POLITECNICO DI MILANO

Parallel Patterns in OpenMP and CUDA

Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

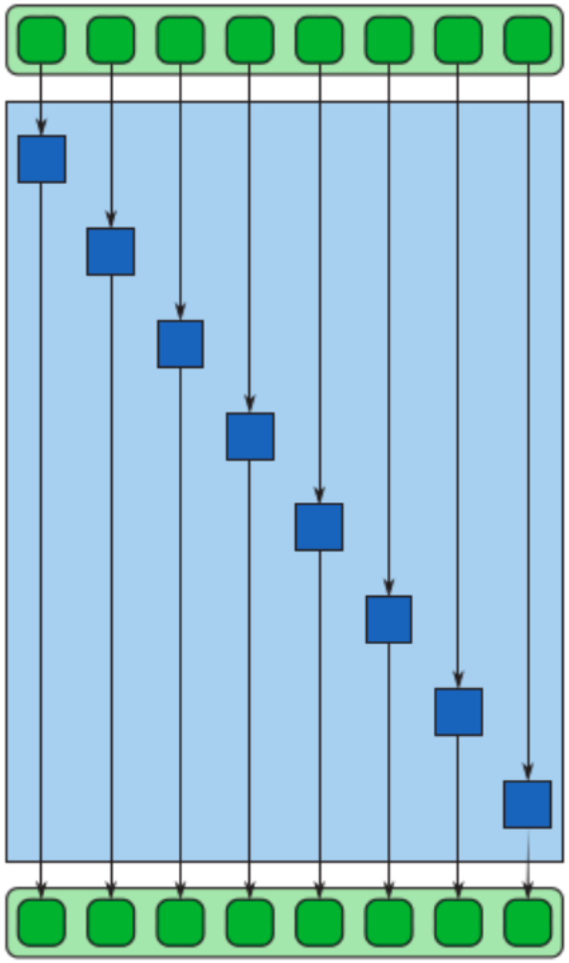
Parallel Patterns in OpenMP

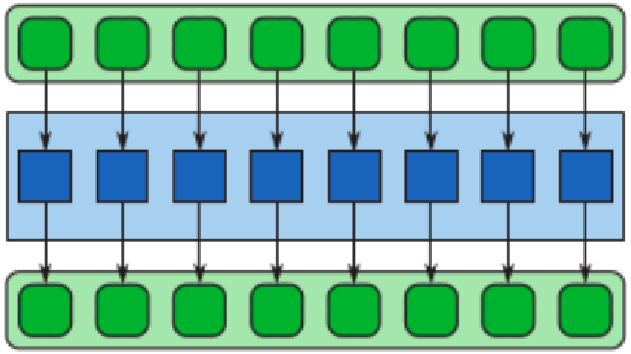
- We can use the OpenMP API to implement parallel patterns
- Some of them come for free!

- Notebook (includes link to download code examples):

  https://colab.research.google.com/drive/18aKcU9i11pqz629vP_-kWOC9bhJtvBfc?usp=share_link

Map Pattern

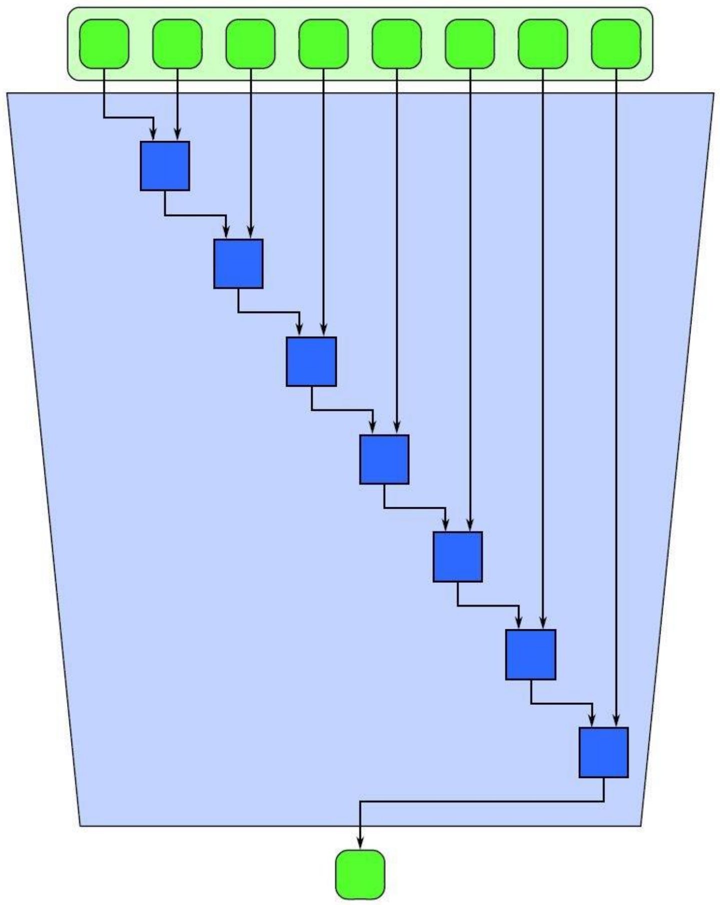


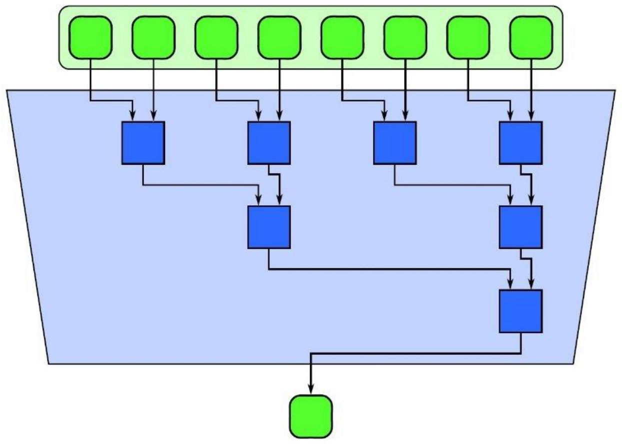


- SIMD, no dependencies between operations
- Comes for free

```
#pragma omp parallel for
```

