EECS 471
Applied GPU Programming

Lecture 5: DRAM, Shared Mem, Coalescing

DRAM Organization & Coalescing

Global Memory (DRAM) Bandwidth

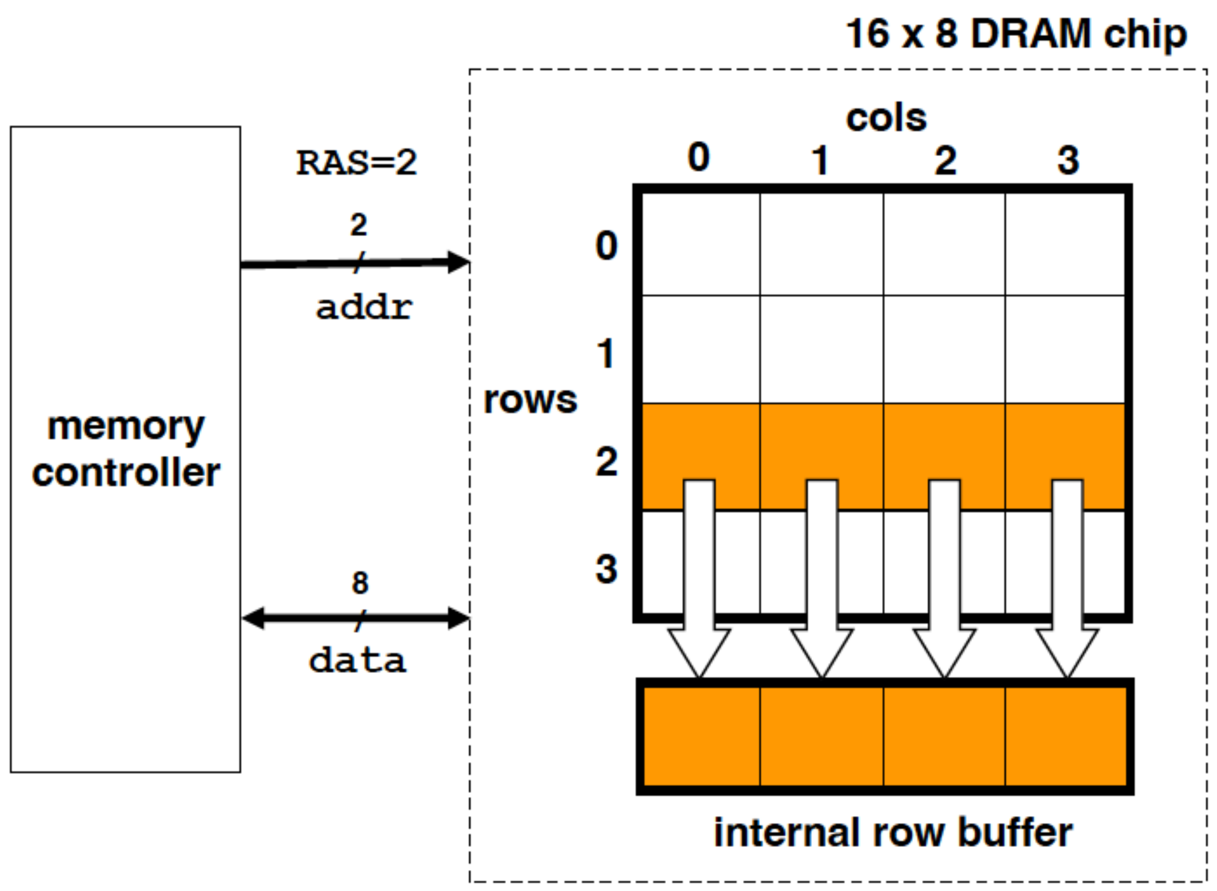
Ideal

Reality

Review: Reading DRAM

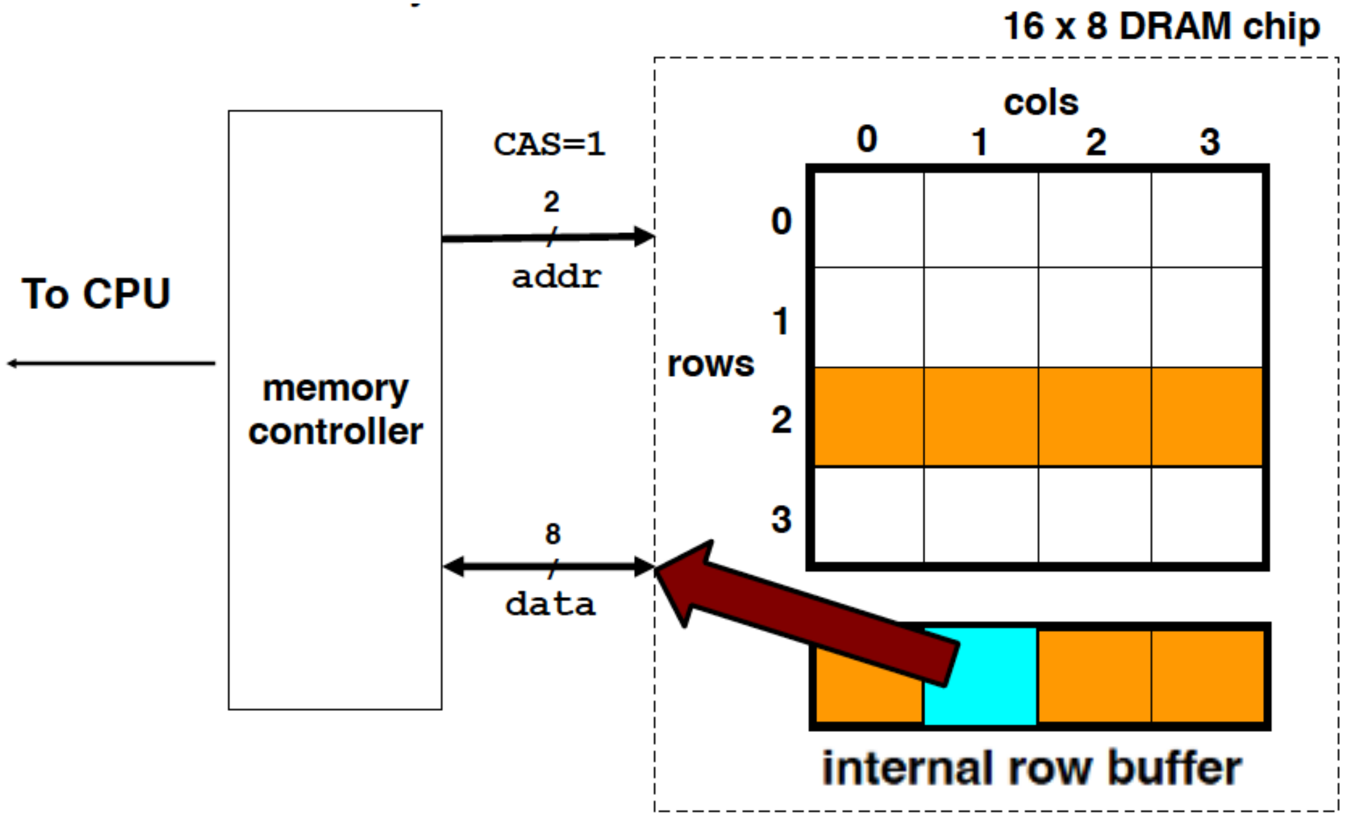
- Access done in two steps
  - Step 1(a): Row access strobe (RAS) selects row 2.

  - Step 1(b): Row copied from DRAM array to row buffer.



Reading DRAM supercell (2,1)

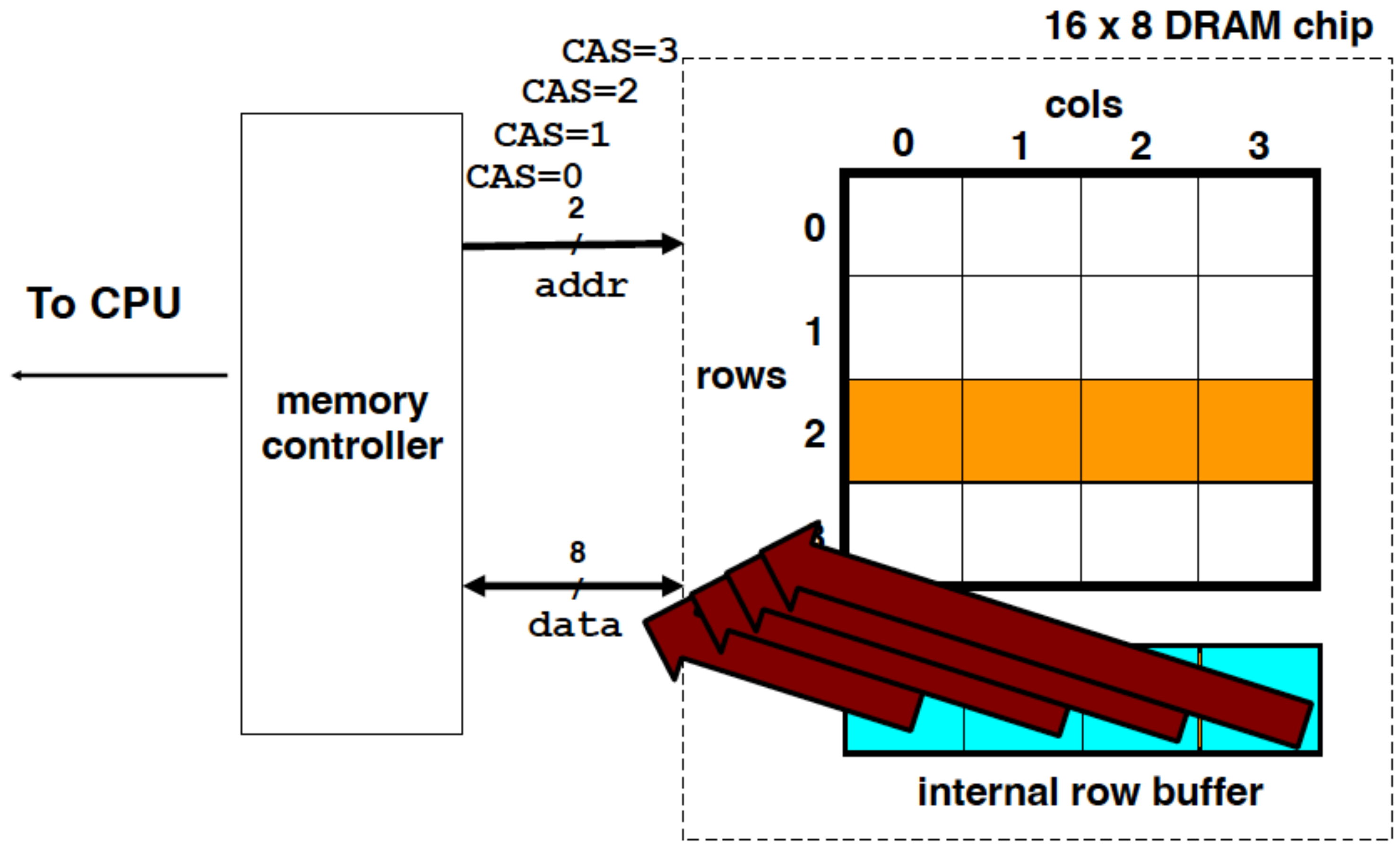
- Access done in two steps
  - Step 2(a): Column access strobe (CAS) selects column 1.
  - Step 2(b): Copy supercell (2,1) from row buffer to data lines, and eventually back to the CPU.



