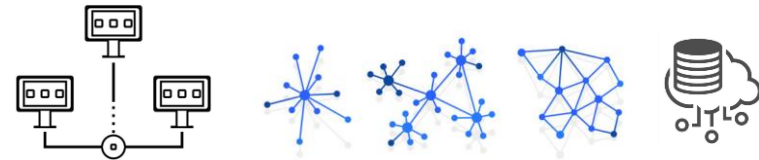




# Introduction to Parallel and Distributed Programming

CS 13  
Massive Multi-core Programming

Saikishor Jangiti



# Overview of Lecture

---

Where data can be stored

- And how to get it there

Some guidelines for where to store data

- Who needs to access it?
- Read only vs. Read/Write
- Footprint of data

High level description of how to write code to optimize for memory hierarchy

- Reference :

Chapter 5, Kirk and Hwu , Programming Massively Parallel Processors A Hands-on Approach, *Third Edition*

# Objective

---

To master Reduction Trees, arguably the most widely used parallel computation pattern

- Basic concept
- A class of computation
- Memory coalescing
- Control divergence
- Thread utilization

# Partition and Summarize

---

A commonly used strategy for processing large input data sets

- There is no required order of processing elements in a data set (associative and commutative)
- Partition the data set into smaller chunks
- Have each thread to process a chunk
- Use a reduction tree to summarize the results from each chunk into the final answer

We will focus on the reduction tree step for now.

Google and Hadoop MapReduce frameworks are examples of this pattern

# Reduction enables other techniques

---

Reduction is also needed to clean up after some commonly used parallelizing transformations

## Privatization

- Multiple threads write into an output location
- Replicate the output location so that each thread has a private output location
- Use a reduction tree to combine the values of private locations into the original output location

# What is a reduction computation

---

Summarize a set of input values into one value using a “reduction operation”

- Max
- Min
- Sum
- Product
- Often with user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)

# An efficient sequential reduction algorithm performs $N$ operations - $O(N)$

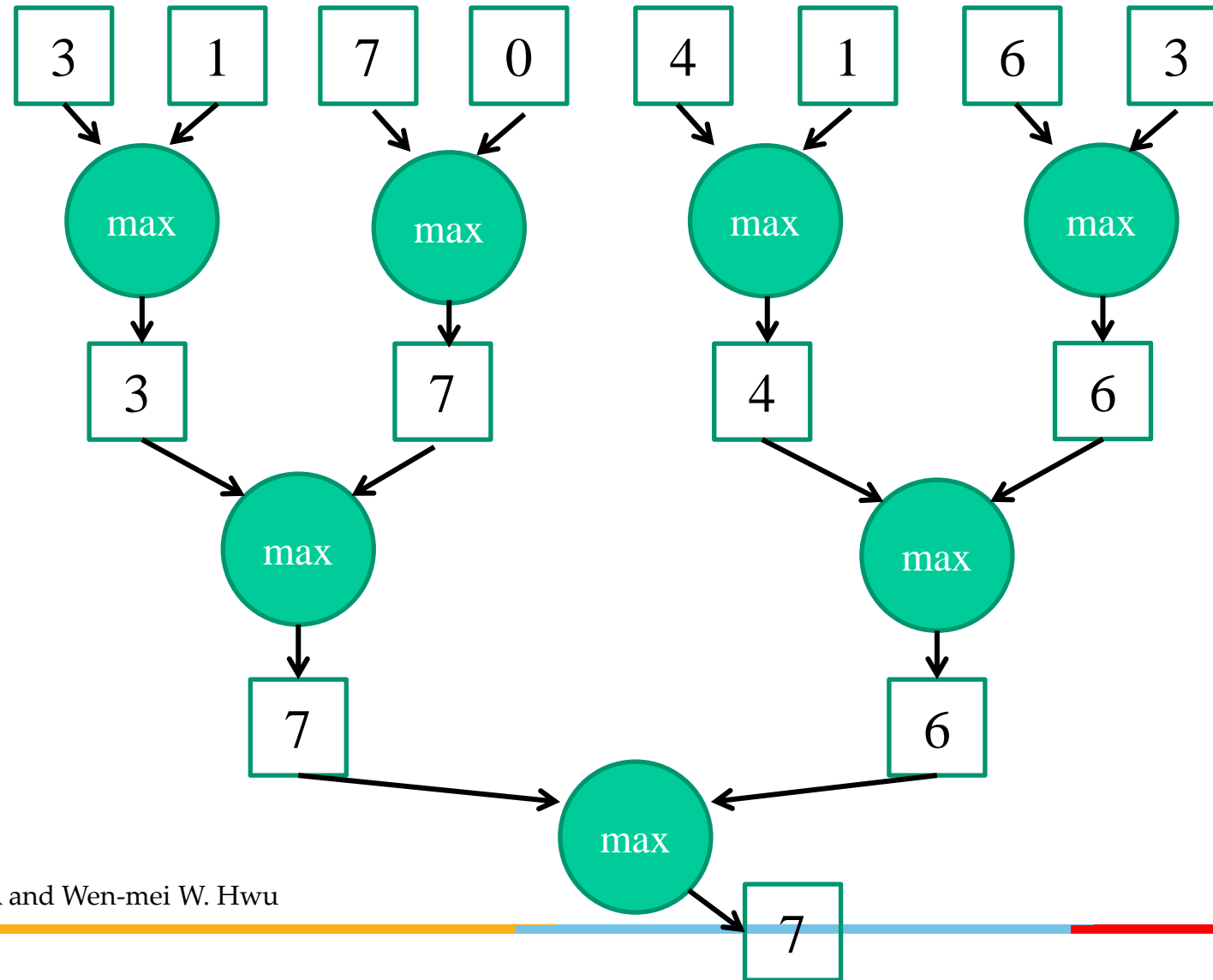
---

Initialize the result as an identity value for the reduction operation

- Smallest possible value for max reduction
- Largest possible value for min reduction
- 0 for sum reduction
- 1 for product reduction

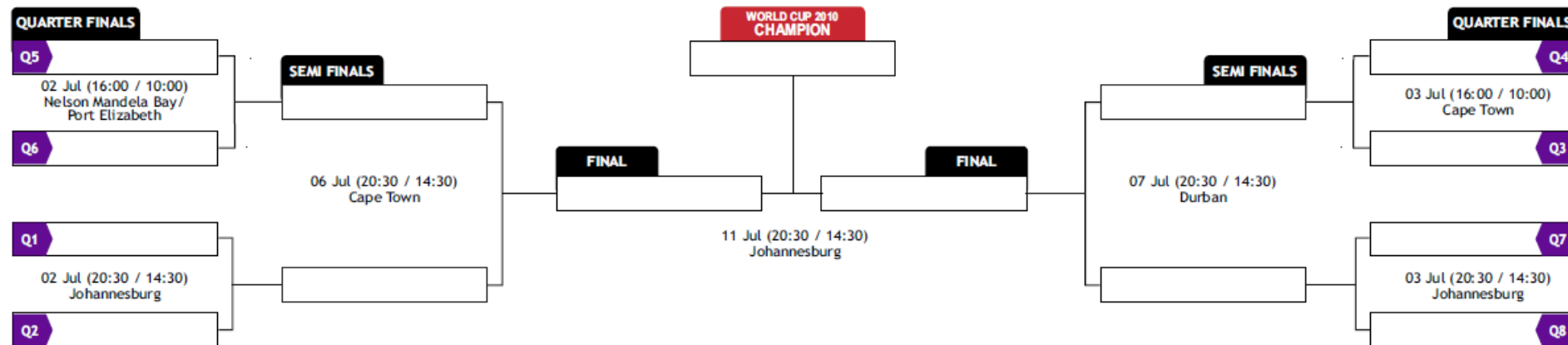
Scan through the input and perform the reduction operation between the result value and the current input value

A parallel reduction tree algorithm performs  $N-1$  Operations in  $\log(N)$  steps





# A tournament is a reduction tree with “max” operation



A more artful rendition of the reduction tree.

# A Quick Analysis

For N input values, the reduction tree performs

- $(1/2)N + (1/4)N + (1/8)N + \dots (1/N) = (1 - (1/N))N = N-1$  operations
- In  $\text{Log}(N)$  steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism  $(N-1)/\text{Log}(N)$ 
  - For  $N = 1,000,000$ , average parallelism is 50,000
  - However, peak resource requirement is 500,000!

This is a work-efficient parallel algorithm

- The amount of work done is comparable to sequential
- Many parallel algorithms are not work efficient

# A Sum Reduction Example

---

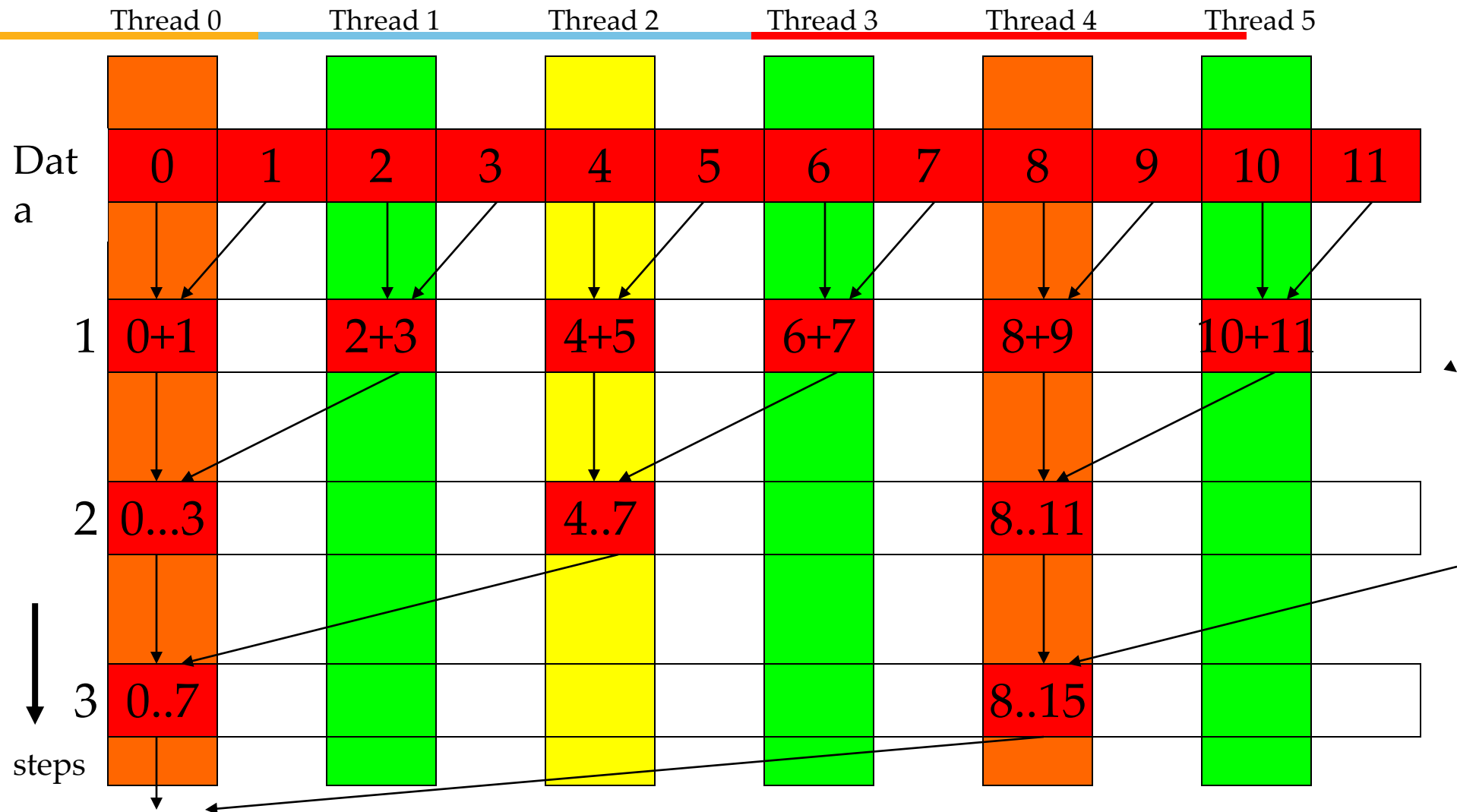
Parallel implementation:

- Recursively halve the # of threads, add two values per thread in each step
- Takes  $\log(n)$  steps for  $n$  elements, requires  $n/2$  threads

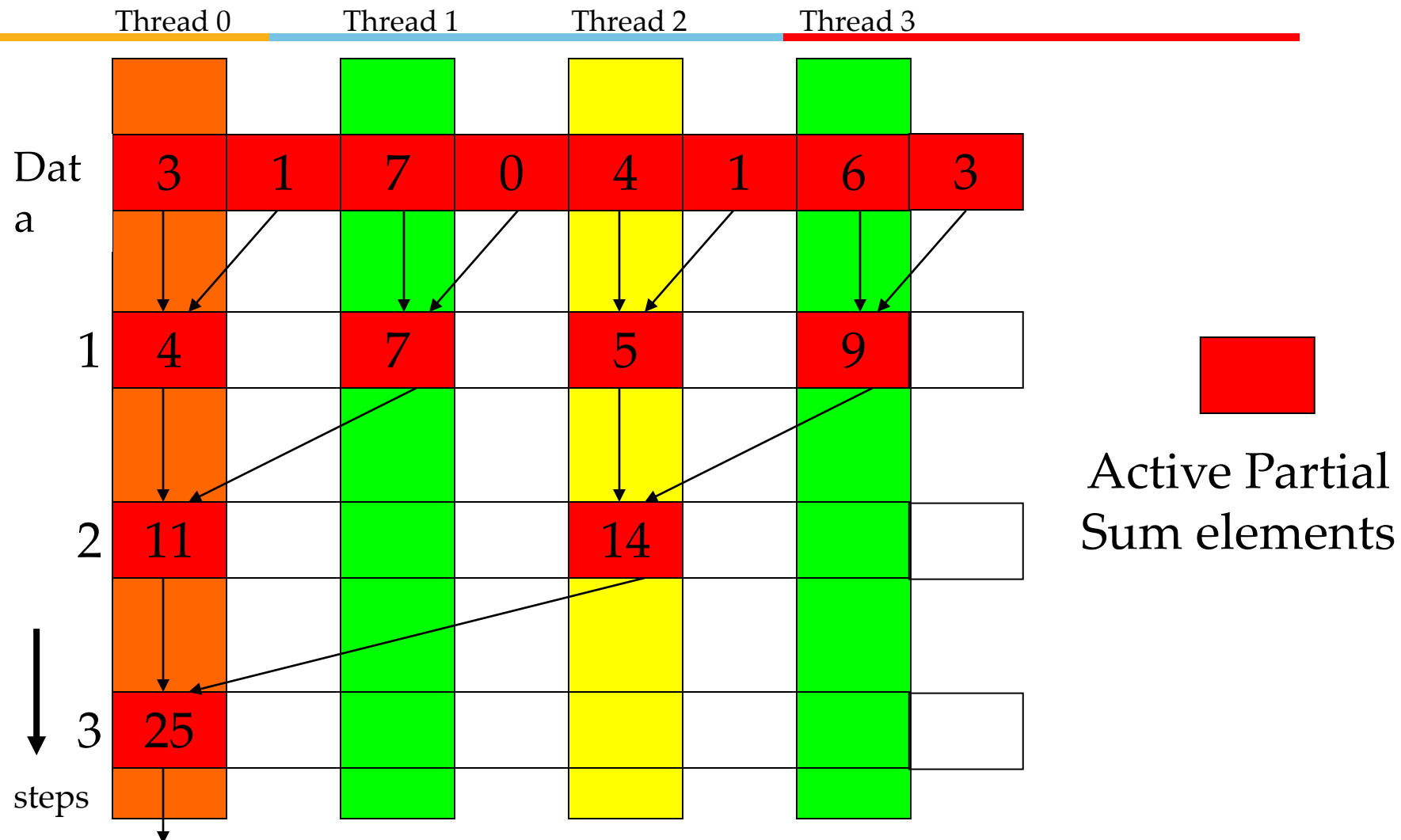
Assume an in-place reduction using shared memory

- The original vector is in device global memory
- The shared memory is used to hold a partial sum vector
- Each step brings the partial sum vector closer to the sum
- The final sum will be in element 0
- Reduces global memory traffic due to partial sum values

# Vector Reduction with Branch Divergence



# A Sum Example



# Simple Thread Index to Data Mapping

---

Each thread is responsible of an even-index location of the partial sum vector

- One input value is at the location of responsibility

After each step, half of the threads are no longer needed

In each step, one of the inputs comes from an increasing distance away



# Objective

---

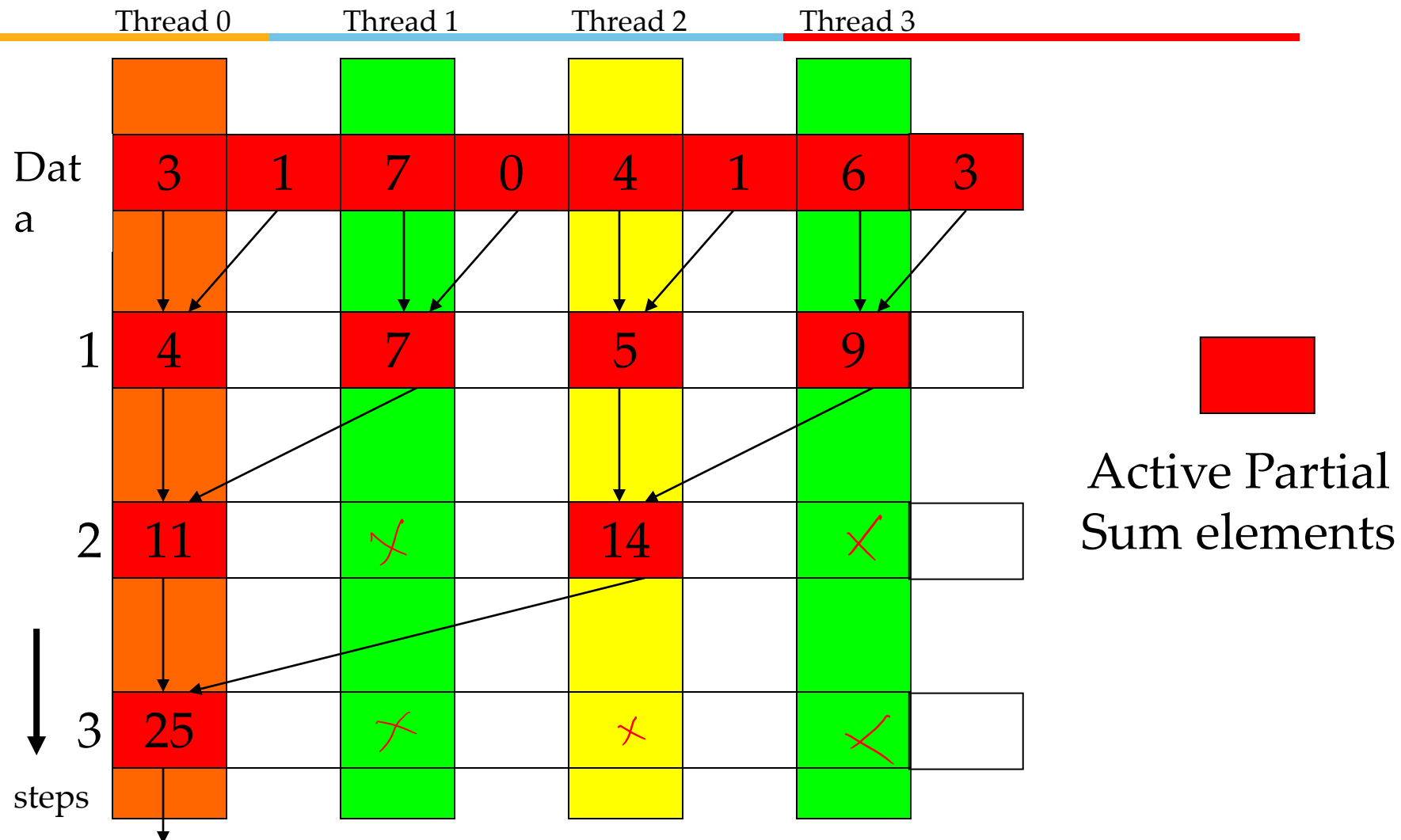
To understand the performance factors of a reduction kernel

- Memory coalescing
- Control divergence
- Thread utilization

To develop a basic kernel and a more optimized kernel



# A Sum Example (review)



# The Reduction Steps

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

Why do we need syncthreads()?

# Barrier Synchronization

---

\_\_syncthreads() are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

Why do we not need another \_\_syncthread() at the end of the reduction loop?

## Back to the Global Picture

Handwritten notes in red ink:

1 0 2 4  
2 4 8  
1 0 2 4

Arrows indicate a sequence or relationship between the numbers.

At the end of the kernel execution, thread 0 in each block writes the sum of the block in `partialSum[0]` into a vector indexed by the value of **blockIdx.x**

There can be a large number of such sums if the original vector is very large

- The host code may iterate and launch another kernel

If there are only a small number of sums, the host can simply transfer the data back and add them together.

Handwritten red mark: a checkmark or similar symbol.

## Some Observations

In each iteration, two control flow paths will be sequentially traversed for each warp

- Threads that perform addition and threads that do not
- Threads that do not perform addition still consume execution resources

No more than half of threads will be executing after the first step

- All odd-index threads are disabled after first step
- After the 5<sup>th</sup> step, entire warps in each block will fail the if-condition, poor resource utilization but no divergence.
  - This can go on for a while, up to 5 more steps ( $1024/32=16=2^5$ ), where each active warp only has one productive thread until all warps in a block retire
- Some warps will still succeed, but with divergence since only one thread will succeed

# Thread Index Usage Matters

---

In some algorithms, one can shift the index usage to improve the divergence behavior

- Commutative and associative operators

Reduction satisfies this criterion.

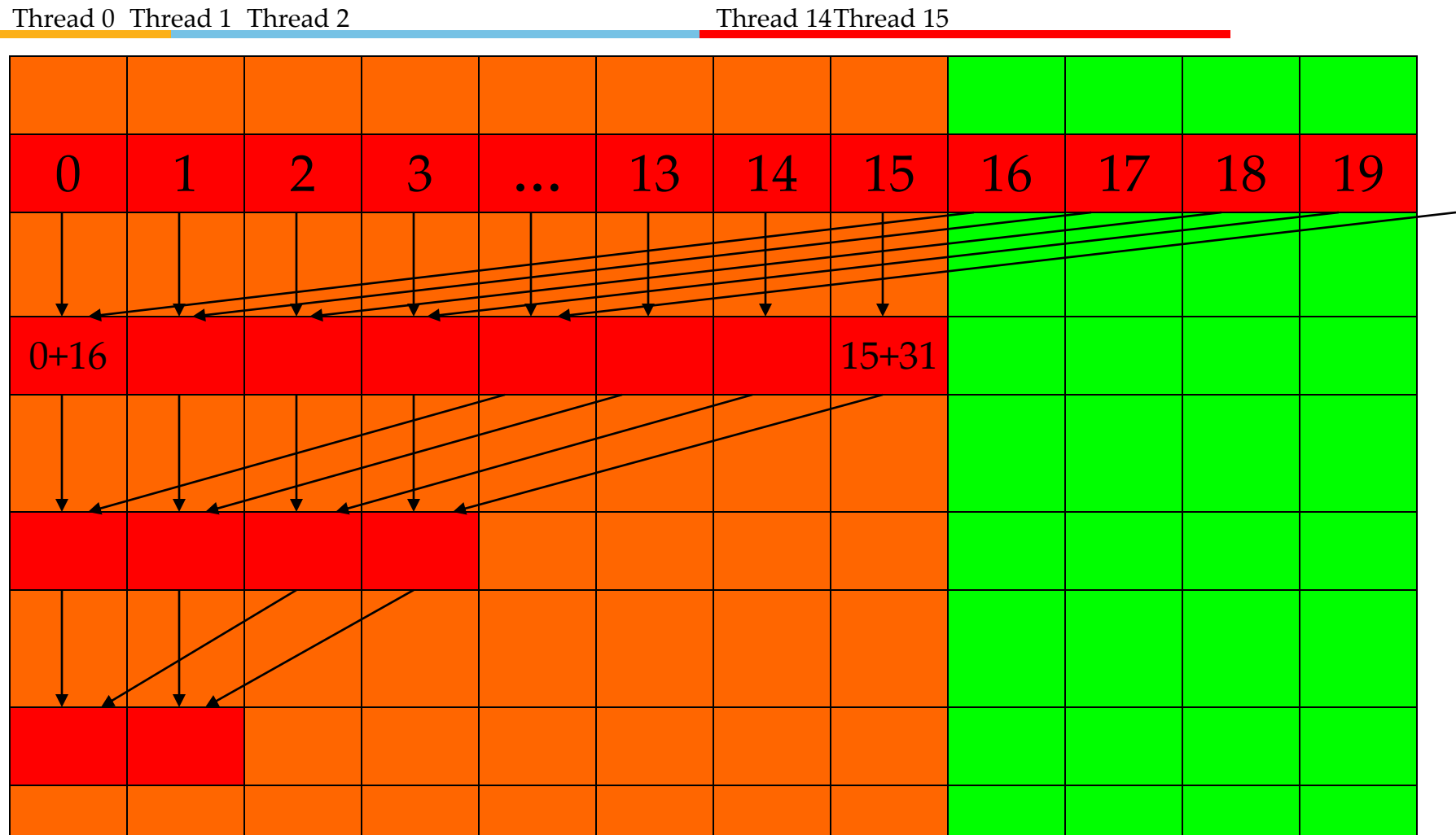
## A Better Strategy

---

Always compact the partial sums into the first locations in the partialSum[] array

Keep the active threads consecutive

# An Example of 16 threads





# A Better Reduction Kernel

---

```
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

# A Quick Analysis

---

For a 1024 thread block

- No divergence in the first 5 steps
- 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
- The final 5 steps will still have divergence

# A Story about an Old Engineer

---

Listen to the recording.

# Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

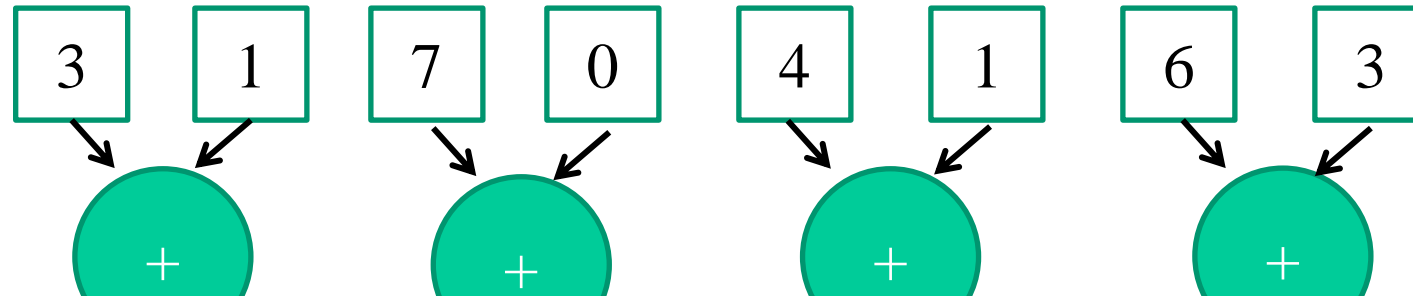
unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

# Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

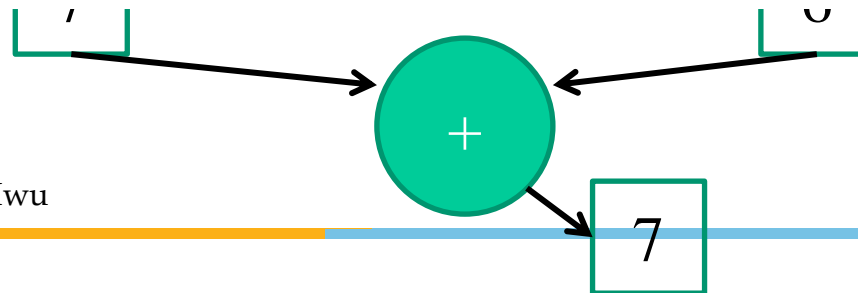
unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

## Parallel Execution Overhead



Although the number of “operations” is  $N$ , each “operation” involves much more ~~complex address calculation and~~ intermediate result manipulation.

If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm.



# Objective

---

To understand atomic operations

- Read-modify-write in parallel computation
- Use of atomic operations in CUDA
- Why atomic operations reduce memory system throughput
- How to avoid atomic operations in some parallel algorithms

Histogramming as an example application of atomic operations

- Basic histogram algorithm
- Privatization

# A Common Collaboration Pattern

---

Multiple bank tellers count the total amount of cash in the safe

Each grab a pile and count

Have a central display of the running total

Whenever someone finishes counting a pile, add the subtotal of the pile to the running total

A bad outcome

- Some of the piles were not accounted for



# A Common Parallel Coordination Pattern

---

Multiple customer service agents serving customers

Each customer gets a number

A central display shows the number of the next customer who will be served

When an agent becomes available, he/she calls the number and he/she adds 1 to the display

Bad outcomes

- Multiple customers get the same number
- Multiple agents serve the same number

# A Common Arbitration Pattern

---

Multiple customers booking air tickets

Each

- Brings up a flight seat map
- Decides on a seat
- Update the the seat map, mark the seat as taken

A bad outcome

- Multiple passengers ended up booking the same seat

# Atomic Operations

thread1:Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

thread2:Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?  
– What does each thread get in their Old variable?

The answer may vary due to data races. To avoid data races, you should use atomic operations

# Timing Scenario #1

Thread 1 Old = 0

Thread 2 Old = 1

Mem[x] = 2 after the sequence

Time	Thread 1	Thread 2
1	(0) Old $\leftarrow$ Mem[x]	
2	(1) New $\leftarrow$ Old + 1	
3	(1) Mem[x] $\leftarrow$ New	
4		(1) Old $\leftarrow$ Mem[x]
5		(2) New $\leftarrow$ Old + 1
6		(2) Mem[x] $\leftarrow$ New

## Timing Scenario #2

Thread 1 Old = 1

Thread 2 Old = 0

Mem[x] = 2 after the sequence

Time	Thread 1	Thread 2
1		(0) Old $\leftarrow$ Mem[x]
2		(1) New $\leftarrow$ Old + 1
3		(1) Mem[x] $\leftarrow$ New
4	(1) Old $\leftarrow$ Mem[x]	
5	(2) New $\leftarrow$ Old + 1	
6	(2) Mem[x] $\leftarrow$ New	

## Timing Scenario #3

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1	(0) Old $\leftarrow$ Mem[x]	
2	(1) New $\leftarrow$ Old + 1	
3		(0) Old $\leftarrow$ Mem[x]
4	(1) Mem[x] $\leftarrow$ New	
5		(1) New $\leftarrow$ Old + 1
6		(1) Mem[x] $\leftarrow$ New

## Timing Scenario #4

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1		(0) Old $\leftarrow$ Mem[x]
2		(1) New $\leftarrow$ Old + 1
3	(0) Old $\leftarrow$ Mem[x]	
4		(1) Mem[x] $\leftarrow$ New
5	(1) New $\leftarrow$ Old + 1	
6	(1) Mem[x] $\leftarrow$ New	

## Atomic Operations – To Ensure Good Outcomes

thread1:Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

thread2Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

Or

thread1:Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

thread2Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New



# Without Atomic Operations

Both threads receive 0  
Mem[x] becomes 1

Mem[x] initialized to 0

thread1: Old  $\leftarrow$  Mem[x]

New  $\leftarrow$  Old + 1

Mem[x]  $\leftarrow$  New

thread2: Old  $\leftarrow$  Mem[x]

New  $\leftarrow$  Old + 1

Mem[x]  $\leftarrow$  New

# Atomic Operations in General

---

Performed by a single ISA instruction on a memory location *address*

- Read the old value, calculate a new value, and write the new value to the location

The hardware ensures that no other threads can access the location until the atomic operation is complete

- Any other threads that access the location will typically be held in a queue until its turn
- All threads perform the atomic operation **serially**

# Atomic Operations in CUDA

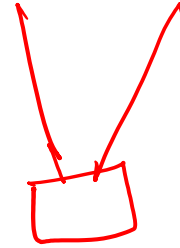
## Atomic Add

*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**.

- Function calls that are translated into single instructions (a.k.a. *intrinsics*)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide 4.0 for details

## More Atomic Adds in CUDA



Unsigned 32-bit integer atomic add

```
unsigned int atomicAdd(unsigned int* address, unsigned int val);
```

Unsigned 64-bit integer atomic add

```
unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);
```

Single-precision floating-point atomic add (capability > 2.0)

