UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Data parallelism: GPU computing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: performance considerations

Memory bandwidth

- One of the most important factors of CUDA kernel performance is accessing data in the global memory (a DRAM memory).

- The process of reading a the status of a bit takes 10s of nanoseconds in modern DRAM chips. This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices.

- Modern DRAMs use parallelism to increase their rate of data access: Each time a DRAM location is accessed, a range of consecutive locations that include the requested location are actually accessed (DRAM bursts).

- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. In this case, the hardware combines, or **coalesces**, all these accesses into a consolidated access to consecutive DRAM locations.

Memory bandwidth

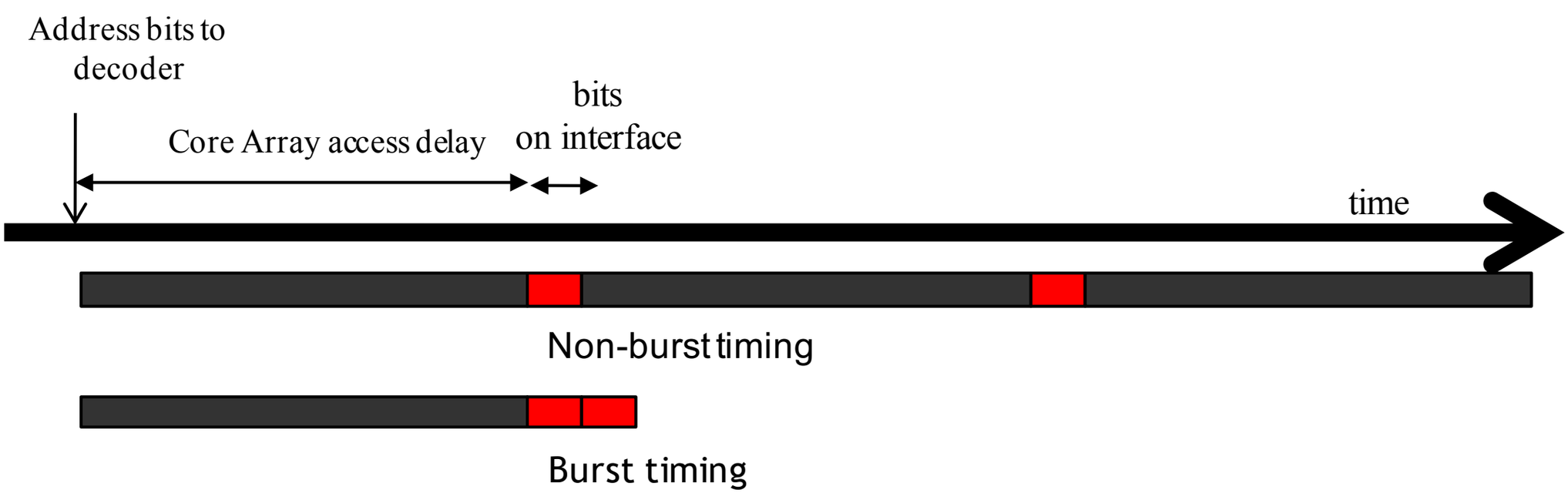
For example, for a given load instruction of a warp, if thread 0 accesses global memory location N, thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be **coalesced**. Such coalesced access allows the DRAMs to deliver data as a burst

- The process of reading a the status of a bit takes 10s of nanoseconds in modern DRAM chips. This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices.

- Modern DRAMs use parallelism to increase their rate of data access: Each time a DRAM location is accessed, a range of consecutive locations that include the requested location are actually accessed (DRAM bursts).

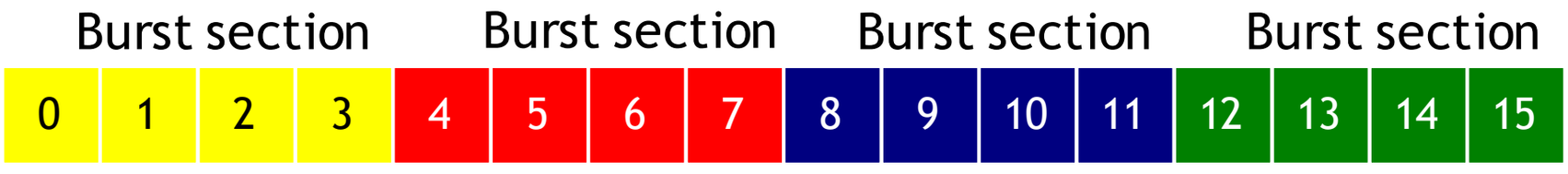
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. In this case, the hardware combines, or **coalesces**, all these accesses into a consolidated access to consecutive DRAM locations.

