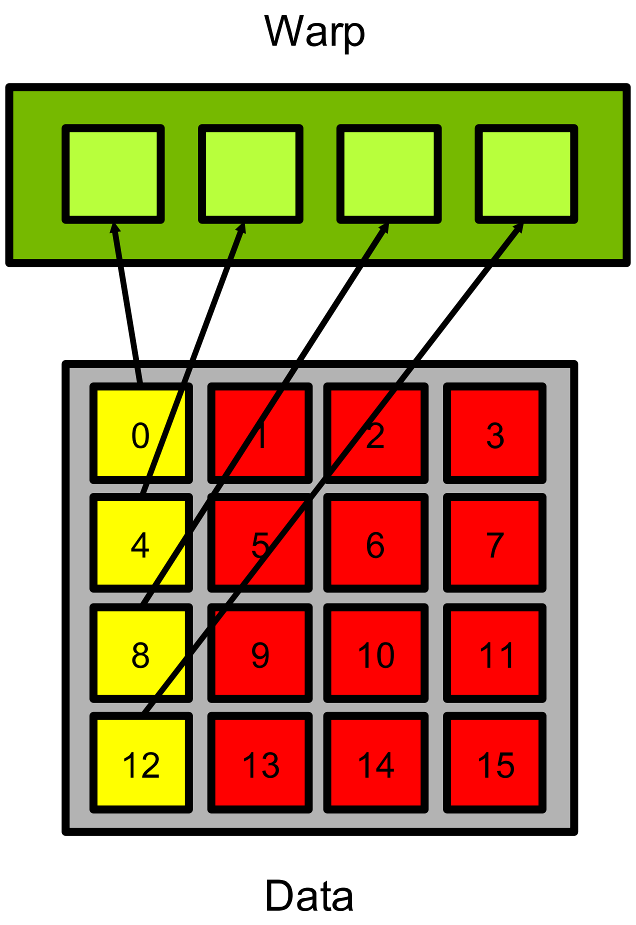


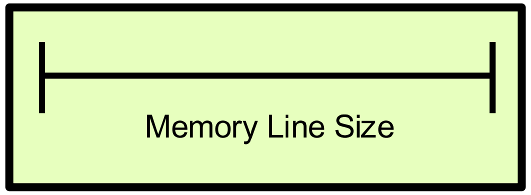
Note that increments to threadIdx.x are mapping to increments in the data along the y axis



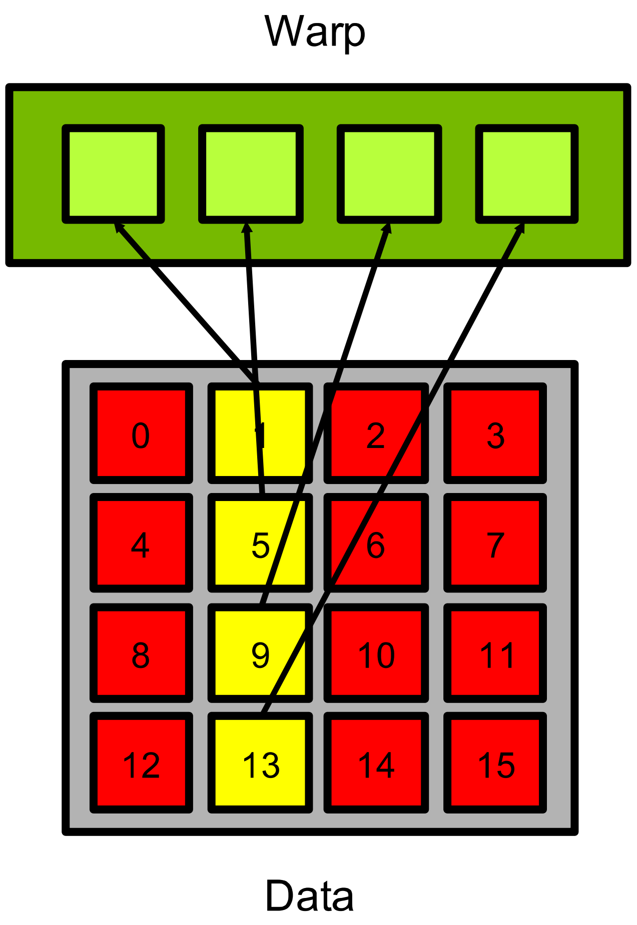


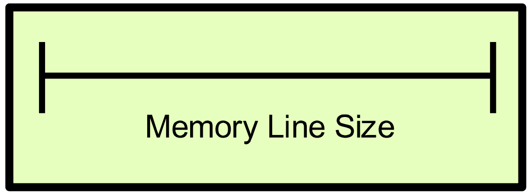
Which means (in our example) 4 lines of data will need to be loaded, and 75% of the data loaded will be unused



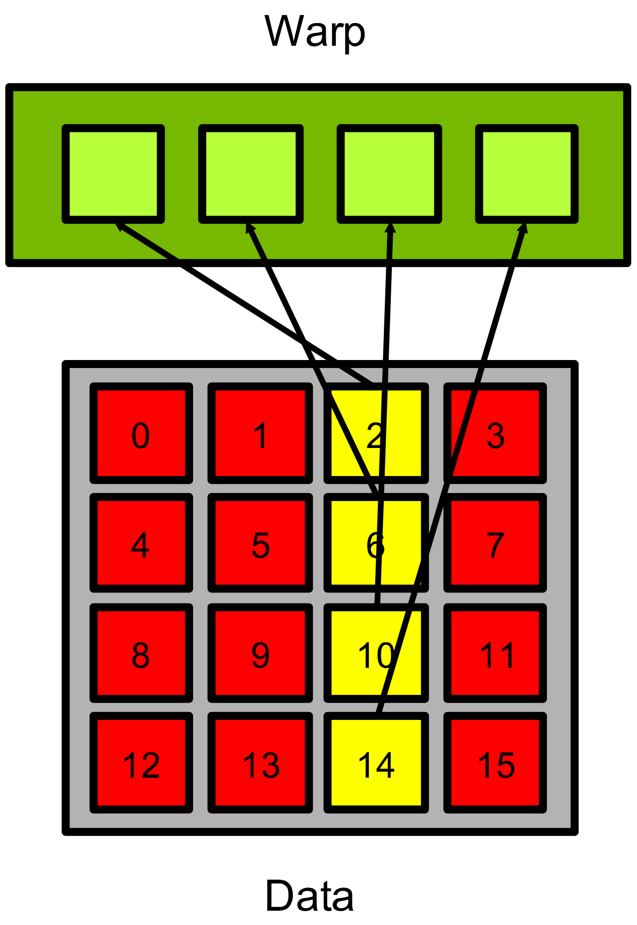


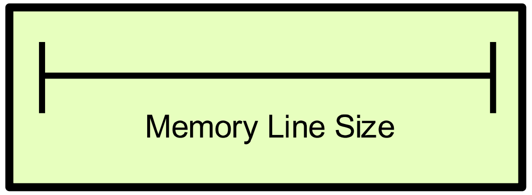
Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues



