CS302

GPU: Fine-grain synchronization

Spring 2025

Prof. Babak Falsafi, Prof. Arkaprava Basu

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
12-May	13-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
  - Fine-grain Synchronization
  - Data races
- Next class
  - Warp primitives, Tensor cores, Multi-tenancy

Review: Basic Synchornizaiton

- Synchornizaiton in GPU for bulk-synchrnous program
- Bulk synchornous programs are common
  - Threads/threadblocks work mostly independently on separate portions of data
  - Many threads synchornize at the same time in coarse granuality (infrequently)
- Threadblock barrier (__syncthreads) to waits for all threads of a block before proceeding
- Kernel decompositon for global synchornizaiton

Fine-grain Synchornizaiton

- Synchronization is defined for each individual thread
  - Not a barrier
  - Each thread can execute these operations independently
- Atomic operations
- Memory fence operations
- Lock/unlock (Mutual exclusion) in CUDA programs

Atomic operations in CUDA

- Recall from Lecture 6.2: Atomics are Read-Modify-Write
  - Reads a memory location, modifies/updates the value read, writes back the result
  - Different flavors of atomics based on the “Modify” function
- Hardware ensures that all parts of an atomic operation happen without interruption
  - Atomic operations are typically performed in shared LLC and are slow
- Serializes if two threads issue atomic operation to the same address

Atomic operations in CUDA

- Atomics in CUDA are intrinsic fucntions
  - Function calls are typically translated to a single ISA instructions
  - Many flavors: add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)

- Example:

  int atomicAdd(int* address, int val)

  Reads the 32-bit word **old** pointed to by **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. The function returns **old**.

  Different operand types possible: unsigned int, long, float…

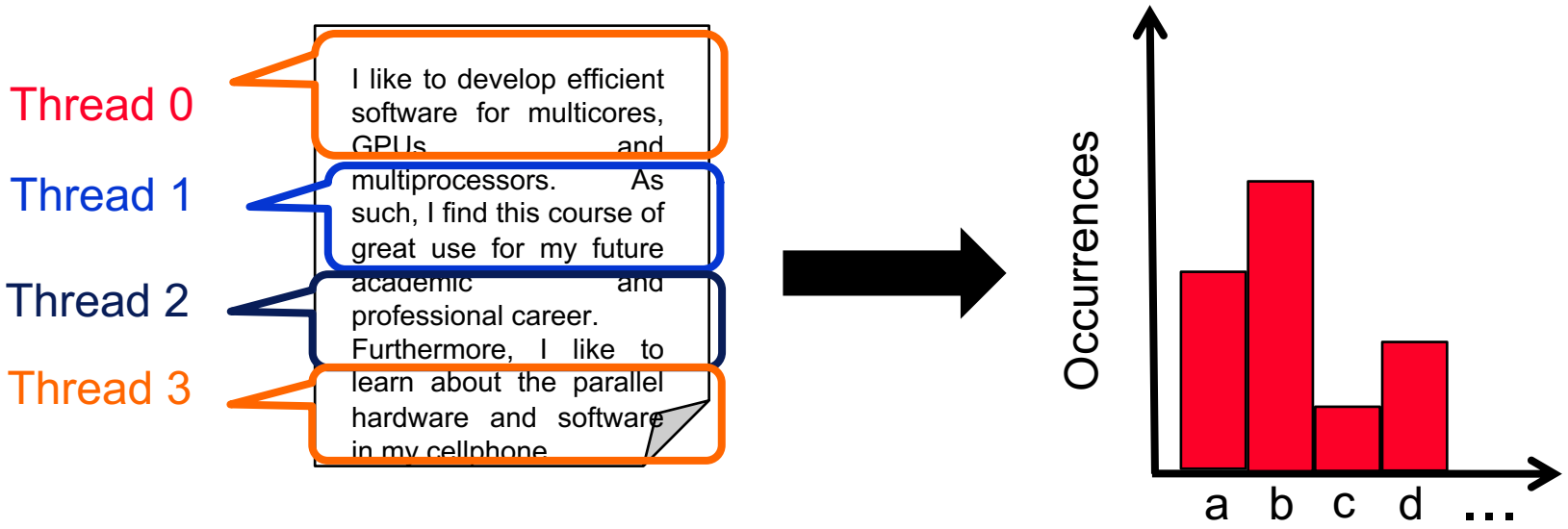
Atomic operations in CUDA

- int atomicExch(int* address, int val)
  - Reads the 32-bit or 64-bit word old located at the address address in global or shared memory and stores val back to memory at the same address. The function returns old.

