GPU Teaching Kit

Accelerated Computing

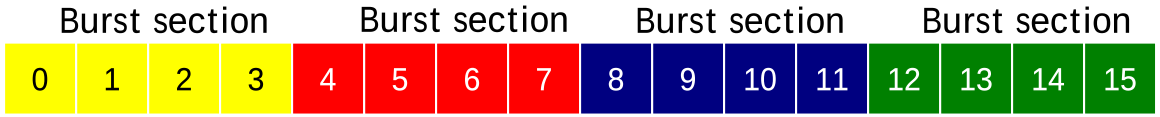
Lecture 6.2 – Performance Considerations

Memory Coalescing in CUDA

Objective

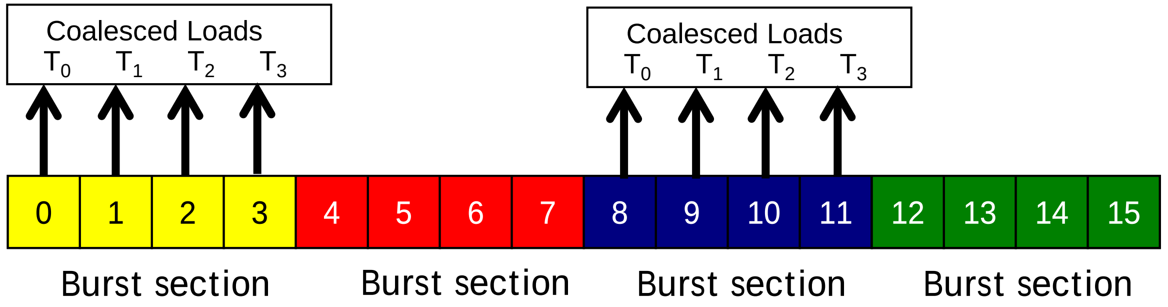
- To learn that memory coalescing is important for effectively utilizing memory bandwidth in CUDA
  - Its origin in DRAM burst
  - Checking if a CUDA memory access is coalesced
  - Techniques for improving memory coalescing in CUDA code

DRAM Burst – A System View



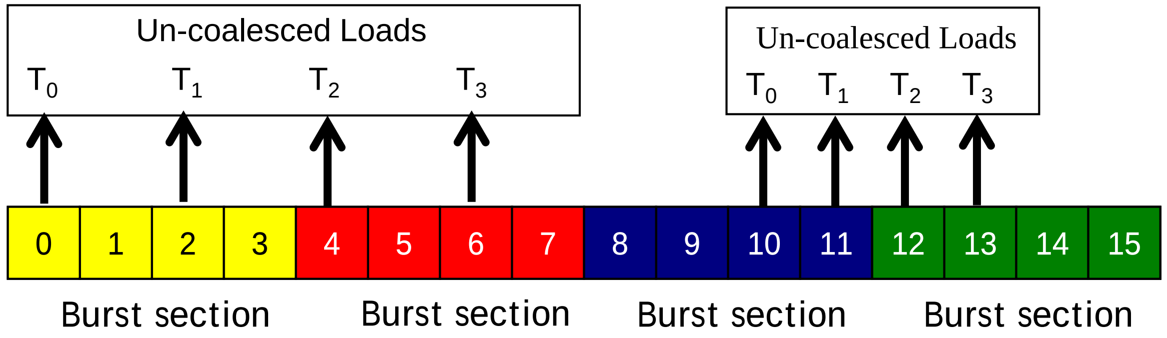
- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Memory Coalescing



– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Un-coalesced Accesses

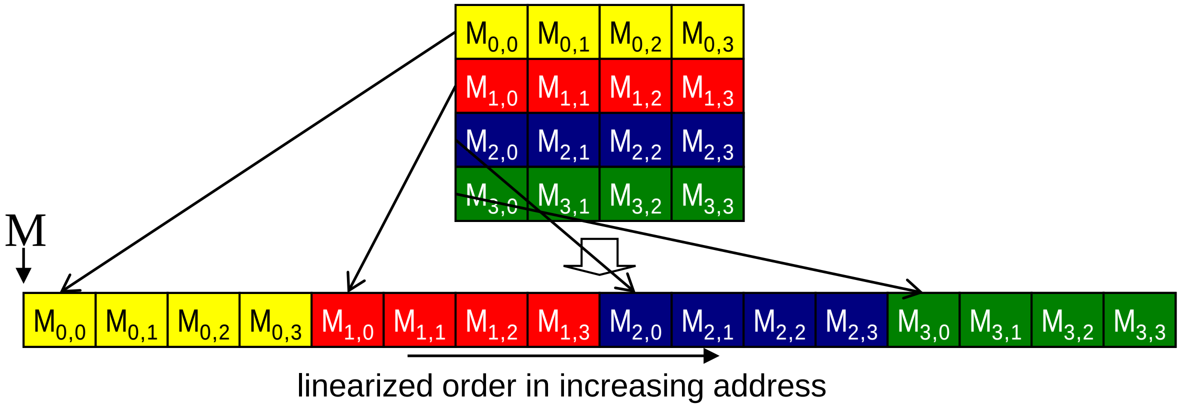


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

How to judge if an access is coalesced?

