CS302

GPU: Memory part II and Parallel Reduction

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Some of the slides are from Derek R Hower, Adwait Jog, Wen-Mei Hwu, Steve Lumetta, Babak Falsafi, Andreas Moshovos, and from the companion material of the book “Programming Massively Parallel Processors”

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- This lecture
  - GPU memory part II
  - Use case study: Optimizing CUDA program
- Thursday exercise session
  - Profiling CUDA programs
- Next class
  - Advanced synchronization

Review: CPU vs. GPU memory system

CPU

- Limited number of registers
  - OS saves/restores registers
- One virtual address space (per process)
- Large caches for latency hiding

GPU

- Many registers
  - H/W saves/restores registers
- Many different address spaces
  - Global memory
    - Caches
  - Local memory
  - Constant memory
  - Shared memory (scratchpad)
- Small caches – for b/w filtering

Review: CPU vs. GPU memory system

CPU

- Limited number of registers
  - OS saves/restores registers
- One virtual address space (per process)
- Large caches for latency hiding

GPU

- Many registers
  - H/W saves/restores registers
- Many different address spaces
  - Global memory
    - Caches
  - Local memory
  - Constant memory
  - Shared memory (scratchpad)
- Small caches – for b/w filtering

GPU Memory Hierarchy: Local Memory

- Local memory (per thread):
  - Private to each GPU thread
    - Each thread can have different data for the same virtual address in local memory
  - 24-bit virtual address space vs. 48-bit Global memory address space
  - Everything on the stack that can't fit in registers
    - Register spilling
    - Allocation of array variables
  - Physically stored in global memory
    - Local memory is a logical address space
    - Same latency as global memory
    - Accesses to the local memory are **always coalesced**
    - Much slower than registers!

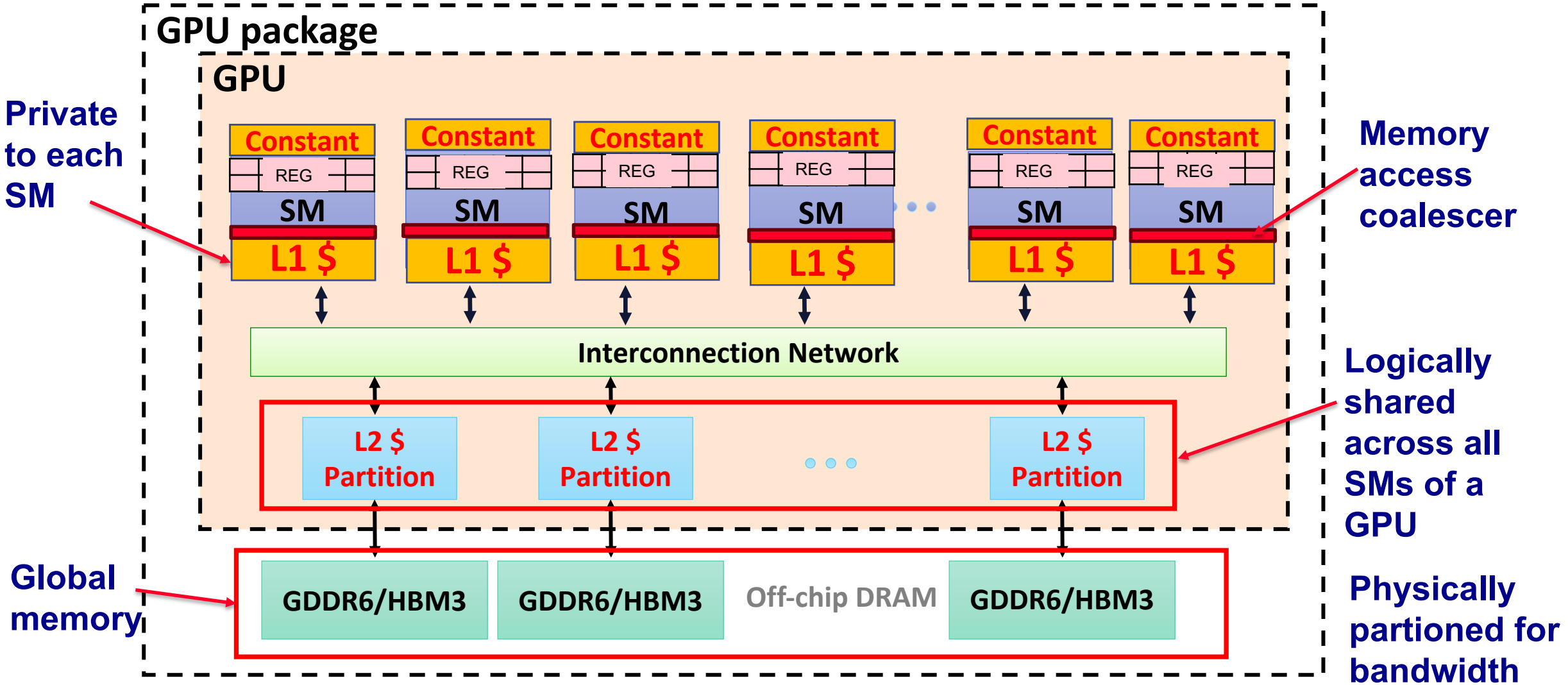

```
__global__ void example(char* global_mem) {
    char local_mem[10];
    local_mem = global_mem;
}
```

GPU Memory Hierarchy: Constant Memory

- A special type of memory allocation for data that won't be written to, i.e., constant
  - Immutable during the execution of the kernel

- Hardware cache contents of constant memory in a specialized, fast cache
  - No need for write ports
  - No need for write-back data in the constant cache

Constant Cache



Allocating Constant Memory

- Declare constant memory array as global variable → accessible to all threads of a kernel

```
__constant__ float filter_c[FILTER_DIM][FILTER_DIM];
```

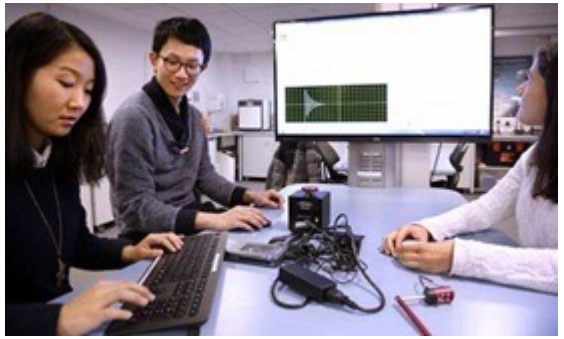
- Must initialize constant memory from the host
  - Cannot be modifies during a kernel's execution

```
cudaMemcpyToSymbol(filter_c, filter, FILTER_DIM*FILTER_DIM*sizeof(float));
```

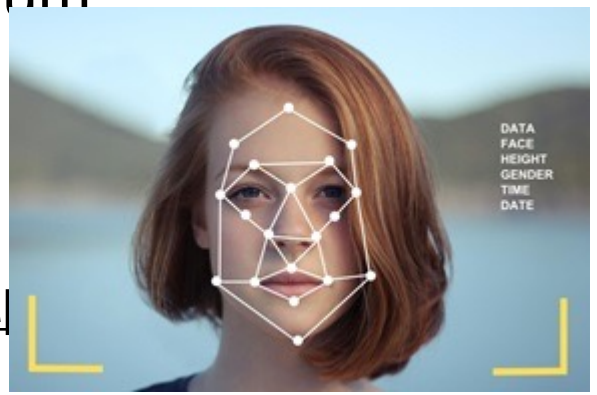
- Can only allocate up to 64KB
  - Otherwise, input is also constant, but it is too large to put in constant memory

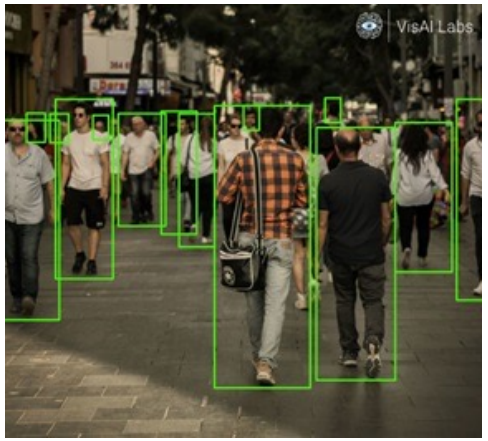
Example use of Constant Memory: Convolution

- Convolution: Widely used operation in video processing, signal processing, CNNs etc.

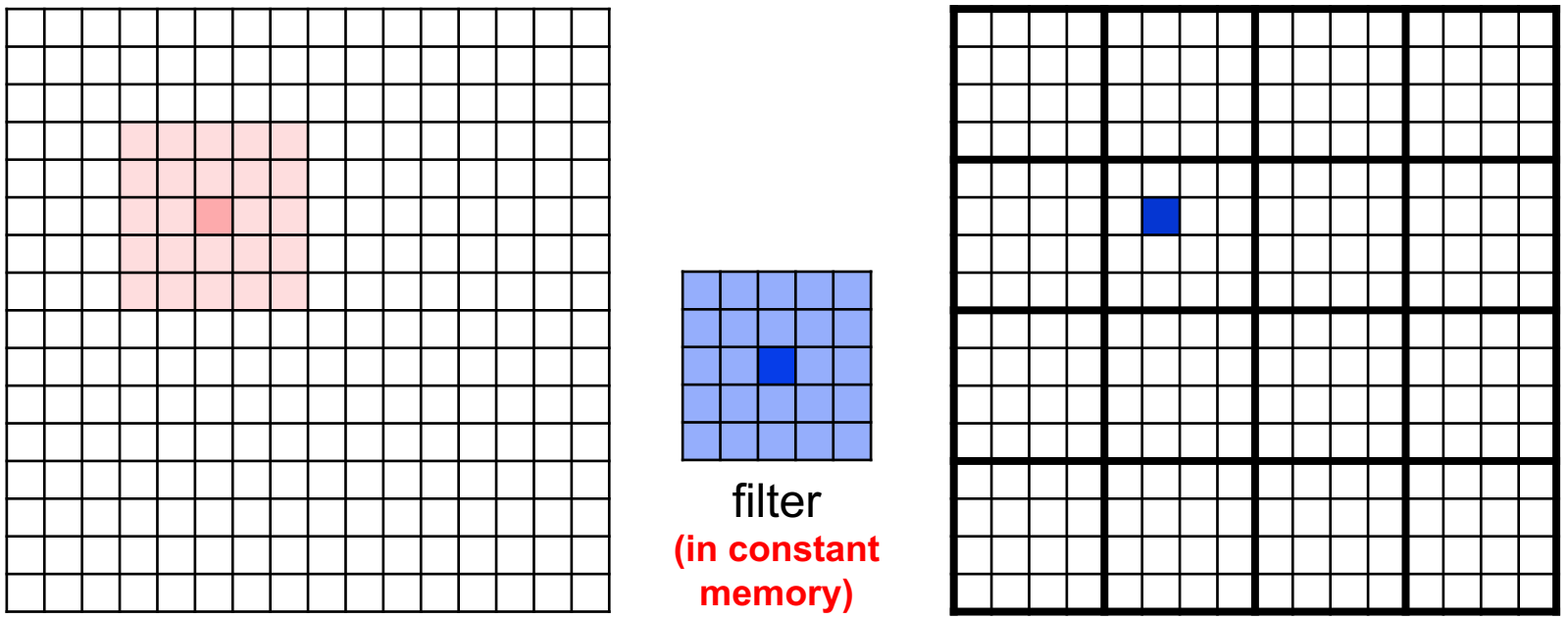


- Converts an array (1D/2D/3D) into another array
  - Each output element is a weighted sum of nearby elements
  - Weights are in a filter array
  - Weights are **constant and are needed to calculate every out**



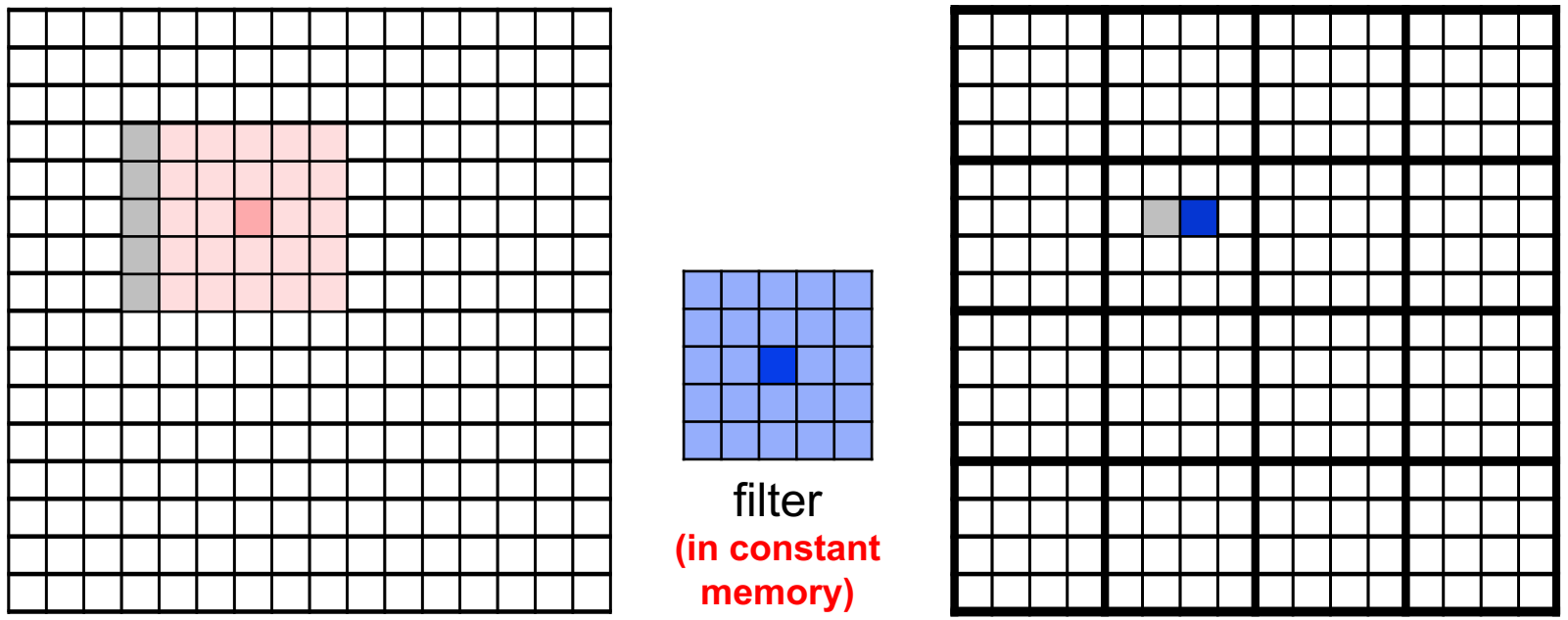


Example use case of Constant Memory: Convolution



An **output element** computed by one thread looping over **input elements** and **filter** weights

