SSE3054 Multicore Systems

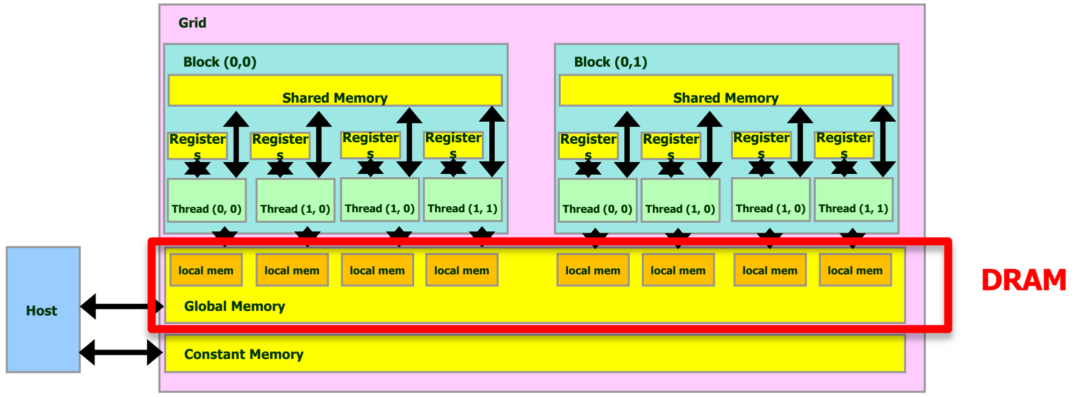
Performance Consideration 1

Prof. Seokin Hong

Agenda

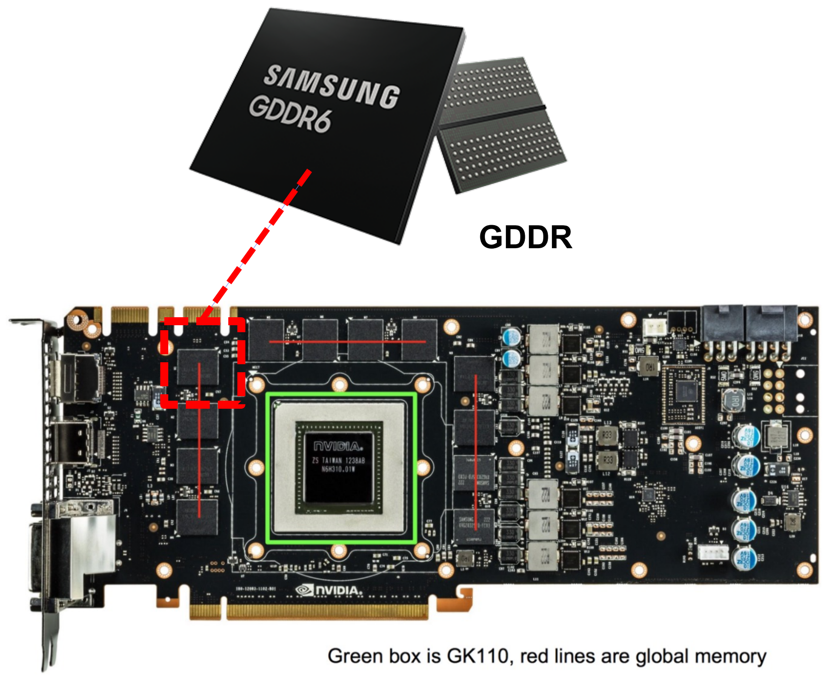
- **Memory Optimizations**
  - **More about Global Memory**
  - **Memory Coalescing** to fully utilize global memory bandwidth
  - **Reducing Bank Conflict** to fully utilize shared memory bandwidth
- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware
- **Considering Occupancy**
  - Dynamic Partitioning of Resources
- **Considering Thread Granularity**

More about Global Memory

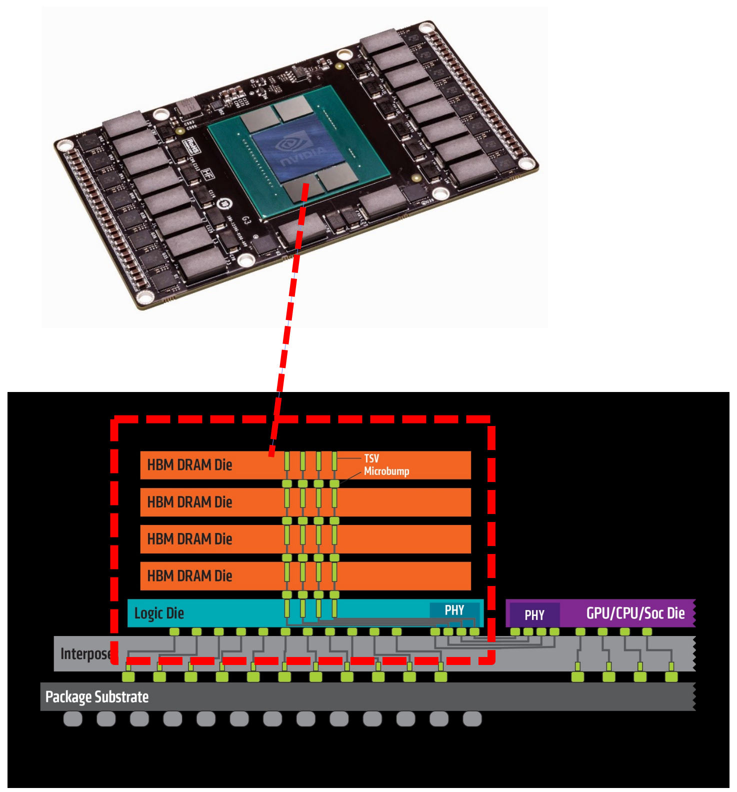


CUDA Global Memory = DRAM

- The global memory of a CUDA device is implemented with DRAM (Dynamic Random Access Memory)
  - Graphic DRAM is built for much **higher bandwidth**
  - GDDR (Graphic DDR)
  - HBM (3D-stacked High Bandwidth Memory)

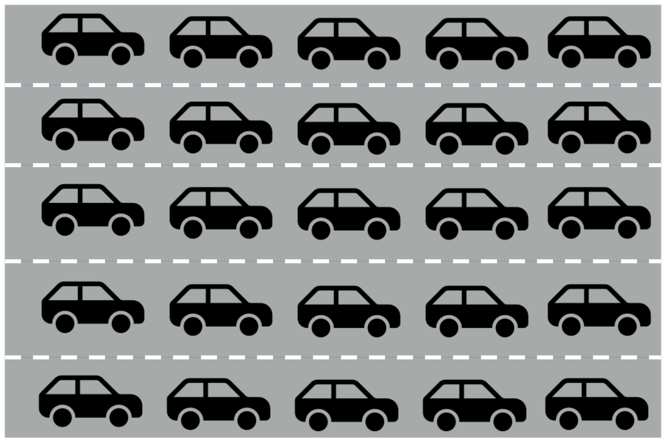


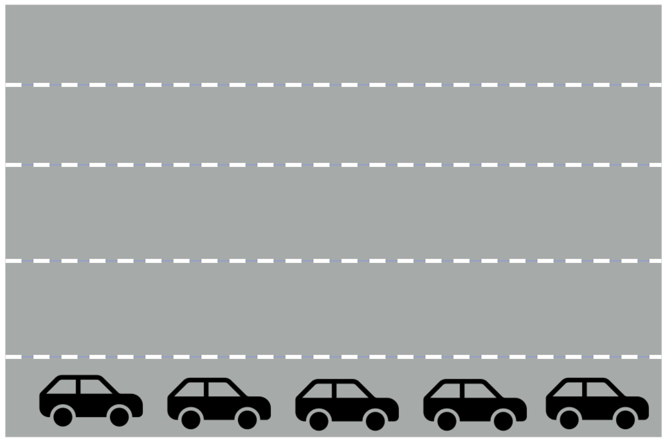
Green box is GK110, red lines are global memory



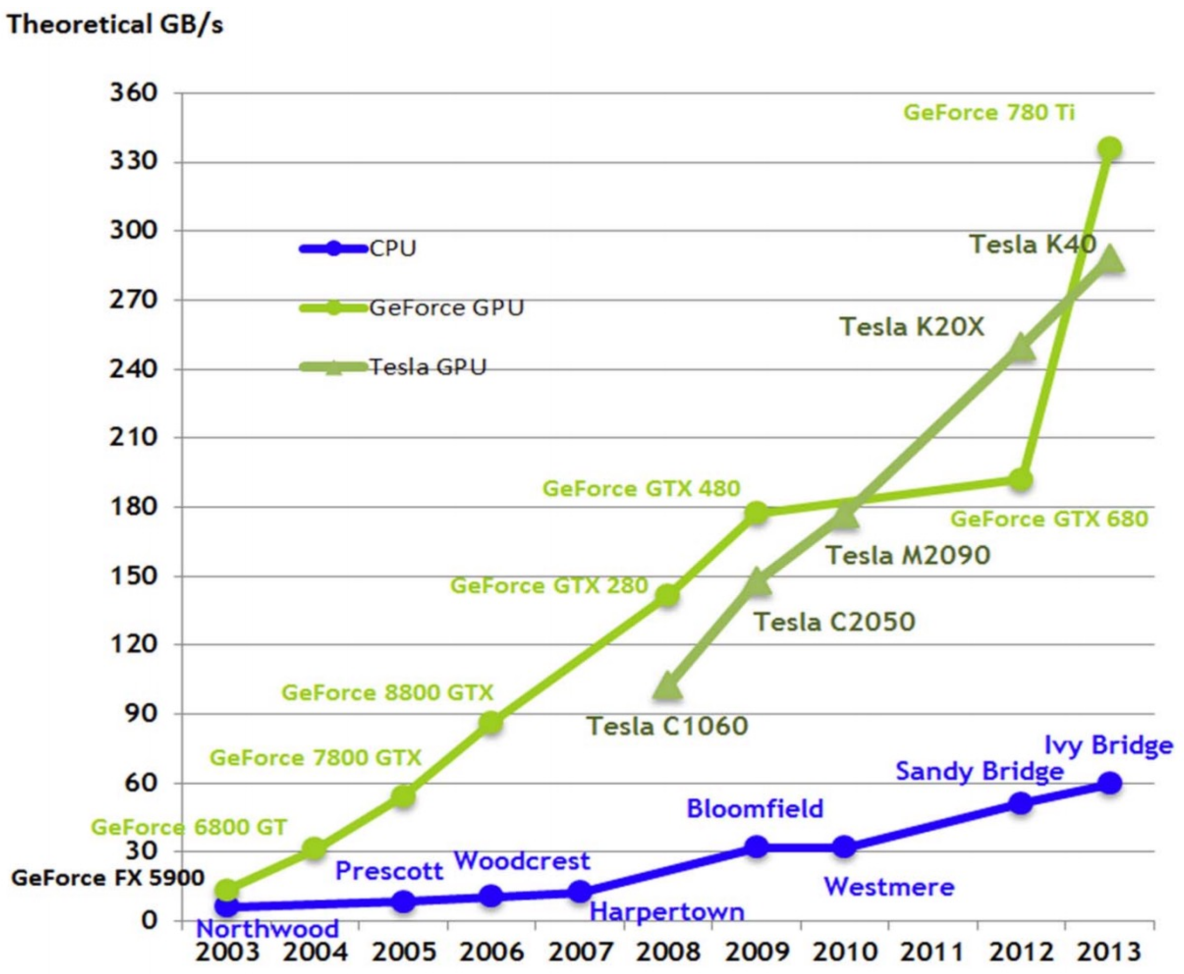
Global Memory is a major bottleneck

- DRAM is slow → long access latency (~ 300 cycles)
- Graphic DRAM has much higher bandwidth than standard DRAM
  - E.g., P100 with 4 HBM : 1TB /s
- So, programming techniques are required to fully utilize the DRAM bandwidth!!
- Need to understand DRAM to know causes of the DRAM bandwidth problem in CUDA programming