- int atomicCAS(int* address, int compare, int val)
  - Reads the 16-bit, 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old == compare ? val : old), and stores the result back to memory at the same address. The function returns old.

Example use of Atomics: Parallel Histogram Generation

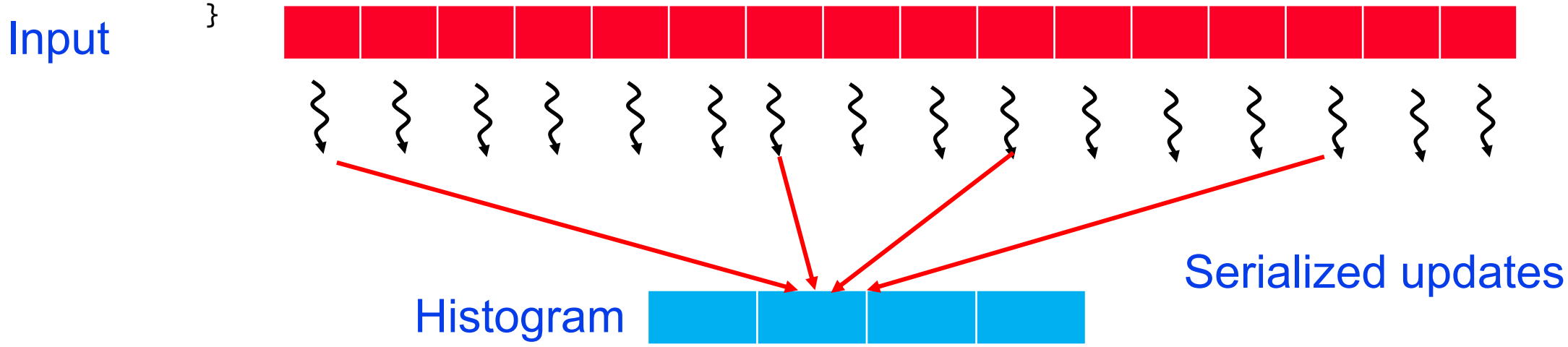
- Count num. of ASCII characters in an input text
- Input: ASCII characters in array format
- Output: Histogram w. each bucket, # occurrences



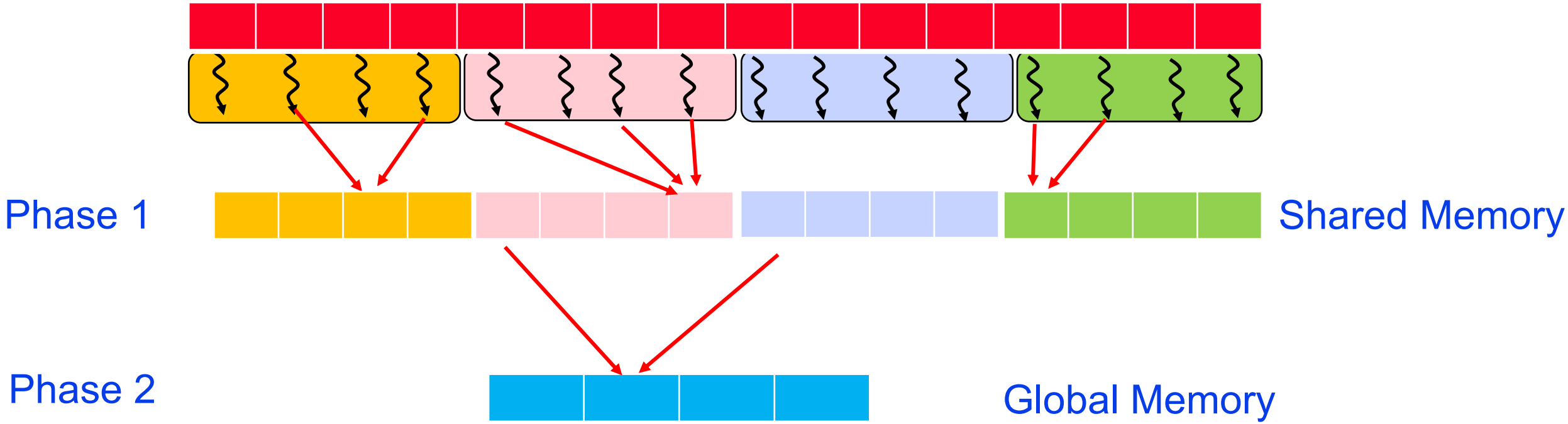
Example use of Atomics: Parallel Histogram Generation

```
__global__ void hist_calc(char *data, unsigned int length,
                          unsigned int * histogram)
{
 unsigned int index = blockIdx.x * blockDim.x + threadIDx.x;

 if (index < length) {
        int position = data[i] – ‘a’;

        if (position >= 0 && position < 26) {
               atomicAdd(&histogram[position], 1);
        }
  }

}
```