Example use case of Constant Memory: Convolution



An **output element** computed by one thread looping over **input elements** and **filter** weights

Example use case of Constant Memory: Convolution

```
static __constant__ float filter_c[FILTER_DIM][FILTER_DIM];



//Allocate input and output arrays using cudaMalloc

//Populate input array

cudaMemcpyToSymbol(filter_c, filter, FILTER_DIM*FILTER_DIM*sizeof(float));

convolution_kernel<<dimGrid, dimBloc>> (input_array, output_array, x_dim, y_dim);
```

Host code

Example use case of Constant Memory: Convolution

```
__global__ void convolution_kernel(float* input, float* output, unsigned int width,
                                                                unsigned int height) {
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    if (outRow < height && outCol < width) {
        float sum = 0.0f;
        for(int filterRow = 0; filterRow < FILTER_DIM; ++filterRow) {
            for(int filterCol = 0; filterCol < FILTER_DIM; ++filterCol) {
                int inRow = outRow - FILTER_RADIUS + filterRow;
                int inCol = outCol - FILTER_RADIUS + filterCol;
                if(inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                    sum += filter_c[filterRow][filterCol]*input[inRow*width + inCol];
                }
            }
        }
        output[outRow*width + outCol] = sum;
    }

}
```

Device code

Review: CPU vs. GPU memory system

CPU

- Limited number of registers
  - OS saves/restores registers
- One virtual address space (per process)
- Large caches for latency hiding

GPU

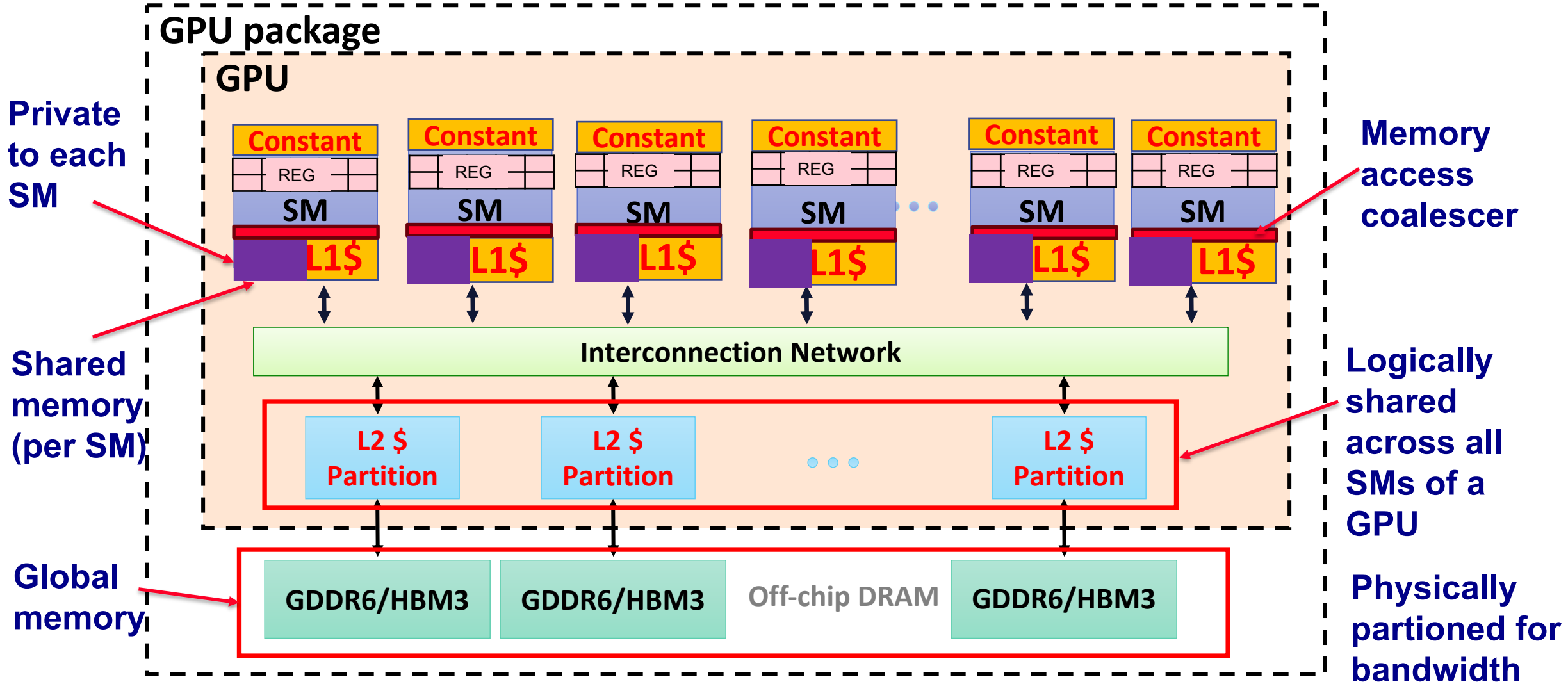
- Many registers
  - H/W saves/restores registers
- Many different address spaces
  - Global memory
    - Caches
  - Local memory
  - Constant memory
  - Shared memory (scratchpad)
- Small caches – for b/w filtering

Shared memory (Scratchpad): A S/W Managed Cache

- H/W Managed Cache (Traditional Caches):
  - Hardware decides what to bring into the cache and what to evict
  - Hardware monitors to access patterns for decision-making
  - Example: L1, L2 caches
  - Pros: No need to modify applications to use caches
  - Cons: Lack of high-level program semantic knowledge

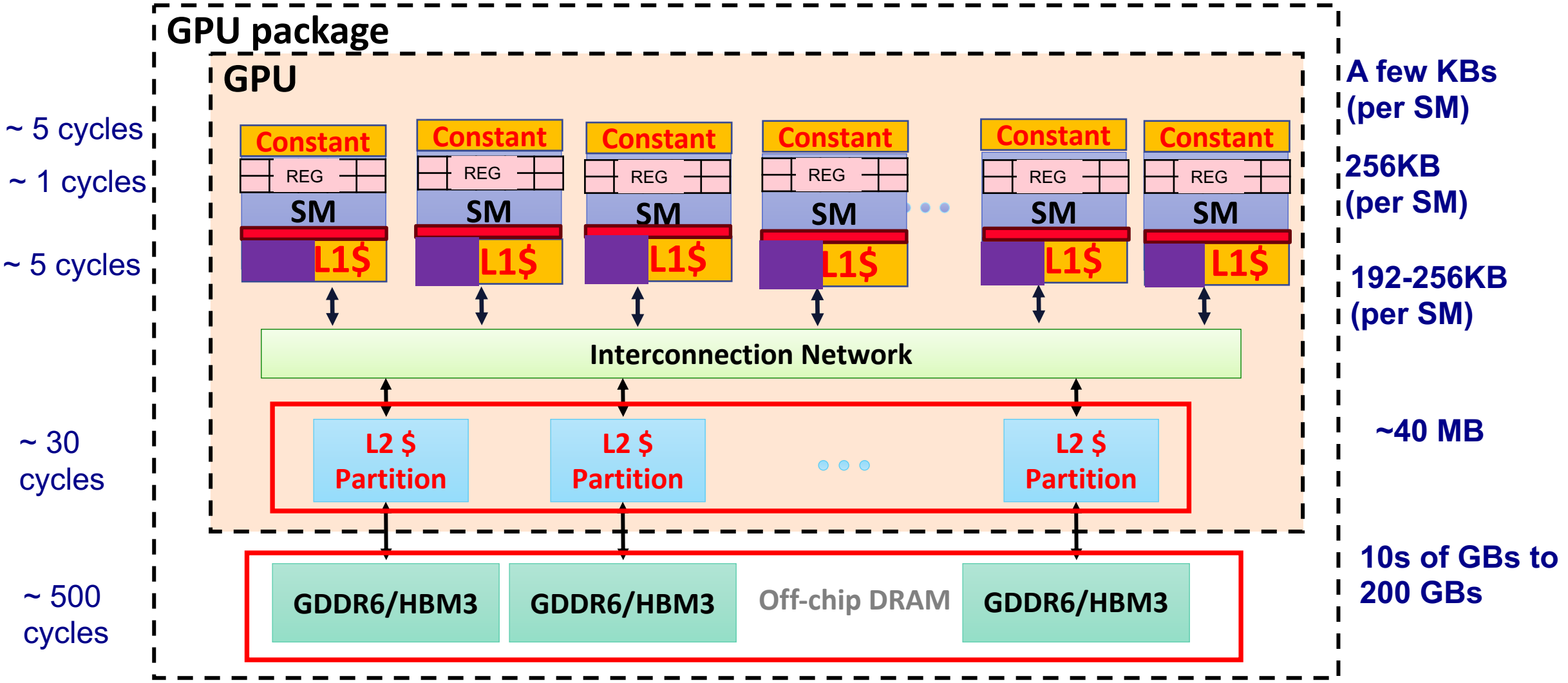
- S/W Managed Cache:
  - On-chip hardware cache structure
  - Software (program) must direct what to bring into the cache and what to evict
  - Use a program's semantic behaviour to decide cache contents
  - Example: GPU scratchpad/shared memory
  - Pros: Use program knowledge for accurate caching
  - Cons: Applications must be modified to use such caches

Shared memory (Scratchpad): A S/W Managed (H/W) Cache



Shared memory/Scratchpad shares the same physical structure as L1 $

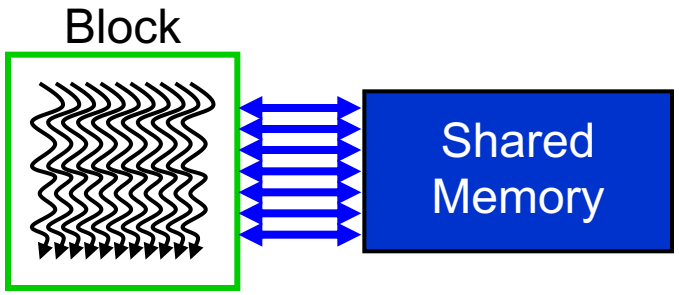
Relative Latencies of Various Memory Types



 Shared memory/Scratchpad shares the same physical structure as L1 $

Shared memory (Scratchpad): A S/W Managed Cache

- Private to each threadblock
  - Called “Shared” memory because it is shared amongst all threads if a threadblock



- Each threadblock has its own copy of shared memory variables
  - # of copies of each shared memory variable allocated == # of threadblocks
- Contents deleted after a threadblock finishes execution

Steps of using Shared Memory (Scratchpad)

1. Allocate shared memory

2. Load data into shared memory from global memory

3. Repeatedly access data from shared memory → Leverage reuse
   - Use of shared memory is useful only if there is significant data reuse
   - Programmer must be aware of the data reuse

4. Once computation is done, write results back to global memory

Allocating and Populating Shared Memory

- Two ways to allocate shared memory

- Static allocation: Allocation size fixed at compile time

```
__shared__ float in_s[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];

if(i >= 0 && i < N && j >= 0 && j < N && k >= 0 && k < N) {
        in_s[threadIdx.z][threadIdx.y][threadIdx.x] = in[i*N*N + j*N + k];
    }
```

- Cons: Maximum up to 48KB per threadblock
  The size of the allocation cannot be input-dependent

Allocating and Populating Shared Memory

- Dynamic allocation: Size of the allocation set during kernel launch
  - Can allocate **one** chunk (one variable) → partition if needed
  - Pass the size of the chunk during kernel launch
  - Can be larger than 48KB

```
//device code
__global__ void alloc_shared () {

extern __shared__ float in_s[];
float *p1_v = &in_s[0];
float *p2_v = &in_s[N]
…….
}

//host code
……….
alloc_shared <<<nBlk, nTh, (2*N*sizeof(float))>>>
```

Allocaiton size not specified statically

Size (per threadblock) provided during kernel launch time

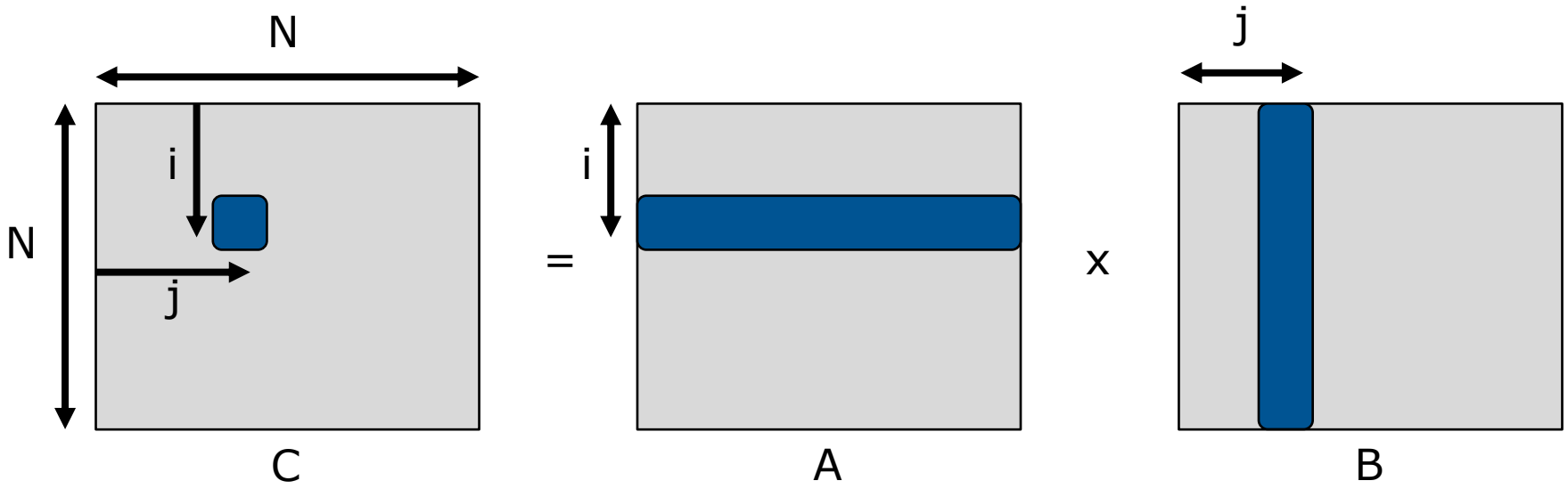
Configuring Shared Memory and L1 Cache sizes