DRAM bursting



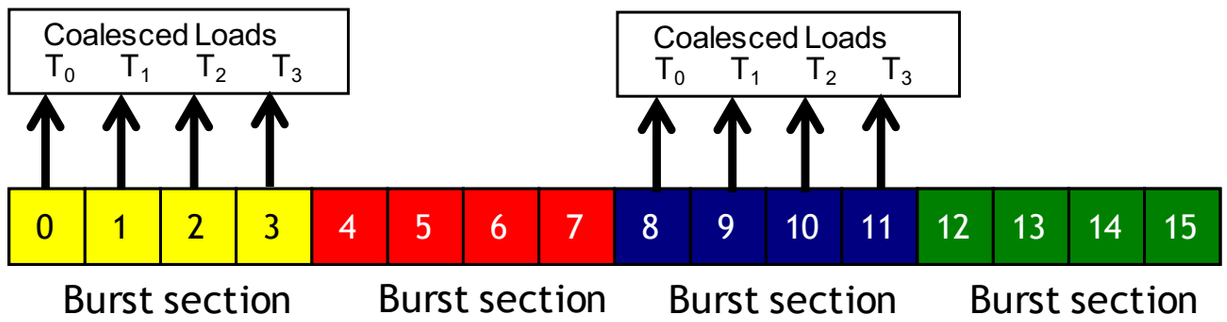
- Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

DRAM Burst – A System View

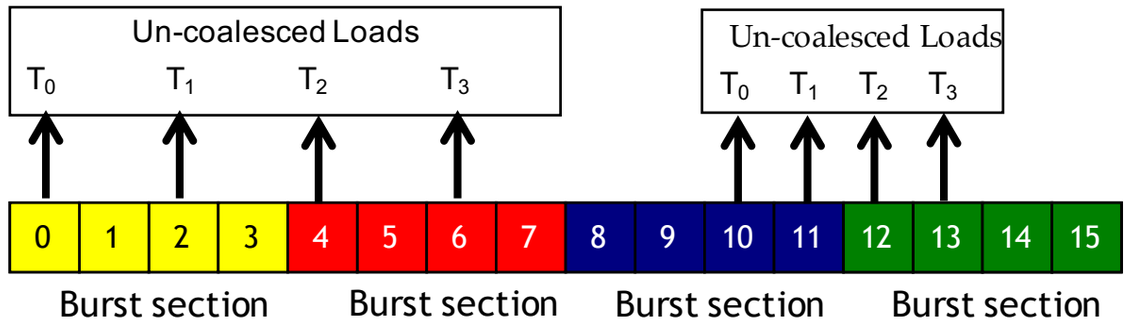


- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes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

Coalesced access



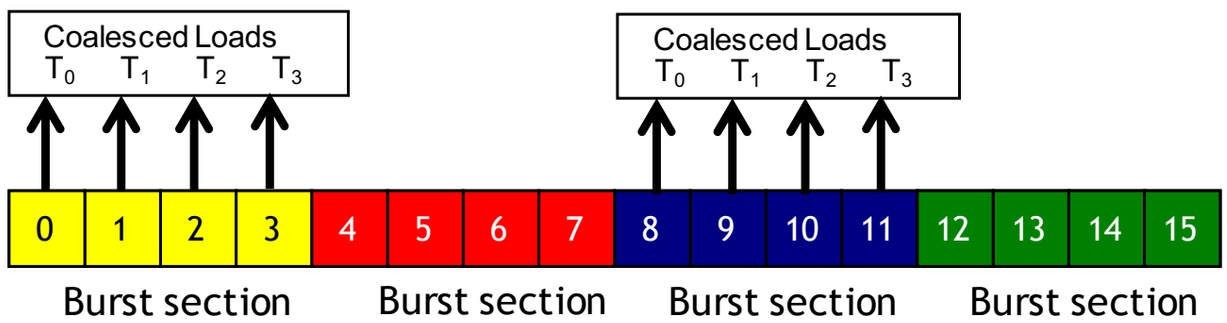
- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.



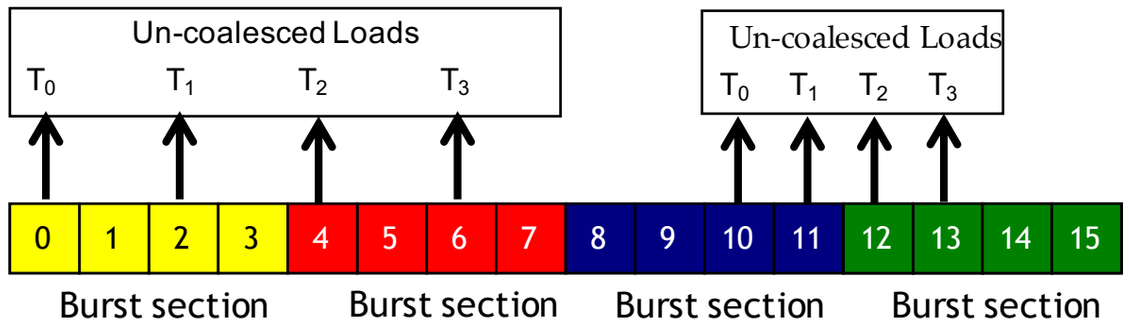
- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