```
– float atomicAdd(float* address, float val);
```

# Histogramming

---

A method for extracting notable features and patterns from large data sets

- Feature extraction for object recognition in images
- Fraud detection in credit card transactions
- Correlating heavenly object movements in astrophysics
- ...

Basic histograms - for each element in the data set, use the value to identify a “bin” to increment

# A Histogram Example

---

How do you do this in parallel?

- In sentence “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), ...

# Objective

---

To learn practical histogram programming techniques

- Basic histogram algorithm using atomic operations
- Atomic operation throughput
- Privatization

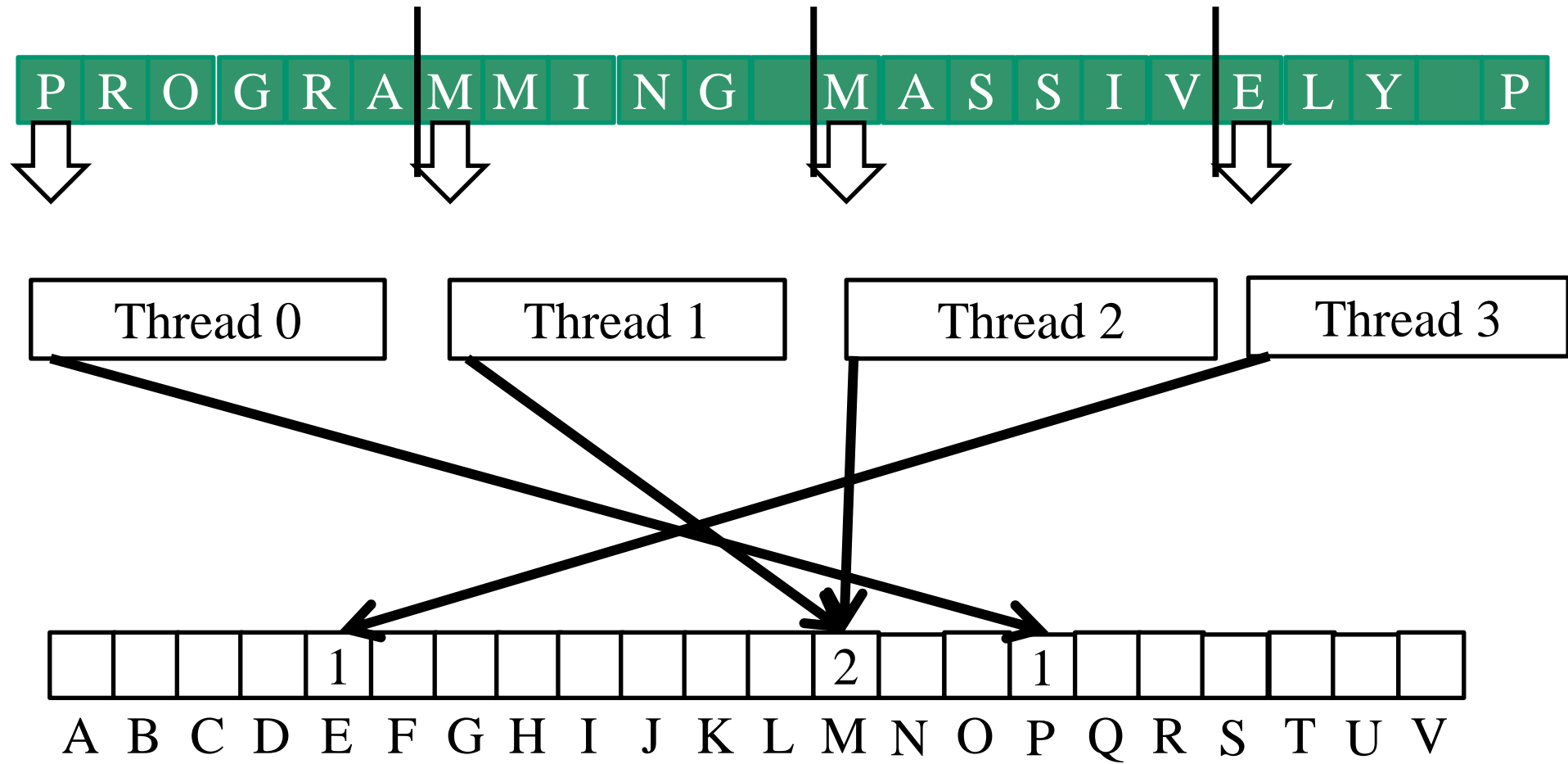
## Review: A Histogram Example

How do you do this in parallel?

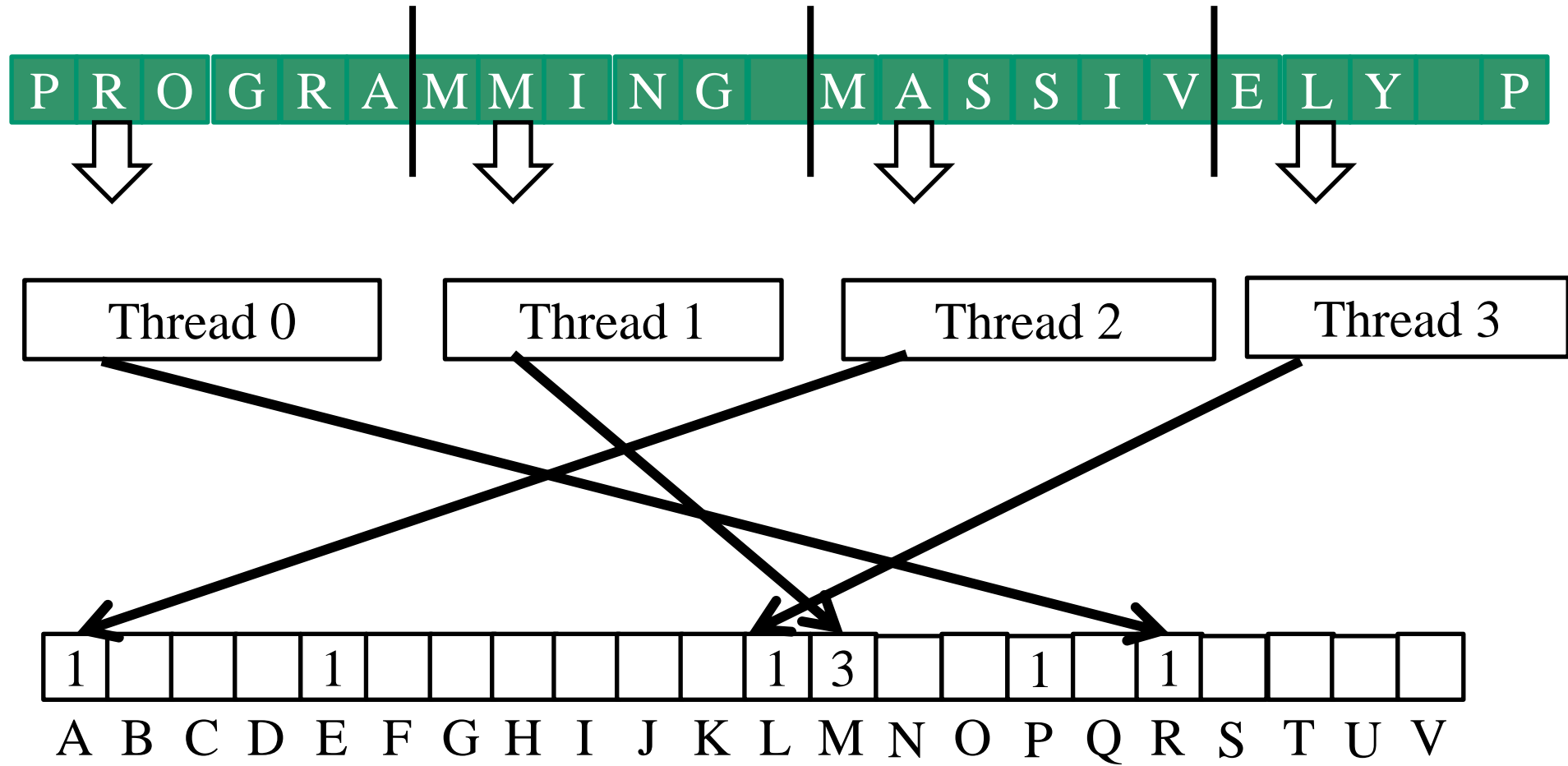
- In phrase “Programming Massively Parallel Processors,” build a histogram of frequencies of each letter
  - Have each thread to take a section of the input
  - For each input letter, use atomic operations to build the histogram
- A(4), C(1), E(1), G(1), ...



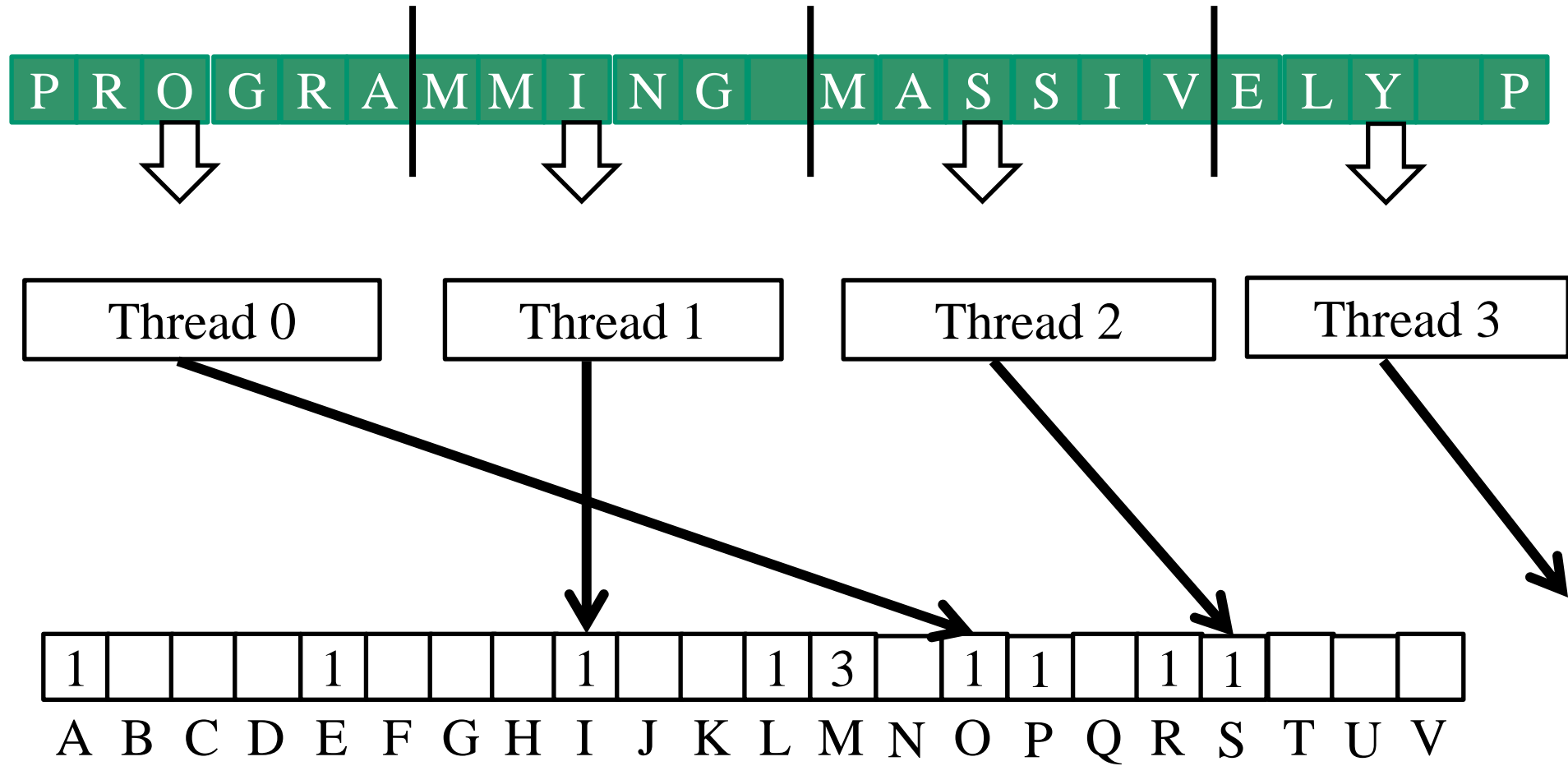
## Iteration #1 – 1<sup>st</sup> letter in each section



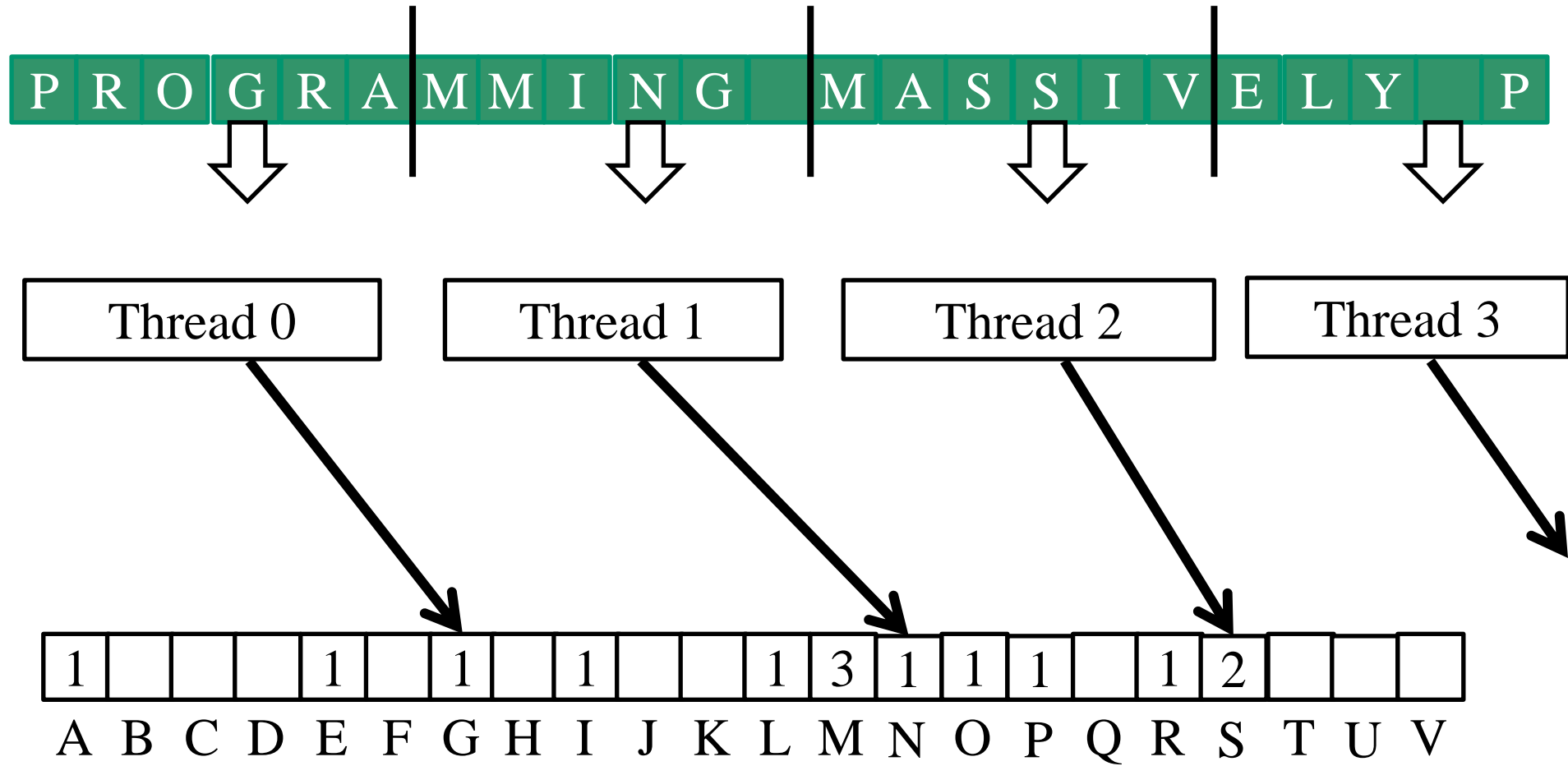
## Iteration #2 – 2<sup>nd</sup> letter in each section



## Iteration #3



## Iteration #4

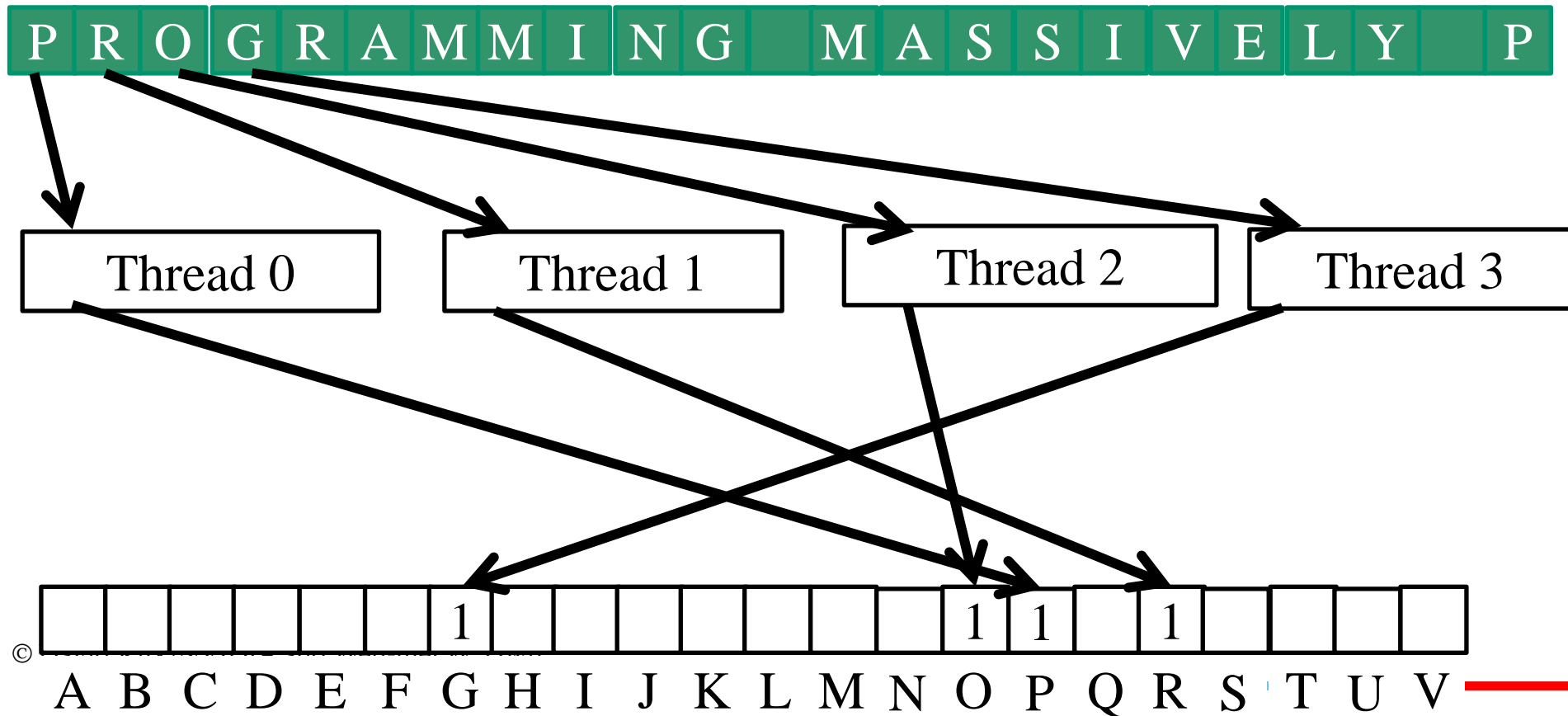




# What is wrong with the algorithm?

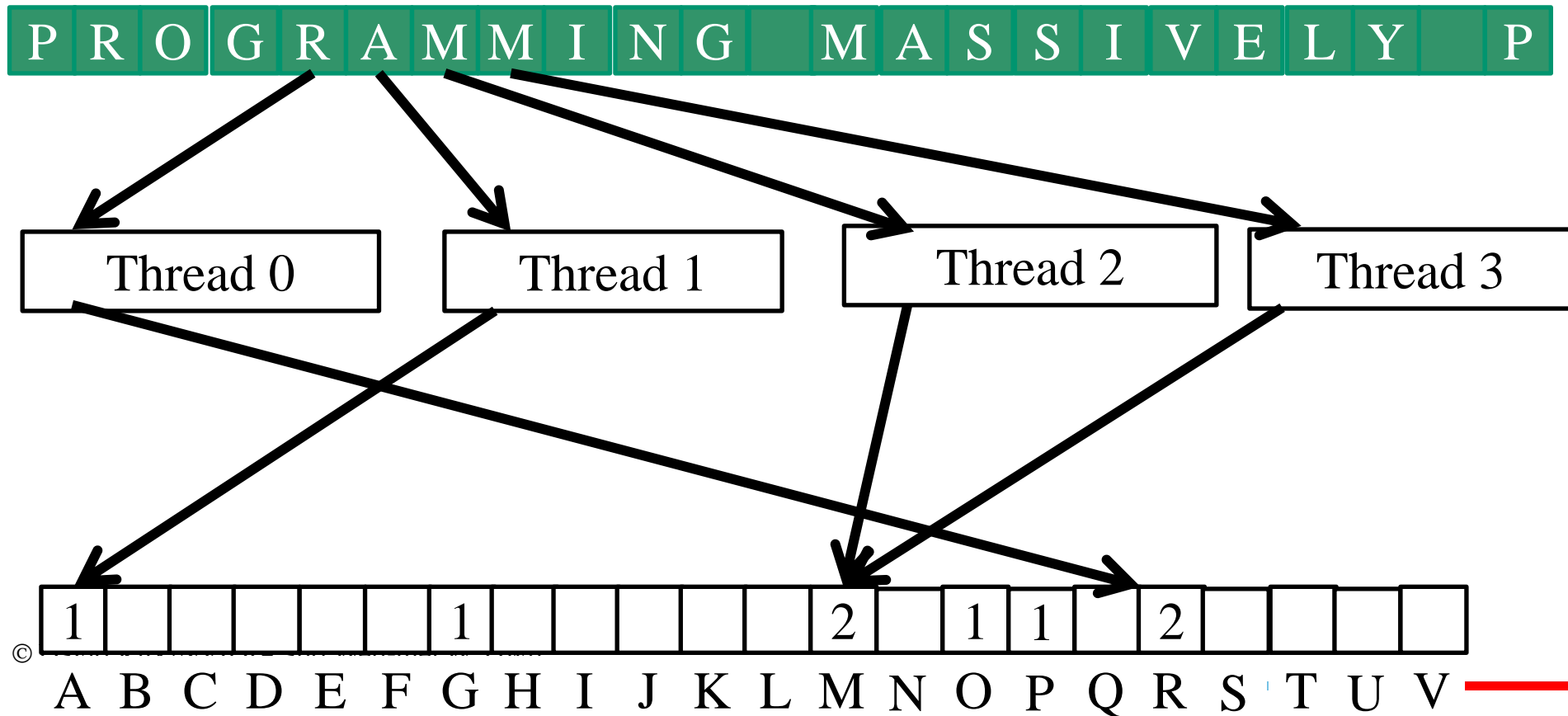
Reads from the input array are not coalesced

- Assign inputs to each thread in a strided pattern
- Adjacent threads process adjacent input letters



## Iteration 2

All threads move to the next section of input



# A Histogram Kernel

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    int i = threadIdx.x + blockDim.x * blockDim.x;  
    // stride is total number of threads  
    int stride = blockDim.x * gridDim.x;
```

- The kernel receives a pointer to the input buffer
- Each thread process the input in a strided pattern

*Handwritten annotations:*

- $CT$  (under `threadIdx.x`)
- $T/B$  (under `blockDim.x`)
- $NB$  (under `blockDim.x`)

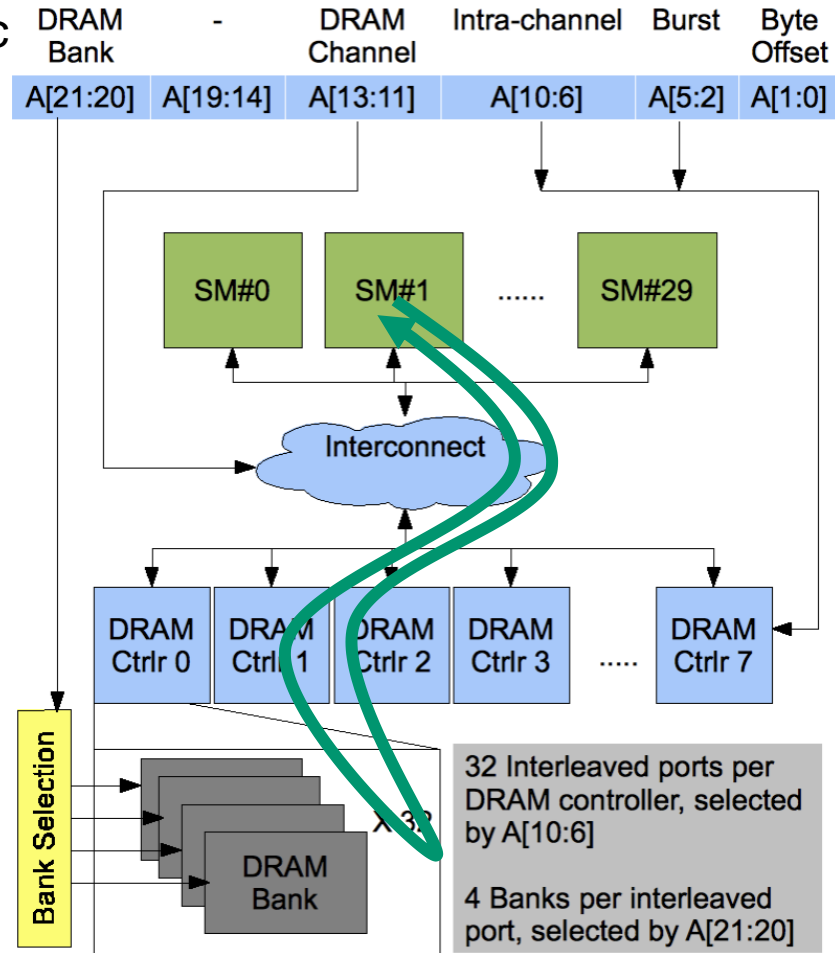


## More on the Histogram Kernel

```
// All threads handle blockDim.x * gridDim.x
// consecutive elements
while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1);
    i += stride;
}
}
```

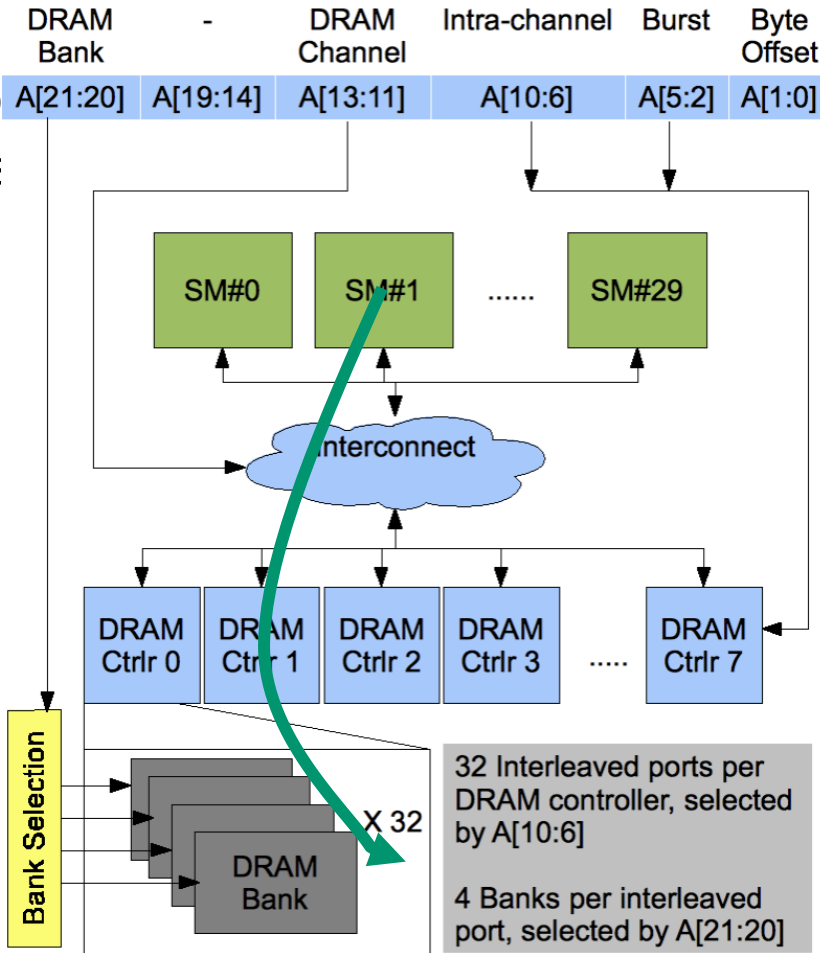
# Atomic Operations on DRAM

An atomic operation of a few hundred cycles



# Atomic Operations on DRAM

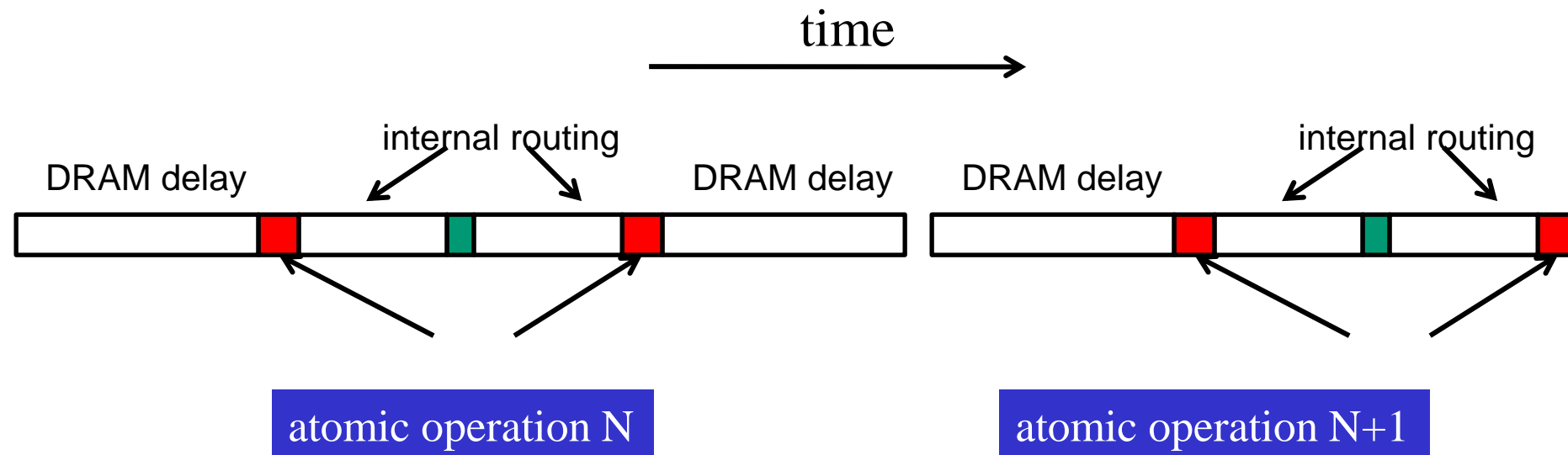
An atomic operation is a sequence of a few hundred cycles  
 The atomicity of a few hundred cycles  
 During this time the location



# Atomic Operations on DRAM

Each Load-Modify-Store has two full memory access delays

- All atomic operations on the same variable (RAM location) are serialized



# Latency determines throughput of atomic operations

---

Throughput of an atomic operation is the rate at which the application can execute an atomic operation on a particular location.

The rate is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.

This means that if many threads attempt to do atomic operation on the same location (contention), the memory bandwidth is reduced to  $< 1/1000$ !

# You may have a similar experience in supermarket checkout

---

Some customers realize that they missed an item after they started to check out

They run to the isle and get the item while the line waits

- The rate of check is reduced due to the long latency of running to the isle and back.

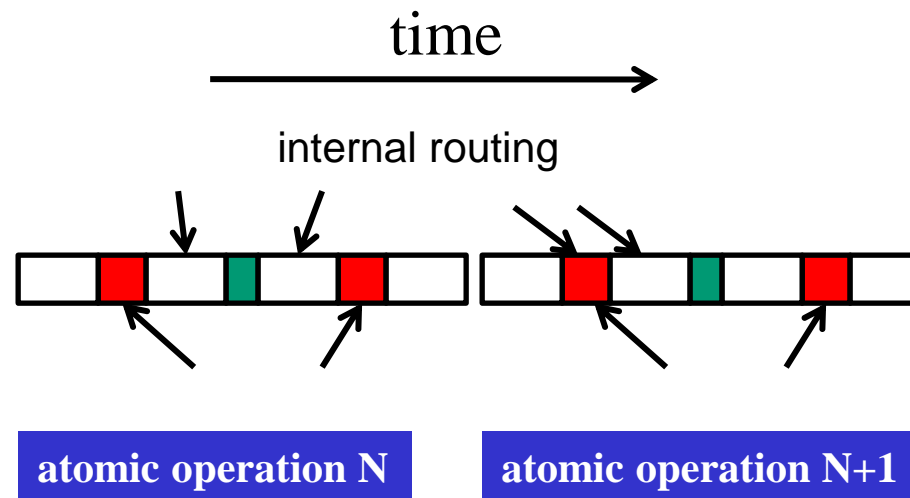
Imagine a store where every customer starts the check out before they even fetch any of the items

- The rate of the checkout will be  $1 / (\text{entire shopping time of each customer})$

## Hardware Improvements (cont.)

### Atomic operations on Fermi L2 cache

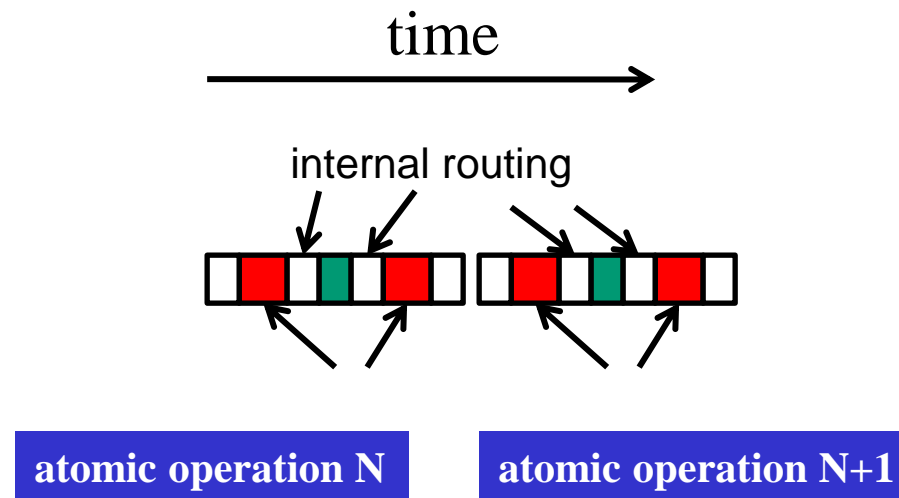
- medium latency, but still serialized
- Global to all blocks
- “Free improvement” on Global Memory atomics



# Hardware Improvements

## Atomic operations on Shared Memory

- Very short latency, but still serialized
- Private to each thread block
- Need algorithm work by programmers (more later)

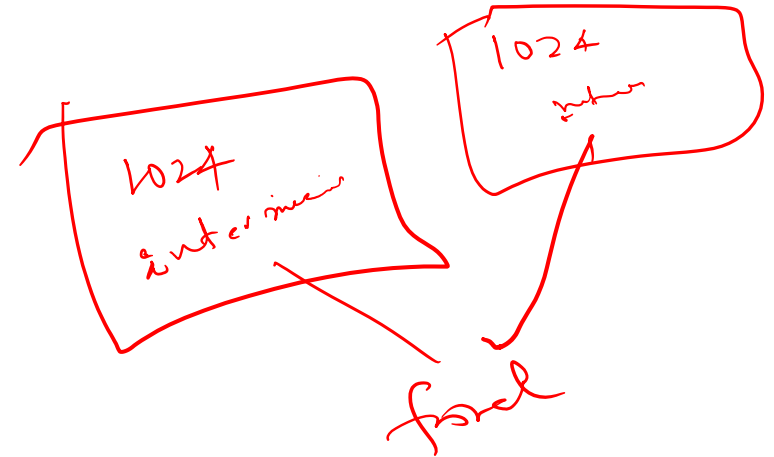




# Atoms in Shared Memory Requires Privatization

Create private copies of the histo[] array for each thread block.