Necessary for correctness



Example use of Atomics: Parallel Histogram Generation



- Reduce serialization through privatization
  - Each threadblock keeps their own copy of histogram in the shared memory
  - Merge private copies in the global memory in the next phase

Example use of Atomics: Parallel Histogram Generation

```
__global__ void hist_calc(char *data, unsigned int length,
                          unsigned int * histogram){

__shared__ unsigned int histo_s[NUM_BINS]; //Number of bins in the histogram

//initialize private copies in parallel (BTW, why is there a loop?)
For (unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
        histo_s[bin] = 0;
}
  __syncthreads();

 unsigned int index = blockIdx.x * blockDim.x + threadIDx.x;
 if (index < length) {
        int position = data[i] – ‘a’;
        if (position >= 0 && position < 26) {
                atomicAdd(&histo_s[position], 1); //Atomic on shared memory
        }
  }
   __syncthreads();
//Merging results onto the final histogram in the global memory
for (unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
        unsigned int val = histo_s[bin];
        if (val > 0) atomicAdd(&histo[bin], val);  //Atomic add on global memory
}
```

Advantages of this implementation:

(1) Less contention

(2) Shared memory atomics are faster

Twin Challenges of Synchronization in CUDA programs

- Weak memory consistency model
- Lack of hardware cache coherence

Weak Memory Model of CUDA

- CUDA has a weak memory model
  - Informally, the order in which a thread writes data to memory is **not** necessarily the order in which the data is observed being written by another thread

```
__device__ int X = 1, Y = 2;
```

thread 1

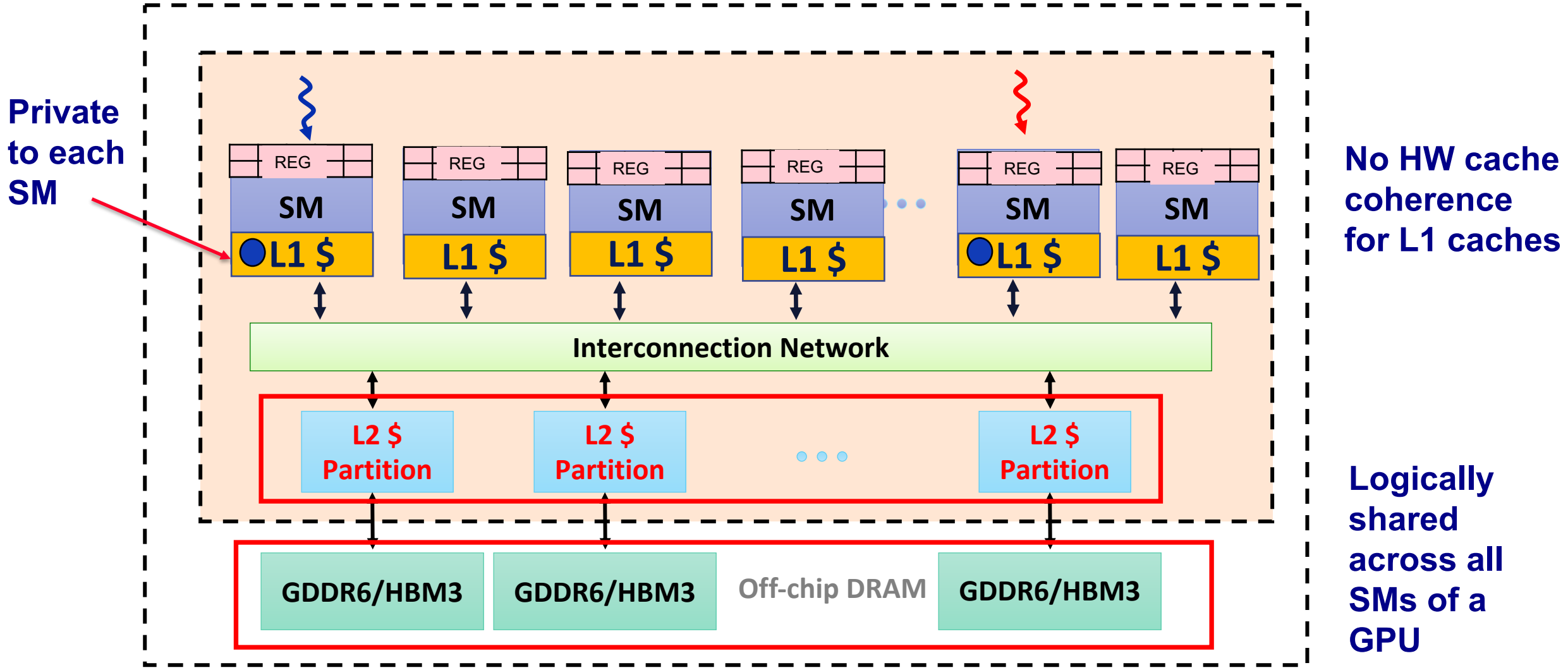
```
__device__ void writeXY()
{
        X = 10;
        Y = 20;
}
```