Reduction Pattern

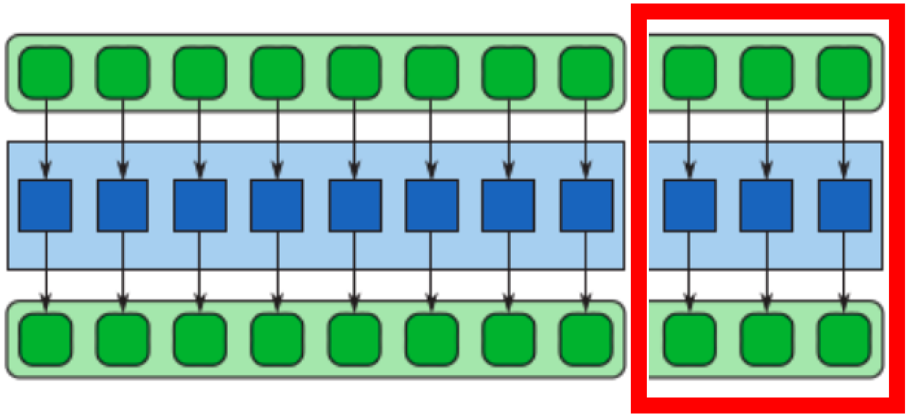




- Also comes for free

```
#pragma omp parallel for
reduction(+:var)
```

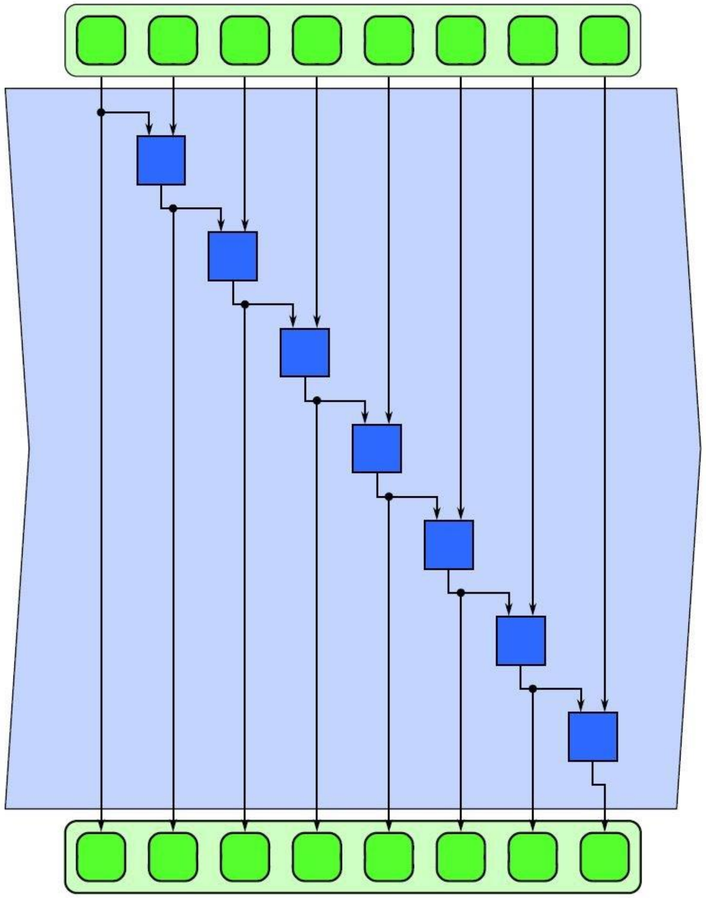
Workpile Pattern

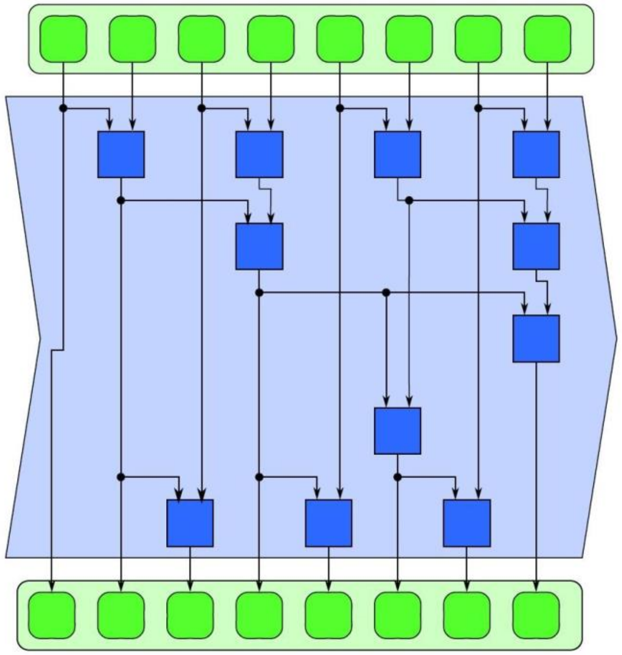


- The structure of the work is not regular, it can change at runtime
- Assumes that all executions are independent
- Typical application: tree searches

```
#pragma omp task
```

Scan Pattern - OpenMP

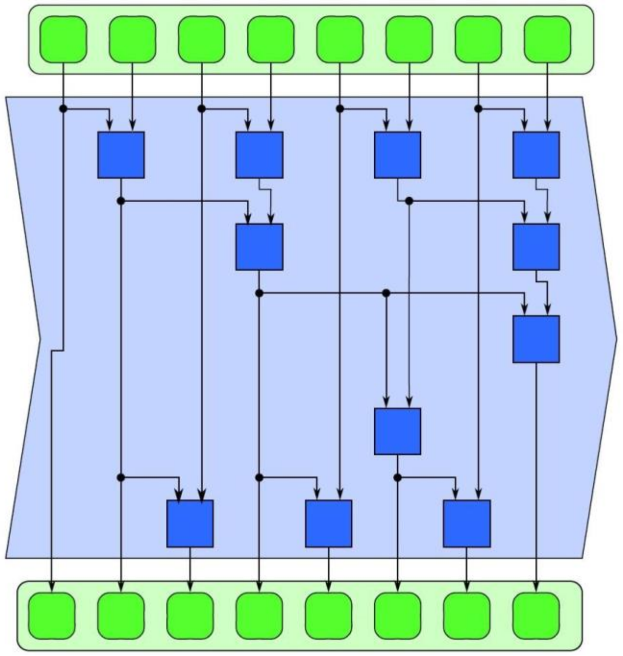




- Dedicated clause available from OpenMP 5.0
- Three-phase approach

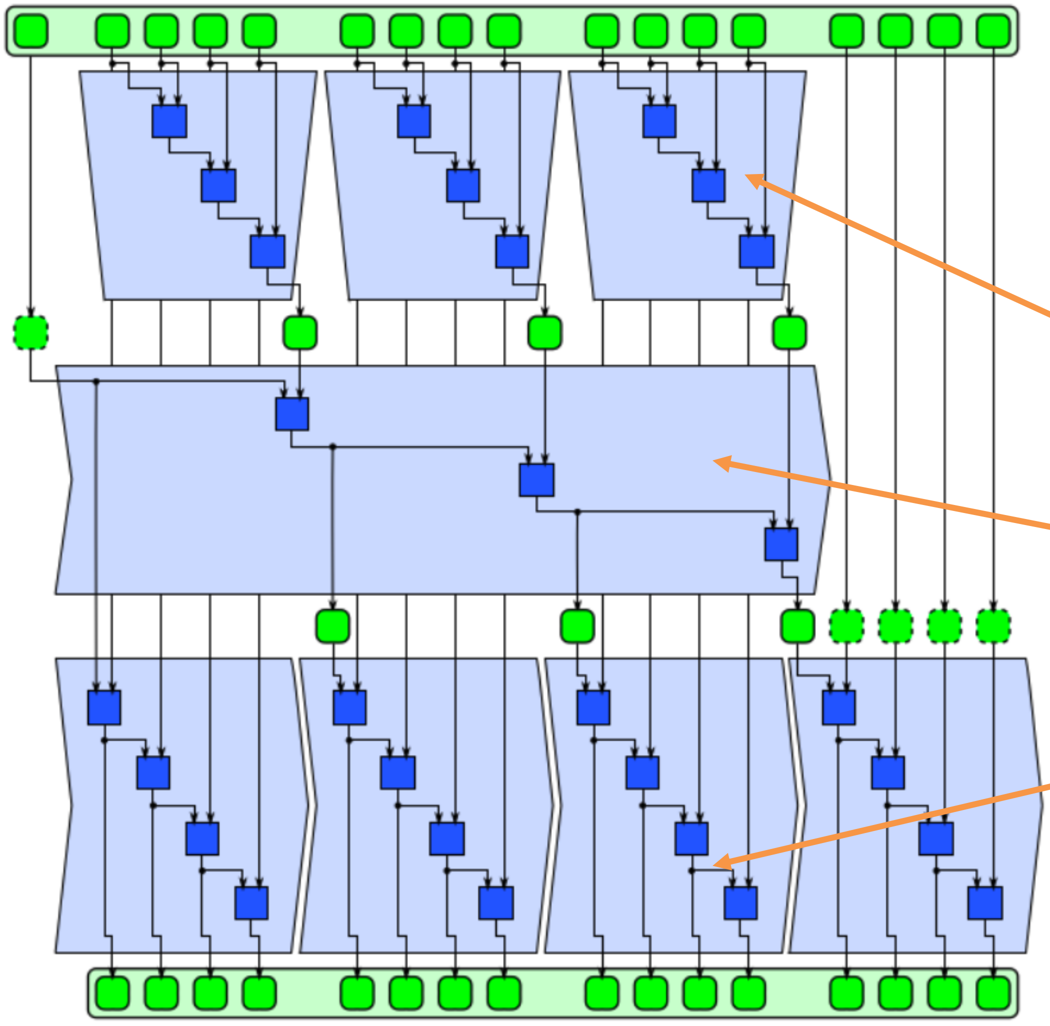
Scan Pattern - OpenMP

- Dedicated clause available from OpenMP 5.0



```
#pragma omp simd reduction(inscan,
+:scan_a)
for(int i = 0; i < N; i++){
 simd_scan[i] = scan_a;
 #pragma omp scan exclusive(scan_a)
 scan_a += a[i];
}
```

Scan Pattern - OpenMP



- Three-phase tiled approach

1 - Build intermediate results in parallel

2 - Sum intermediate results in pairs

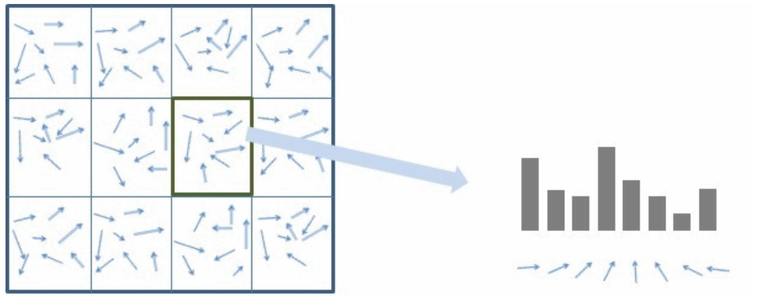
3 - Build output results in parallel

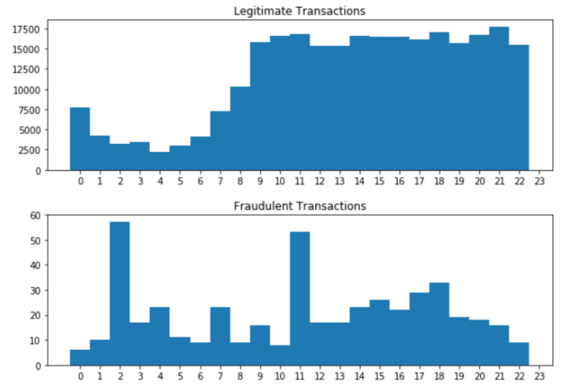
Histogram pattern

- The histogram is an important and useful computation pattern
- Allows us to illustrate the concepts of
  - Partitioning
  - Atomic operations
  - Privatization
- Key issue to solve: output interference

