Parallel Programming

Memory architecture in CUDA (Part 2)

Phạm Trọng Nghĩa
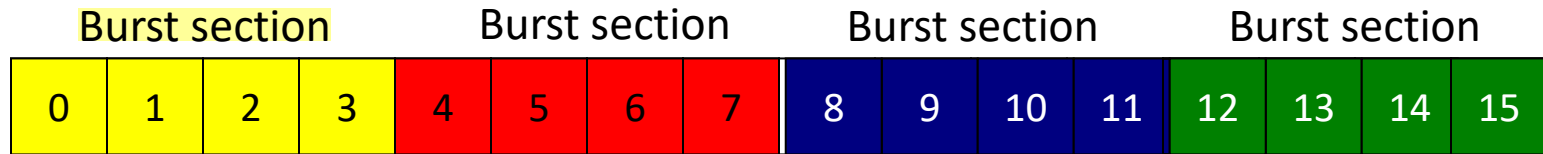
ptnghia@fit.hcmus.edu.vn

Review: previous lecture
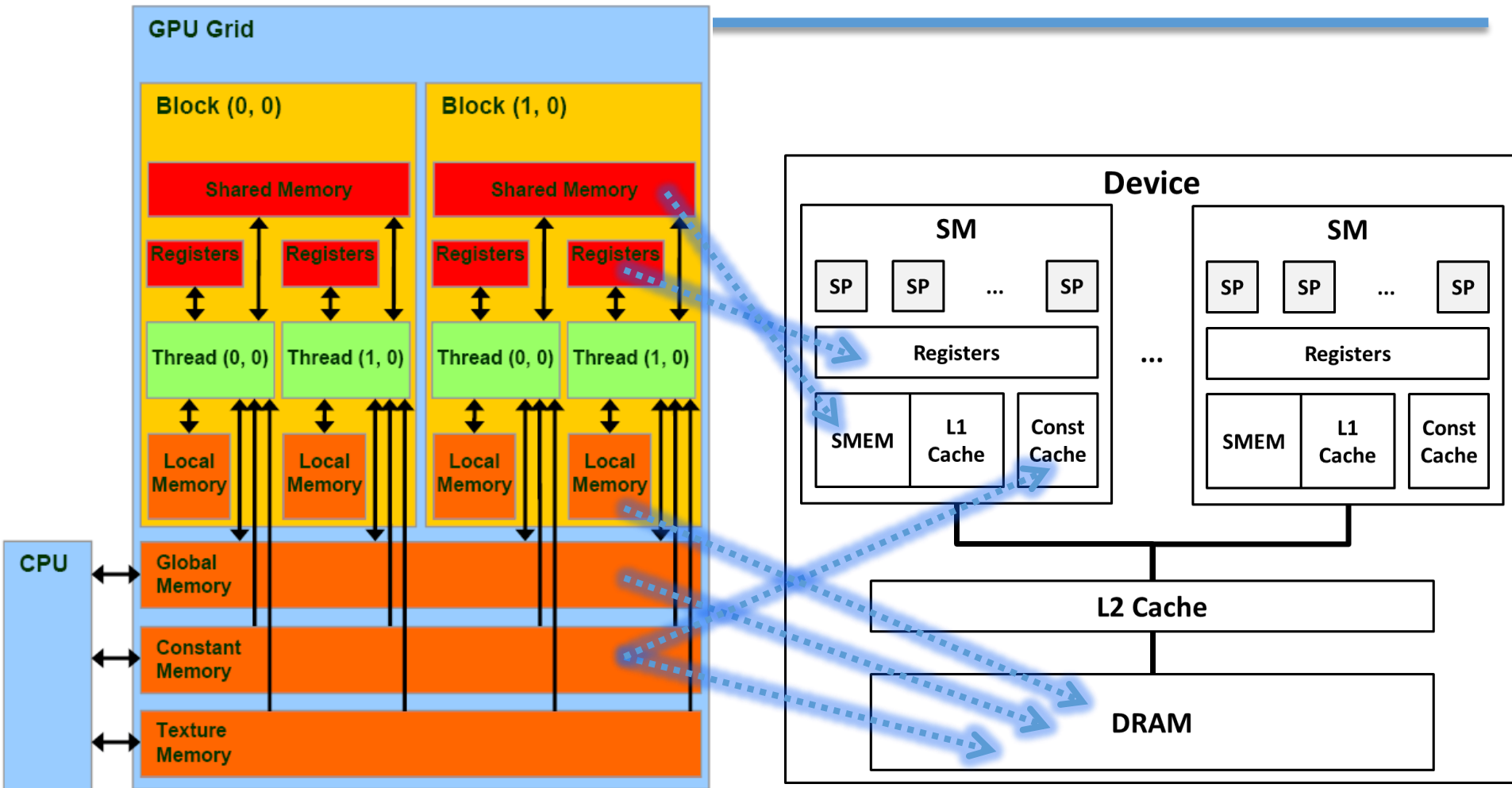
- *Utilize high speed memories residing in SMs to store data, reduce DRAM accesses*
- Price: it can decrease occupancy (e.g., if SM has 48 KB SMEM and block consumes 40 KB SMEM then SM can only contain one block)

Today

- Access GMEM efficiently <small>global</small>
- Access SMEM efficiently <small>shared</small>

DRAM Burst – A System View

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

- 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

Cache line & sector

Moving data between L1, L2, DRAM

- Memory access granularity = **32 Bytes** = **1 sector**

(32B for newer cards. 32B or 128B for older card, depending on architecture, access type, caching / non-caching options)

- A **cache line** is **128 Bytes**, made of **4 sectors**.

- Cache "management" granularity = 1 cache line


```
128-Byte alignment
↓
| Sector 0 | Sector 1 | Sector 2 | Sector 3 |
        128 Byte cache line
```
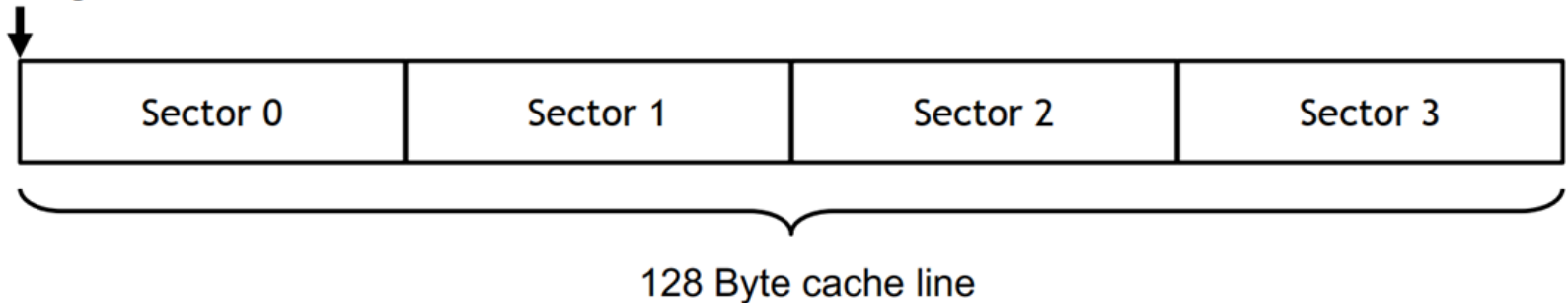
Khi đọc từ memory sẽ đọc nguyên 1 cache line hoặc sector

Read GMEM (older cards)

- Caching (default mode)

  - **Use L1 cache**

  - If cannot find data in **L1** then go to **L2**, if cannot find data in **L2** then go to **DRAM**

  - Load granularity is **128-byte line**

- Non-caching

  - At compile time, pass this flag:

    `-Xptxas -dlcm=cg`

  - **Not use L1 cache**

  - Go straight to L2, if cannot find data in L2 then go to DRAM

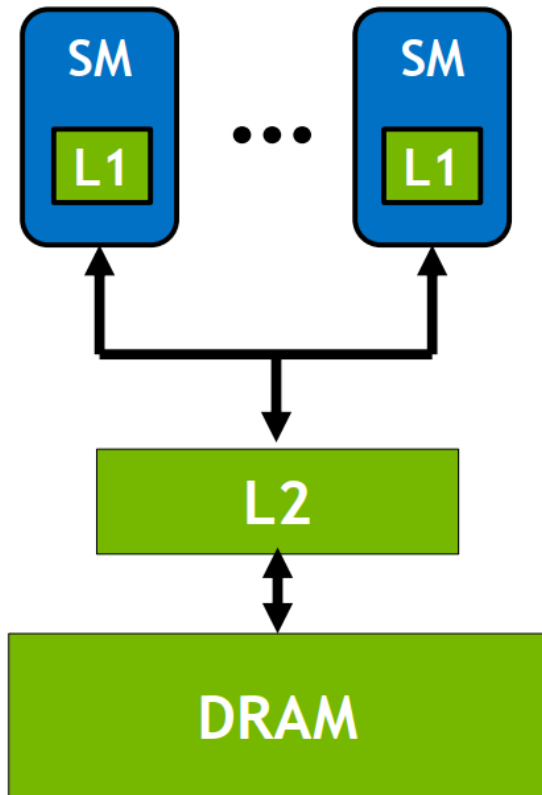 - Load granularity is **32-byte line**

6
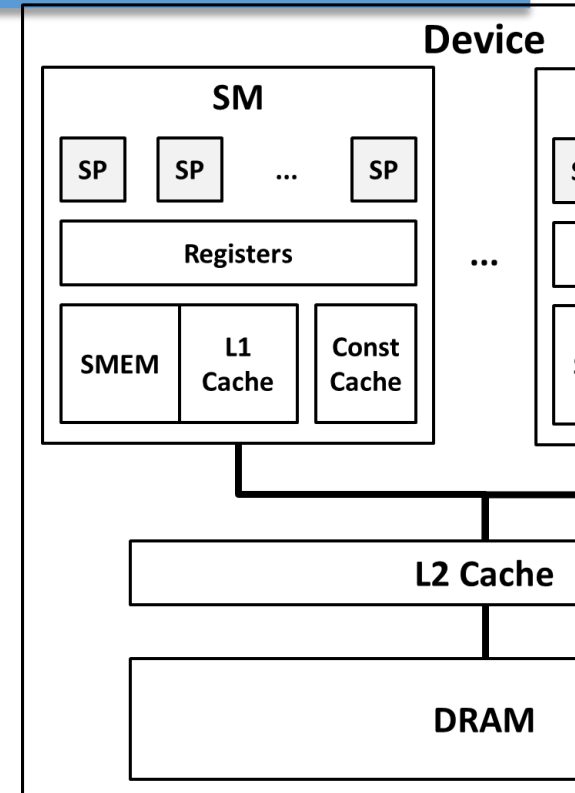
Memory Reads

DRAM
Getting data from Global Memory

SM   SM
L1   L1

L2
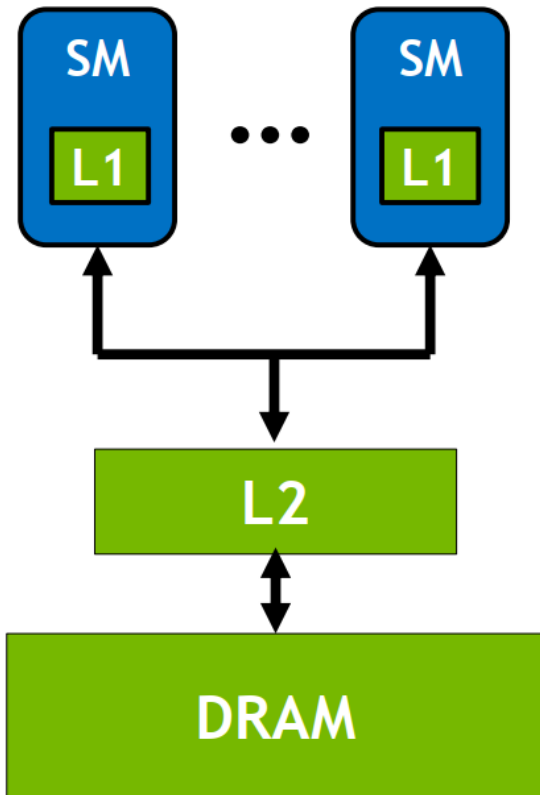
DRAM

- Checking if the data is in **L1** (if not, check **L2**)
- Checking if the data is in **L2** (if not, get in **DRAM**)
- Unit of data moved: **Sectors**

Memory Writes

Getting data from Global Memory

- **L1 will cache writes** (new cards)
- **L1 is <mark>write-through</mark>**: Write to L1 AND L2. ghi xuống L1 và L2
- **L2 is write back**: Will flush data to DRAM only when needed.

- If threads in a warp write to the same address
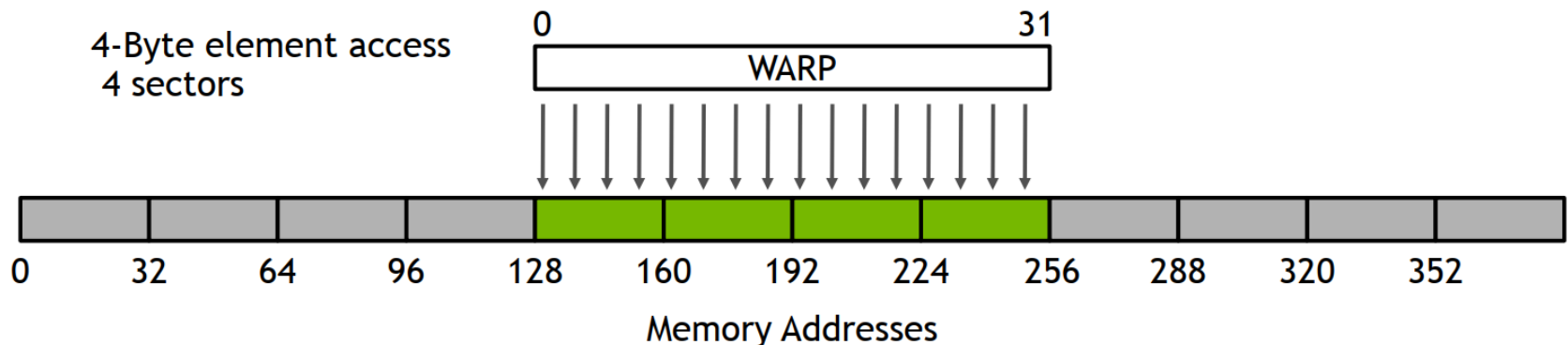 - One thread will win
  - But we don't know which one

Coalesce global memory access

- **32 threads (1 warp) access memory together**
- Can coalesce into a **single reference**, if address within a 128-byte block

  hợp nhất

  - Instead of use 32 memory accesses with 4 byte each
  - Use 1 memory access with 128 bytes wide

- Ideal: 1 warp -> 128 bytes of consecutive memory
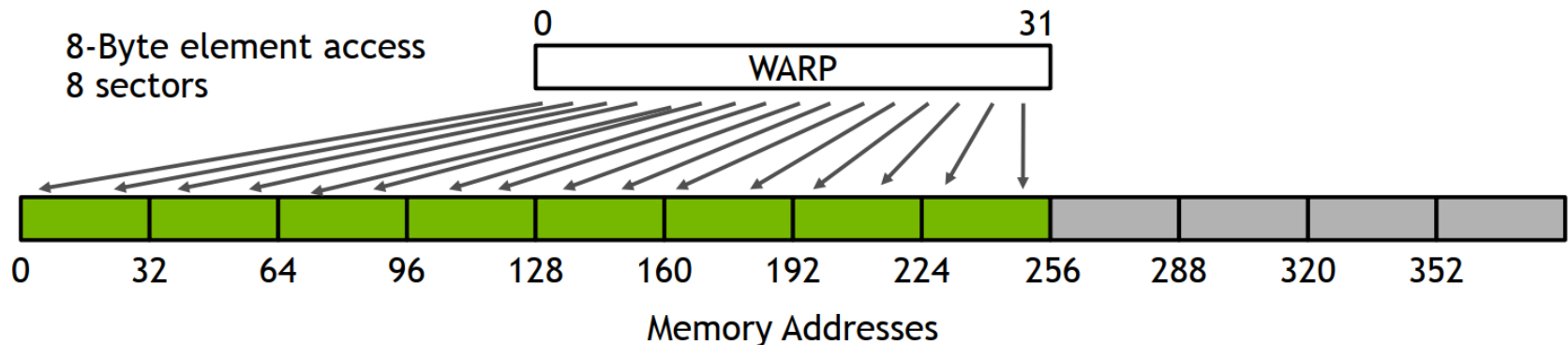
  - Aligned to 128-byte boundary

Global memory Load

- Warp requests 32 aligned, permuted 4-byte words

- Warp needs 128 bytes, 4 sectors

- Load: 4 sector.

- Bus utilization: **100%**

- int c = a[idx];

4-Byte element access
4 sectors

0   31
WARP

Memory Addresses
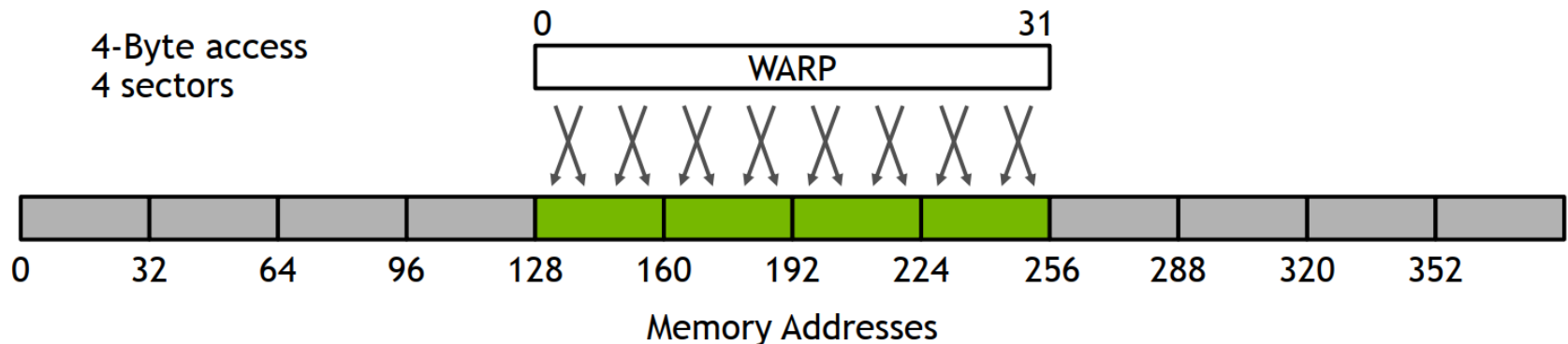0   32   64   96   128   160   192   224   256   288   320   352

Global memory Load

- Warp requests 32 aligned, permuted 8-byte words

- Warp needs 256 bytes, 8 sectors

- Load: 8 sectors

- Bus utilization: **100%**

- double c = a[idx];