thread2

```
__device __void readXY ()
{
        int B = Y;
        int A = X;
}
```

- CUDA also has relaxed atomics
  - Loads, stores can bypass atomic operation
- ___

GPU L1 Caches are NOT Coherent



Memory Fence Operations in CUDA

- Fence instruction (*__threadfence*) for ordering read/writes

> *"ensures that no writes made by the calling thread after the __threadfence() are observed by any thread in the device as occurring before any write by the calling thread before the __threadfence()"* *--- NVIDIA documentation*

Memory Fence Operations in CUDA

- Informally, if a thread:
- makes a change A to a data item in global or shared memory
- executes a __threadfence
- makes another change B to that data in the global or shared item
- Then another thread in the grid cannot read that item and observe B, and then later read that item and observe A.

Memory Fence operations in CUDA

- More generally, __threadfence serves two purposes
- **Visibility:** Writes appearing before the fence by the calling thread must become visible to any threads in the grid before any writes after the fence in the calling thread.
- **Ordering:** Load, stores, atomics in the calling thread cannot be re-ordered across a __threadfence.

Memory Fence Operations in CUDA

```
__device__ int X = 1, Y = 2;
```

thread 1

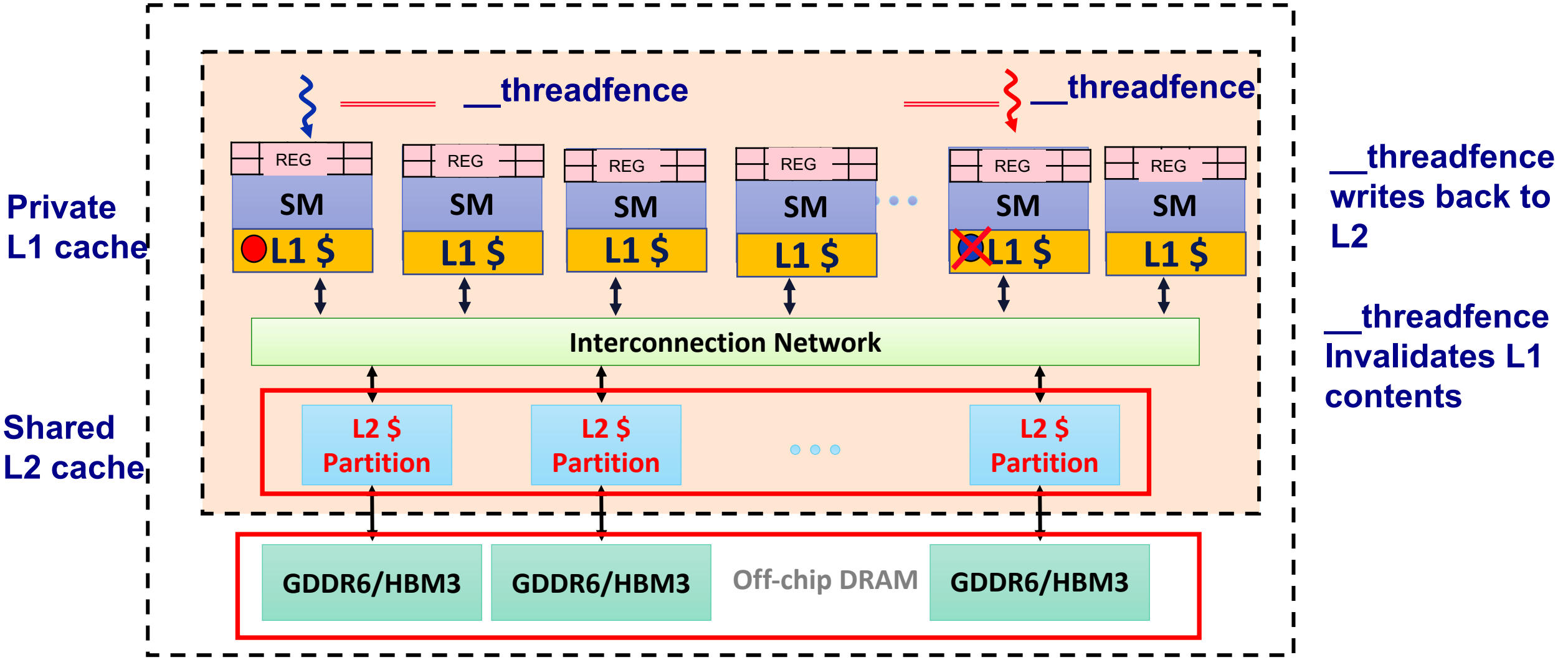
```
__device__ void writeXY()
{
        X = 10;
        __threadfence;
        Y = 20;
}
```

A= 1 and B = 20
is not possible

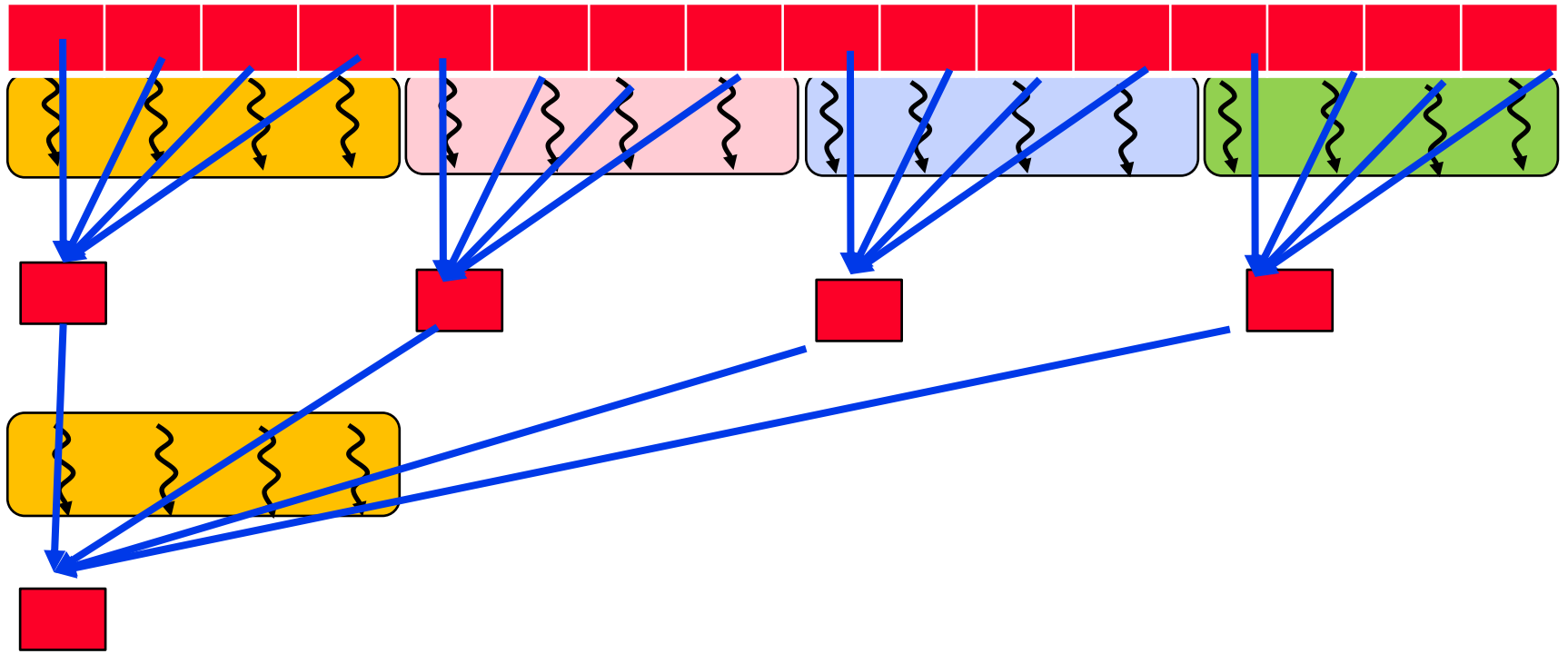
thread2

```
__device __void readXY ()
{
        int B = Y;
        __threadfence;
        int A = X;
}
```