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];

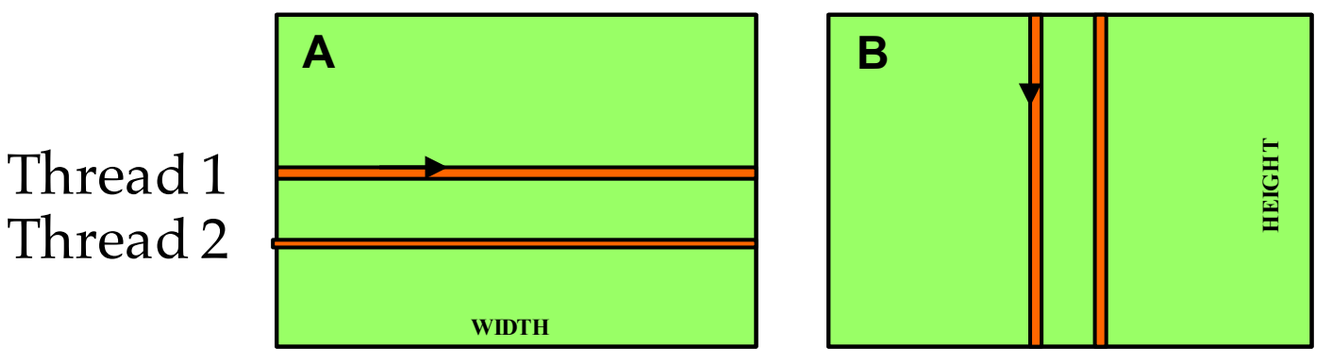


- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.



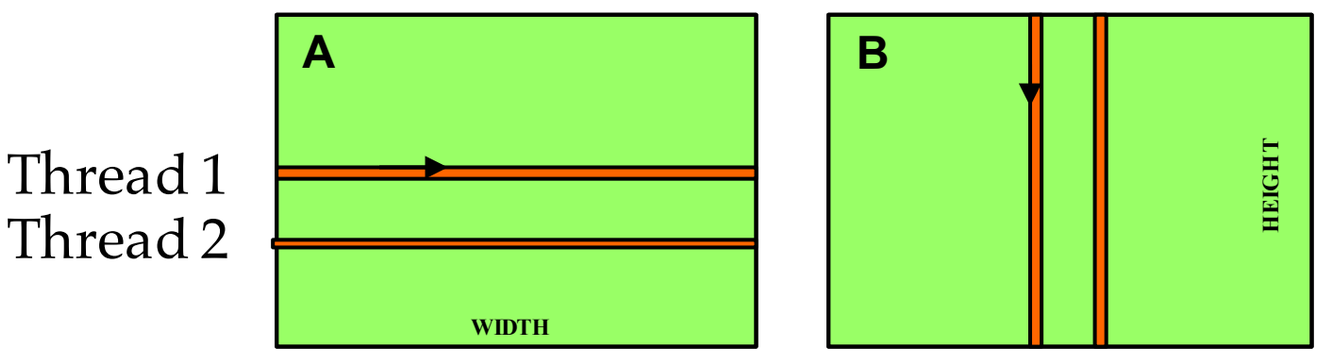
- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