8-Byte element access
8 sectors

0   31

WARP

Memory Addresses

0   32   64   96   128   160   192   224   256   288   320   352
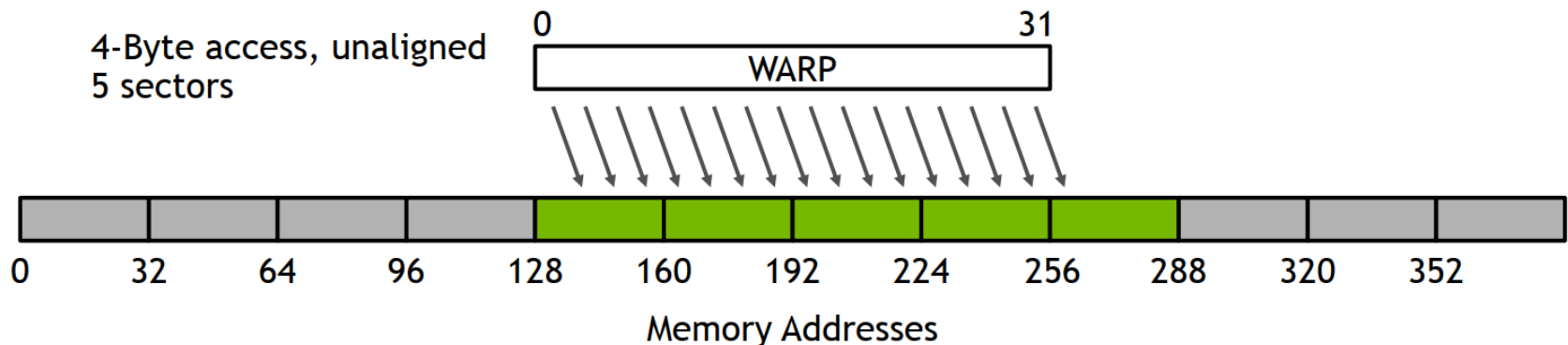
Global memory Load

- Warp requests 32 aligned, consecutive 4-byte words

- Warp needs 128 bytes, 4 sectors

- Load: 4 sector

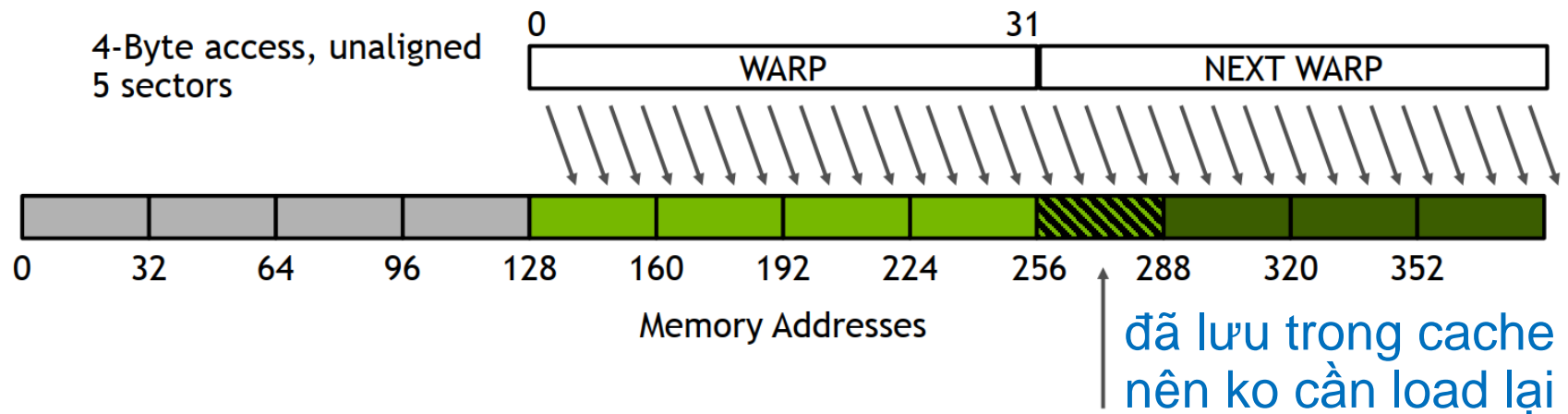- Bus utilization: **100%**

- int c = a[rand()%warpSize];

4-Byte access
4 sectors

0   31

WARP

0   32   64   96   128   160   192   224   256   288   320   352

Memory Addresses

Global memory Load

- Warp requests 32 <mark>misaligned</mark>, consecutive 4-byte words

- Warp needs 128 bytes, 4 sectors

- Load: 5 sector

- Bus utilization: **80%** = 4 need / 5 actual

- int c = a[idx+2];

4-Byte access, unaligned
5 sectors

0   31

WARP

Memory Addresses

0   32   64   96   128   160   192   224   256   288   320   352
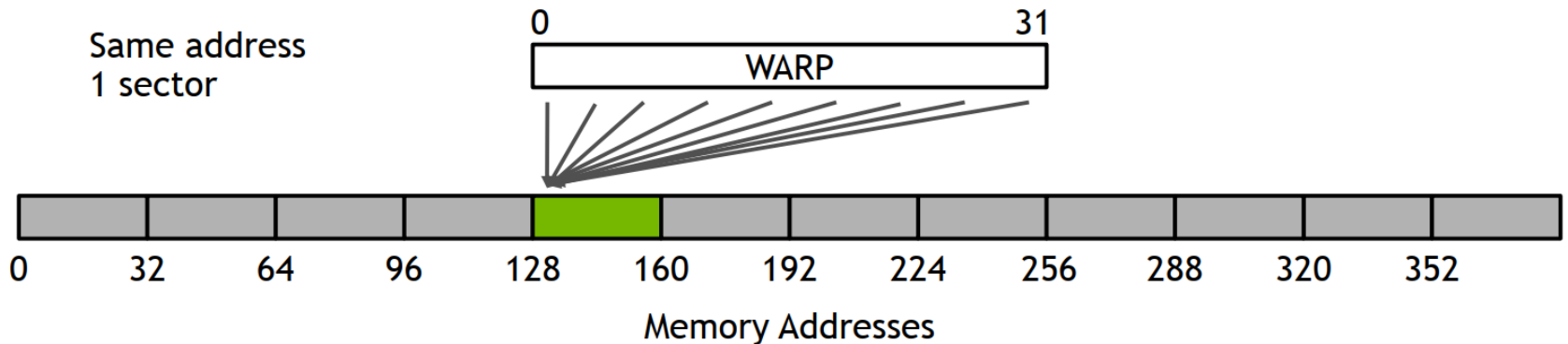
Global memory Load

- Warp requests 32 misaligned, consecutive 4-byte words

- Warp needs 128 bytes, 4 sectors

- Load: 5 sector

- Bus utilization: **80%**

- int c = a[idx+2];

4-Byte access, unaligned 5 sectors

WARP   NEXT WARP

0   31

0   32   64   96   128   160   192   224   256   288   320   352

Memory Addresses

đã lưu trong cache nên ko cần load lại

With >1 warp per block, this sector might be found in L1 or L2

Global memory Load

- All threads in a warp request the same 4-byte word

- Warp needs 4 bytes

- Load: 1 sector, 32 bytes

- Bus utilization: **3.125%**   12.5% = 4 need / 32 actual

- $int\ c = a[40];$

Global memory Load

- Warp requests 32 scattered 4-byte words

- Warp needs 128 bytes, 4 sector

- Load: 32 sectors

- Bus utilization: **3.125%** 12.5% = 4 need / 32 actual

- int c = a[rand()];
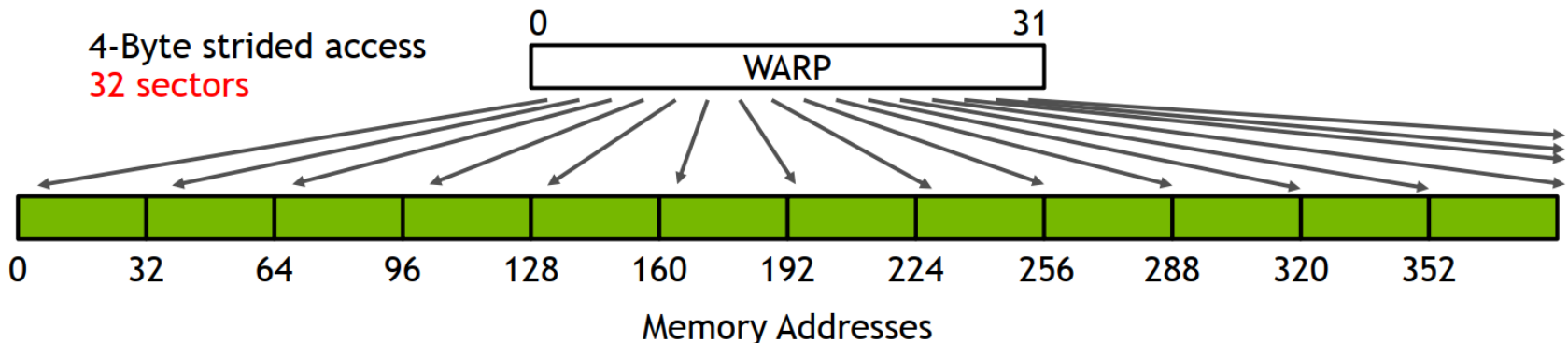
128 bytes requested, 1024 bytes transferred!
Using only a few bytes per sector. Wasting lots of BW!
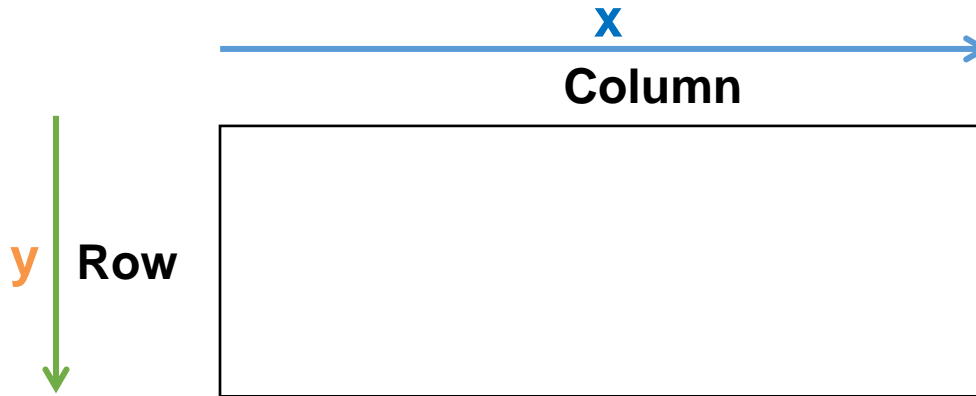
Access GMEM efficiently

- Strive for perfect coalescing
    - (Align starting address - may require padding)
    - A warp should access within a contiguous region
- Have enough concurrent accesses to saturate the bus
    - Process several elements per thread
        - Multiple loads get pipelined
        - Indexing calculations can often be reused
    - Launch enough threads to maximize throughput
        - Latency is hidden by switching threads (warps)
- Use all the caches!