Histogram pattern

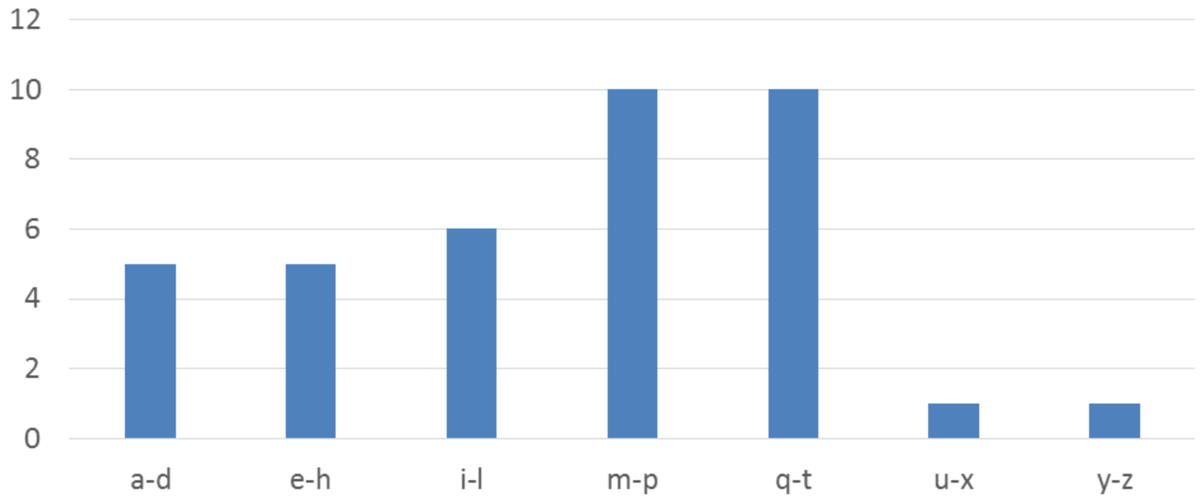
- The histogram is a method for extracting notable features and patterns from large data sets. Examples:
  - Feature extraction in images
  - Fraud detection in credit card transactions
  - Speech recognition
- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment





Histogram pattern

- Histogram that counts the occurrence of letters in a string (grouped in 4-letter bins)
  - For each character in the input string, increment the appropriate bin counter
- Example – histogram for the string "programming massively parallel processors"



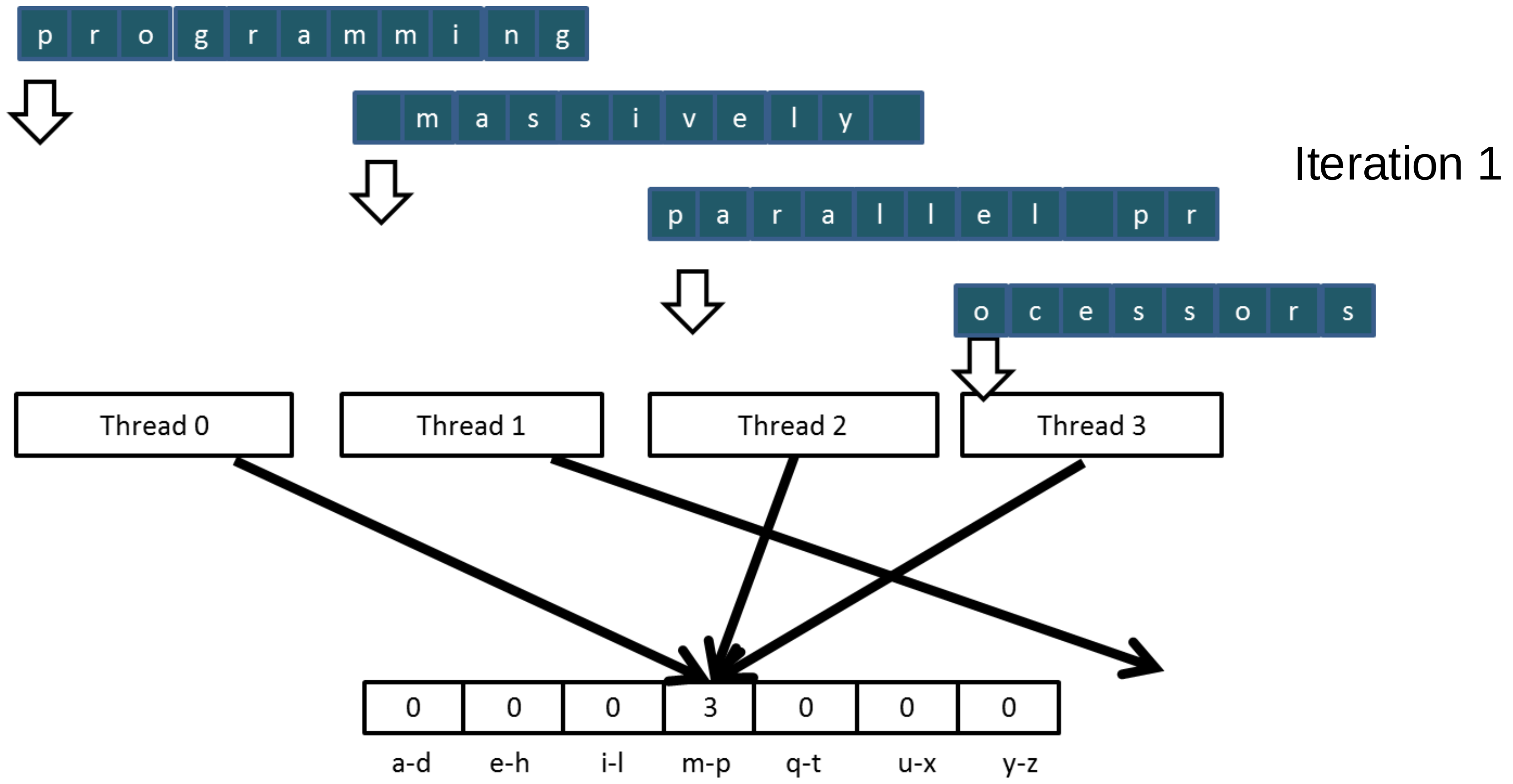
Histogram pattern

- Sequential implementation in C

```
sequential_Histogram(char *data, int length, int *histo)
{

  for (int i = 0; i < length; i++) {
    int alphabet_position = data[i] - ‘a’;
    if (alphabet_position >= 0 && alphabet_position < 26)
    {
      histo[alphabet_position/4]++
    }
  }
}
```

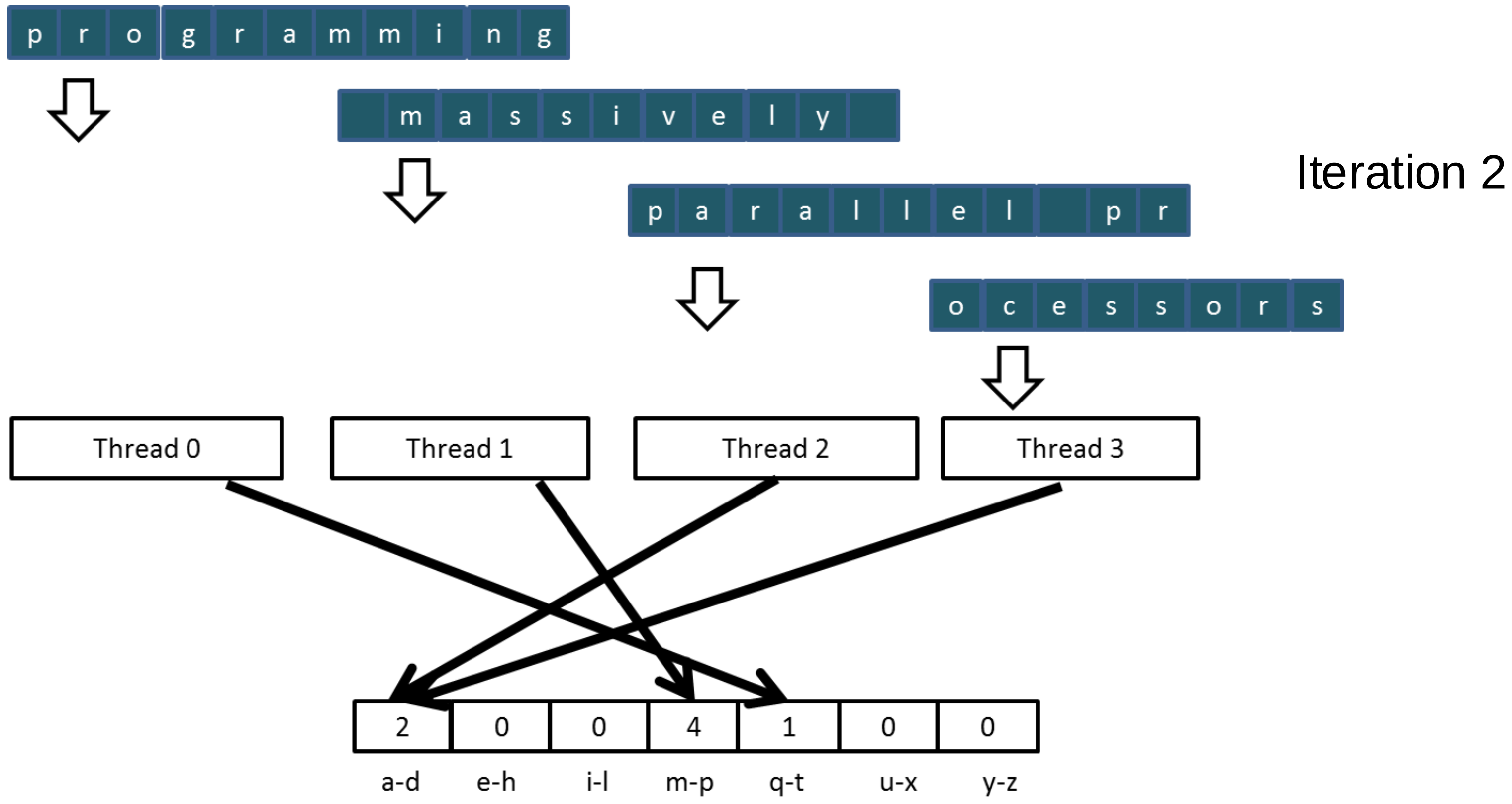
Histogram pattern

- Simplest parallel version:
  - Partition the input into sections
  - Each thread iterates through one section