GPU L1 Caches are not Coherent: Need Fence



Example use of Fence: Parallel Sum of an Array of Integers



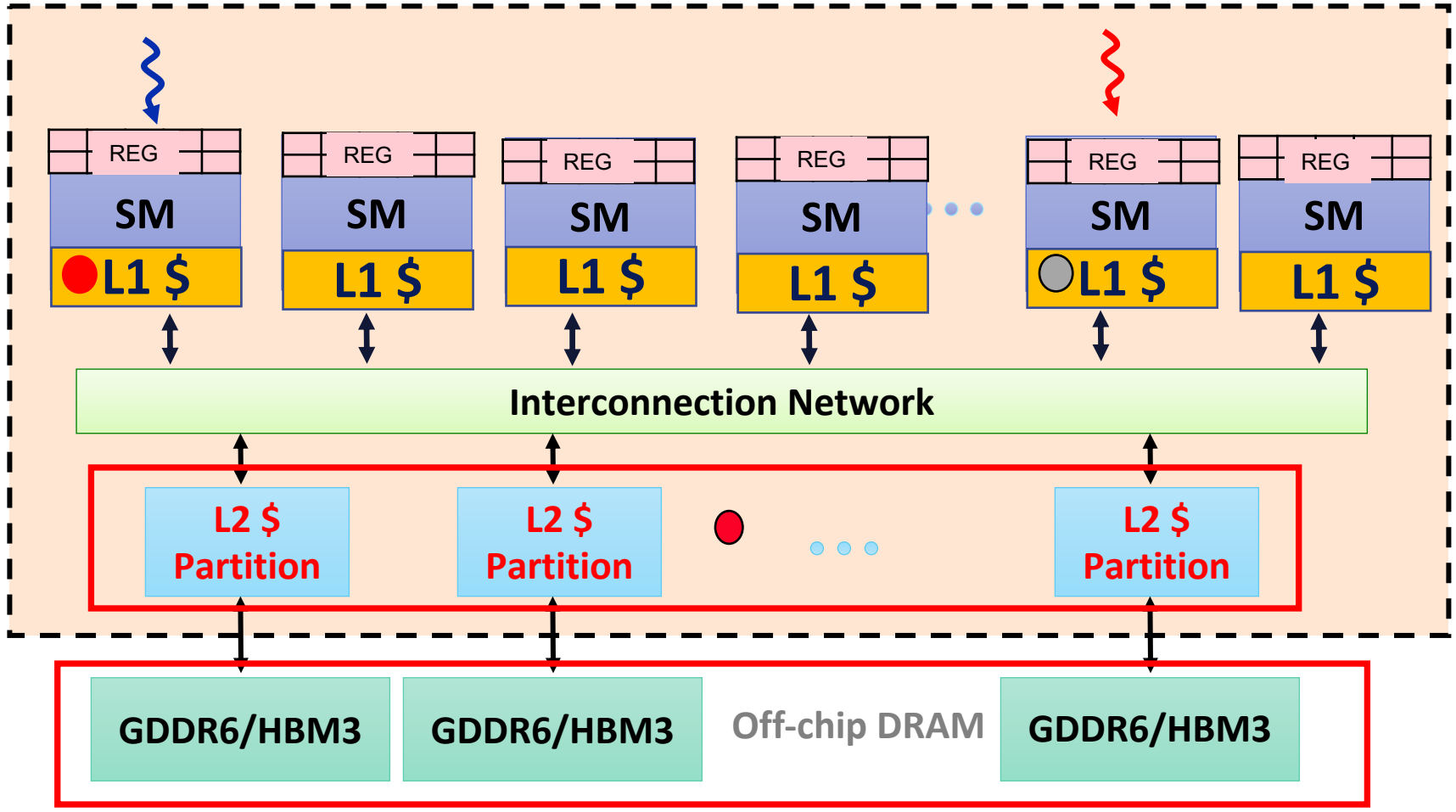
- Each threadblock calculates the partial sum of a sub-array
- A thread from each threadblock writes its partial result to memory
- The block that finishes last adds partial sums to the final sum