Fast Page Mode: Reading (2,0)(2,1)(2,2)(2,3)

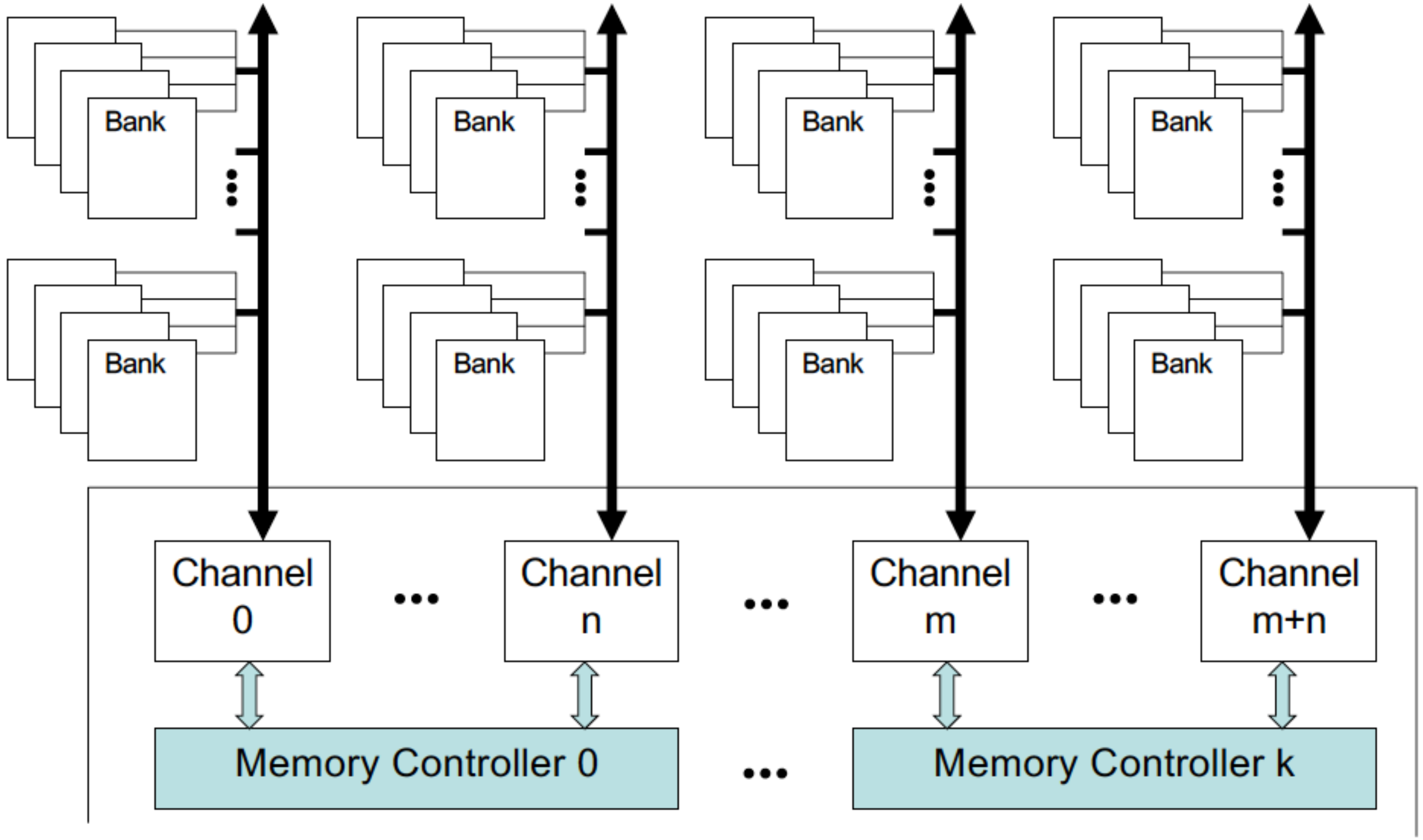
– Consecutive CAS select consecutive columns of the same row

– Direct row buffer access, no precharge VERY fast memory access

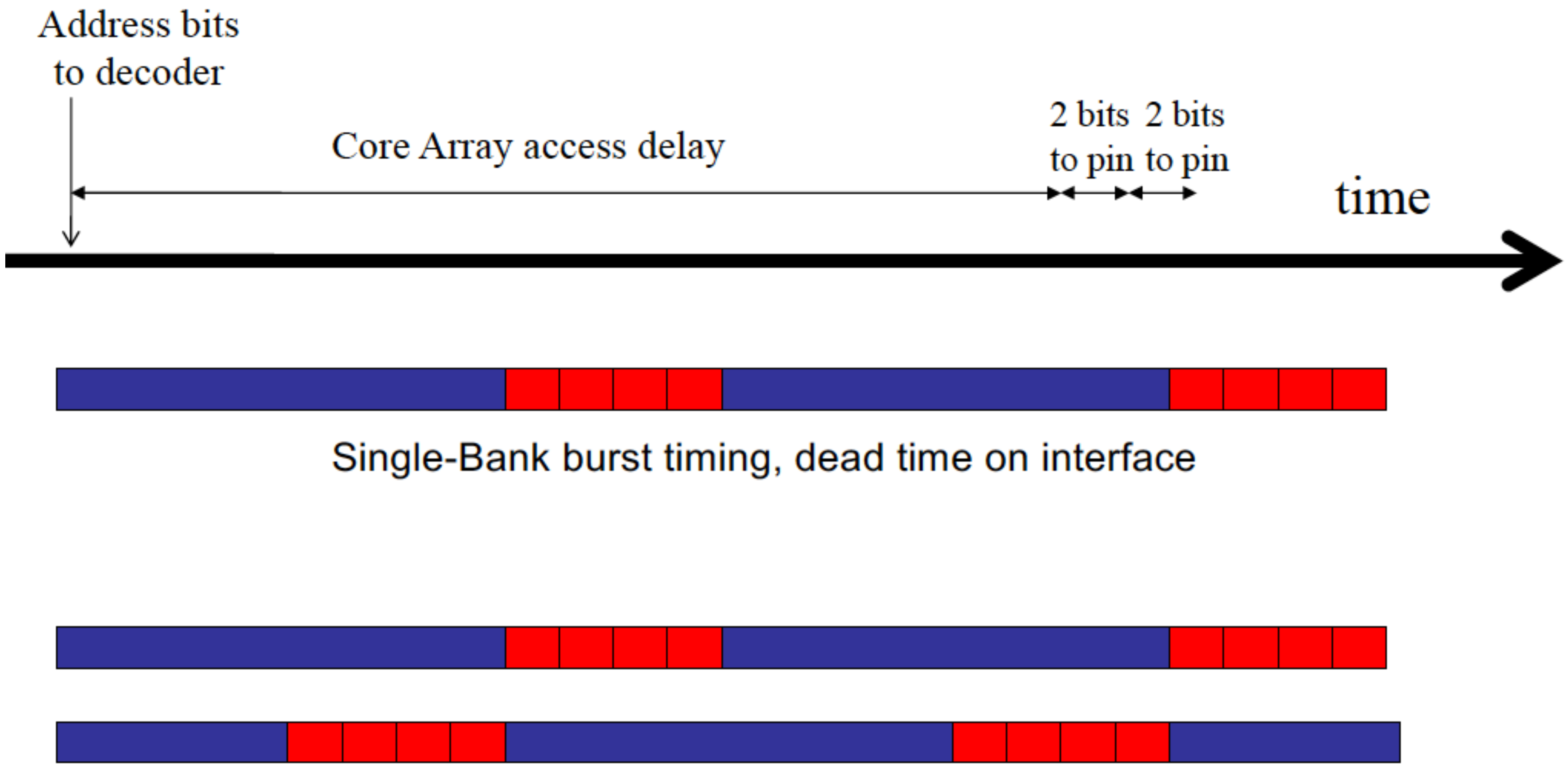


Multiple Banks & Memory channels

- Divide the memory address space into N parts
  - N is number of memory channels
  - Assign each memory portion to a channel
  - Further subdivide each channel's worth of memory into banks



DRAM bursting for the 8x2 bank



Single-Bank burst timing, dead time on interface

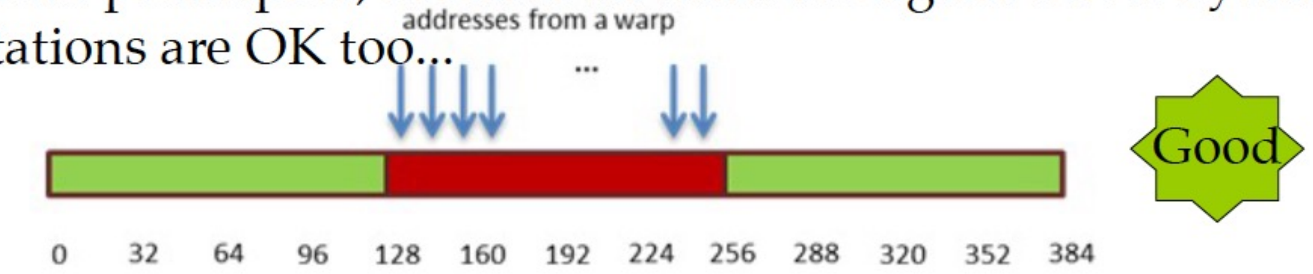
Multi-Bank burst timing, reduced dead time

How to get high performance with Global Memory?

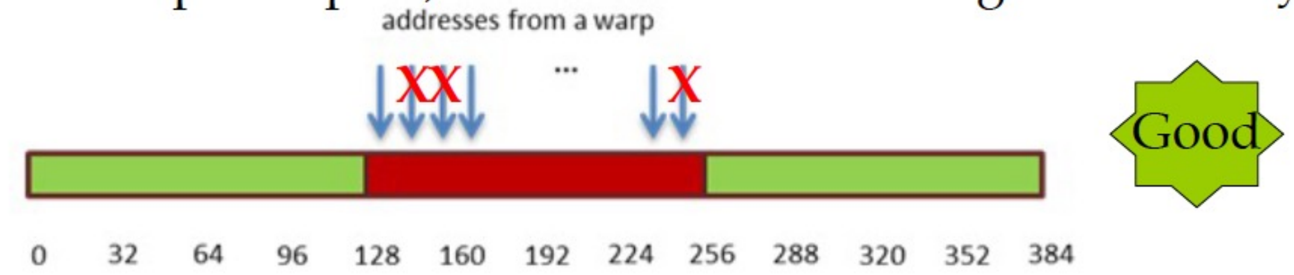
- Coalesce global memory accesses
  - 32 threads access memory together
  - Can coalesce into single reference, if addresses within a 128-byte block
    - Instead of issuing 32 memory accesses of 4 bytes each
    - Issue 1 memory access that is 128 bytes wide (load granularity)
    - GDDR5 provides 32 bytes/mem_clock à 128 bytes in 4 Mem cycles
    - But GDDR5 @ 6.008GHz, while cores @ 1.006GHz
      - one core cycle is 5+ memory cycles
      - get 128 bytes in 1 core cycle
  - Coalesce requests to get one access as wide as possible
    - e.g., a [threadID] works well
  - Ideal: 1 warp → 128 bytes of consecutive memory
    - Aligned to 128-byte boundary…

Coalesced access: reading floats

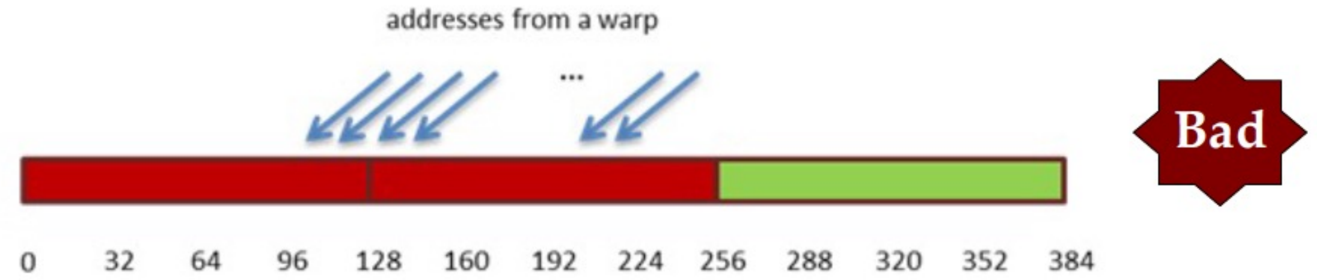
All threads participate, accesses fit within a region of 128 bytes.
Permutations are OK too...