Two Access Patterns of Basic Matrix Multiplication



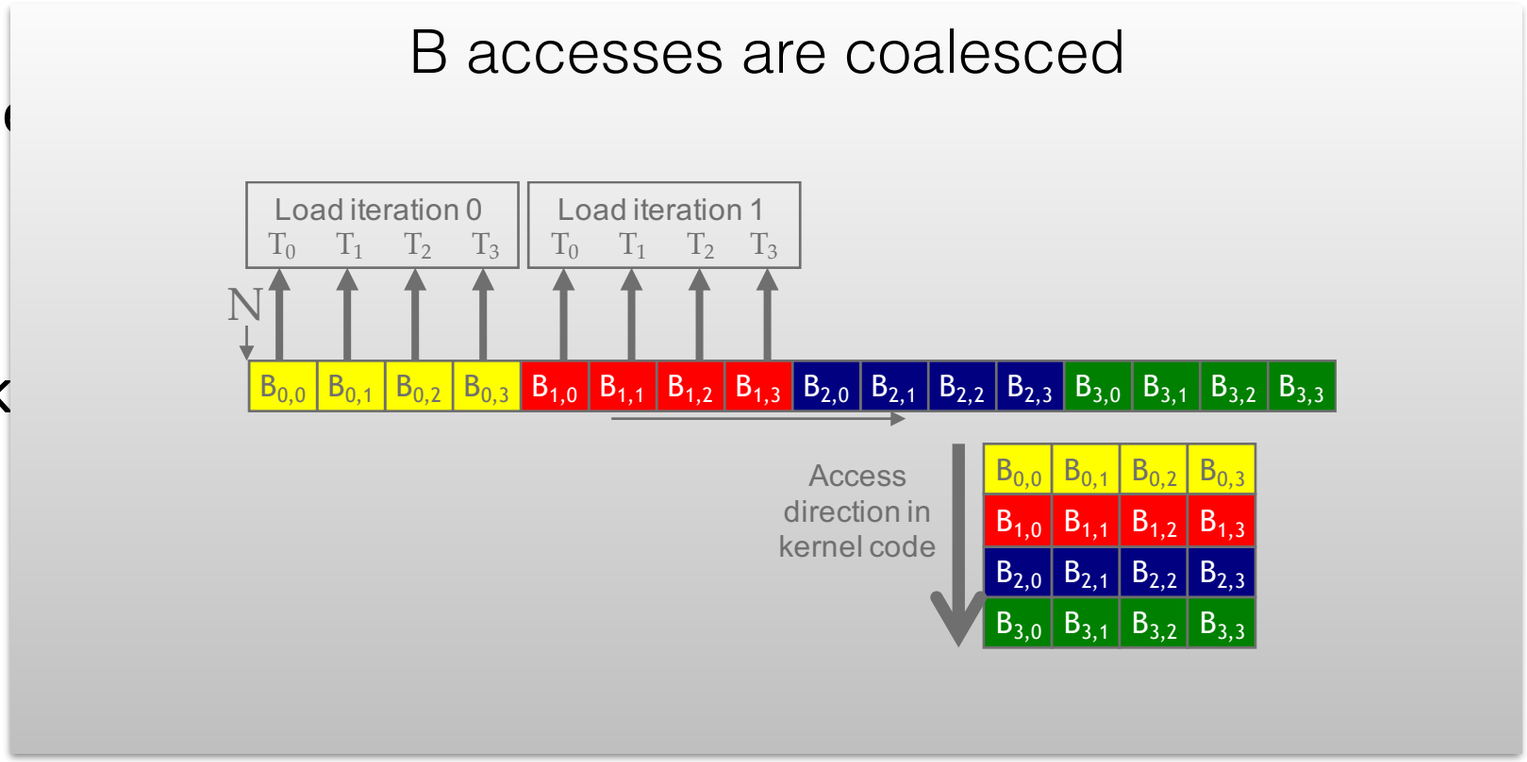
- i is the loop counter in the inner product loop of the kernel code
- A is m × n, B is n × k
- `Col = blockIdx.x*blockDim.x + threadIdx.x`