- Can provide hints to the CUDA runtime on preference of L1 cache vs. Shared memory

cudaFuncSetCacheConfig (kernel_name, enum cacheConfig)

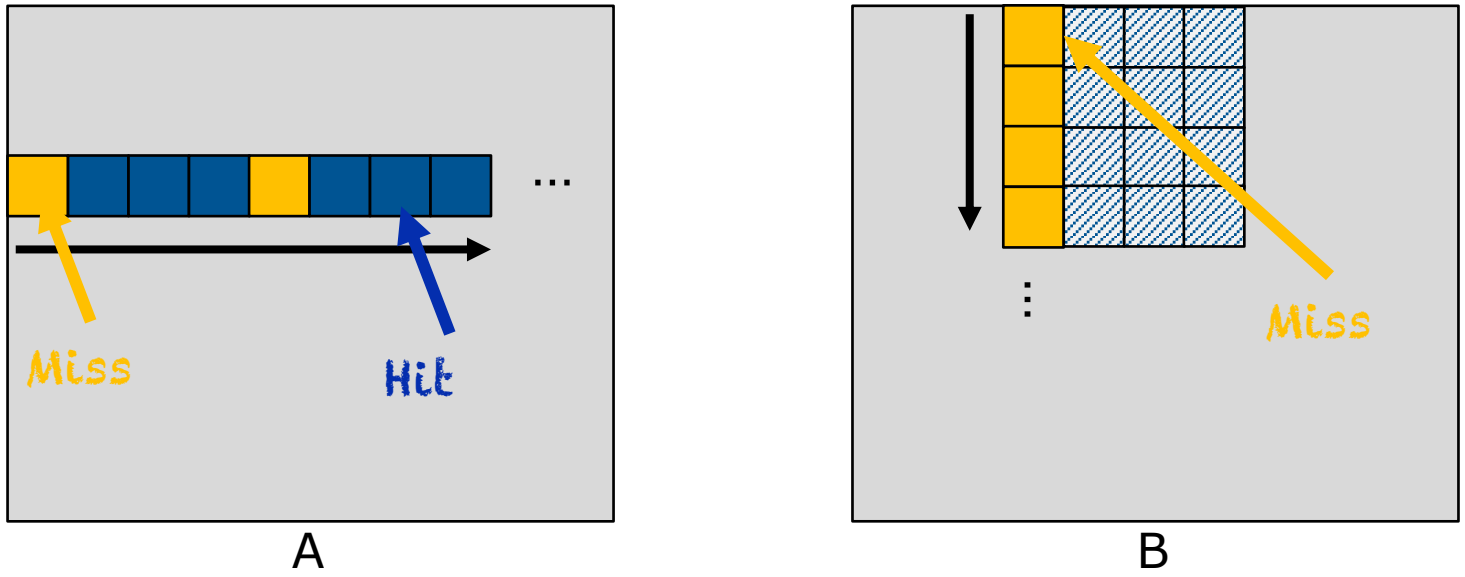
- Example cacheConfig options:
  - cudaFuncCachePreferShared
  - *cudaFuncCachePreferL1*
  - *cudaFuncCachePreferNone*

Example use of Shared Memory: Titled Matrix Multiplication



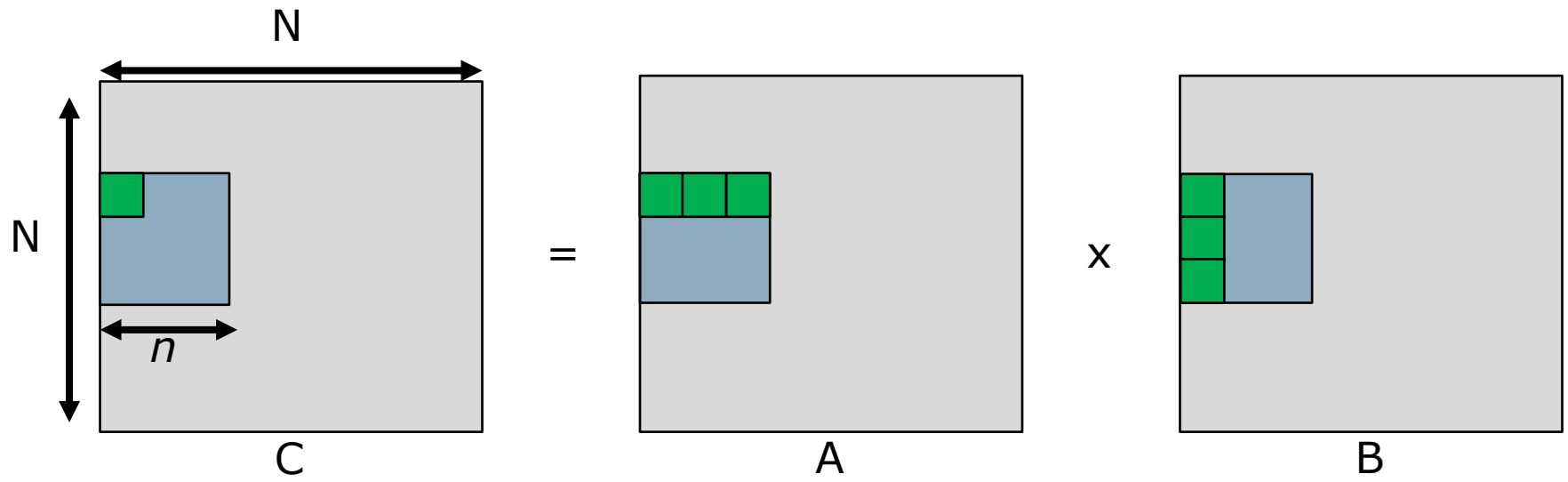
- Review (Week 3): Naïve matrix multiplication causes too many $ miss

Example use of Shared Memory: Titled Matrix Multiplication



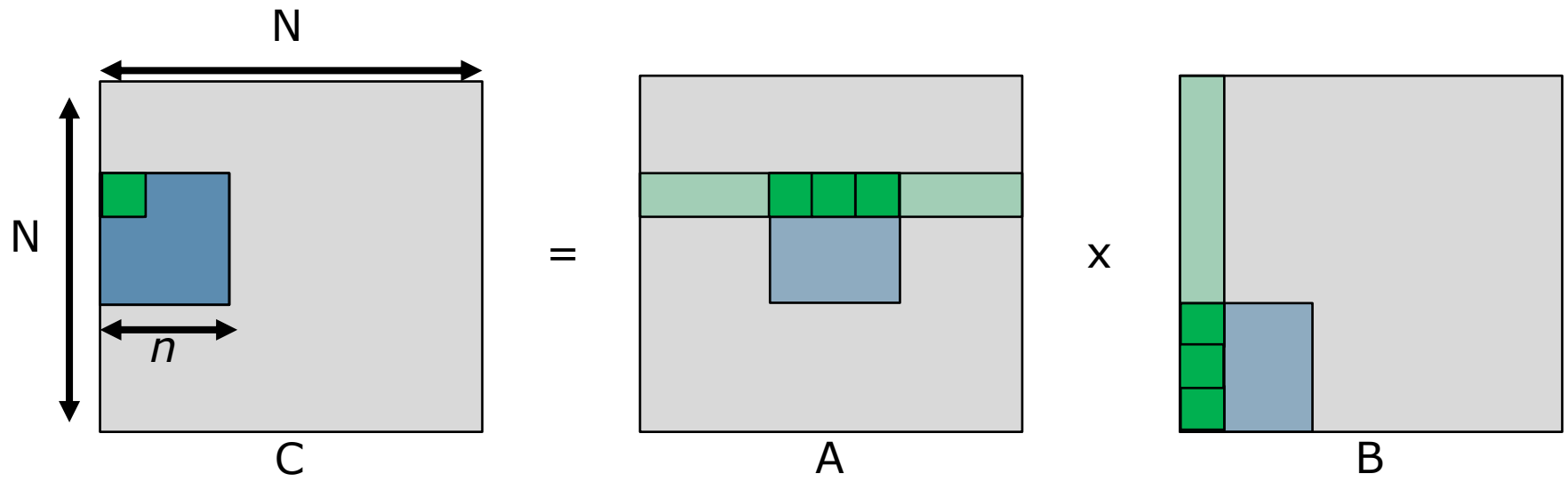
- Review (Week 3): Naïve matrix multiplication causes too many $ miss

Example use of Shared Memory: Titled Matrix Multiplication



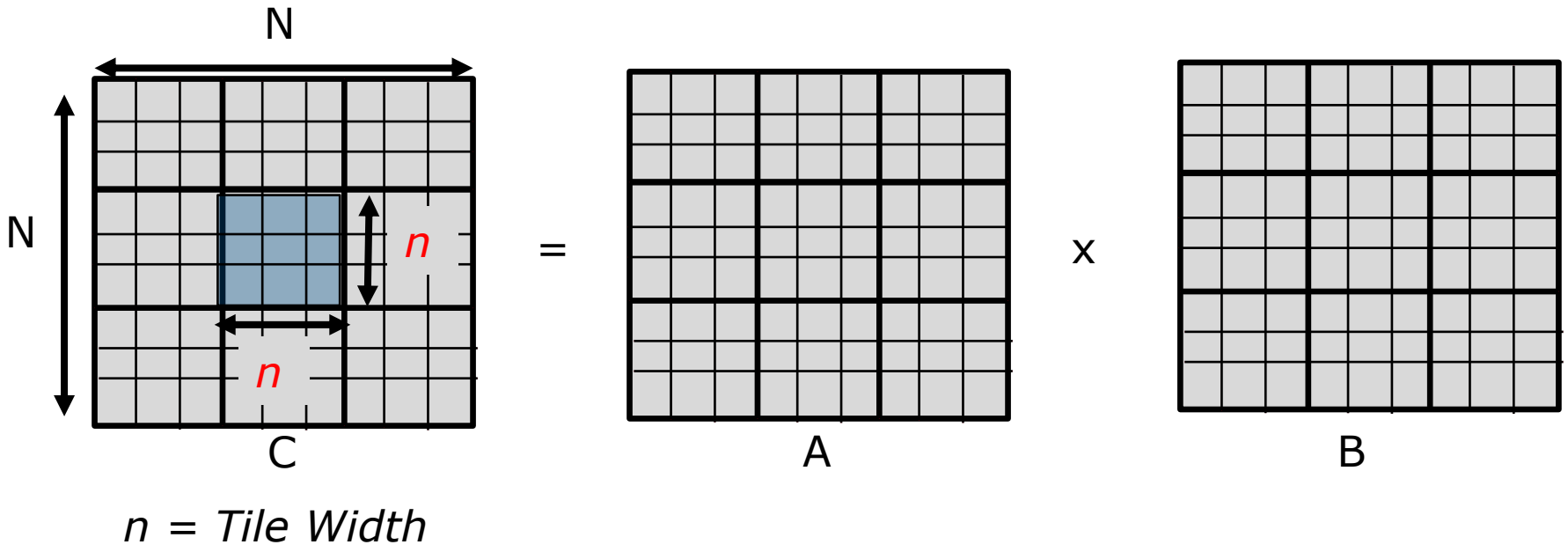
- Review (Week 3): Tiling to improve data reuse → less cache misses
  - Compute the partial results of each submatrices (tile) before moving to the next

Example use of Shared Memory: Titled Matrix Multiplication



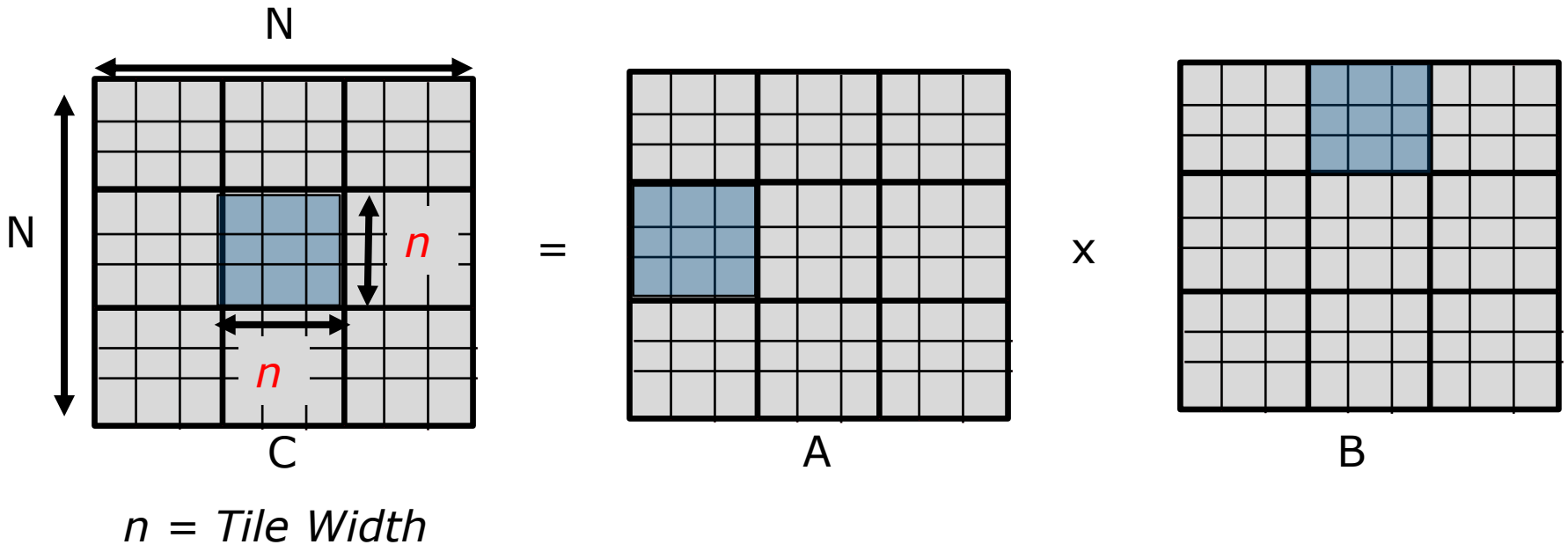
- Review (Week 3): Tiling to improve data reuse → less cache misses
  - Compute the partial results of each submatrices (tile) before moving to the next
  - Limitations: Cache hits are not guaranteed.
    L1 cache is shared amongst all threadblocks of a SM
    L2 cache is shared amongst all threadblocks of a grid