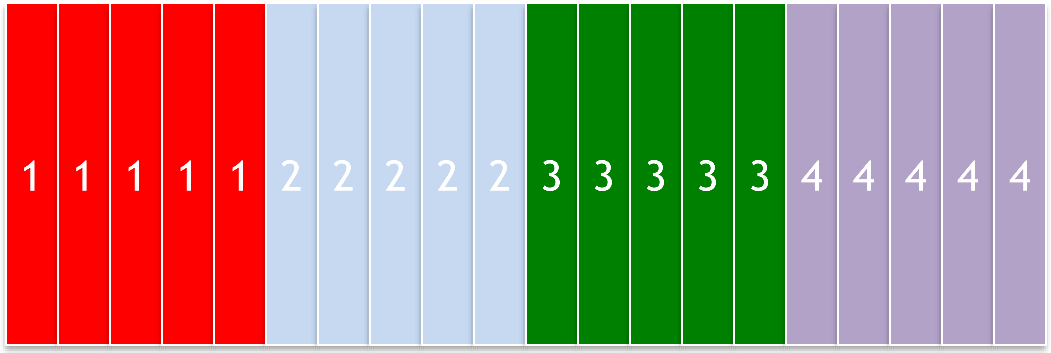
Histogram pattern

- Simplest parallel version:
  - Partition the input into sections
  - Each thread iterates through one section



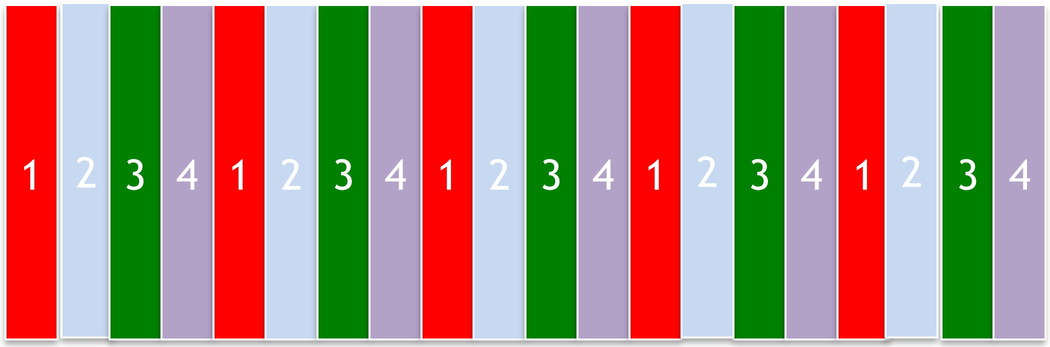
Histogram pattern

- Issue: input partitioning affects memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized



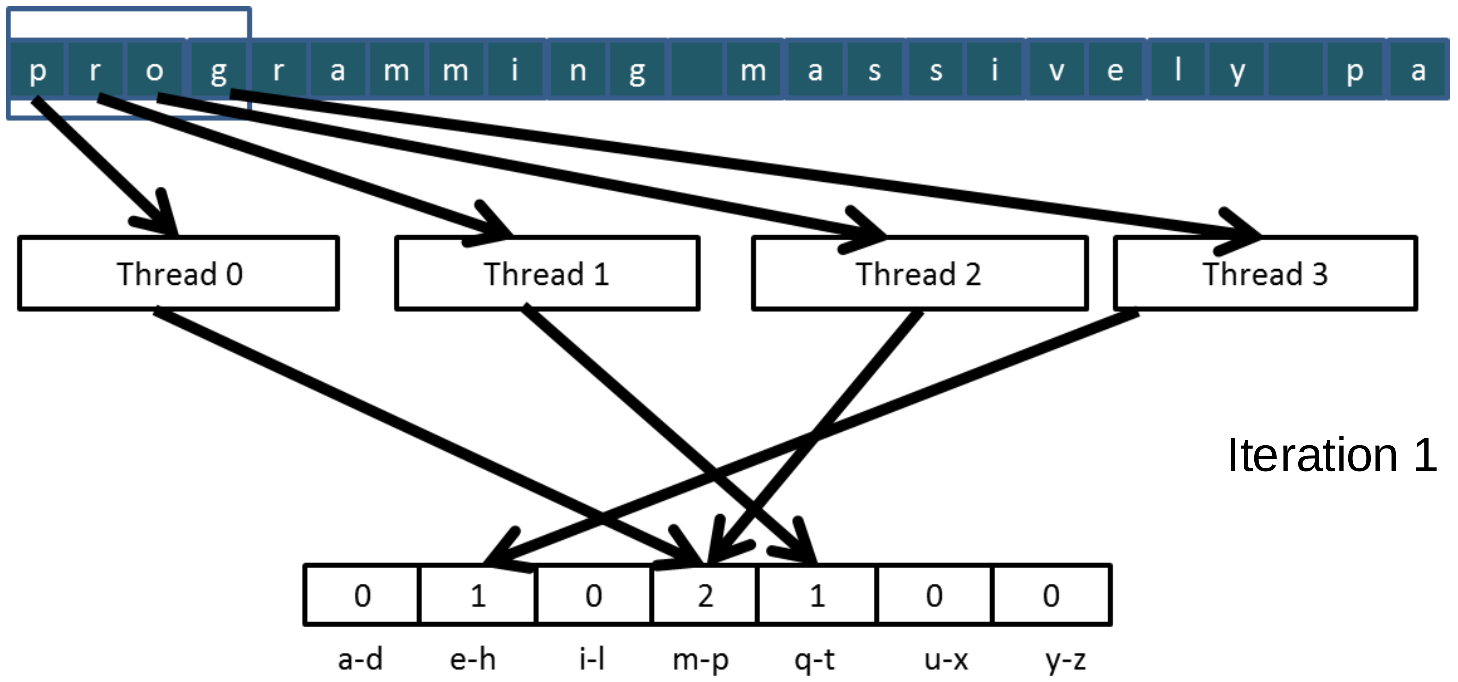
Data partitioning

- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - Memory accesses are coalesced



Data partitioning

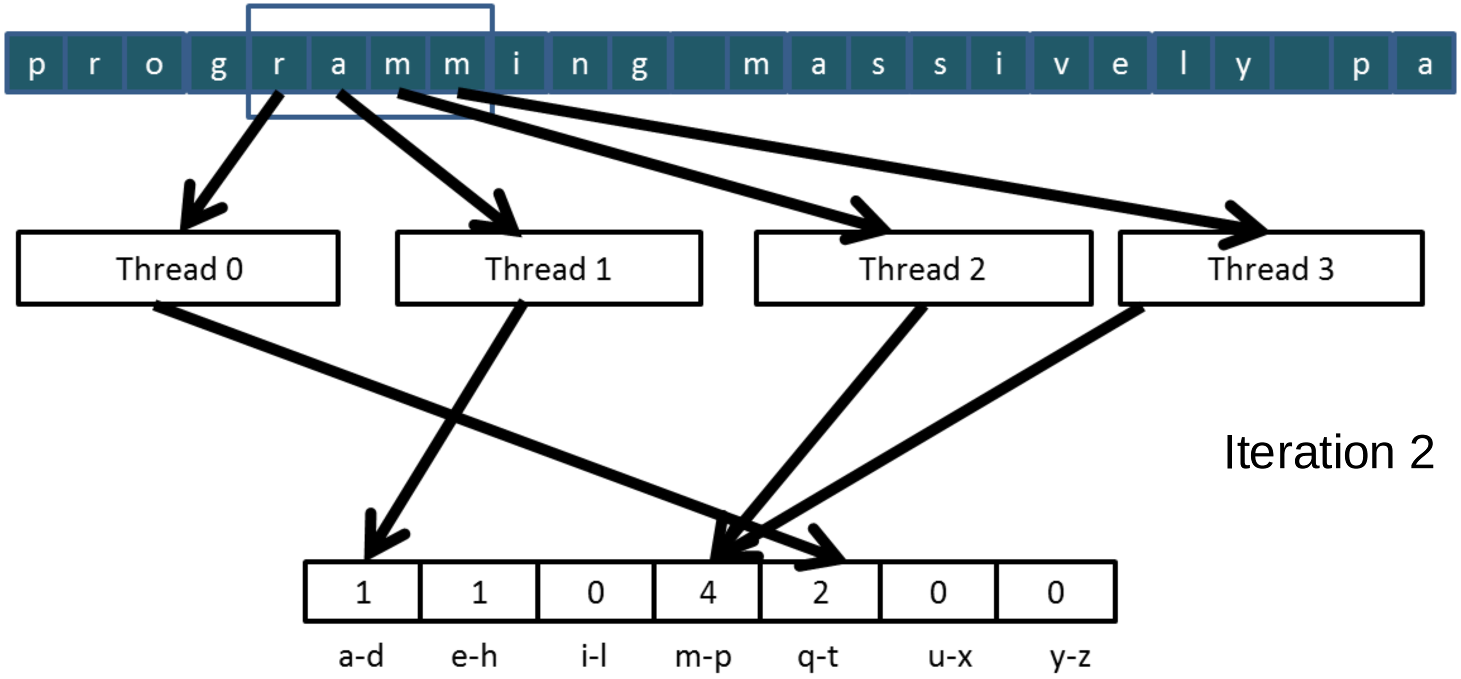
- Parallel histogram with interleaved partitioning