Not all threads participate, accesses fit within a region of 128 bytes



Misaligned accesses (fit in two regions of 128 bytes):



Coalescing experiment code

```
__global__ void kernel (float *a) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  const int iiTest=0;

  switch (iiTest) {
  case 0:
    a[i]+=i;
    break;
  case 1:
    if ((i & 0x3) != 00) a[i]++;
    break;
  case 2:
    if ((i & 0x3) != 00) a[i]+=1.0;
    else a[0]+=1.0;
    break;
  case 3:
    if ((i & 0x3) != 00) a[i]++;
    else a[blockIdx.x * blockDim.x]++;
  }
}
```

Coalescing experiment code

```
auto time_hres_start=std::chrono::high_resolution_clock::now();
auto time_now=std::chrono::high_resolution_clock::now();
auto time_span=std::chrono::duration_cast<std::chrono::duration<double>>(time_now-time_hres_start);

 for (int i = 0; i < TIMES; i++) {
   time_hres_start=std::chrono::high_resolution_clock::now();
   kernel <<<n_blocks, block_size>>> (a_d);
   cudaDeviceSynchronize();

   time_now=std::chrono::high_resolution_clock::now();
   time_span+=std::chrono::duration_cast<std::chrono::duration<double>>(time_now-time_hres_start);
 }

  printf ("Time %e\n", time_span/TIMES);
```

Coalescing experiment code

```
__global__ void kernel (float *a) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  const int iiTest=0;

  switch (iiTest) {
  case 0:
    a[i]+=i;                              t=1.0
    break;
  case 1:
    if ((i & 0x3) != 00) a[i]++;          t=1.0
    break;
  case 2:
    if ((i & 0x3) != 00) a[i]+=1.0;       t=24.6
    else a[0]+=1.0;
    break;
  case 3:
    if ((i & 0x3) != 00) a[i]++;          t=3.7
    else a[blockIdx.x * blockDim.x]++;
  }
}
```

Coalescing experiment code

- Kernel:
  - Read float, increment, write back: a[i]++;
  - 3M floats (12MB)
  - Times averaged over 10K runs

- 12K blocks x 256 threads/block
  - Coalesced (a[i]++)
    - **t=1.0**
  - Coalesced / some don't participate (if (i & 0x3 != 0) a[i]++)
    - 3 out of 4 participate
    - **t=1.0**
  - Uncoalesced / outside the region
    - Every 4 access a[0]
    - **t=24.6** → 24x slowdown: 4x from not coalescing and another 8x from contention for a[0] (25% of all threads of all blocks access a[0])

      ```
      if (i & 0x3 != 0) a[i]++;
      else a[0]++;
      ```
    - **t=3.7**→ 4x slowdown: mostly from not coalescing (only threads in block access a[startOfBlock])

      ```
      If (i & 0x3) != 0) a[i]++;
      else a[startOfBlock]++;
      ```

Uncoalesced float3 access code

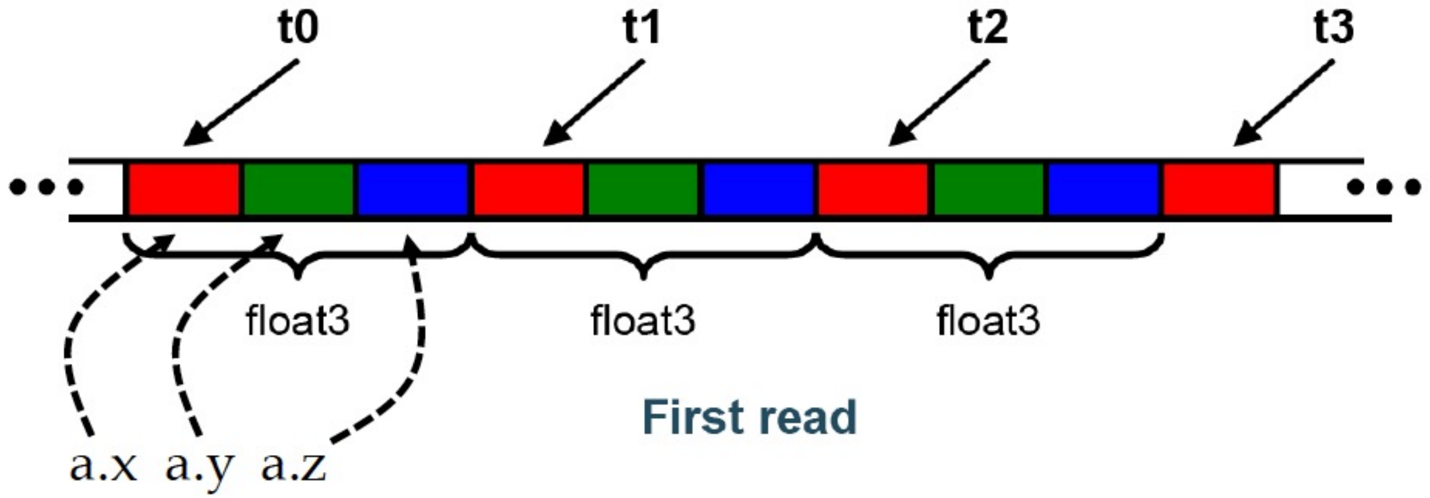
```
__global__ void
accessFloat3(float3 *d_in, float3 d_out)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  float3 a = d_in[index];

  a.x += 2;
  a.y += 2;
  a.z += 2;

  d_out[index] = a;
}
```

This will be implemented as:

```
ld d_in[index].x
ld d_in[index].y
ld d_in[index].z
```

Simple float3 access sequence

- float3 is 12 bytes
  - Each thread ends up executing 3 32bit reads
  - sizeof(float3) = 12
  - Offsets:
    - 0, 12, 24, …, 180 (for loading the .x part)
    - 4, 16, 28, …, 184 (for loading the .y part)
    - 8, 20, 32, …, 188 (for loading the .z part)
  - Warp reads three 128-byte non-contiguous regions each time
    - 12 bytes/thread * 32 threads = 384 bytes = 3 * 128 bytes



Coalescing + tiling float3 strategy

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes of SMEM

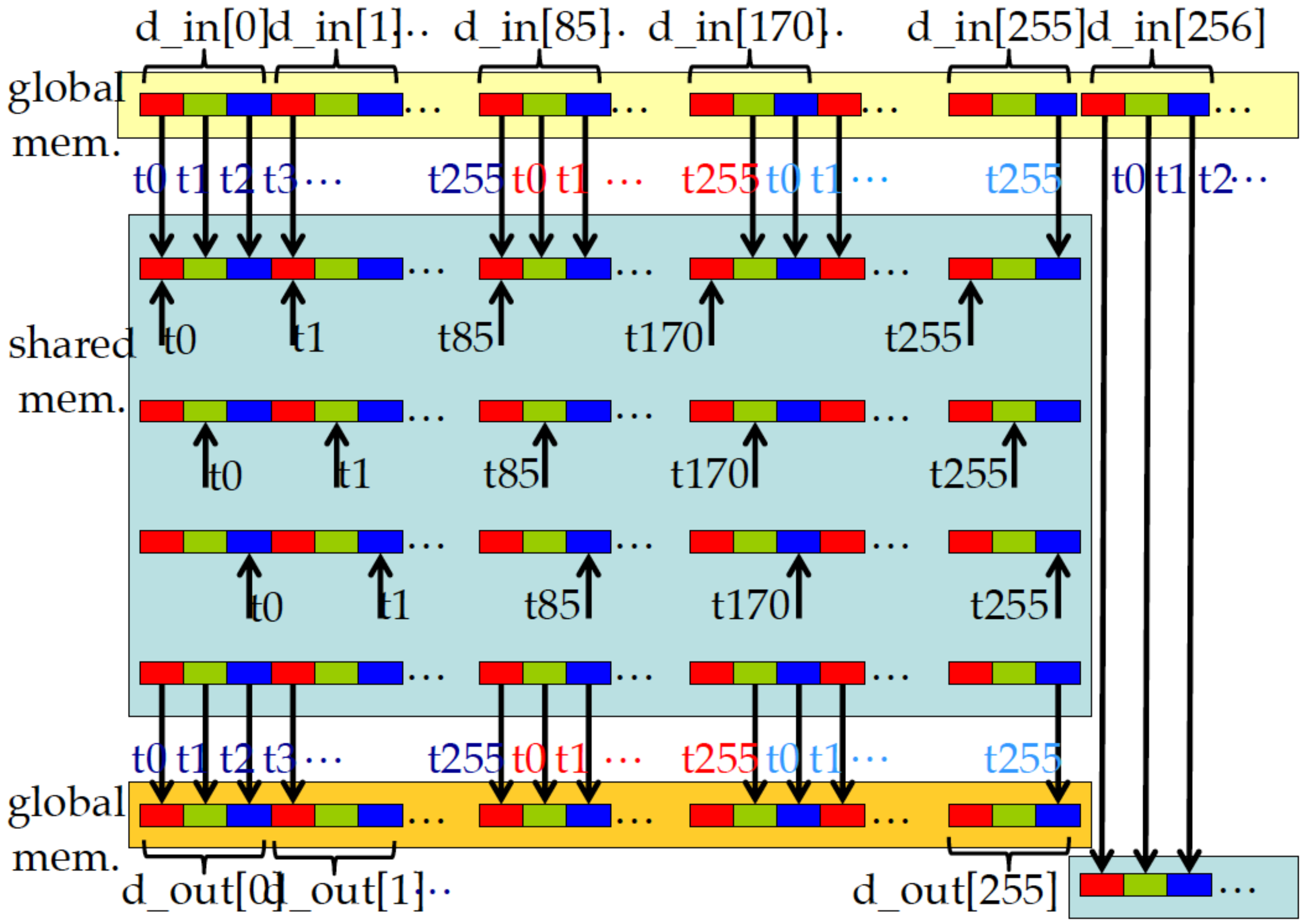
- Three Phases:
  - Phase 1: Fetch data in shared memory
    - Each thread reads 3 scalar floats
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync

  - Phase 2: Processing
    - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
    - Rest of the compute code does not change

  - Phase 3: Write results back to global memory
    - Each thread writes 3 scalar floats
    - Offsets: 0, (threads/block), 2*(threads/block)

Coalescing + tiling float3 access

- 256 threads/block



Coalescing + tiling float3 access

- 256 threads/block

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
```

Read the input through SMEM

```
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x]     = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];
```

Compute code is not changed

```
    a.x += 2;
    a.y += 2;
    a.z += 2;
```

Write the result through SMEM

```
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index]     = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Experiment: float3

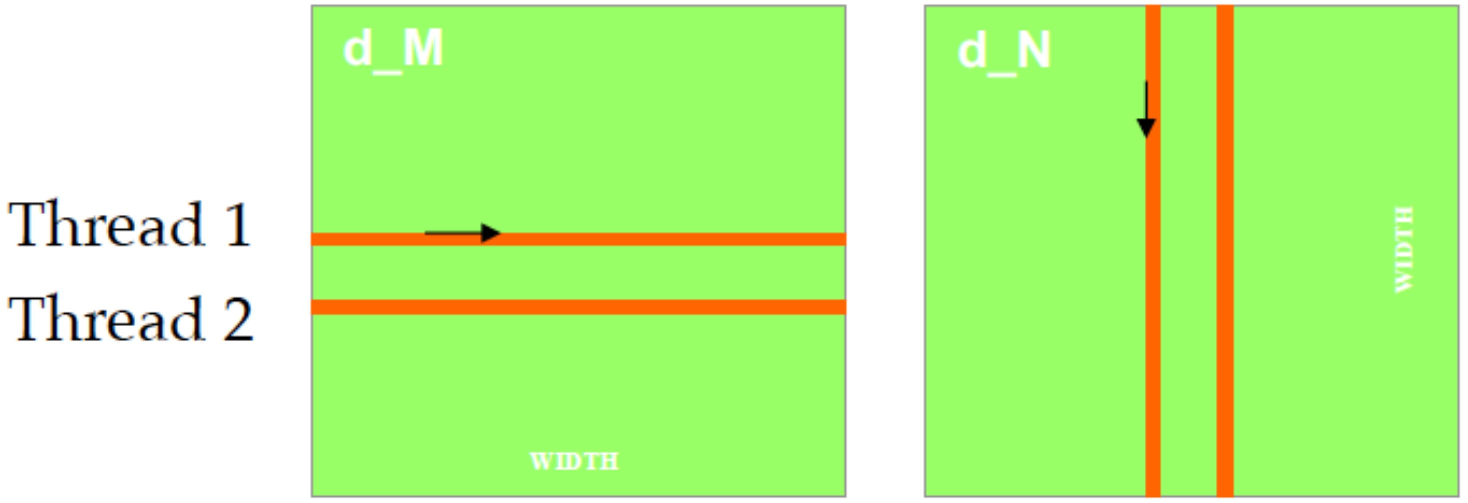
- Experiment:
  - Kernel: read a float3, increment each element, write back
  - 1M float3s (12MB)
  - Times averaged over 10K runs

- Results:
  - 4K blocks x 256 threads:
    - **t=2.6**– float3 uncoalesced
    - **t=1.0** – float3 coalesced through shared memory

Coalescing in Simple Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 // Calculate the row index of the P element and M
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 // Calculate the column index of P and N
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;

    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
      Pvalue += M[Row*Width+k] * N[k*Width+Col];

    P[Row*Width+Col] = Pvalue;
  }
}
```

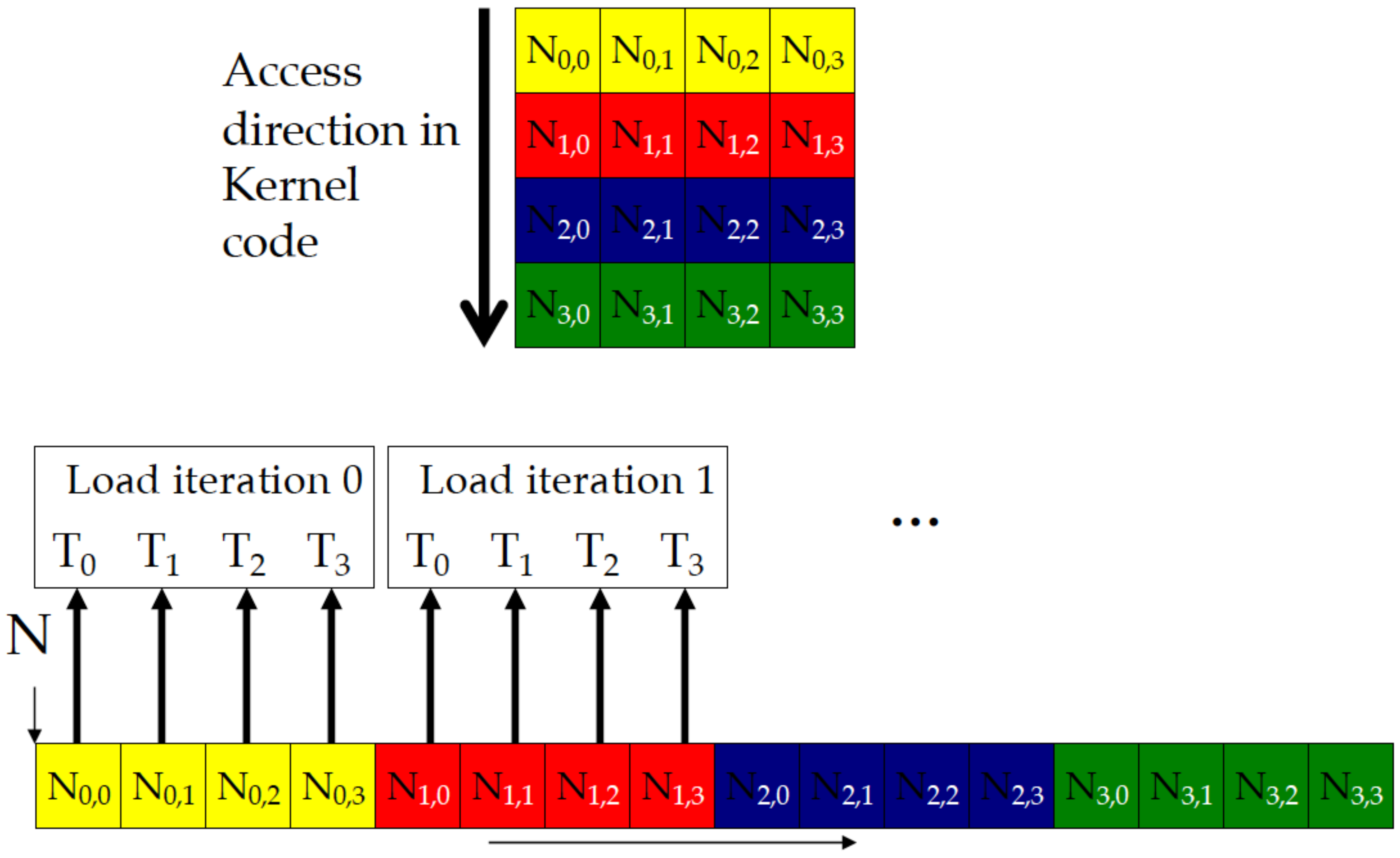
Two Access Patterns



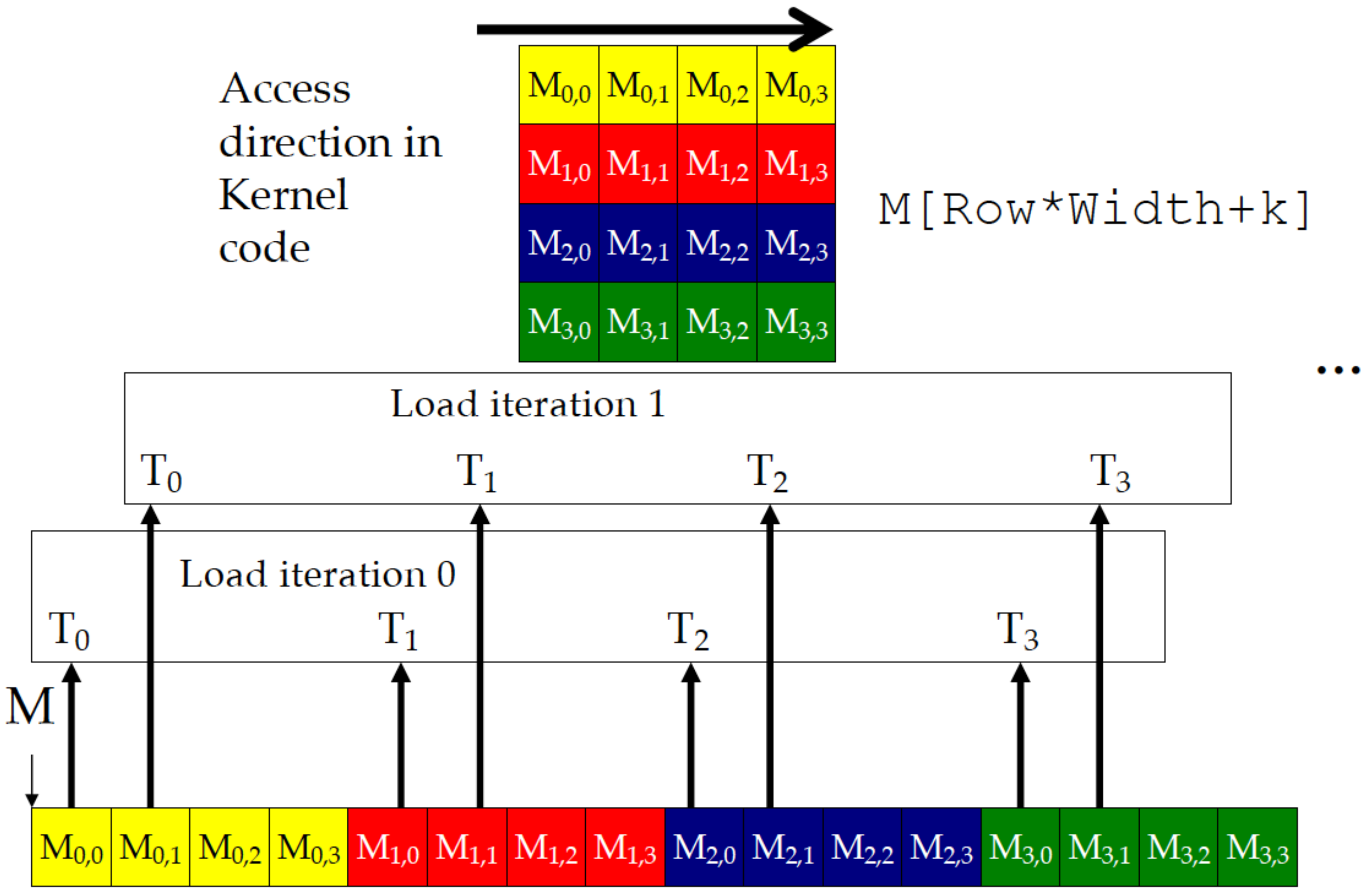
`M[Row*Width+k]`   `N[k*Width+Col]`

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

N accesses are coalesced

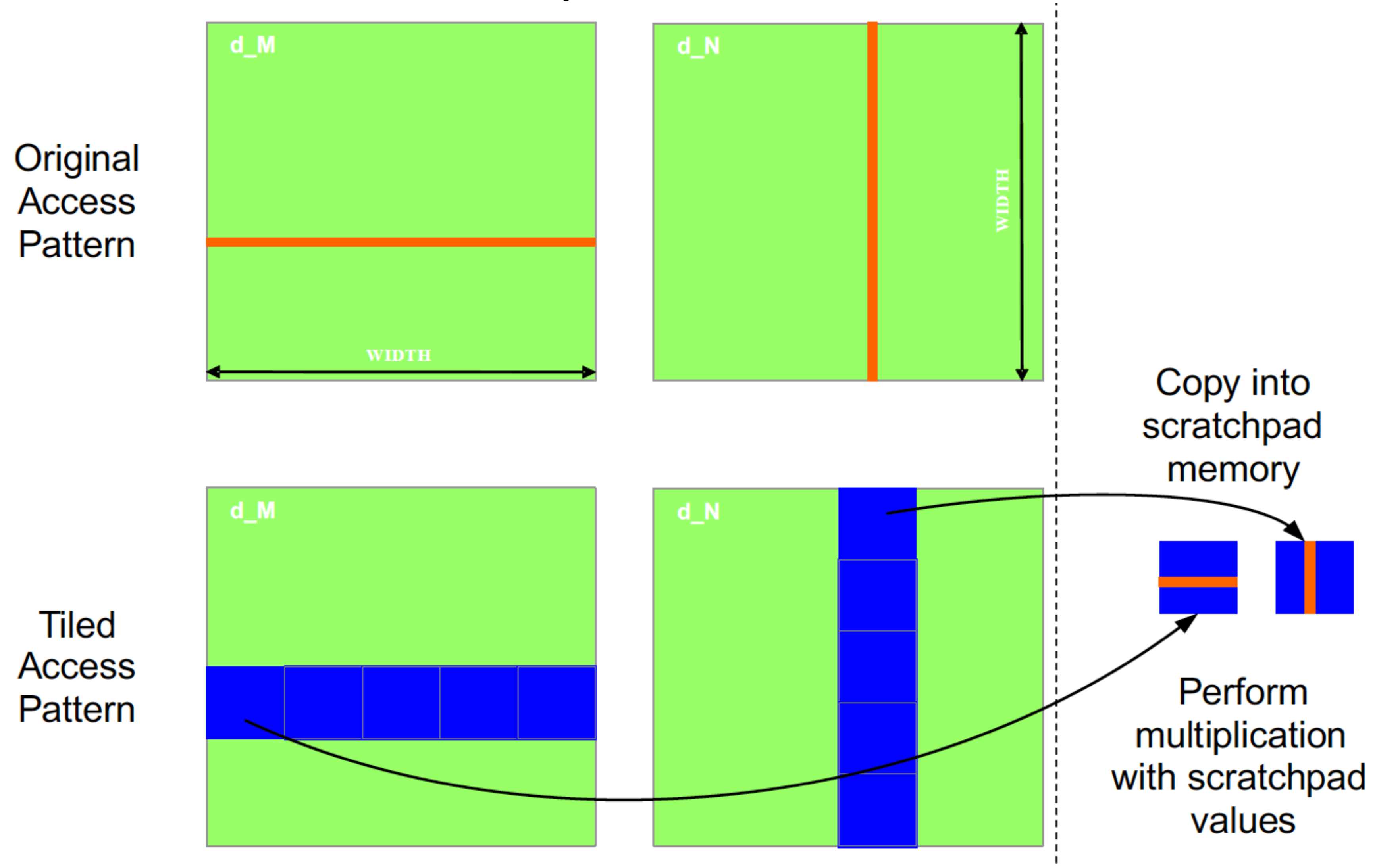

Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

M accesses are not coalesced



Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M[Row*Width+k]
…
Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Use shared memory to enable coalescing in tiled matrix multiplication


Original Access Pattern
d_M
WIDTH
d_N
WIDTH
Copy into scratchpad memory
Tiled Access Pattern
d_M
d_N
Perform multiplication with scratchpad values

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[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 m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of M and N tiles into shared memory
         subTileM[? ][? ] = M[          ?              ];
         subTileN[? ][? ] = N[          ?              ];
         __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
             Pvalue += subTileM[ty][k] * subTileN[k][tx];
         __synchthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Global memory coalescing summary

Coalescing greatly improves throughput

Critical to small or memory-bound kernels

- Accessing structs (or non-fundamental types, like float3) in global memory may break coalescing
- Prefer Structures of Arrays (SoA) over Arrays of Structures (AoS)

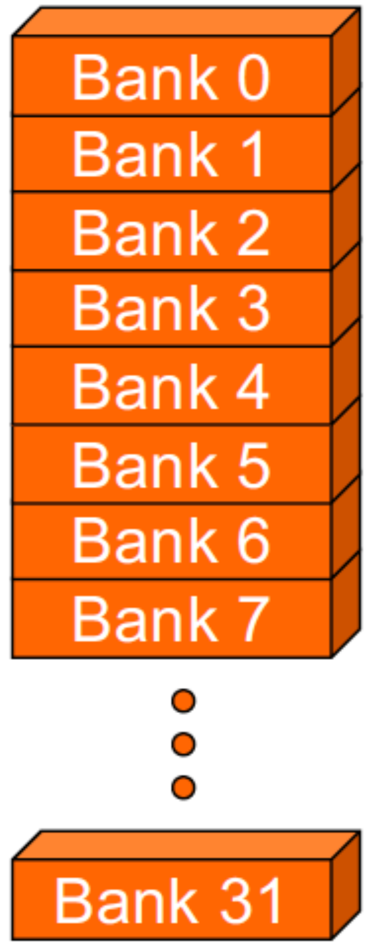
```
struct {
    typeA a;
    typeB b;
    typeC c;
} array_of_structs[N];