Example use of Shared Memory: Titled Matrix Multiplication



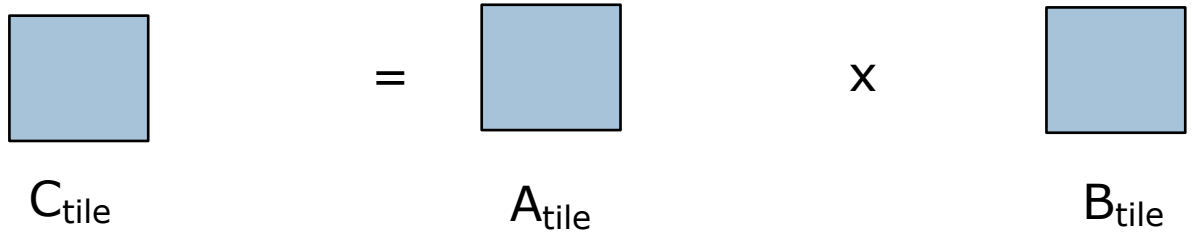
- One threadblock to compute results of one tile of the output matrix

Example use of Shared Memory: Titled Matrix Multiplication



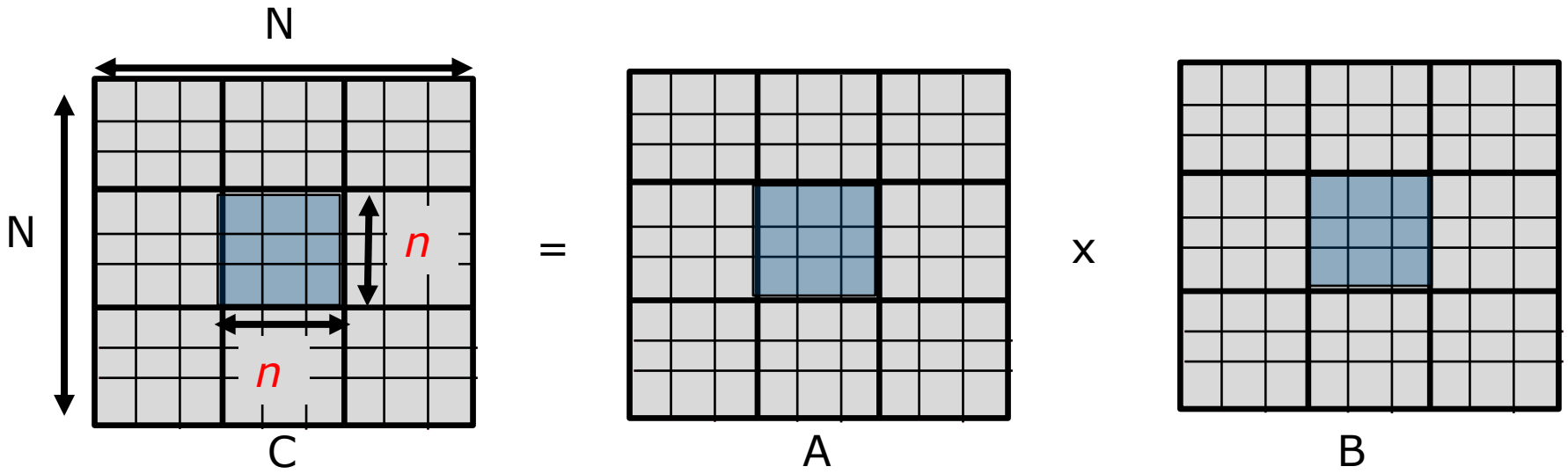
- Threads of a threadblock load one tile each from two operand matrices to its copy of shared memory variables

Example use of Shared Memory: Titled Matrix Multiplication



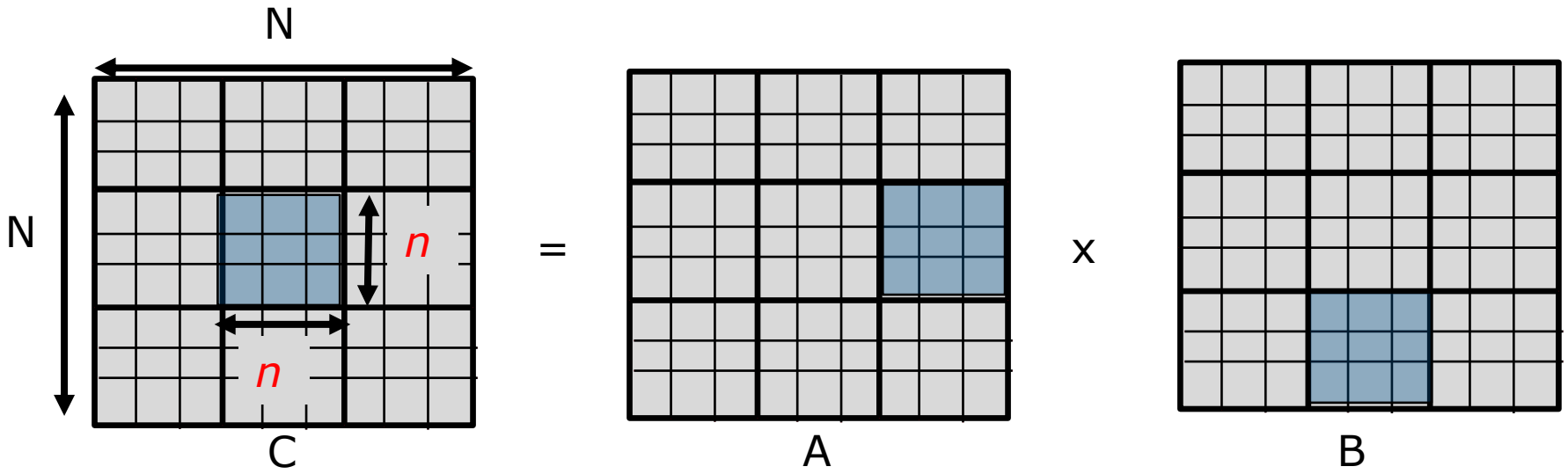
- Threads of a threadblocks computes partial results for a tile

Example use of Shared Memory: Titled Matrix Multiplication



- Repeat for next tiles ..

Example use of Shared Memory: Titled Matrix Multiplication



- Repeat for next tiles

Example use of Shared Memory: Tiled Matrix Multiplication

```
__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];

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

float sum = 0.0f;

for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {

    // Load tile to shared memory
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];

     __syncthreads();

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();
}

C[row*N + col] = sum;
```

// Each threadblock his its copy of shared mem variables

// The element of the output matrix to be computed by a given thread

// Repeat N/TILE_DIM (# phases)

//Loads a tile to shared memory from global mem.

//Wait for loading to be compete

//Each thread calculates partial result

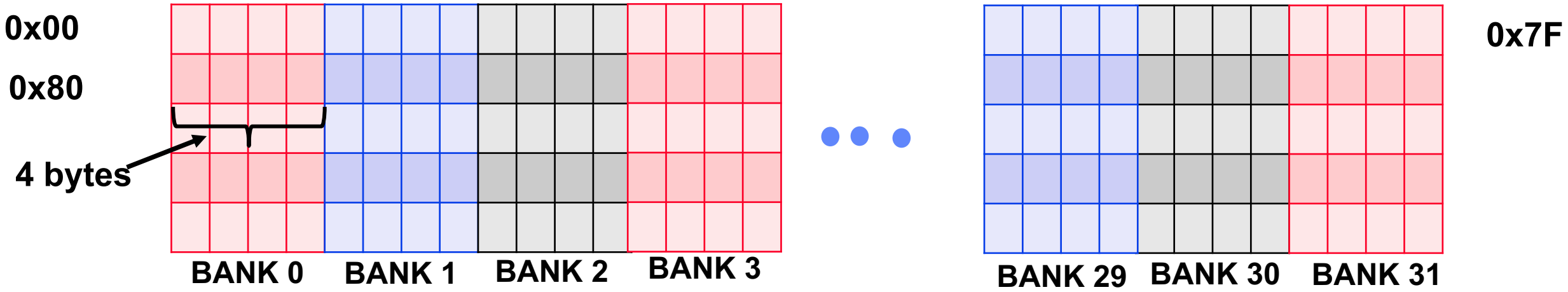
//Wait for threads in a threadblock to complete computation before next round/phase

//Final result of individual output element

Advantage of using Shared Memory

- Saves global memory access
- Access to shared memory is as fast as L1 cache
- Unlike caches, reductions in global memory access are guaranteed
  - Not dependent on hardware's cache replacement policies
- Every threadblock has its own copy of shared memory variables

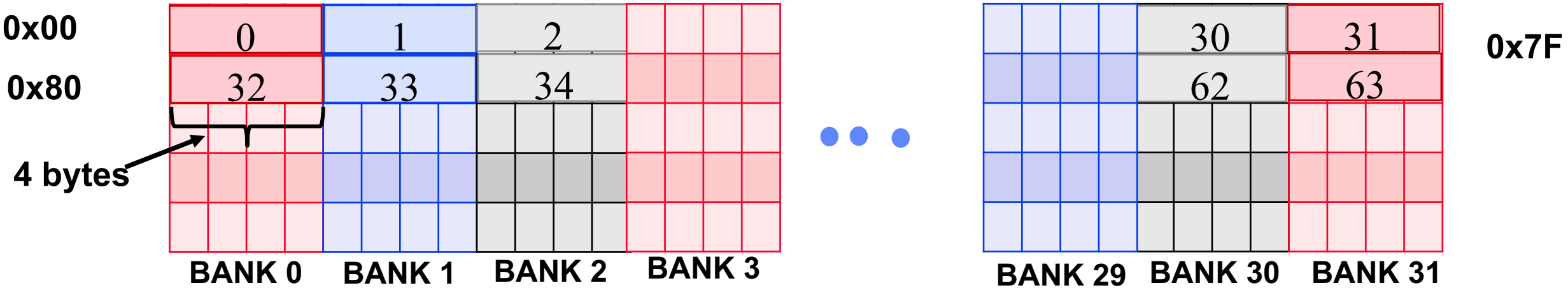
Performance pitfall: Beware of Shared Memory Bank Conflicts



- Shared memory is organized in 32 banks
  - Each bank can service one address at a time
  - At most 32 simultaneous accesses
- Successive 32-bit (4 bytes) words are assigned to successive banks

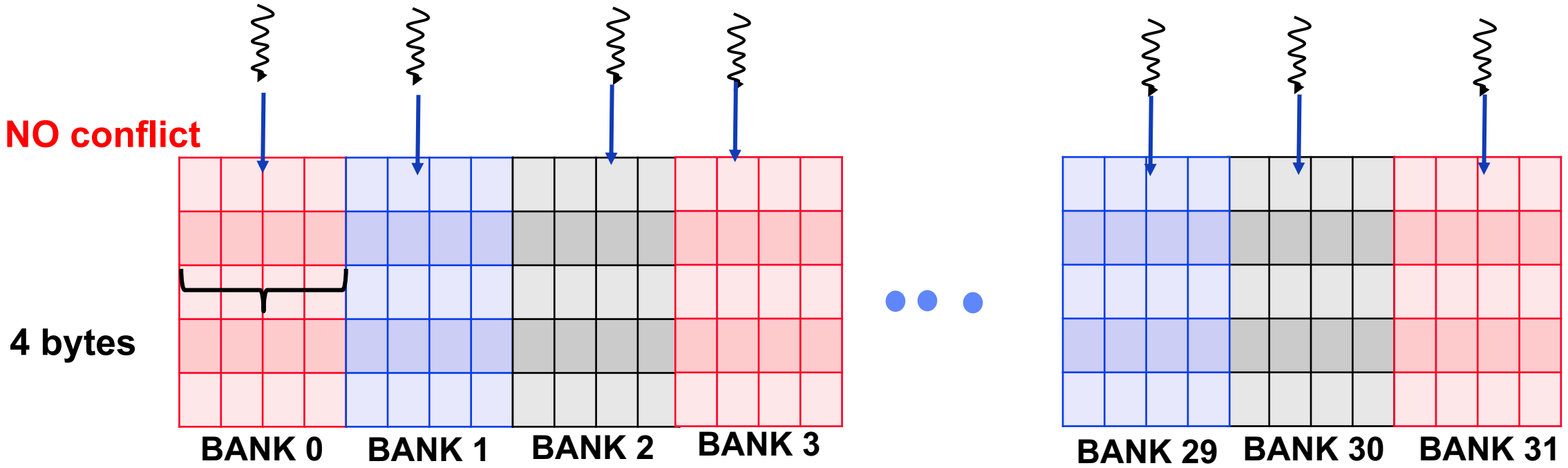
Performance pitfall: Beware of Shared Memory Bank Conflicts

__shared int arr[64]



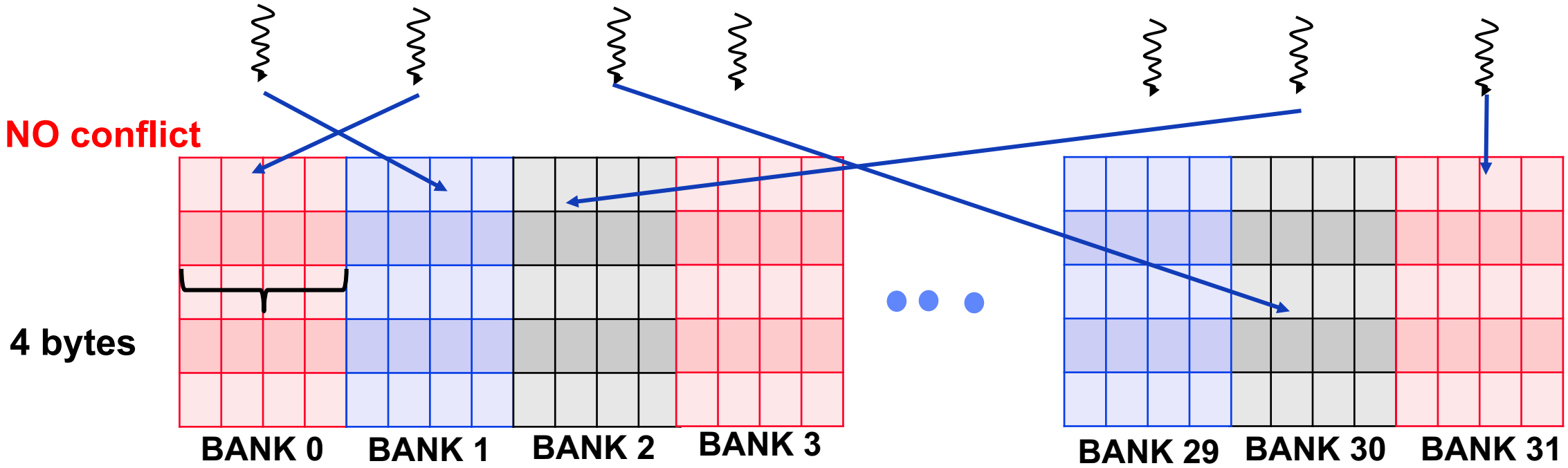
- Shared memory is organized in 32 banks
  - Each bank can service one address at a time
  - At most 32 simultaneous accesses
- Successive 32-bit (4 bytes) words are assigned to successive banks