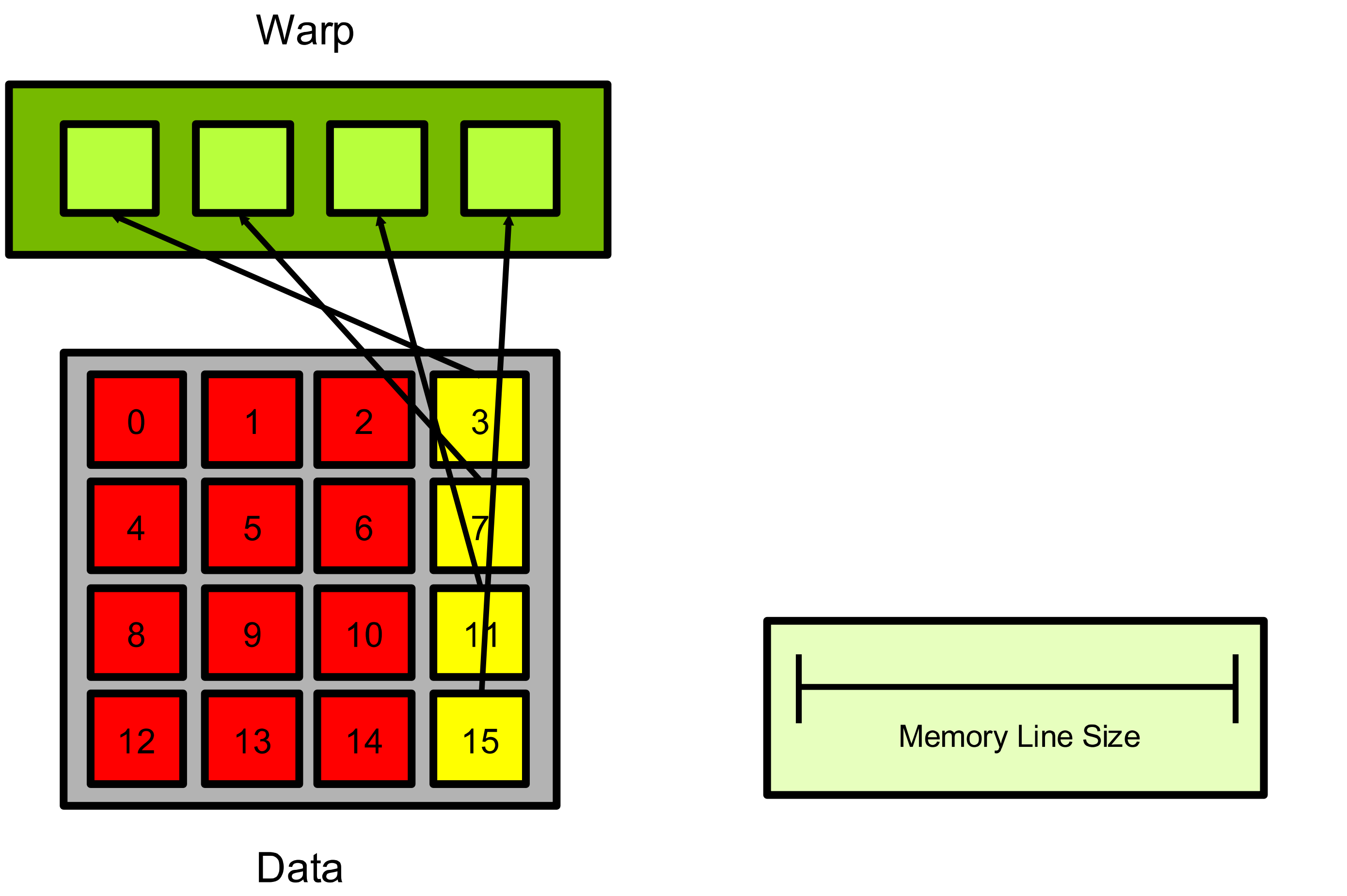


Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues

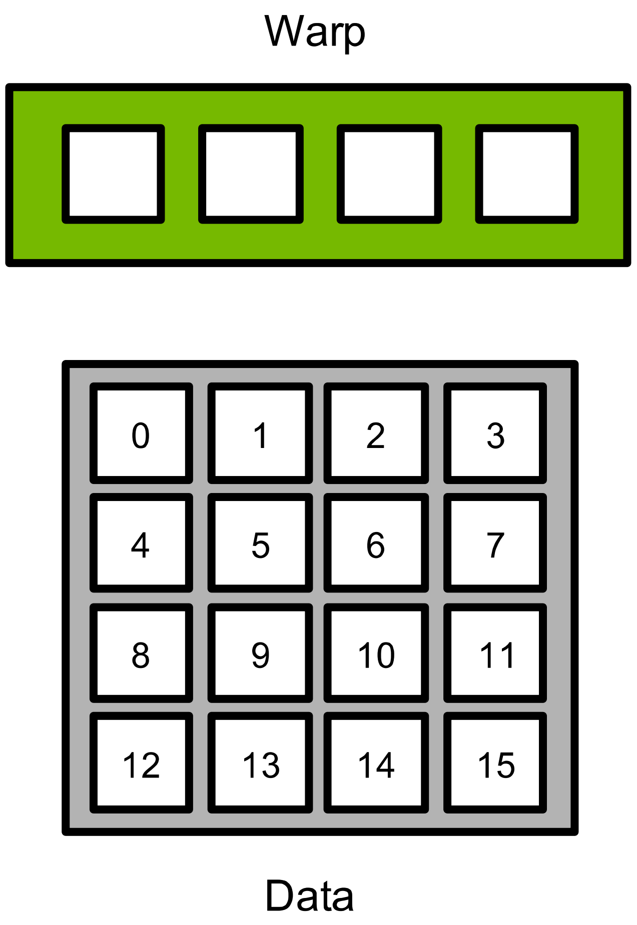




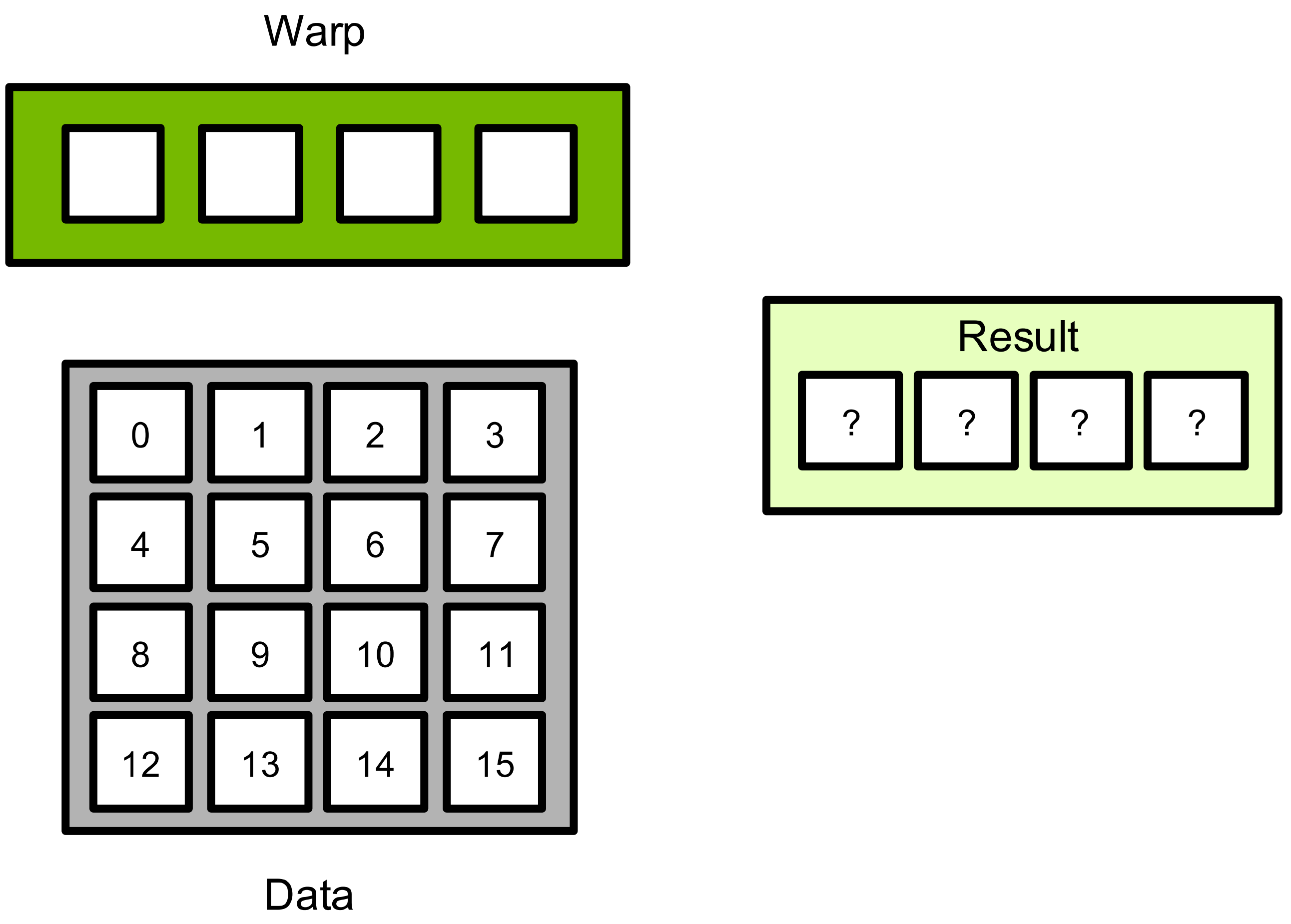