Memory Bandwidth Comparison



P100 with 4 HBM : 1TB /s

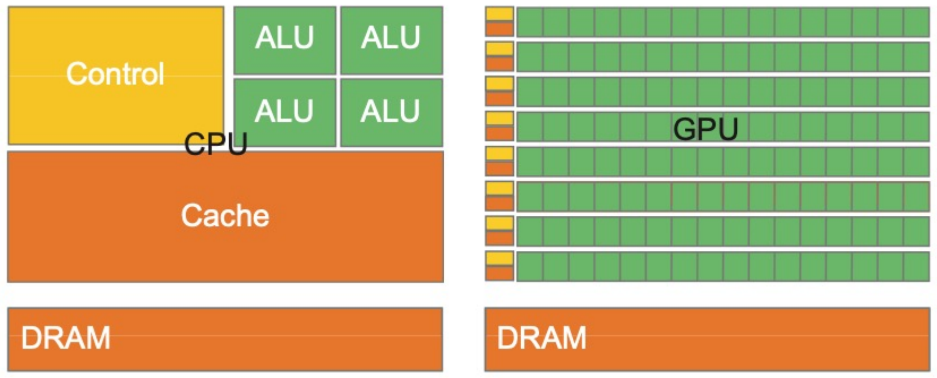
Why do GPUs need higher bandwidth DRAM?

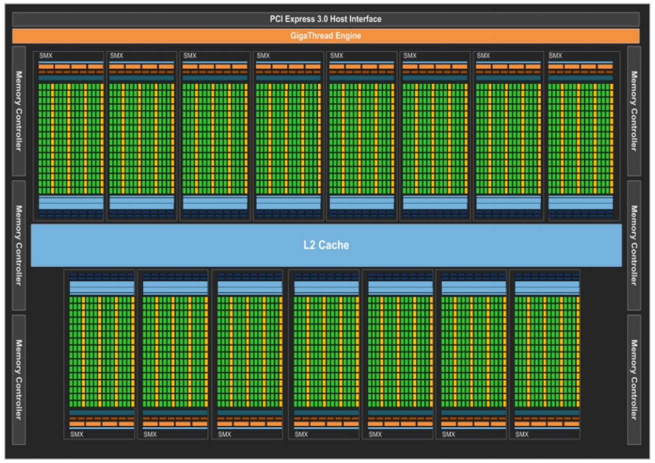
- **Lots of compute**
  - 24 Streaming Multiprocessors, each with 128 execution units

- **Lots of threads**
  - 64 warps of 32 threads per Streaming Multiprocessors

    → 49,152 threads executing simultaneously on 3072 execution units

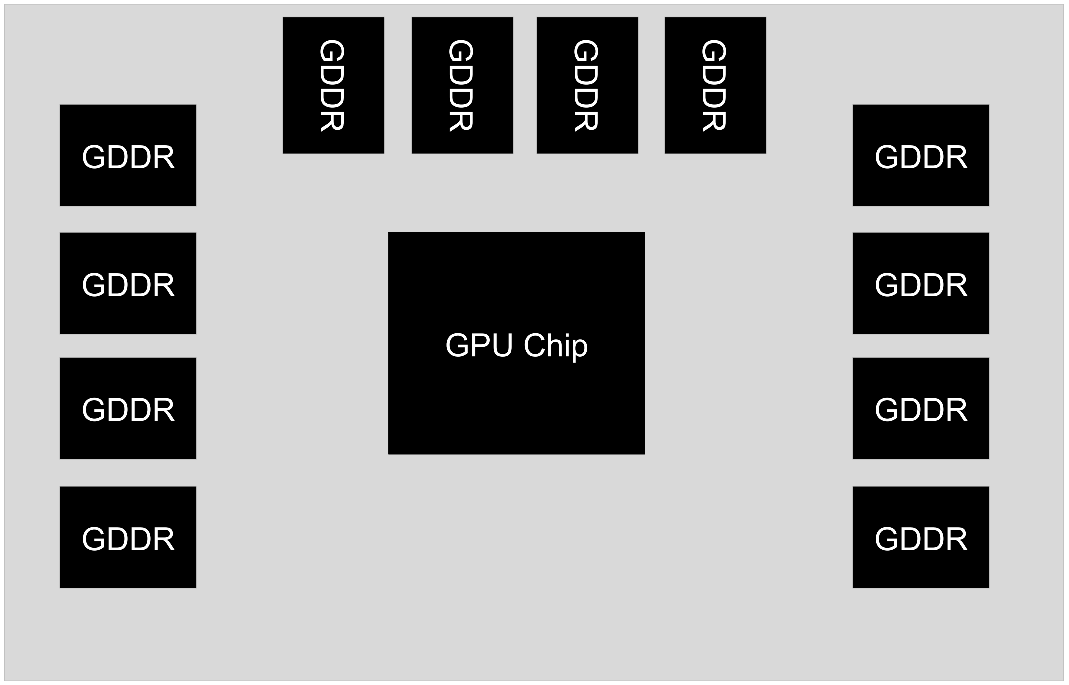
→ To feed the threads, GPU must be capable of moving extremely large amount of data in and out of Global memory

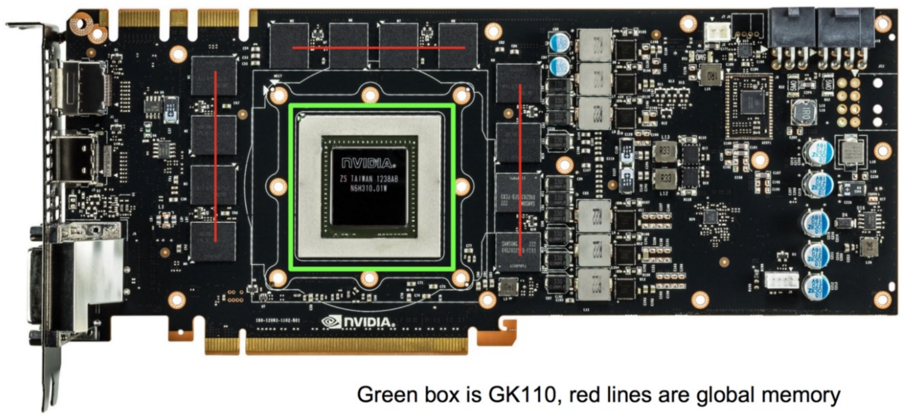




DRAM Subsystem Organization

- Channel
- Chip
- Bank
- Row/Column

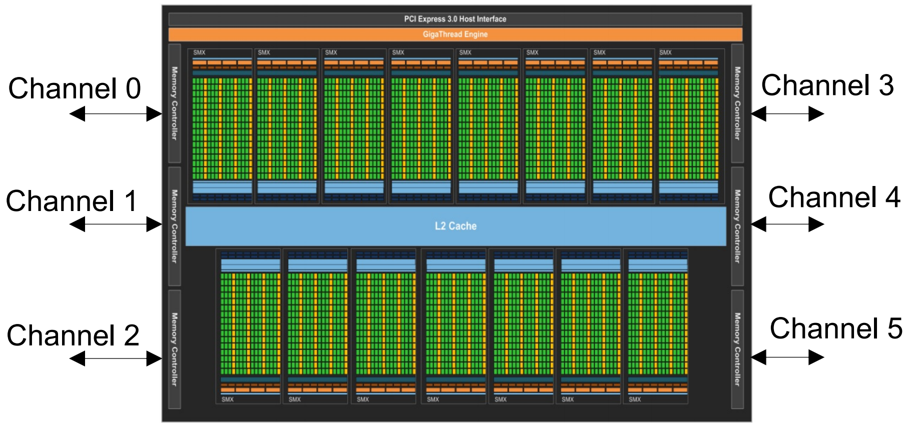


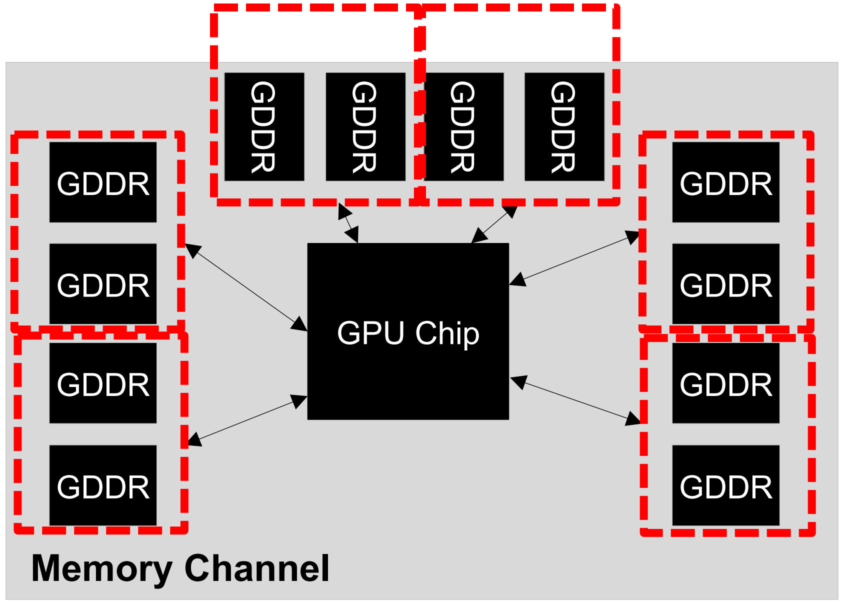


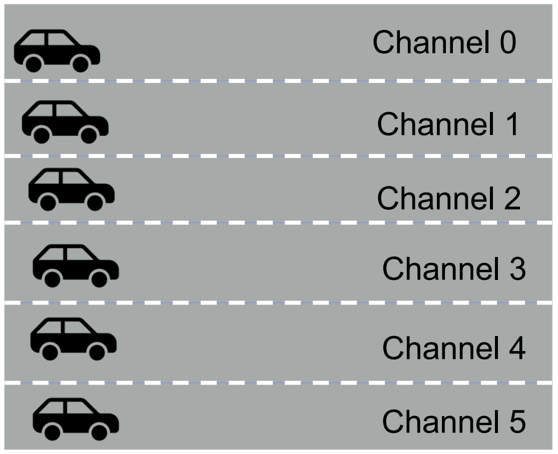
Green box is GK110, red lines are global memory

Channel

- **Each memory channel operates independently**
  - Can serve memory requests independently to read or write cache blocks (32B or 64B)
  - Typically 64-bits wide with the command and address bus
  - It is important to exploit **channel-level parallelism** by uniformly distributing memory requests across memory channels