Example use of Fence: Parallel Sum of an Array of Integers

```
__device__ unsigned int count = 0;__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    volatile float* result)
{
        float partialSum = calculatePartialSum(array, N); //Computes partial sum of the array
        if (threadIdx.x == 0) {
                result[blockIdx.x] = partialSum;
                __threadfence();       // Ensures partialSum is written before count is incremented
                unsigned int value = atomicInc(&count, gridDim.x);  //Increments count if below # of blocks
                isLastBlockDone = (value == (gridDim.x - 1));   // Is it the last block?
        }
        __syncthreads();  //Guarantees that all threads of a block see the updated 'isLastBlockDone'

         if (isLastBlockDone) {
                float totalSum = calculateTotalSum(result);
                 if (threadIdx.x == 0) {   //Thread  0 of the last block updates the final sum
                        result[0] = totalSum;
        }
    }
}
```

Compiler can Allocate Variables in Registers

'volatile' qualifier directs compiler NOT to allocate a variable in register or L1



~~Compiler can cache global or shared memory variables in registers/L1 cache~~

Remember L1 cache is incoherent

Example (2) use of fences: Producer – Consumer Pattern

- Producer writes to shared data items it wants to communicate
- Producer updates a flag to notify data item is ready

- Consumer waits for the flag to be updated in a loop
- Consumer reads the data items for processing

Example (2) use of fences: Producer – Consumer Pattern

(global) **volatile** Flags *flags; //Structure Flags contain both data and status flag

Producer

```
volatile Flags *mFlags = flags + blockIdx.x;

mFlag->sum = SUM; // Write to data

__threadfence(); //Ensures status is updated
                 // after write to data

mFlag->status = 1; //Set flag to notify consumer
```