Unfortunately, as each thread iterates over its row, the same uncoalesced pattern continues



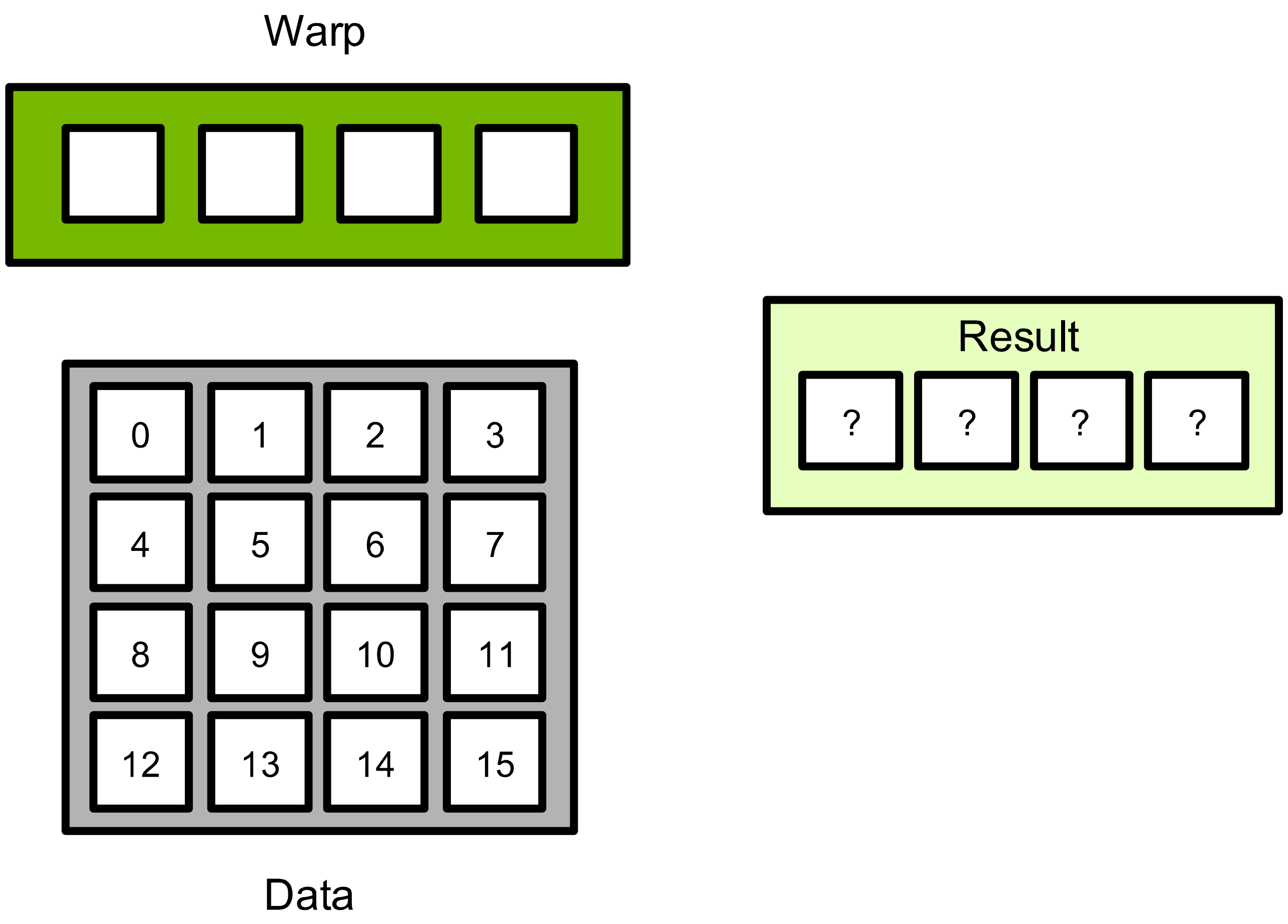
In this example we transferred 16 memory lines, and used 25% of the data for each line transferred