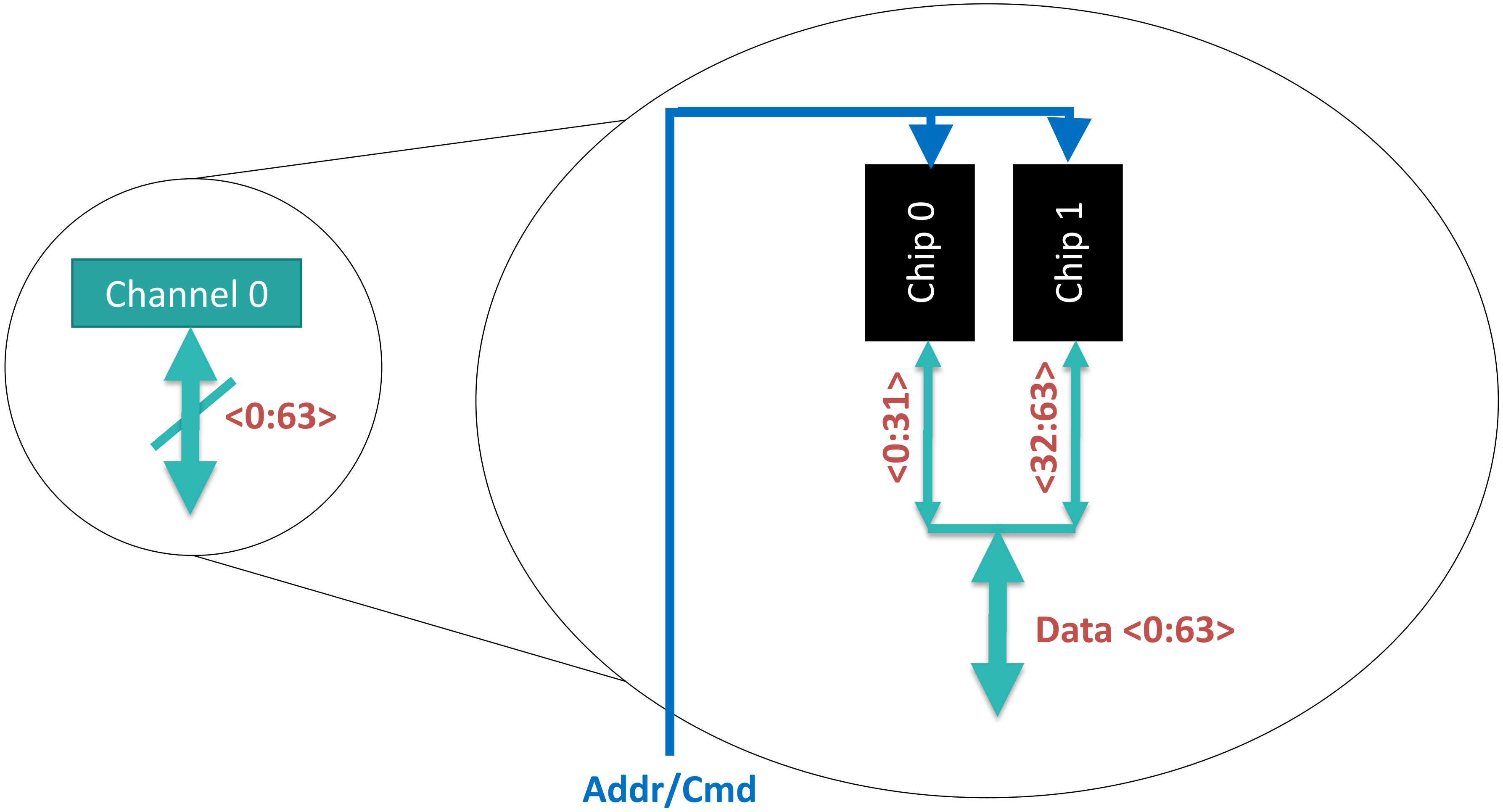






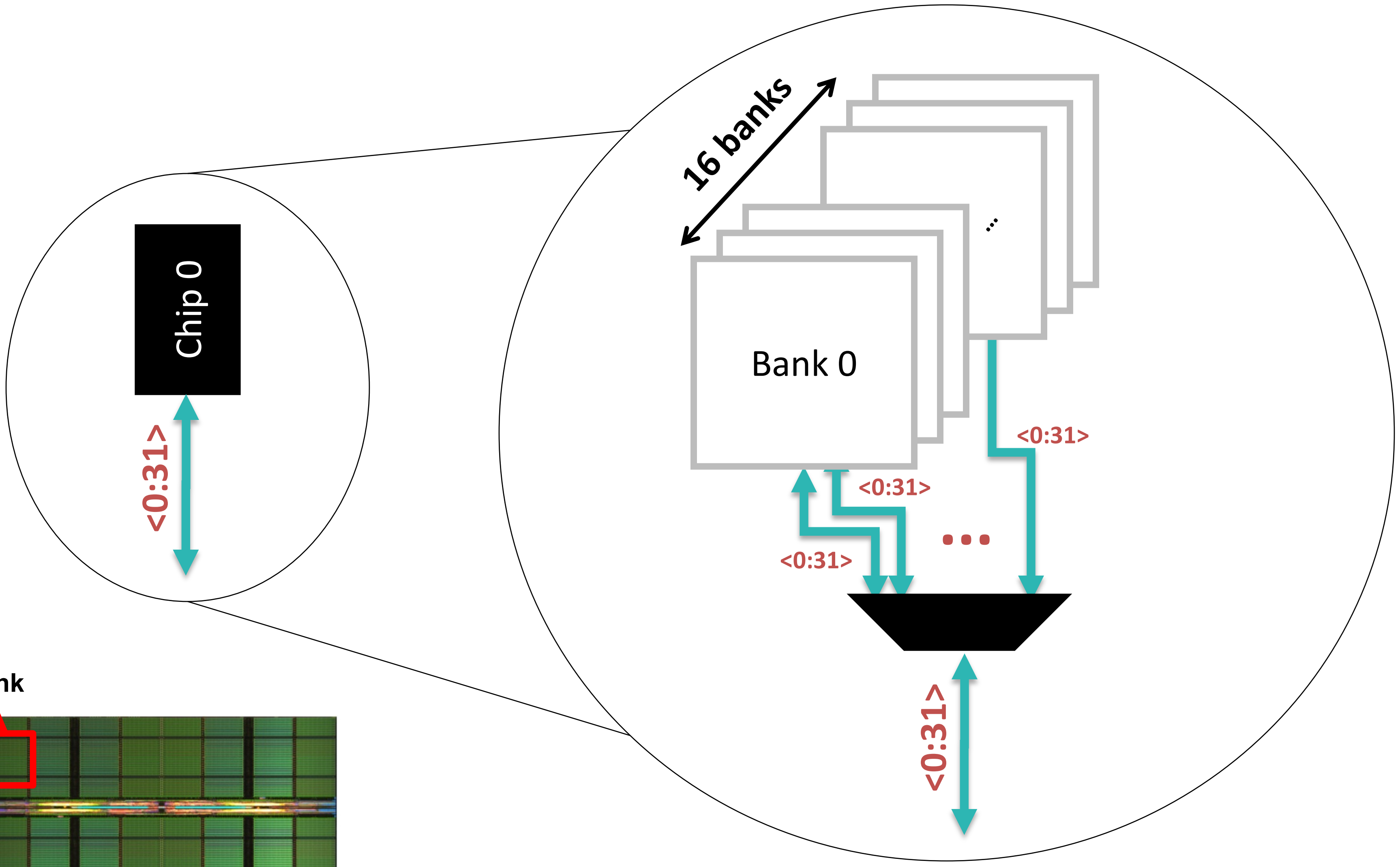
Chip

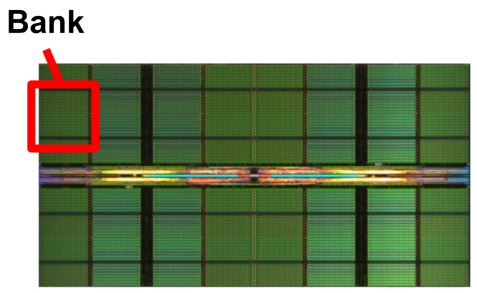
- **All chips comprising a channel are controlled at the same time**
  - Respond to a single command
  - Share address and command buses, but provide different data