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    __shared__ unsigned int histo_private[256];  
    if (threadIdx.x < 256) histo_private[threadIdx.x] = 0;  
    __syncthreads();
```



# Build Private Histogram

---

```
int i = threadIdx.x + blockIdx.x * blockDim.x;  
// stride is total number of threads  
int stride = blockDim.x * gridDim.x;  
while (i < size) {  
    atomicAdd( &(amp;private_histo[buffer[i]]), 1);  
    i += stride;  
}
```

# Build Final Histogram

---

```
// wait for all other threads in the block to finish
__syncthreads();

if (threadIdx.x < 256)
    atomicAdd( &(histo[threadIdx.x]),
               private_histo[threadIdx.x] );
}
```

# More on Privatization

---

Privatization is a powerful and frequently used techniques for parallelizing applications

The operation needs to be associative and commutative

- Histogram add operation is associative and commutative

The histogram size needs to be small

- Fits into shared memory

What if the histogram is too large to privatize?



## Additional Slide

Not part of the evaluation



# Objective

---

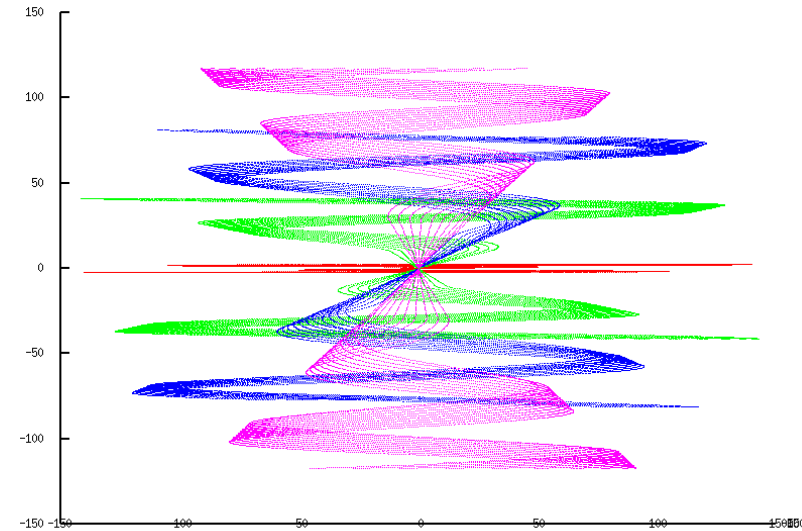
To learn the key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth

- better utilization of on-chip memory
- fewer bytes transferred to on-chip memory
- retaining regularity

# Sparse Data

## Motivation for Compaction

- Many real-world inputs are sparse/non-uniform
- Signal samples, mesh models, transportation networks, communication networks, etc.



# Sparse Matrix

---

Many real-world systems are sparse in nature

- Solving these systems require inversion of the coefficient matrix
- Traditional inversion algorithms such as Gaussian elimination can create too many “fill-in” elements and explode the size of the matrix
- Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication is preferred

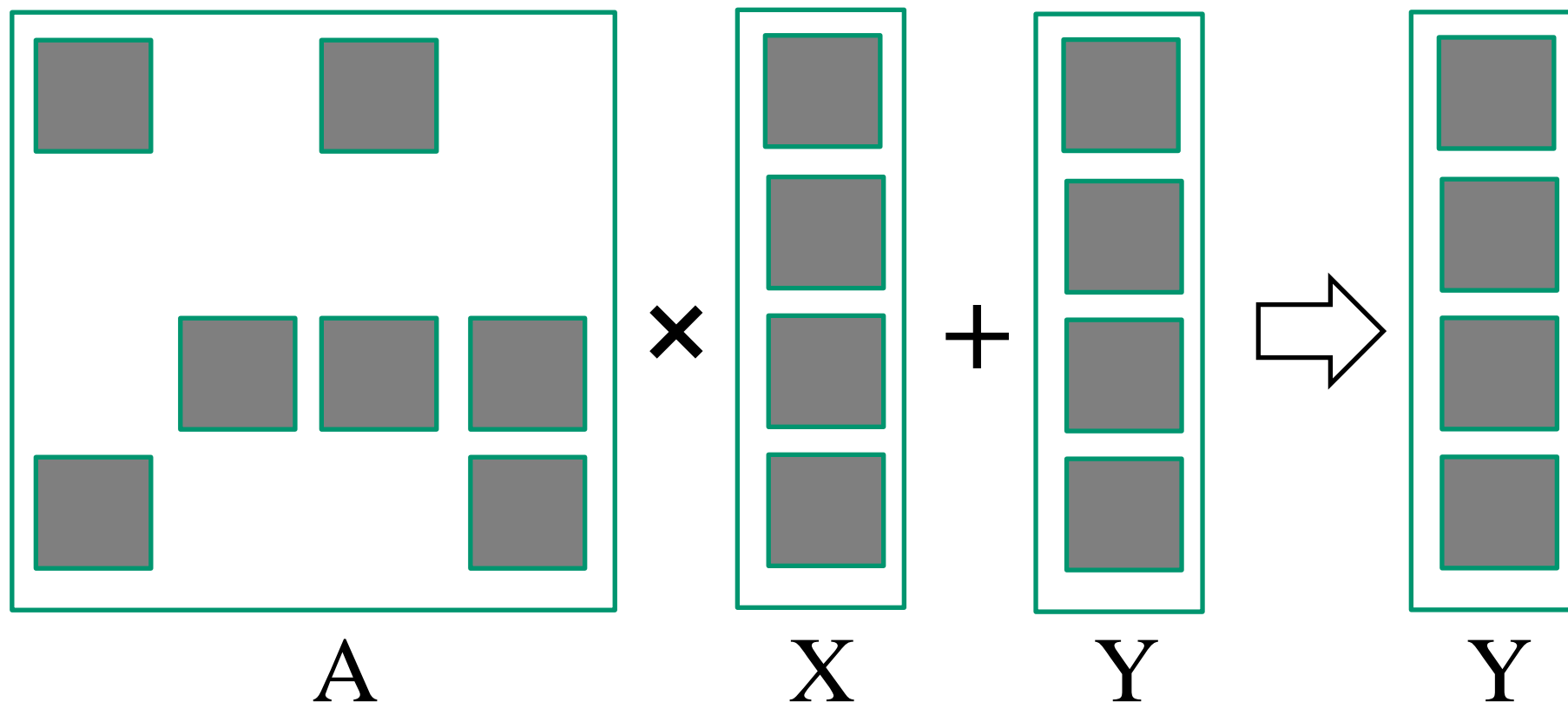
Solution of PDE systems can be formulated into linear operations using sparse matrix-vector multiplication



Science Area	Number of Teams	Codes	Struct Grids	Unstruct Grids	Dense Matrix	Sparse Matrix	N-Body	Monte Carlo	FFT	PIC	Sig I/O
Climate and Weather	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
Plasmas/ Magnetosphere	2	H3D(M), VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
Stellar Atmospheres and Supernovae	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
Cosmology	2	Enzo, pGADGET	X			X	X				
Combustion/ Turbulence	2	PSDNS, DISTUF	X						X		
General Relativity	2	Cactus, Harm3D, LazEV	X			X					
Molecular Dynamics	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
Quantum Chemistry	2	SIAL, GAMESS, NWChem			X	X	X	X			X
Material Science	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
Earthquakes/ Seismology	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
Quantum Chromo Dynamics	1	Chroma, MILC, USQCD	X		X	X					
Social Networks	1	EPISIMDEMICS									
Evolution	1	Eve									
Engineering/System of Systems	1	GRIPS, Revisit						X			
Computer Science	1			X	X	X			X		X

© Wen-mei W. Hwu and David Kirk/NVIDIA, 2010-2016

# Sparse Matrix-Vector Multiplication (SpMV)



# Challenges

---

Compared to dense matrix multiplication, SpMV

- Is irregular/unstructured
- Has little input data reuse
- Benefits little from compiler transformation tools

Key to maximal performance

- Maximize regularity (by reducing divergence and load imbalance)
- Maximize DRAM burst utilization (layout arrangement)

## A Simple Parallel SpMV

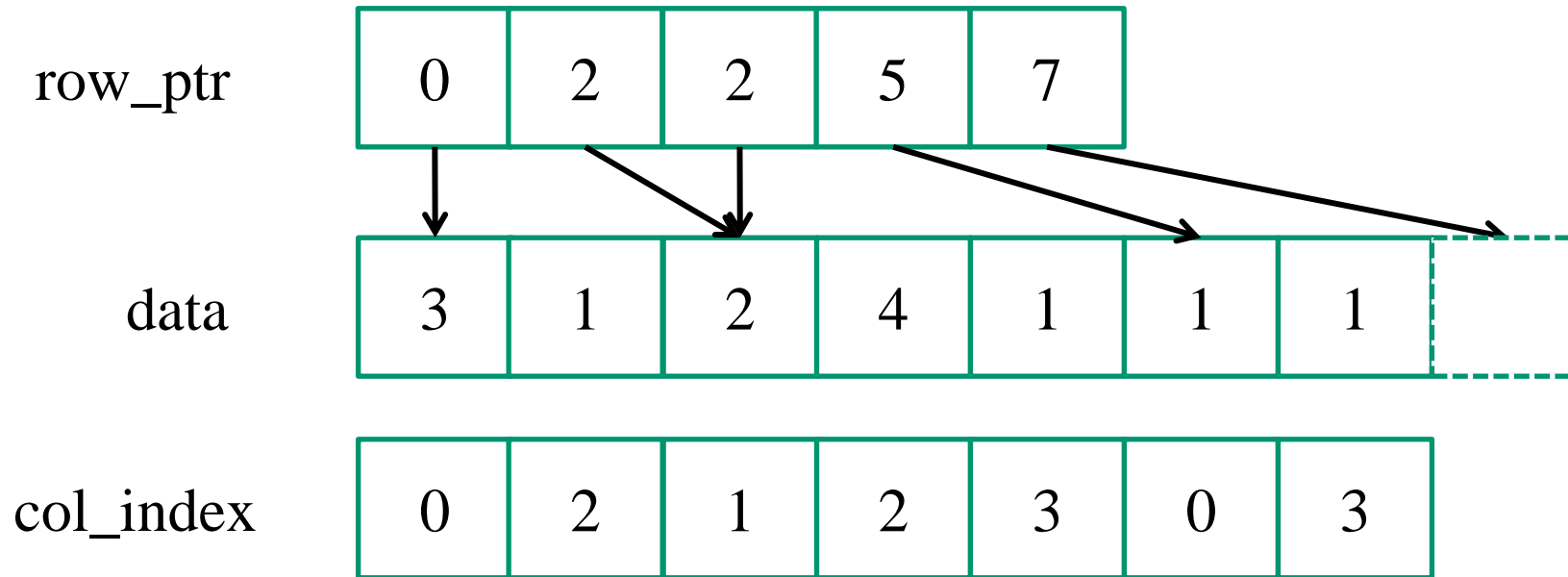
Each thread processes one row

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

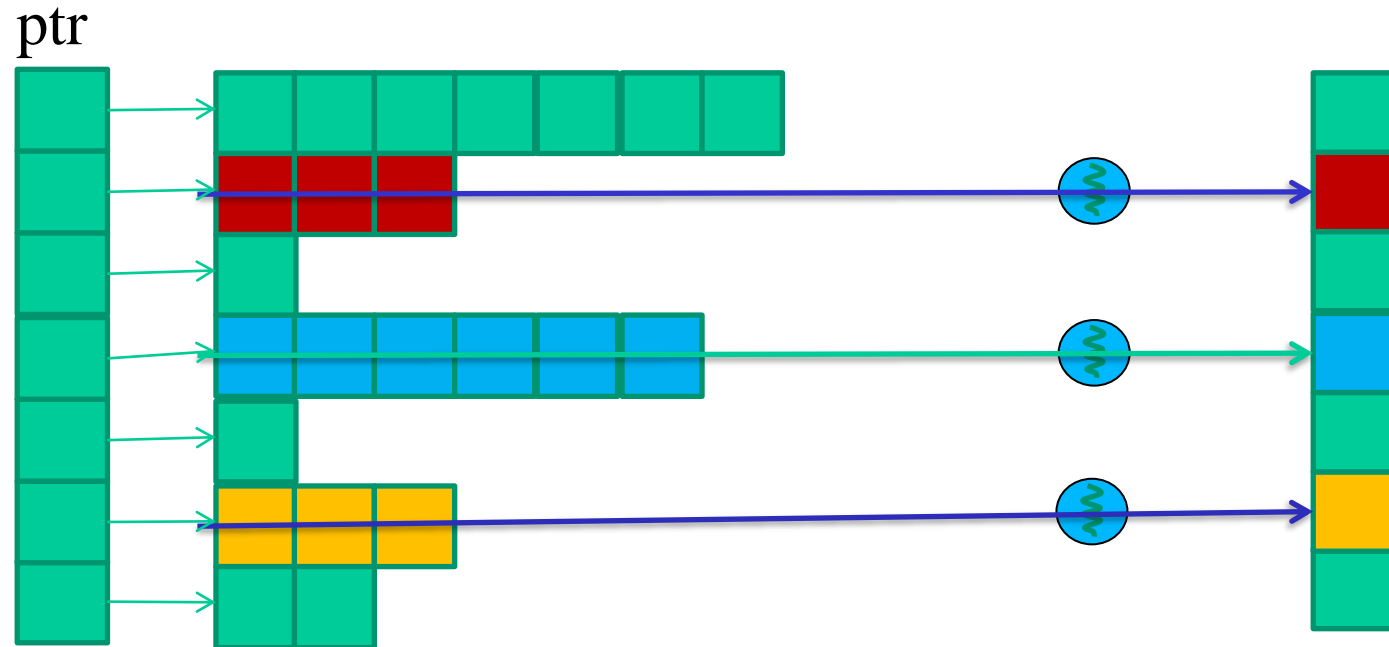
# Compressed Sparse Row (CSR) Format

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers	ptr[5]	{ 0, 2,	2, 5, 7 }	

# CSR Data Layout



# CSR Kernel Design



# A Parallel SpMV/CSR Kernel (CUDA)

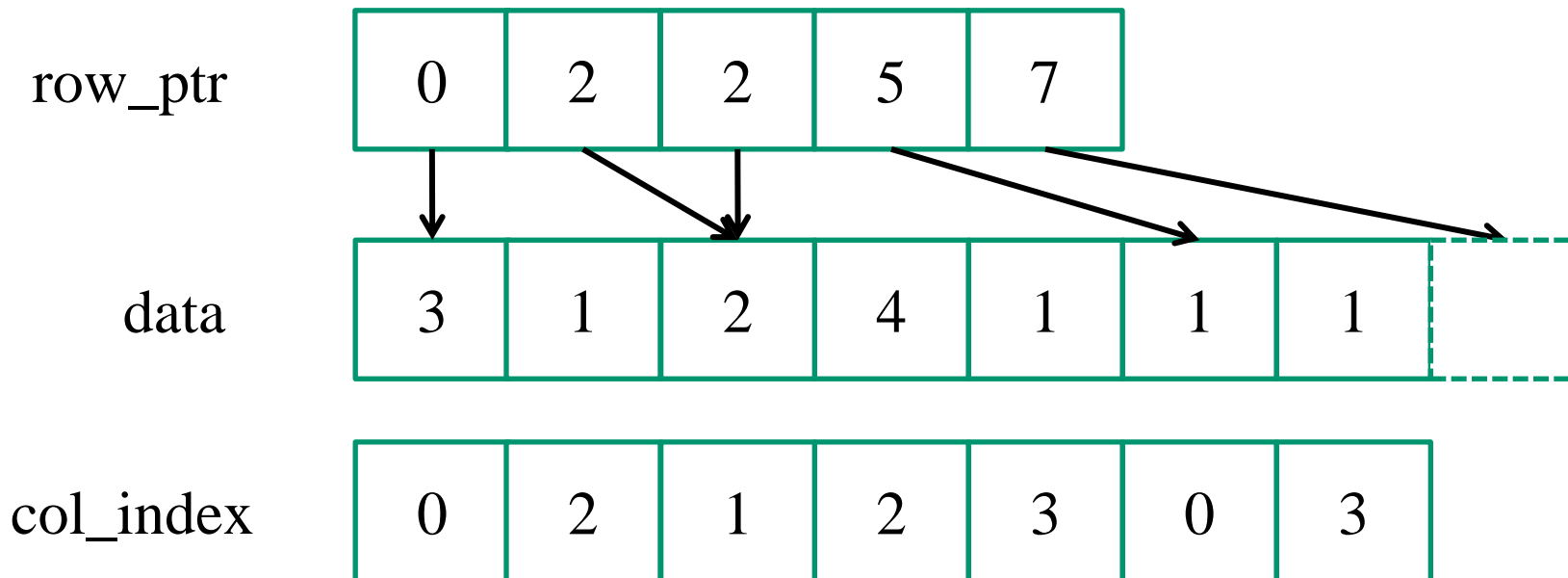
```
1. __global__ void SpMV_CSR(int num_rows, float *data,  
    int *col_index, int *row_ptr, float *x, float *y) {  
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;  
3.     if (row < num_rows) {  
4.         float dot = 0;  
5.         int row_start = row_ptr[row];  
6.         int row_end = row_ptr[row+1];  
7.         for (int elem = row_start; elem < row_end; elem++) {  
8.             dot += data[elem] * x[col_index[elem]];  
9.         }  
        y[row] = dot;  
    }  
}
```

	Row 0	Row 2	Row 3
Nonzero values data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers row_ptr[5]	{ 0, 2,	2, 5, 7 }	



# CSR Kernel Control Divergence

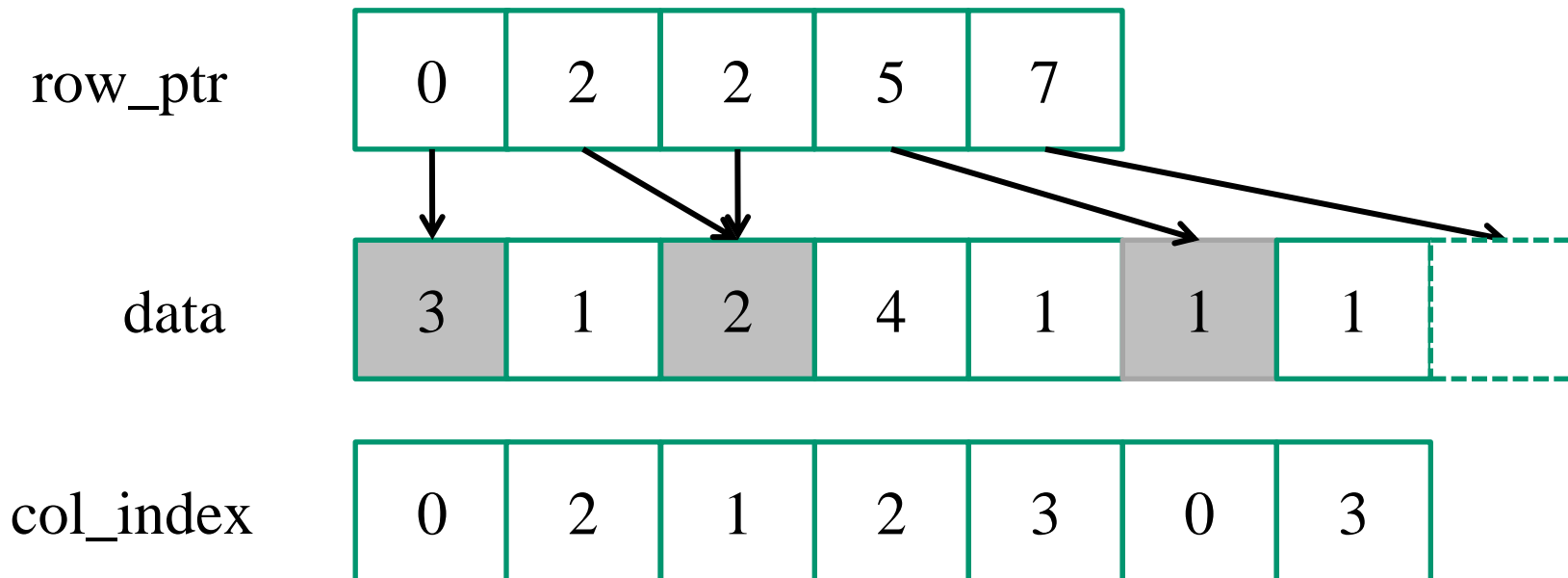
Threads execute different number of iterations in the kernel for-loop



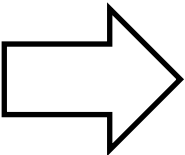
# CSR Kernel Memory Divergence

Adjacent threads access non-adjacent memory locations

- Grey elements are accessed by all threads in iteration 0



# Regularizing SpMV with ELL(PACK) Format



3	1	*
*	*	*
2	4	1
1	1	*

CSR with Padding

Pad all rows to the same length

- Inefficient if a few rows are much longer than others

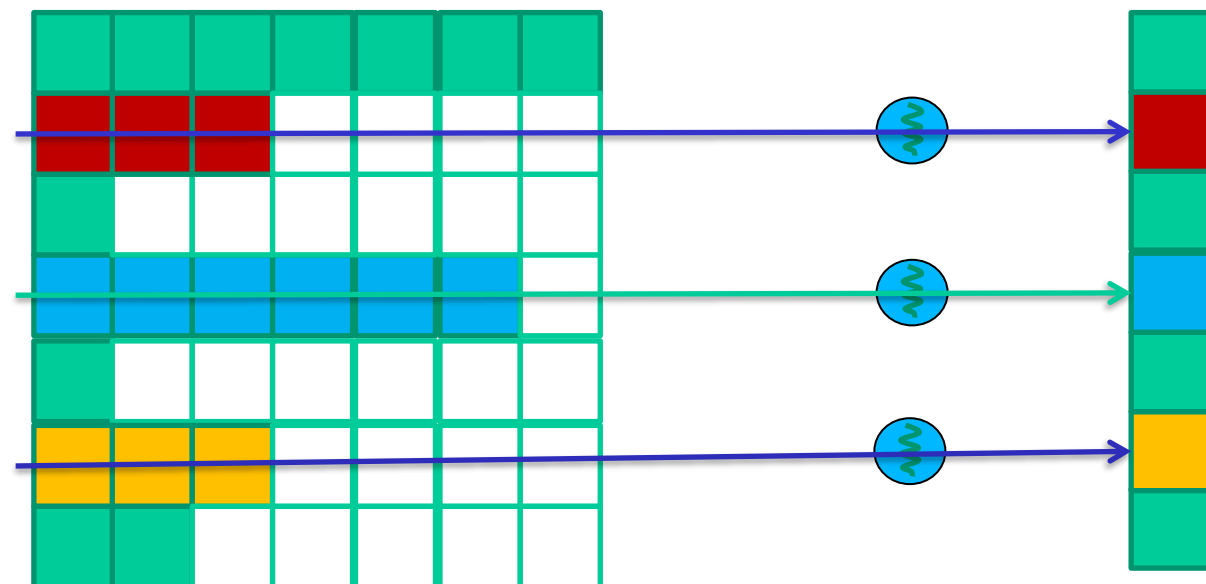
Transpose (Column Major) for DRAM efficiency

Both data and col\_index padded/transposed

Thread 0	Thread 1	Thread 2	Thread 3
3	*	2	1
1	*	4	1
*	*	1	*

Transposed

# ELL Kernel Design

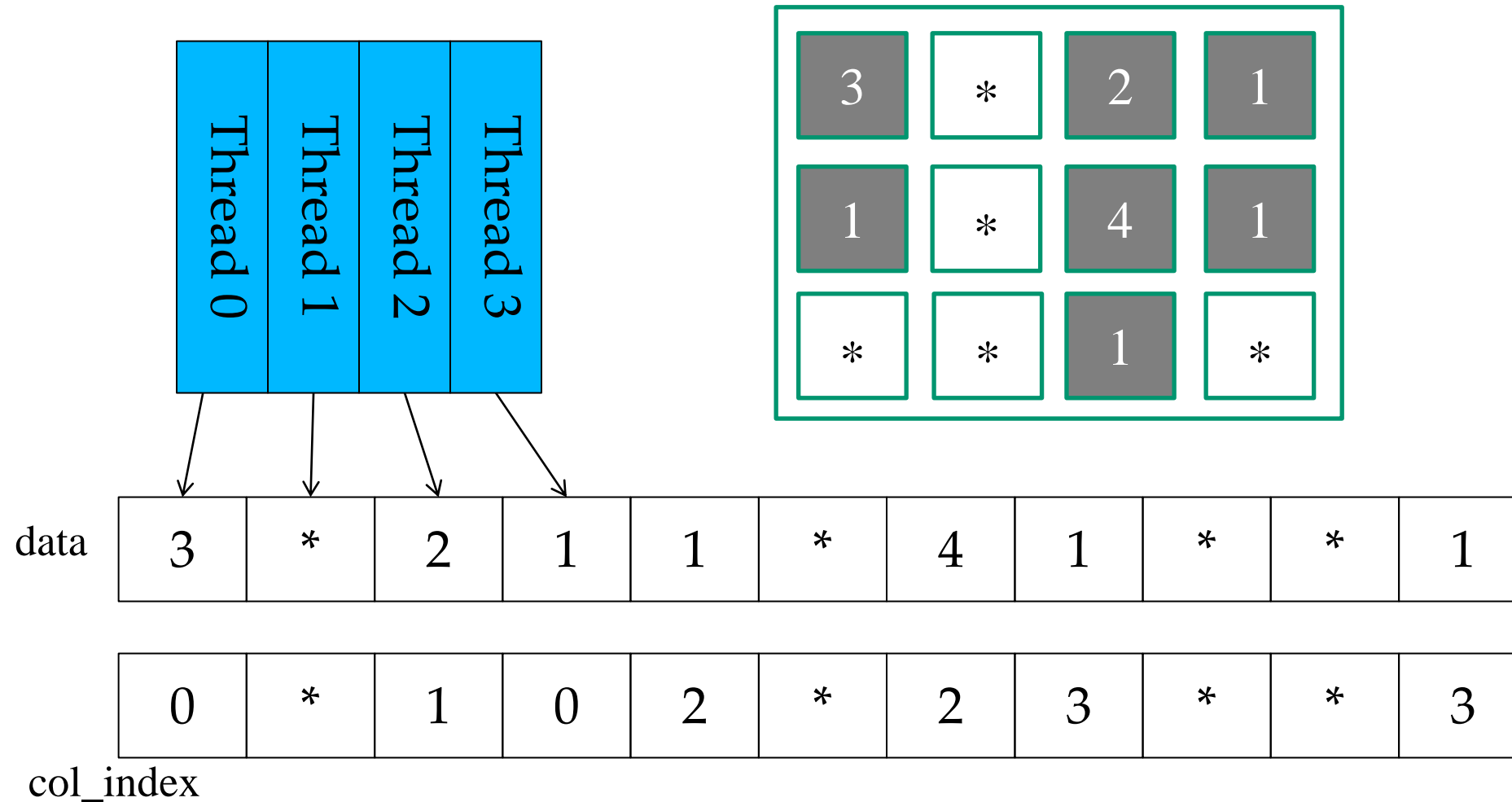


## A parallel SpMV/ELL kernel

```
1. __global__ void SpMV_ELL(int num_rows, float *data,
    int *col_index, int num_elem, float *x, float *y) {

2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         for (int i = 0; i < num_elem; i++) {
6.             dot += data[row+i*num_rows]*x[col_index[row+i*num_rows]];
7.             }
8.         y[row] = dot;
9.     }
10. }
```

# Memory Coalescing with ELL



# Coordinate (COO) format

Explicitly list the column and row indices for every non-zero element

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

## COO Allows Reordering of Elements

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

Nonzero values	data[7]	{ 1 1, 2, 4, 3, 1 1 }
Column indices	col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
Row indices	row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }



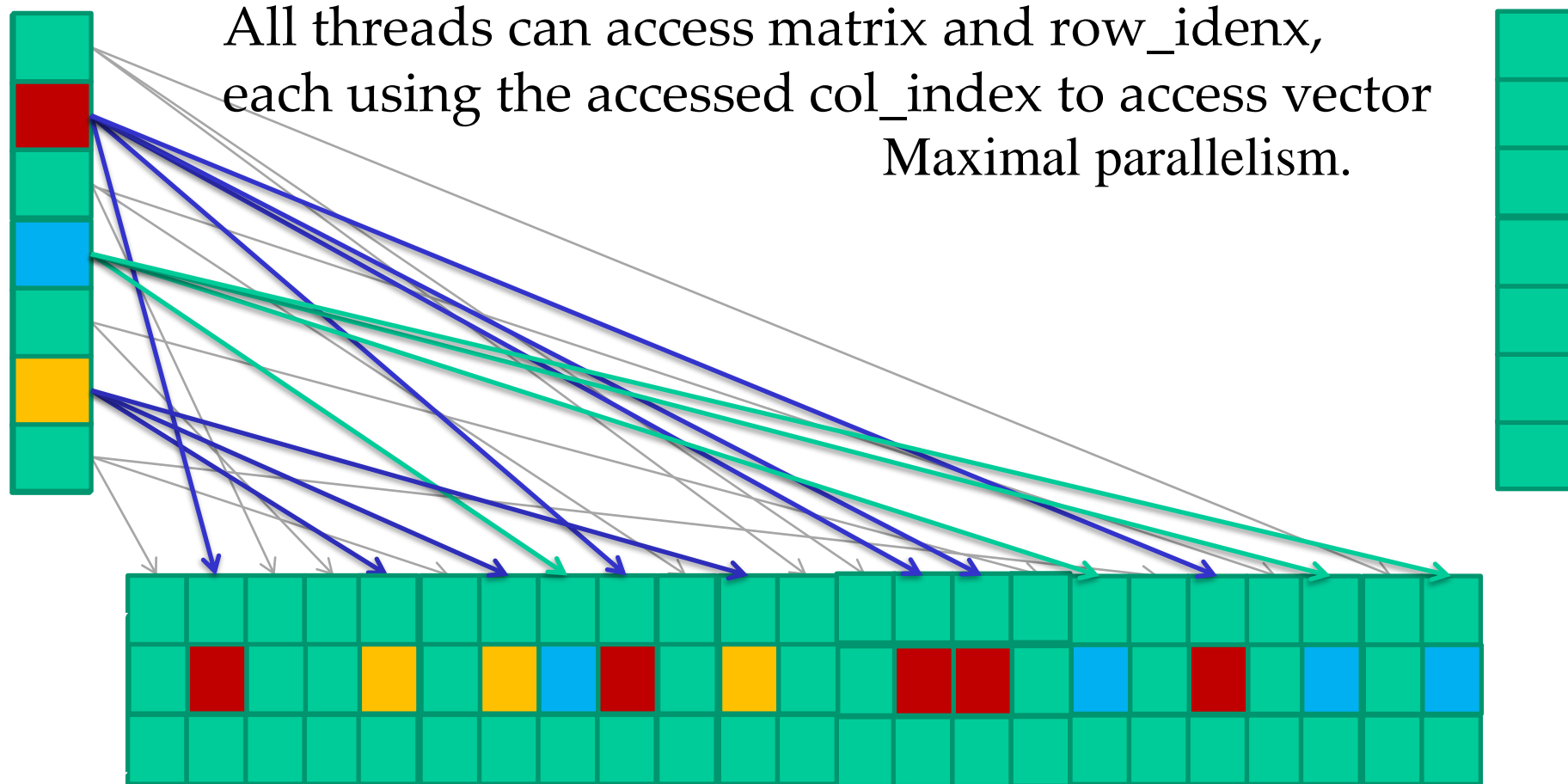
---