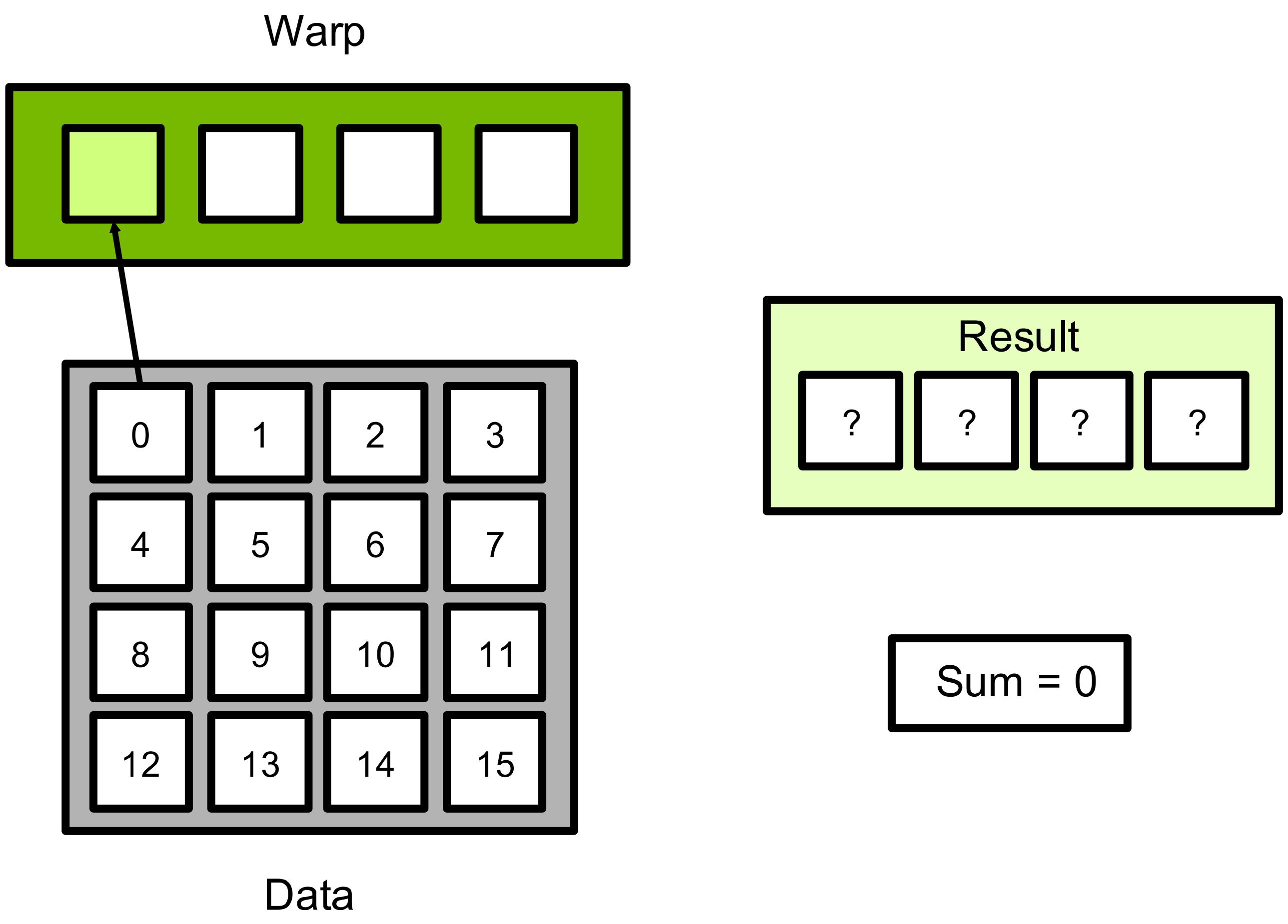
Let’s compare a kernel that stores the sum of each **column** of a matrix in a result vector



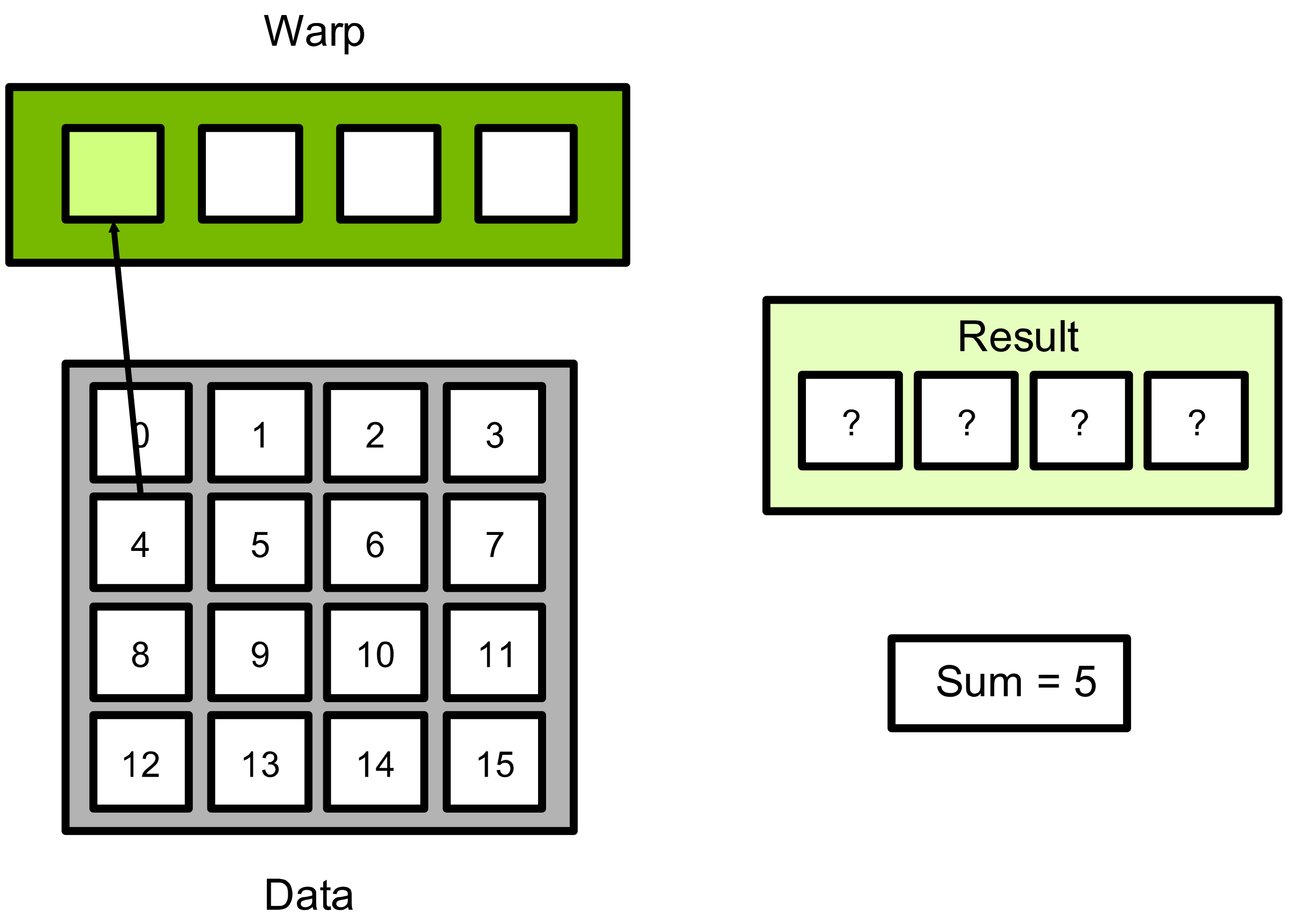
A single thread could iterate over a column, summing it, and then write the result in the solution vector