Two Access Patterns of Basic Matrix Multiplication

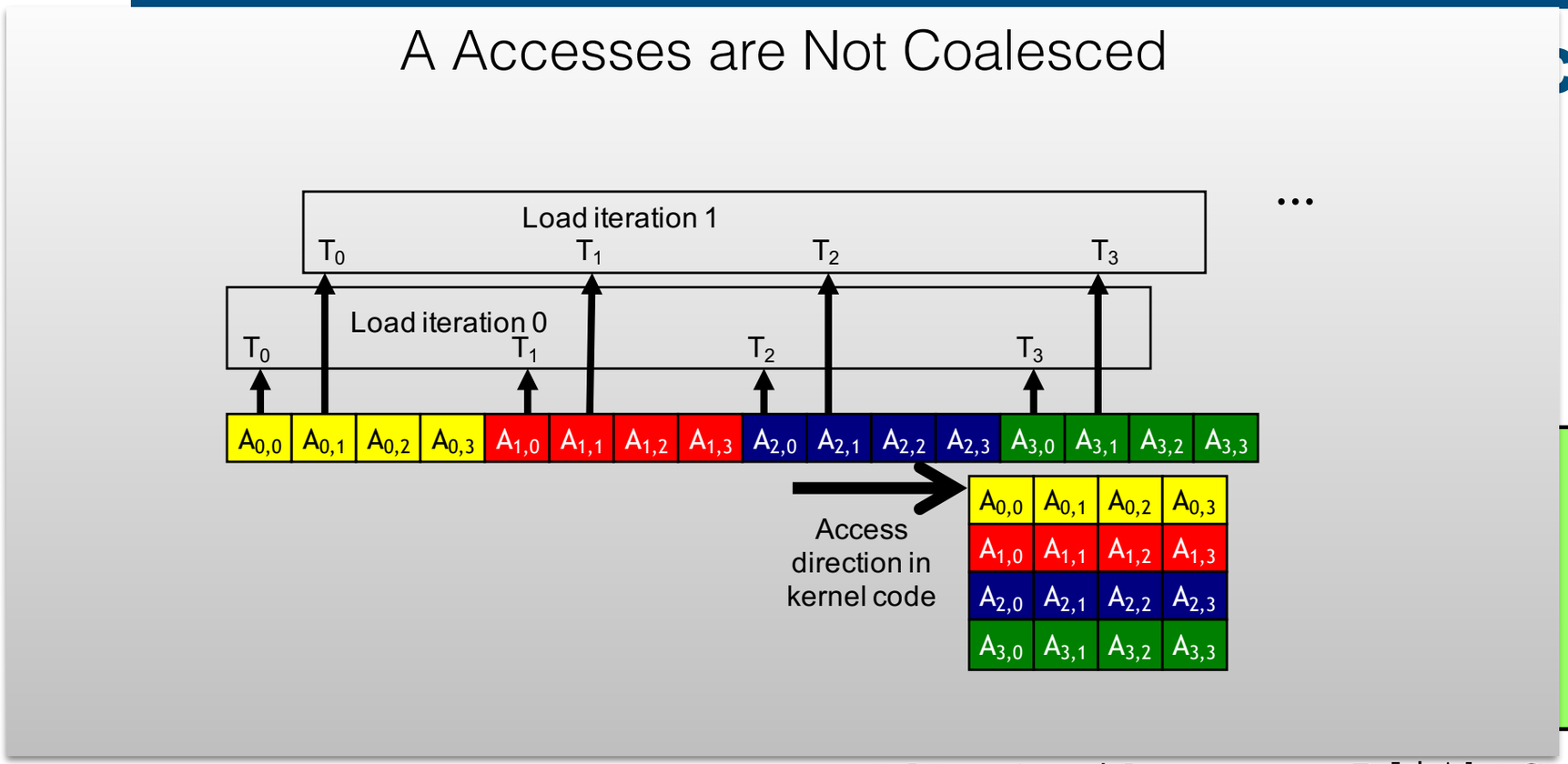


`A[Row*n+i]` `B[i*k+Col]`

- i is the loop counter in the inn
- A is m × n, B is n × k
- `Col = blockIdx.x*block`