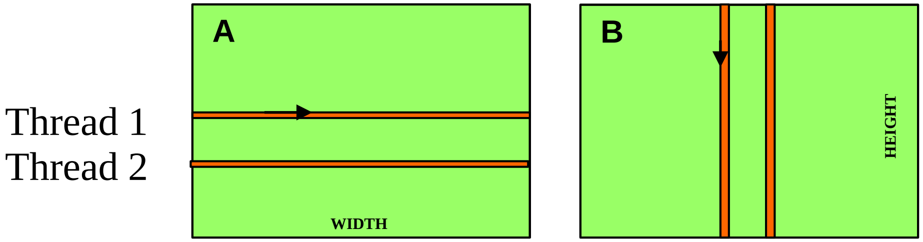
- Accesses in a warp are to consecutive locations if the index in an array access is in the form of
  - A[(expression with terms independent of threadIdx.x) + threadIdx.x];

A 2D C Array in Linear Memory Space



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
linearized order in increasing address

Two Access Patterns of Basic Matrix Multiplication



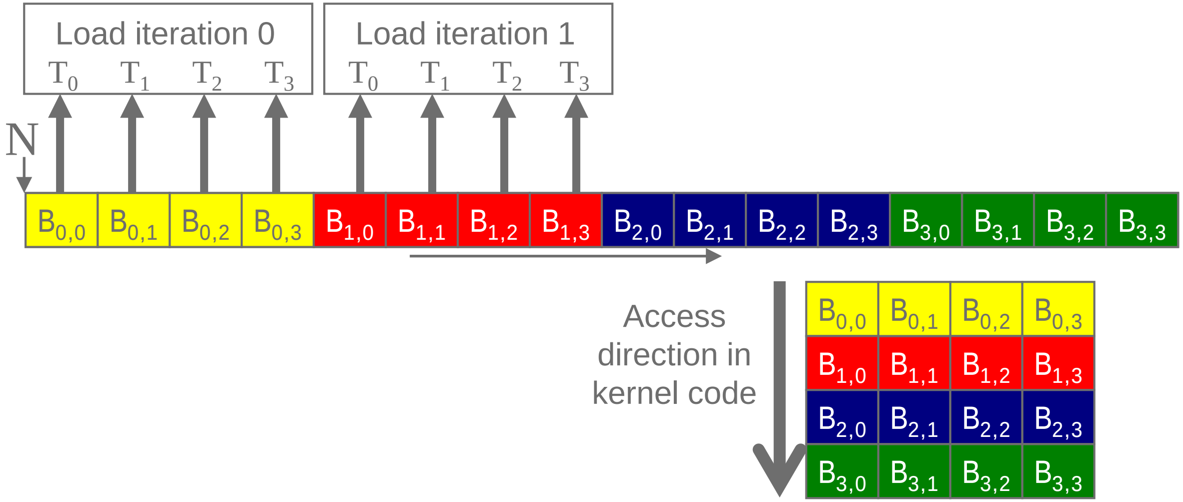
`A[Row*n+i]` `B[i*k+Col]`

i is the loop counter in the inner product loop of the kernel code

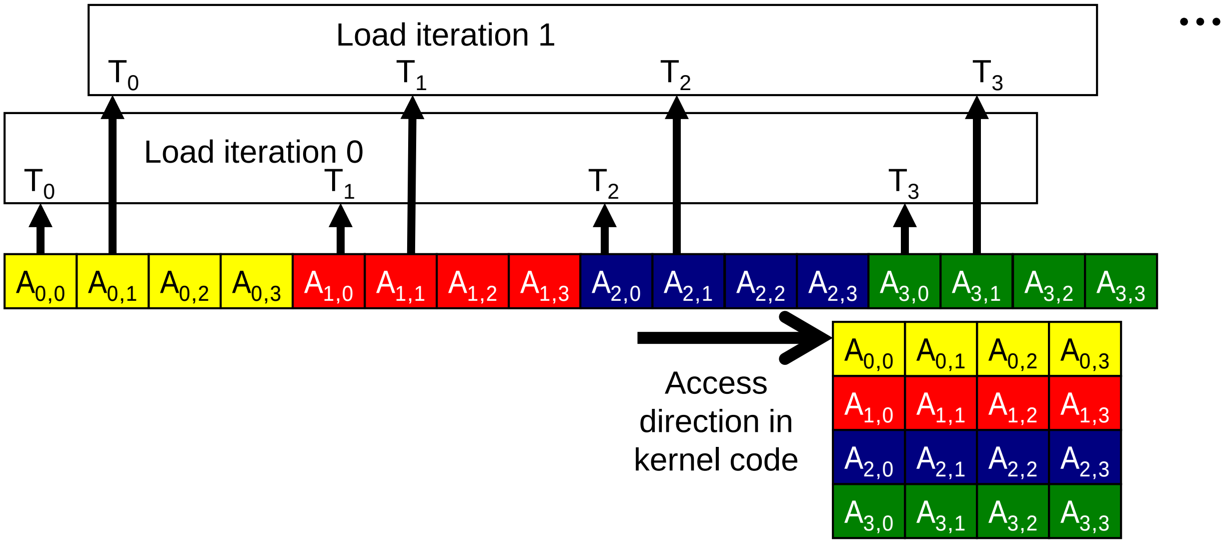
A is m × n, B is n × k

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

B accesses are coalesced


Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