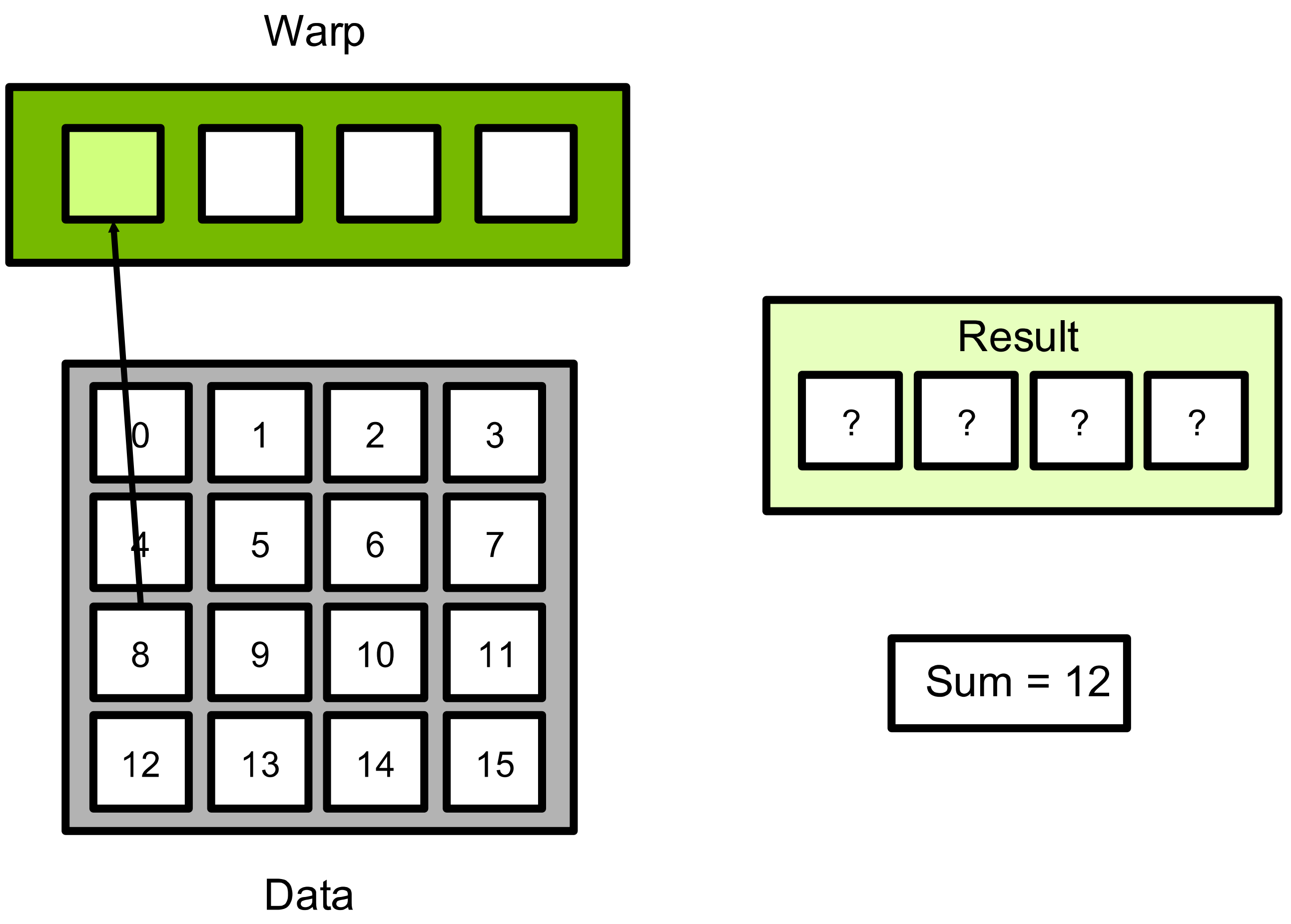
A single thread could iterate over a column, summing it, and then write the result in the solution vector