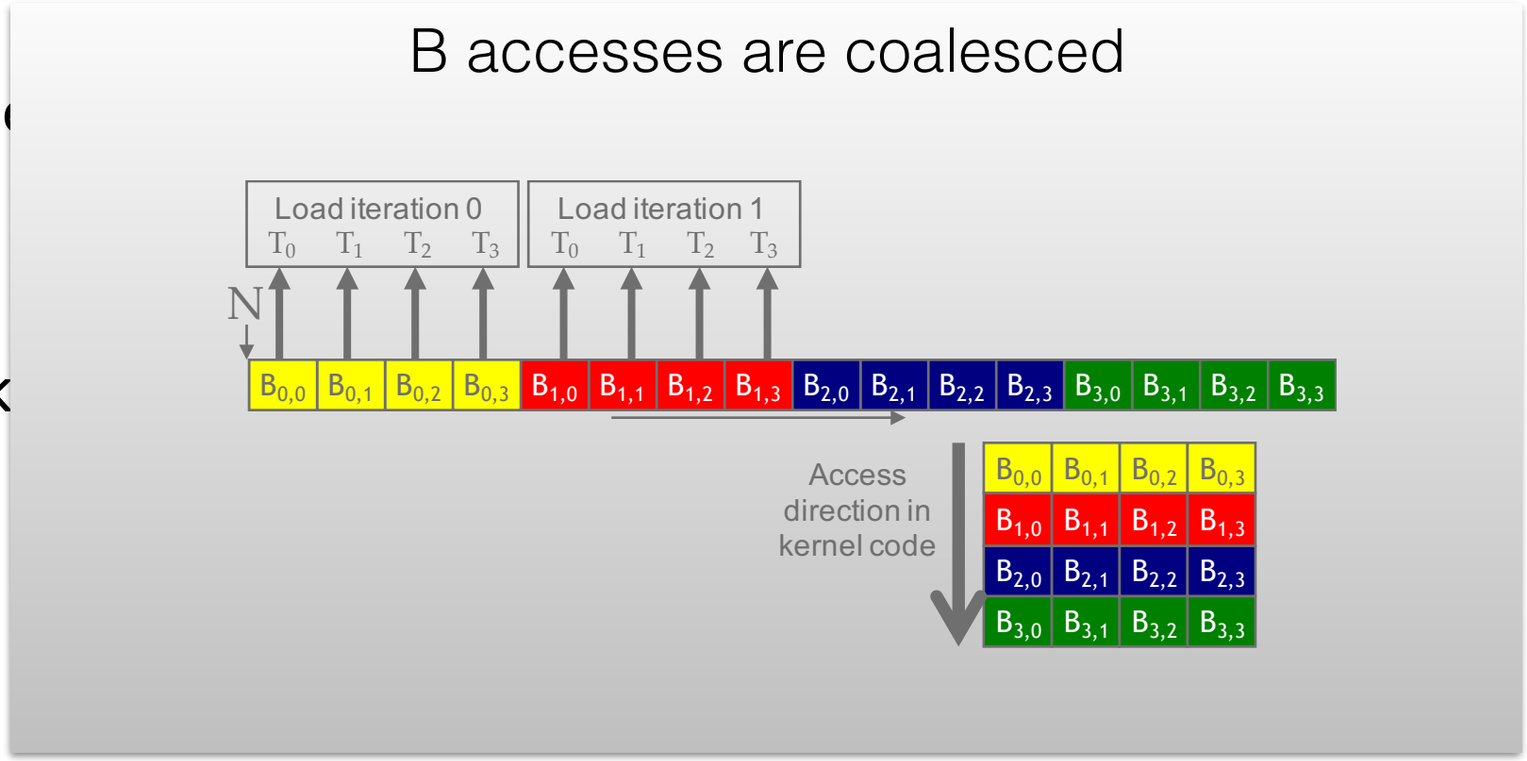


c Matrix



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

- i is the loop counter in the inn
- A is m × n, B is n × k
- `Col = blockIdx.x*block`



Non coalesced read

- If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing.

- The technique is called **corner turning**

- A tiled algorithm can be used to enable coalescing

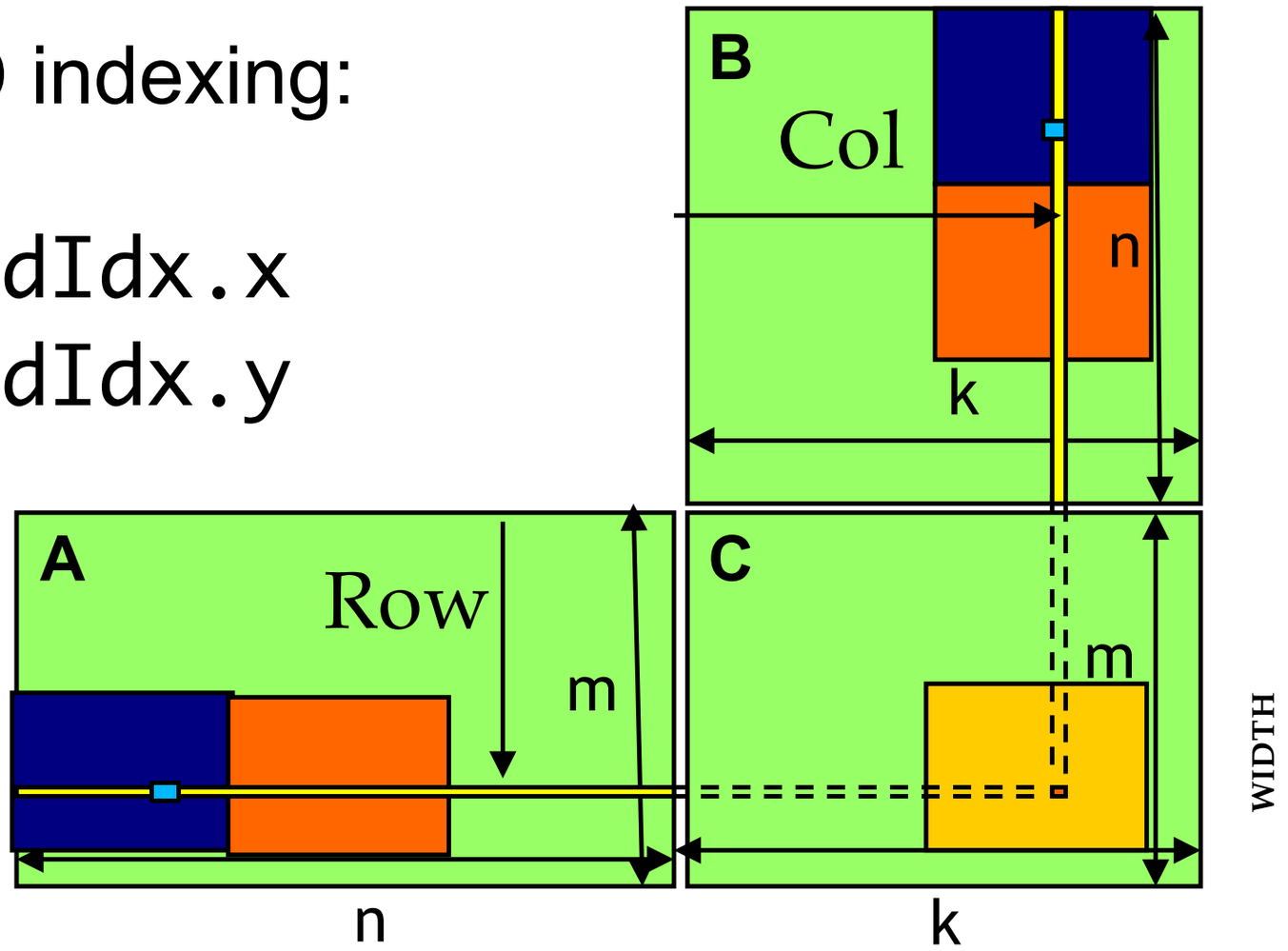
- Once the data is in shared memory, they can be accessed either on a row basis or a column basis