Slide credit : modified from Prof. Onur's slide

Bank

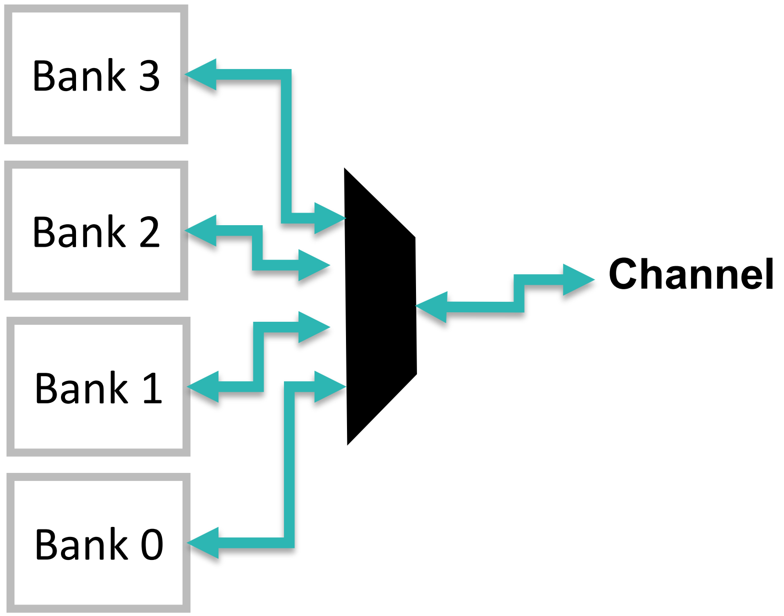


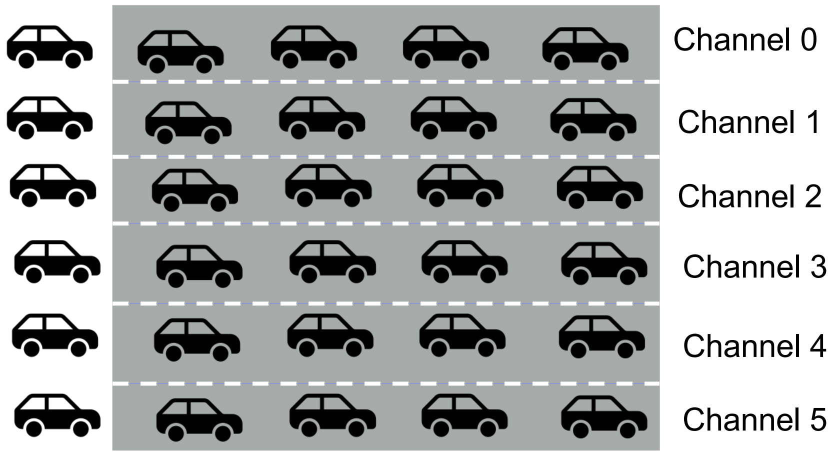


Slide credit : modified from Prof. Onur's slide

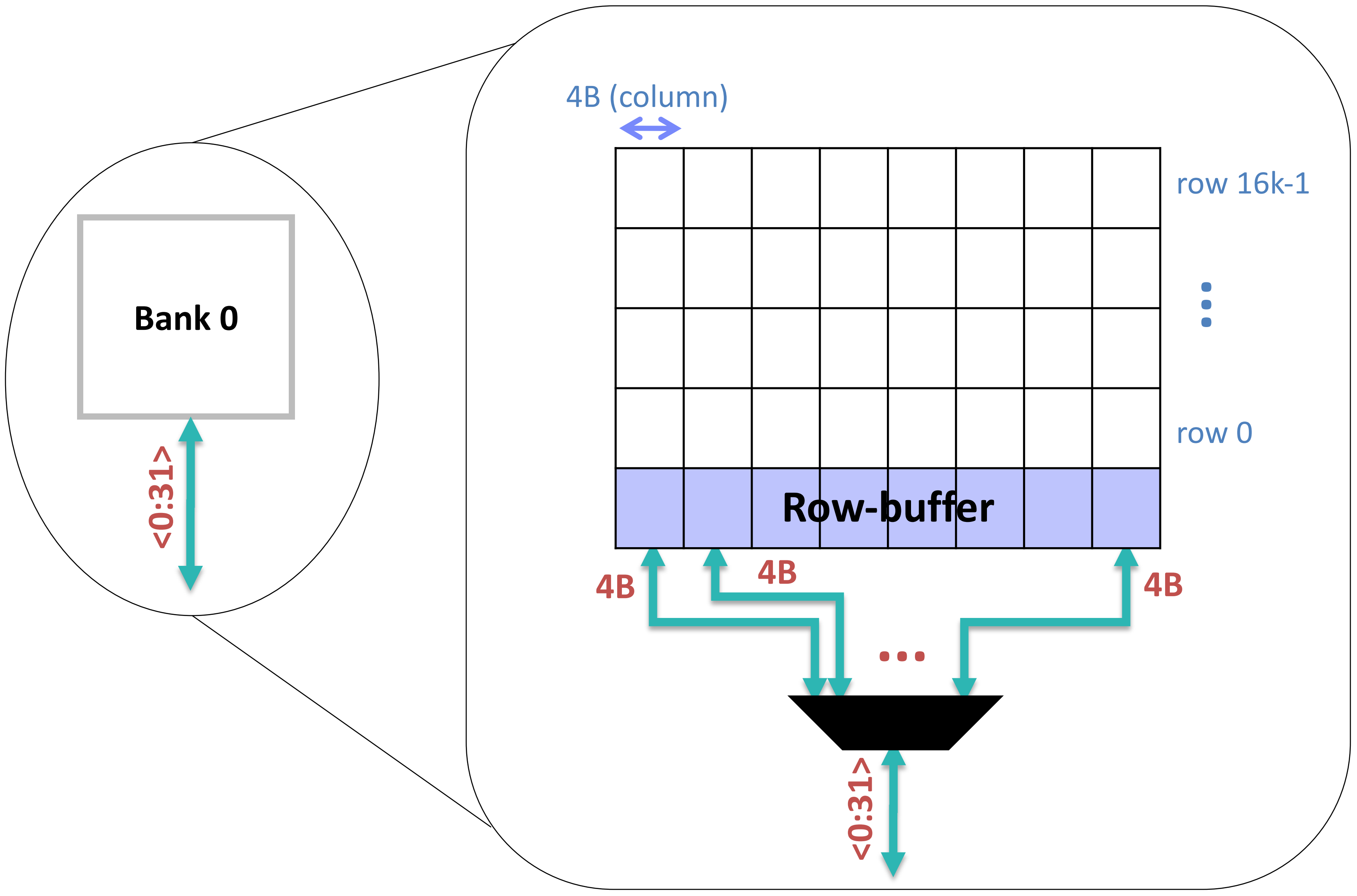
Bank (cont'd)

- A chip consists of multiple banks
- Banks share command/address/data buses
- **Each bank operate independently**
- It is important to exploit **bank-level parallelism** by uniformly distributing memory requests across banks





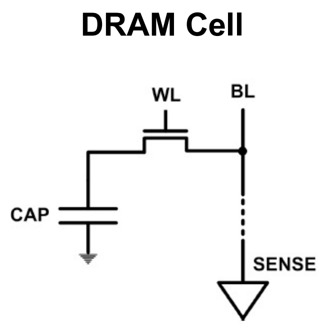
Breaking down a Bank

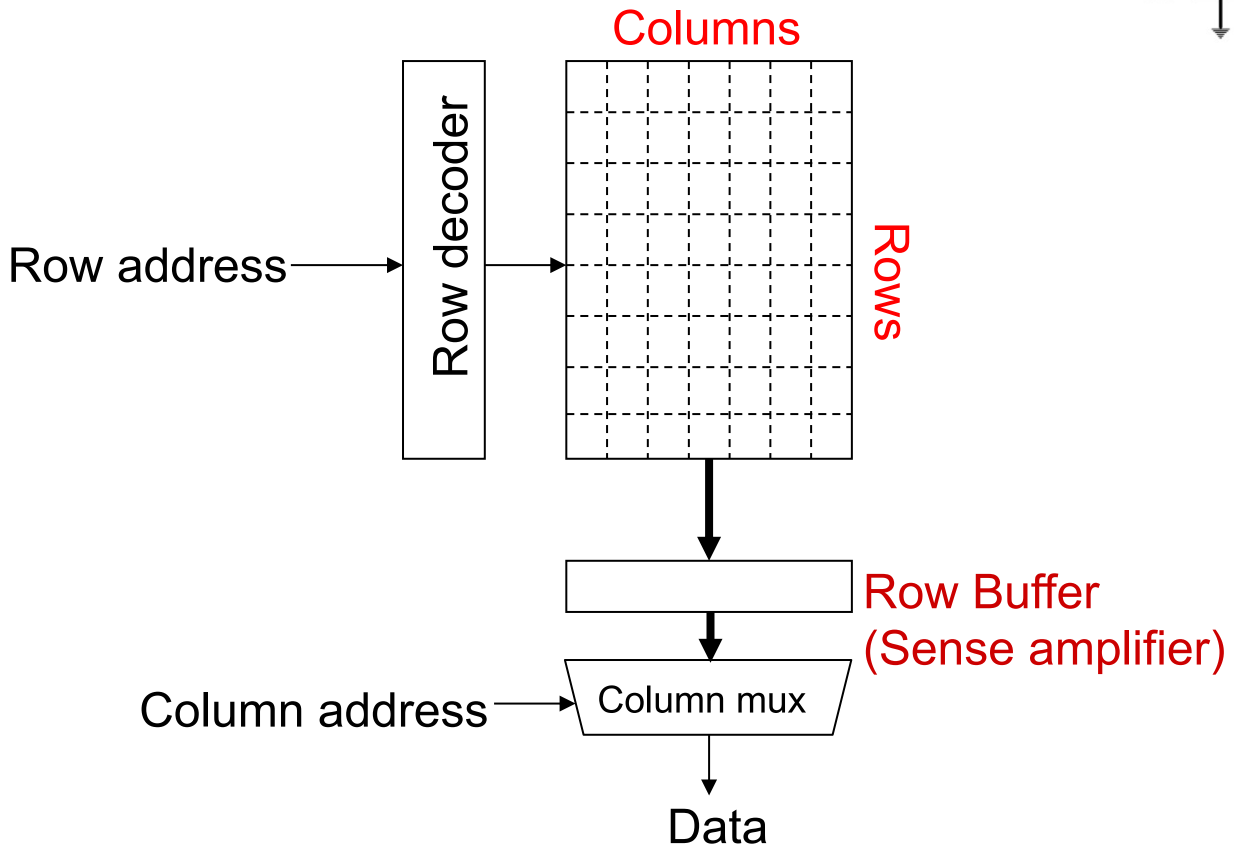


Slide credit : modified from Prof. Onur's slide

Breaking down a Bank

- A DRAM bank is a 2D array of cells: rows x columns
- A "DRAM row" is also called a "DRAM page"
- "Sense amplifiers" is also called "row buffer"





Slide credit : modified from Prof. Onur's slide

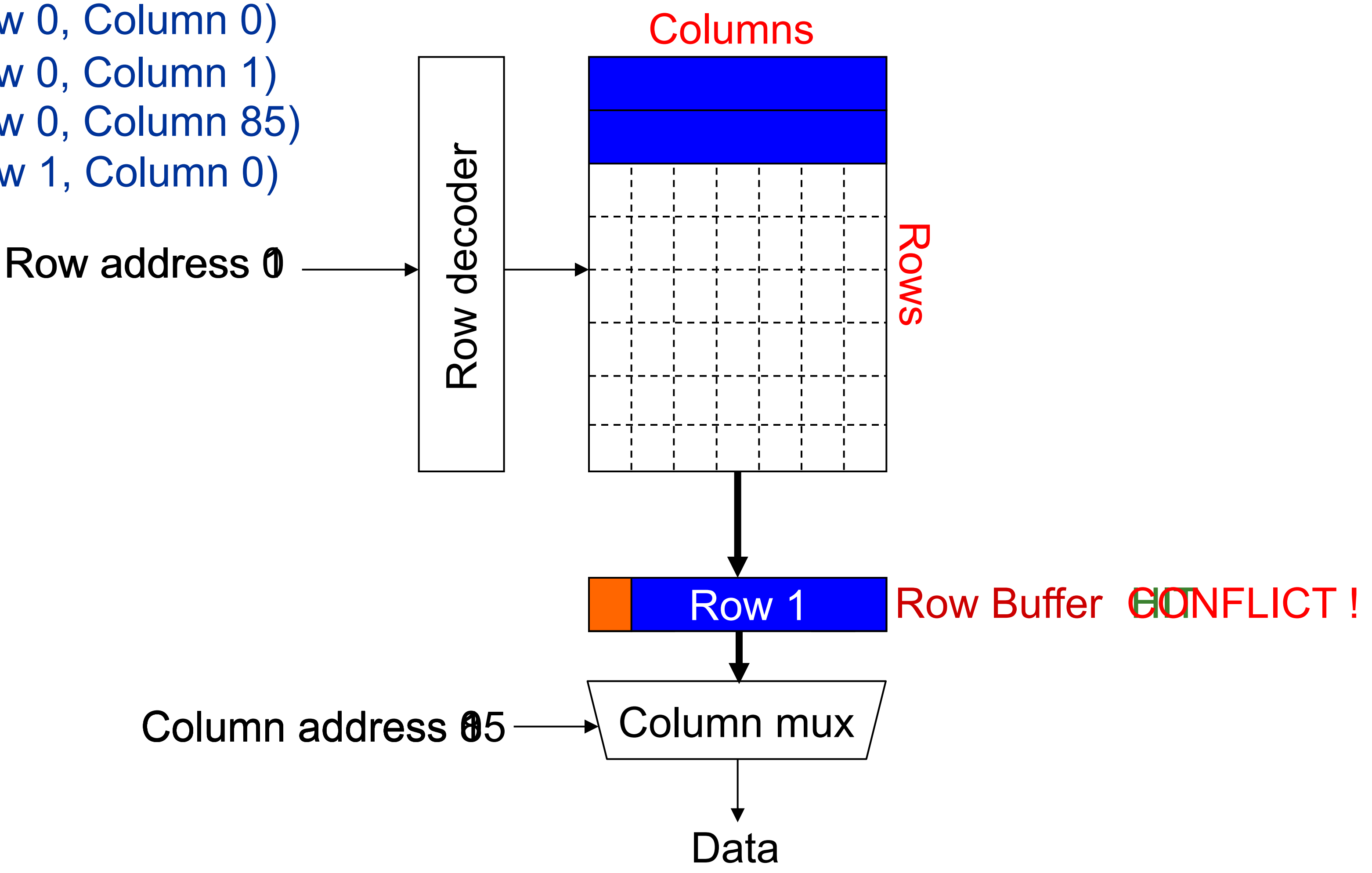
DRAM Bank Operation

- Each address is a <row, column> pair
- Access to an “close row”
  - Activate command opens row (placed into row buffer)
  - Read/write command reads/writes column in the row buffer
  - Precharge command closes the row and prepares the bank for next access
- Activation and Precharge are very slow!!!
- Access to an “open row”
  - No need for activation → **Fast**