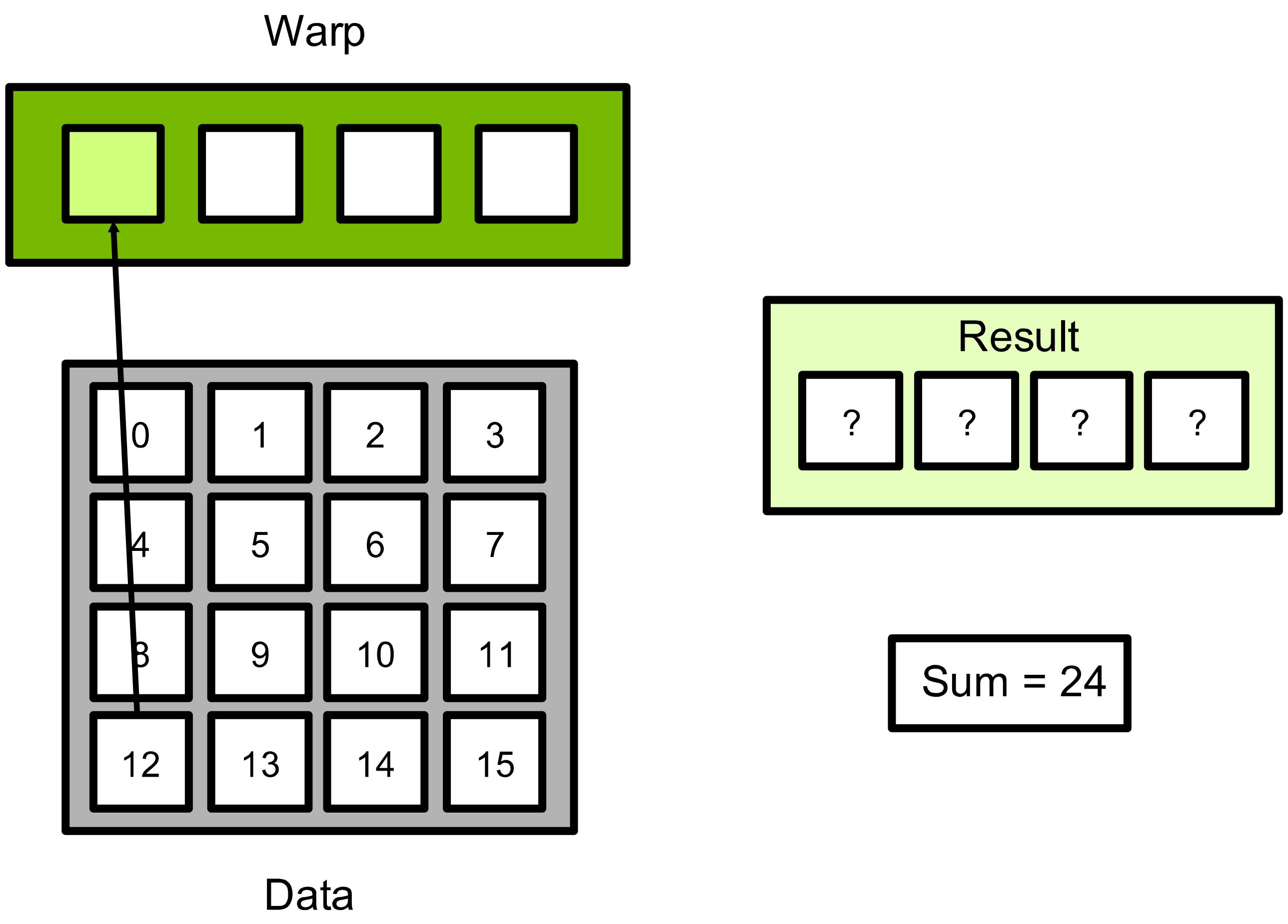
A single thread could iterate over a column, summing it, and then write the result in the solution vector



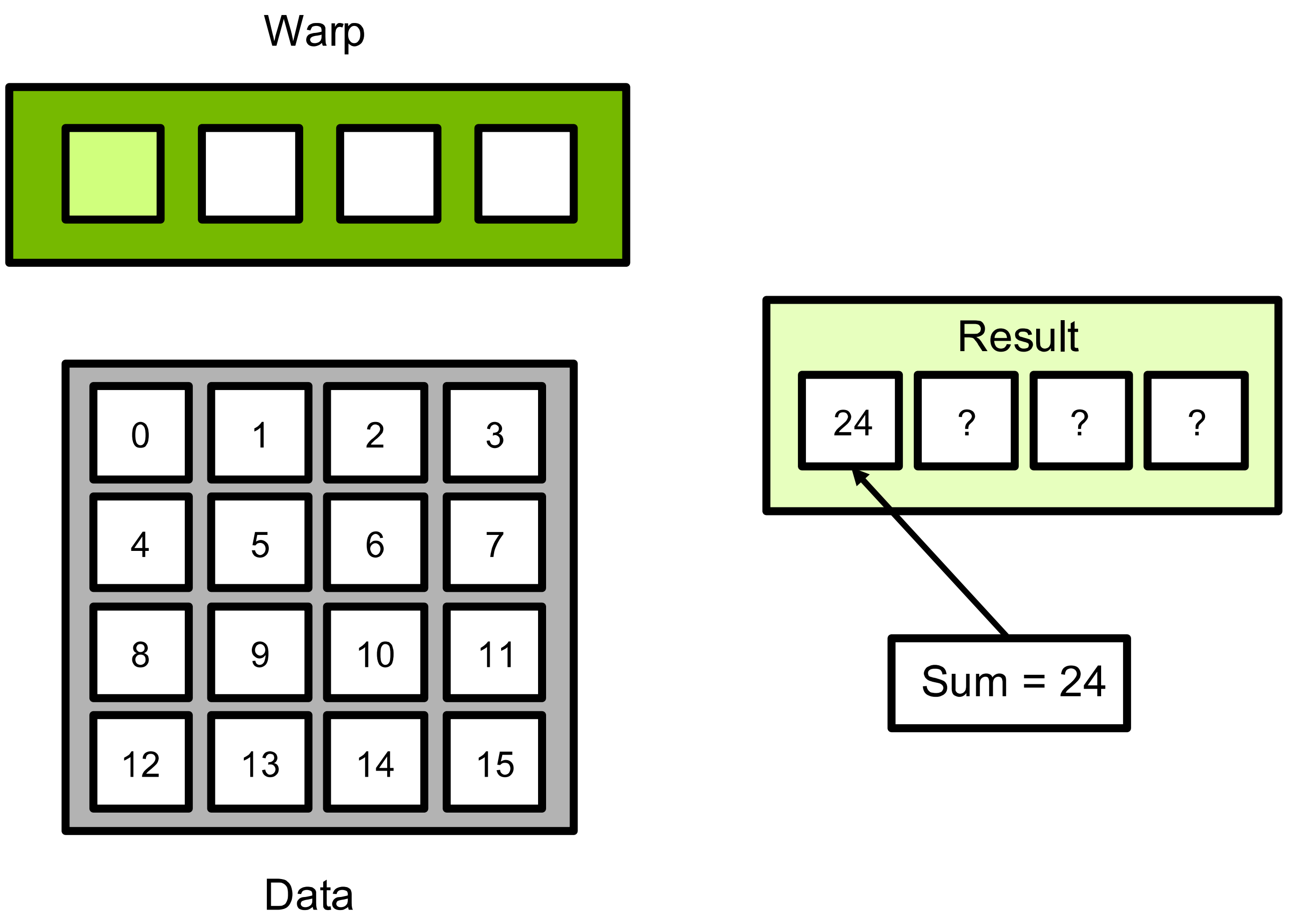
A single thread could iterate over a column, summing it, and then write the result in the solution vector