```
1.    for (int i = 0; i < num_elem; row++)
2.        y[row_index[i]] += data[i] * x[col_index[i]];
```

a sequential loop that implements SpMV/COO

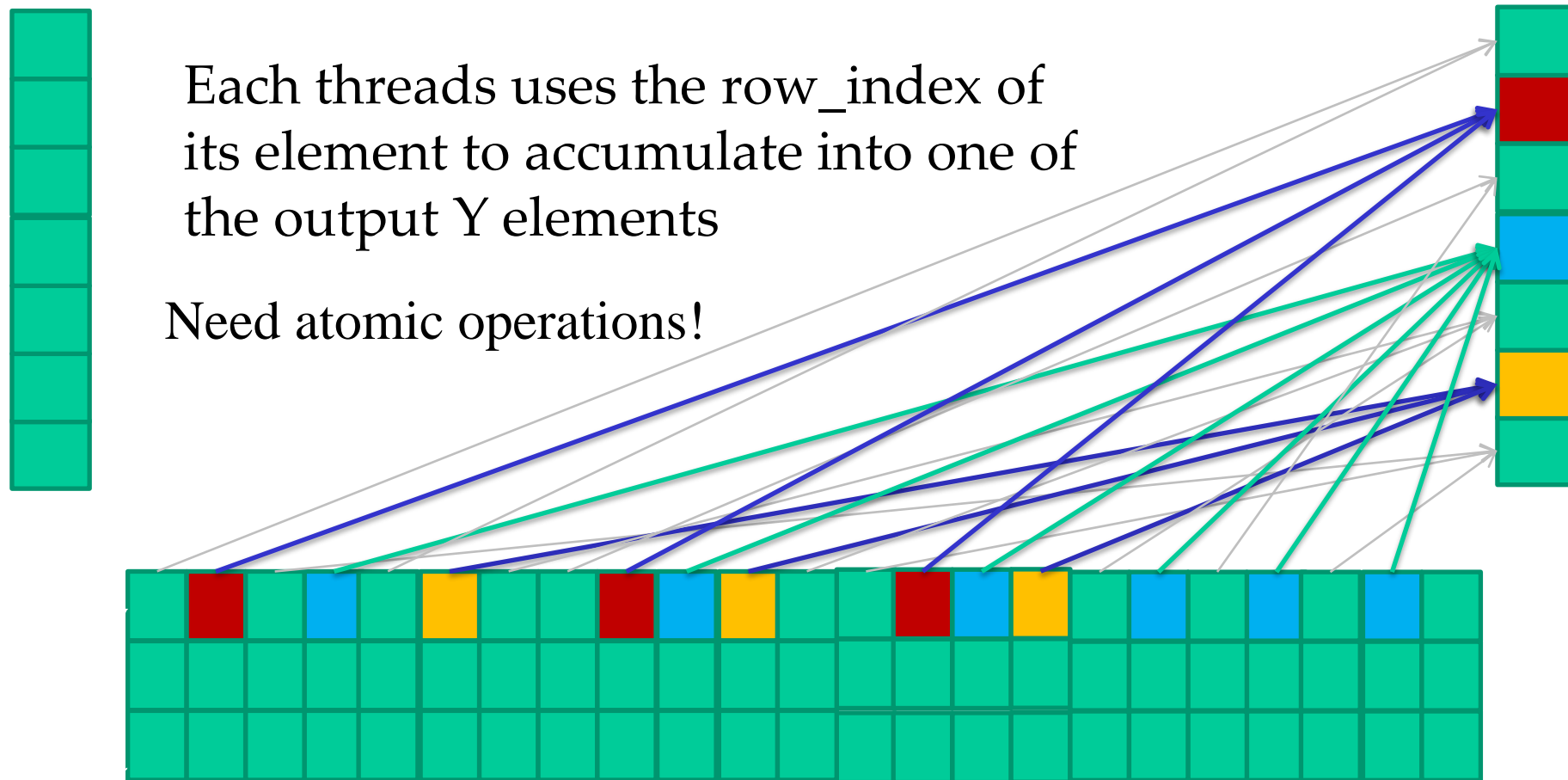
# COO Kernel Design

## Accessing Input Matrix and Vector



# COO kernel Design

## Accumulating into Output Vector

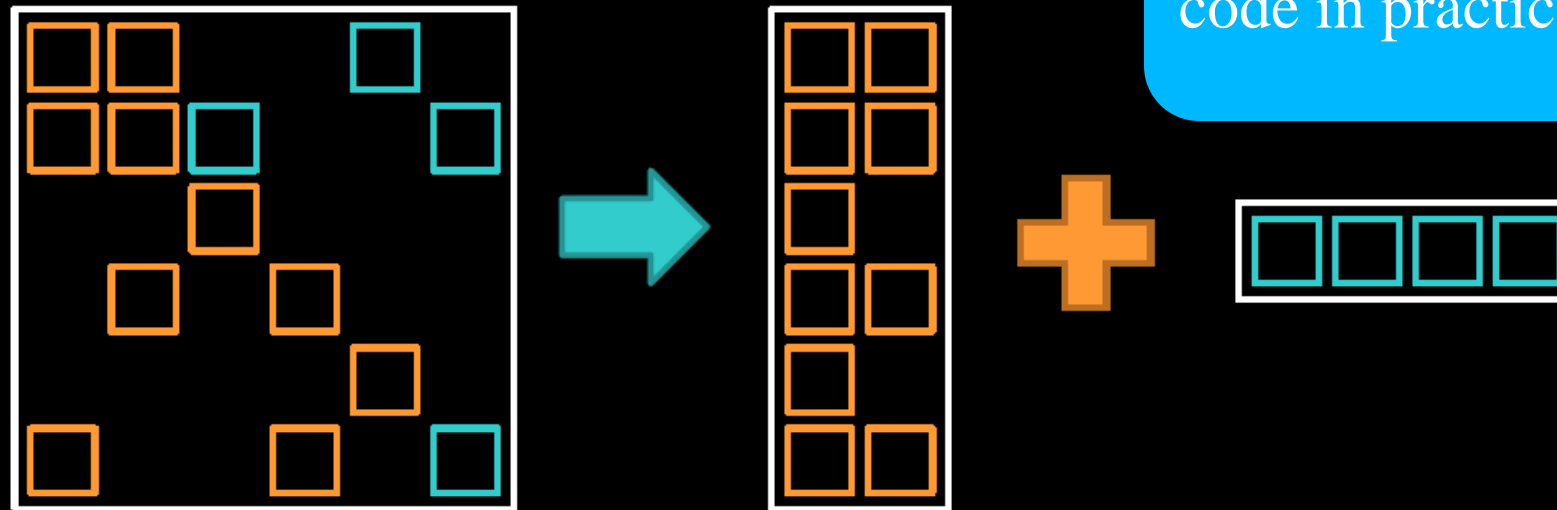


# Hybrid Format

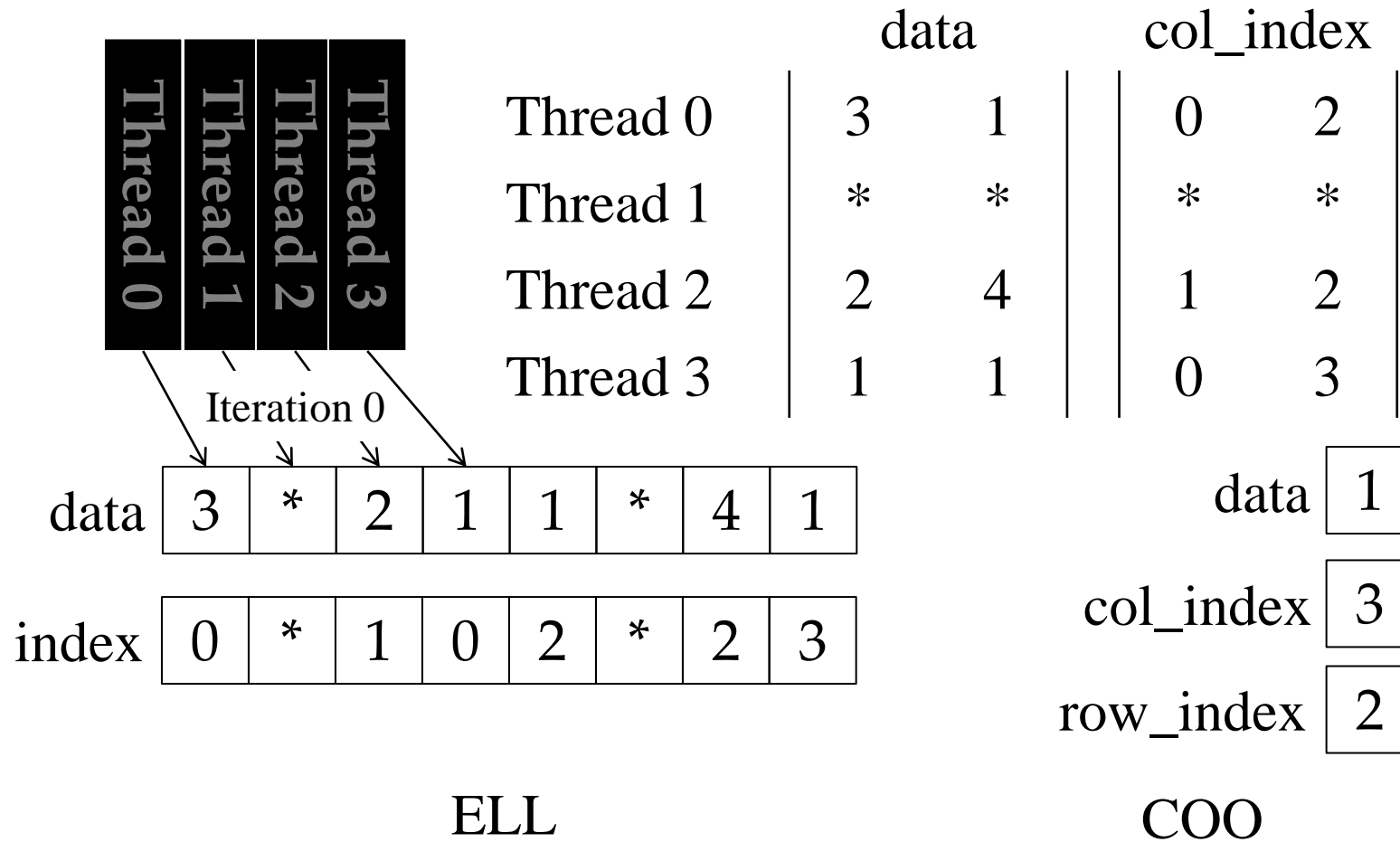


- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction

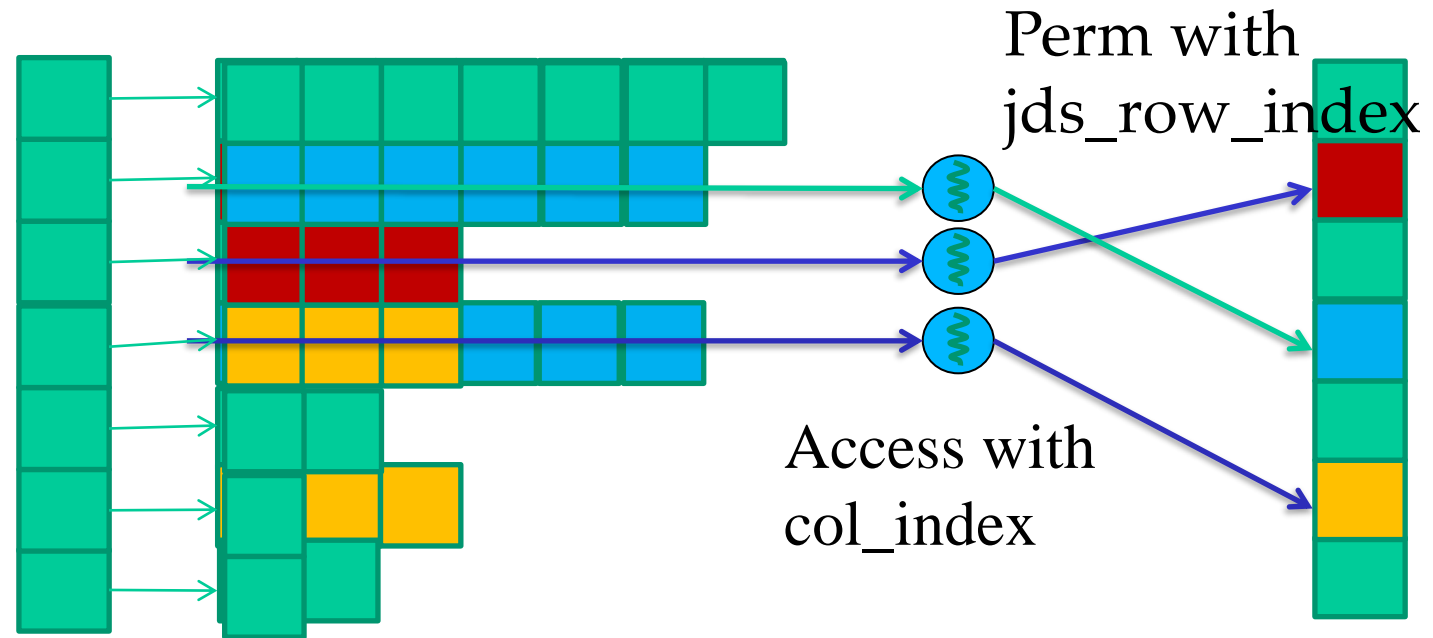
Often implemented  
in sequential host  
code in practice



# Reduced Padding with Hybrid Format

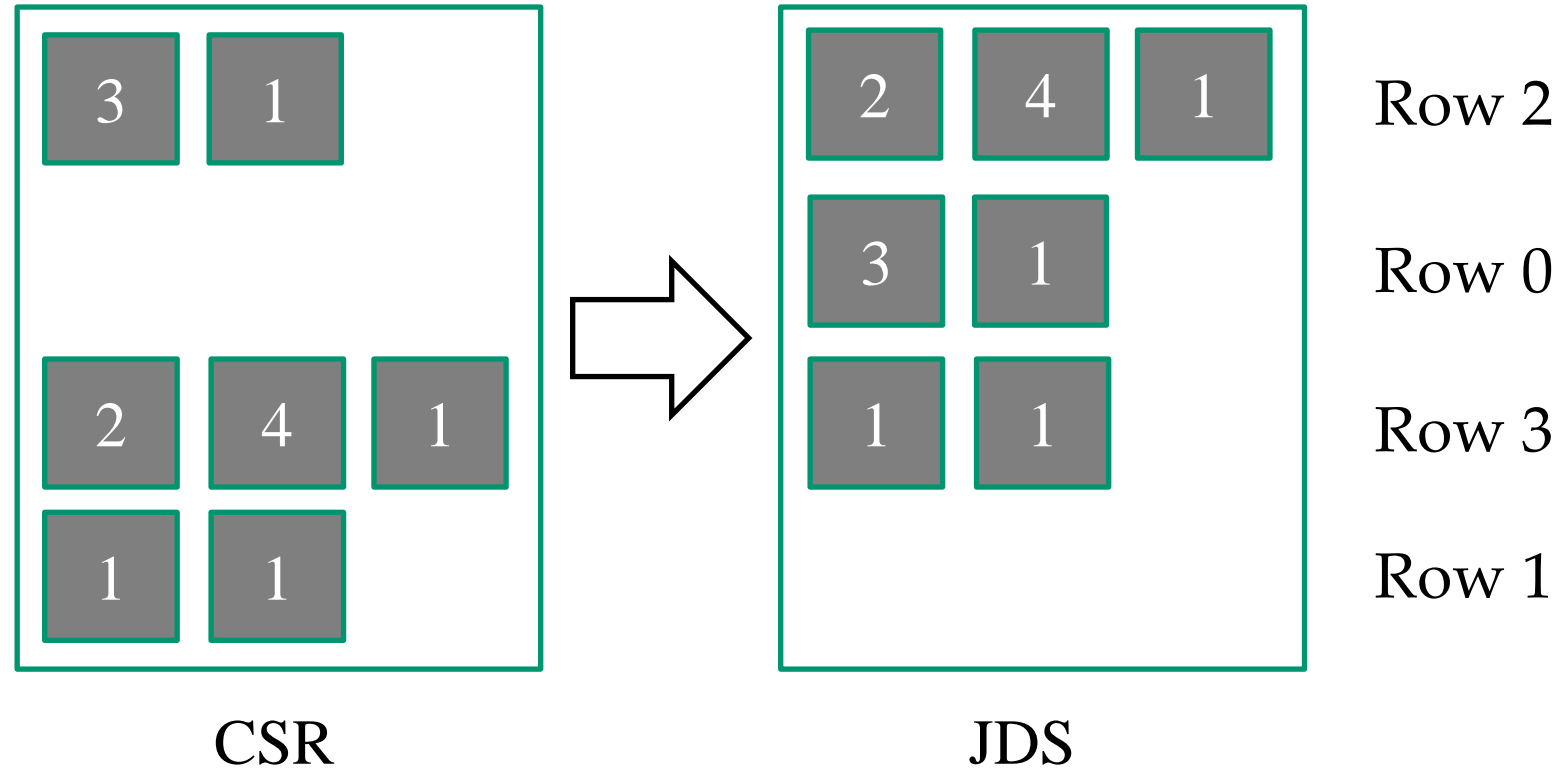


# JDS (Jagged Diagonal Sparse) Kernel Design for Load Balancing



Sort rows into descending order according to number of non-zero. Keep track of the original row numbers so that the output vector can be generated correctly.

## Sorting Rows According to Length (Regularization)



# CSR to JDS Conversion

	Row 0	Row 2	Row 3
Nonzero values data[7]	{ 3, 1, }	{ 2, 4, 1, }	{ 1, 1 }
Column indices col_index[7]	{ 0, 2, }	{ 1, 2, 3, }	{ 0, 3 }
Row Pointers row_ptr[5]	{ 0, 2, 2, }		{ 5, 7 }

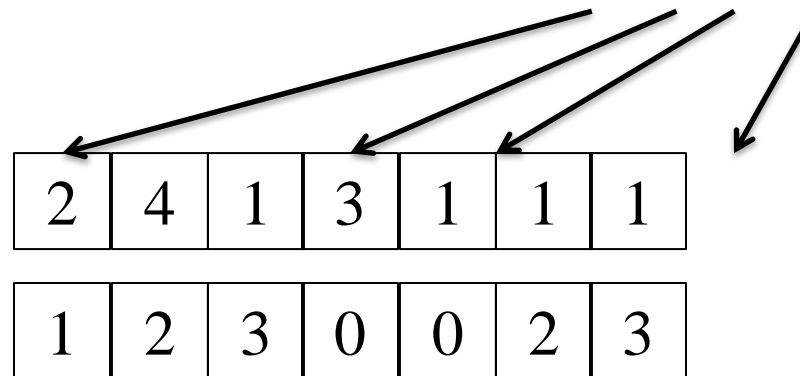
↙

	Row 2	Row 0	Row 3
Nonzero values data[7]	{ 2, 4, 1, }	{ 3, 1, }	{ 1, 1 }
Column indices col_index[7]	{ 1, 2, 3, }	{ 0, 2, }	{ 0, 3 }
JDS Row Pointers jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }		
JDS Row Indices jds_row_index	{ 2, 0, 3, 1 }		



## JDS Summary

Nonzero values   data[7]                    { 2, 4, 1, 3, 1, 1, 1 }  
Column indices   col\_index[7]            { 1, 2, 3, 0, 2, 0, 3 }  
JDS row indices   Jds\_row\_index[4]       { 2, 0, 3, 1 }  
JDS Row Ptrs   Jds\_row\_ptr[4]           { 0, 3, 5, 7, 7 }



# A Parallel SpMV/JDS Kernel

```

1. __global__ void SpMV_JDS(int num_rows, float *data,
    int *col_index, int *jds_row_ptr, int jds_row_index,
    float *x, float *y) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         int row_start = jds_row_ptr[row];
6.         int row_end = jds_row_ptr[row+1];
7.         for (int elem = row_start; elem < row_end; elem++) {
8.             dot += data[elem] * x[col_index[elem]];
9.         }
10.        y[jds_row_index[row]] = dot;
11.    }
12.}

```

	Row 2	Row 0	Row 3
Nonzero values data[7]	{ 2, 4, 1,	3, 1,	1 1 }

Column indices col_index[7]	{ 1, 2, 3,	0, 2,	0, 3 }
-----------------------------	------------	-------	--------

JDS Row Pointers jds_row_ptr[5]	{ 0,	3,	5,	7,7 }
---------------------------------	------	----	----	-------

JDS Row Indices jds_row_index	{ 2,	0,	3,	1 }
-------------------------------	------	----	----	-----

# JDS vs. CSR - Control Divergence

Threads still execute different number of iterations in the JDS kernel for-loop

- However, neighboring threads tend to execute similar number of iterations because of sorting.
- Better thread utilization

Nonzero values   data[7]                    { 2, 4, 1, 3, 1, 1, 1 }

Column indices   col\_index[7]            { 1, 2, 3, 0, 2, 0, 3 }

JDS row indices   Jds\_row\_index[4]    { 2, 0, 3, 1 }

JDS Row Ptrs   Jds\_row\_ptr[4]           { 0, 3, 5, 7, 7 }

data



2	4	1	3	1	1	1
---	---	---	---	---	---	---

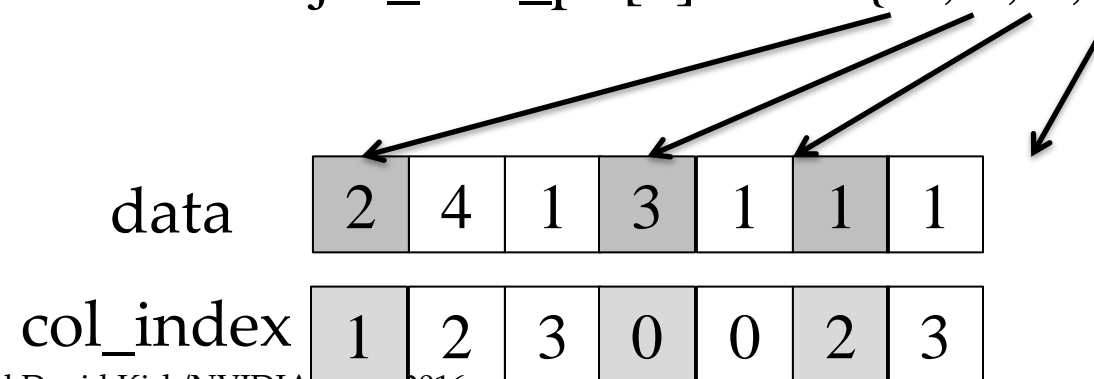
col\_index

1	2	3	0	0	2	3
---	---	---	---	---	---	---

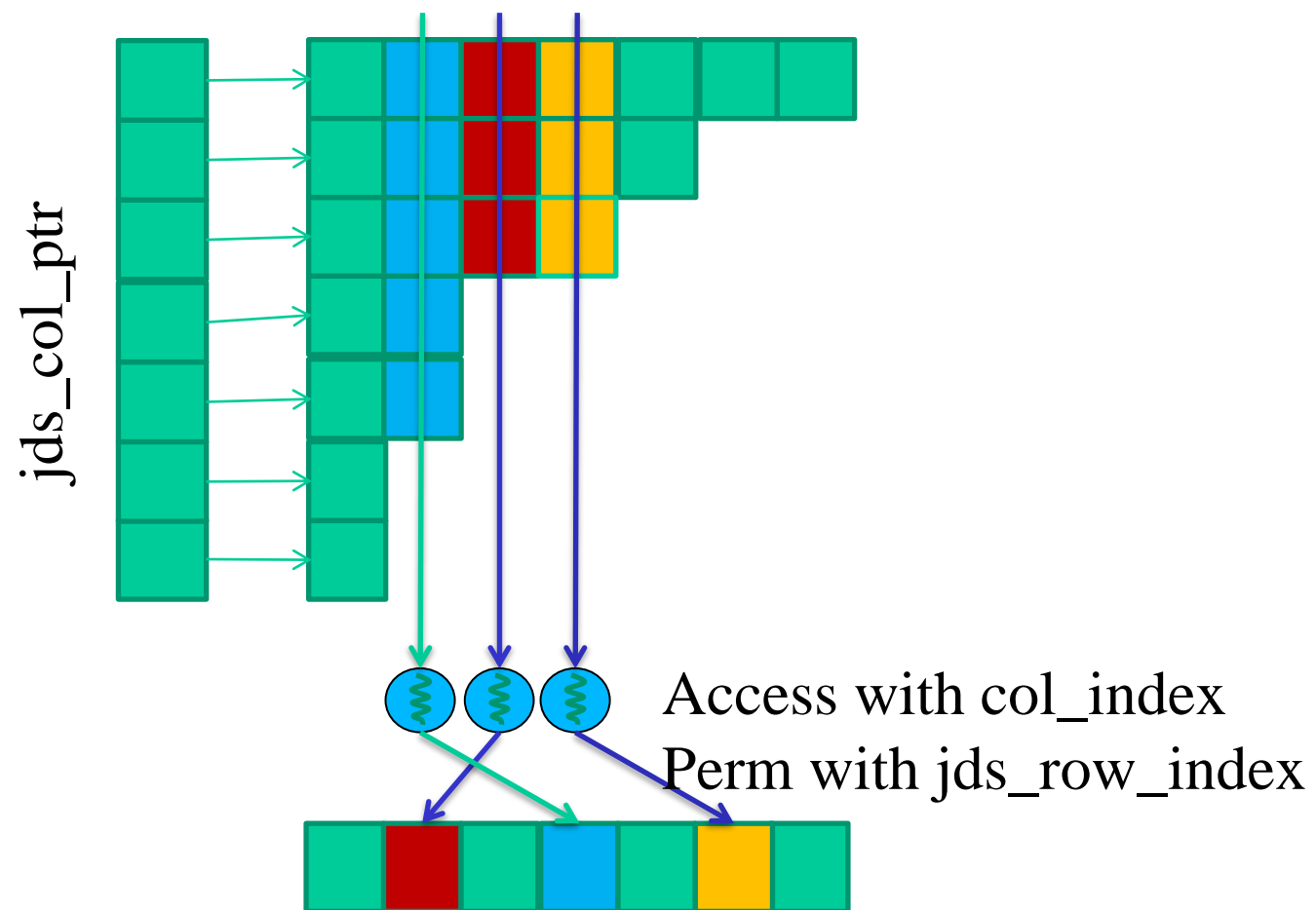
# JDS vs. CSR Memory Divergence

Adjacent threads still access non-adjacent memory locations

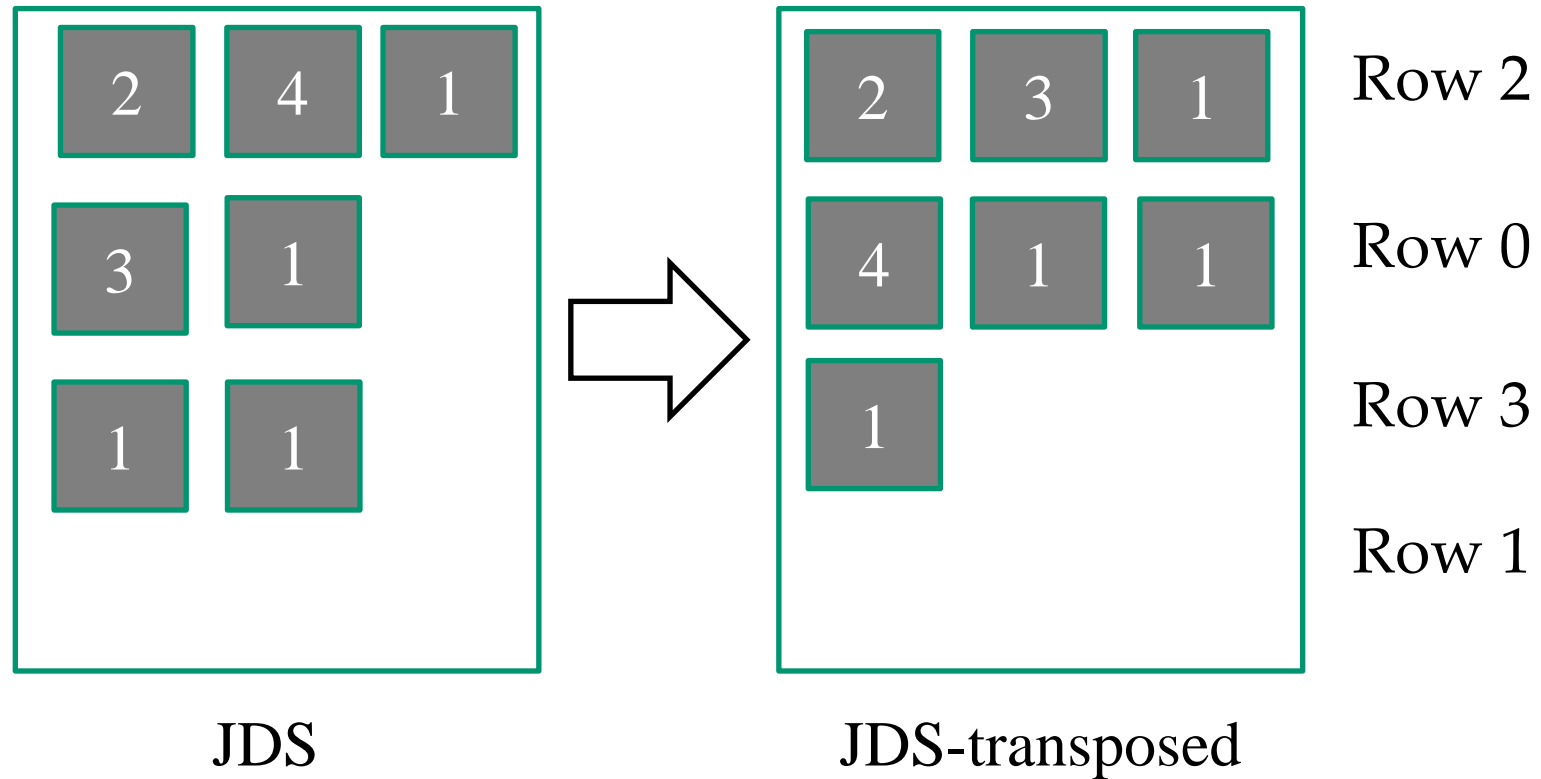
Nonzero values data[7] { 2, 4, 1, 3, 1, 1, 1 }  
Column indices col\_index[7] { 1, 2, 3, 0, 2, 0, 3 }  
JDS row indices jds\_row\_index[4] { 2, 0, 3, 1 }  
JDS Row Ptrs jds\_row\_ptr[4] { 0, 3, 5, 7, 7 }



# JDS with Trasposition



# Transposition for Memory Coalescing

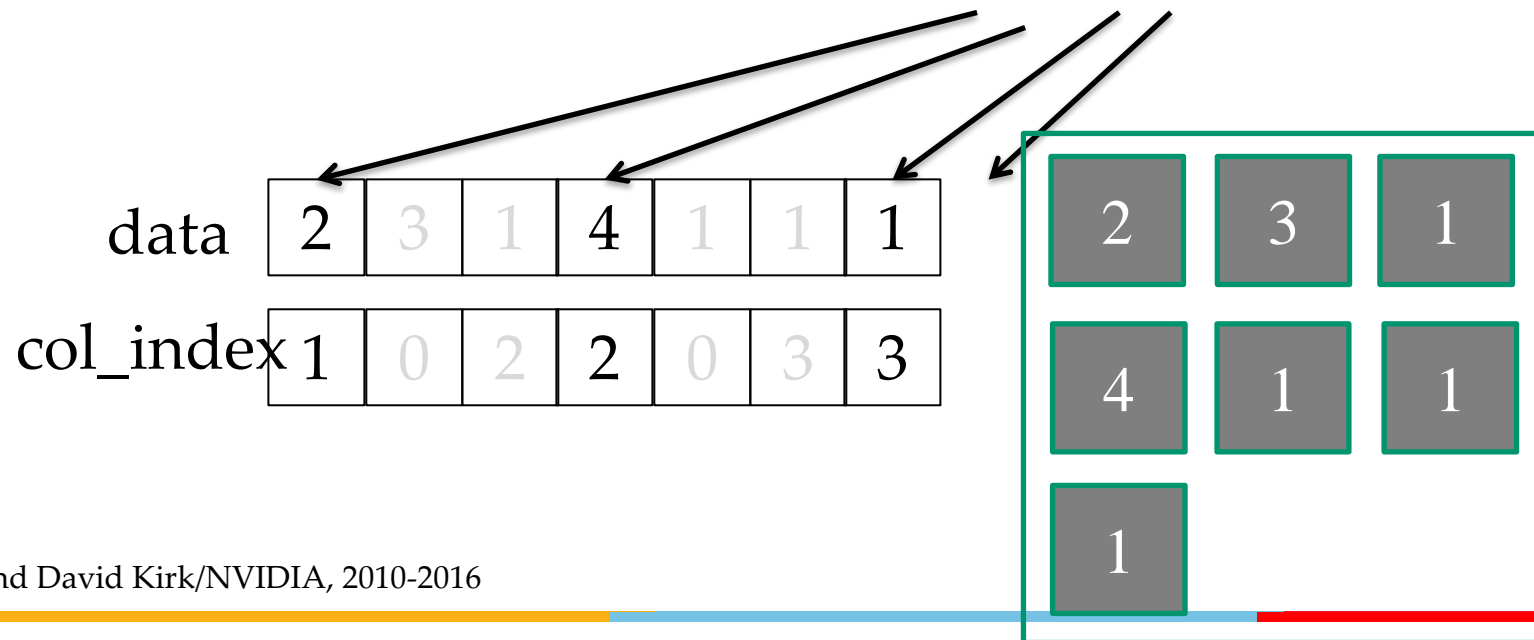


# JDS Format with Transposed Layout

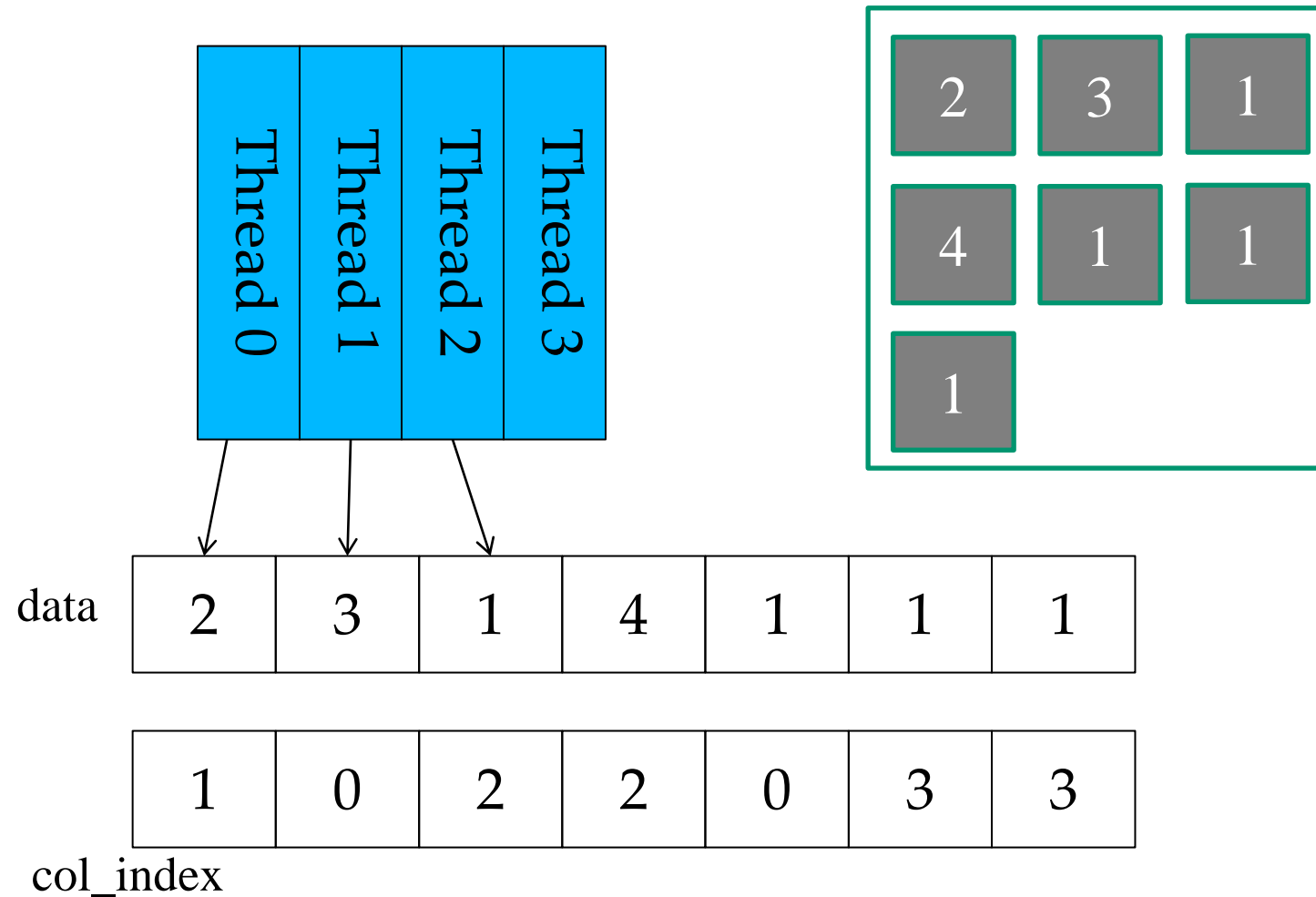
Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

JDS row indices `jds_row_index[4]` { 2, 0, 3, 1 }

JDS column pointers `jds_t_col_ptr[4]` { 0, 3, 6, 7 }

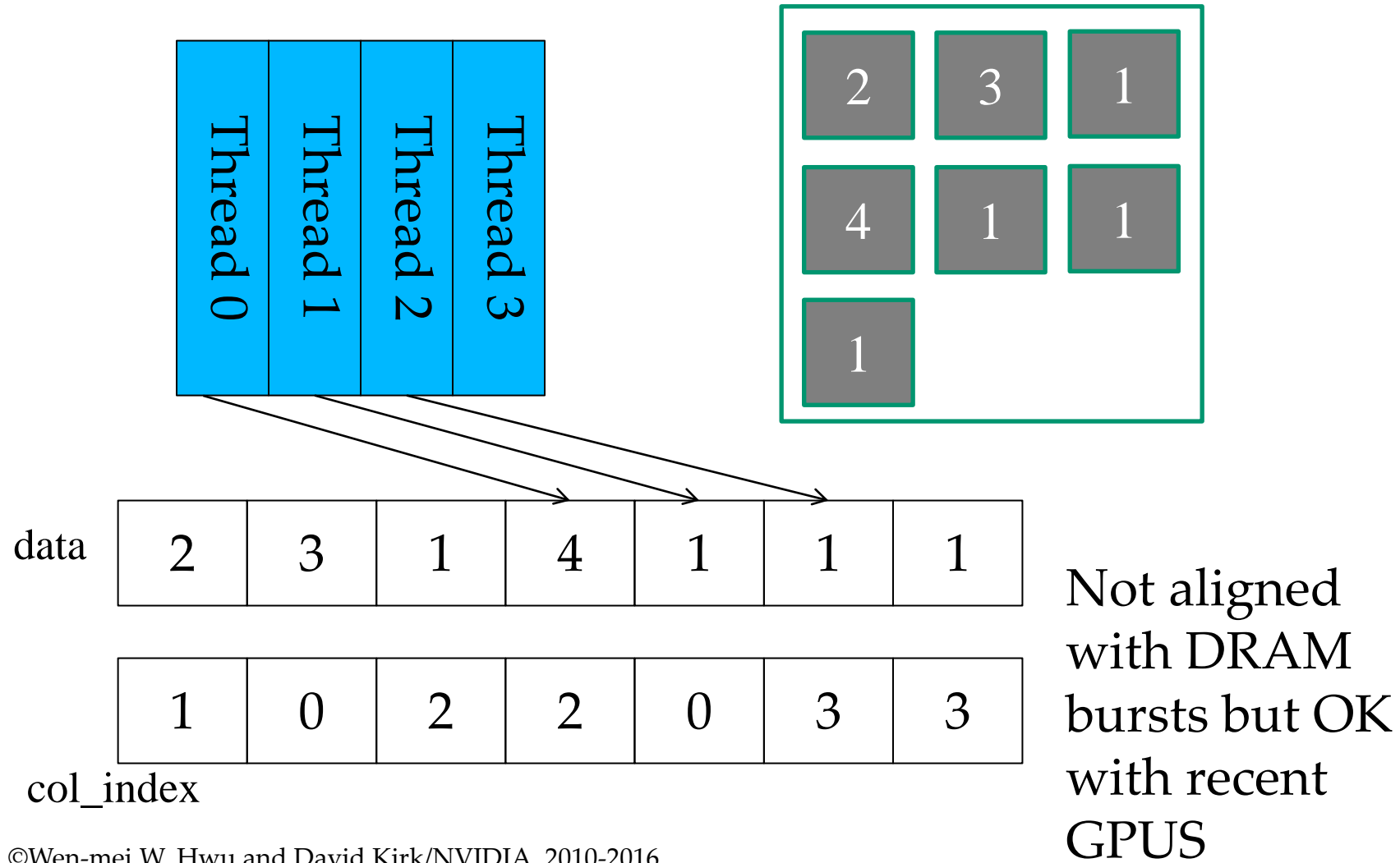


# JDS with Transposition Memory Coalescing

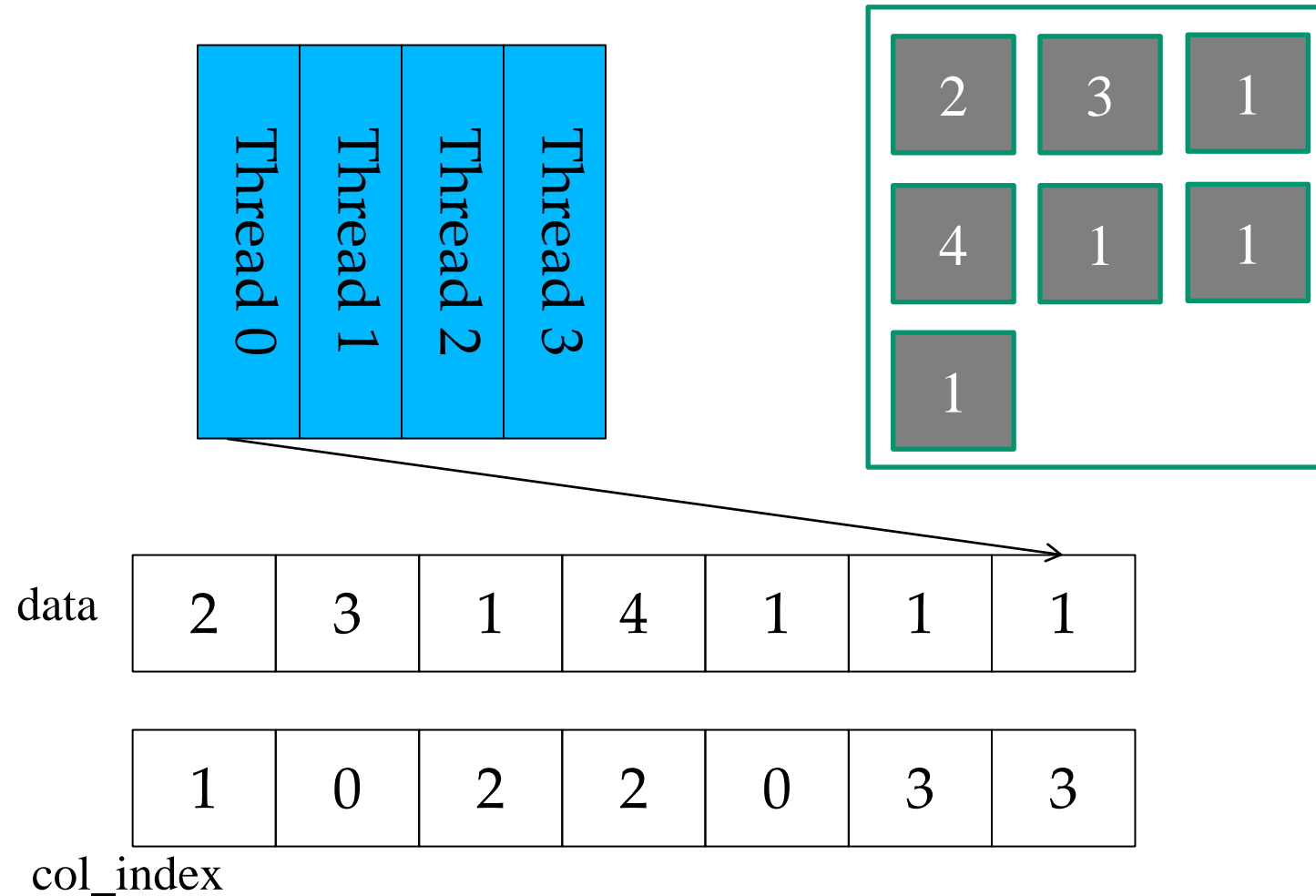




# JDS with Transposition Memory Coalescing



# JDS with Transposition Memory Coalescing



# A Parallel SpMV/JDS\_T Kernel