Slide credit : modified from Prof. Onur’s slide

DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

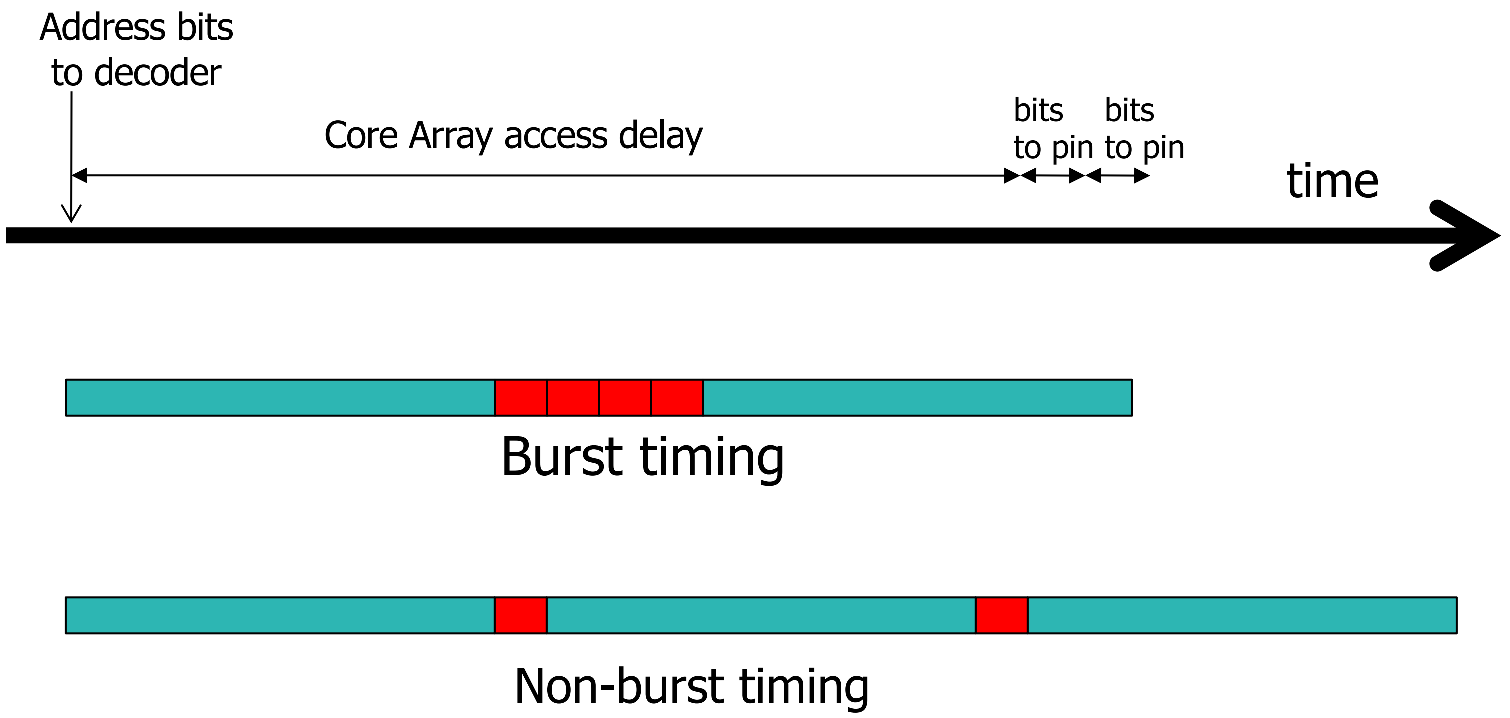


It is important to maximize the row buffer hit !!
Accesses on consecutive memory address → High row buffer hit rate!

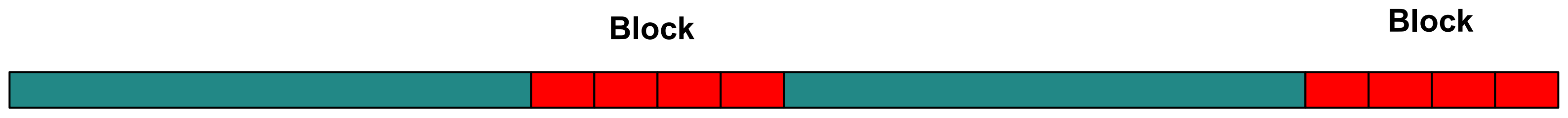
Slide credit : modified from Prof. Onur's slide

DRAM Bursting

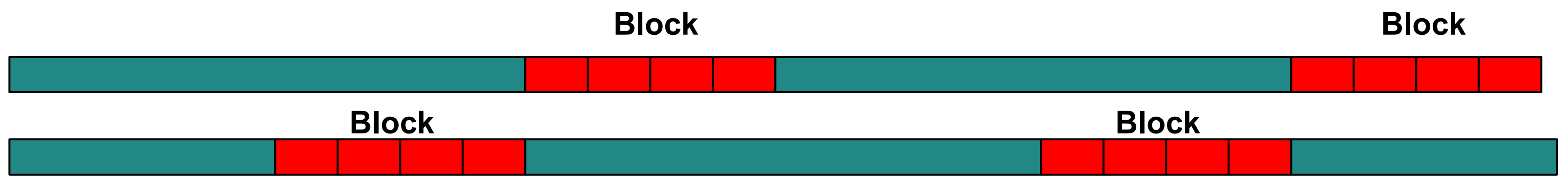
- **DRAM Bursting:** transfer a block (e.g., 64B) in N steps through a memory channel
- Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.



DRAM Bursting with Banking



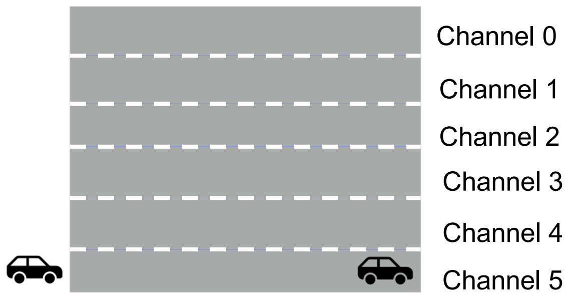
Single-Bank burst timing, dead time on interface



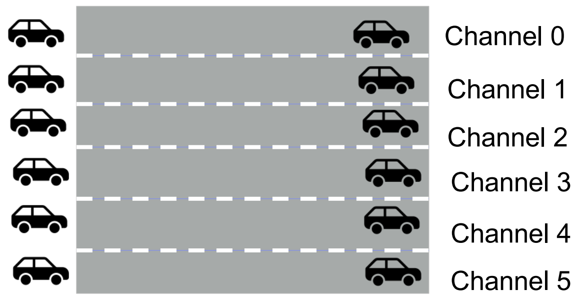
Multi-Bank burst timing, reduced dead time

Summary: To fully utilize the DRAM bandwidth

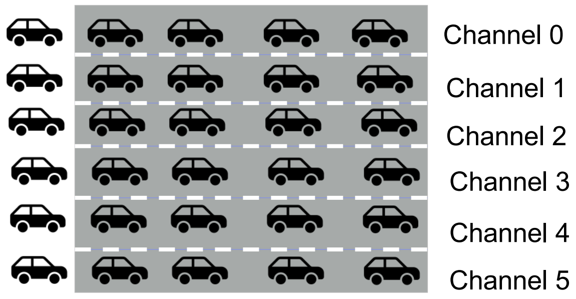
- Exploit **channel-level parallelism** by uniformly distributing memory requests across memory channels
- Exploit **bank-level parallelism** by uniformly distributing memory requests across banks
- Maximize the row buffer hit rate by referencing consecutive memory addresses
- **Coalescing memory requests**



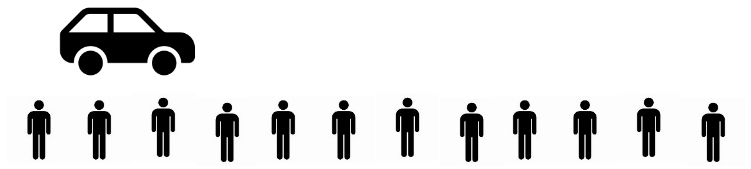
Exploiting **channel-level parallelism**



Exploiting **bank-level parallelism**



Coalescing memory requests



Memory Coalescing to fully utilize global memory bandwidth

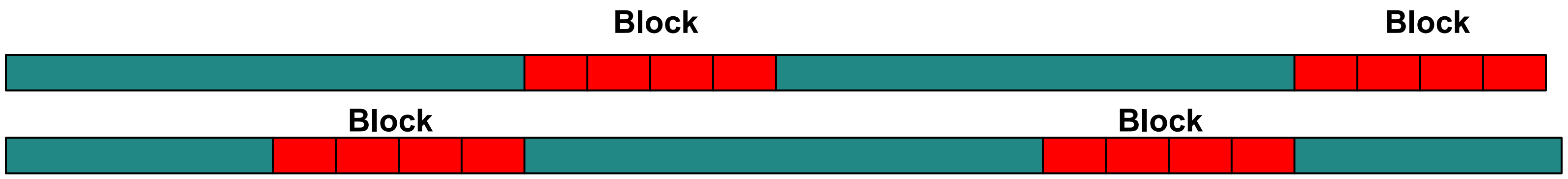
Coalesce: 합체하다

Memory Coalescing

- If an kernel uses data from **consecutive memory addresses**, the DRAMs work close to the peak memory bandwidth!!

- **Off-chip memory is accessed in chunks** (aka. Cache block)
  - **Even if you read only a single word**
  - If you don't use whole chunk, bandwidth is wasted

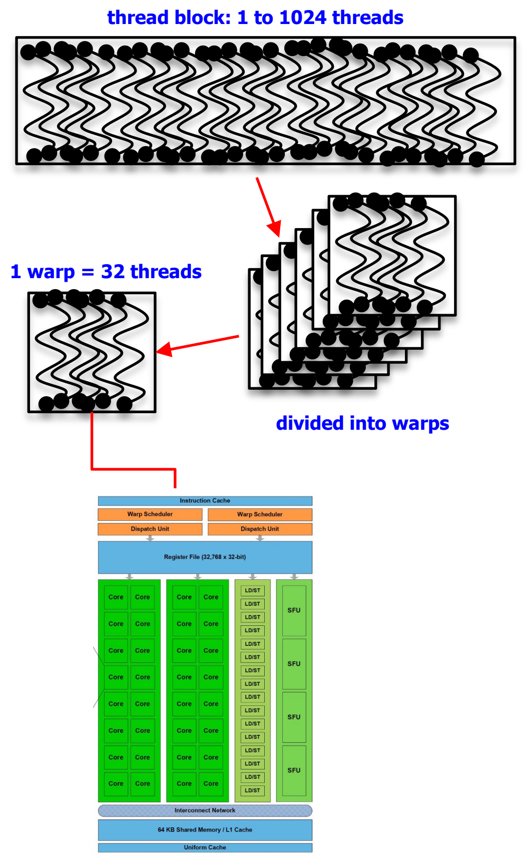
- **Chunks are aligned to multiples of 32/64/128 bytes**
  - Unaligned accesses will cost more !!