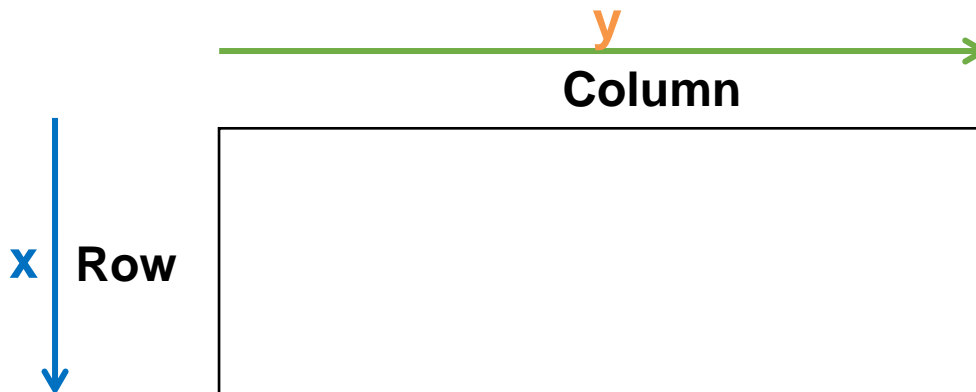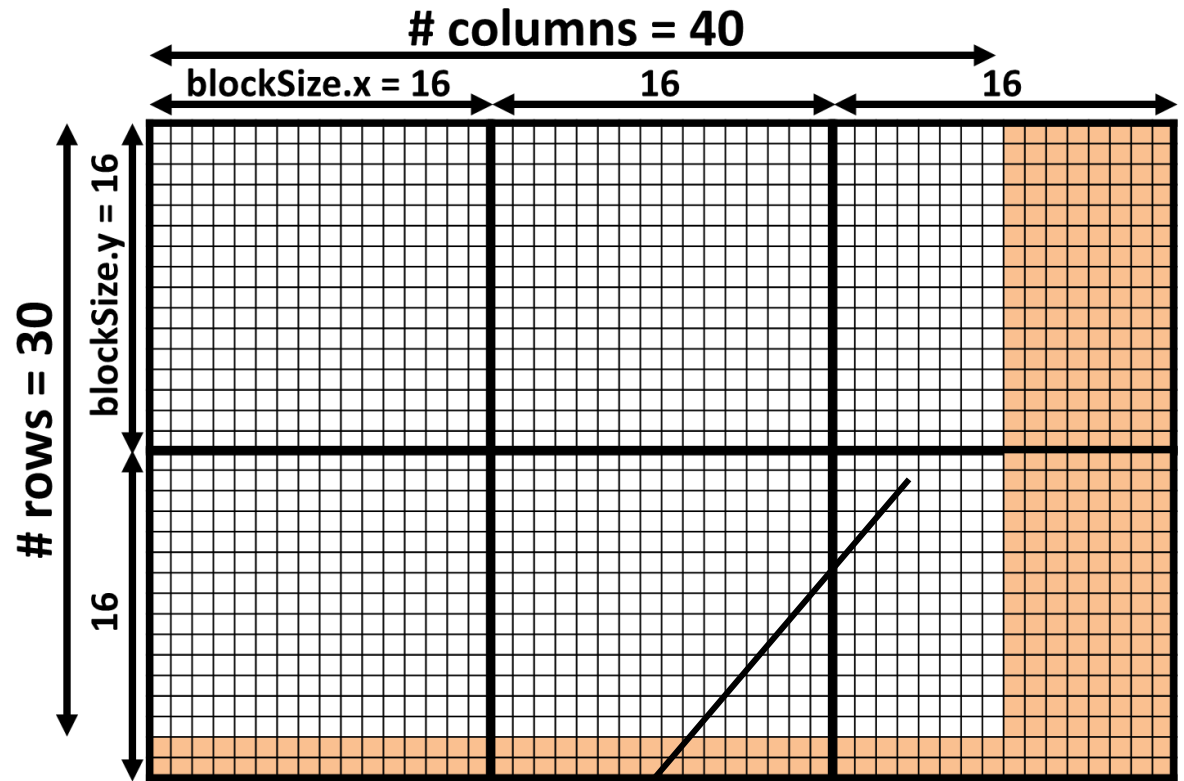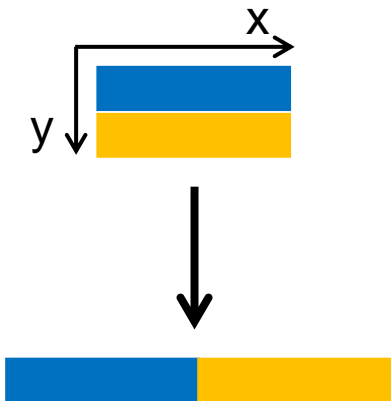# Example 1: why is kernel 3 faster than kernel 2?
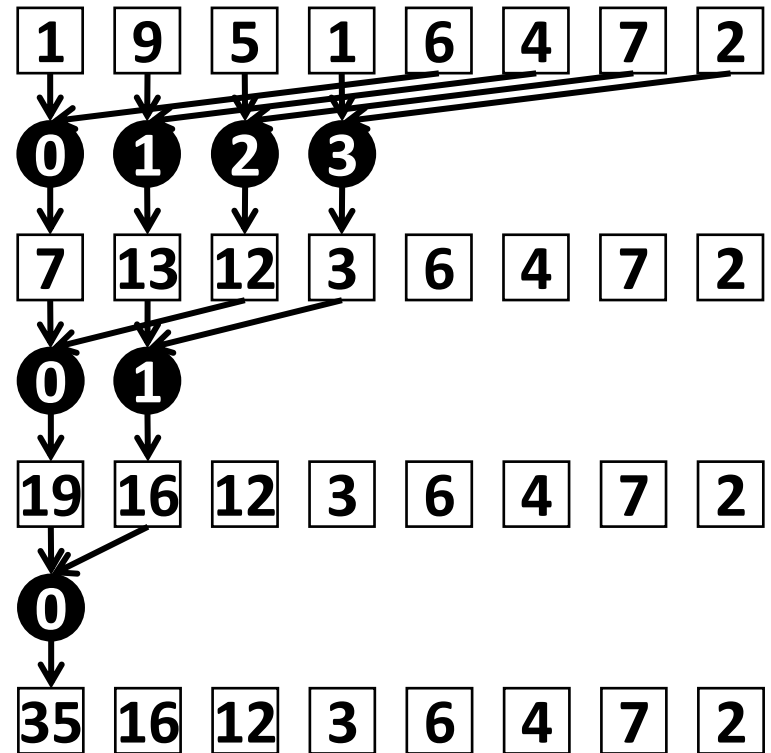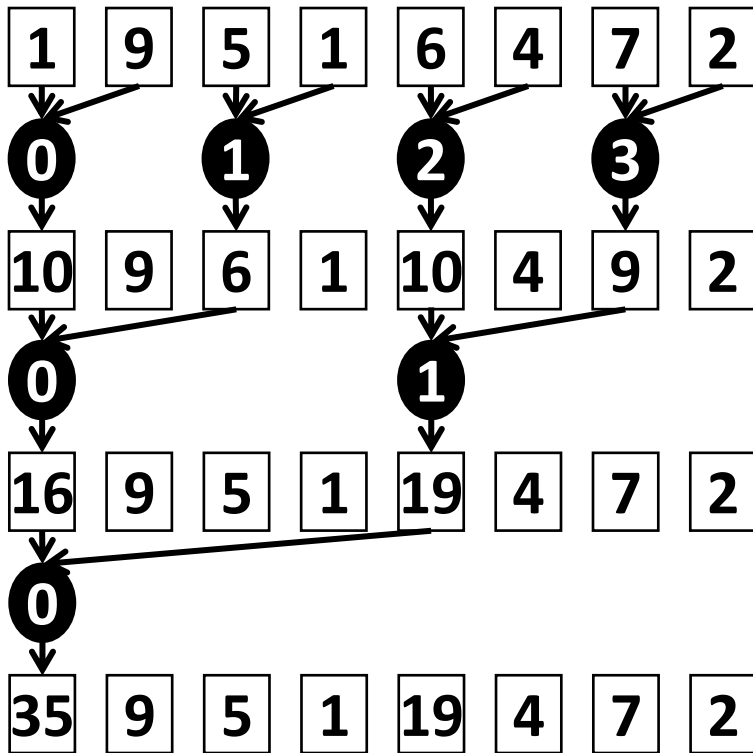
Example 2: x and y dimension in 2D