Performance pitfall: Beware of Shared Memory Bank Conflicts



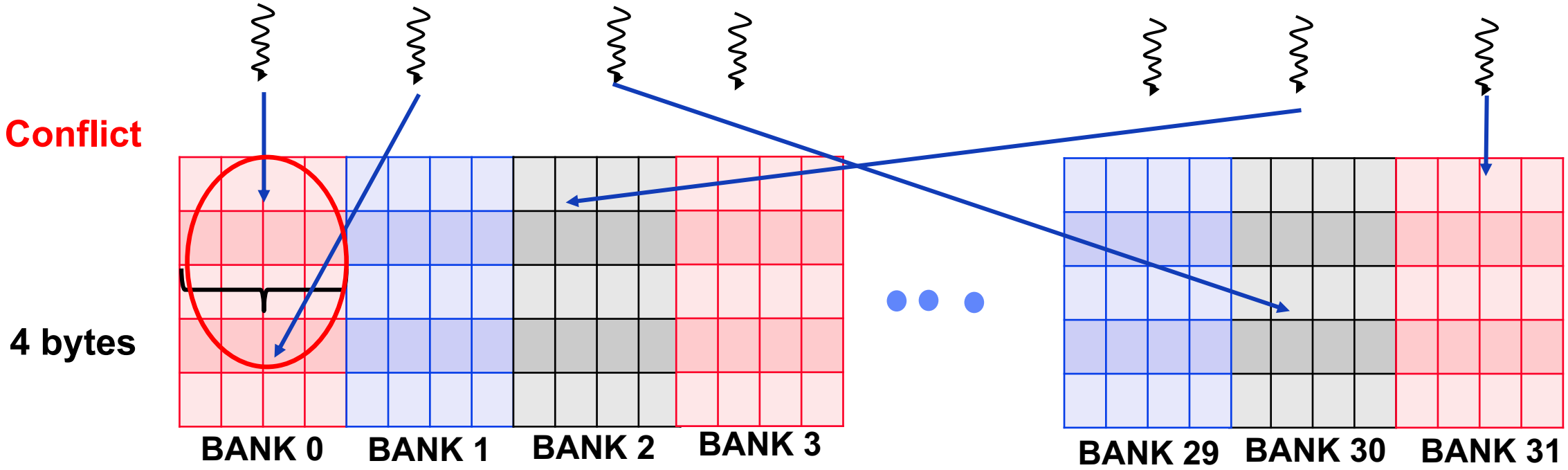
- Multiple simultaneous accesses to the same bank
  - Different 4-byte words:
    - Bank conflict! - Conflicting accesses are serialized
  - The same 4-byte word:
    - Multicast – 1 fetch (could be different bytes within the word)

Performance pitfall: Beware of Shared Memory Bank Conflicts



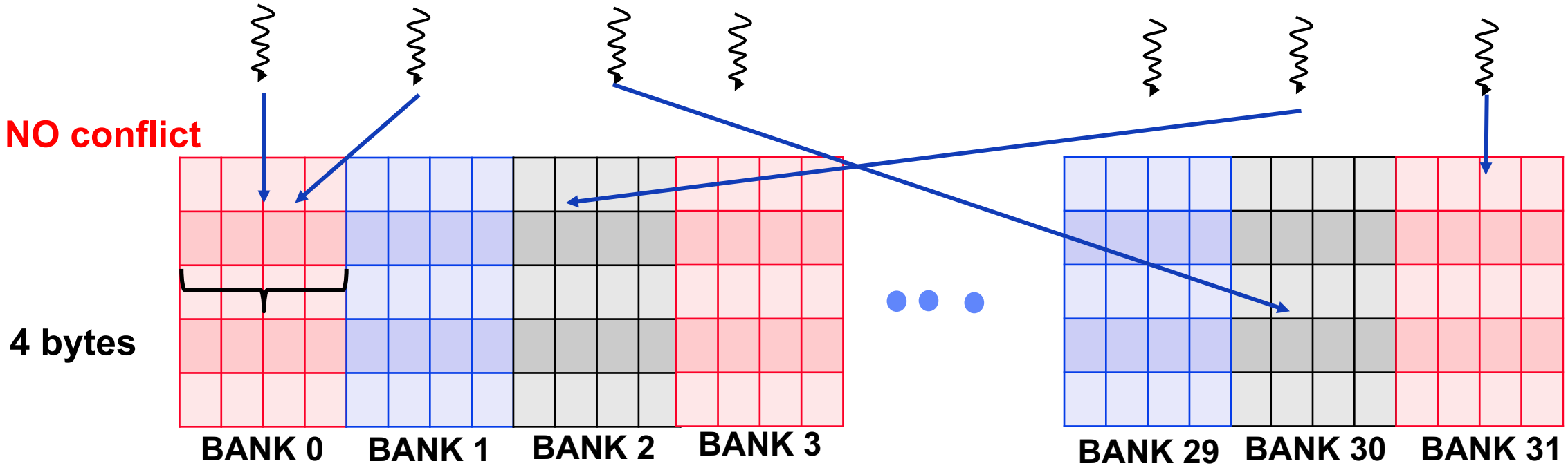
- Multiple simultaneous accesses to the same bank
  - Different 4-byte words:
    - Bank conflict! - Conflicting accesses are serialized
  - The same 4-byte word:
    - Multicast – 1 fetch (could be different bytes within the word)

Performance pitfall: Beware of Shared Memory Bank Conflicts



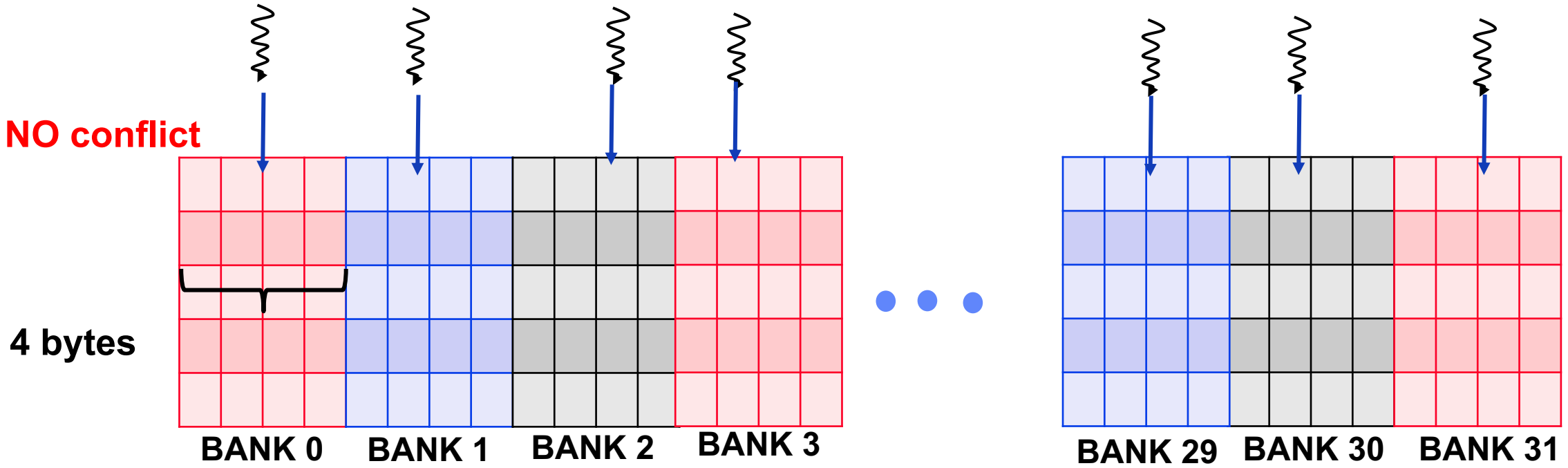
- Multiple simultaneous accesses to the same bank
  - Different 4-byte words:
    - Bank conflict! - Conflicting accesses are serialized
  - The same 4-byte word:
    - Multicast – 1 fetch (could be different bytes within the word)

Performance pitfall: Beware of Shared Memory Bank Conflicts



- Multiple simultaneous accesses to the same bank
  - Different 4-byte words:
    - Bank conflict! - Conflicting accesses are serialized
  - The same 4-byte word:
    - Multicast – 1 fetch (could be different bytes within the word)

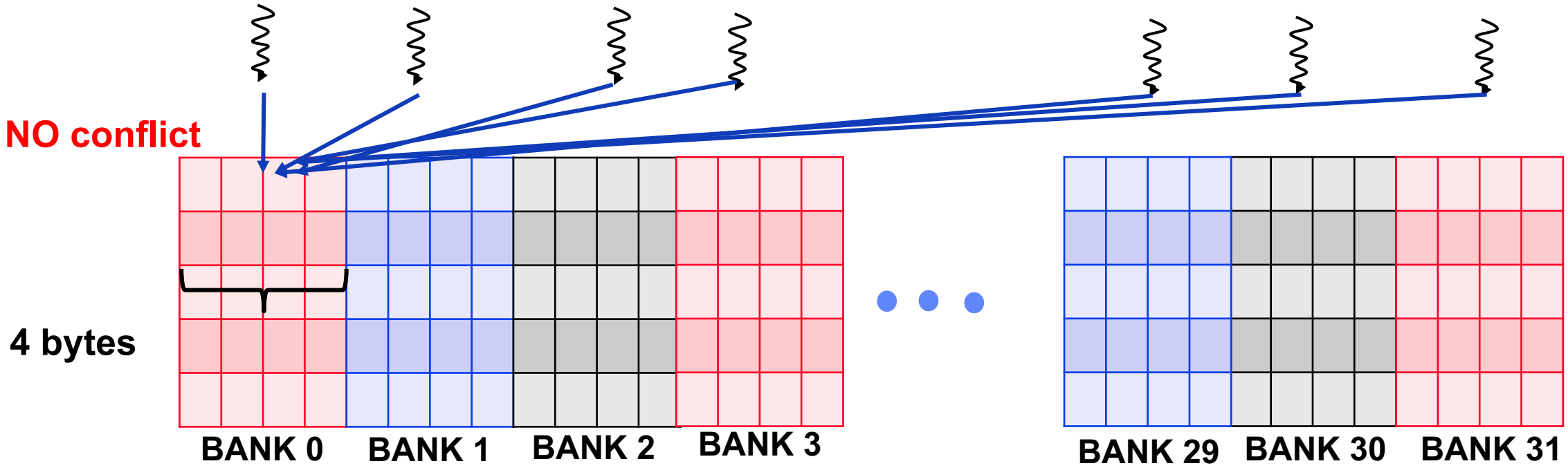
Performance pitfall: Beware of Shared Memory Bank Conflicts



- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

```
__shared int arr[64]
  arr[threadIdx.x]++;
```

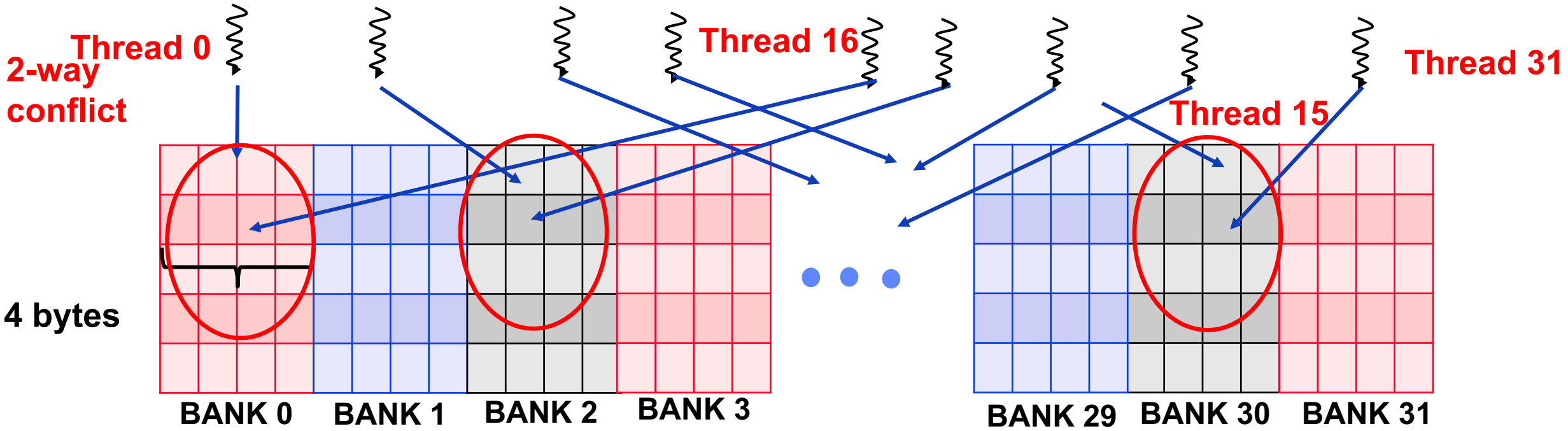
Performance pitfall: Beware of Shared Memory Bank Conflicts



- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

```
__shared int arr[64]
arr[blockIdx.x]++;
```