N
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
Access direction in kernel code
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

A Accesses are Not Coalesced

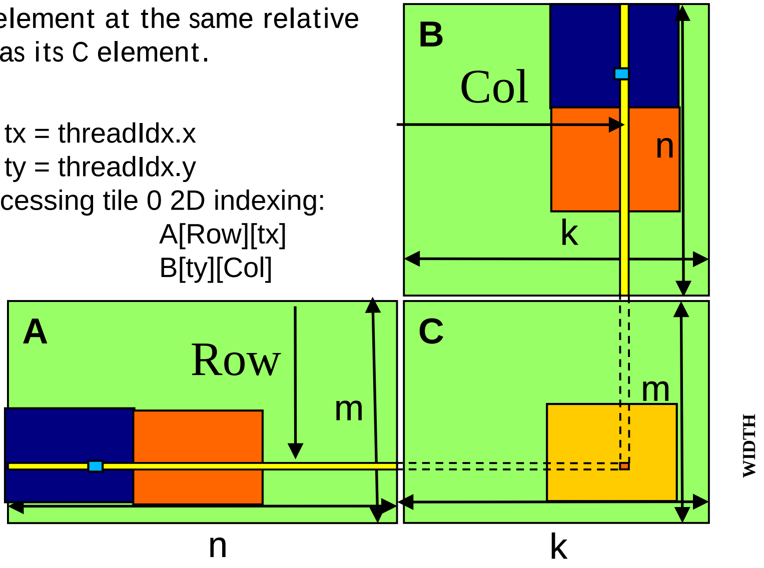


Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
…
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
Access direction in kernel code
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$

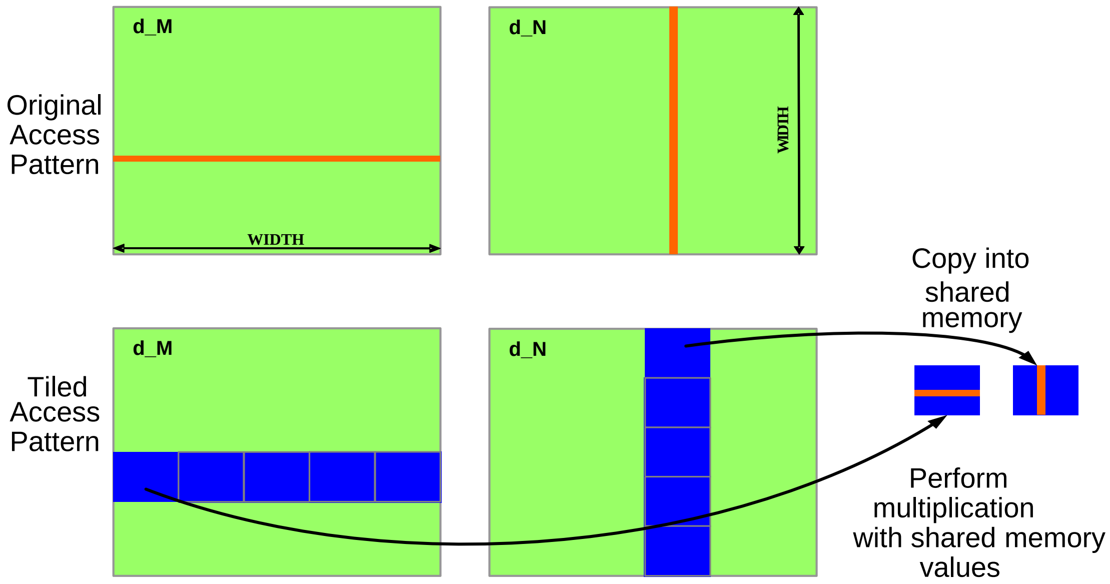
Loading an Input Tile

Have each thread load an A element and a B element at the same relative position as its C element.

```
int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
                A[Row][tx]
                B[ty][Col]
```



Corner Turning



GPU Teaching Kit