struct {
    typeA a[N];
    typeB b[N];
    typeC c[N];
} struct_of_arrays;
```

- If SoA is not viable, read/write through SMEM

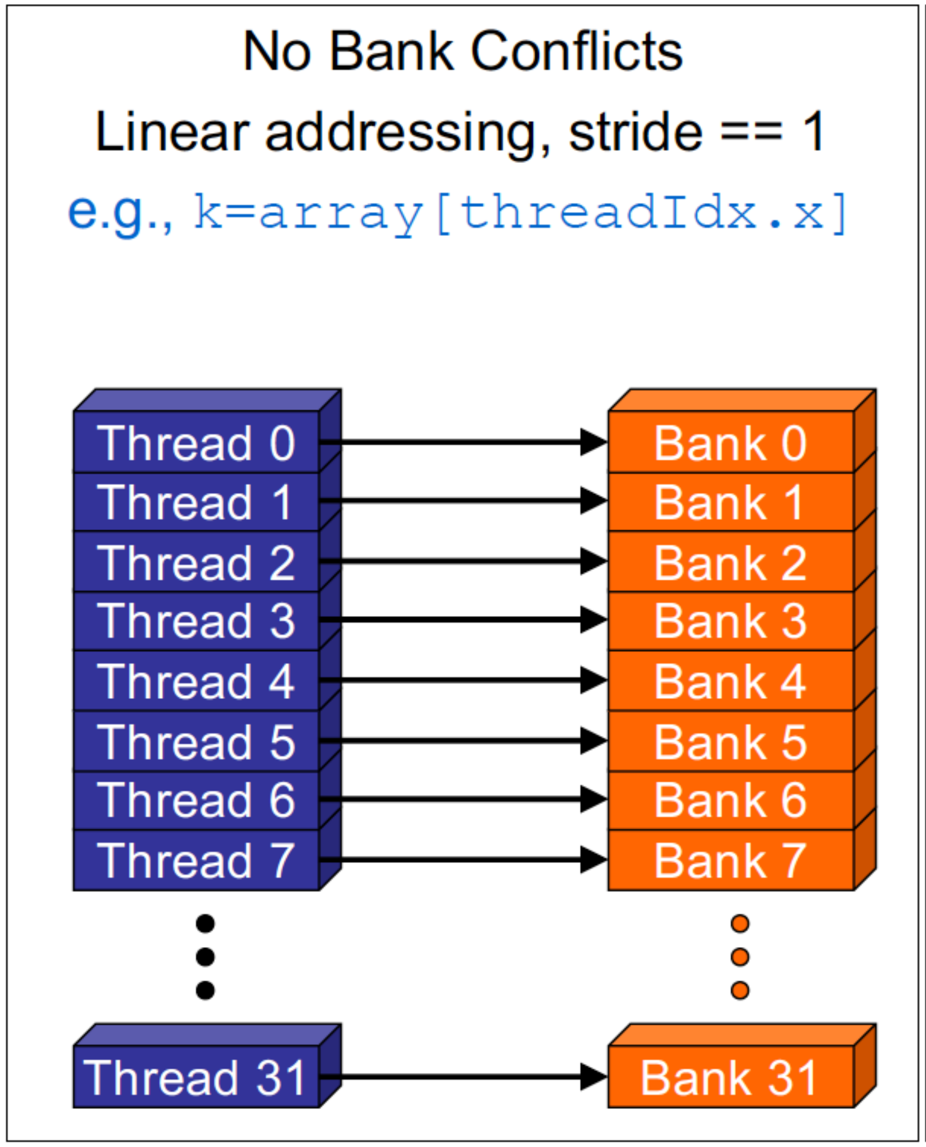
Shared Memory Organization & Bank Conflicts

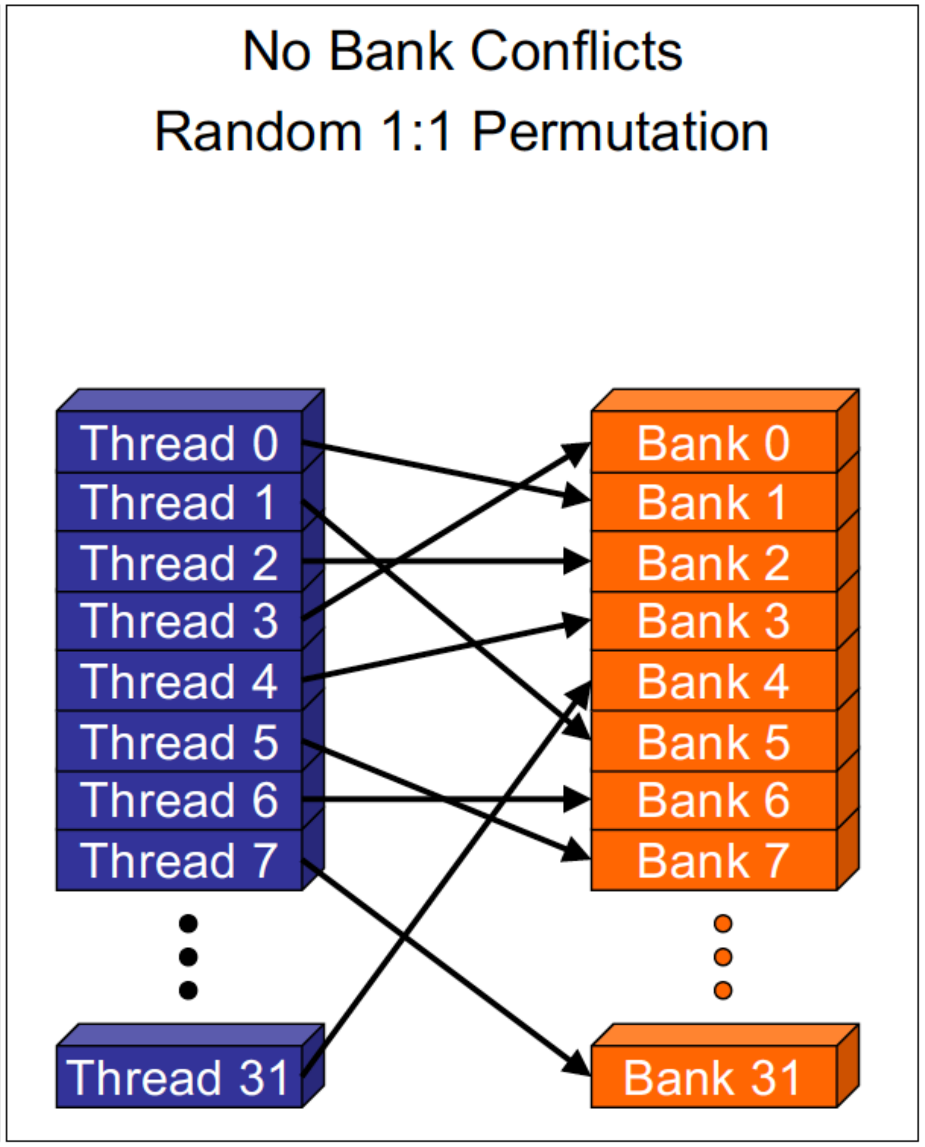
Shared memory (SMEM)

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Shared memory bank addressing examples



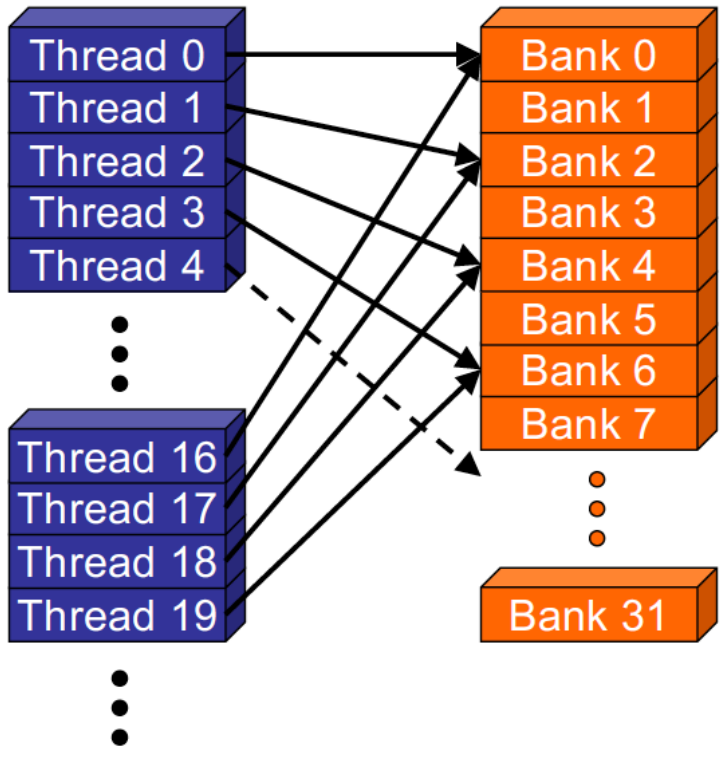


Shared memory bank addressing examples

2-way Bank Conflicts

Linear addressing, stride == 2

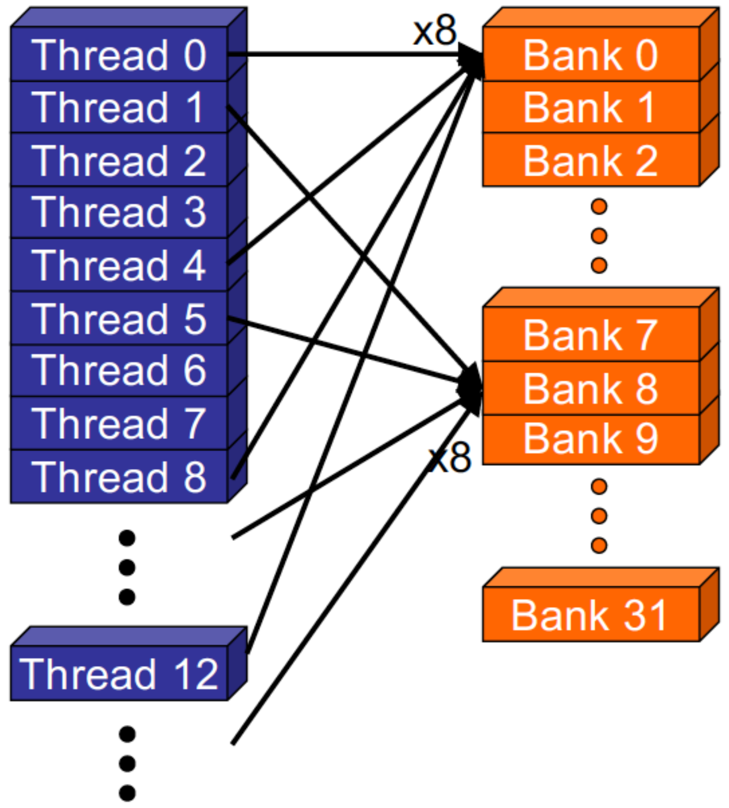
e.g., `k=array[threadIdx.x*2]`



8-way Bank Conflicts

Linear addressing, stride == 8

e.g., `k=array[threadIdx.x*8]`

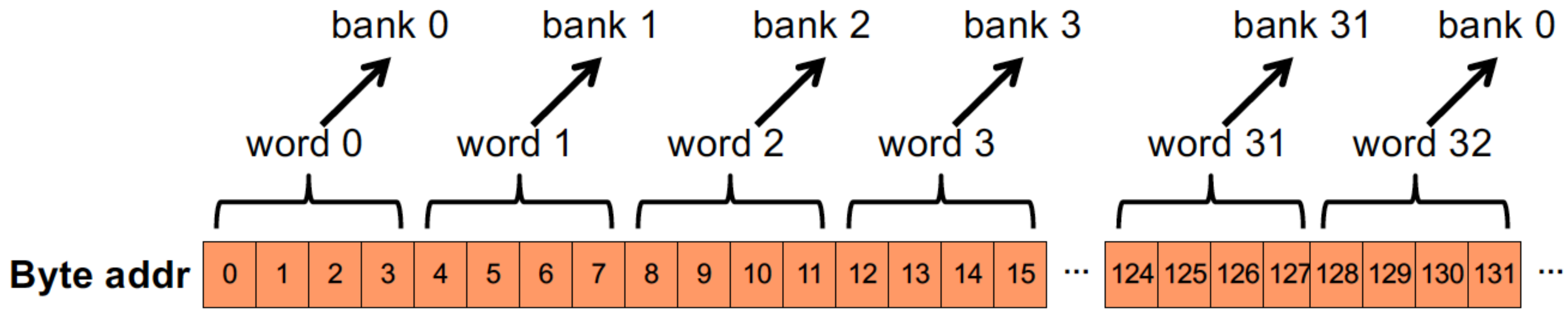


Mapping addresses to SMEM banks (GTX680)

Each bank has a bandwidth of 64 bits per clock cycle

Successive 64-bit words are assigned to successive banks (8-byte-wide mode)

Successive 32-bit words are assigned to successive banks (4-byte-wide mode)