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

x
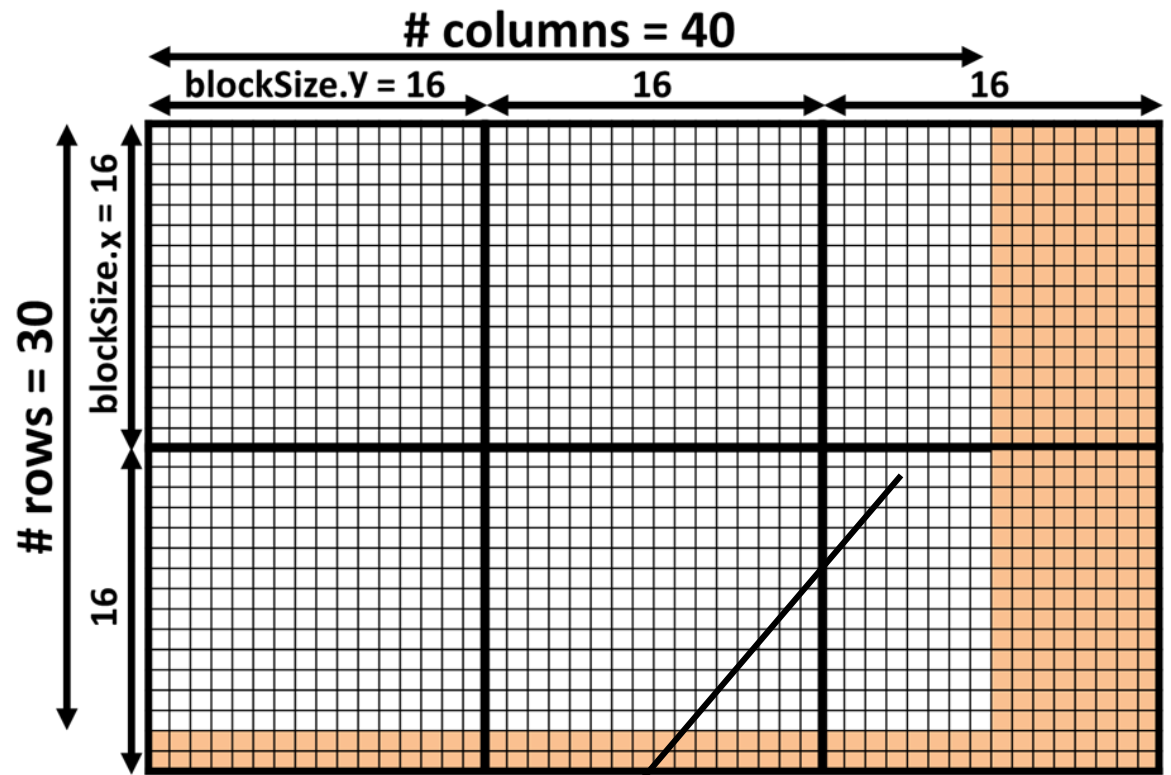**Column**

y | **Row**

What will happen if we reverse x and y (reverse `row` and `col` formula)?

y
Column

x | **Row**

19

columns = 40

blockSize.x = 16   16   16

blockSize.y = 16

16

rows = 30

x

y

blockIdx.x = 2   threadIdx.x = 3
blockIdx.y = 1   threadIdx.y = 1

r = blockIdx.y*blockDim.y + threadIdx.y
  = 1*16 + 1
  = 17
c = blockIdx.x*blockDim.x + threadIdx.x
  = 2*16 + 3
  = 35

20

columns = 40

blockSize.Y = 16   16   16

rows = 30

blockSize.x = 16

16

x

y
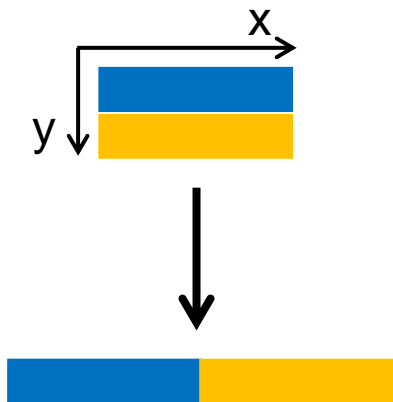
blockIdx.x = 1   threadIdx.x = 1
blockIdx.y = 2   threadIdx.y = 3

r = blockIdx.x*blockDim.x + threadIdx.x
  = 1*16 + 1
  = 17
c = blockIdx.y*blockDim.y + threadIdx.y
  = 2*16 + 3
  = 35

Access GMEM efficiently

- Accessing GMEM is most efficient when: threads in the same warp access consecutive elements in GMEM and the first element's address is aligned

- This is only true when element size is native: 1, 2, 4, 8, 16 bytes

- If element size is non-native (e.g., struct defined by programmers) then compiler will convert instruction accessing non-native size to instructions accessing native sizes (as usual, they must be aligned)

    The consequence is ...

```
struct Point
{
    float x;
    float y;
    float z;
};
...
Point *d_data;
cudaMalloc(&d_data, ...);

...
__global__ void kernel(Point *d_data, ...)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    Point p = d_data[i];
    ...
}
```
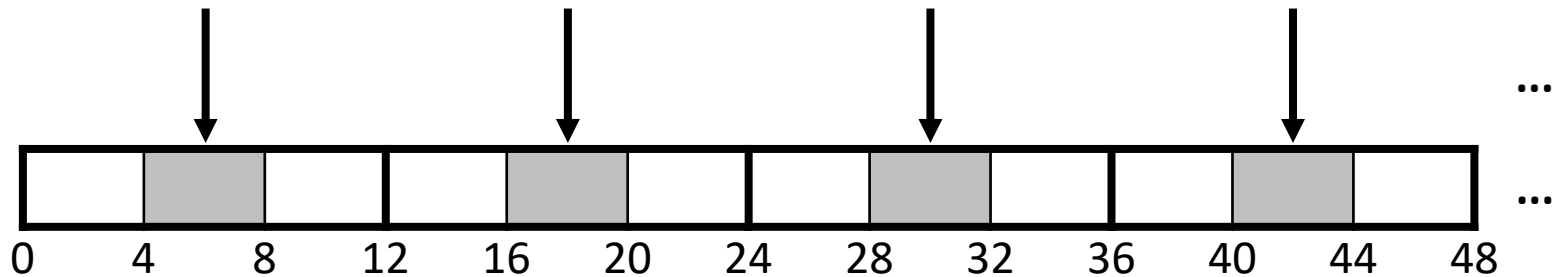
Access **12** bytes element → will be converted to 3 instructions accessing 4 bytes

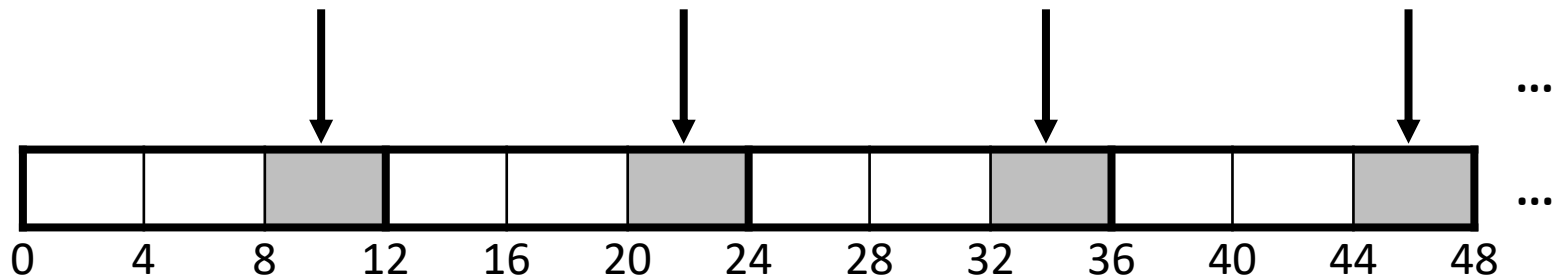Convert to 2 instructions: one accessing 8 bytes, one accessing 4 bytes?

Warp does 1ˢᵗ instruction accessing 4 bytes

0	4	8	12	16	20	24	28	32	36	40	44	48

...
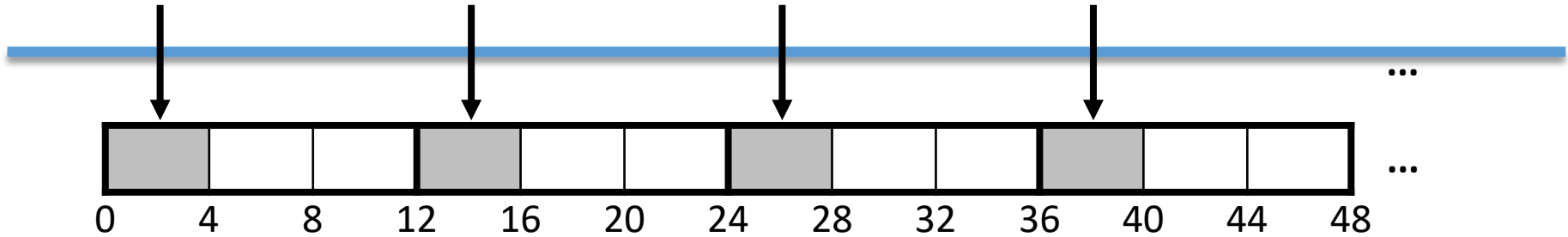
Warp does 2ⁿᵈ instruction accessing 4 bytes

0	4	8	12	16	20	24	28	32	36	40	44	48

...

Warp does 3ʳᵈ instruction accessing 4 bytes

0	4	8	12	16	20	24	28	32	36	40	44	48

...