Iteration 1

Data partitioning

- Parallel histogram with interleaved partitioning



Iteration 2

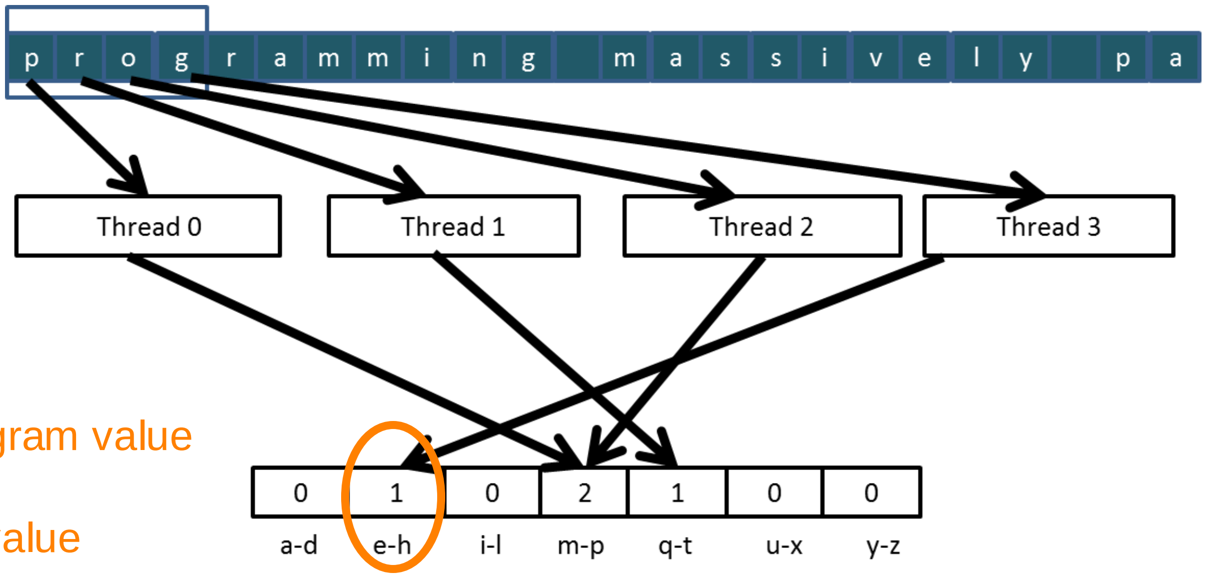
Data partitioning

- Parallel histogram in CUDA with interleaved partitioning

```
__global__ void histo_kernel(unsigned char *buffer, long
size, unsigned int *histo)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x*gridDim.x;

  for (unsigned int i = tid; i < size; i += stride ) {
    int alphabet_position = buffer[i] – ‘a’;
    if (alphabet_position >= 0 && alpha_position < 26)
      histo[alphabet_position/4] += 1;
  }
}
```

Histogram pattern

- Issue: data races
  - Data races occur in read-modify-write operations
  - Data races cause errors that are hard to reproduce
  - Histograms (and other collaboration patterns) include read-modify-write operations


p r o g r a m m i n g m a s s i v e l y p a
Thread 0
Thread 1
Thread 2
Thread 3
0 1 0 2 1 0 0
a-d e-h i-l m-p q-t u-x y-z

Read histogram value
Sum 1
Write new value

Data races

thread1:

```
Old = Mem[x]
New = Old + 1
Mem[x] = New
```

Read
Modify
Write

thread2:

```
Old = Mem[x]
New = Old + 1
Mem[x] = New
```

- Data race:
  - If Mem[x] was initially 0, what is its value when threads 1 and 2 have completed?
  - What does each thread get in its Old variable?
  - The answers depend on the relative execution order!

Data races

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

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

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

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

Wrong result

Correct result

Atomic operations

- Atomic operations perform read-modify-write as a single hardware instruction on a memory address
  - The hardware ensures that no other thread can perform another read-modify-write operation on the same location until the current atomic operation is complete
  - Any other threads that attempt to perform an atomic operation on the same location will be held in a queue
  - All threads perform their atomic operations serially on the same location

Atomic operations

- Atomic operations in CUDA are performed by calling functions that are translated into single instructions (*intrinsic functions* or intrinsics)
  - Available intrinsics for atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
- Example: atomic add
  - `int atomicAdd(int* address, int val);`
  - reads a 32-bit word from `address`, computes (old value + `val`), and stores the result back to memory at the same address
  - the three operations are performed in one atomic transaction

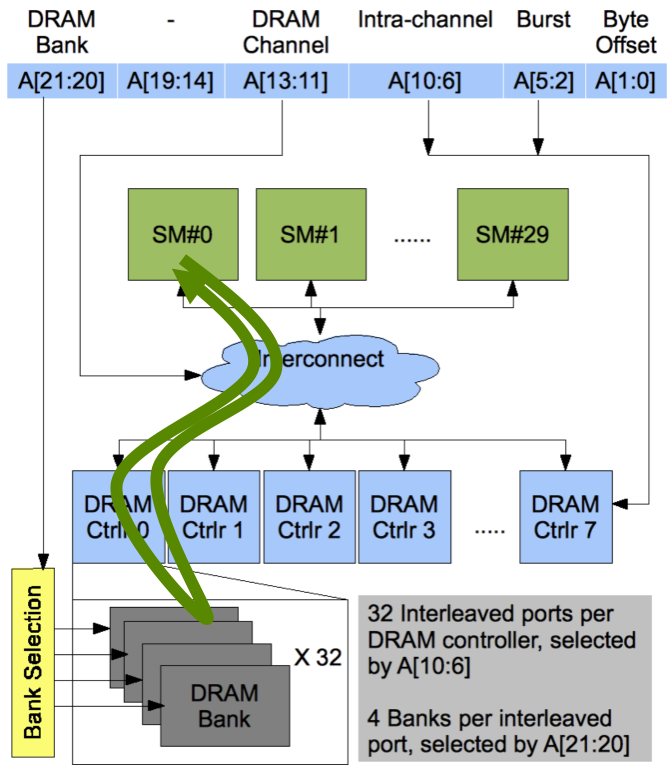
Atomic operations

- Parallel histogram in CUDA with atomic operations

```
__global__ void histo_kernel(unsigned char *buffer,
                long size, unsigned int *histo)
{
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;

   for (unsigned int i = tid; i < size; i += stride ) {
     int alphabet_position = buffer[i] – "a";
     if (alphabet_position >= 0 && alpha_position < 26)
       atomicAdd(&(histo[alphabet_position/4]), 1);
   }
}
```

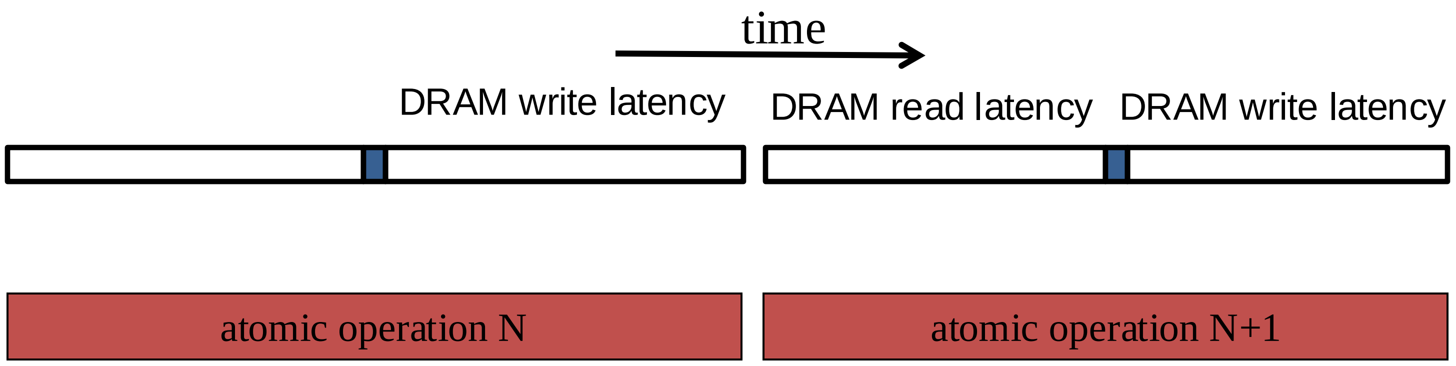
Atomic operations

- Performance considerations:
  - Atomic operations can access different types of memory (global, shared, caches)
  - Atomic operations impact latency and throughput
  - For example, a DRAM access has a latency of a few hundred cycles



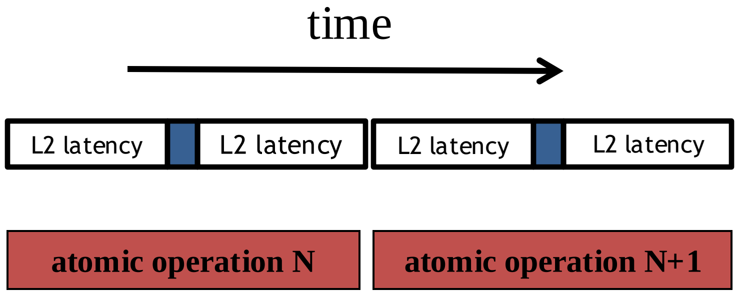
Atomic operations

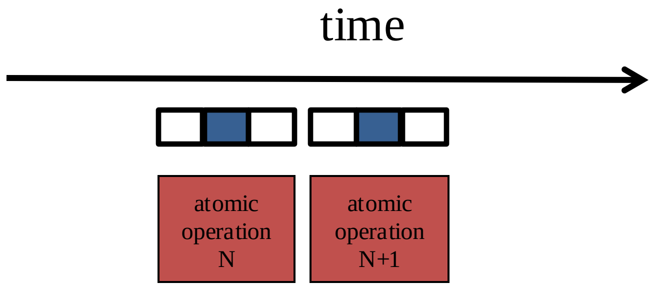
- The total latency for an atomic operation on global memory (DRAM) is typically more than 1000 cycles
- If many threads attempt to do an atomic operation on the same location, the throughput is reduced to 1/1000 of the peak memory throughput
  - Throughput = rate at which the application can execute an atomic operation