GTX680 has 32 banks

- So (for 4-byte mode) bank = word_id % 32 = (address / 4) % 32
  - a.k.a. ***address interleaving at 4-byte granularity***
- Same as the size of a warp
  - No bank conflicts between different warps, only within a single warp

Shared memory bank conflicts

- Shared memory is as fast as registers, provided that:
  1. There are no bank conflicts
  2. There is sufficient shared memory bandwidth
  - if either condition fails, shared memory is much slower than registers

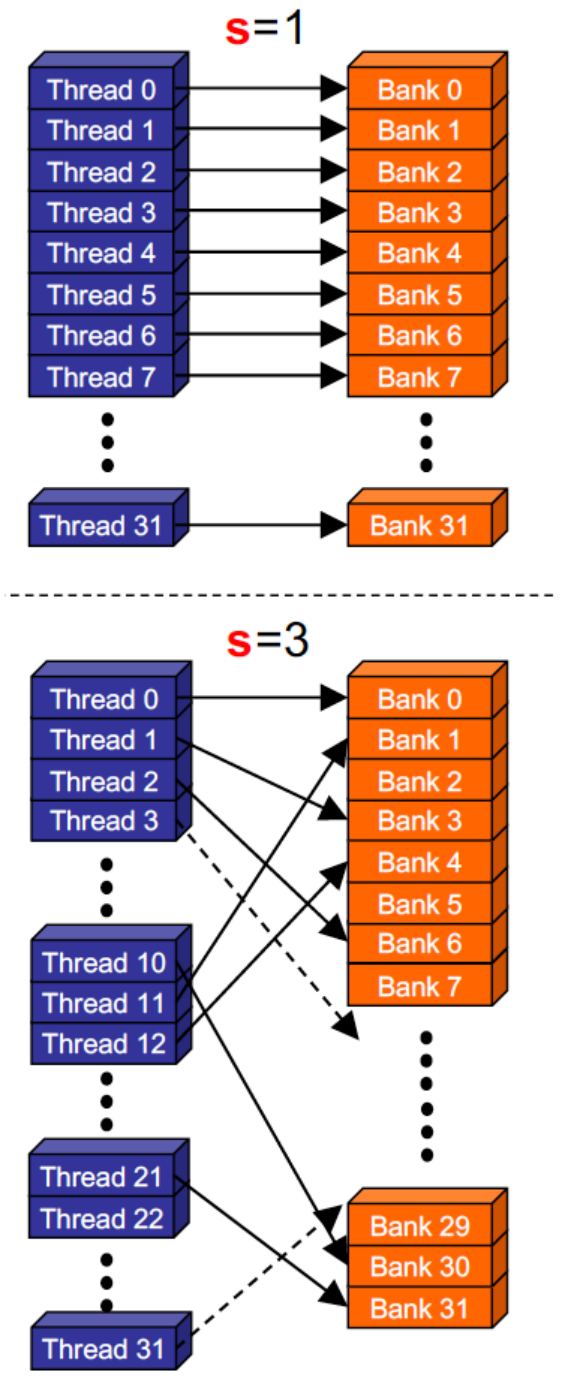
- The fast case:
  - If all threads of a warp access different banks
    → there is no bank conflict
  - If all threads of a warp read an identical address
    → there is no bank conflict (broadcast)

- The slow case:
  - Bank Conflict: multiple threads in the same warp access the same bank for a different row
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

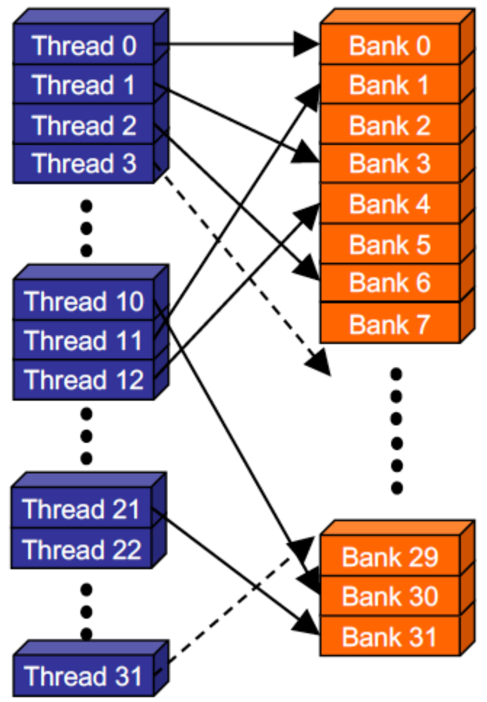
Linear addressing