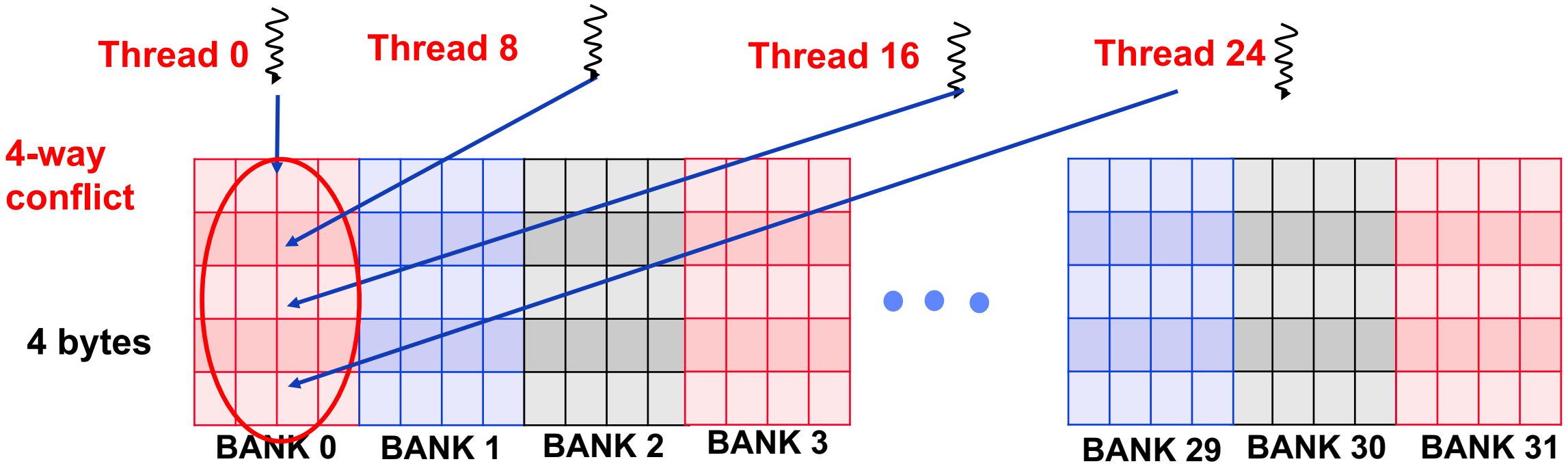
Performance pitfall: Beware of Shared Memory Bank Conflicts



- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

```
__shared int arr[64]
  arr[threadIdx.x*2]++;
```

Performance pitfall: Beware of Shared Memory Bank Conflicts

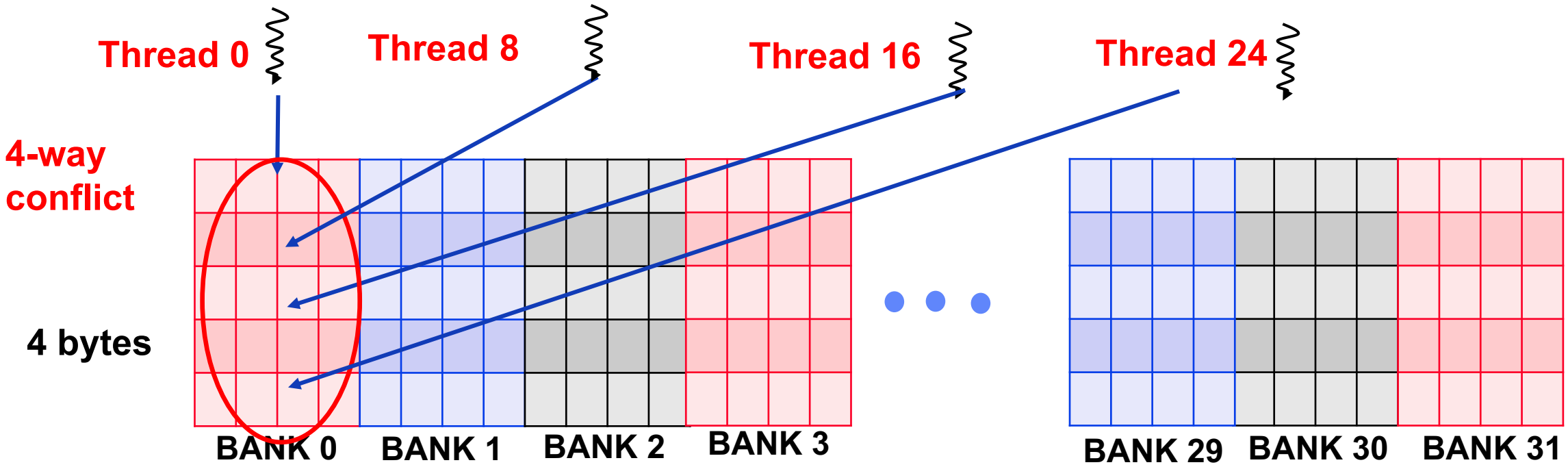


- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

```
__shared int arr[128]
  arr[threadIdx.x*4]++;
```

Worst case: 32-way conflict

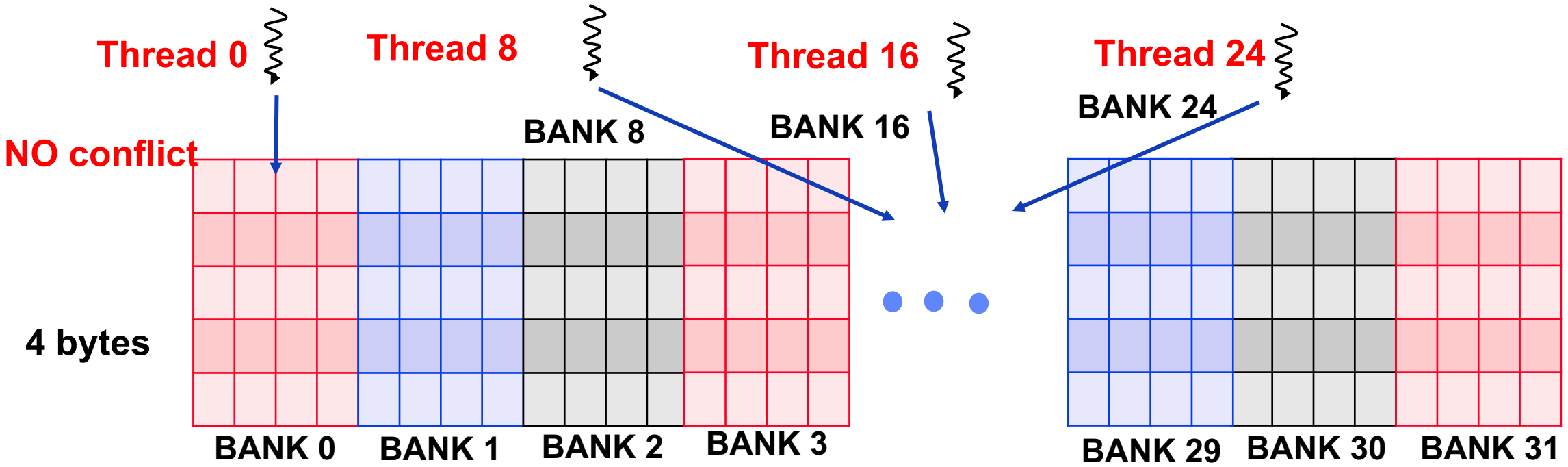
Performance pitfall: Beware of Shared Memory Bank Conflicts



- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

```
struct d4{
int x, y, z, w;
};

__shared d4 arr[128]
arr[threadIdx.x]++;
```

Avoiding Shared Memory Conflicts through Padding



- When would you have bank conflicts?
- Allocation (element) size
- Index expression into the data structure

  Alternative: Change indexing strategy (if possible)

```
struct d4{
int x, y, z, w, pad;
};

__shared d4 arr[128]
  arr[threadIdx.x]++;
```

Impact of Shared Memory on Occupancy

- Recall how occupancy is affected by various GPU resources
  - Max. 64 warps per SM
  - Max. 2048 threads per SM
  - Max. 32 threadblocks per SM
  - Max. 64K registers per SM
  - **Up to 164KB per SM**

Resource limitations on NVIDIA A100 GPU

Number of warp schedulable is limited by whichever resource hits limit first

- The amount of shared memory allocation can limit occupancy

Impact of Shared Memory on Occupancy

Constraints in scheduling threadblock:

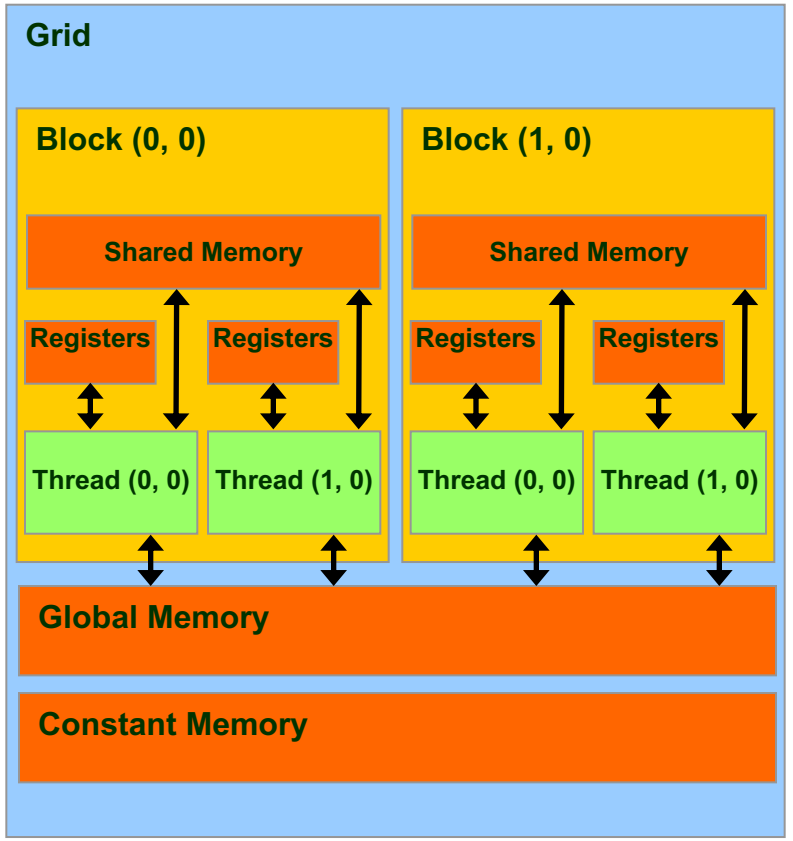
- All threads of a threadblock must execute on the same SM
- Each threadblock has own copy of the shared memory allocations
- Shared memory capacity on an SM must accommodate the needs of all threadblocks on the SM

Impact of Shared Memory on Occupancy

- Example: Maximum number of threads per SM 2048
- Maximum shared memory capacity per SM: 164 KB
- A kernel allocates 32KB of shared memory (per threadblock)
- Each threadblock launched with 256 threads
- What is the maximum occupancy? Ans: 1280/2048 (62.5%)

Summary: Different Memory Types of GPU

Variable declaration	Memory	Scope	Lifetime
`__device__ /cudaMalloc int globalVar;`	global	grid	application
`__device__ __constant__ int constantVar;`	constant	grid	application
`__device__ __shared__ int sharedVar;`	shared	block	block
`int localVar;`	register	thread	thread
`int localArr[N];`	global	thread	thread

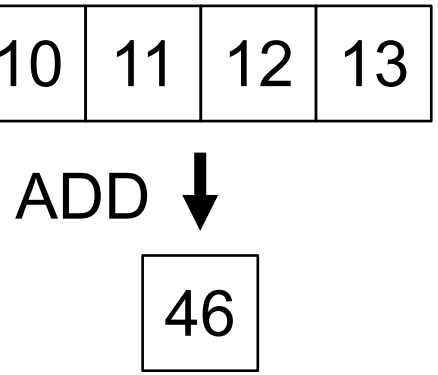


END-TO-END EXAMPLE

PARALLEL REDUCTION IN CUDA

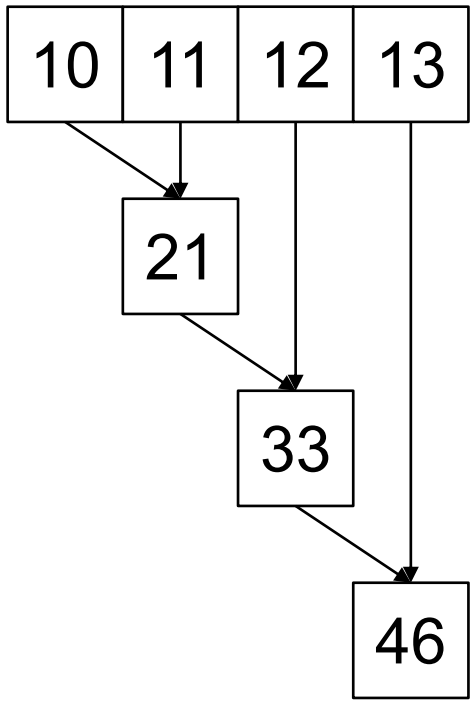
Reduction Operations

- Reduce multiple values to a single value
  - ADD, MUL, AND, OR, …
- Useful primitive when operating on large datasets



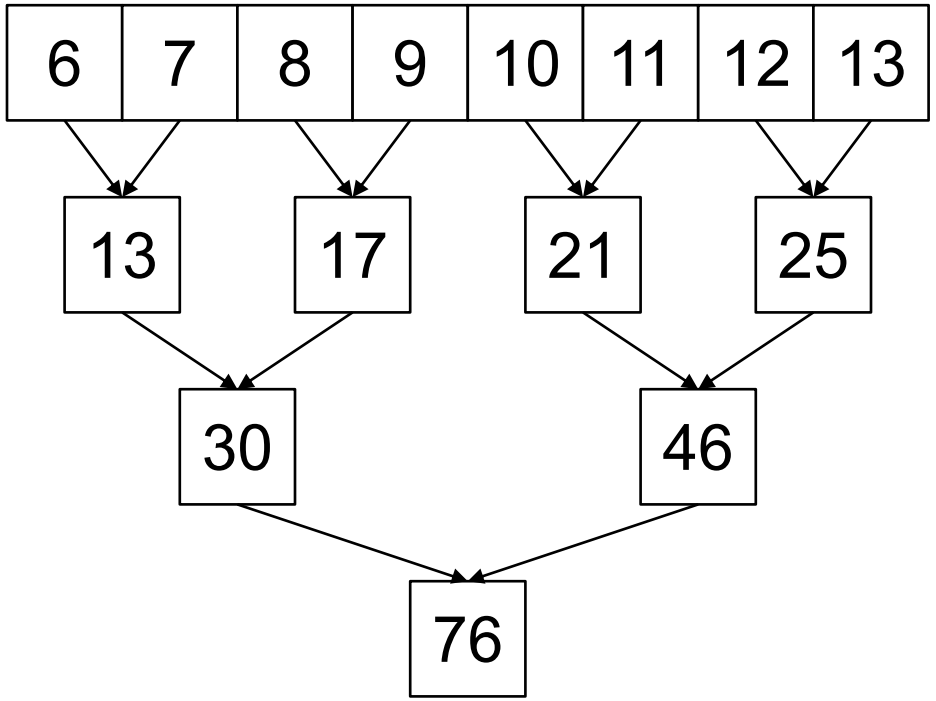
Sequential Reduction

- Process first two elements, produce partial result
  - Every level takes intermediate, and adds 1 element
- O(N) steps