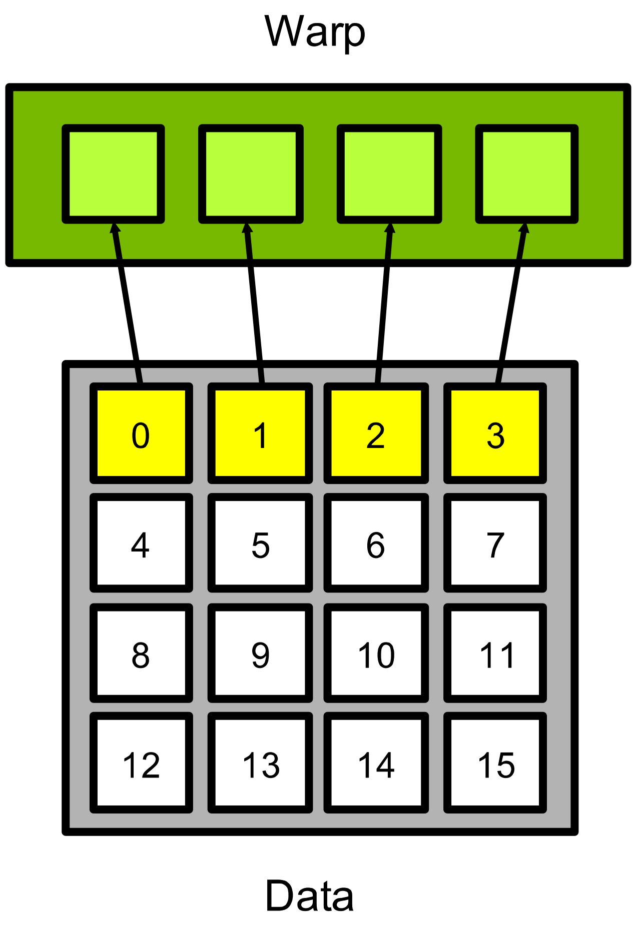
A single thread could iterate over a column, summing it, and then write the result in the solution vector

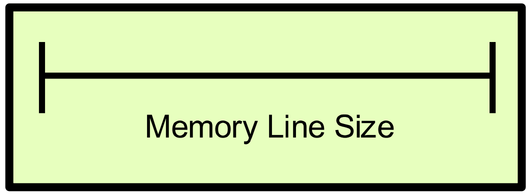


A single thread could iterate over a column, summing it, and then write the result in the solution vector

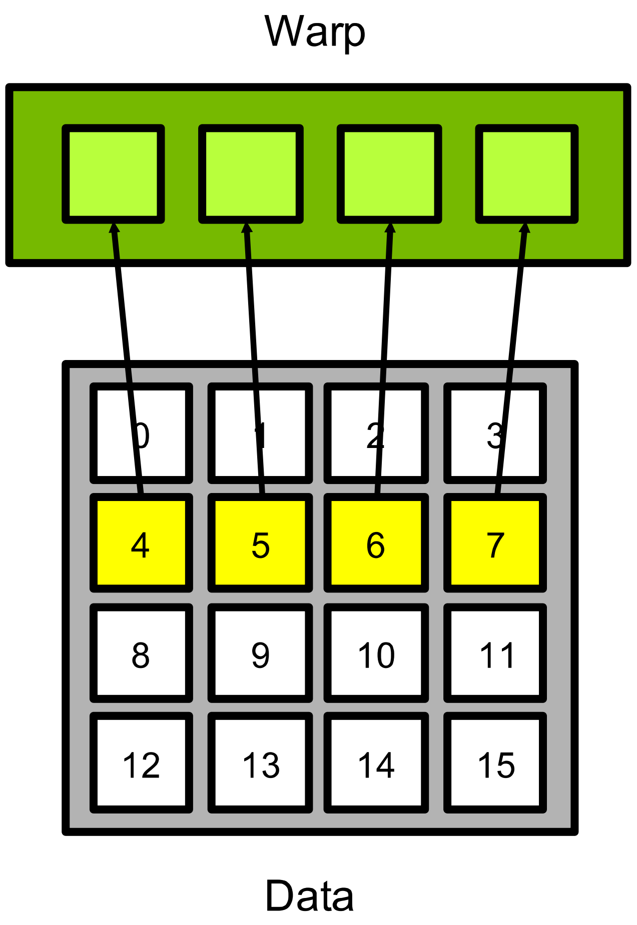


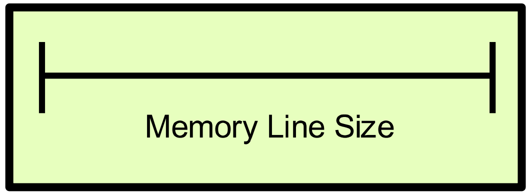
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



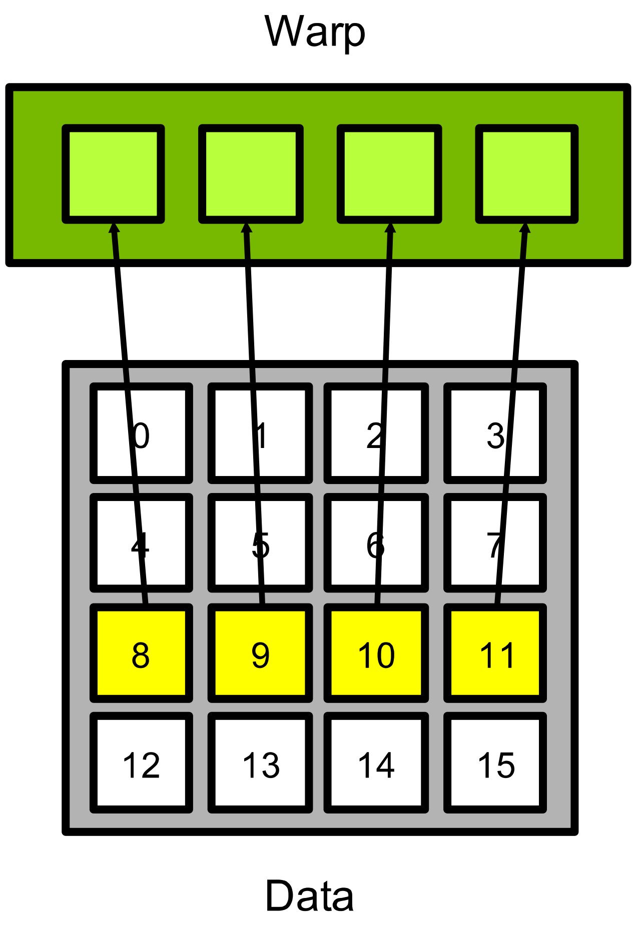


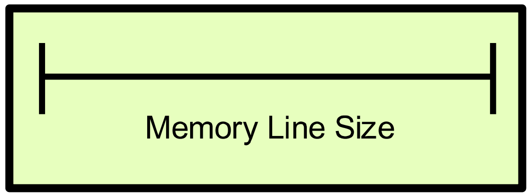
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