```
__shared__ float shared[256];
float foo;
foo =  shared[baseIndex + s*threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 32 banks on GTX680, so s must be odd



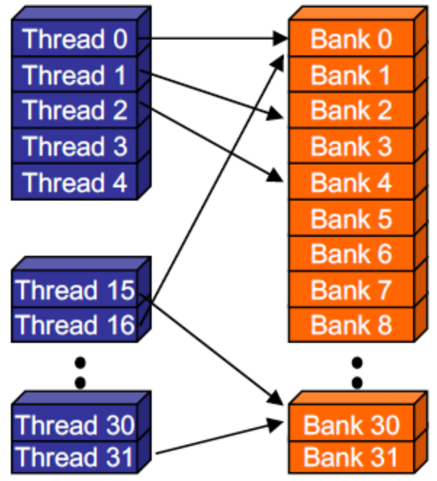
Structs and bank conflicts



```
struct vector3 { float x, y, z; };
__shared__ struct vector3 v3[64];
```

- This has no bank conflicts for vector struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 32)

```
struct vector3 v = v3[baseIndex + threadIdx.x];
```



```
struct vector2 { float f; int c; };
__shared__ struct vector2 v2[64];
```

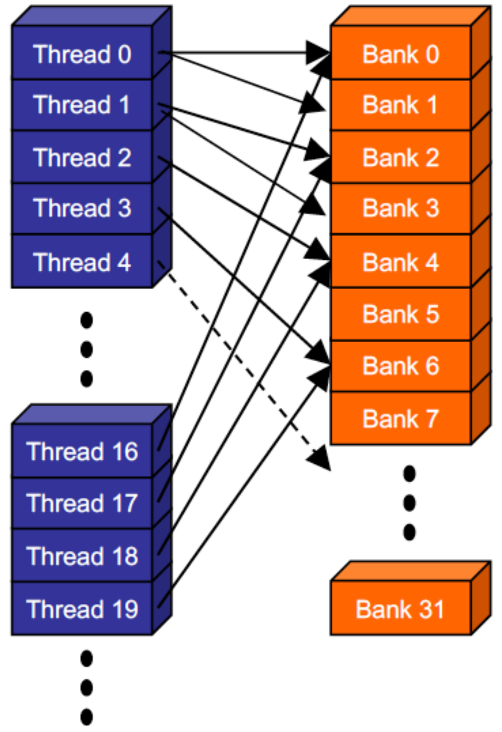
- This has 2-way bank conflicts for `vector2` (2 words)
  - 2 accesses per thread

```
struct vector2 v = v2[baseIndex + threadIdx.x];
```

Common array bank conflict patterns 1D

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts:

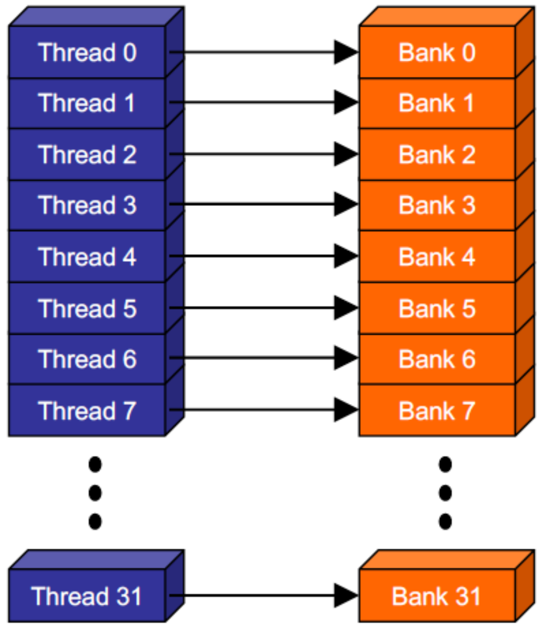
```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```



- This makes sense for traditional CPU threads
  - Locality in cache line usage and reduced sharing traffic.
  - But, not in shared memory usage
    - There are no cache line effects but banking effects

A better array access pattern

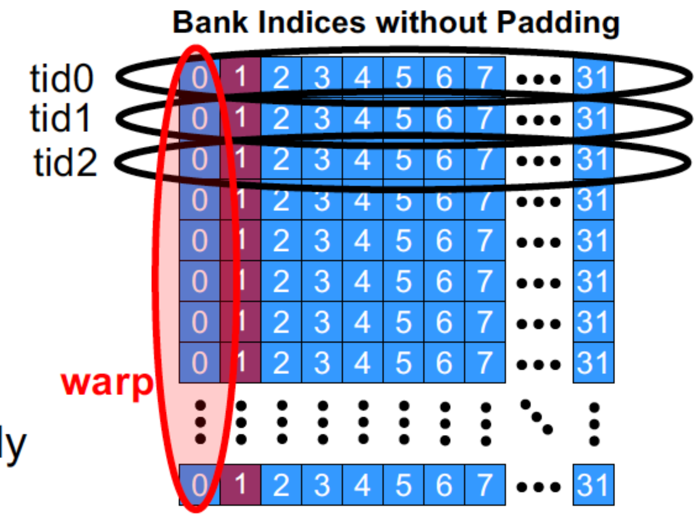
- Each thread loads one element in every consecutive group of blockDim elements.

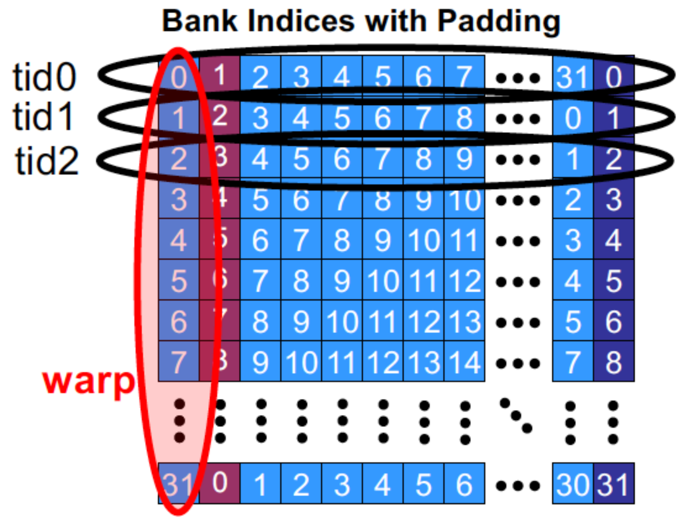


```
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```

Common bank conflict patterns (2D)

- Operating on 2D array of floats in shared memory
  - e.g., image processing
- Example: 32x32 block
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously (example: row 1 in purple)
  - 32-way bank conflicts: rows all start at bank 0
- Solution 1. Padding
  - Add one float to the end of each row