Parallel Reduction

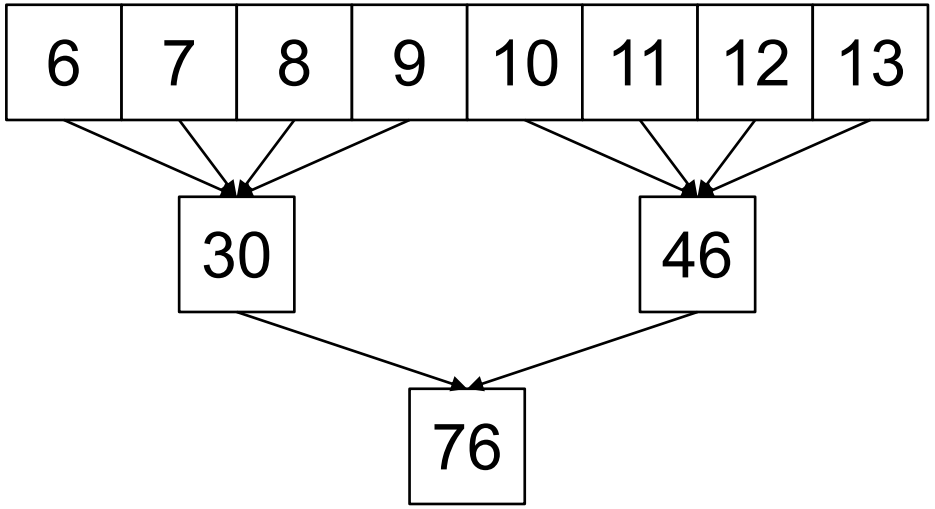
- Pair-wise reduction in tree-like steps
- $\log_2(N)$ where N is the number of elements



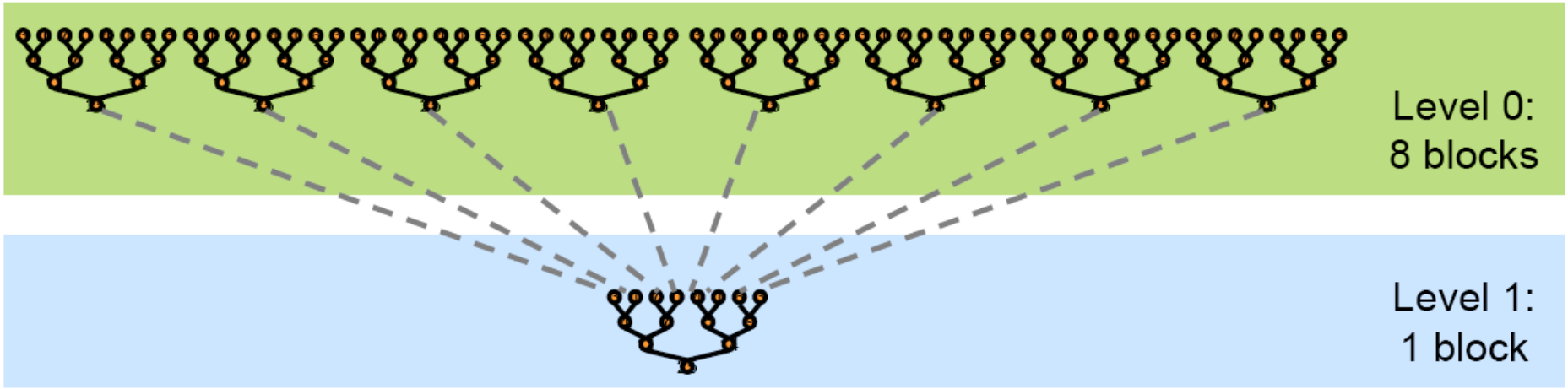
Parallel Reduction with Higher Degree

- Process x elements per step
  - $\log_x(N)$

- For x = 4:



Reduction: The Big Picture



- Use the same code for all reduction levels
- The same kernel is launched multiple times with different numbers of TBs
- This example uses 128 threads per TB…
  - For every 128 elements, there is one TB
  - In this graphic, smaller scale:
    - 8 elements per TB, 8 blocks = 64 elements total

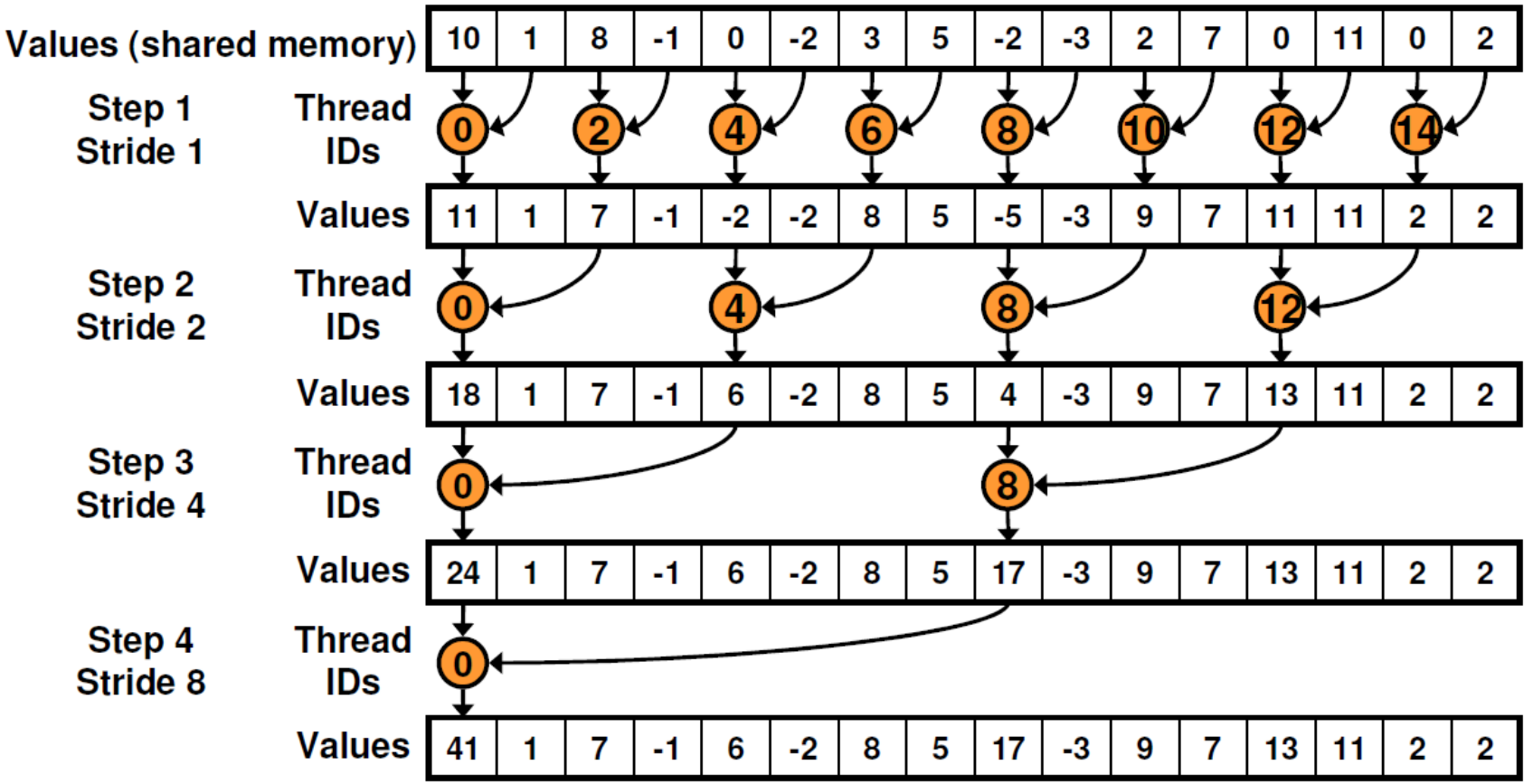
Reduction Optimization

- Reduction has low arithmetic intensity
  - One Flop / output element (add, compare, etc…)
  - All input elements are loaded from memory
- Therefore, we must optimize for bandwidth
  - Goal: reduce global memory accesses
  - Since partial sums reused, load to **shared memory** first

Reduction 1: Interleaved Addressing

- Load Data
  - Loads 1 element/thread from **global** to **shared** memory
- Reduction: proceed in $\log_2 N$ steps
  - A thread reduces two elements
    - The first two elements by the first thread
    - The next two by the next thread and so on
  - At the end of each step
    - Deactivate half of the threads
  - Terminate when one thread left
- Write back to global memory, visible to next kernel

Reduction 1: Visualization



Reduction 1: Interleaved Addressing

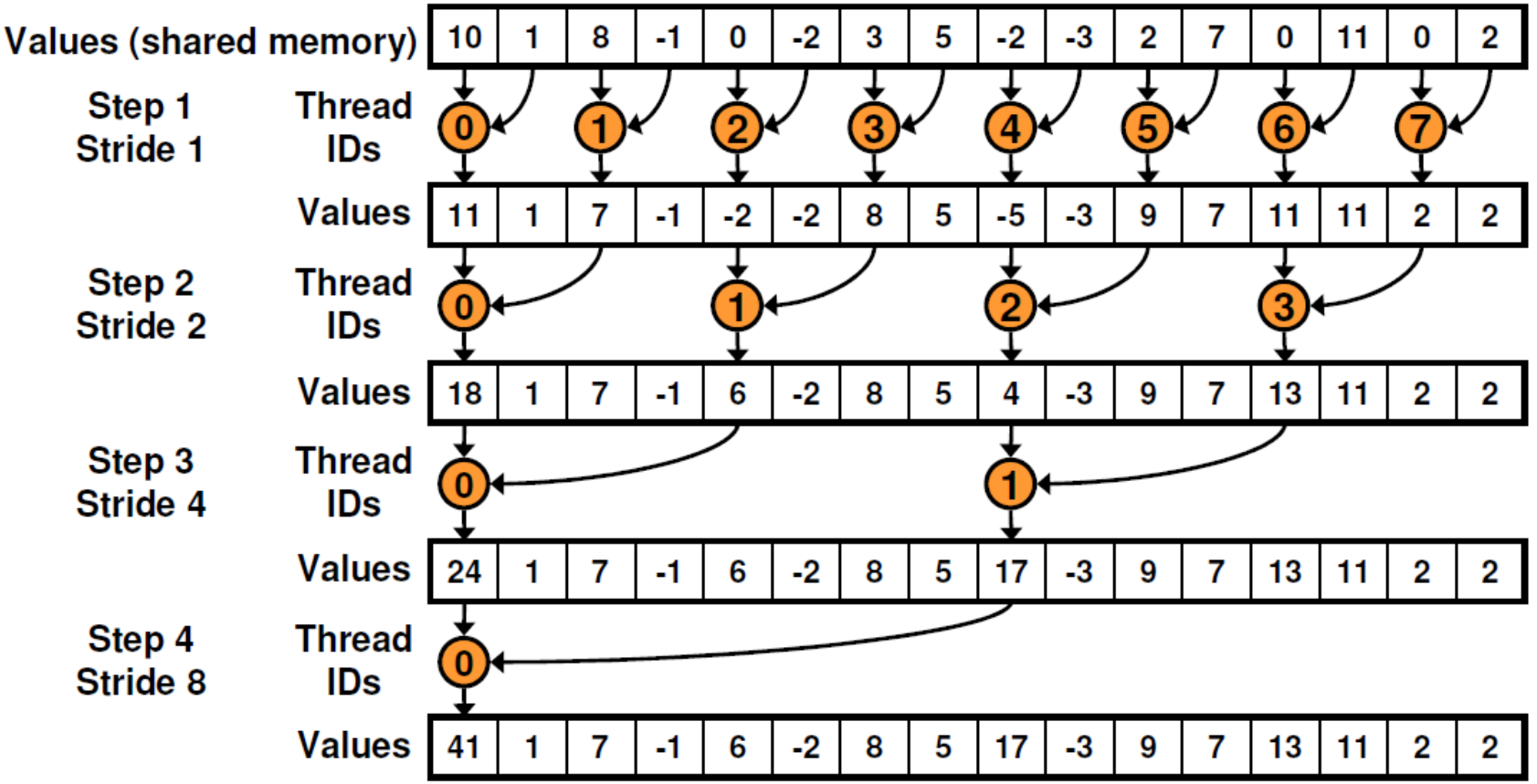
```
__global__ void reduce0(int *g_idata, int *g_odata, int n){
        extern __shared__ int sdata[];

        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = (i < n) ? g_idata[i] : 0;
        __syncthreads();

        // do reduction in shared mem
        for (unsigned int s=1; s < blockDim.x; s *= 2) {
                // step = 2*s
                if (tid % (2*s) == 0) {
                        // only threadIDs divisible by the step do work
                        sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }

        // write result for this block to global mem
        if (tid == 0) g_odata[blockIdx.x] = sdata[0];
```

Reduction 1: Divergence Problem

```
__global__ void reduce0(int *g_idata, int *g_odata, int n){
        extern __shared__ int sdata[];

        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = (i < n) ? g_idata[i] : 0;
        __syncthreads();

        // do reduction in shared mem
        for (unsigned int s=1; s < blockDim.x; s *= 2) {
                // step = s x 2
                if (tid % (2*s) == 0) {
                // only threadIDs divisible by the step participate
                        sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }

        // write result for this block to global mem
        if (tid == 0) g_odata[blockIdx.x] = sdata[0];
```