Consumer

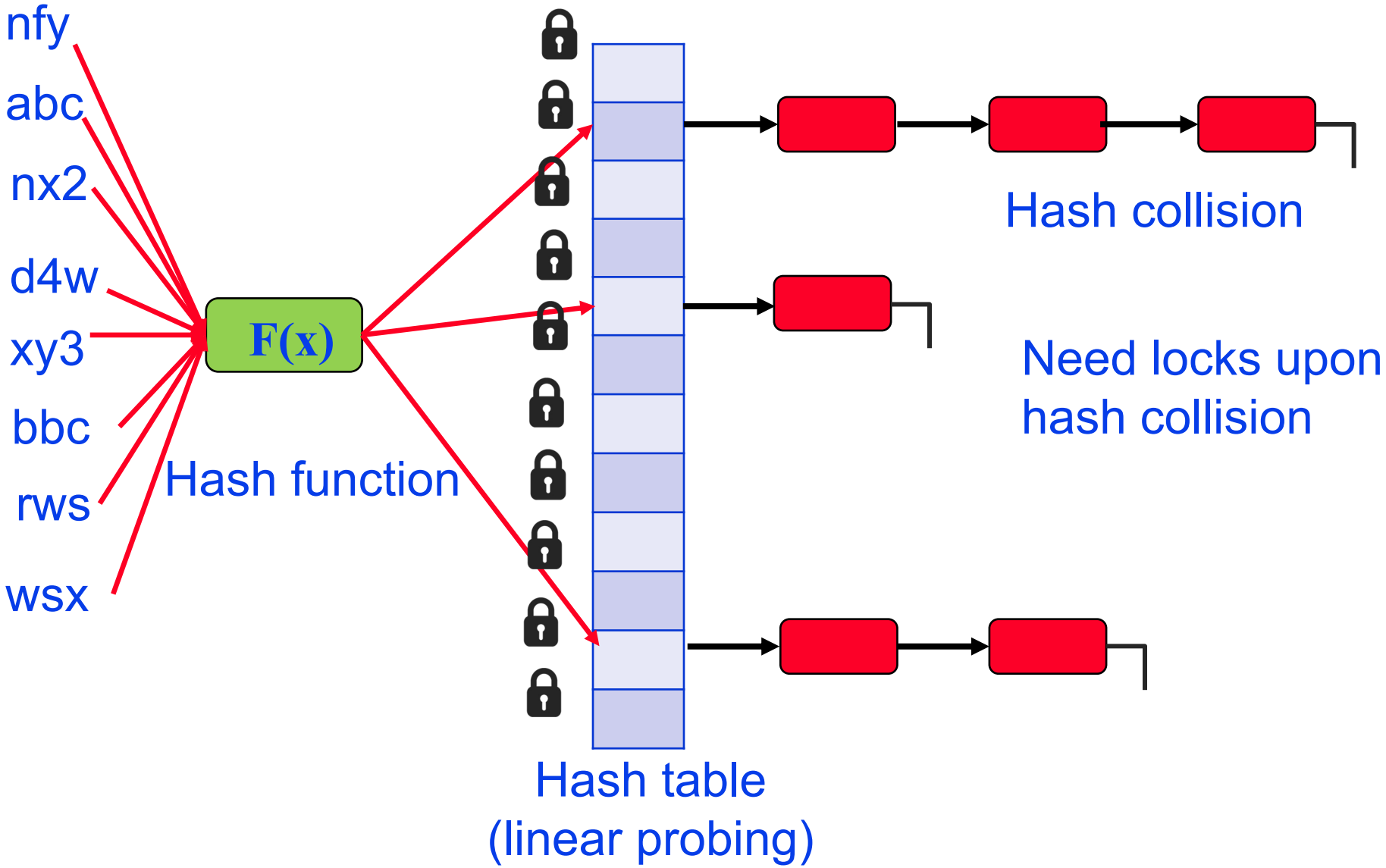
```
volatile Flags *sFlags = flags + (blockIdx.x - 1)

while (sFlag->status == 0);  // wait in busy loop

__threadfence();  //Ensures data is not read
                  // before updated status is read

accumulate += sFlag->sum; //Read data item
```

Stripped down code snippet from NVIDIA's cuML library

Lock and Unlock (Mutual Exclusion) in CUDA program

```
__global__ int lock = 0;  // 0 → lock is free. 1 → locked

while (atomicCAS(&lock, 0, 1) == 0);  // wait until lock value is zero
 __threadfence();                 // Ensures that code in the critical section
                                  //does not start executing before lock is acquired
        //critical section
        -------
        ------

__threadfence();          //Ensures the critical section finishes before lock release
atomicExcg(&lock, 0);          // Release lock by setting it to zero
```

Example Use of Locks: Parallel Hash Table Insertion



Example Use of Locks: Parallel Hash Table Insertion

```
__device__ void hash_insert ( tHashTable *g_hashtable, unsigned key, unsigned value)
{
    ……..
    ……..
   unsigned hash = hash_func(key);  //hash index
   BaseEntry *base = &g_hashtable->mValues[hash];

   while(atomicCAS(&base->mLock, 0, 1) == 0);   //Lock acquire
          __threadfence;

      ent->mKey = key; ent->mValue = value;
      ent->mNext = base->mIndex;
      g_hashtable->mValues[hash].mIndex = ent;

      __threadfence;
      atomicExcg(&base->mLock, 0);   //Unlock
    }
  }
}
```

Summary: Fine-grain Synchronization in CUDA

- Bulk synchronous programs use barrier and kernel decomposition
- Fence and atomic operations allow fine-grain synchronization defined for each individual thread
- Fence and atomics together can create lock and unlock operations