```

1. __global__ void SpMV_JDS_T(int num_rows, float *data,
    int *col_index, int *jds_t_col_ptr, int *jds_row_index,
    float *x, float *y) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         unsigned in sec = 0;
6.         while (jds_t_col_ptr[sec+1]-jds_t_col_ptr[sec] > row) {
7.             dot += data[jds_t_col_ptr[sec]+row] *
                x[col_index[jds_t_col_ptr[sec]+row]];
8.             sec++;
9.         }
10.        y[jds_row_index[row]] = dot;
11.    }
12.}

```

	Sec 0	Sec 1	Sec 2
Nonzero values data[7]	{ 2, 3, 1,	4, 1,	1 1 }
Column indices col_index[7]	{ 1, 0, 3,	2, 2,	3, 3 }

JDS\_T Column Pointers jds\_t\_col\_ptr[5] { 0, 3, 5, 7, 7 }

JDS Row Indices jds\_row\_index[4] { 2, 0, 3, 1 }

## Objective

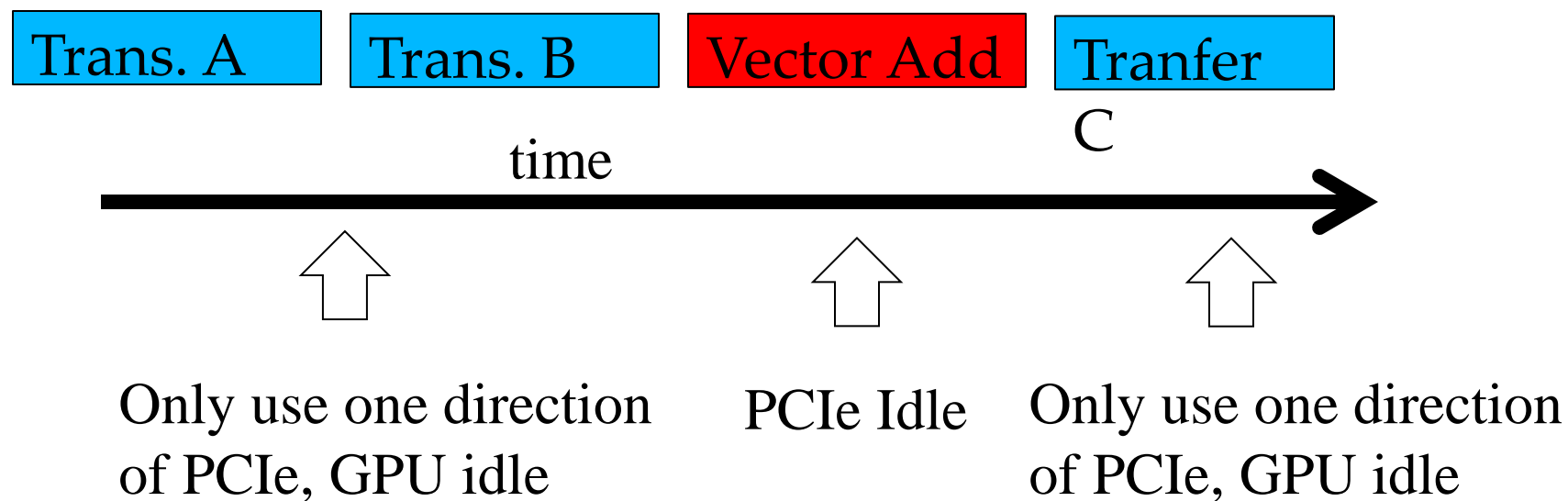
---

To learn more advanced features of the CUDA APIs for data transfer and kernel launch

- Task parallelism for overlapping data transfer with kernel computation
- CUDA streams

# Serialized Data Transfer and GPU computation

So far, the way we use cudaMemcpy serializes data transfer and GPU computation



# Device Overlap

---

Some CUDA devices support *device overlap*

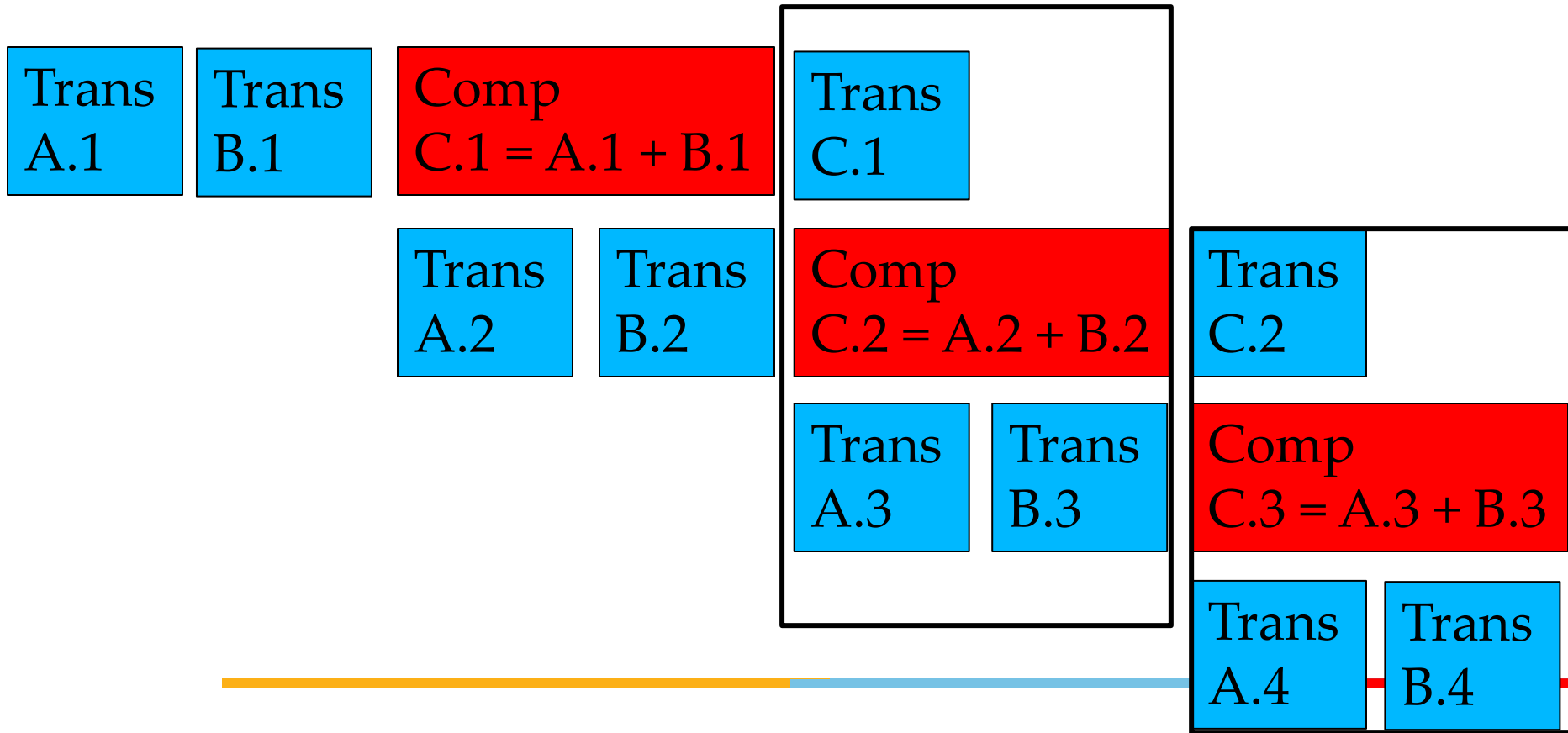
- *Simultaneously execute a kernel while performing a copy between device and host memory*

```
int dev_count;  
cudaDeviceProp prop;  
  
cudaGetDeviceCount( &dev_count);  
for (int i = 0; i < dev_count; i++) {  
    cudaGetDeviceProperties(&prop, i);  
  
    if (prop.deviceOverlap) ...
```

# Overlapped (Pipelined) Timing

Divide large vectors into segments

Overlap transfer and compute of adjacent segments



# Using CUDA Streams and Asynchronous Memcpy

---

CUDA supports parallel execution of kernels and cudaMemcpy with “Streams”

Each stream is a queue of operations (kernel launches and cudaMemcpy's)

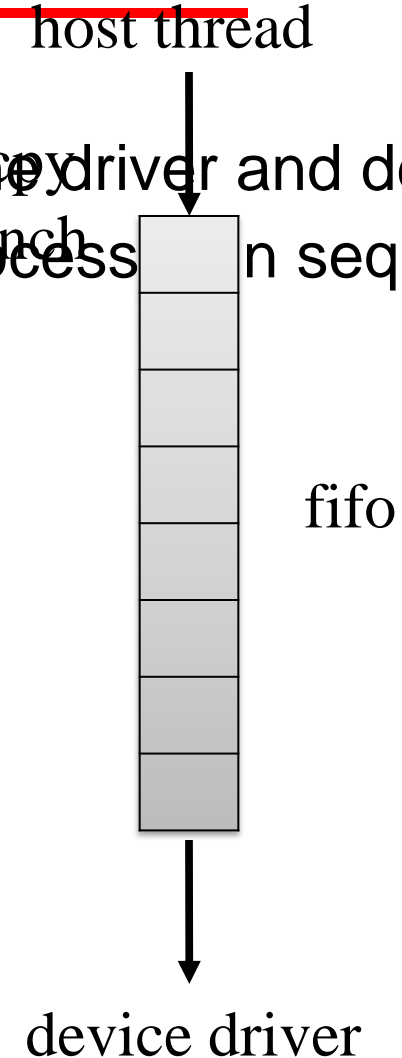
Operations (tasks) in different streams can go in parallel

- “Task parallelism”



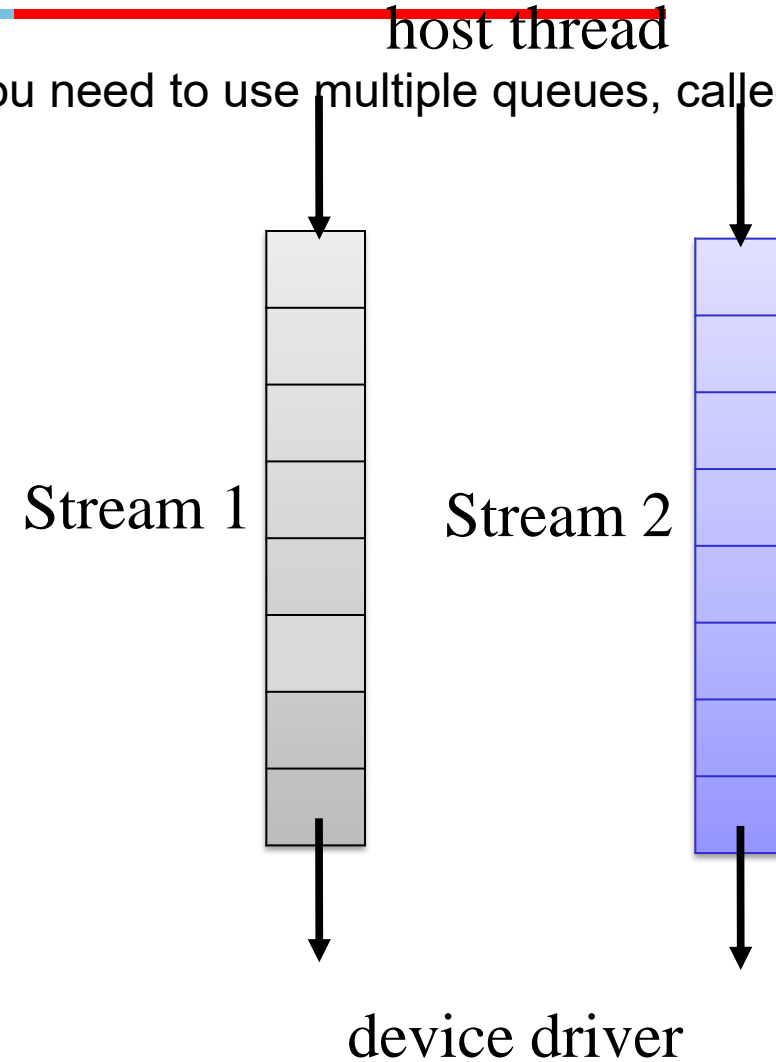
# Streams

- Device requests made from the host code are put into a queue
  - Queue is read and processed asynchronously by the driver and device
  - Driver ensures that commands in the queue are processed in sequence. Memory copies end before kernel launch, etc.

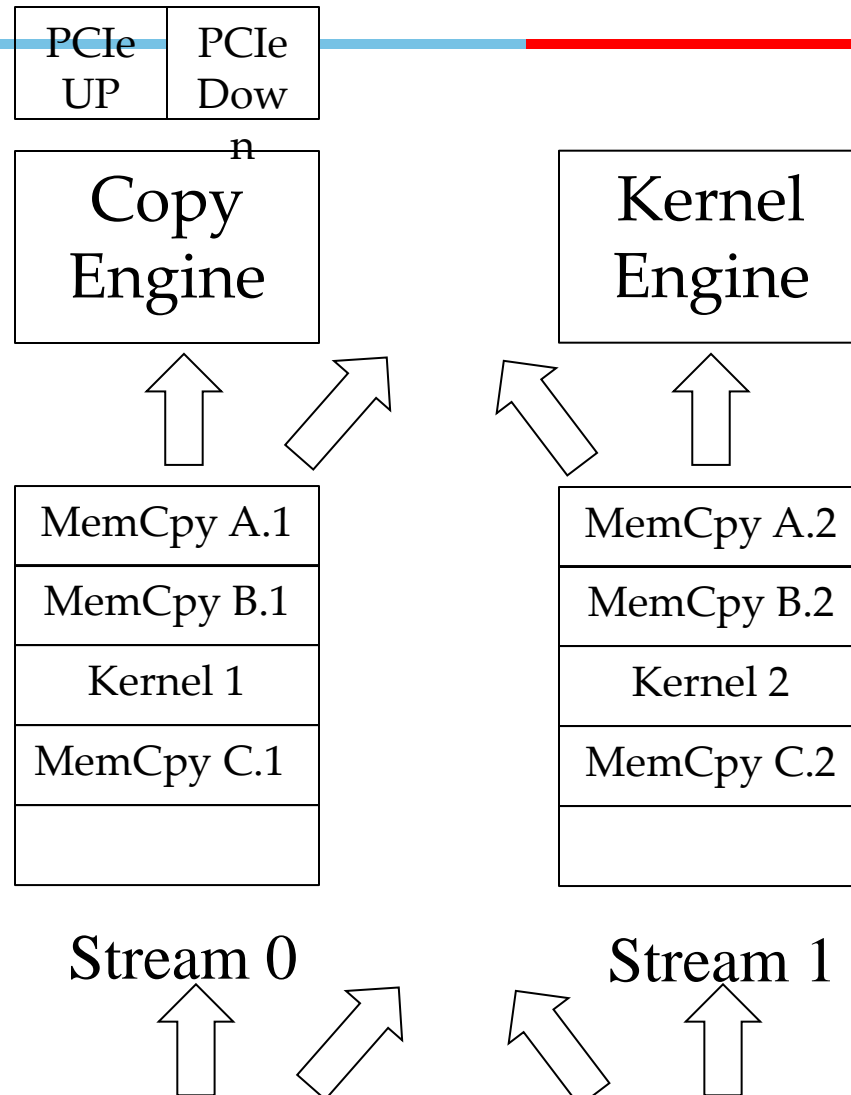


## Streams cont.

- To allow concurrent copying and kernel execution, you need to use multiple queues, called “streams”



# Conceptual View of Streams



Operations (Kernels, MemCpys)

## A Simple Multi-Stream Host Code

---

```
cudaStream_t stream0, stream1;
cudaStreamCreate( &stream0);
cudaStreamCreate( &stream1);
float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

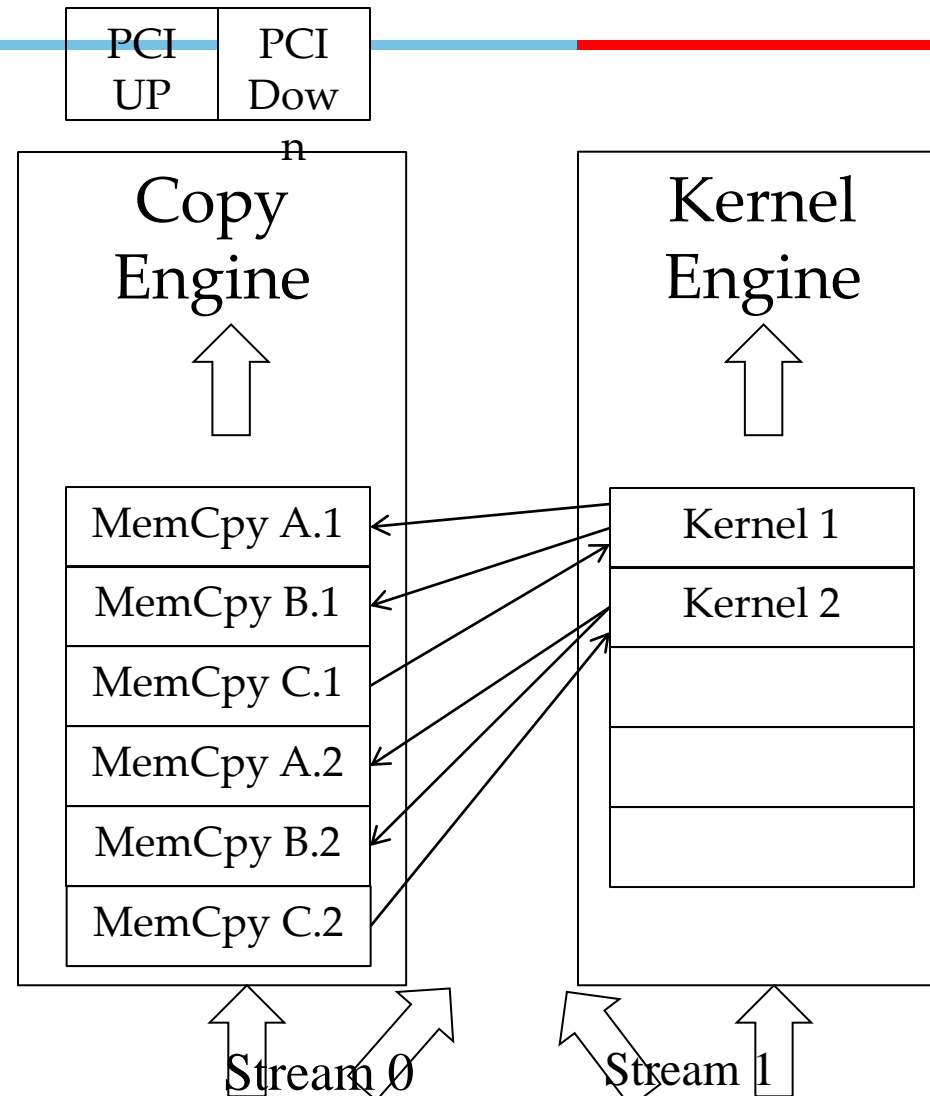
// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0);
    cudaMemcpyAsync(d_C0, h_C+I; SegSize*sizeof(float),..., stream0);
```

## A Simple Multi-Stream Host Code (Cont.)

```
for (int i=0; i<n; i+=SegSize*2) {  
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);  
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);  
    cudaMemcpyAsync(d_C0, h_C+i; SegSize*sizeof(float),..., stream0);  
  
    cudaMemcpyAsync(d_A1, h_A+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
    cudaMemcpyAsync(d_B1, h_B+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);  
    cudaMemcpyAsync(d_C1, h_C+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
}
```

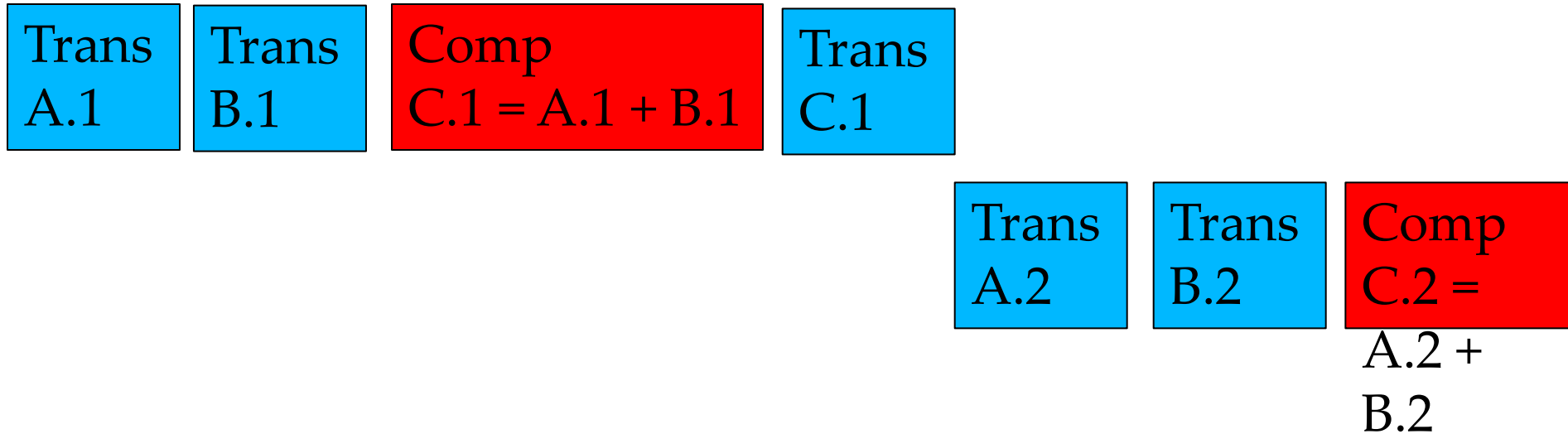
# A View Closer to Reality in GPUs without hardware stream queues



Operations (Kernels, MemCpys)

## Not quite the overlap we want

C.1 blocks A.2 and B.2 in the copy engine queue



## A Better Multi-Stream Host Code (Cont.)

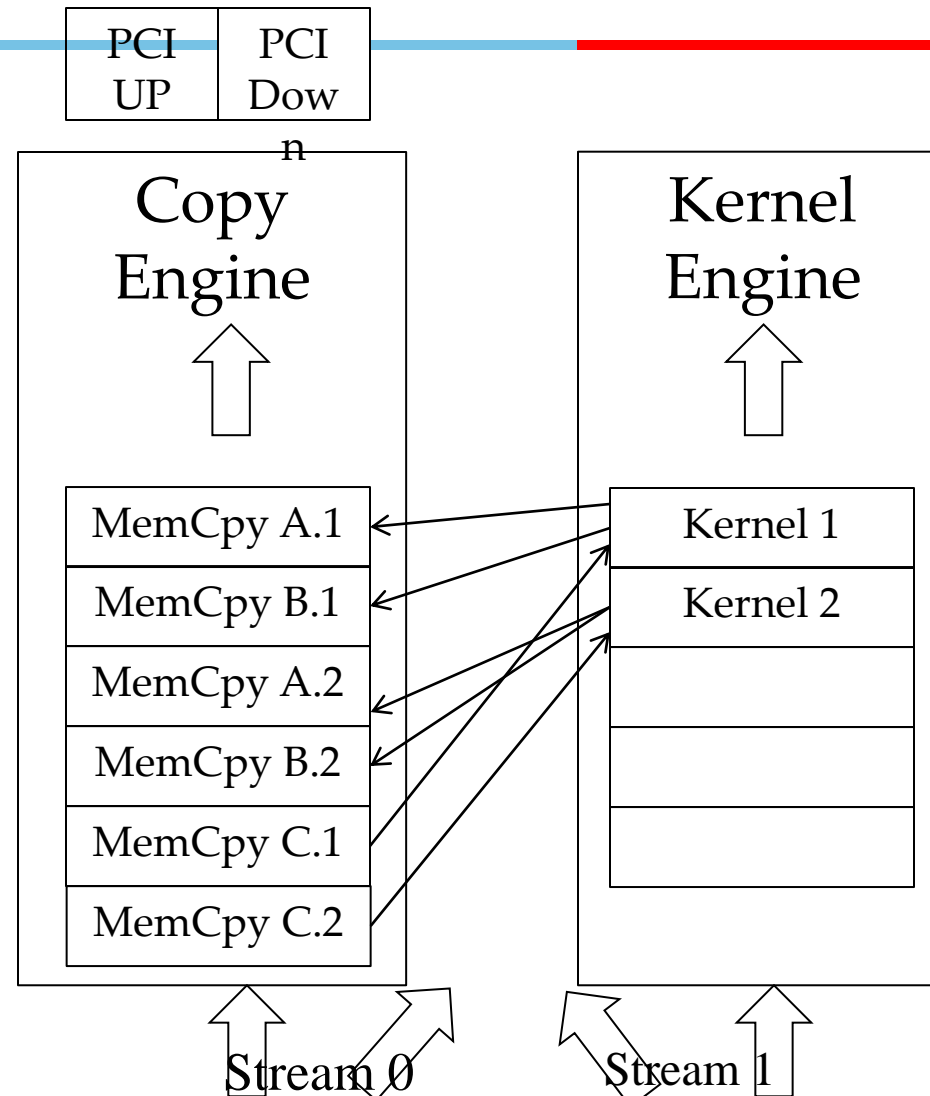
---

```
for (int i=0; i<n; i+=SegSize*2) {  
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_A1, h_A+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
    cudaMemcpyAsync(d_B1, h_B+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
  