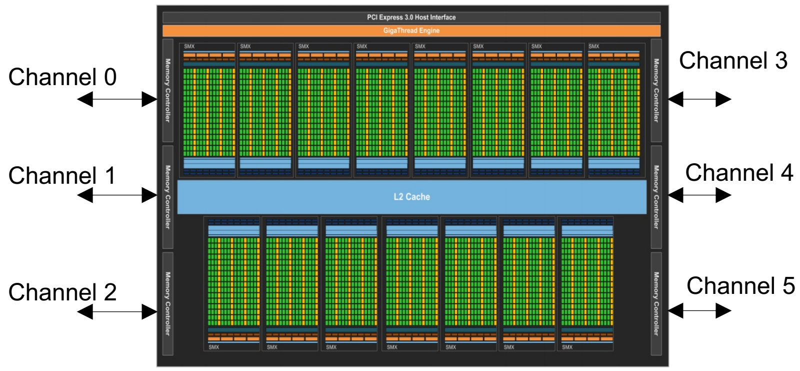


Multi-Bank burst timing, reduced dead time

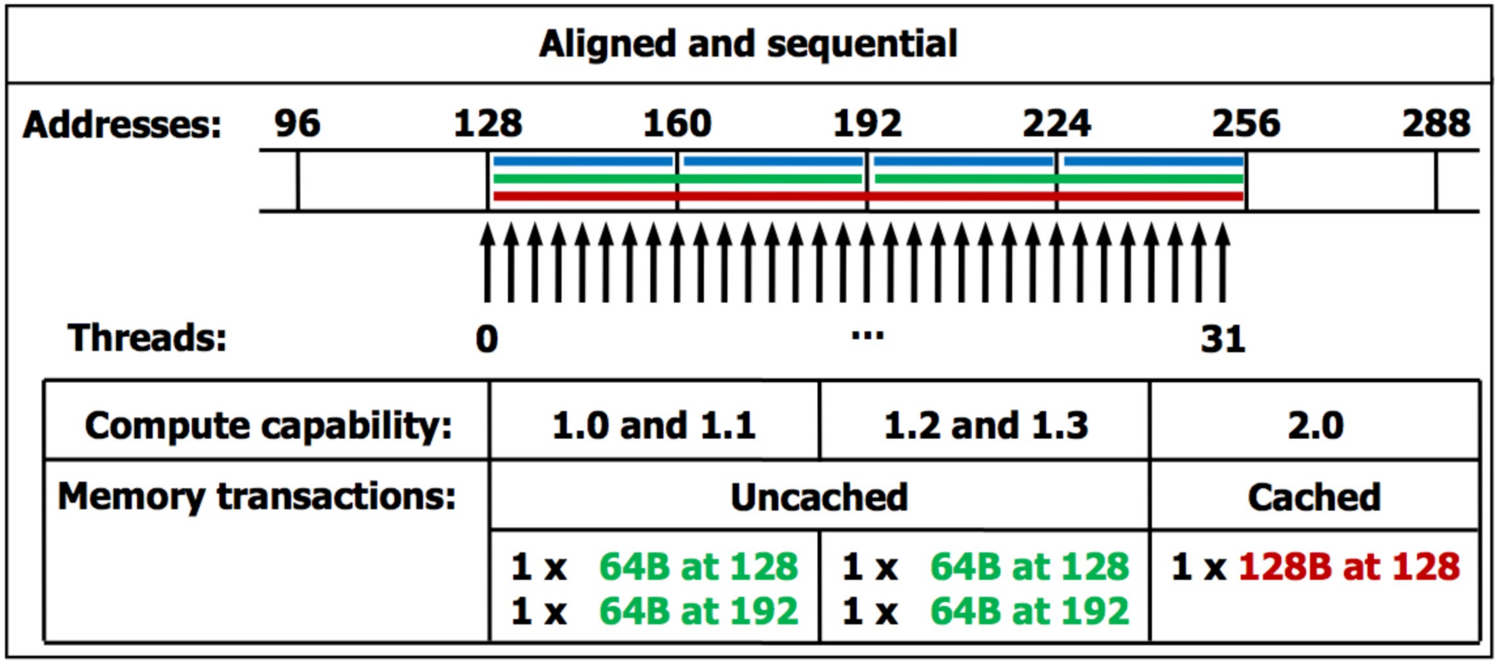
Memory Coalescing (Cont'd)

- Recall, all threads in a warp execute the same instruction
- When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory addresses
  - If so, the hardware ***coalesces*** all memory accesses into **a consolidated access** to consecutive DRAM locations
  - 32 threads x 4B = 128B
    - With Coalescing → 1 memory requests
    - Without Coalescing → 32 memory requests





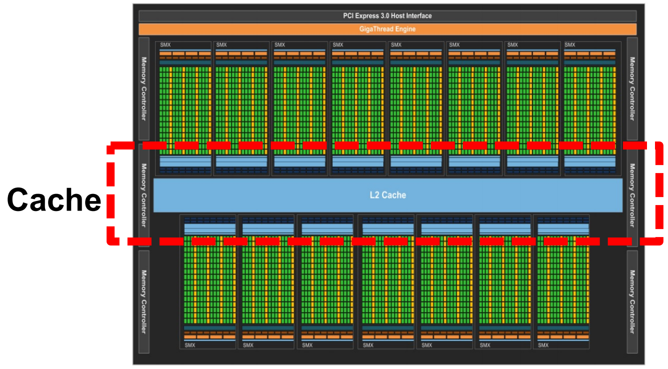
Memory Coalescing (Cont'd)



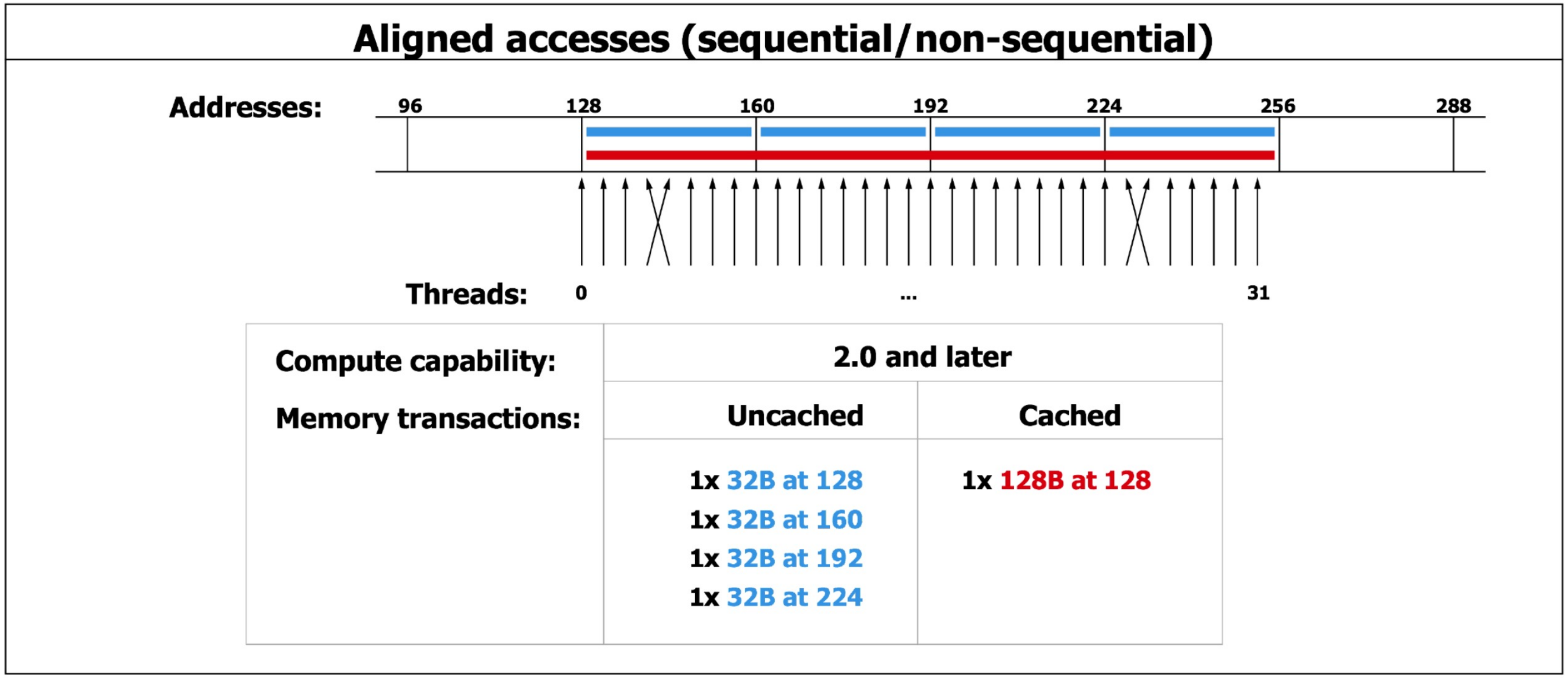
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

From NVIDIA Programming Guide

If thread 0 accesses location 128, thread 1 accesses location 132, … thread 31 accesses location 255, then all these accesses are ***coalesced***, that is: combined into **one single access**



Memory Coalescing (Cont'd)

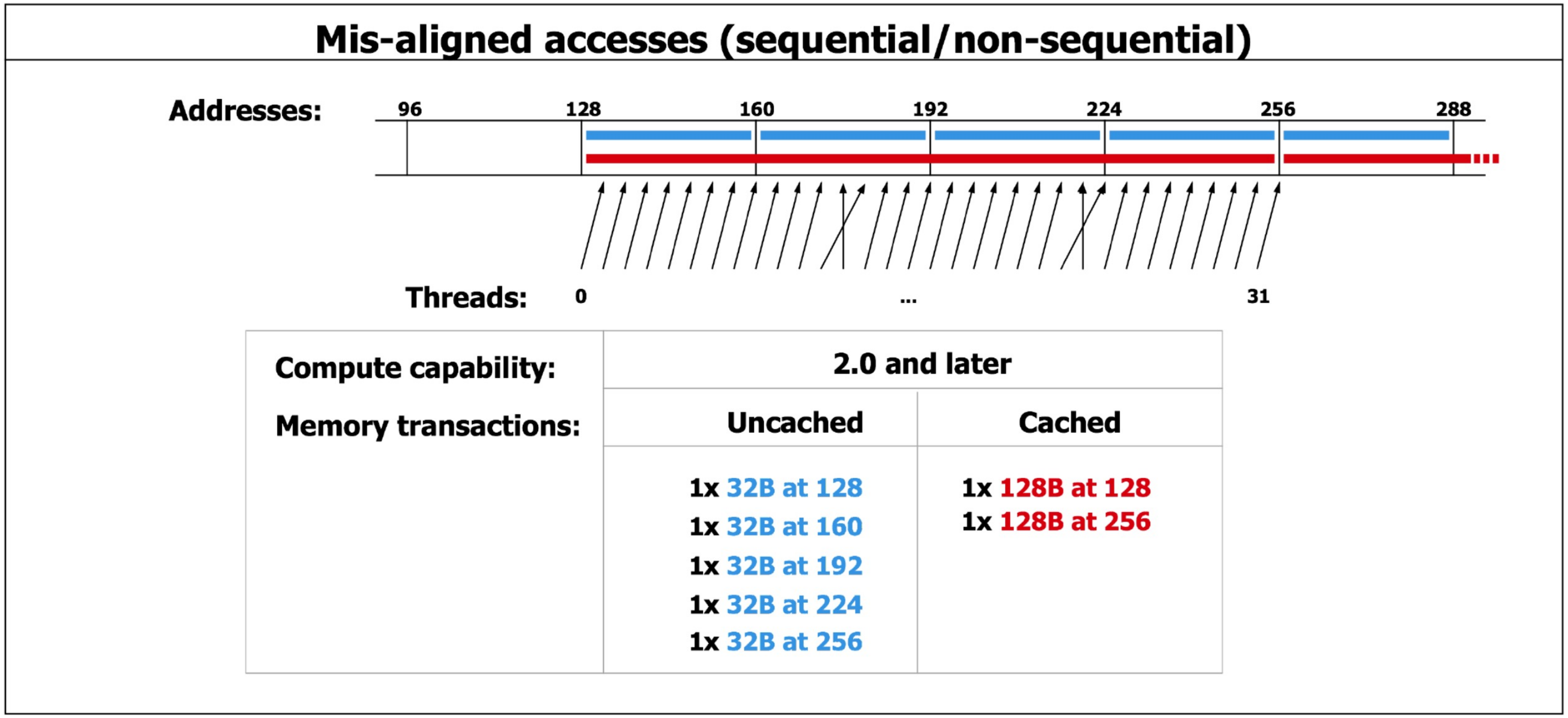


Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

From NVIDIA Programming Guide

If thread 0 accesses location 128, thread 1 accesses location 132, … thread 31 accesses location 255, then all these accesses are ***coalesced***, that is: combined into **one single access**