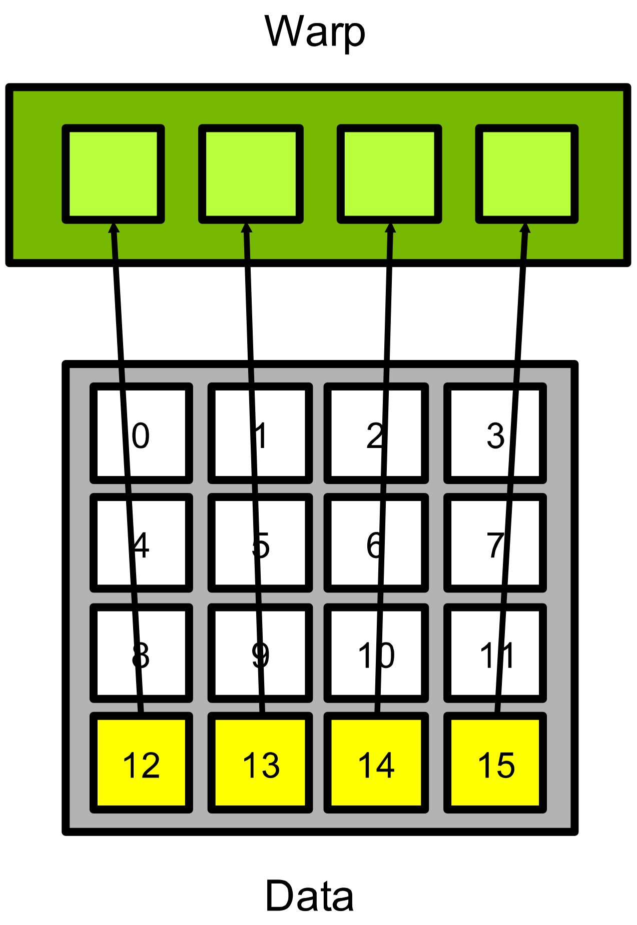


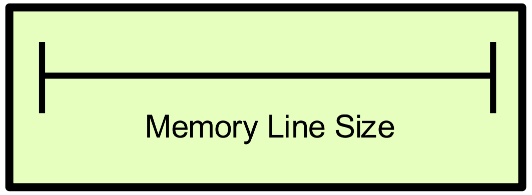
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



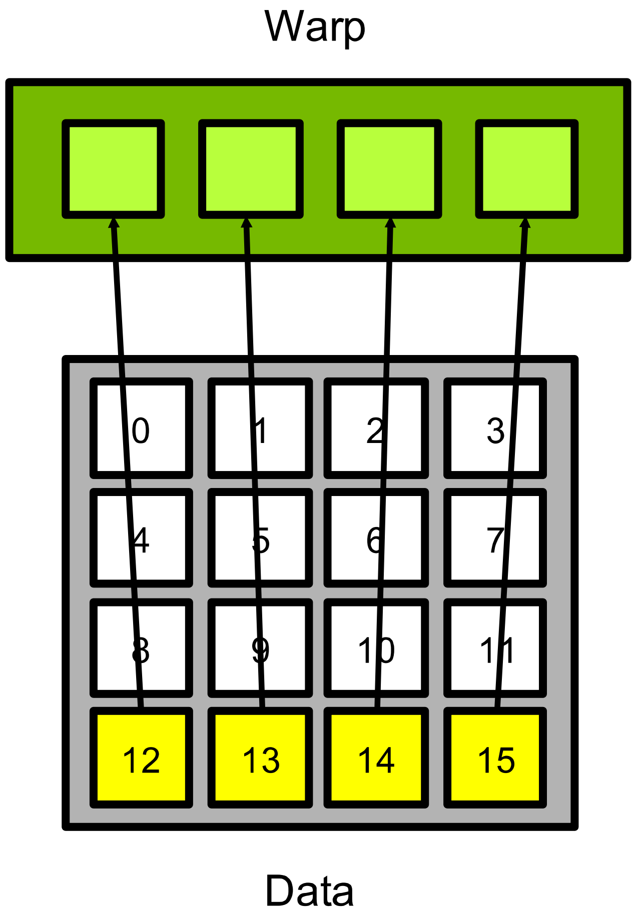


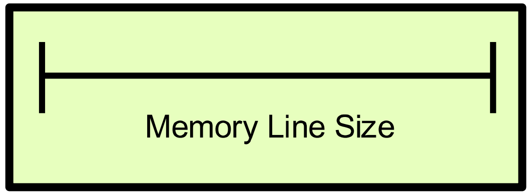
Here when we consider the parallel execution, we see that the warp's memory access is coalesced



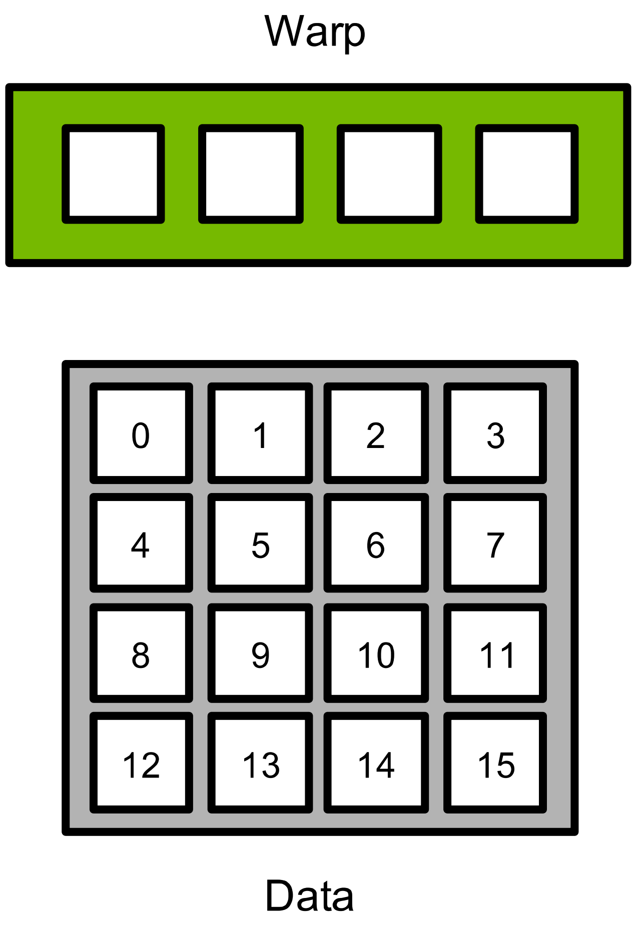


A useful tip to keep in mind is that increments to threadIdx.x should map to increments in data in the direction of fastest changing index – in this case the x axis





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



NVIDIA

DEEP
LEARNING
INSTITUTE

www.nvidia.com/dli