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);  
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);  
    cudaMemcpyAsync(d_C0, h_C+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_C1, h_C+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
}
```

---



## A View Closer to Reality

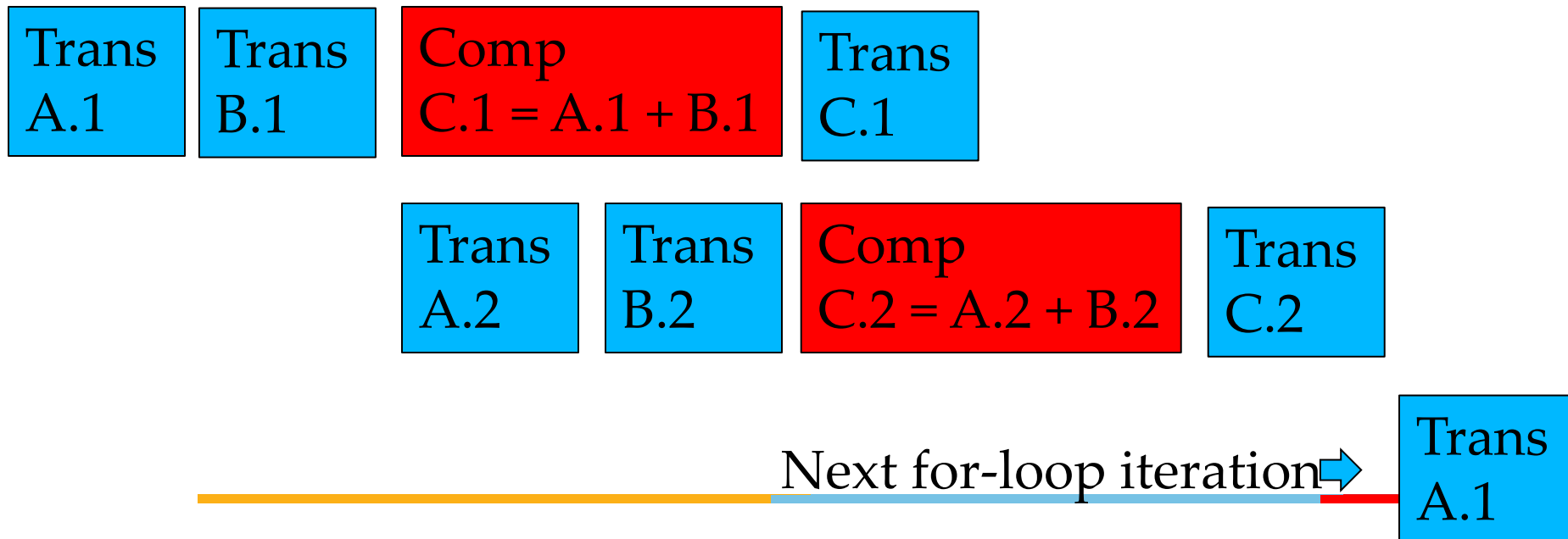


Operations (Kernels, MemCpys)

# Better Overlap with Two Streams

C.1 no longer blocks A.2 and B.2 in the copy engine queue

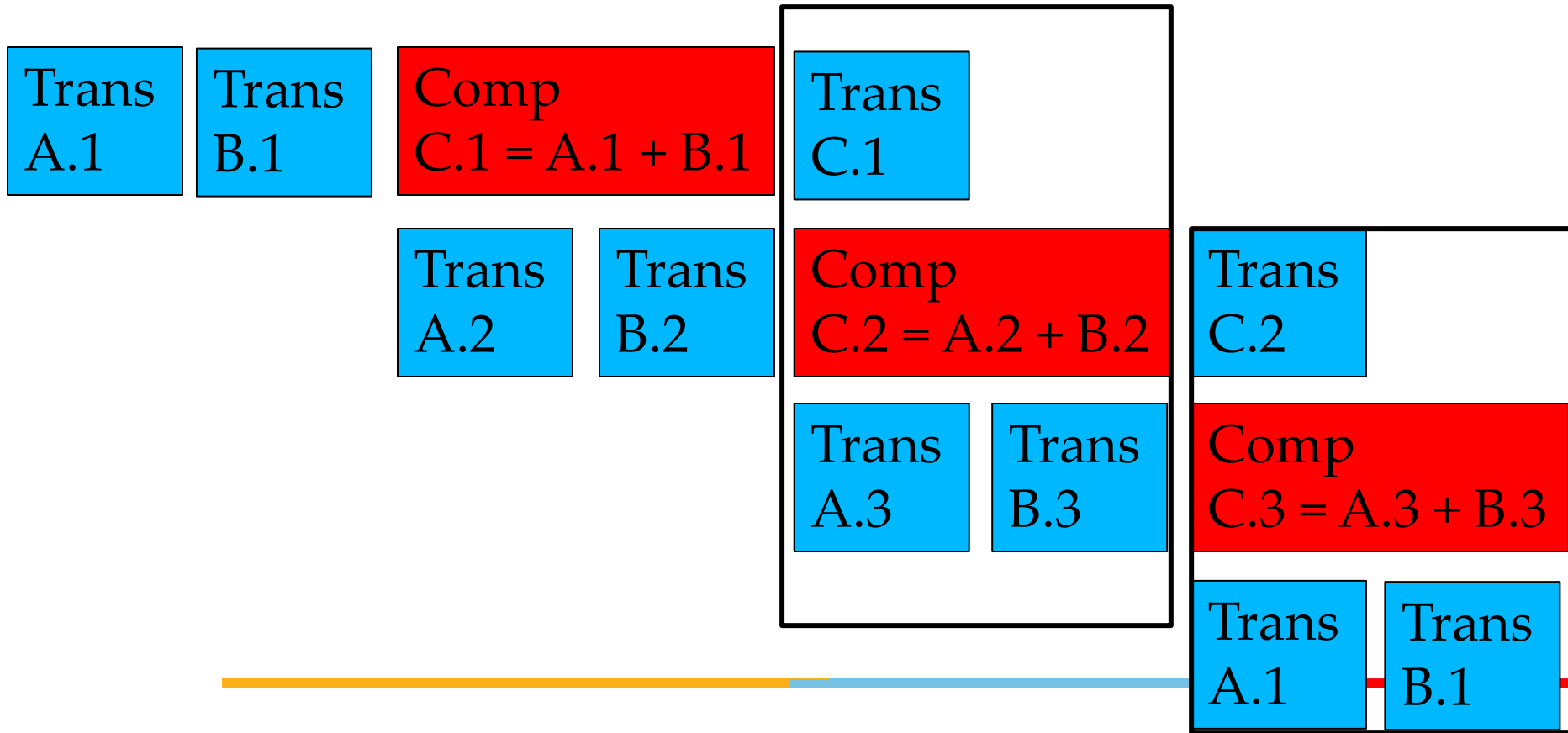
However, C.2 still blocks A.1 and A.2 from the next iteration – PCIe used for only one direction



# Three streams needed for continuously pipelined) timing

Divide large vectors into segments

Overlap transfer and compute of adjacent segments



# Hyper Queue

---

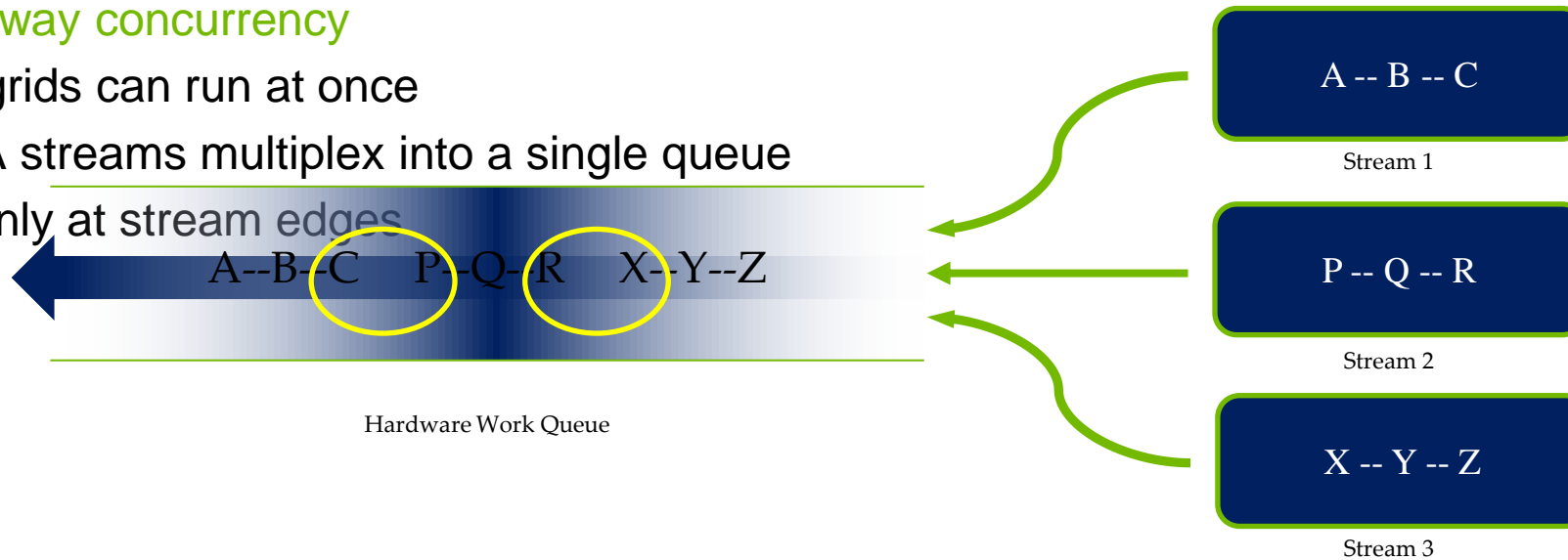
Provide multiple real stream queues for each engine

Allow much more concurrency by allowing some streams to make progress for an engine while others are blocked

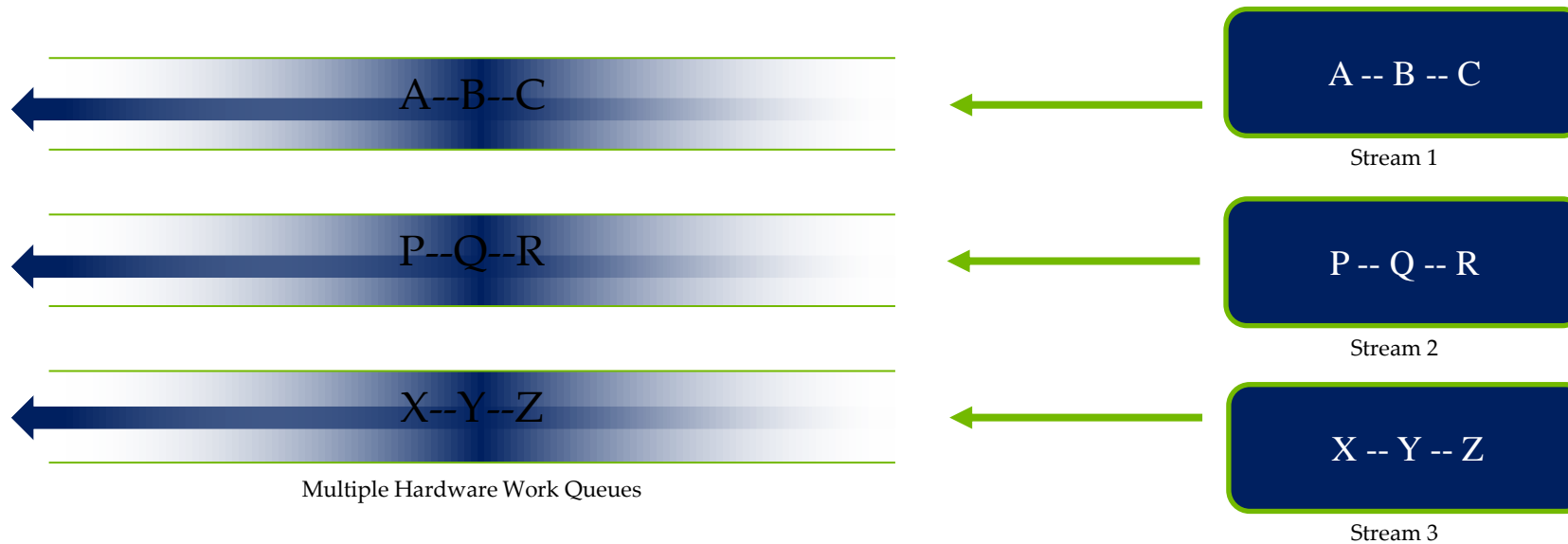
# Fermi (and older) Concurrency

## Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges



# Kepler Improved Concurrency

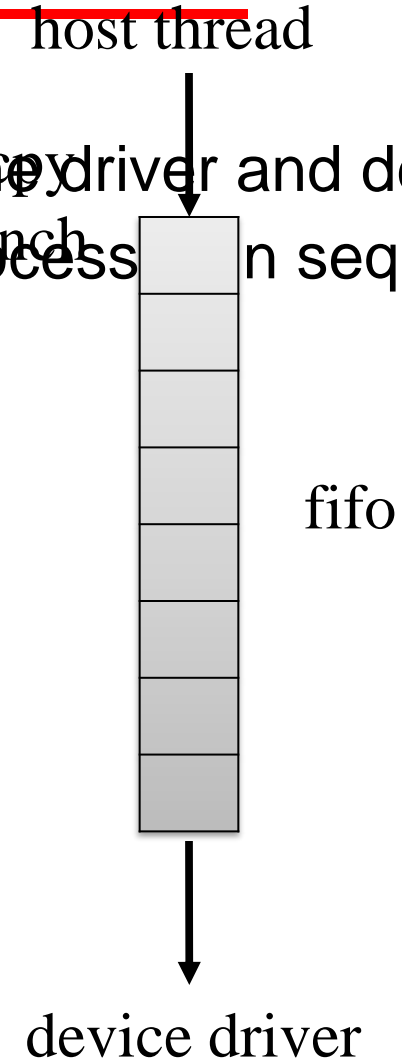


## Kepler allows 32-way concurrency

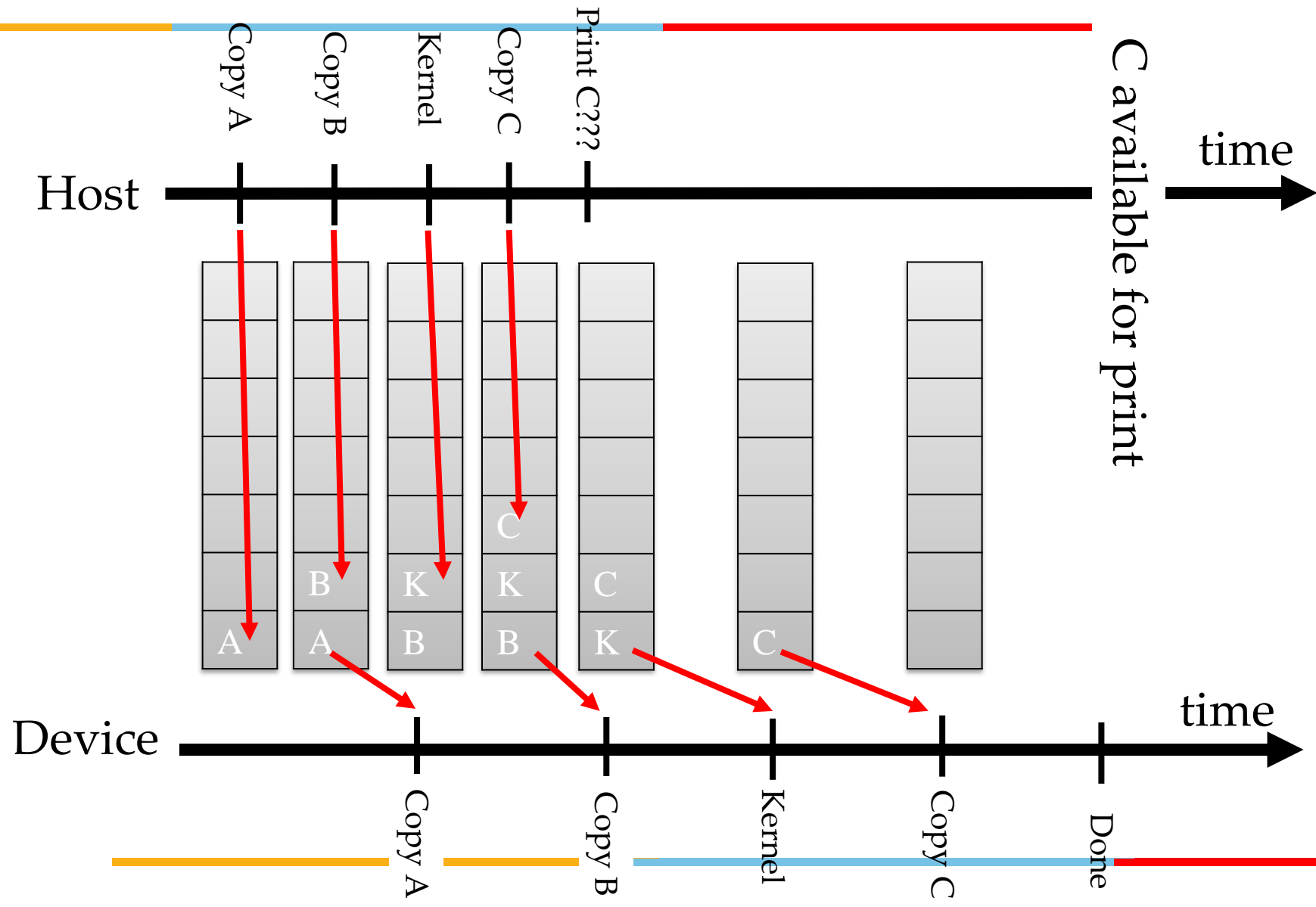
- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

# Streams (review)

- Device requests made from the host code are put into a queue
  - Queue is read and processed asynchronously by the driver and device
  - Driver ensures that commands in the queue are processed in sequence. Memory copies end before kernel launch, etc.



# Asynchronous Execution

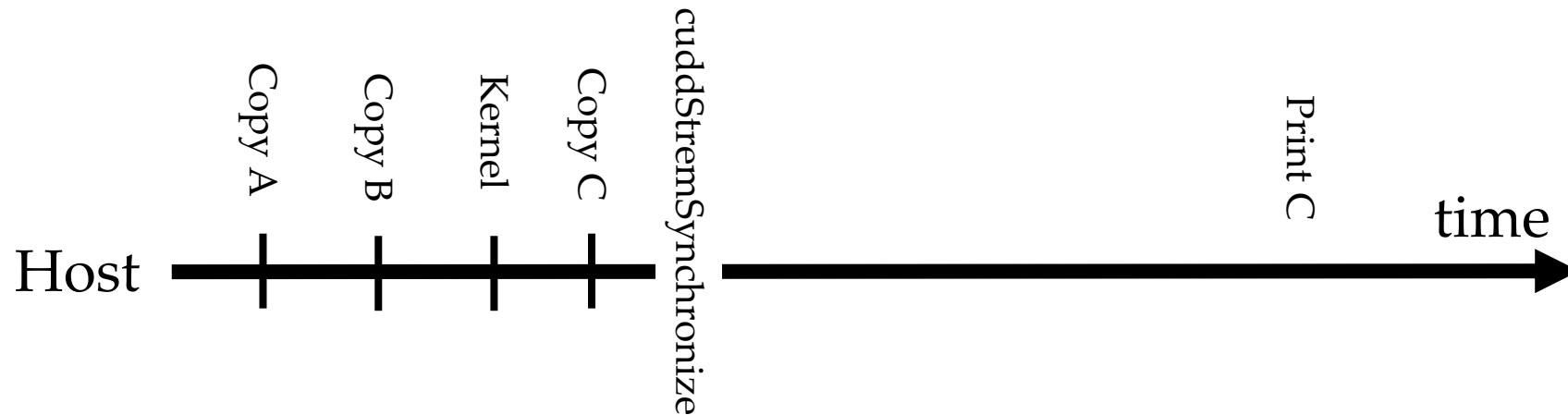




# Host-Device Synchronization

`cudaStreamSynchronize( stream )`

- The host blocks its execution until the queue of the specified stream is empty and all stream tasks complete



## Accurate Timing with Events

---

```
cudaEvent_t    start, stop;
cudaEventCreate( &start);
cudaEventCreate( &stop);
cudaEventRecord( start, stream);
// GPU tasks for stream
cudaEventRecord(stop, stream);
cudaEventSynchronize( stop);
cudaEventElapsedTime( &elapsedTime, start, stop);
```

# Machine Learning

---

An important way of building applications whose logic is not fully understood.

- No catastrophic consequence for incorrect output
- Use labeled data – data that come with the input values and their desired output values – to learn what the logic should be
- Capture each labeled data item by adjusting the program logic
- Learn by example!

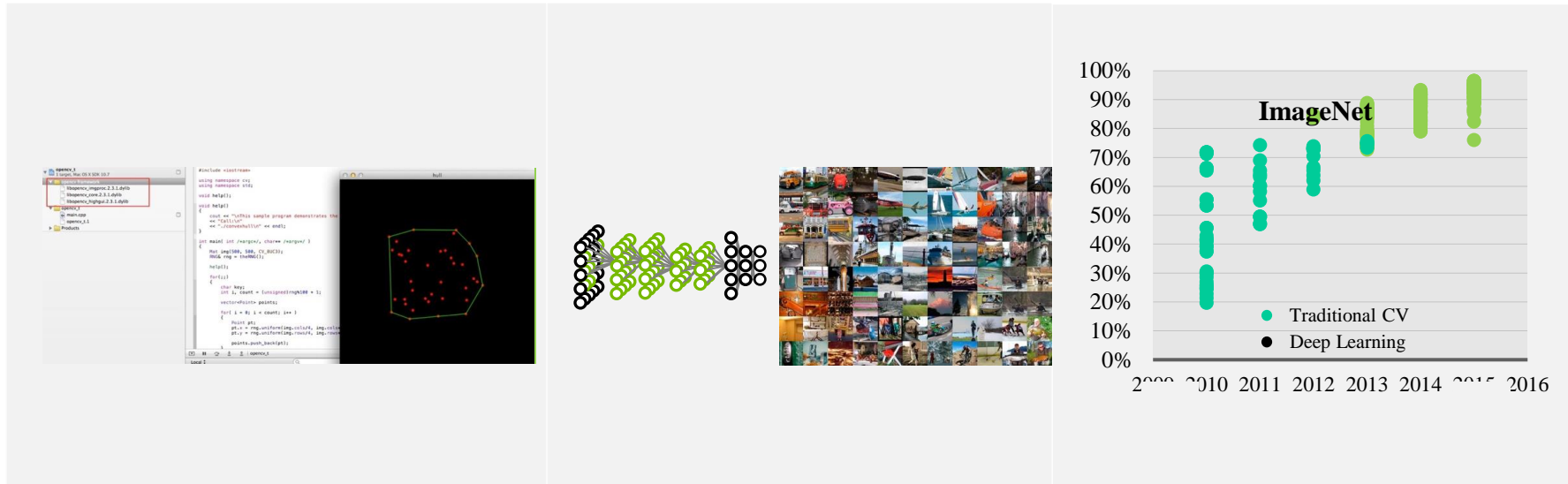
## Training Phase

- The system learns the logic for the application from labeled data.

## Deployment (inference) Phase

- The system applies the learned program logic in processing data

# Deep Learning in Computer Vision



Slide courtesy of Steve Oberlin, NVIDIA

# Recent Explosion of Deep Learning Applications

---

GPU computing hardware and programming interfaces such as CUDA has enabled very fast research cycle of deep neural net training

Computer Vision, Speech Recognition, Document Translation, Self Driving Cars, ...

Most involve logic that were previously not effectively constructed with imperial programming

Using big labeled data to train and specialize DNN based classifiers

- Deriving a large quantity of quality labeled data is a challenge

## Behind the Scenes

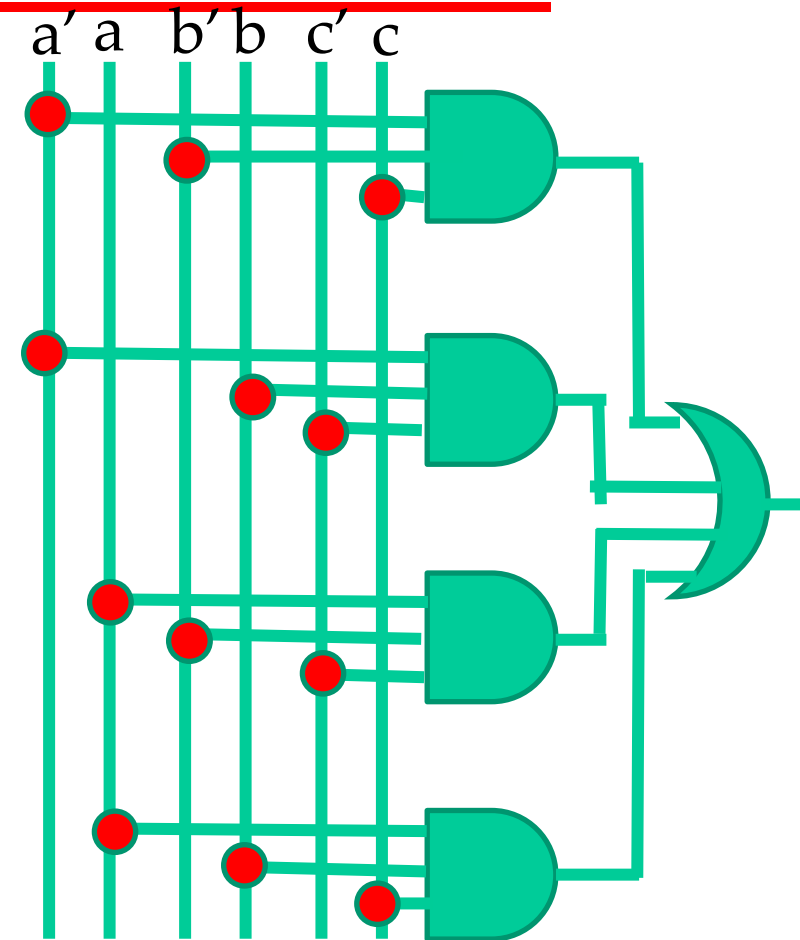
---

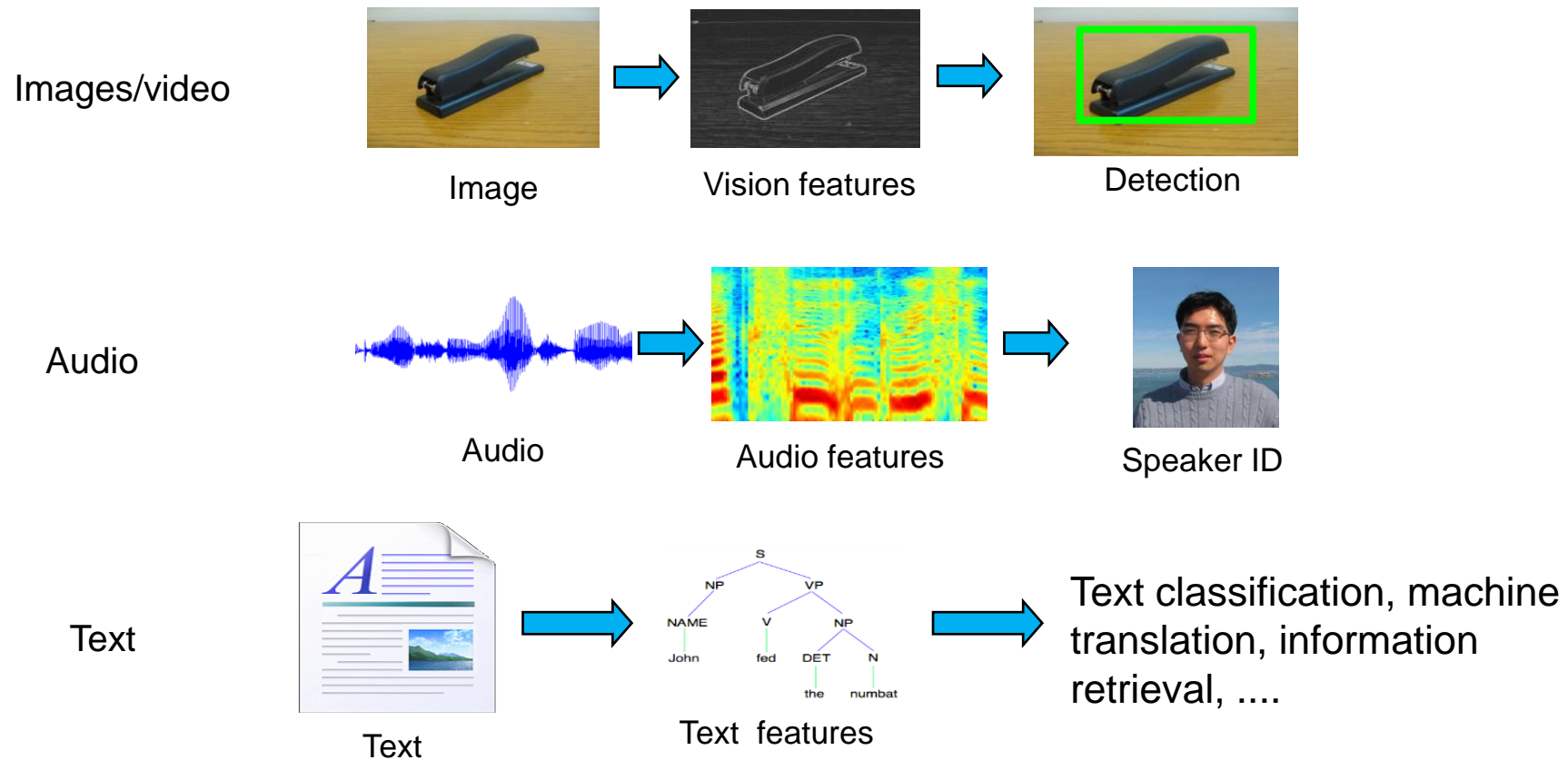
In 2010 Prof. Andreas Moshovos at University of Toronto adopted the ECE498AL Programming Massively Parallel Programming Class

Several of Prof. Geoffrey Hinton's graduate students took the course  
These students developed the GPU implementation of the DNN that was trained with 1.2M images to win the ImageNet competition

# Background: Combinations Logic Specification – Truth Table

Input			output
a	b	c	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1





Slide courtesy of Andrew Ng, Stanford University



## What if we did not know the truth table?

---

Look at enough observation data to construct the rule

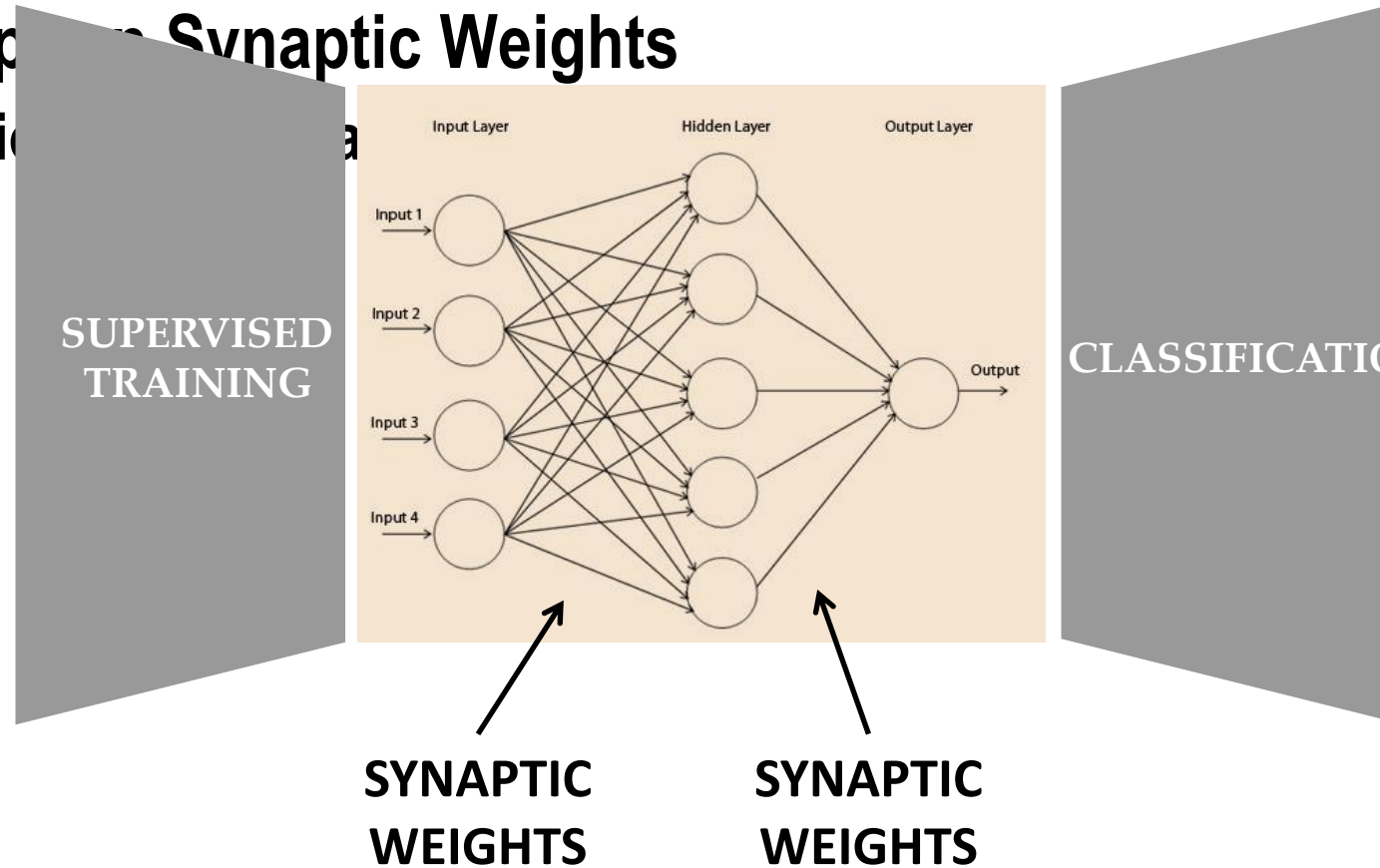
- $000 \rightarrow 0$
- $011 \rightarrow 0$
- $100 \rightarrow 1$
- $110 \rightarrow 0$

If we have enough observational data to cover all input patterns, we can construct the truth table and derive the logic!

Alien story.

# Multilayer Perceptron Synaptic Weights

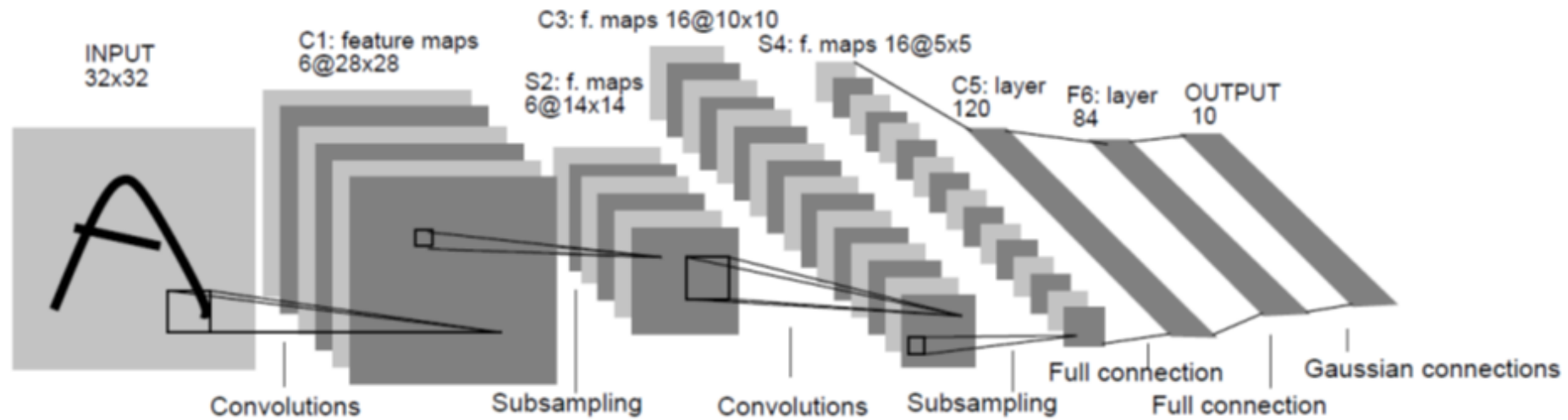
Universal for function approximation



[Cybenko, 1989; etc.]

## Combinational Logic of AI

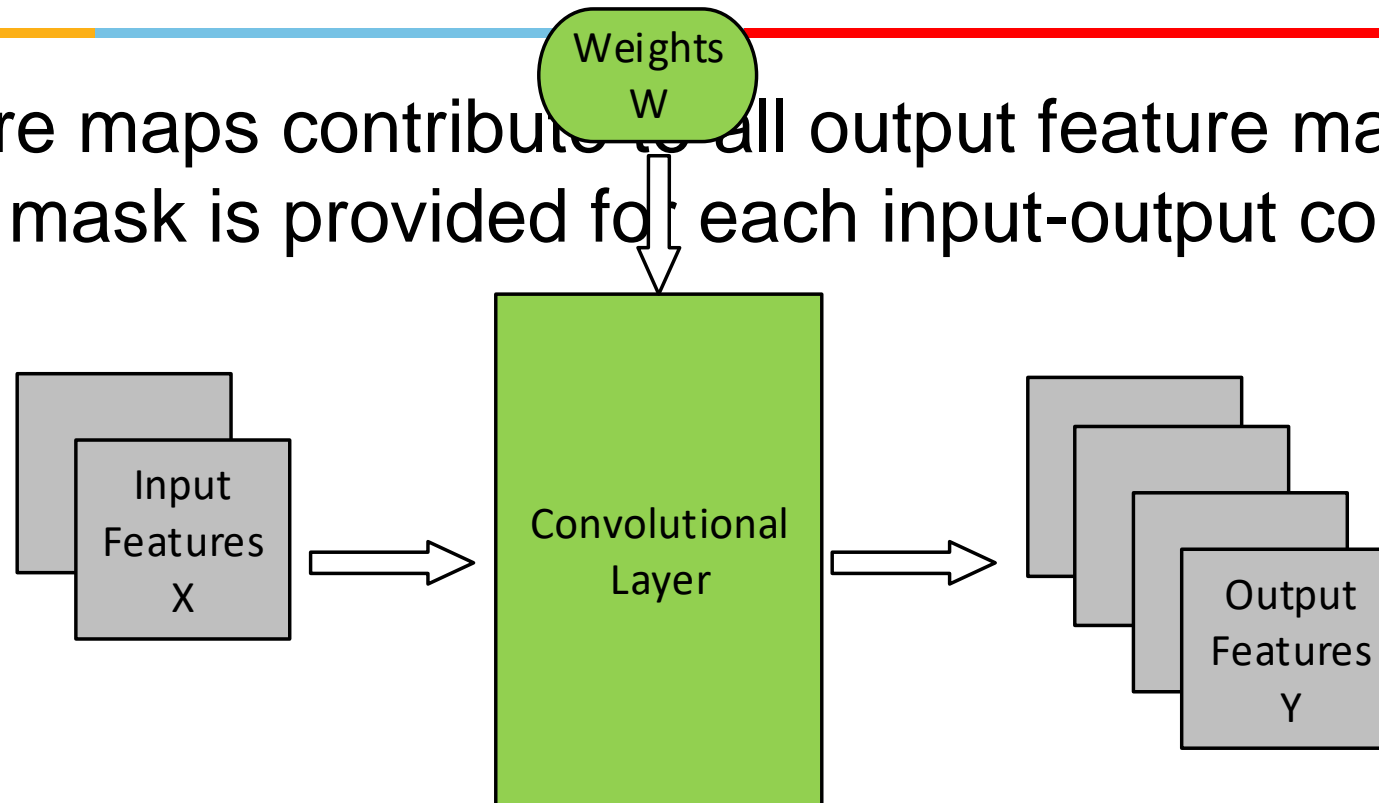
# LeNet-5, a convolutional neural network for hand-written digit recognition



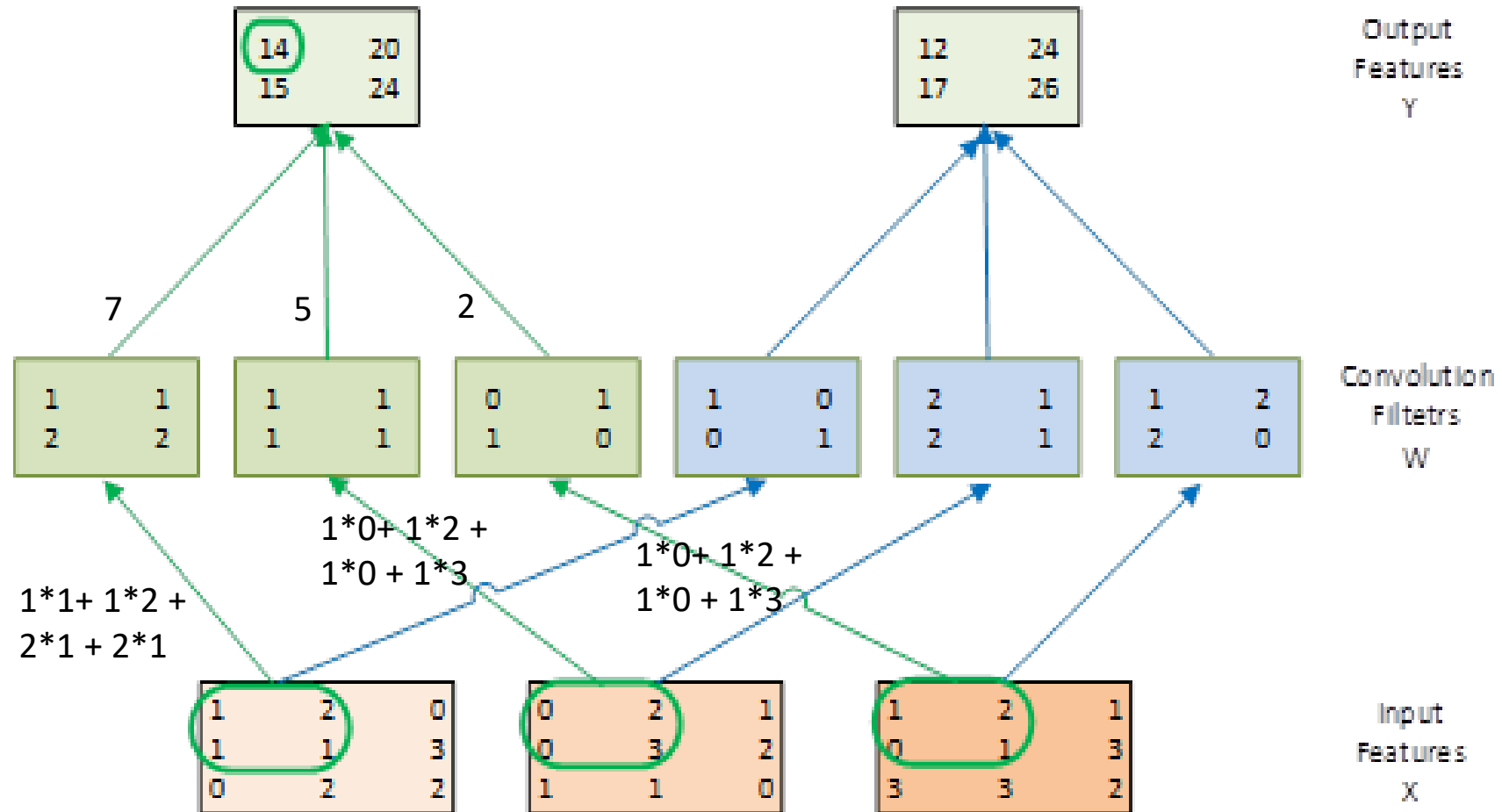
This is a 1024\*8 bit input, which will have a truth table of  $2^{8196}$  entries

# Forward Propagation Path of a Convolution Layer

All input feature maps contribute to all output feature maps. One convolution mask is provided for each input-output combination.

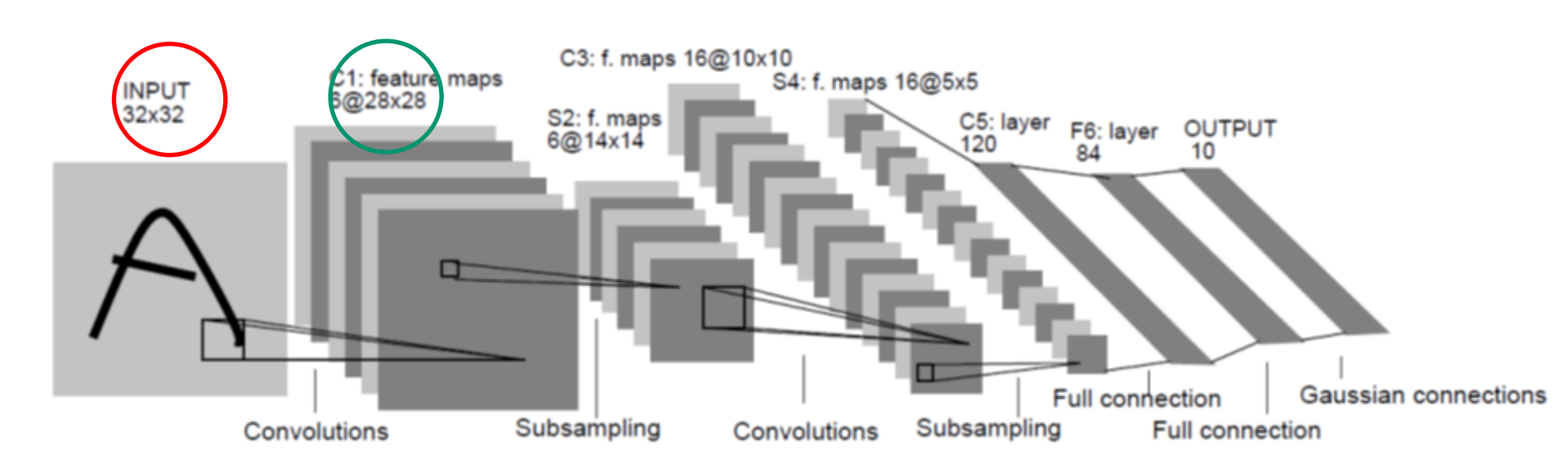


# Example of the Forward Path of a Convolution Layer



LeNet-5, a convolutional neural network for hand-written digit recognition.

LeNet-5, a convolutional neural network for hand-written digit recognition



# Sequential Code for the Forward Path of a Convolution Layer

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    int m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;

    for(int m = 0; m < M; m++)                // for each output feature map
        for(int h = 0; h < H_out; h++)        // for each output element
            for(int w = 0; w < W_out; w++) {
                Y[m, h, w] = 0;
                for(int c = 0; c < C; c++)      // sum over all input feature maps
                    for(int p = 0; p < K; p++)  // KxK filter
                        for(int q = 0; q < K; q++)
                            Y[m, h, w] += X[c, h + p, w + q] * W[m, c, p, q];
            }
}
```