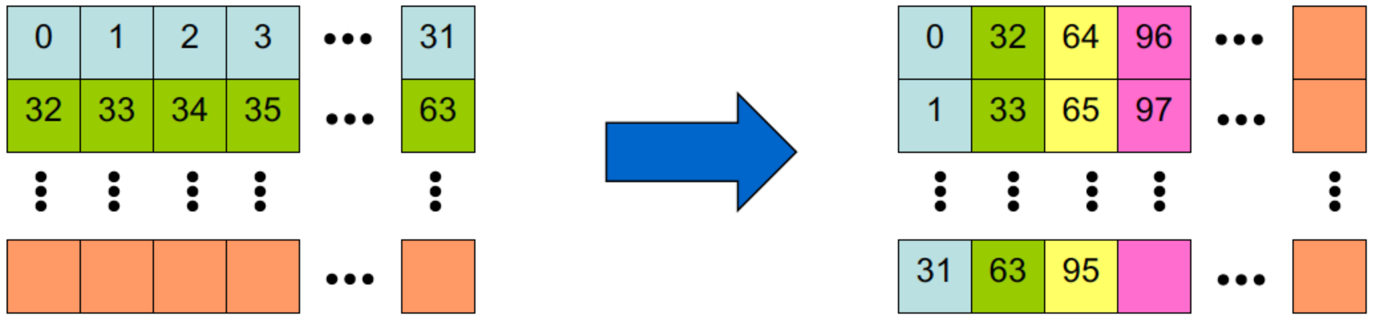


Matrix transpose example

Solution 2. Transpose before processing

- Suffer bank conflicts during transpose
- But possibly save them later

- Avoiding shared memory bank conflicts

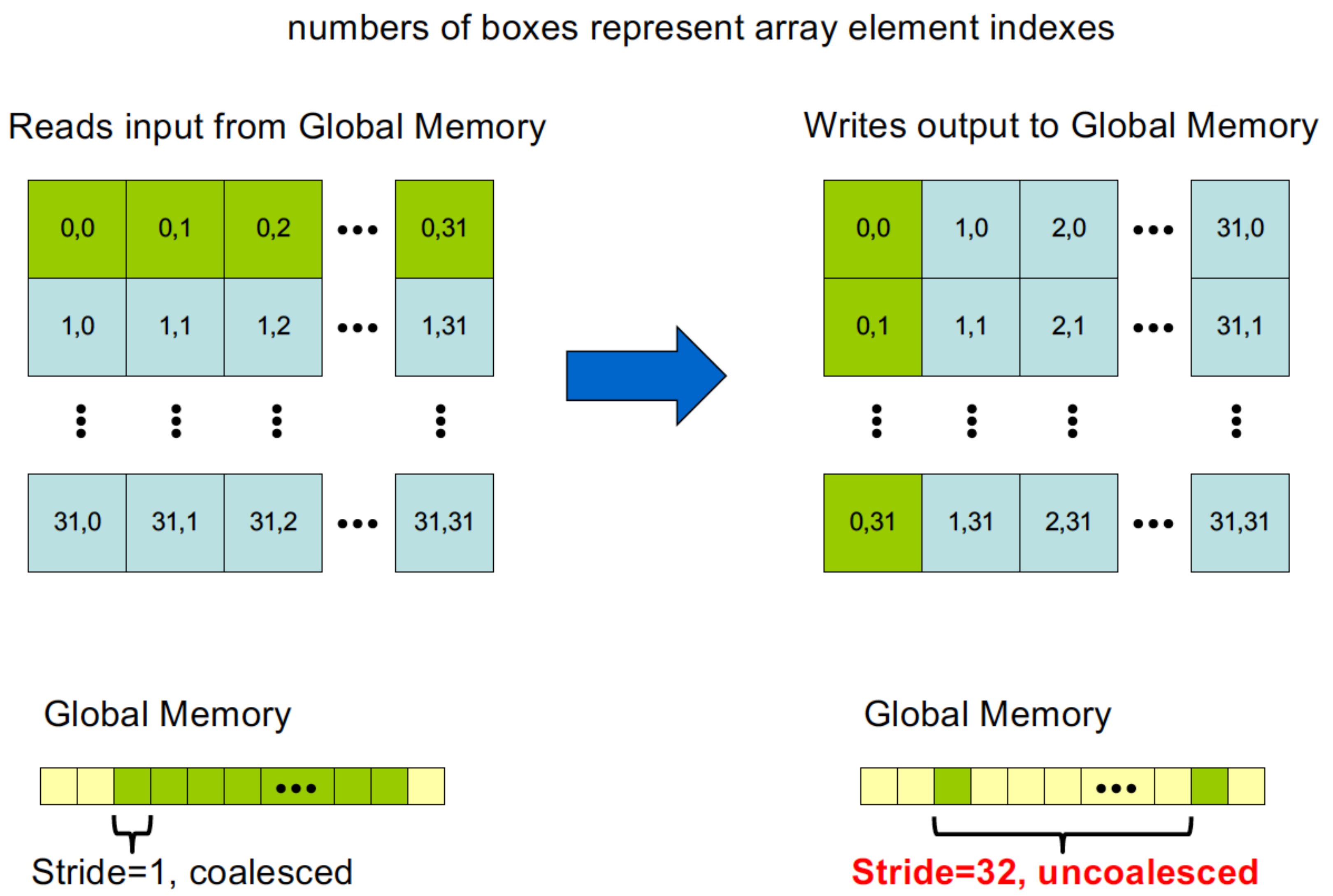


numbers indicate array element

Uncoalesced transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

  if (xIndex < width && yIndex < height)
  {
     unsigned int index_in   = xIndex + width * yIndex;
     unsigned int index_out = yIndex + height * xIndex;
     odata[index_out] = idata[index_in];
  }
}
```

Uncoalesced transpose: memory access pattern



Coalesced transpose

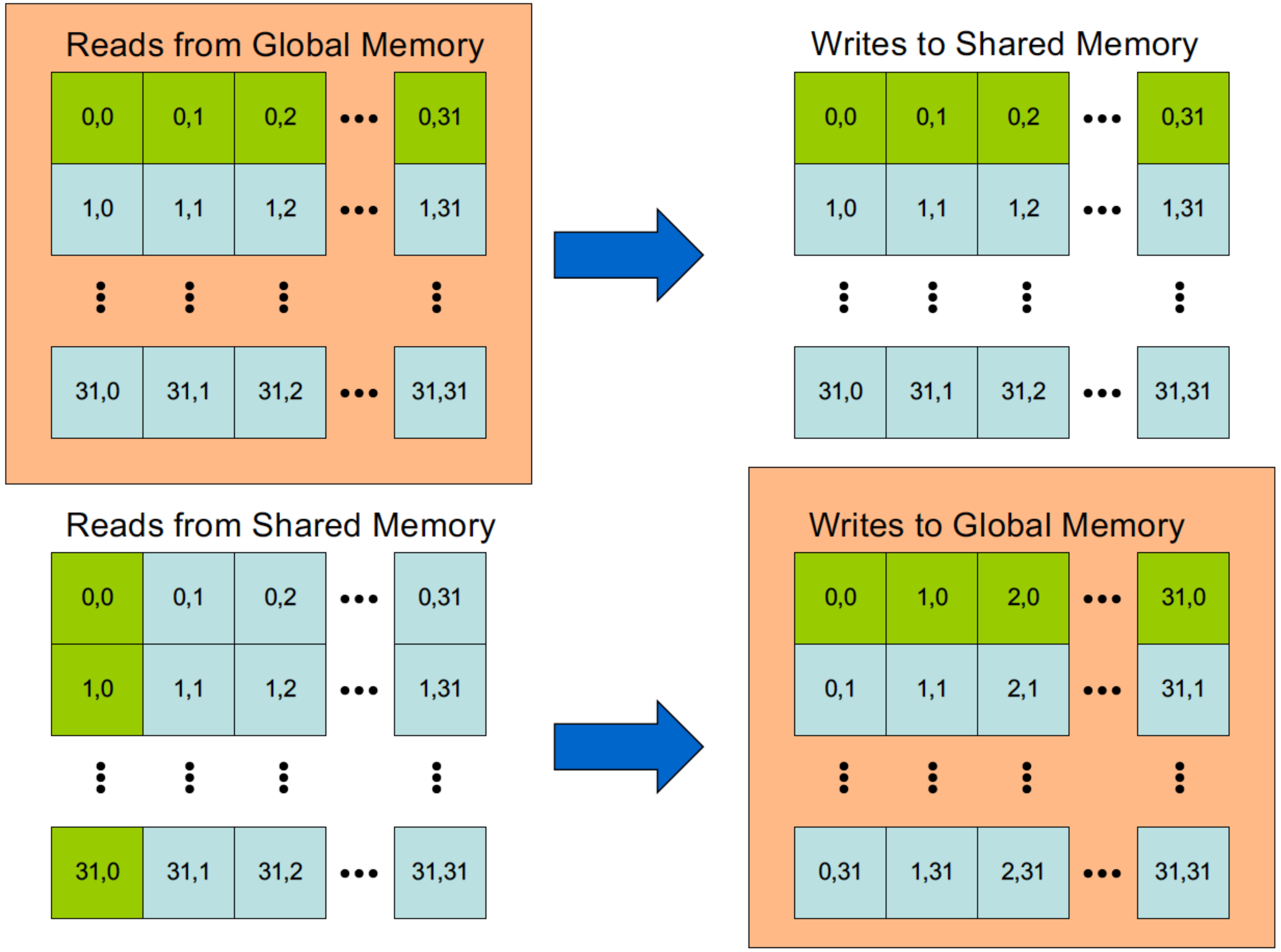
- Conceptually partition the input matrix into square tiles

- Threadblock (bx, by):
  - Read the (bx,by) input tile, store into SMEM
  - Write the SMEM data to (by,bx) output tile
    - Transpose the indexing into SMEM

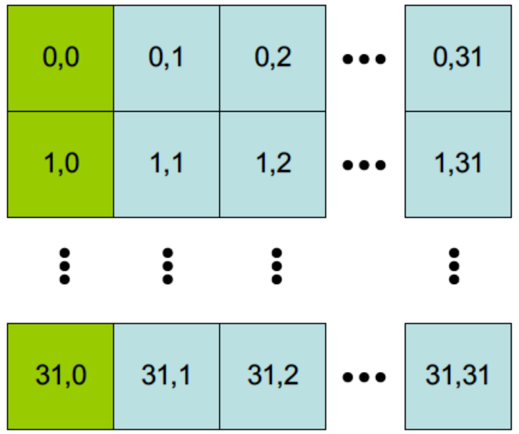
- Thread (tx,ty):
  - Reads element (tx,ty) from input tile
  - Writes element (tx,ty) into output tile

- Coalescing is achieved if:
  - Block/tile dimensions are multiples of 32

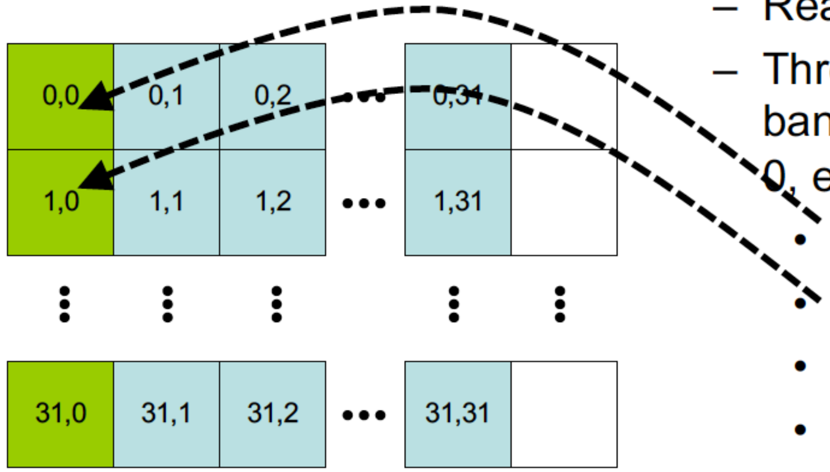
Coalesced transpose: access patterns



Avoiding bank conflicts in shared memory



- Threads read SMEM with stride
  - 32-way bank conflicts
  - 32x slower than no conflicts



- Solution: Padding
  Allocate an "extra" column
  - Read stride = 33 elements
  - Threads read from consecutive banks. Assuming starting address is 0, element size is 4 (float):
    - ((33*4)*0)/4 mod 32 → bank 0
    - ((33*4)*1)/4 mod 32 → bank 1
    - ((33*4)*2)/4 mod 32 → bank 2
    - …
    - remember: bank = (addr/4) % 32

Coalesced transpose

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
  __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

  unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
  unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;

  if (xIndex < width && yIndex < height)
  {
      unsigned int index_in = __mul24(width, yIndex) + xIndex;
      unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;
      block[index_block] = idata[index_in];
      index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
      index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
  }
  __syncthreads();

  if (xIndex < width && yIndex < height)
      odata[index_out] = block[index_transpose];
}
```

Transpose measurements

- Average over 10K runs
- 32x32 blocks

- 128x128 array
  - Optimized vs Simple: 1.3x

- 512x512 array
  - Optimized vs Simple: 8x

- 1024x1024 array
  - Optimized vs Sample: 10x

Transpose detail

- 512x512 array, 32x32 blocks

- Simple: t=8
- Optimized w/ shared memory: t=4
- Optimized w/ shared memory & padding: t=1

Data types and bank conflicts

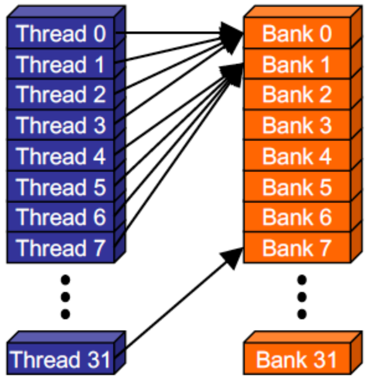
- This has no conflicts if type of shared is 32-bits (e.g., int, float):

```
foo = shared[baseIndex + threadIdx.x]
```

- If **reads** fall within same 32-bit word
  → No conflicts, broadcast !

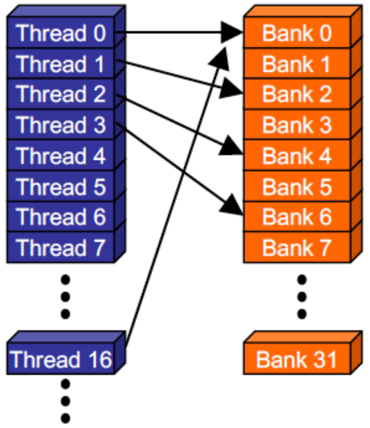
- Assume the array is 4-byte aligned:

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```



- 2-way conflicts for larger types:

```
__shared__ double shared[];
foo = shared[baseIndex + threadIdx.x];
```


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 16
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 31