Loading an input tile

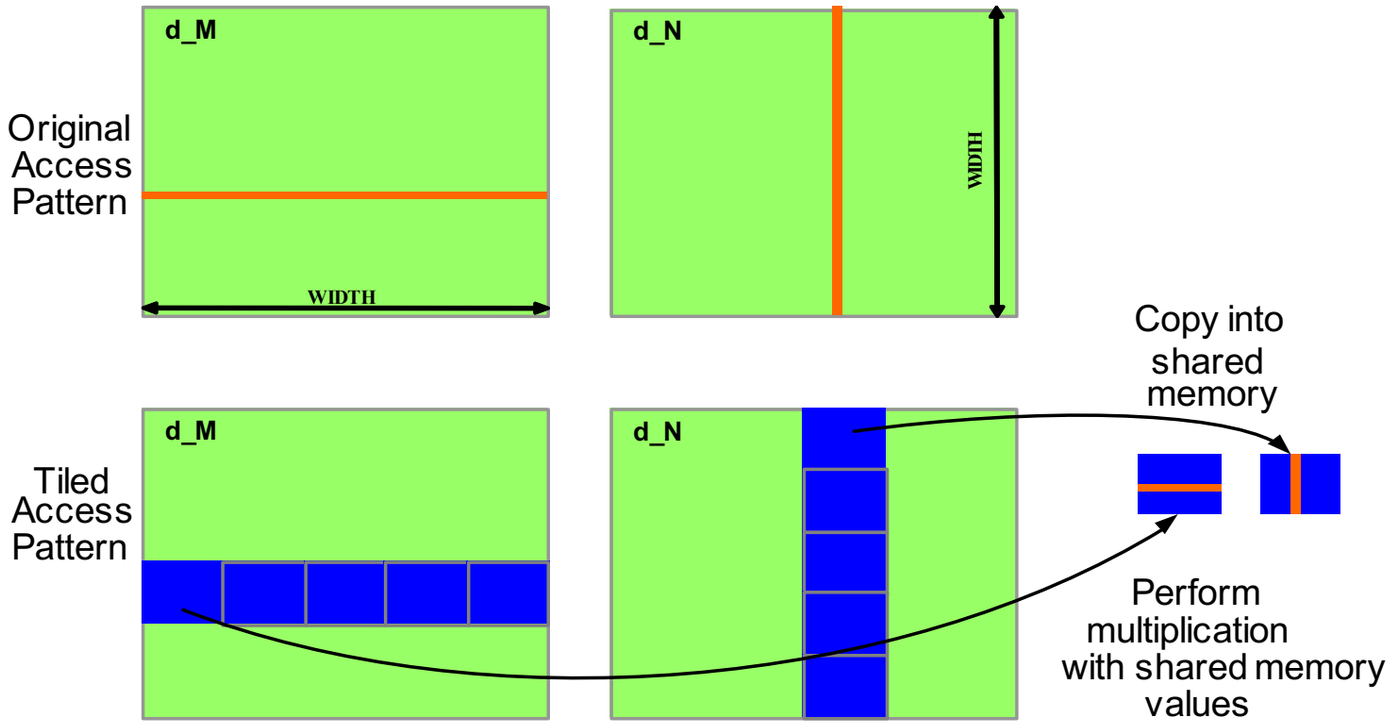
- Have each thread load an A element and a B element at the same relative position as its C element.

- Accessing tile 0 2D indexing:

```
int tx = threadIdx.x
int ty = threadIdx.y
A[Row][tx]
B[ty][Col]
```



Corner turning



Corner turning

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   // Identify the row and column of the P element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;
   float Pvalue = 0;

   // Loop over the M and N tiles required to compute the P element
   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
       // Collaborative loading of M and N tiles into shared memory
       Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
       Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k) {
           Pvalue += Mds[ty][k] * Nds[k][tx];
       }
       __syncthreads();
   }
   P[Row*Width + Col] = Pvalue;
}
```

Corner turning

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   // Identify the row and column of the P element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;
   float Pvalue = 0;

// Loop over the M and N tiles required to compute the P element
for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
    // Collaborative loading of M and N tiles into shared memory
    Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
```

The linearized index calculation is equivalent to `M[Row][ph*TILE_SIZE+tx]`.

Note that the column index used by the threads only differ in terms of `threadIdx`.