One solution:
array of structs → struct of arrays

```
struct SoA
{
    float *xArr;
    float *yArr;
    float *zArr;
};
...
SoA d_data;
cudaMalloc(&d_data.xArr, ...);
cudaMalloc(&d_data.yArr, ...);
cudaMalloc(&d_data.zArr, ...);
...
__global__ void kernel(SoA d_data, ...)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = d_data.xArr[i];
    float y = d_data.yArr[i];
    float z = d_data.zArr[i];
    ...
}
```

Today

- Access GMEM efficiently

- Access SMEM efficiently

Example: matrix transpose

- Input: a matrix `iMatrix`

  Output: the transpose matrix `oMatrix`

  `oMatrix[r][c] = iMatrix[c][r]`

- Simplify:

  - Square matrix `w×w`

  - Square block 32×32, and `w` is a multiple of 32

Kernel 1

```
__global__ void transpose1(int *iMatrix, int *oMatrix, int w)
{
    int r = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.x * blockDim.x + threadIdx.x;

    oMatrix[r * w + c] = iMatrix[          ];
}
```
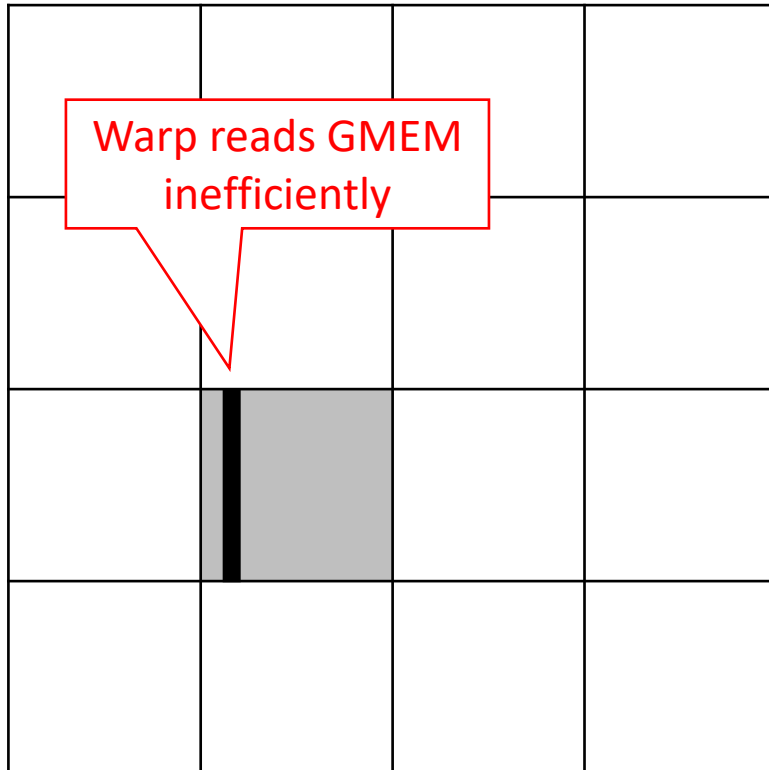
Kernel 1

```
__global__ void transpose1(int *iMatrix, int *oMatrix, int w)
{
    int r = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.x * blockDim.x + threadIdx.x;

    oMatrix[r * w + c] = iMatrix[c * w + r];
}
```
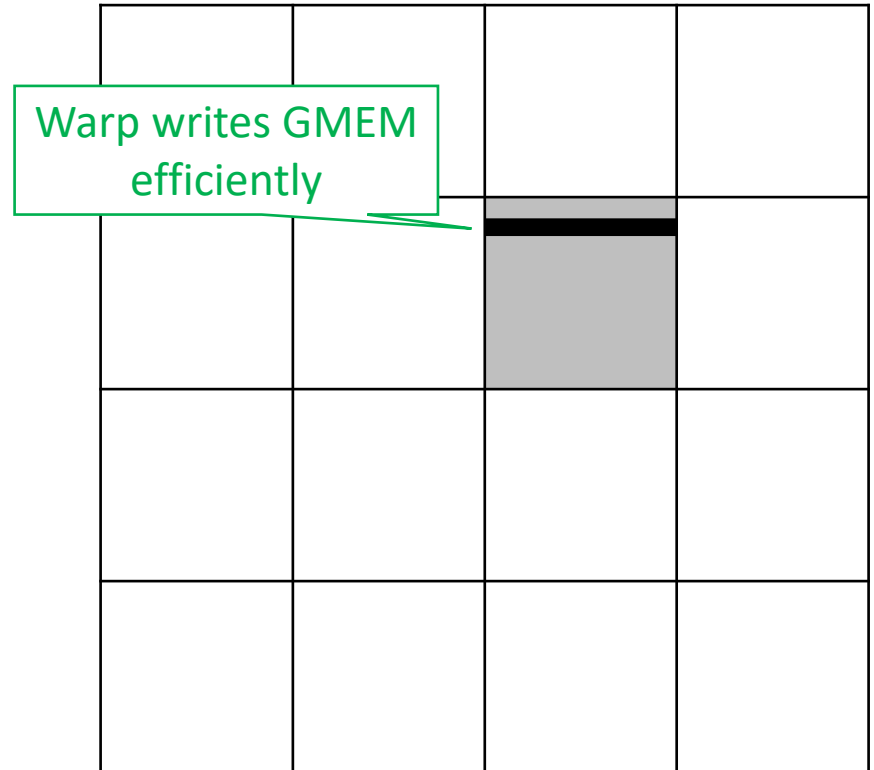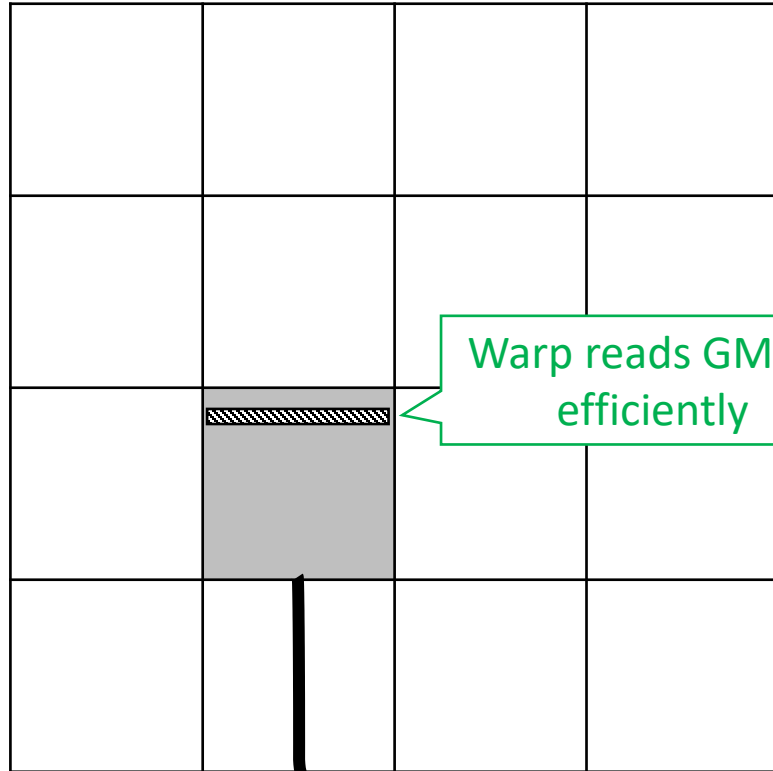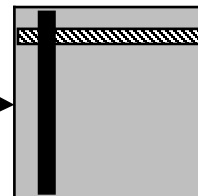
Kernel 1

Kernel 2
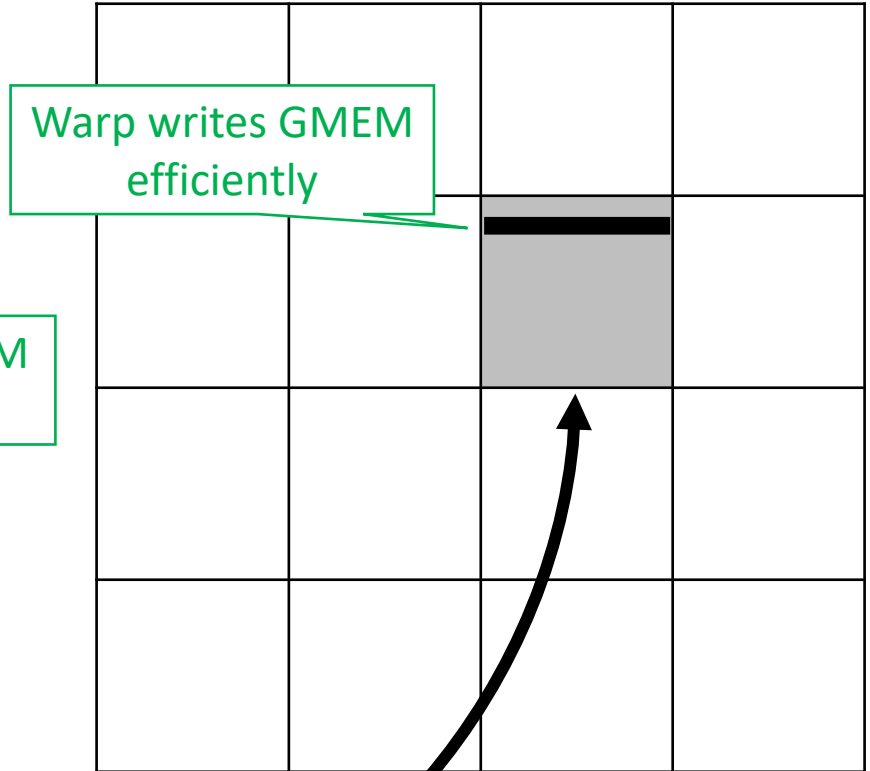
Kernel 2

```
__global__ void transpose2(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][32];

    // Each block load data efficiently from GMEM to SMEM
    int iR =
    int iC =
    s_blkData[            ][              ] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[            ][            ];
}
```

Kernel 2

```
__global__ void transpose2(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][32];

    // Each block load data efficiently from GMEM to SMEM
    int iR = blockIdx.x * blockDim.x + threadIdx.y;
    int iC = blockIdx.y * blockDim.y + threadIdx.x;
    s_blkData[threadIdx.y][threadIdx.x] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[threadIdx.x][threadIdx.y];
}
```