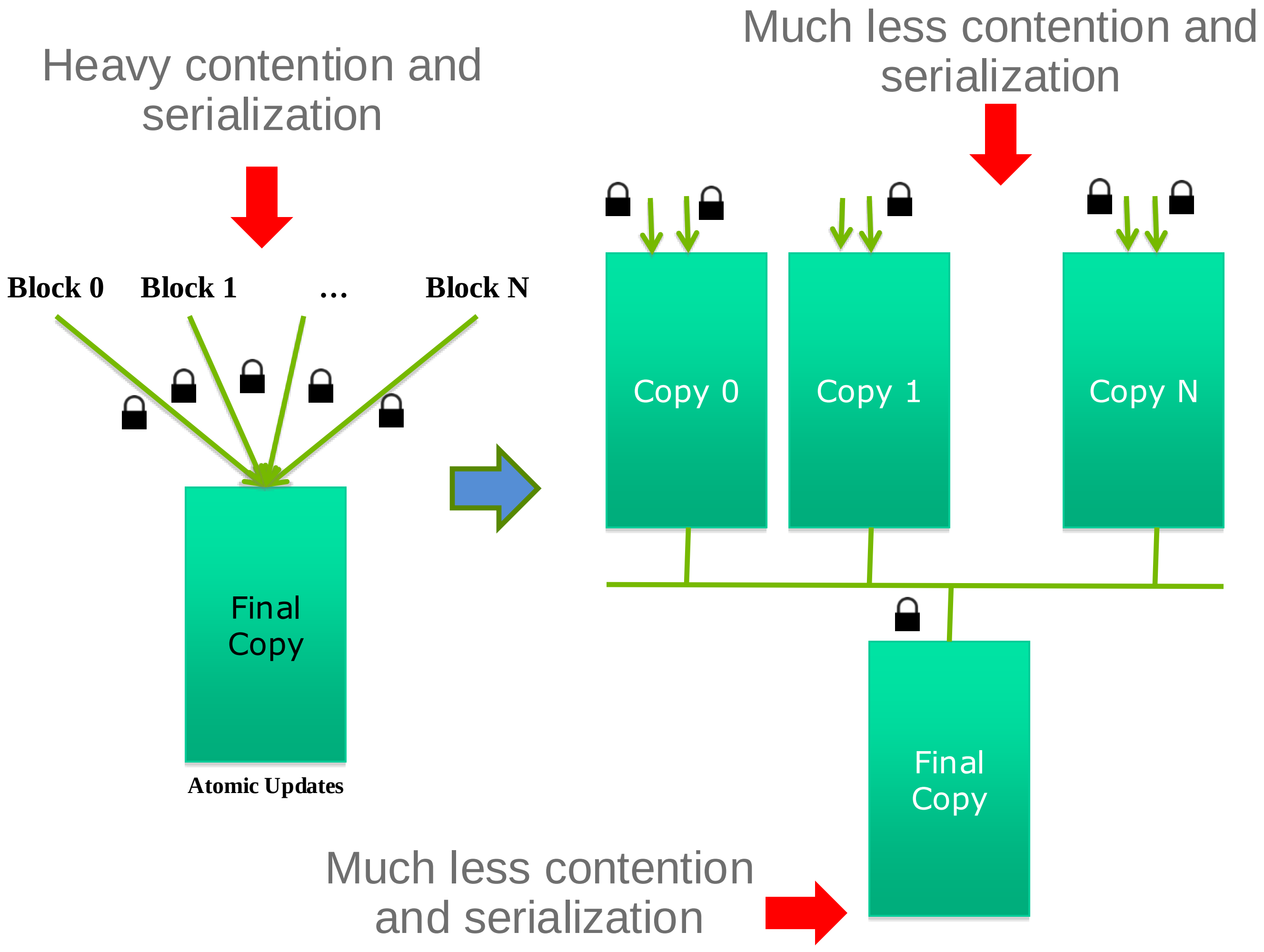
Atomic operations

- Atomic operations on L2 cache
  - About 1/10 of the DRAM latency
  - Shared among all blocks
- Atomic operations on shared memory
  - Very short latency
  - Private to each thread block





Privatization


Heavy contention and serialization
Block 0
Block 1
…
Block N
Final Copy
Atomic Updates
Much less contention and serialization
Copy 0
Copy 1
Copy N
Final Copy
Much less contention and serialization

Privatization

- Cost
  - Overhead for creating and initializing private copies
  - Overhead for accumulating the contents of private copies into the final shared copy
- Benefits
  - Much less contention and serialization in accessing both the private copies and the final copy
  - The overall performance can often be improved more than 10x

Privatization

- Privatization is a powerful and frequently used technique for parallelizing applications
- The operation needs to be associative and commutative
  - Histogram add operation is associative and commutative
- The private data must fit into shared memory
  - Works for small histograms
  - It is possible to partially privatize the histogram and only go to the copy in global memory when needed

Privatization

- The histogram pattern is an important use case for privatization
  - All threads working on an input section are in the same block
  - Threads in the same block can access shared memory
  - Throughput for atomics on shared memory is 10-100x better than on DRAM or L2 cache

Privatization

- Parallel histogram in CUDA with privatization (part 1)

```
__global__ void histogram_privatized_kernel(unsigned char*
input, unsigned int* bins, unsigned int num_elements,
unsigned int num_bins) {

  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
  extern __shared__ unsigned int histo_s[];

  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins;
binIdx +=blockDim.x) {
    histo_s[binIdx] = 0u;
  }

  __syncthreads();
```

Privatization

- Parallel histogram in CUDA with privatization (part 2)

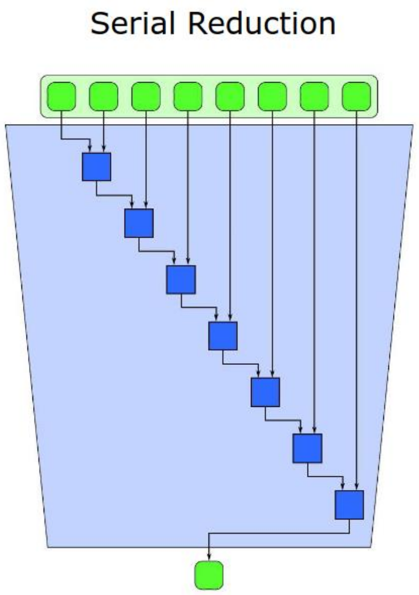
```
   for (unsigned int i = tid; i < num_elements; i +=
 blockDim.x*gridDim.x) {

     int alphabet_position = buffer[i] – “a”;
     if (alphabet_position >= 0 && alpha_position < 26)
       atomicAdd(&(histo_s[alphabet_position/4]), 1);
   }
   __synchthreads();

   for(unsigned int binIdx = threadIdx.x; binIdx <
 num_bins; binIdx += blockDim.x) {
     atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
   }
 }
```

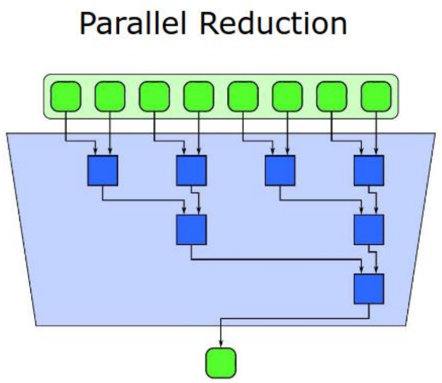
Reduction pattern

- The reduction is another typical computation pattern that can be parallelized
- We can use it to analyze
  - Work efficiency
  - Resource efficiency
- Key issue to solve: control divergence

Reduction pattern

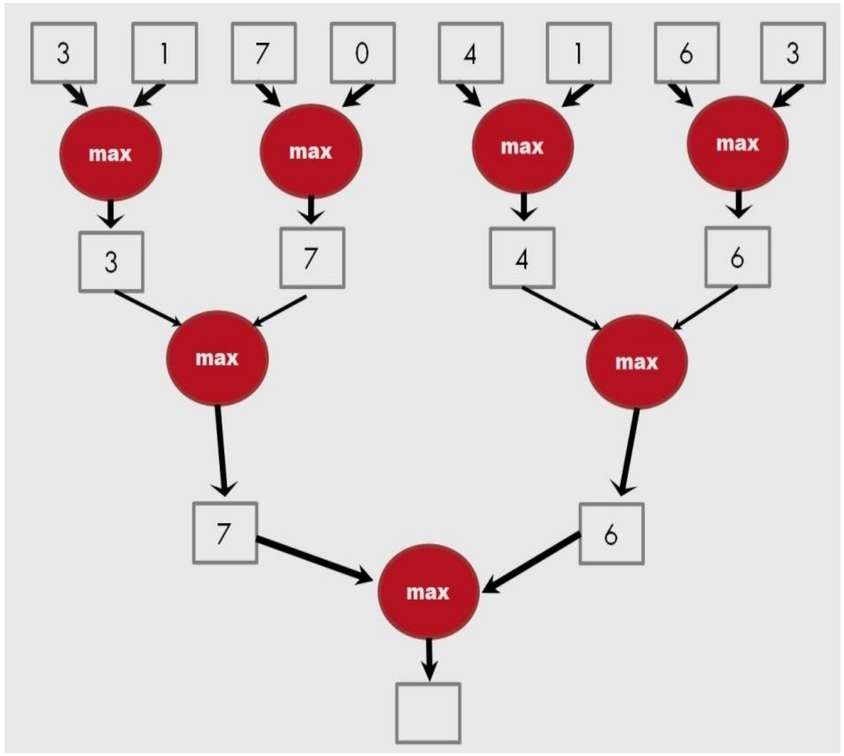
- Reduction is the final step in “partition and summarize” used to process large data sets
- Needs an associative (and commutative) combiner function
- Sequential reduction has O(N) complexity