Highly divergent code leads to poor performance

Reduction 2: Visualization



Reduction 2: Non-Divergent Branching

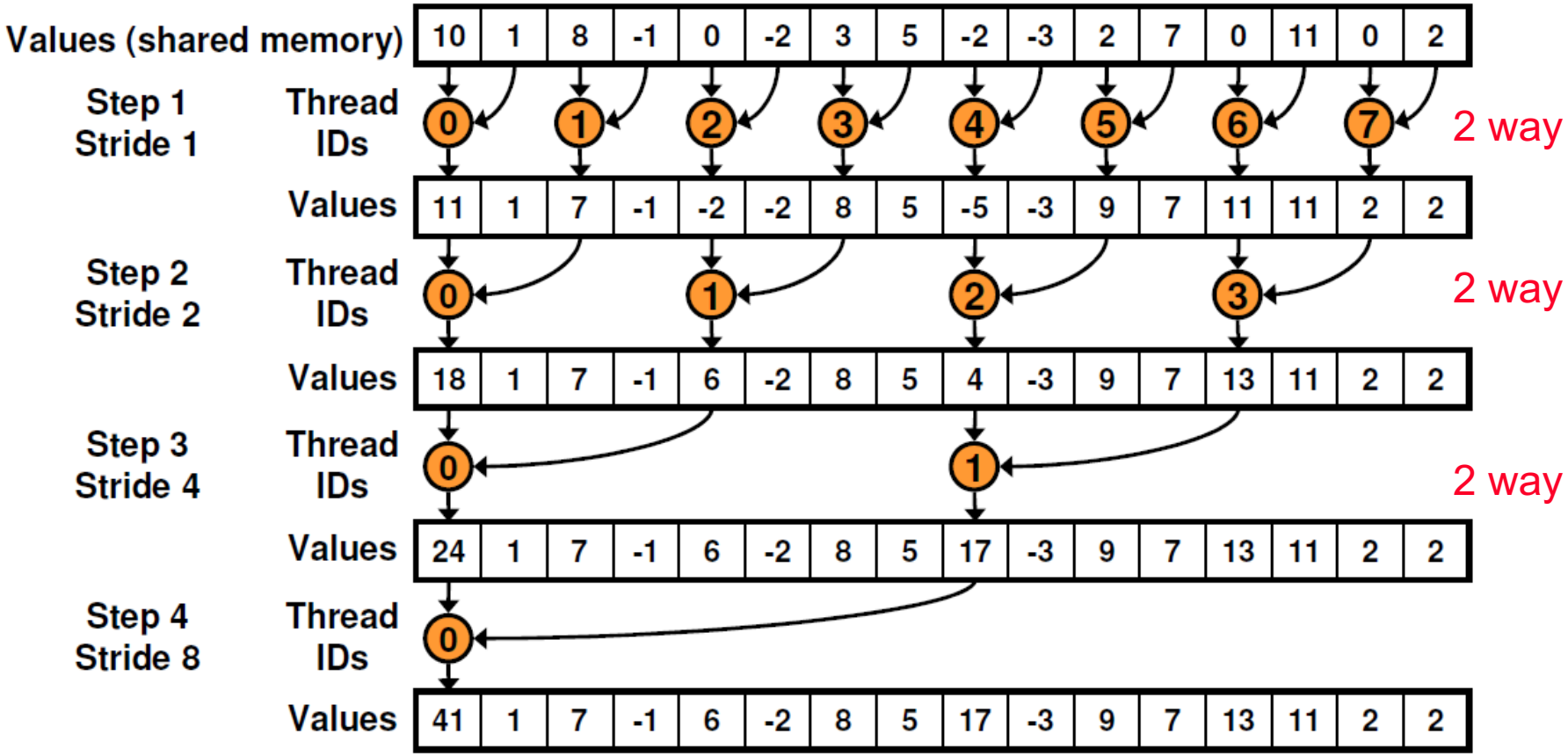
- Replace divergent branching code

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

- With strided index and non-divergent branching

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

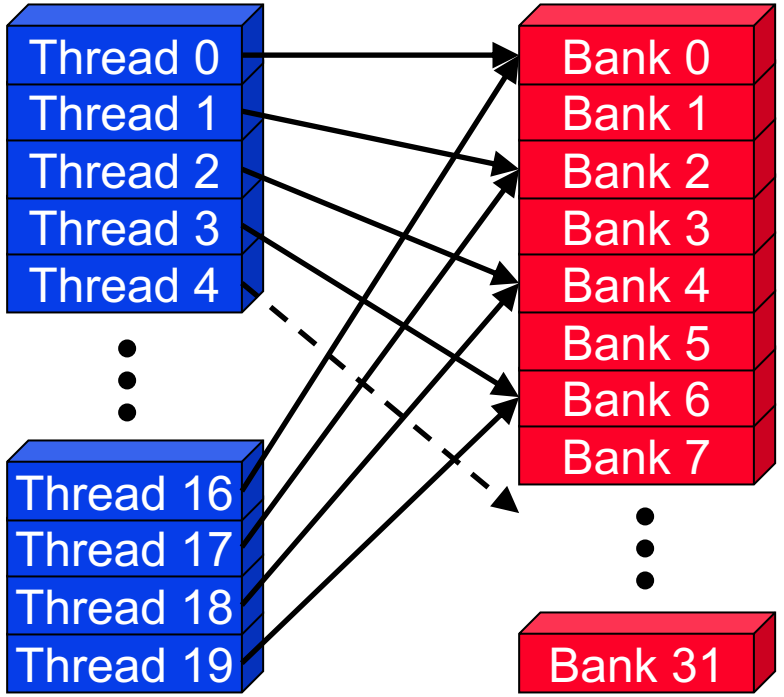
Reduction 2: Bank Conflicts



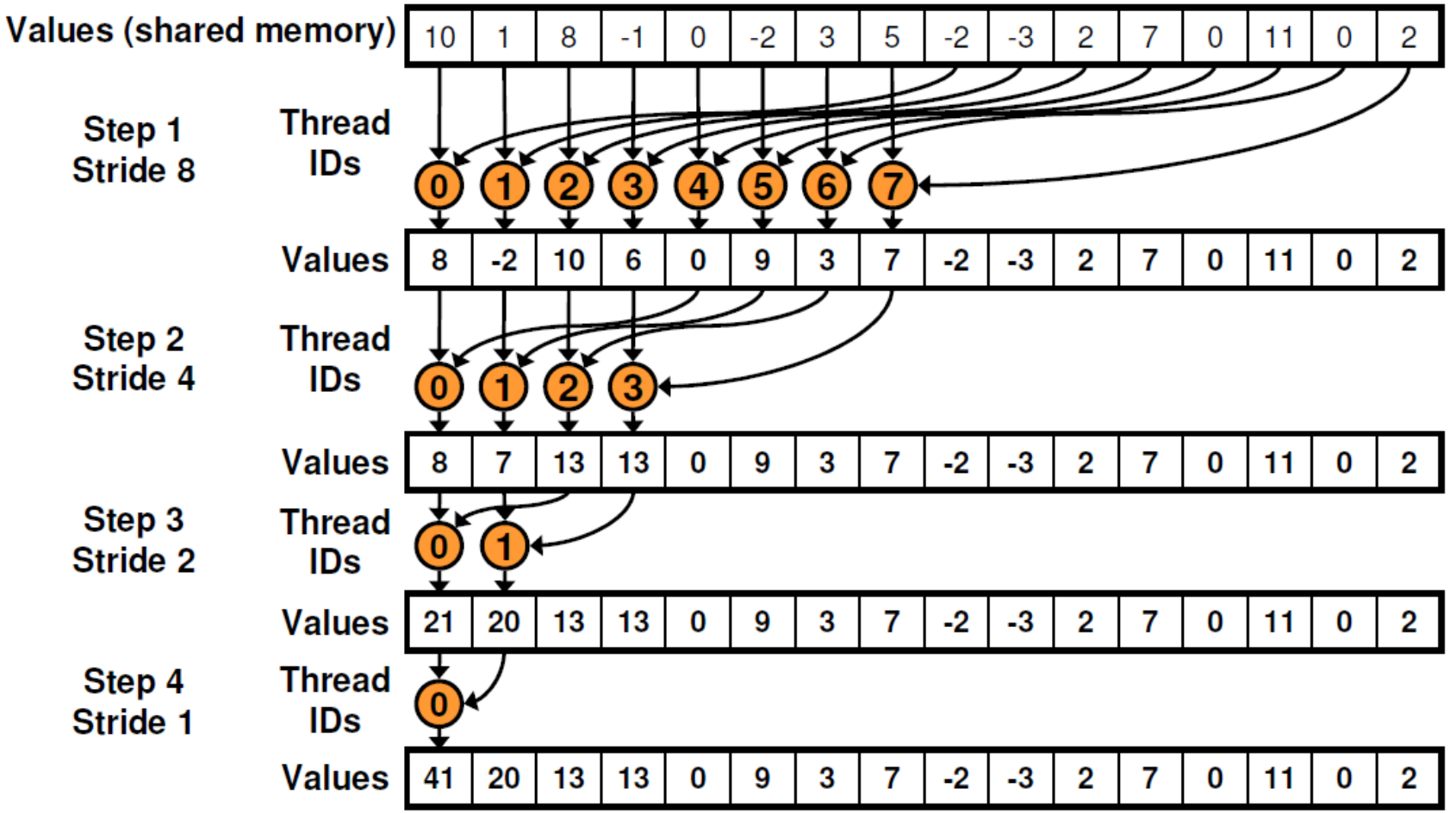
Why Bank Conflicts?

- First memory access from threads 0 & 1 goes to:
  - Banks 0 and 2
  - Keep increasing bank #, thread 16 accesses bank 0

- Oops, conflict!
  - These accesses are split into 2 sequential ones
  - Doubles our shared mem latency → slowdown!



Reduction 3: Visualization



Reduction 3: Sequential Accessing

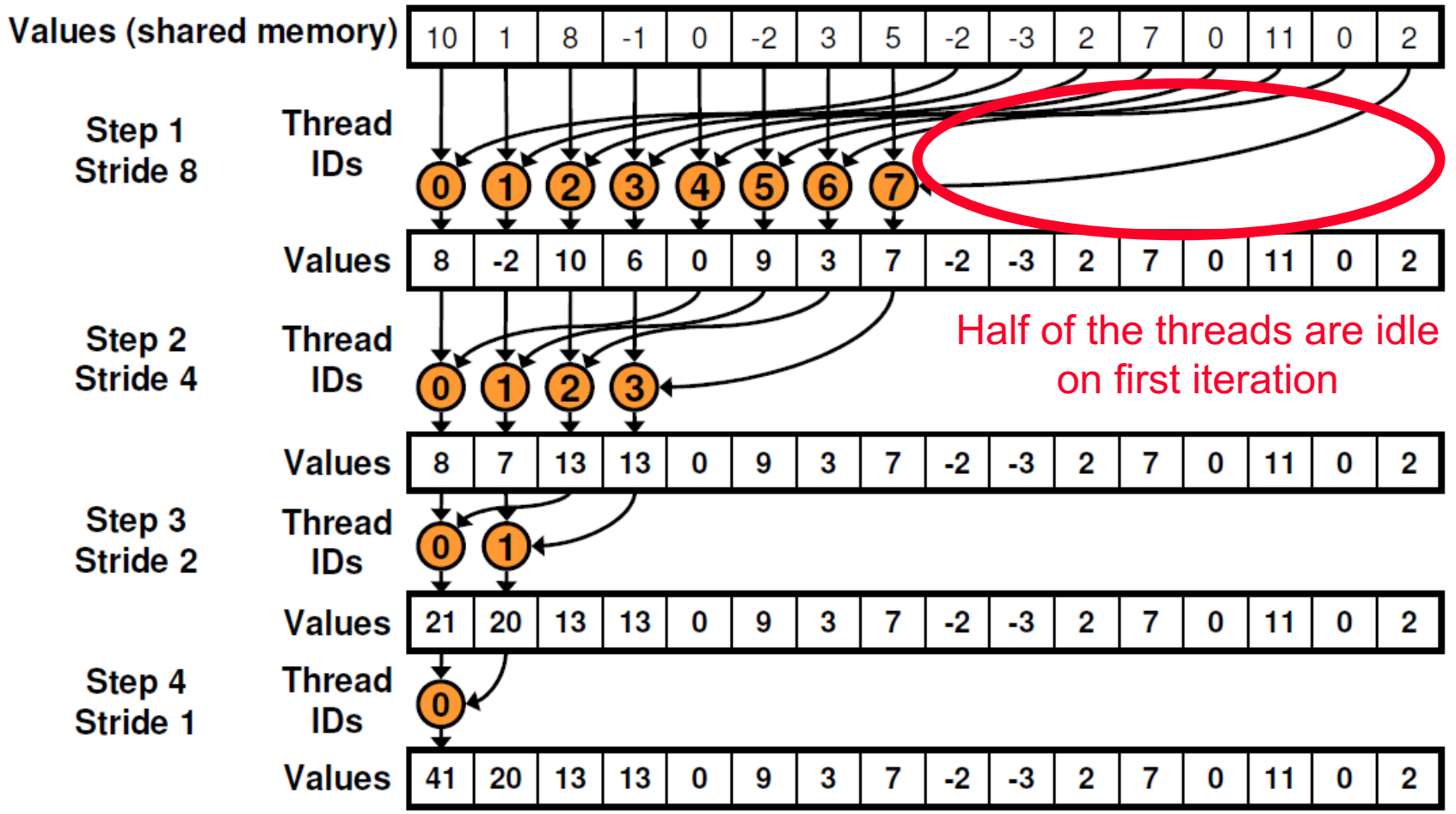
- Replace strided indexing

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

- With reversed loop and threadID-based indexing

```
for (unsigned int s = blockDim.x/2; s > 0; s /= 2) {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

Reduction 3: Idle Threads



Reduction 4: Reduce During Load

- Instead of each thread loading one element

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
```

- Allow each thread to load two elements
  - Do the first reduction step alongside the load
  - Halves the number of threads in a threadblock

With four optimizaitons kernel time reduced from 327 ms to 9 ms

```
// each thread loads two elements from global to shared mem
// and performs the first step of the reduction
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x* blockDim.x * 2 + threadIdx.x;
sdata[tid] = (i < N) ? ((i + blockDim.x) < N ?
        (g_idata[i] + g_idata[i + blockDim.x]) : g_idata[i]) : 0
__syncthreads();
```

Summary

- GPUs have large compute but are often limited by memory system
- GPUs use large register files to fast context switch for throughput
- Important to write programs to leverage memory coalescing
- Important to use shared memory where possible