Shared memory

- **Organization**:

  - **32** banks, **4-byte** wide banks

  - Successive 4-byte words belong to different banks

- **Performance**:

  - Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor

  - Shared accesses are issued per 32 threads (warp)

  - Serialization: if N threads of 32 access different 4-byte words in the same bank, N accesses are executed serially

  - Multicast: N threads access the same word in one fetch

    - Could be different bytes within the same word

Bank addressing examples

No Bank Conflicts
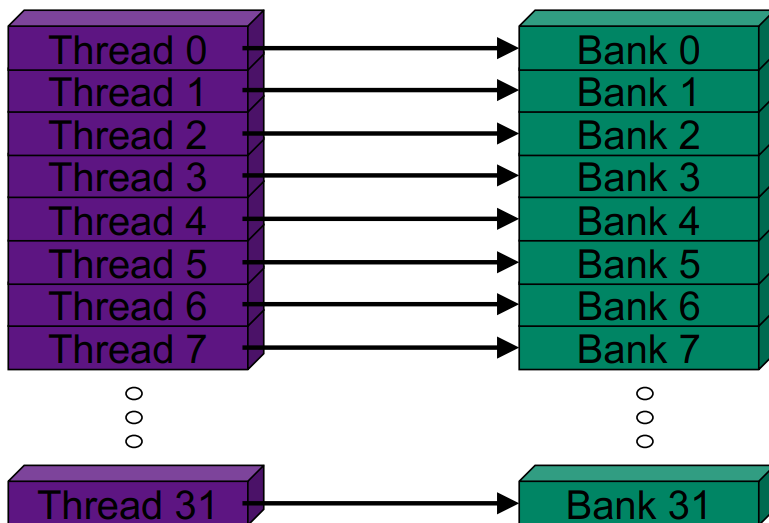
No Bank Conflicts
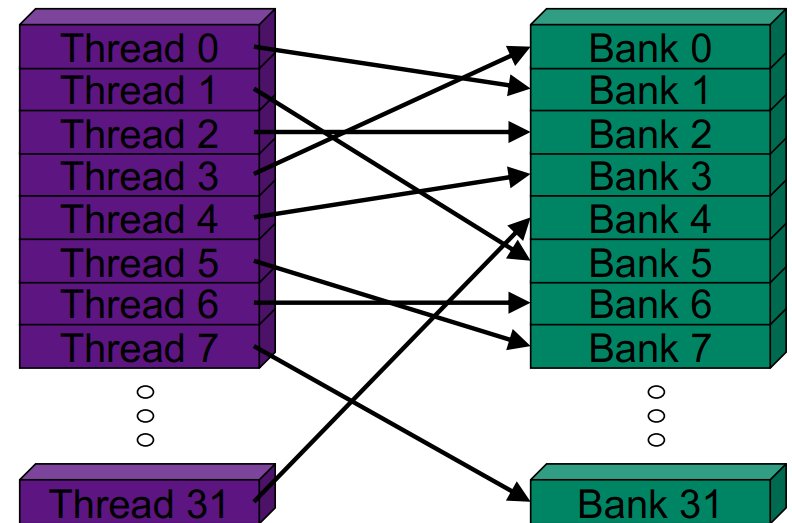
Bank addressing examples

tuần tự trên bank conflict

2-way Bank Conflicts

2-thread truy xuất 1 bank

16-way Bank Conflicts

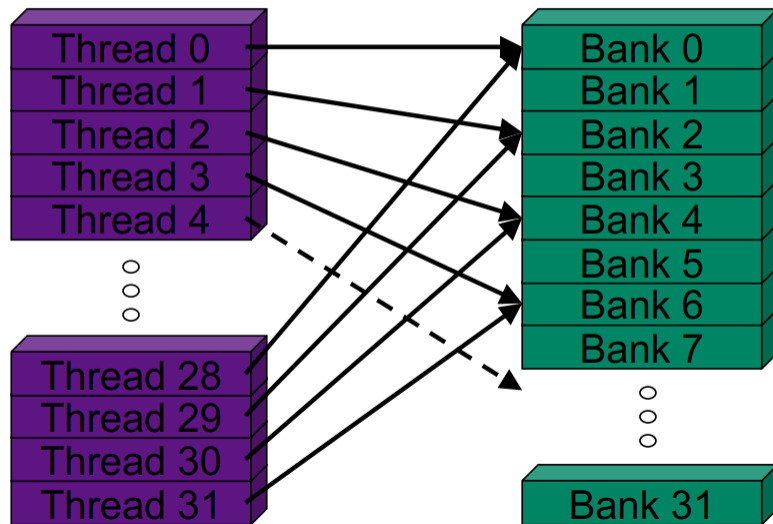# Bank conflict when accessing SMEM

SMEM is organized into **bank**s; in most CCs, each bank is 4-byte wide:

An array of
4-byte elements
in SMEM

0	1	2	3	4	5	6	7	8	...
0	4	8	12	16	20	24	28	32	36

… is organized
into banks

0	1	2	3	4	5	...	30	31
32	33	34	35	36	37	...	62	63
...	...	...	...	...	...	...	...	...
Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	...	Bank 30	Bank 31

Bank conflict when accessing SMEM

SMEM is organized into **bank**s; in most CCs, each bank is 4-byte wide:

An array of
4-byte elements
in SMEM

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |

0   4   8   12   16   20   24   28   32   36

… is organized
into banks

0	1	2	3	4	5	...	30	31
32	33	34	35	36	37	...	62	63
...	...	...	...	...	...	...	...	...
Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	...	Bank 30	Bank 31

If threads in a warp access 4-byte elements belonging to different banks, these accesses will run in parallel

Bank conflict when accessing SMEM

SMEM is organized into **bank**s; in most CCs, each bank is 4-byte wide:

An array of 4-byte elements in SMEM

0	1	2	3	4	5	6	7	8	...
0	4	8	12	16	20	24	28	32	36

… is organized into banks

0	1	2	3	4	5	...	30	31
32	33	34	35	36	37	...	62	63
...	...	...	...	...	...	...	...	...
Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	...	Bank 30	Bank 31

If threads in a warp access 4-byte elements belonging to different banks, these accesses will run in parallel

Bank conflict when accessing SMEM

SMEM is organized into **bank**s; in most CCs, each bank is 4-byte wide:

An array of 4-byte elements in SMEM

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |

0   4   8   12   16   20   24   28   32   36

3-way bank conflict

... is organized into banks

0	1	2	3	4	5	...	30	31
32	33	34	35	36	37	...	62	63
...	...	...	...	...	...	...	...	...
Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	...	Bank 30	Bank 31

If threads in a warp access different 4-byte elements belonging to the same banks, bank conflict will happen; these accesses will run sequentially

Bank conflict when accessing SMEM

SMEM is organized into **bank**s; in most CCs, each bank is 4-byte wide:

cơ chế broadcast

An array of 4-byte elements in SMEM

… is organized into banks

If threads in a warp read the same 4-byte element belonging to a bank, we will need to read one time; if write, one thread will win but we don't know which one
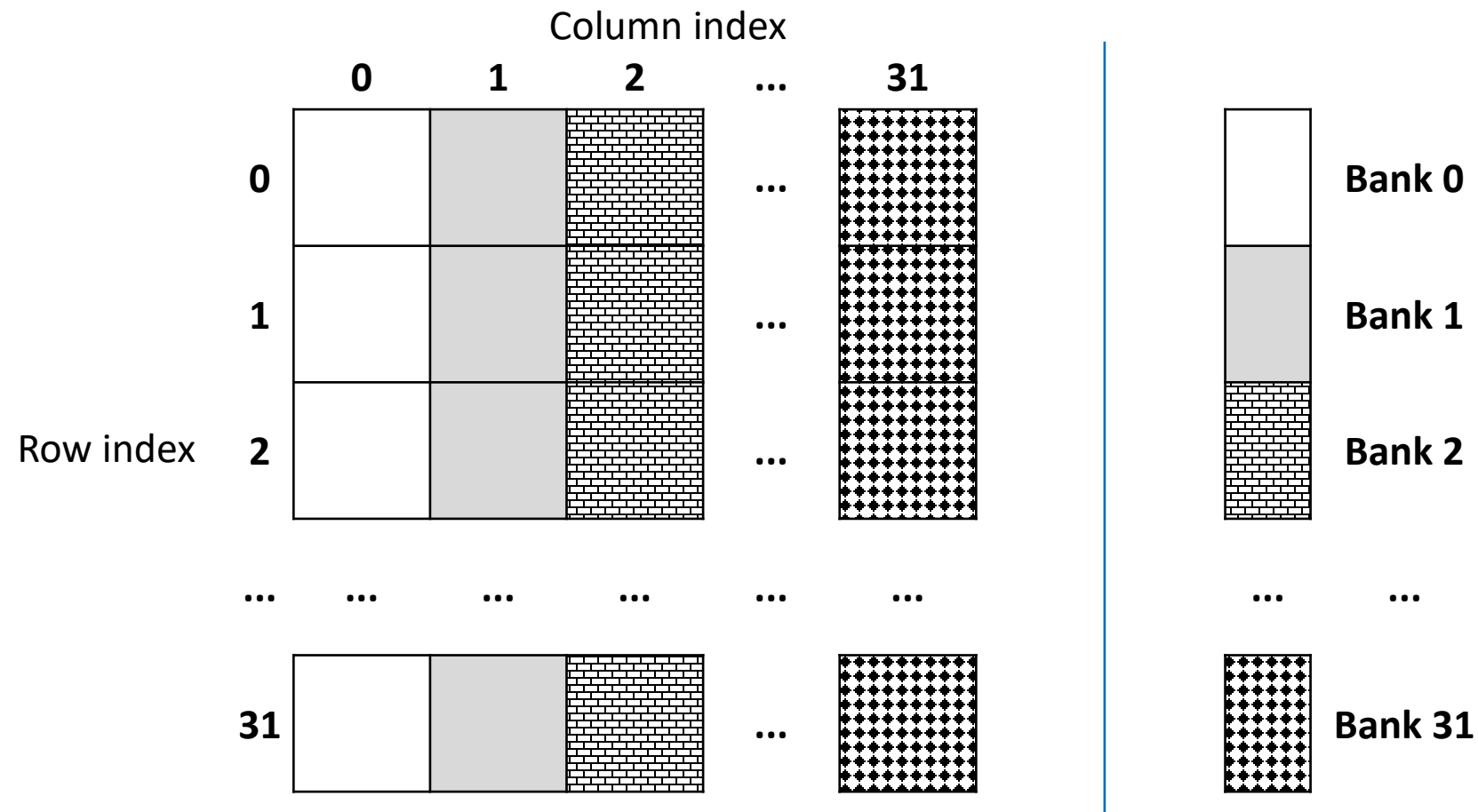
Let's reconsider kernel 2 in matrix transpose example …