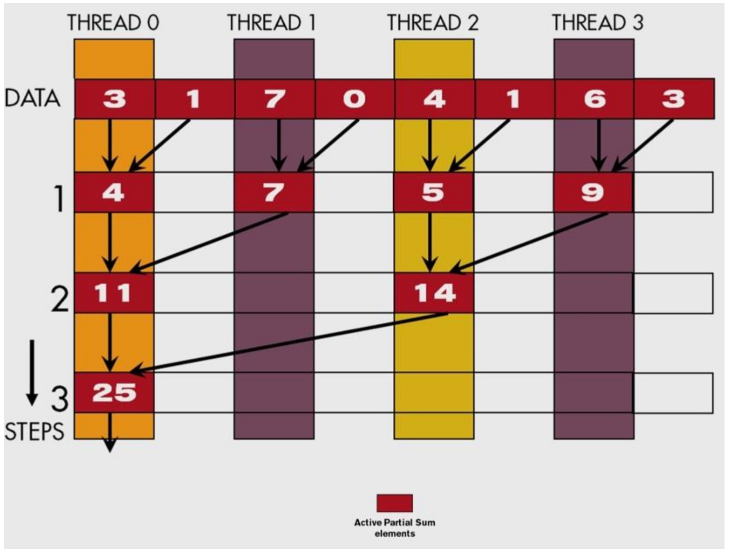
Reduction pattern

- Parallel reduction tree
  - Assuming there are enough resources, it can perform N-1 operations in $\log_2(N)$ steps
  - Average Parallelism $(N-1)/\log_2(N)$
  - Work efficient, but not resource efficient!



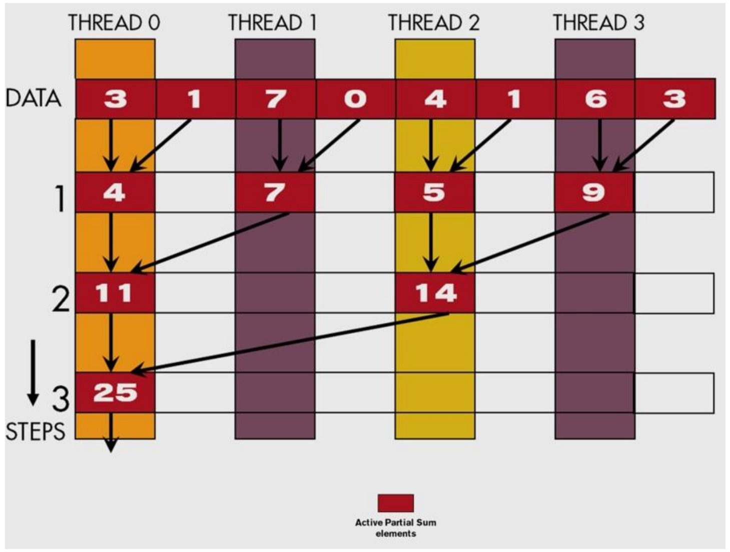
Reduction pattern

- Simple parallel implementation
  - N/2 threads
  - $\log_2(N)$ steps
  - At each step each thread operates on 2 elements, then the number of threads is halved
- In-place reduction in shared memory
  - Reduced global memory traffic
- N has to be lower than the maximum number of threads in a block



Reduction pattern

- Thread to data mapping:
  - Each thread writes to an even-index location in the partial sum
  - One of the inputs comes from the same index
  - The second input is increasingly far away



Parallel reduction

- Baseline CUDA implementation

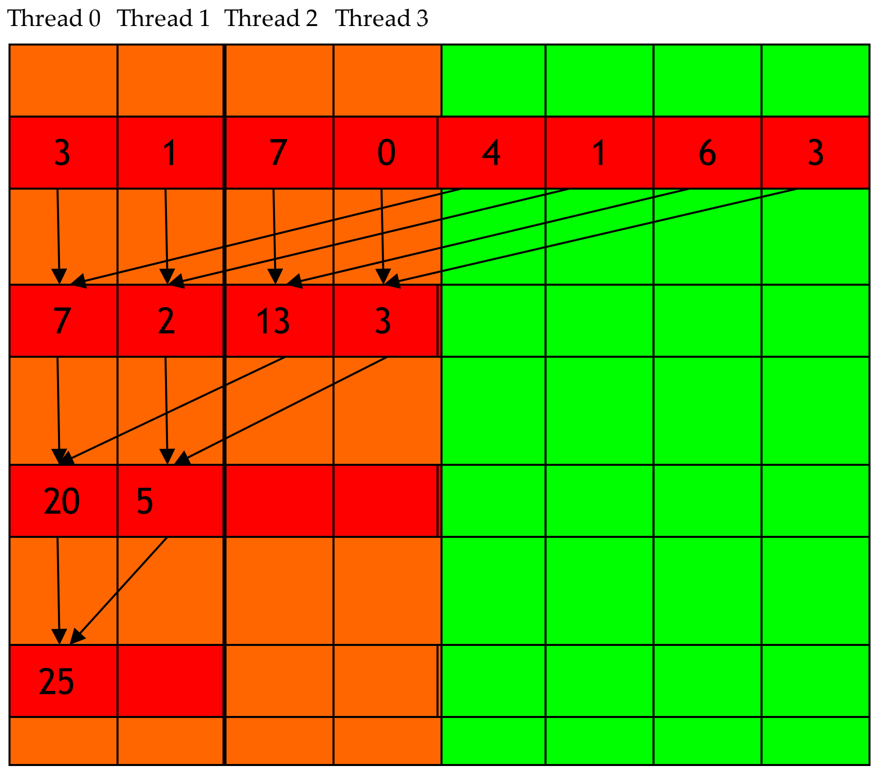
```
__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] =
X[blockIdx.x*blockDim.x+threadIdx.x];

unsigned int t = threadIdx.x;

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

Baseline implementation

- 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
- Half or fewer of threads will be executing after the first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence

Better implementation

- Change index usage to reduce divergence
  - Possible because reduction operators are commutative and associative
- Partial sums are stored in the front of the array



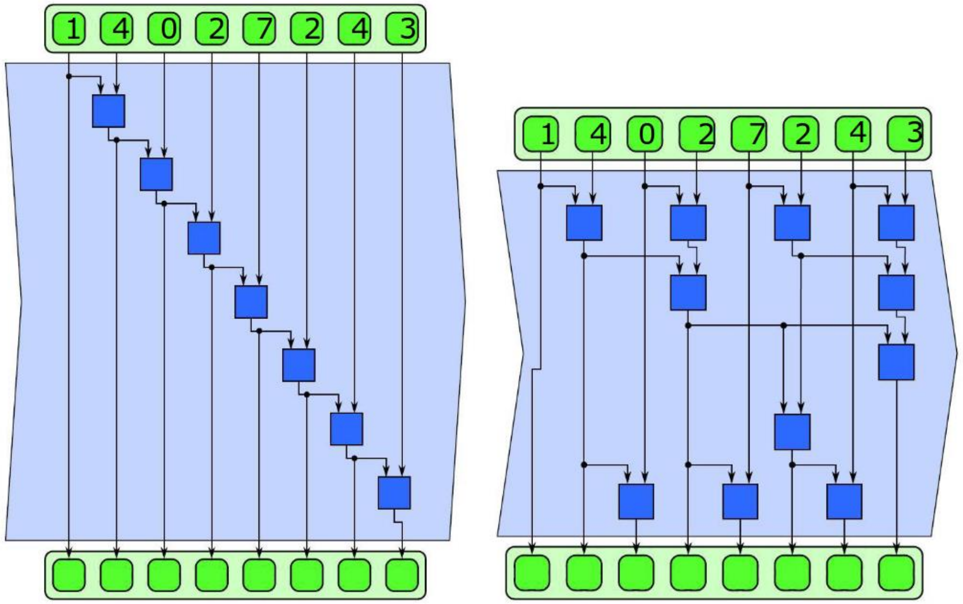
Parallel reduction

- Improved CUDA implementation

```
__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] =
X[blockIdx.x*blockDim.x+threadIdx.x];

unsigned int t = threadIdx.x;

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

Scan pattern - CUDA

- Scan is frequently used to parallelize sequential recursive algorithms
- Sequential scan is extremely efficient, parallel scan can be slower in some cases
- Key issue to solve: work efficiency