Memory Coalescing (Cont’d)



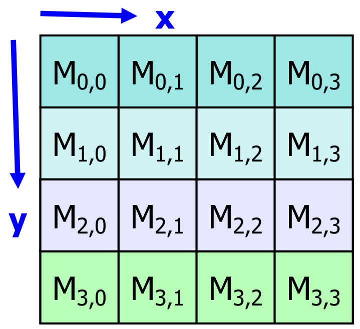
Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

From NVIDIA Programming Guide

If thread 0 accesses location 129, thread 1 accesses location 133, … thread 31 accesses location 256, then all these accesses are ***coalesced***, that is: combined into **two accesses**

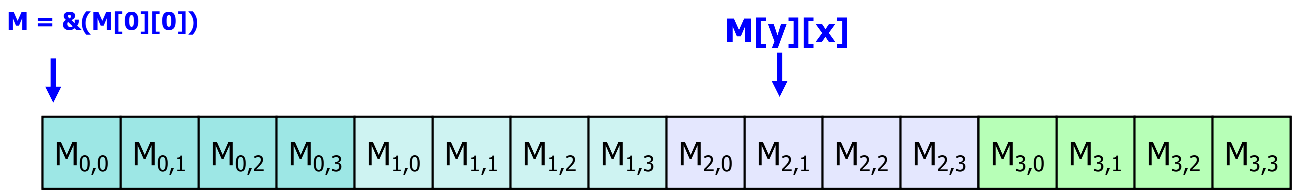
Review: Row-major Matrix Layout in C/C++

- logical layout:



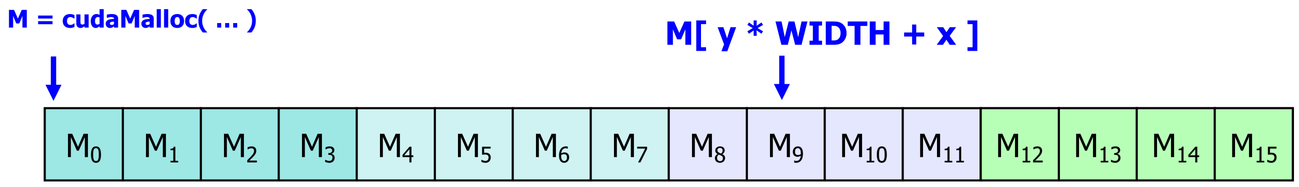
x
y
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

- physical layout: 1D array



M = &(M[0][0])
M[y][x]
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

- re-interpret:



M = cudaMalloc(…)
M[y * WIDTH + x]
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

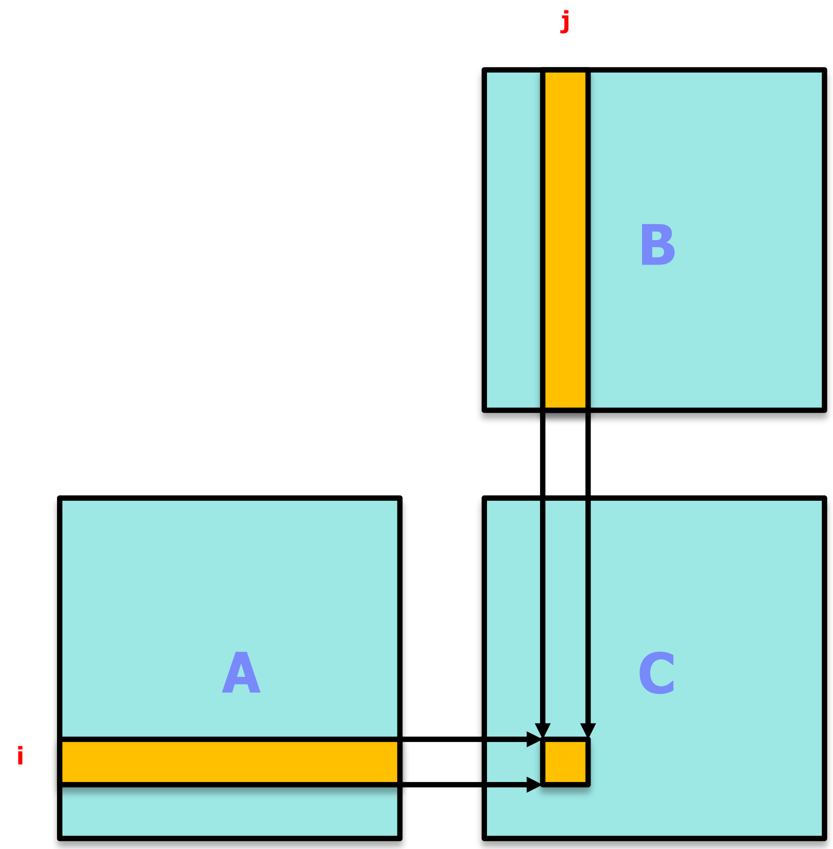
Slide credit : Prof. Baek

Review: CUDA Matrix Multiplication

- One block of threads compute matrix C
  - Each thread computes one element of C

- Each thread
  - loads a row of matrix A
  - loads a column of matrix B
  - Perform one multiply and addition for each pair of A and B elements

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



Slide credit : Prof. Baek

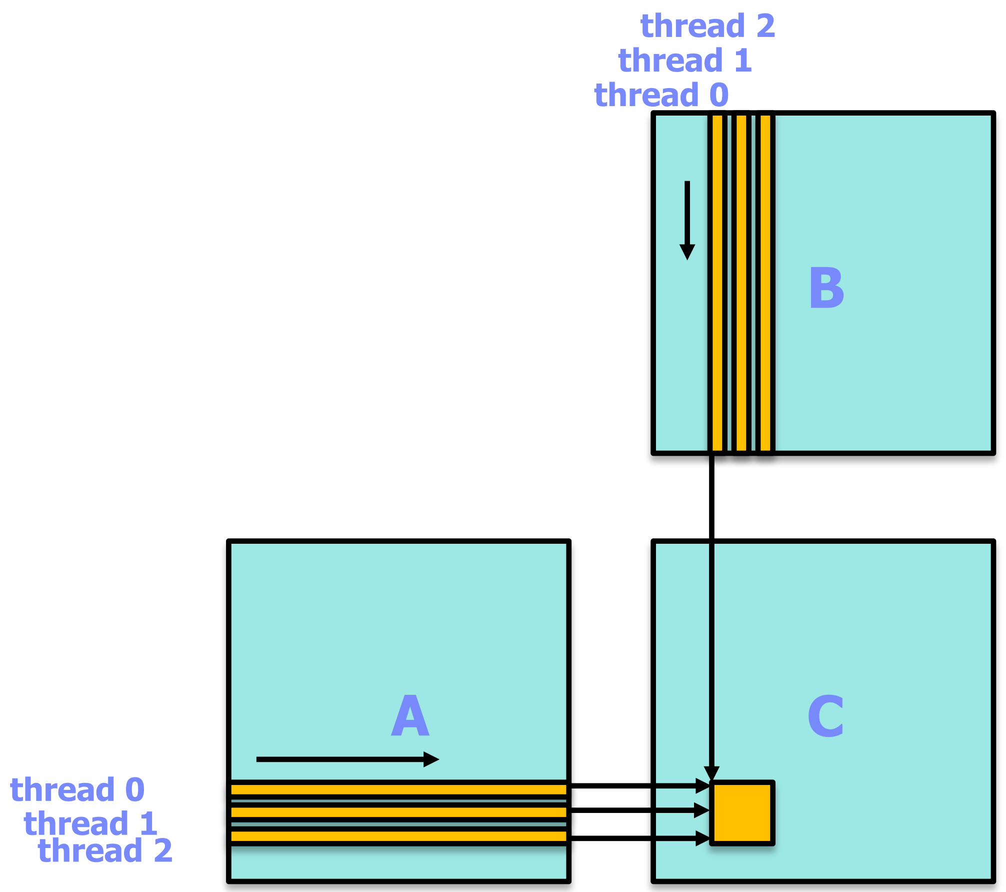
CUDA Matrix Multiplication Kernel

```
// kernel program for the device (GPU): compiled by NVCC
__global__ void addKernel(int* c, const int* a, const int* b, const int WIDTH) {
    int x = threadIdx.x;
    int y = threadIdx.y;
    int i = y * WIDTH + x;      // [y][x] = y * WIDTH + x;
    int sum = 0;
    for (int k = 0; k < WIDTH; ++k) {
            sum += a[y * WIDTH + k] * b[k * WIDTH + x];
    }
    c[i] = sum;
}
```

Slide credit : Prof. Baek

DRAM Access Patterns

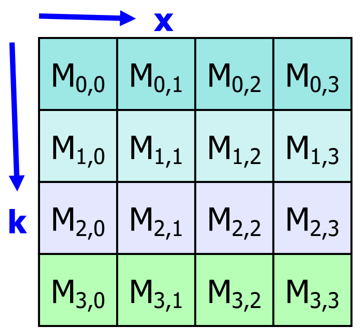
- Multiple threads are working in a warp
- for A matrix,
  - a[y * WIDTH + k]
- for B matrix,
  - b[k * WIDTH + x]

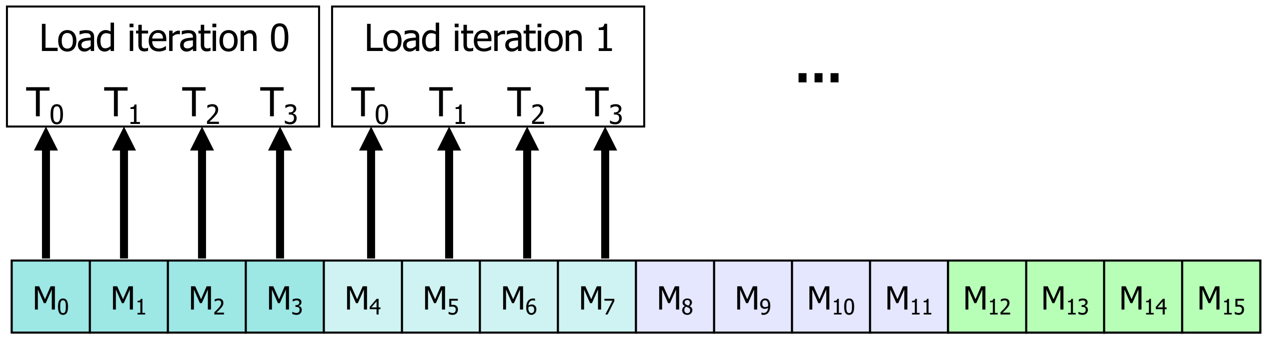


Slide credit : Prof. Baek

B accesses are coalesced

- b[k * WIDTH + x]
- k = 0, 1, 2, …
  - thread 0: b[k * WIDTH + **0**]
  - thread 1: b[k * WIDTH + **1**]
  - thread 2: b[k * WIDTH + **2**]
  - thread 3: b[k * WIDTH + **3**]