# Sequential code for the Forward Path of a Sub-sampling Layer

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
    for(int m = 0; m < M; m++)                // for each output feature maps
        for(int h = 0; h < H/K; h++)           // for each output element
            for(int w = 0; w < W/K; w++) {
                S[m, x, y] = 0.;
                for(int p = 0; p < K; p++) {     // loop over KxK input samples
                    for(int q = 0; q < K; q++)
                        S[m, h, w] += Y[m, K*h + p, K*w + q] / (K*K);
                }
                // add bias and apply non-linear activation
                S[m, h, w] = sigmoid(S[m, h, w] + b[m])
            }
}
```



# Objective

---

To learn more about the implementation of a convolutional neural network

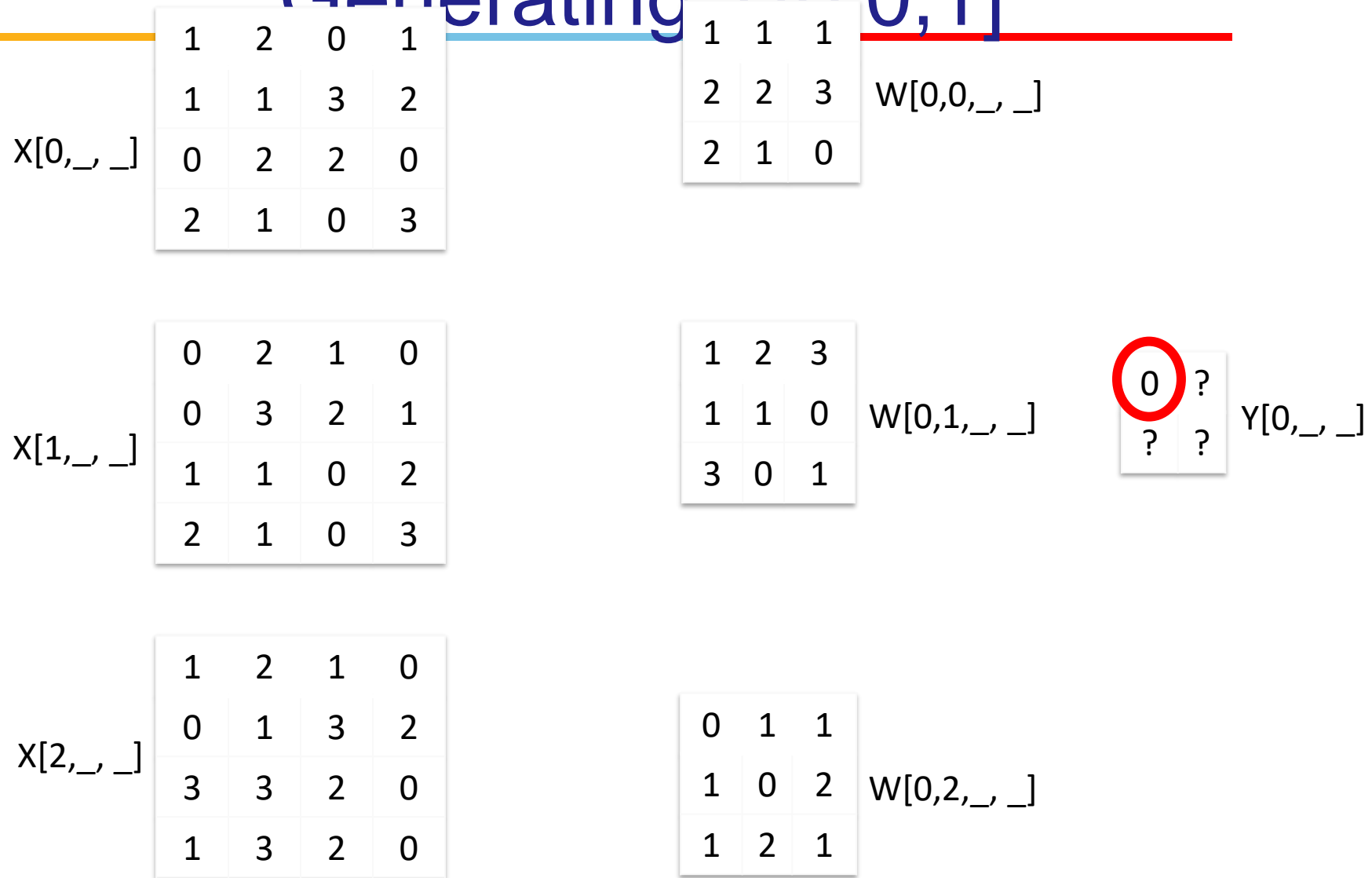
- Levels of parallelism
- Loop transformations
- Basic kernel design

# Sequential Code for the Forward Path of a Convolution Layer

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    for(int m = 0; m < M; m++)                // for each output feature map
        for(int h = 0; h < H_out; h++)          // for each output element
            for(int w = 0; w < W_out; w++) {
                Y[m, h, w] = 0;
                for(int c = 0; c < C; c++)        // sum over all input feature maps
                    for(int p = 0; p < K; p++)    // KxK filter
                        for(int q = 0; q < K; q++)
                            Y[m, h, w] += X[c, h + p, w + q] * W[m, c, p, q];
            }
}
```

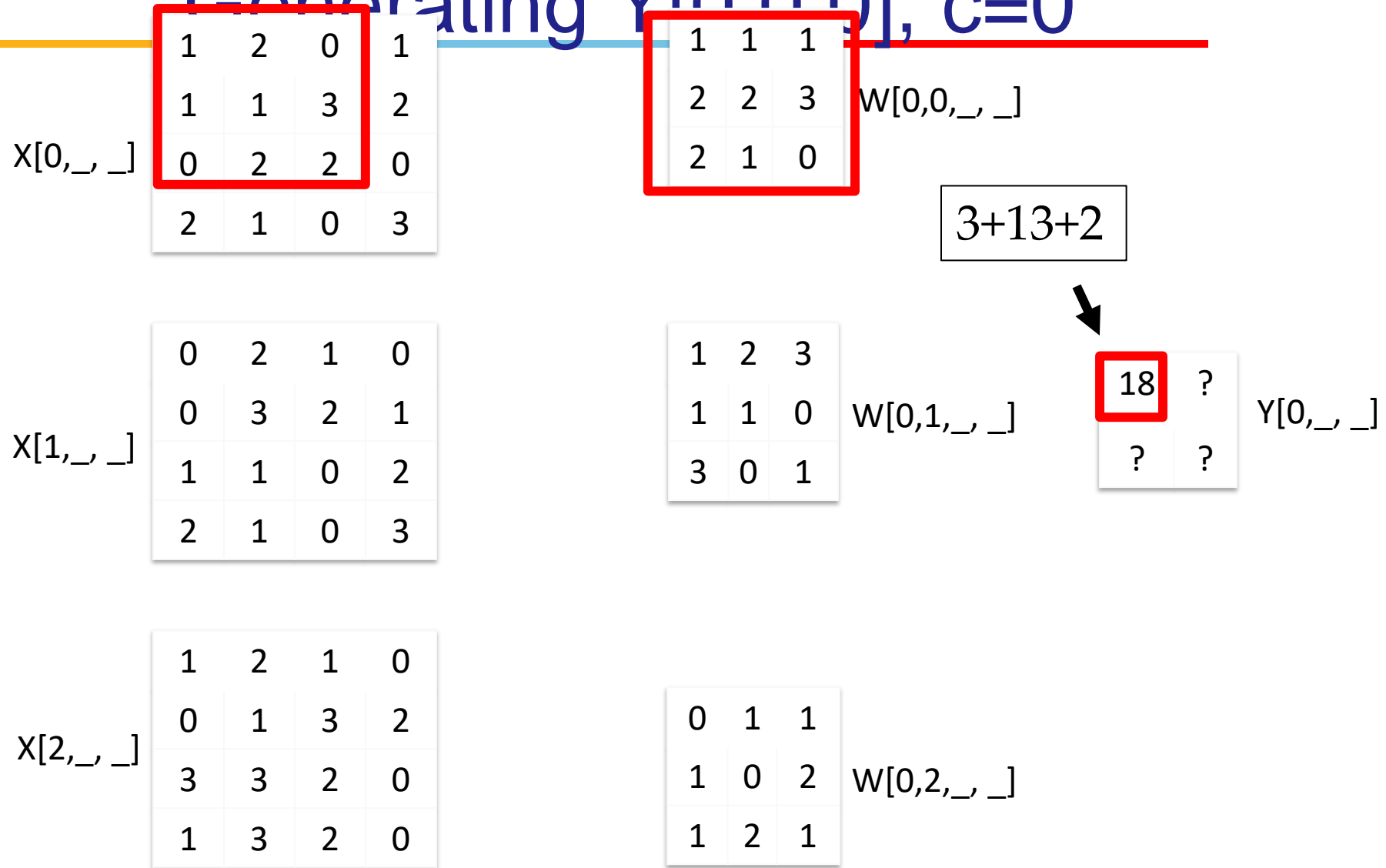
# A Small Convolution Layer Example

## Generating $Y[n,0,1]$



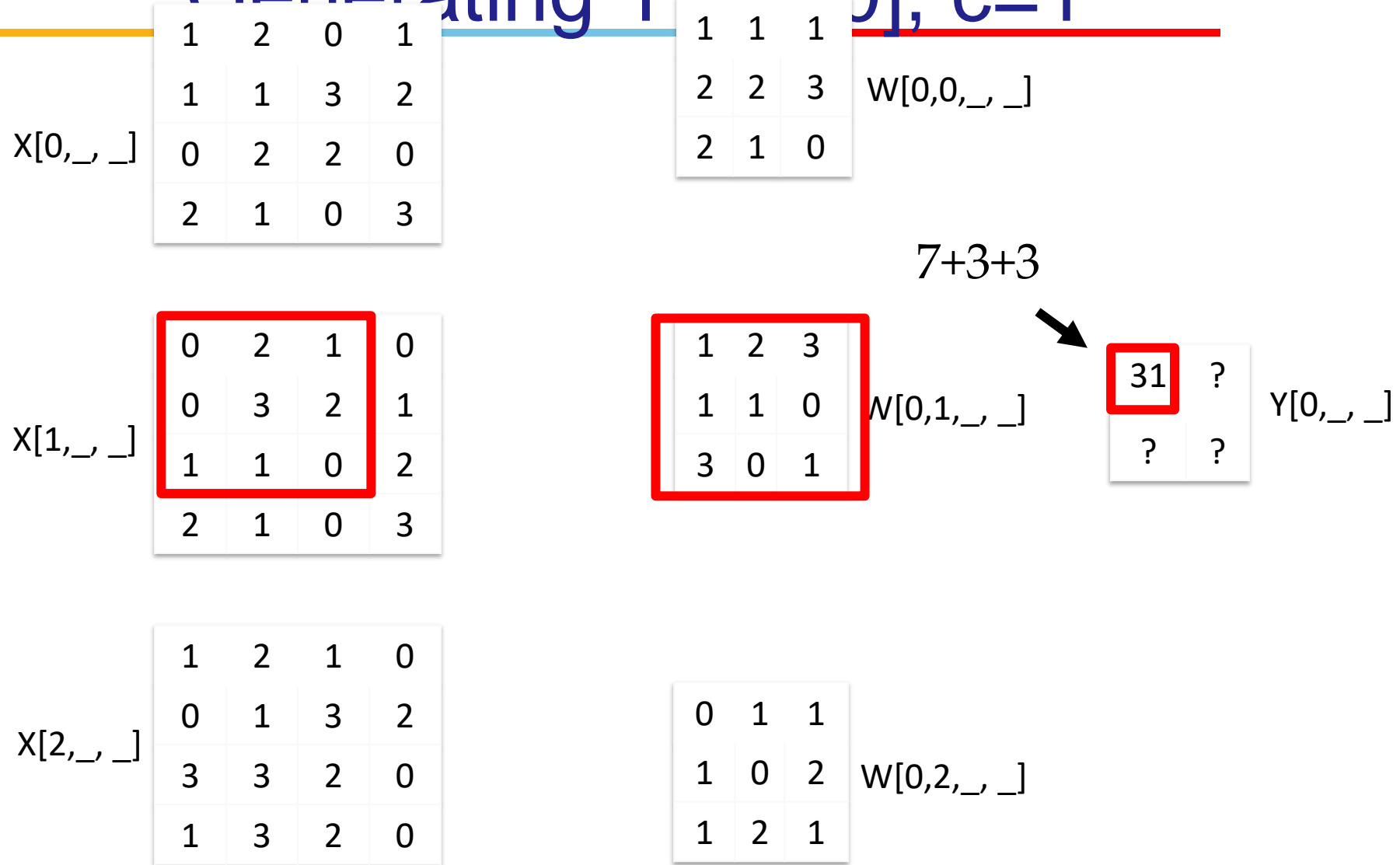
# A Small Convolution Layer Example

Generating  $Y[0,0,0]$ ,  $c=0$



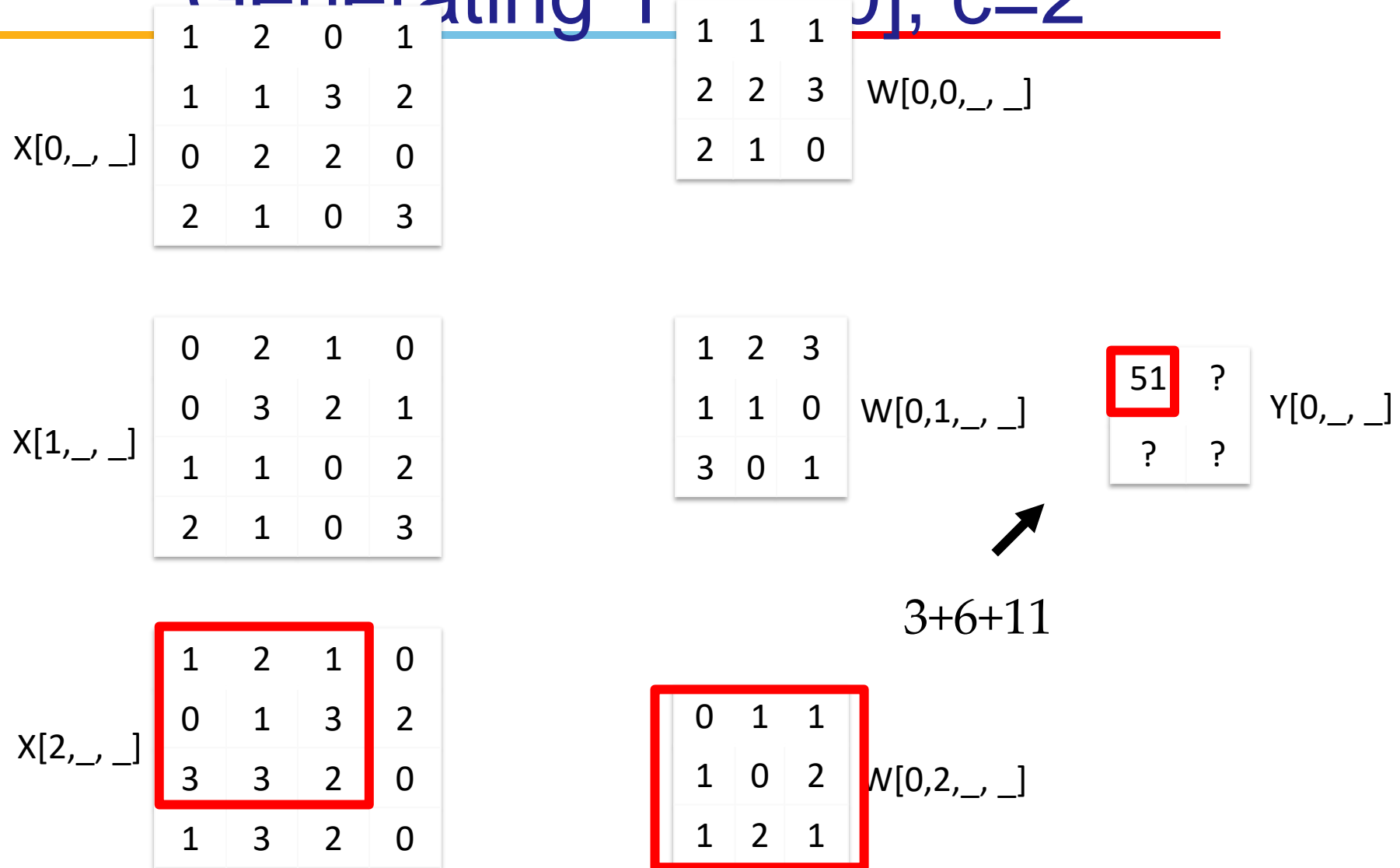
# A Small Convolution Layer Example

Generating  $Y[0,0,0]$ ,  $c=1$



# A Small Convolution Layer Example

## Generating $Y[n,0,0]$ , $c=2$



## Parallelism in a Convolution Layer

---

All output feature maps can be calculated in parallel

- A small number in general, not sufficient to fully utilize a GPU

All output feature map pixels can be calculated in parallel

- All rows can be done in parallel
- All pixels in each row can be done in parallel
- Large number but diminishes as we go into deeper layers

All input feature maps can be processed in parallel, but will need atomic operation or tree reduction

# Design of a Basic Kernel

---

Each block computes a tile of output pixels

- TILE\_WIDTH pixels in each dimension

The first (x) dimension in the grid maps to the M output feature maps

The second (y) dimension in the grid maps to the tiles in the output feature maps



# Host Code for the Basic Kernel

Defining the grid configuration

- W\_out and H\_out are the output feature map width and height

```
# define TILE_WIDTH 16          // We will use 4 for small examples.
W_grid = W_out/TILE_WIDTH;      // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH;      // number of vertical tiles per output map
Y = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(M, Y, 1);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(...);
```

## A Small Example

---

Assume that we will produce 4 output feature maps

- Each output feature map is 8x8 image
- We have 4 blocks in the x dimension

If we use tiles of 4 pixels on each side ( $\text{TILE\_SIZE} = 4$ )

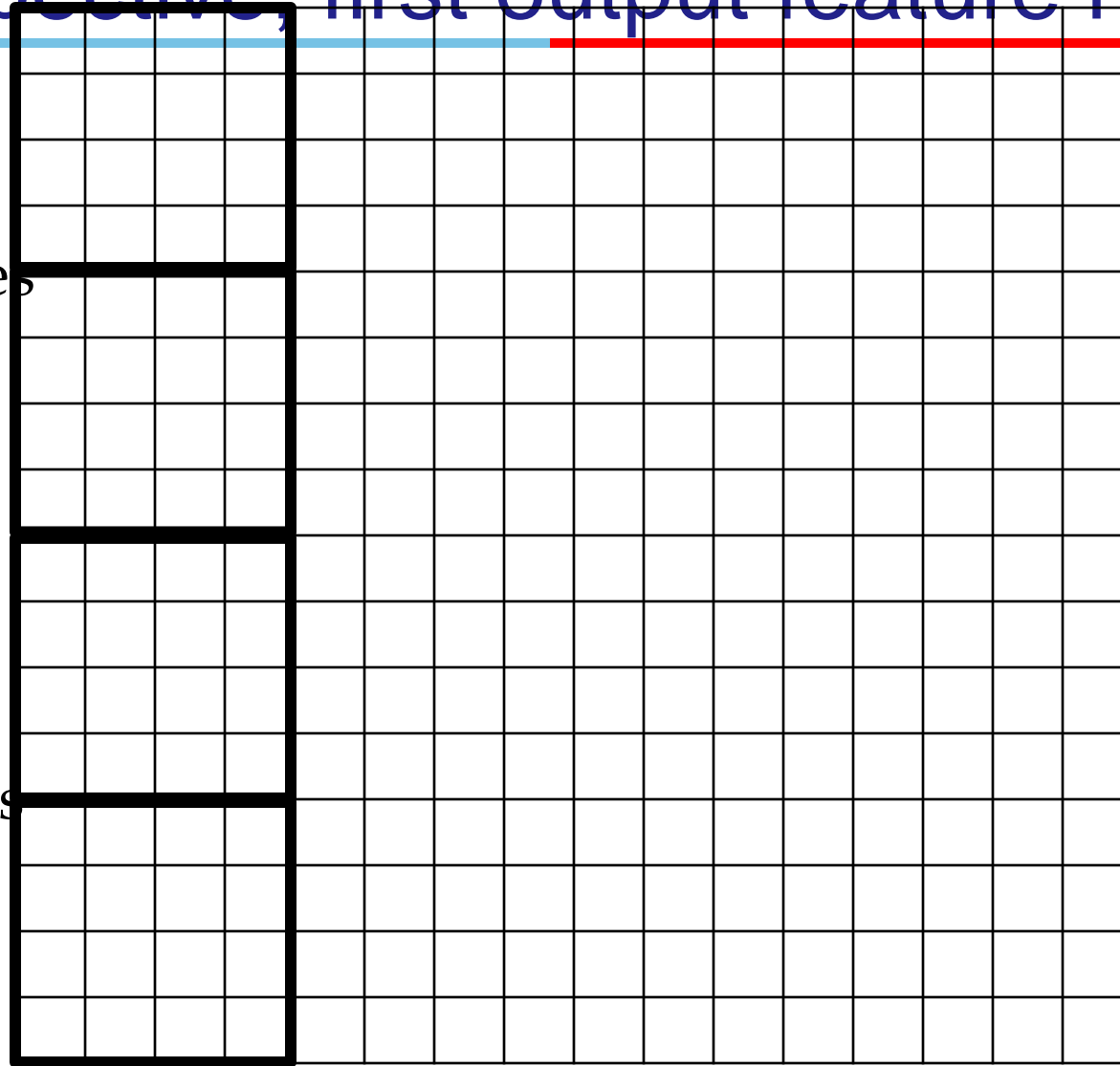
- We have 4 blocks in the x dimension
  - Top two blocks in each column calculates the top row of tiles in the corresponding output feature map
  - Bottom two block in each column calculates the bottom row of tiles in the corresponding output feature map

# Mapping Threads to Output Feature Maps

## Grid Perspective, first output feature map

Row of Tiles  
First

Row of Tiles



# A Basic Conv. Layer Forward Kernel (Code is incomplete!)

```
__global__ void ConvLayerForward_Basic_Kernel(int C, int W_grid, int K,
float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = blockIdx.y / W_grid + threadIdx.y;
    int w = blockIdx.y % W_grid + threadIdx.x;
    float acc = 0.;
    for (int c = 0; c < C; c++) {                // sum over all input channels
        for (int p = 0; p < K; p++)                // loop over KxK filter
            for (int q = 0; q < K; q++)
                acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```

# Some Observations

---

The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large

- This matches the CNN architecture well

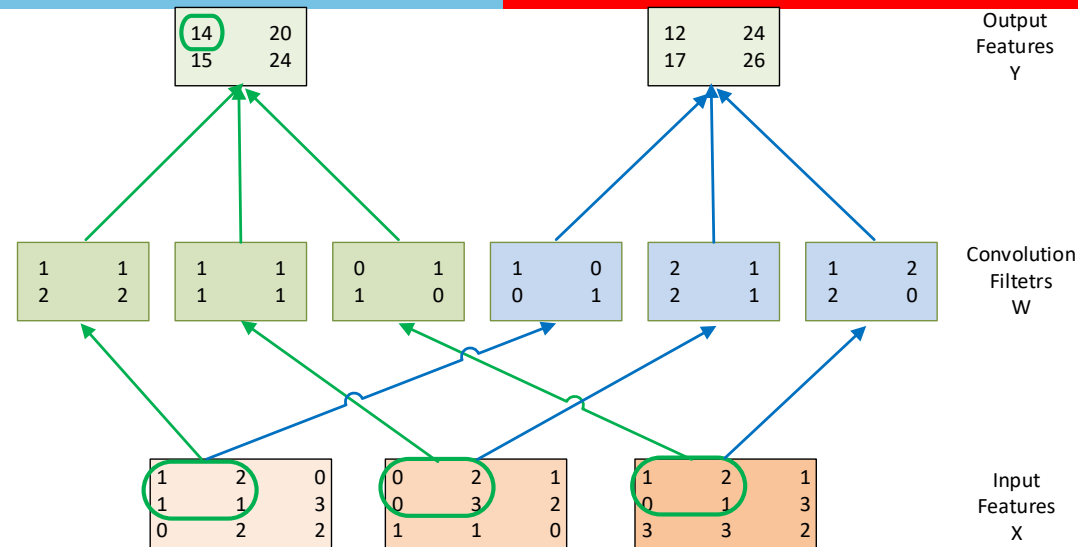
Each input tile is loaded multiple times, once for each block that calculates the output tile that requires the input tile

- Not very efficient in global memory bandwidth

# Sequential code for the Forward Path of a Sub-sampling Layer

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
    for(int m = 0; m < M; m++)                // for each output feature maps
        for(int h = 0; h < H/K; h++)           // for each output element
            for(int w = 0; w < W/K; w++) {
                S[m, x, y] = 0.;
                for(int p = 0; p < K; p++) {     // loop over KxK input samples
                    for(int q = 0; q < K; q++)
                        S[m, h, w] += Y[m, K*h + p, K*w + q] / (K*K);
                }
                // add bias and apply non-linear activation
                S[m, h, w] = sigmoid(S[m, h, w] + b[m])
            }
}
```

# Implementing a convolution layer with matrix multiplication



$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline 2 & 1 & 2 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 1 & 2 & 2 & 0 \\ \hline \end{array}
 \quad * \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 3 \\ \hline 1 & 1 & 0 & 2 \\ \hline 1 & 3 & 2 & 2 \\ \hline 0 & 2 & 0 & 3 \\ \hline 2 & 1 & 3 & 2 \\ \hline 0 & 3 & 1 & 1 \\ \hline 3 & 2 & 1 & 0 \\ \hline 1 & 2 & 1 & 1 \\ \hline 2 & 1 & 0 & 3 \\ \hline 0 & 1 & 3 & 3 \\ \hline 1 & 3 & 3 & 2 \\ \hline \end{array}
 \quad = \quad
 \begin{array}{|c|c|c|c|} \hline 14 & 20 & 15 & 24 \\ \hline 12 & 24 & 17 & 26 \\ \hline \end{array}$$

Convolution Filters  $W'$ 
Input Features  $X_{\text{unrolled}}$ 
Output Features  $Y$





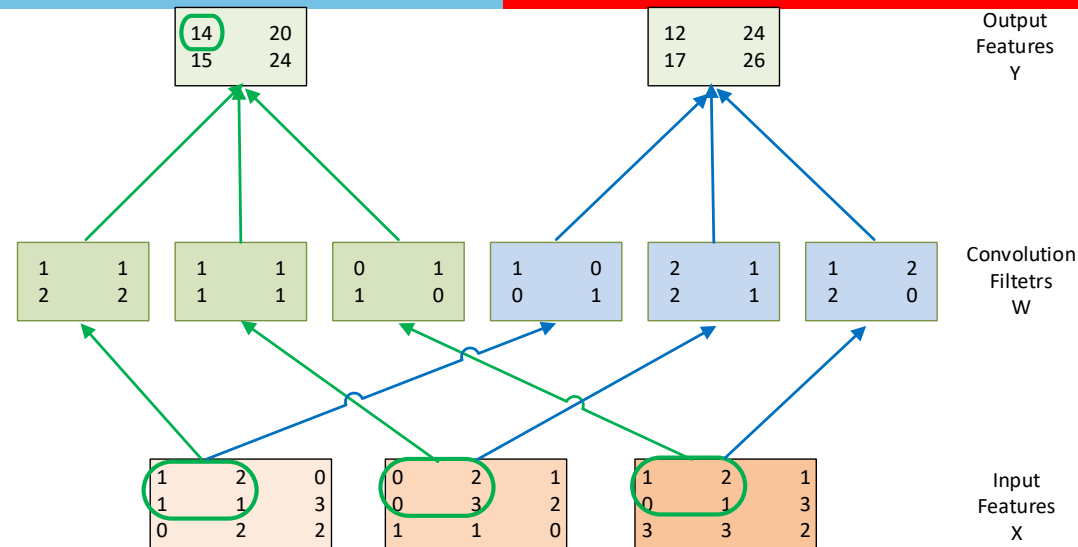
# Objective

---

To learn more about the implementation of a convolutional neural network

- Optimizations

# Implementing a convolution layer with matrix multiplication



$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline 2 & 1 & 2 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 1 & 2 & 2 & 0 \\ \hline \end{array}
 \quad * \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 3 \\ \hline 1 & 1 & 0 & 2 \\ \hline 1 & 3 & 2 & 2 \\ \hline 0 & 2 & 0 & 3 \\ \hline 2 & 1 & 3 & 2 \\ \hline 0 & 3 & 1 & 1 \\ \hline 3 & 2 & 1 & 0 \\ \hline 1 & 2 & 1 & 1 \\ \hline 2 & 1 & 0 & 3 \\ \hline 0 & 1 & 3 & 3 \\ \hline 1 & 3 & 3 & 2 \\ \hline \end{array}
 \quad = \quad
 \begin{array}{|c|c|c|c|} \hline 14 & 20 & 15 & 24 \\ \hline 12 & 24 & 17 & 26 \\ \hline \end{array}$$

Convolution Filters  $W'$ 
Input Features  $X_{\text{unrolled}}$ 
Output Features  $Y$

# Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.