The Row Index is determined by blockIdx.y and threadIdx.y, which means that threads in the same thread block with identical blockIdx.y/threadIdx.y and adjacent threadIdx.x values will access adjacent M elements.

That is, each row of the tile is loaded by TILE_WIDTH threads whose threadIdx are identical in the y dimension and consecutive in the x dimension.

The hardware will coalesce these loads.

Corner turning

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   // Identify the row and column of the P element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;
   float Pvalue = 0;

   // Loop over the M and N tiles required to compute the P element
   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
       // Collaborative loading of M and N tiles into shared memory
       Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
       Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k) {
           Pvalue += Mds[ty][k] * Nds[k][tx];
       }
       __syncthreads();
   }
   P[Row*Width + Col] = Pvalue;
}
```

Corner turning

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   // Identify the row and column of the P element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;
   float Pvalue = 0;

// Loop over the M and N tiles required to compute the P element
for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
    // Collaborative loading of M and N tiles into shared memory
    Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
    Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k) {
```

In the case of N, the row index `ph*TILE_SIZE+ty` has the same value for all threads with the same threadIdx.y value. Note that the column index calculation for each thread, Col = bx*TILE_SIZE+tx . The first term, bx*TILE_SIZE, is the same for all threads in the same block. The second term, tx, is simply the threadIdx.x value. Therefore, threads with adjacent threadIdx.x values access adjacent N elements in a row.

The hardware will coalesce these loads.

Corner turning and tiling

- Note that in the simple algorithm, threads with adjacent `threadIdx.x` values access vertically adjacent elements that are not physically adjacent in the row major layout.

- The tiled algorithm “transformed” this into a different access pattern where threads with adjacent `threadIdx.x` values access horizontally adjacent elements.
  That is we turned a vertical access pattern into a horizontal access pattern, which is sometimes referred to as **corner turning**.

- In the tiled algorithm, loads to both the M and N elements are coalesced.

Corner turning and tiling

- Note that in the simple algorithm, threads with adjacent `threadIdx.x` values access vertically adjacent elements that are not physically adjacent in the row major layout.

The tiled matrix multiplication algorithm has two advantages over the simple matrix multiplication:

1. number of memory loads are reduced due to the reuse of data in the shared memory.
2. the remaining memory loads are coalesced so the DRAM bandwidth utilization is further improved.

elements.
That is we turned a vertical access pattern into a horizontal access pattern, which is sometimes referred to as **corner turning**.

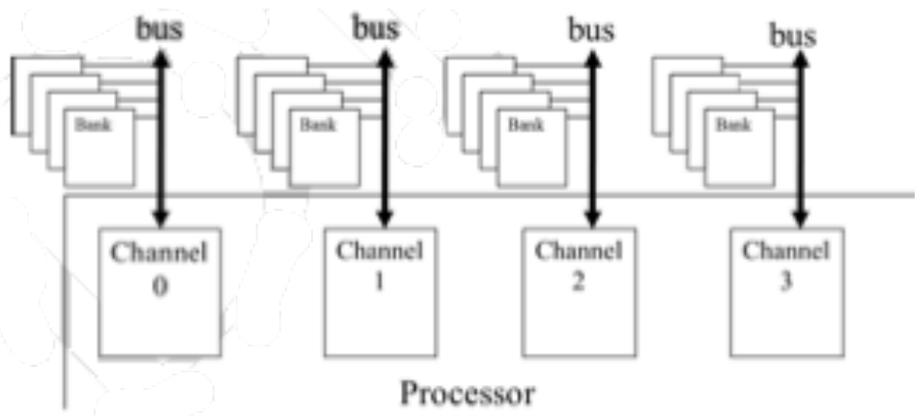
- In the tiled algorithm, loads to both the M and N elements are coalesced.

Memory parallelism

- DRAM bursting is a form of parallel organization: multiple locations around are accessed in the DRAM core array in parallel. However, bursting alone is not sufficient to realize the level of DRAM access bandwidth required by modern processors.

- DRAM systems typically employ two more forms of parallel organization – banks and channels.

- At the highest level, a processor contains one or more channels. Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor.



Banks

- For each channel, the number of banks connected to it is determined by the number of banks required to fully utilize the data transfer bandwidth of the bus.
  - The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern double data rate (DDR) busses perform two data transfers per clock cycle, one at the rising edge and one at the falling edge of each clock cycle. For example, a 64-bit DDR bus with a clock frequency of 1 GHz has a bandwidth of 8B * 2 * 1GHz = 16GB/sec.

Banks and DRAM bursting

Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

- In order to achieve the memory access bandwidth specified for device, there must be a sufficient number of threads making simultaneous memory accesses.
- A distribution scheme referred to as interleaved data distribution, spreads the elements across the banks and channels in the system.

Credits

- These slides report material from:
  - NVIDIA GPU Teaching Kit

Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - Chapt. 4-6