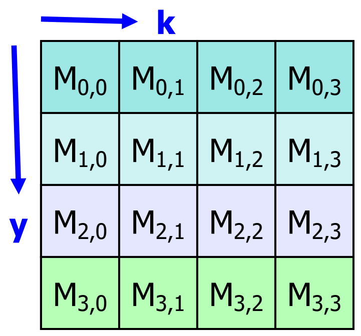

x
k
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

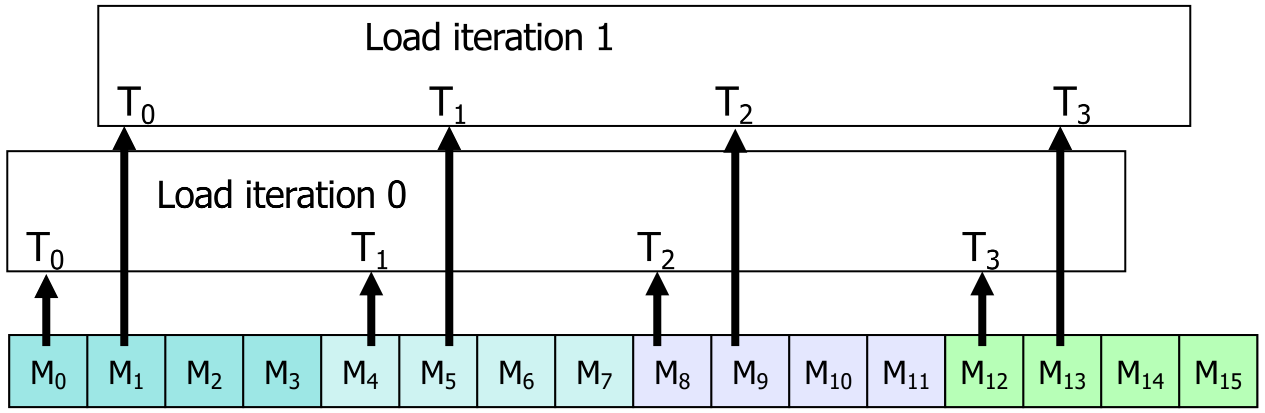

Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Slide credit : Prof. Baek

A accesses are not coalesced

- a[y * WIDTH + k]
- k = 0, 1, 2, …
  - thread 0: a[**0** * WIDTH + k]
  - thread 1: a[**1** * WIDTH + k]
  - thread 2: a[**2** * WIDTH + k]
  - thread 3: a[**3** * WIDTH + k]

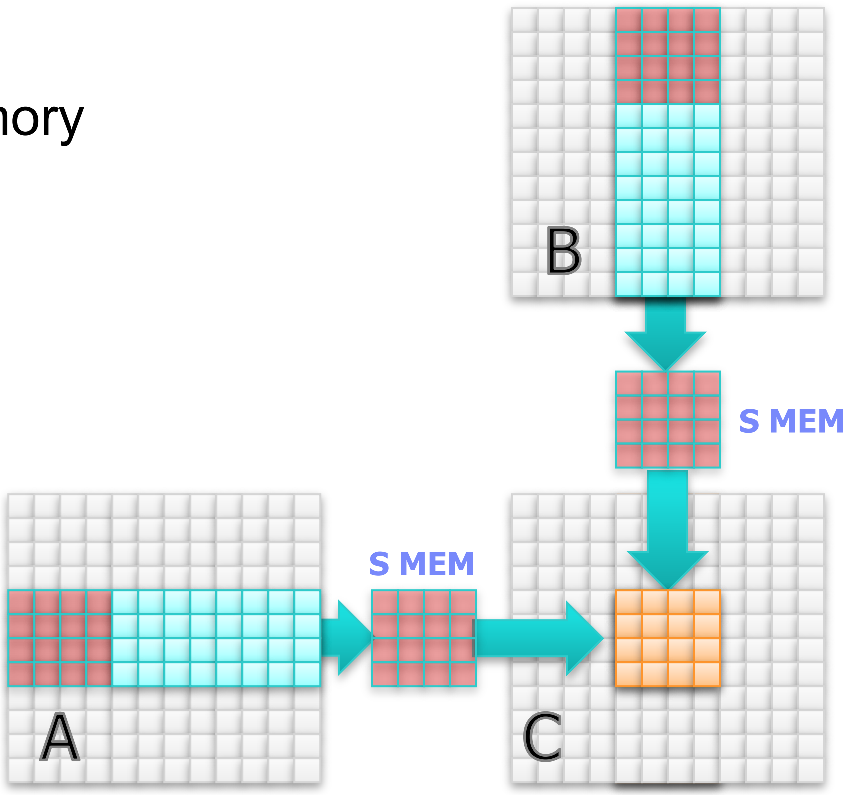

k
y
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$


Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Slide credit : Prof. Baek

Shared Memory Matrix Multiplication

- make a thread block as a tile
- partition the global memory into tiles
- then, load it to the shared memory



Slide credit : Prof. Baek

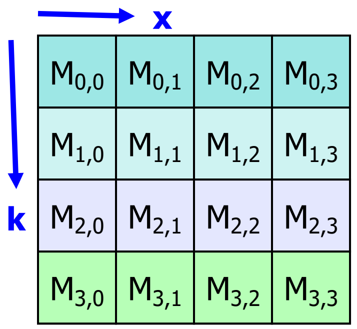
Tiled Matrix Multiplication Kernel

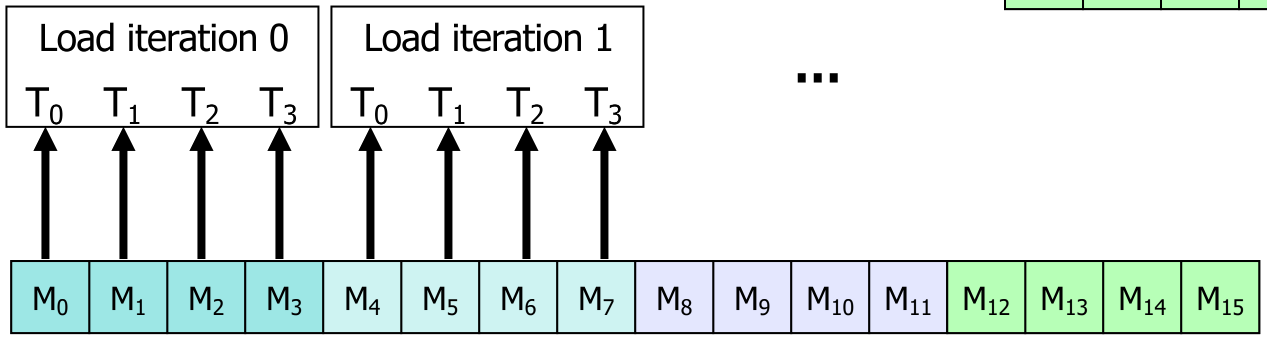
```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
    __shared__ float s_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B[TILE_WIDTH][TILE_WIDTH];
    int by = blockIdx.y; int bx = blockIdx.x;
    int ty = threadIdx.y; int tx = threadIdx.x;
    int gy = by * TILE_WIDTH + ty; // global y index
    int gx = bx * TILE_WIDTH + tx; // global x index
    float sum = 0.0F;
    for (register int m = 0; m < width / TILE_WIDTH; ++m) {
        // read into the shared memory blocks
        s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + tx)];
        s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
        __syncthreads();
        // use the shared memory blocks to get the partial sum
        for (register int k = 0; k < TILE_WIDTH; ++k) {
            sum += s_A[ty][k] * s_B[k][tx];
        }
        __syncthreads();
    }
    g_C[gy * width + gx] = sum;
}
```

Slide credit : Prof. Baek

B accesses are coalesced

- gx = bx * TILE_WIDTH + **tx**
- s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
- tx = 0, 1, 2, …
  - thread 0: b[… + **0**]
  - thread 1: b[… + **1**]
  - thread 2: b[… + **2**]
  - thread 3: b[… + **3**]

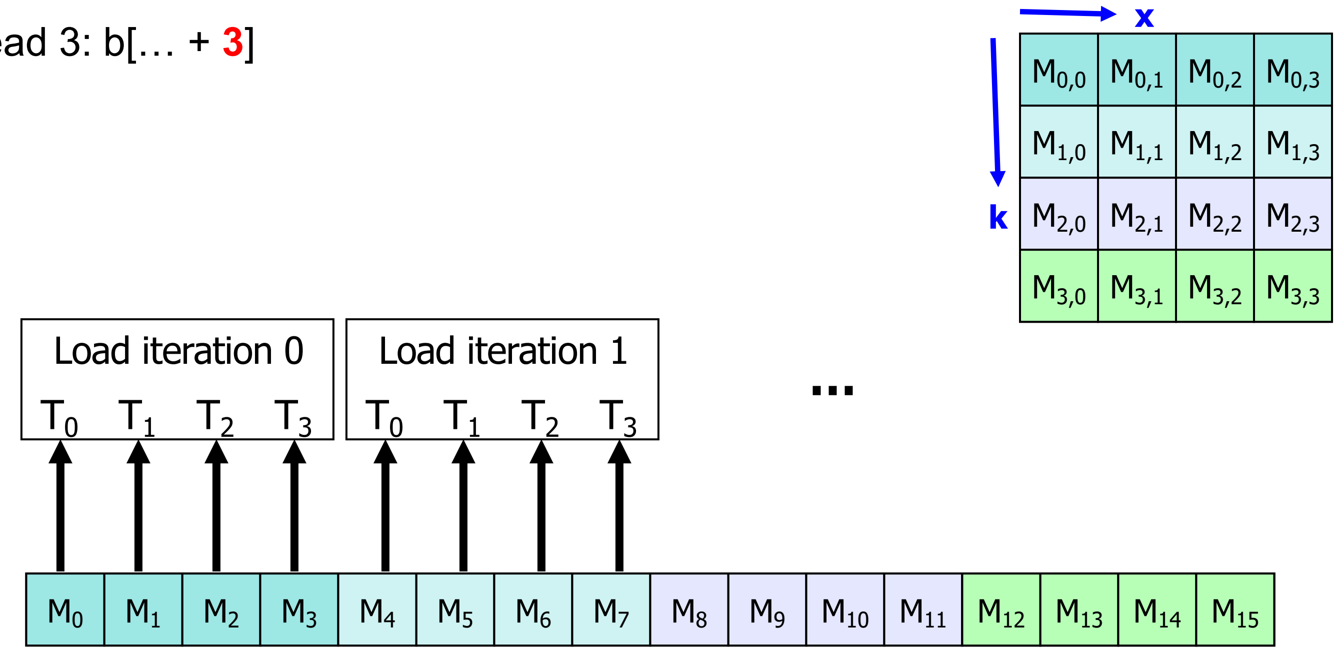

x
k
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$


Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Slide credit : Prof. Baek

A accesses are coalesced

- s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + **tx**)];
- tx = 0, 1, 2, …
  - thread 0: b[… + **0**]
  - thread 1: b[… + **1**]
  - thread 2: b[… + **2**]
  - thread 3: b[… + **3**]


x
k
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Slide credit : Prof. Baek

Next?

- Memory Optimizations
  - More about Global Memory
  - Memory Coalescing to fully utilize global memory bandwidth
  - **Reducing Bank Conflict** to fully utilize shared memory bandwidth

- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware

- **Considering Occupancy**
  - Dynamic Partitioning of Resources

- **Considering Thread Granularity**