## Convolution Filters

0	1	1	2	2	1	1	1	1	0	1	1	0
1	1	0	0	1	2	1	2	1	1	2	2	0

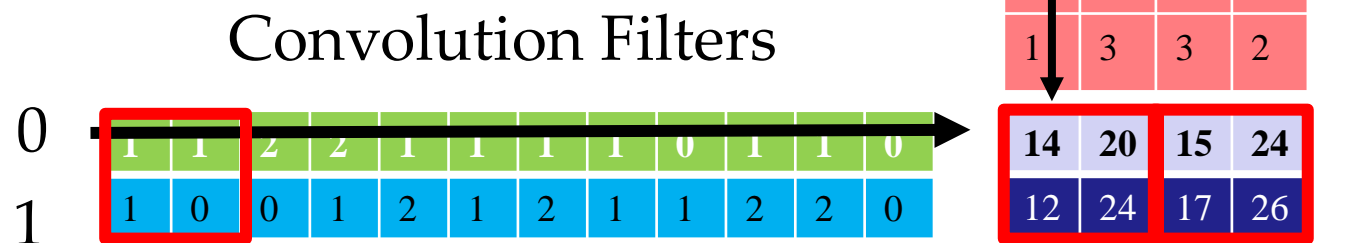
Input feature maps	0	1	2	1	1
		2	0	1	3
		1	1	0	2
		1	3	2	2
	1	0	2	0	3
		2	1	3	2
		0	3	1	1
		3	2	1	0
	2	1	2	1	1
		2	1	0	3
		0	1	3	3
		1	3	3	2
		14	20	15	24
		12	24	17	26

# Tiled Matrix Multiplication

## 2x2 example

Each block calculates one output tile  
– 2 elements from each output map

Each input element is reused 2 times  
in the shared memory

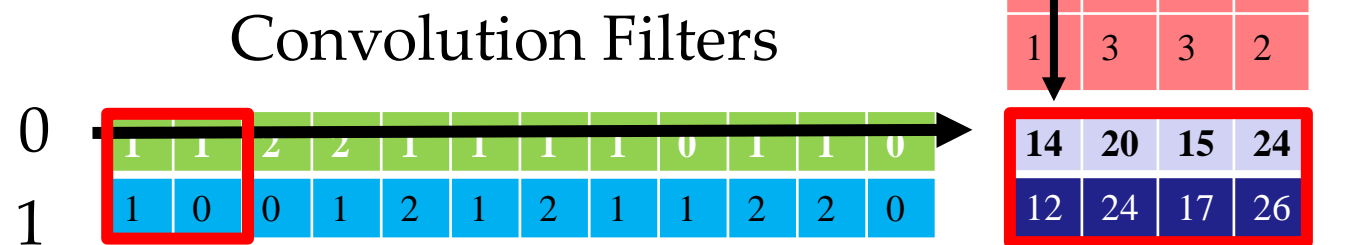


# Tiled Matrix Multiplication

## 2x4 example

Each block calculates one output tile  
– 4 elements from each output map

Each input element is reused 2 times  
in the shared memory



## Analysis of Efficiency

### Total Input Replication

---

Each output map requires its own replicated  $K \times K$  input feature map elements

- Not replicated for different output feature maps
- There are  $H_{out} \times W_{out}$  output feature map elements
- Each requires  $K \times K$  replicated input feature map elements
- So, the total number of input element after replication is  $H_{out} \times W_{out} \times K \times K$  times for each input feature map
- The total number of elements in each original input feature map is  $(H_{out} - K + 1) \times (W_{out} - K + 1)$

# Analysis of Small Example

$$H_{out} = 2$$

$$W_{out} = 2$$

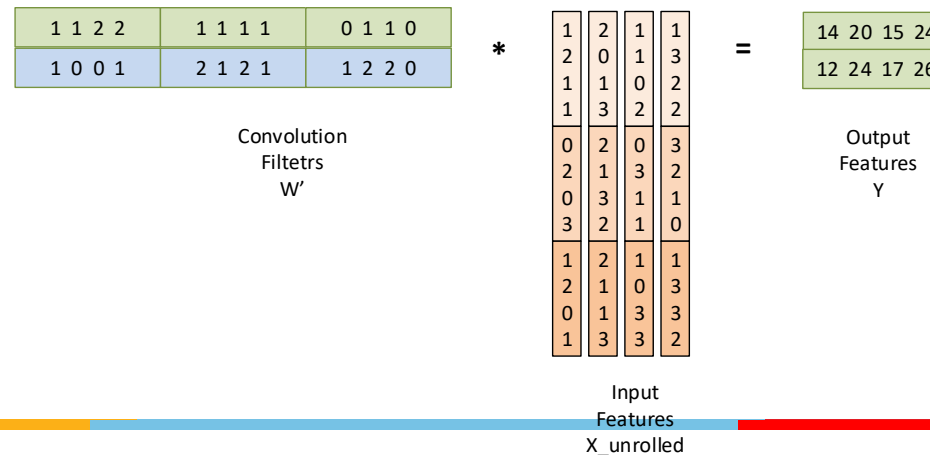
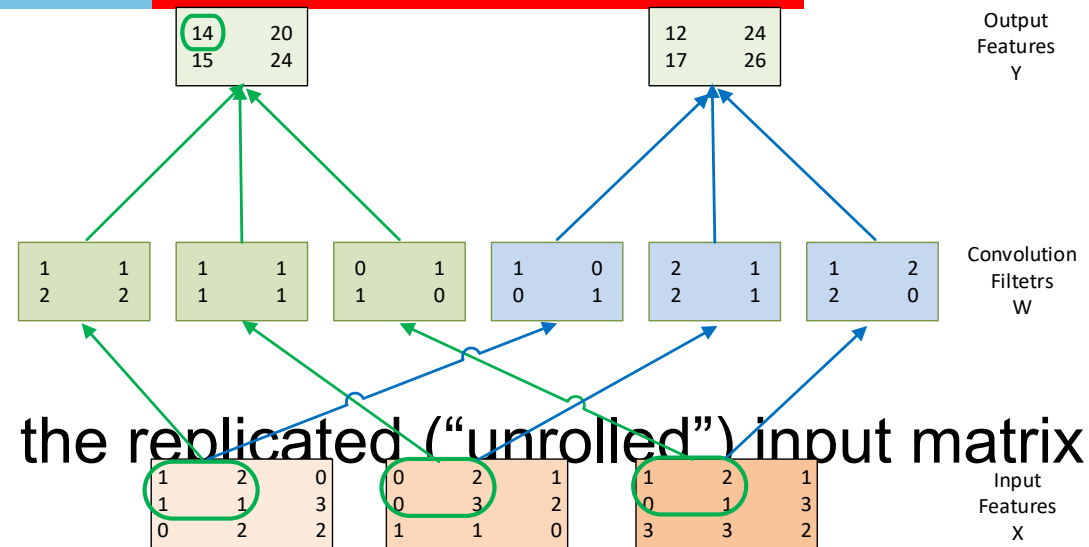
$$K = 2$$

There are 3 input maps (channels)

The total number of input elements in the replicated ("unrolled") input matrix is  
 $3*2*2*2*2$

The replicating factor is

$$(3*2*2*2*2)/(3*3*3) = 1.78$$



# Memory Access Efficiency of Original Convolution Algorithm

---

Assume that we use tiled 2D convolution

For input elements

- Each output tile has  $\text{TILE\_WIDTH}^2$  elements
- Each input tile has  $(\text{TILE\_WIDTH}+K-1)^2$
- The total number of input feature map element accesses was  $\text{TILE\_WIDTH}^2 * K^2$
- The reduction factor of the tiled algorithm is  $K^2 * \text{TILE\_WIDTH}^2 / (\text{TILE\_WIDTH}+K-1)^2$

The convolution filter weight elements are reused within each output tile



# Efficiency of Tiled Matrix Multiplication

---

Assuming we use  $\text{TILE\_WIDTH}^2$  input and output tiles

- Each replicated input feature map element is reused  $\text{TILE\_WIDTH}$  times
- Each convolution filter weight element is reused  $\text{TILE\_WIDTH}$  times
- Matrix multiplication better if  $\text{TILE\_WIDTH}$  is larger than  $K^2 \cdot \text{TILE\_WIDTH}^2 / (\text{TILE\_WIDTH} - K + 1)^2$

# Problem with the Later Stages

---

The size ( $H_{out}$ ,  $W_{out}$ ) of each output feature map decreases as we go to the later stages of the CNN

- The TILE\_WIDTH may be limited to very small sizes relative to  $K$
- The benefit of 2D tiling will diminish as we go down the pipeline
- This is an intrinsic problem for 2D tiled convolution

## Tiled Matrix-Multiplication is more stable in matrix sizes

---

The filter-bank matrix is an  $M \times C \cdot K \cdot K$  matrix.

The expanded input feature map matrix is a  $C \cdot K \cdot K \times H_{out} \cdot W_{out}$  matrix.

Except for the height of the filter-bank matrix, the sizes of all dimensions depend on products of the parameters to the convolution, not the parameters themselves.

While individual parameters can be small, their products tend to be large.

- The amount of work for each kernel launch will remain large as we go to the later stages of the CNN

# Mini-Batching

---

One can use mini-batching to further increase the amount of work done in each kernel launch

- Collect several sets of input feature maps of an input sequence
- Use a larger unrolled input feature matrix that has all the inputs from the mini-batch

# Some Other Optimization

---

Use streams to overlap the reading of the next set of input feature maps with the processing of the previous input feature maps.

Create unrolled matrix elements on the fly, only when they are loaded into shared memory

Use more advanced algorithms such as FFT to implement convolution

# Gradient Back-Propagation

---

Training of ConvNets is based on a procedure called gradient back-propagation.

The training data set is labeled with the “correct answer.”

- In the hand writing recognition example, the labels give the correct letter in the image. The label information can be used to generate the “correct” output of the last stage: the “correct” probability values of the 10-element vector.

For each training image, the final stage of the network calculates the loss function or the error as the difference between the generated output vector element values and the “correct” output vector element values.

Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the output vector. Intuitively, it gives the rate at which the error changes when the value of the output vector changes –  $dE/dY$

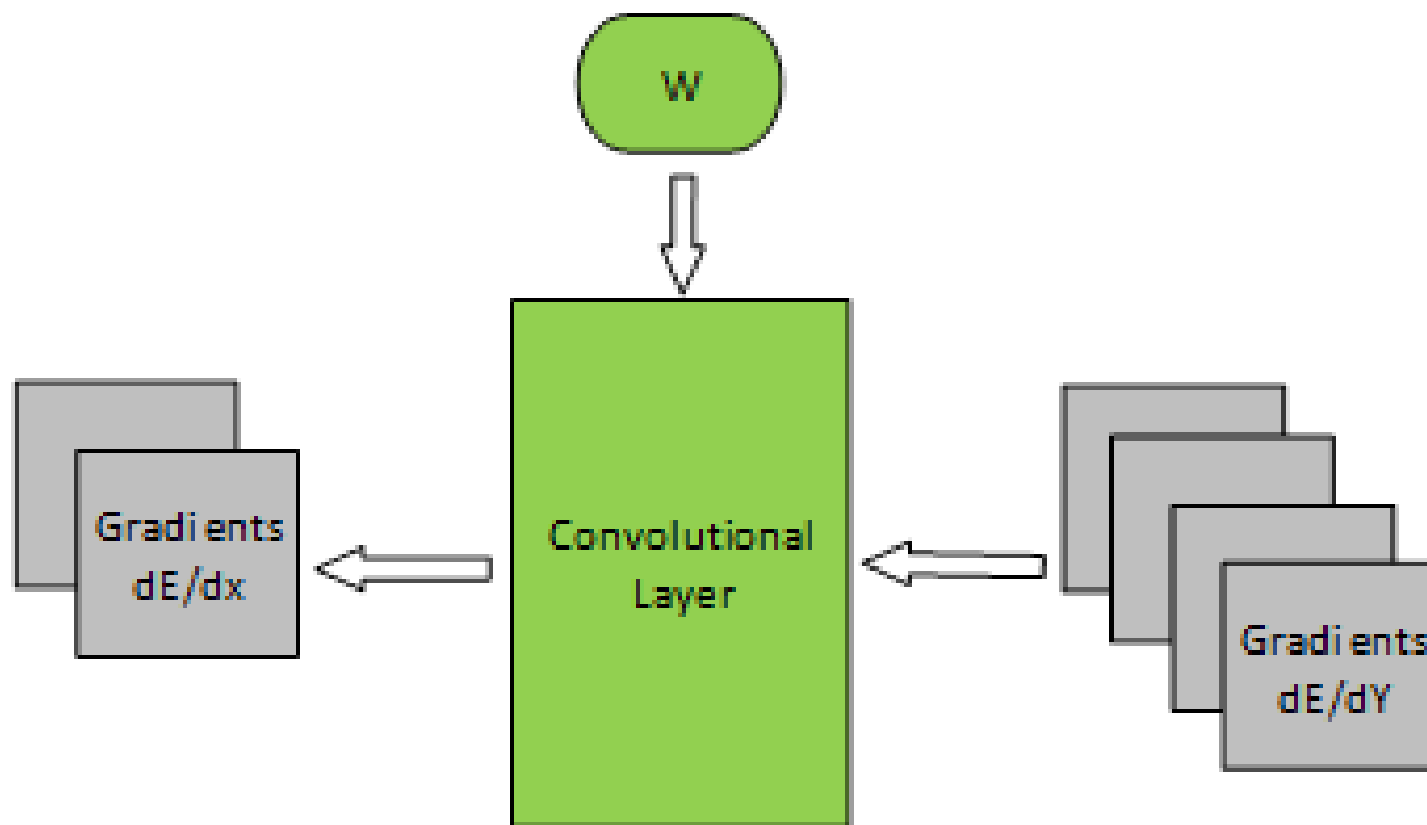
## Gradient Back Propagation (Cont.)

---

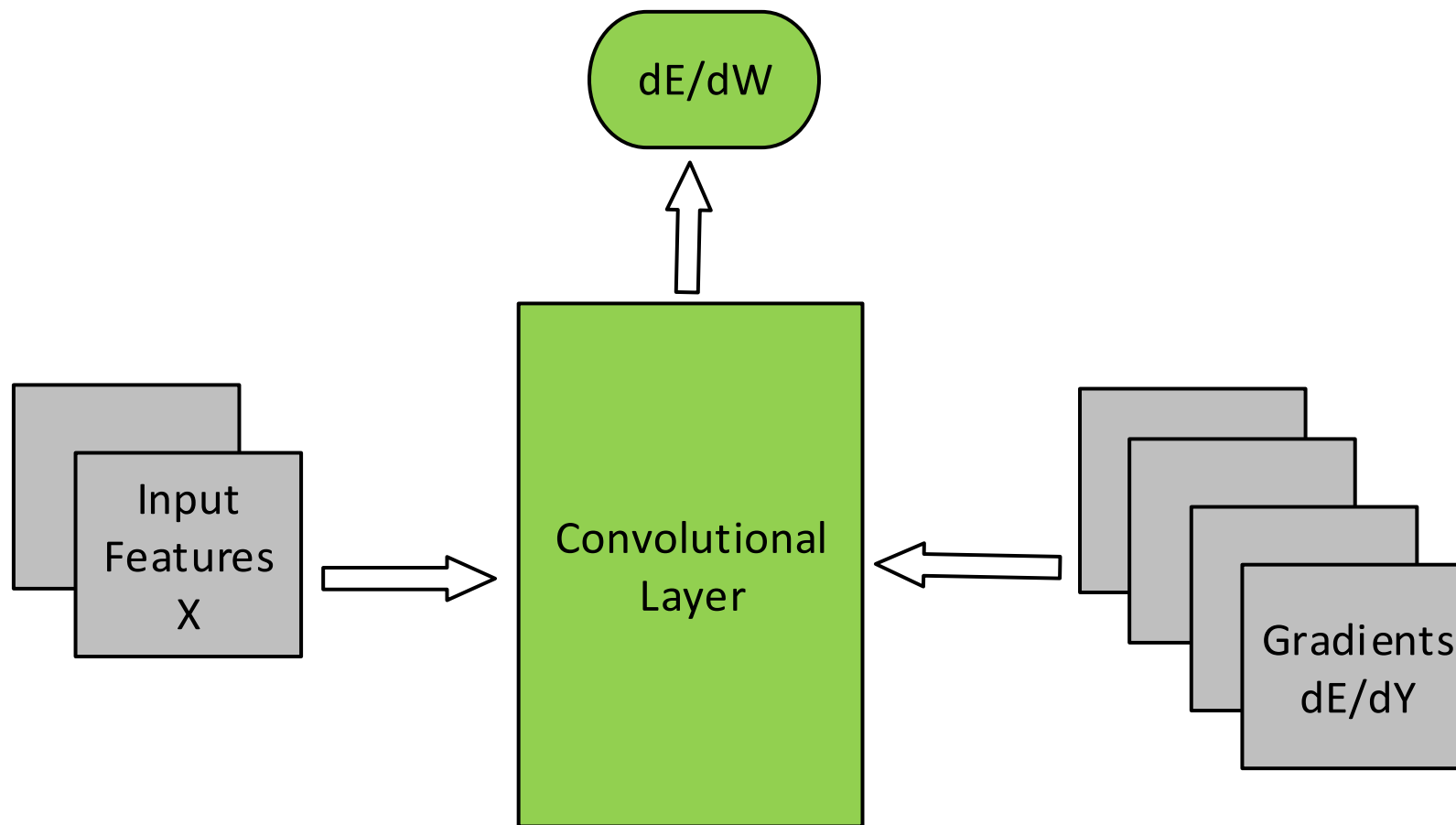
The process propagates the gradient from the last layer towards the first layer through all layers of network.

Each layer receives as  $dE/dY$  – gradient with respect to its output feature maps and computes  $dE/dX$  – gradient with respect to its input feature maps

## Convolution Layer – Back Propagation of $dE/dY$







# Adjusting Weights

---

After the  $dE/dW$  values at all feature map element positions are computed, weights are updated:

For each weight value

$$w(t+1) = w(t) - \lambda / dE/dw,$$

where  $\lambda$  is the learning rate.

## Calculating $dE/dX$

```
void convLayer_backward_dgrad(int M, int C, int H, int W, int K,  
    float* dE_dY, float* W, float* dE_dX)  
{  
    int m, c, h, w, p, q;  
    int H_out = H - K + 1;  
    int W_out = W - K + 1;  
    for(c = 0; c < C; c++)  
        for(h = 0; h < H; h++)  
            for(w = 0; w < W; w++)  
                dE_dX[c, h, w] = 0.;  
  
    for(m = 0; m < M; m++)  
        for(h = 0; h < H_out; h++)  
            for(w = 0; w < W_out; w++)  
                for(c = 0; c < C; c++)  
                    for(p = 0; p < K; p++)
```

## Calculating $dE/dW$

```
void convLayer_backward_wgrad(int M, int C, int H, int W, int K,  
    float* dE_dY, float* X, float* dE_dW)  
{  
    int m, c, h, w, p, q;  
    int H_out = H - K + 1;  
    int W_out = W - K + 1;  
    for(m = 0; m < M; m++)  
        for(c = 0; c < C; c++)  
            for(p = 0; p < K; p++)  
                for(q = 0; q < K; q++)  
                    dE_dW[m, c, p, q] = 0.;  
    for(m = 0; m < M; m++)  
        for(h = 0; h < H_out; h++)  
            for(w = 0; w < W_out; w++)  
                for(c = 0; c < C; c++)  
                    for(p = 0; p < K; p++)
```



# Objective

---

To solidify your understanding and point you to the future

- Future directions of programming models
- Next courses
- Important takeaways from the course, exam plan
- Course evaluation

# Higher-Level Programming Interfaces

---

# OpenACC

---

The OpenACC Application Programming Interface provides a set of

- compiler directives (pragmas)
- library routines and
- environment variables

that can be used to write data parallel FORTRAN, C and C++ programs that run on accelerator devices including GPUs and CPUs



# OpenACC Pragmas

---

In C and C++, the `#pragma` directive is the method to provide, to the compiler, information that is not specified in the standard language.

# Simple Matrix-Matrix Multiplication in OpenACC

```
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2 {
3
4 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
5 for (int i=0; i<Mh; i++) {
6     #pragma acc loop
7     for (int j=0; j<Nw; j++) {
8         float sum = 0;
9         for (int k=0; k<Mw; k++) {
10             float a = M[i*Mw+k];
11             float b = N[k*Nw+j];
12             sum += a*b;
13         }
14         P[i*Nw+j] = sum;
15     }
16 }
17 }
```

## Some Observations

---

The code is almost identical to the sequential version, except for the two lines with `#pragma` at line 4 and line 6.

OpenACC uses the compiler directive mechanism to extend the base language.

- `#pragma` at line 4 tells the compiler to generate code for the 'i' loop at line 5 through 16 so that the loop iterations are executed in parallel on the accelerator.
- The `copyin` clause and the `copyout` clause specify how the matrix data should be transferred between the host and the accelerator. The `#pragma` at line 6 instructs the compiler to map the inner 'j' loop to the second level of parallelism on the accelerator.

# Motivation

---

OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.

- leave most of the details in generating a kernel and data transfers to the OpenACC compiler.

OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

# Frequently Encountered Issues

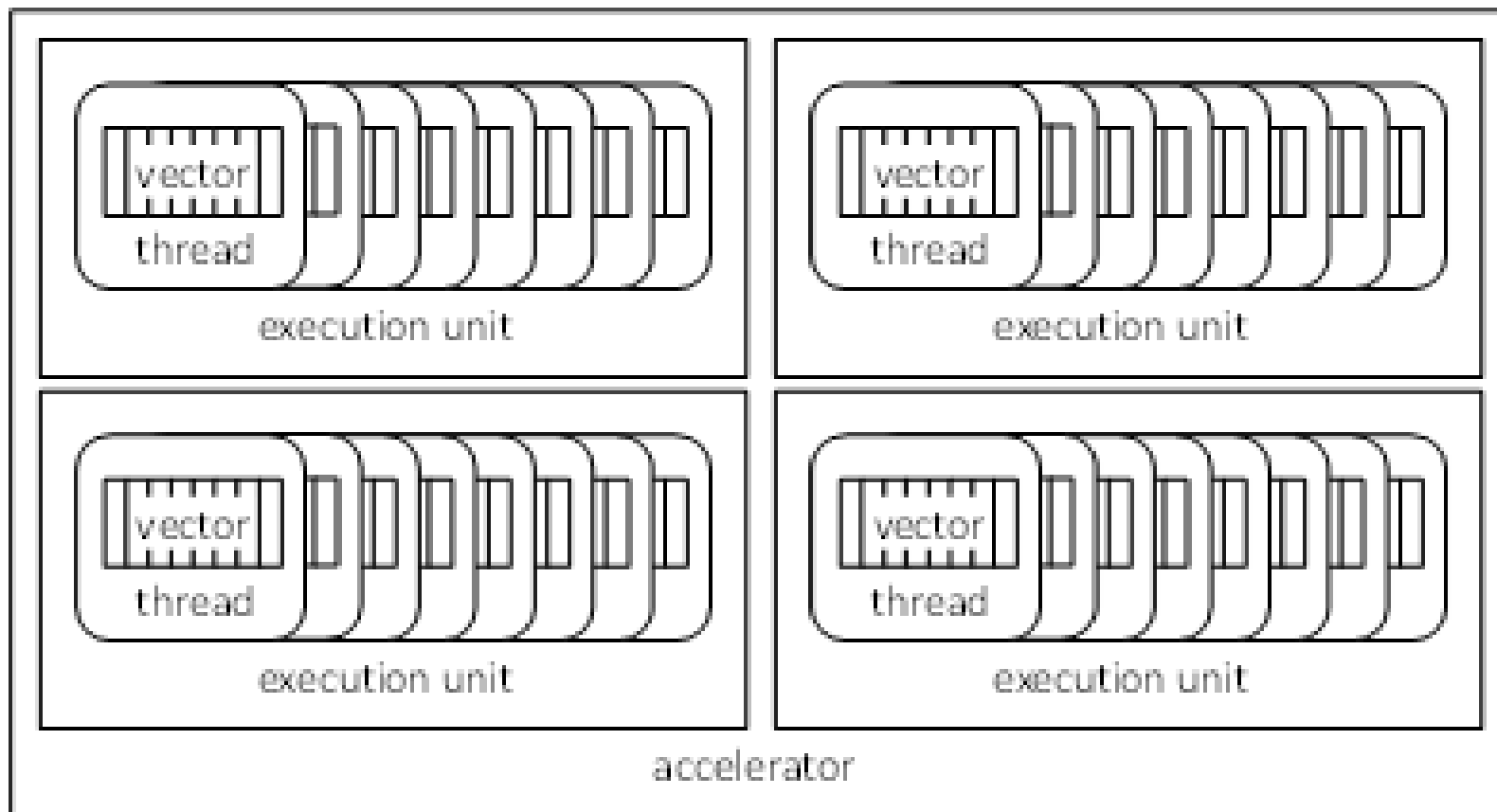
---

Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly

- The performance of an OpenACC depends heavily on the quality of the compiler.
- Much less so in CUDA or OpenCL

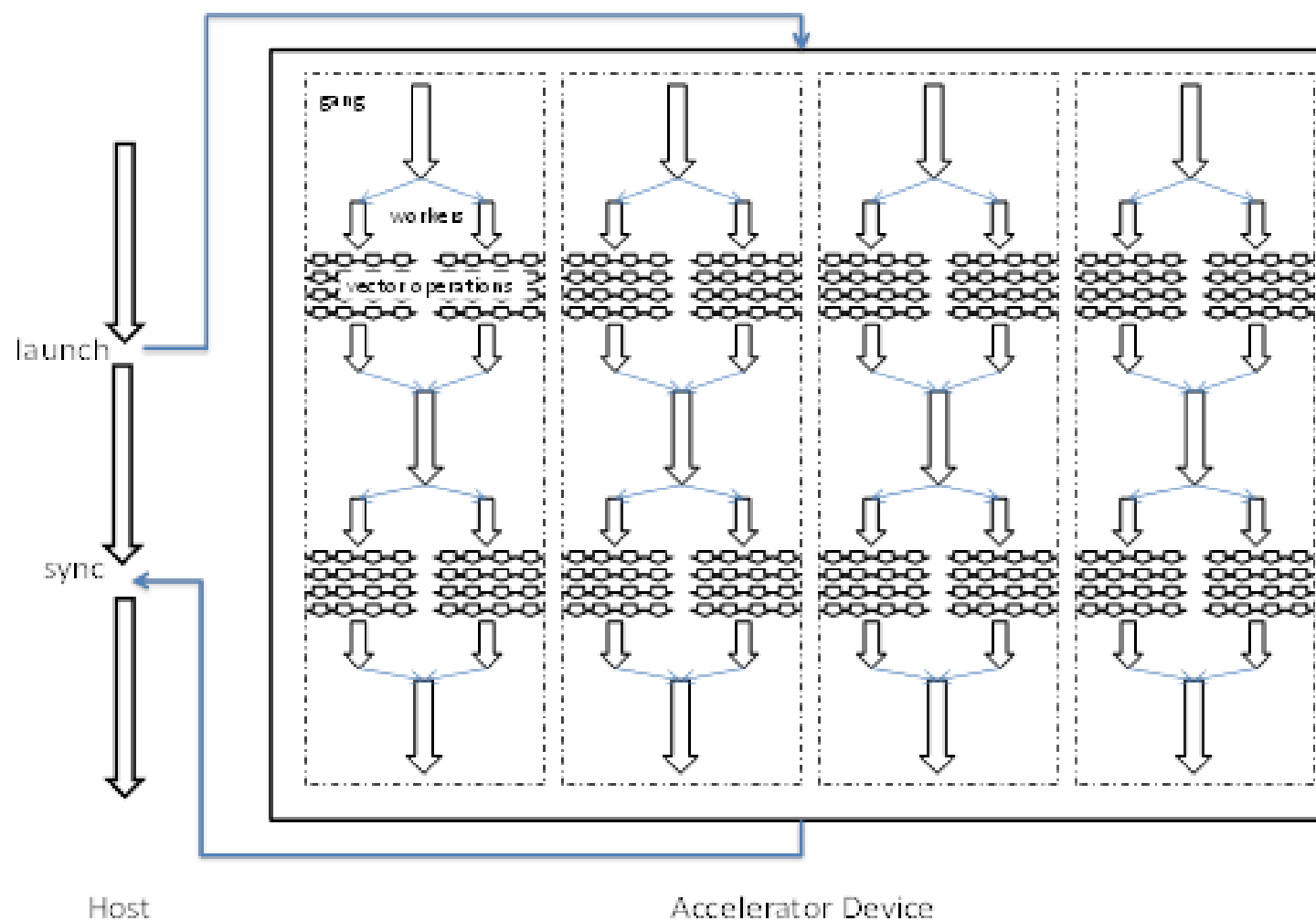
Some OpenACC programs may behave differently or even incorrectly if pragmas are ignored

# OpenACC Device Model



Currently OpenACC does not allow synchronization across threads.

# OpenACC Execution Model



# Parallel vs. Loop Constructs

```
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])  
for (int i=0; i<Mh; i++) {  
    ...  
}
```

is equivalent to:

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])  
{  
    #pragma acc loop  
    for (int i=0; i<Mh; i++) {  
        ...  
    }  
}
```

(a parallel region that consists of just a loop)



# Parallel Construct

---

A parallel construct is executed on an accelerator

One can specify the number of gangs and number of works in each gang

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
    a = 23;
}
```

1024\*32 workers will be created. a=23 will be executed  
redundantly by all 1024 gang leads

## What does each “Gang Loop” do?

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        ...
    }
}
```

```
#pragma acc parallel
num_gangs(1024)
{
    #pragma acc loop gang
        for (int i=0; i<2048; i++) {
            ...
        }
}
```

# Worker Loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}
```

1024\*32=32K workers will be created, each executing 1M/32K = 32 instance of foo()

---

```
#pragma acc parallel num_gangs(32)
{
    Statement 1; Statement 2;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 3; Statement 4;
    }
    Statement 5; Statement 6;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 7; Statement 8;
    }
    Statement 9;
    if (condition)
        Statement 10;
}
```

- Statements 1 and 2 are redundantly executed by 32 gangs
- The n for-loop iterations are distributed to 32 gangs

# Key to Programming to OpenACC

---

Understanding the Gangs (thread blocks) and Workers (threads)

Understanding the data copyin copyout

Understanding the amount of work done by each worker

The best way to learn to program in OpenACC well is to learn CUDA first

# Next Steps



Computational Thinking and Algorithm Techniques

# Fundamentals of Parallel Computing

---

Parallel computing requires that

- The problem can be decomposed into sub-problems that can be safely solved at the same time
- The programmer structures the code and data to solve these sub-problems concurrently

The goals of parallel computing are

- To solve problems in less time, and/or
- To solve bigger problems, and/or
- To achieve better solutions

**The problems must be large enough to *justify* parallel computing and to exhibit *exploitable concurrency*.**

# Shared Memory vs. Message Passing

---

We have focused on shared memory programming

- This is what CUDA (and OpenMP/OpenACC, OpenCL) is based on
- Future massively parallel microprocessors are expected to support shared memory at the chip level
- This is different than global address space (single pointer space)

The programming considerations of message passing model is quite different!

- Look at MPI (Message Passing Interface) and its relatives such as Charm++



# Program Models

---

## SPMD (Single Program, Multiple Data)

- All PE's (Processor Elements) execute the same program in parallel, but has its own data
- Each PE uses a unique ID to access its portion of data
- Different PE can follow different paths through the same code
- This is essentially the CUDA Grid model (also MPI)
- SIMD is a special case - WARP

Master/Worker (CUDA Streams)

Loop Parallelism (OpenMP/OpenACC)

Fork/Join (Posix p-threads)

# A Simple Example of Parallel Programming Pitfalls

---

Assume

- The computation being parallelized has 1,000,000 iterations.

Sequential code:

```
Num_steps = 1000000;  
  
for (i=0; i< num_steps, i++) {  
    ...  
}
```

## SPMD Code Version 1

### Assign a chunk of iterations to each thread

- The last thread also finishes up the remainder iterations

```
num_steps = 1000000;  
  
i_start = my_id * (num_steps/num_threads);  
i_end = i_start + (num_steps/num_threads);  
if (my_id == (num_threads-1)) i_end = num_steps;  
  
for (i = i_start; i < i_end; i++) {  
    ....  
}  
Reconciliation of results across threads if necessary.
```

## Problems with Version 1

---

The last thread executes more iterations than others

The number of extra iterations is up to the total number of threads –

1

- This is not a big problem when the number of threads is small
- When there are thousands of threads, this can create serious load imbalance problems.

## SPMD Code Version 2

Assign one more iteration to some of the threads

```
int rem = num_steps % num_threads;
```

```
i_start = my_id * (num_steps/num_threads);
```

```
i_end = i_start + (num_steps/num_threads);
```

```
if (rem != 0) {
```

```
    if (my_id < rem) {
```

```
        i_start += my_id;
```

```
        i_end += (my_id + 1);
```

```
    }
```

```
    else {
```

```
        i_start += rem;
```

```
        i_end += rem;
```

```
    }
```

**Less load imbalance**

**More branch divergence.**

## SPMD Code Version 3

---

Use cyclic distribution of iteration

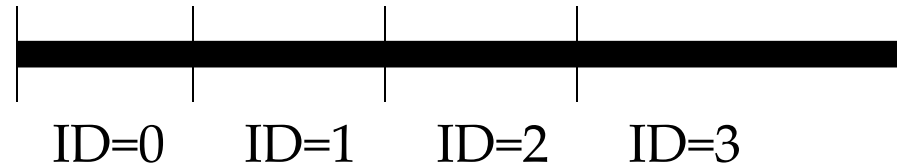
```
num_steps = 1000000;  
  
for (i = my_id; i < num_steps; i += num_threads) {  
    ....  
}
```

**Less load imbalance**

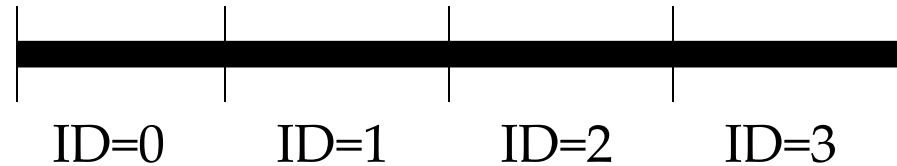
**Loop branch divergence in the last Warp**

## Comparing the Three Versions

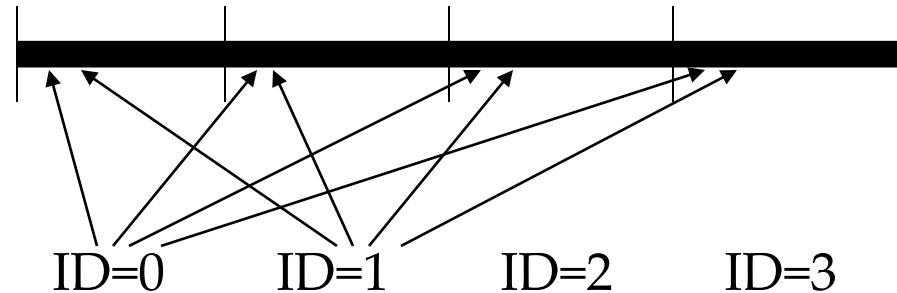
Version 1



Version 2



Version 3



Padded version 1 may be best  
for some data access patterns.