Bank conflict in kernel 2
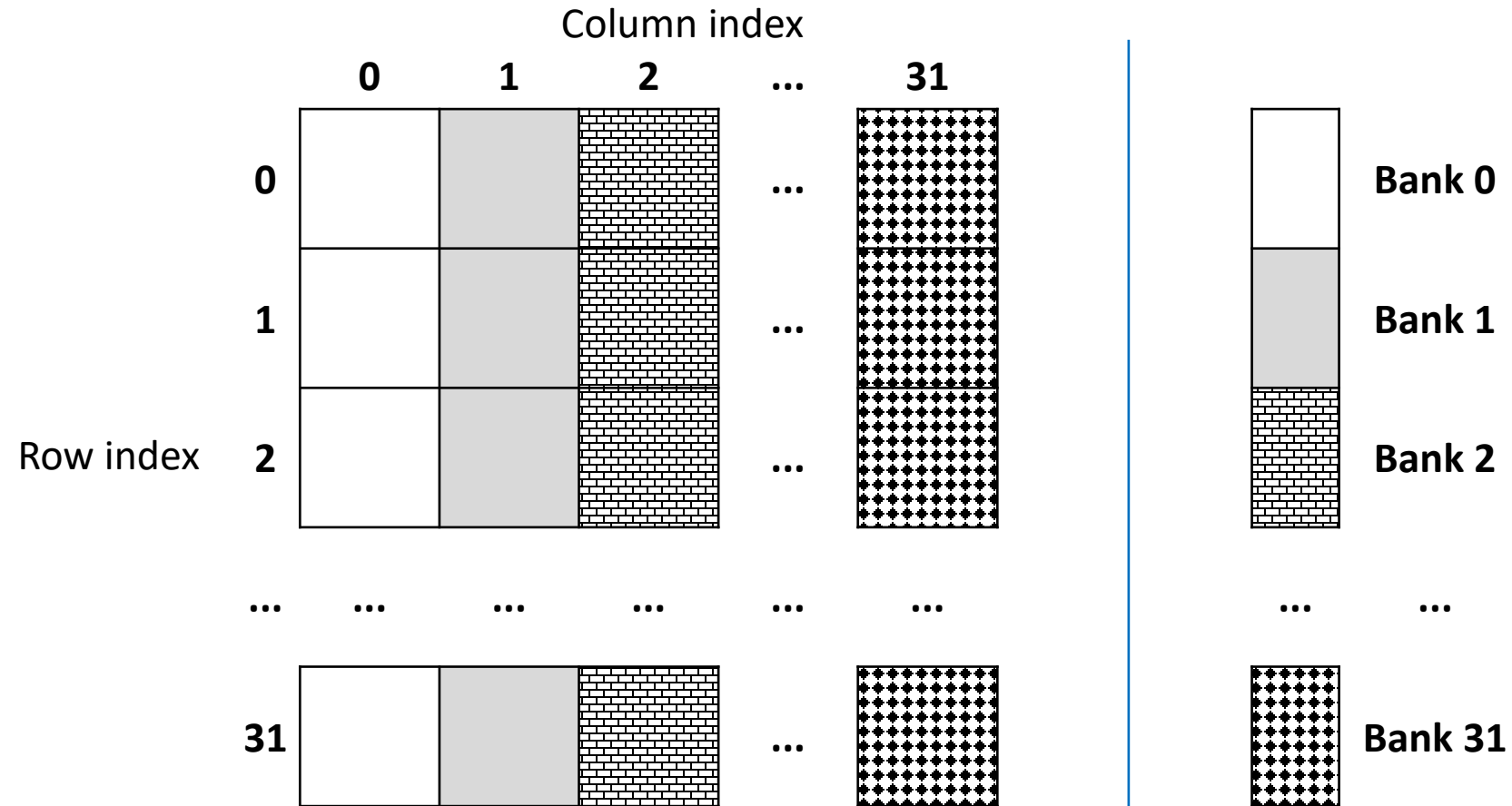
Warp writes a row on SMEM: bank conflict?
Warp reads a column on SMEM: bank conflict?

Bank conflict in kernel 2

Warp writes a row on SMEM: no bank conflict
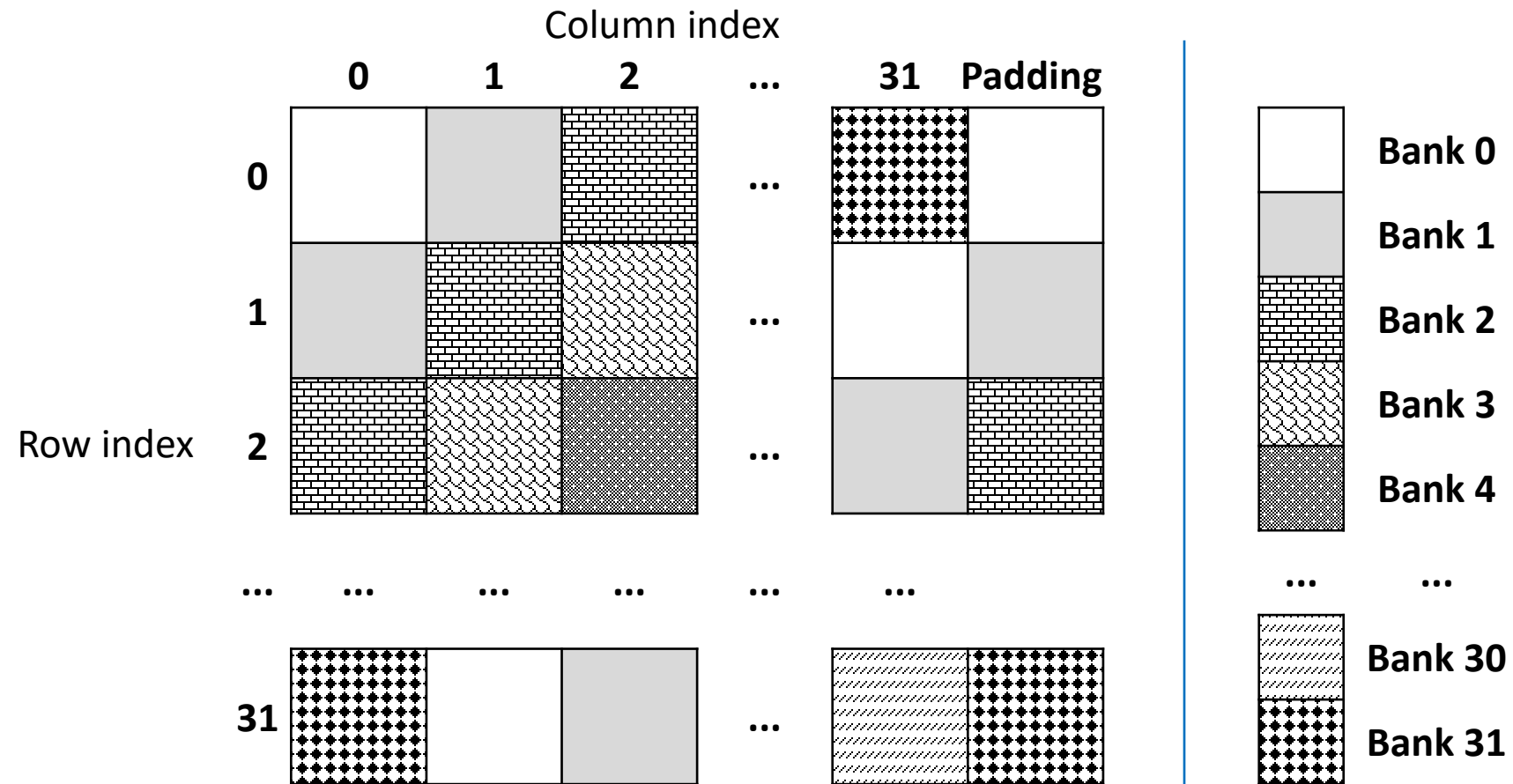Warp reads a column on SMEM: bank conflict

Kernel 3 – solve bank conflict in kernel 2

```
__global__ void transpose3(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][33];

    // Each block load data efficiently from GMEM to SMEM
    int iR = blockIdx.x * blockDim.x + threadIdx.y;
    int iC = blockIdx.y * blockDim.y + threadIdx.x;
    s_blkData[threadIdx.y][threadIdx.x] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[threadIdx.x][threadIdx.y];
}
```

Kernel 3 – solve bank conflict in kernel 2

Column index

Row index

Warp writes a row on SMEM: no bank conflict
Warp reads a column on SMEM: no bank conflict

THE END

Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022

- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014

- [3] VOLTA Architecture and performance optimization, Guillaume Thomas-Collignon, Paulius Micikevicius