Scan pattern

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
  y[i] = y [i-1] + x[i];
```

- Sequential C implementation of an inclusive addition scan
  - N additions for N elements, O(N) complexity
  - Only 1 operation more than sequential reduction
  - Computationally efficient!

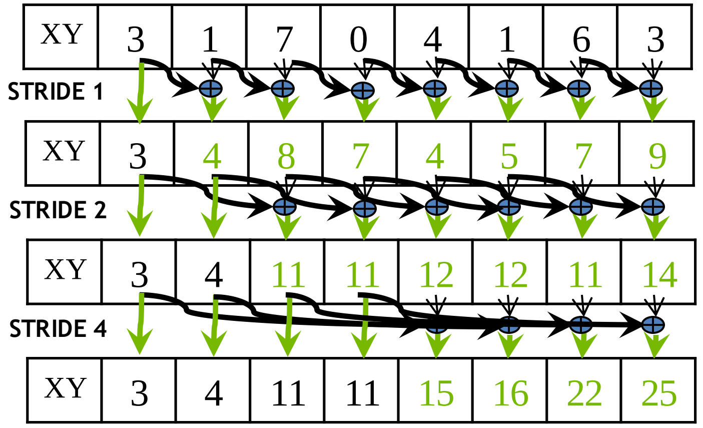
Scan pattern

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

- Naive parallel implementation of an inclusive addition scan
  - Threads can work independently on y elements
  - A lot of redundant computation!
  - The last thread is performing a sequential reduction, so execution time is the same or worse as the sequential scan

Scan pattern

- Parallel implementation with interleaved reduction trees
  - Read input into shared memory
  - Thread *j* adds elements *j* and *j-stride* and writes result into element *j*
  - Double stride and repeat



ITERATION = 1
STRIDE = 1

ITERATION = 2
STRIDE = 2

ITERATION = 3
STRIDE = 4

Parallel scan

- Work-inefficient CUDA implementation

```
__shared__ float XY[SECTION_SIZE];
int i = blockIdx.x * blockDim.x + threadIdx.x;

if (i < InputSize) {
  XY[threadIdx.x] = X[i]; }

  // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride < blockDim.x;
stride *= 2) {
    __syncthreads();
    if (stride <= threadIdx.x) {
      XY[threadIdx.x] += XY[threadIdx.x - stride]; }
  }
```

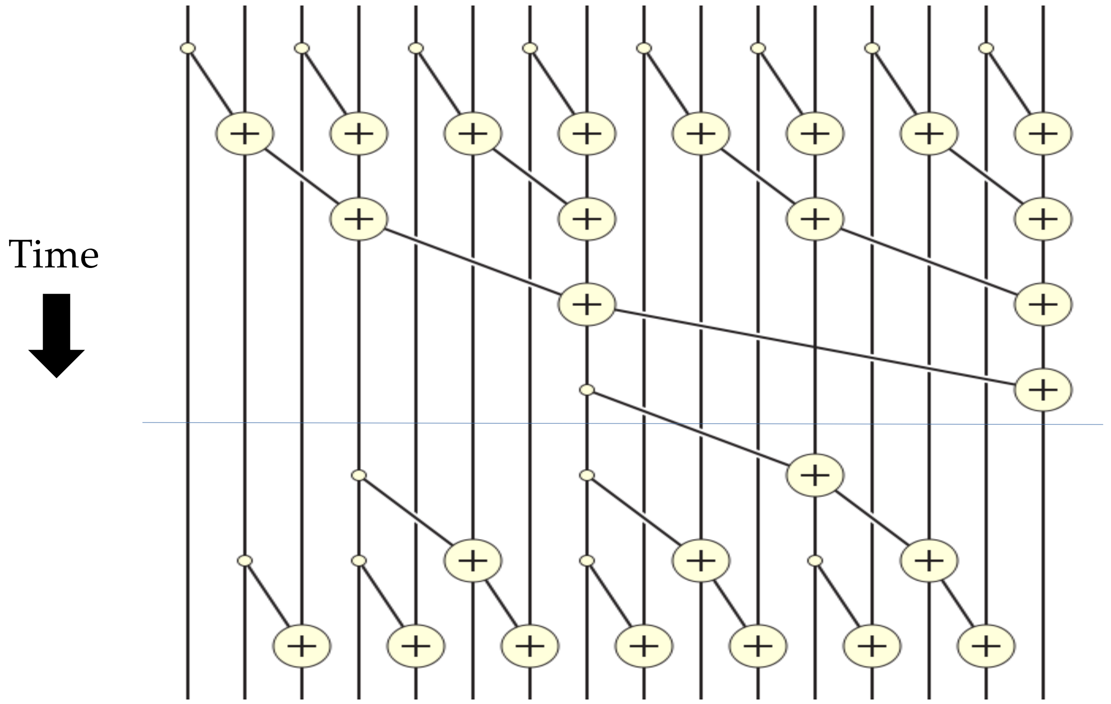
Parallel scan

- The parallel scan executes $\log_2(N)$ iterations
- The iterations do (N-1), (N-2), (N-4), … (N-N/2) additions each
  - Total number of additions: $O(N*\log_2(N))$
- Compare to O(N) for the sequential implementation!
  - 10x overhead with N=1024
- Work inefficient and consumes extra energy due to inactive hardware resources

Parallel scan

- Improving efficiency with:
  - Two-phased approach
  - Reuse of intermediate results
  - More complex thread index-data index mapping
- The two-phased approach is based on balanced trees
  - Not an actual data structure, just to describe what each thread does at each step
  - Create partial sums (leaves to root)
  - Build output values (root to leaves)

Parallel scan

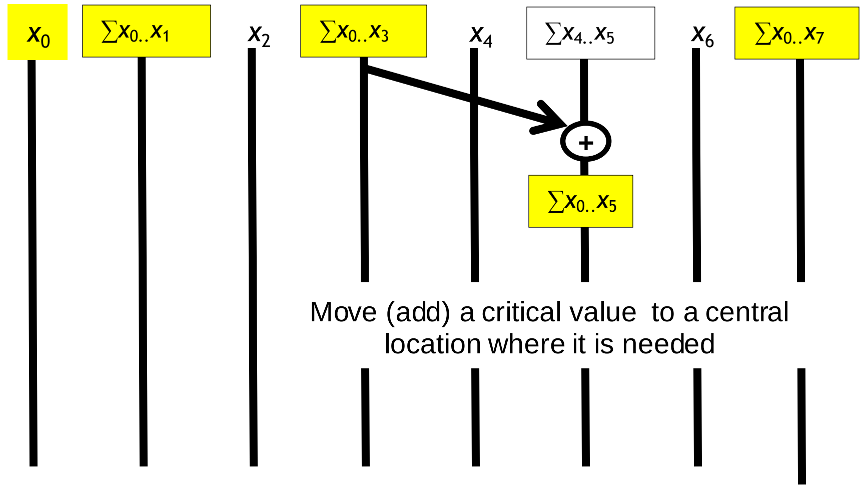


Reduction phase

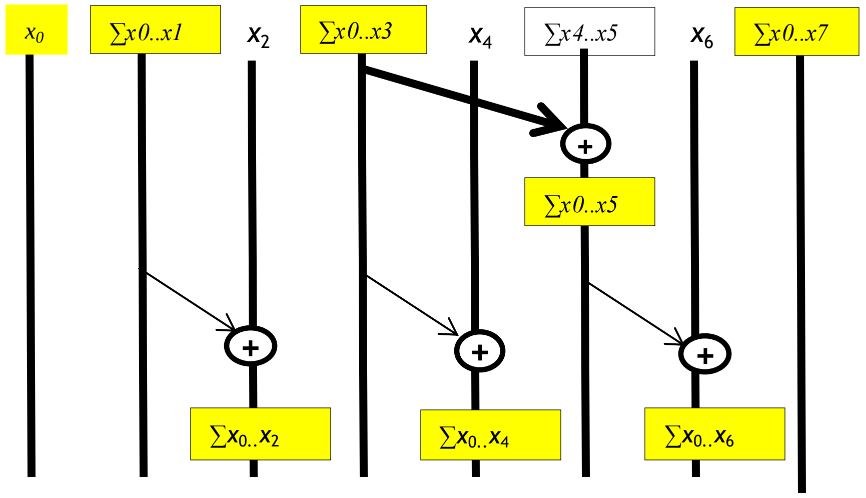
Post-reduction reverse phase

Parallel scan

Post-reduction reverse phase



Move (add) a critical value to a central location where it is needed



Parallel scan

- Reduction phase

```
// XY[2*BLOCK_SIZE] is in shared memory

for (unsigned int stride = 1; stride <= BLOCK_SIZE;
stride *= 2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    __synchthreads();
}
```

Parallel scan

- Distribution phase

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0;
stride /= 2) {
   __syncthreads();
   int index = (threadIdx.x+1)*stride*2 - 1;
   if(index+stride < 2*BLOCK_SIZE) {
     XY[index + stride] += XY[index];
   }
}
__syncthreads();
if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Parallel scan

- The work efficient version executes $\log_2(N)$ iterations in the reduction step
  - Total number of operations: N-1
- In the distribution phase the iterations do (2-1), (4-1), …. (N/2-1) operations
  - Total: $(N-2) - (\log_2(N)-1)$
- Both phases are O(N) and they do not perform more than 2(N-1) operations
- With enough hardware resources, the 2x extra work is compensated by the reduced execution time

Credits & References

- NVIDIA Deep Learning Institute, “Accelerated Computing” Teaching Kit
- D.B. Kirk and W.W. Hwu, “Programming Massively Parallel Processors. A hands-on approach”, 3rd edition, Morgan Kaufmann 2017
- Mark Harris, Parallel Prefix Sum with CUDA https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html