CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA
Advanced Techniques 3

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

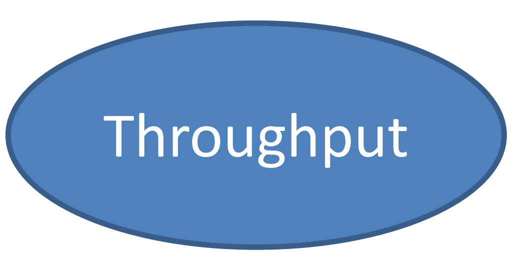
Some slides are used and slightly modified from:
NVIDIA teaching kit

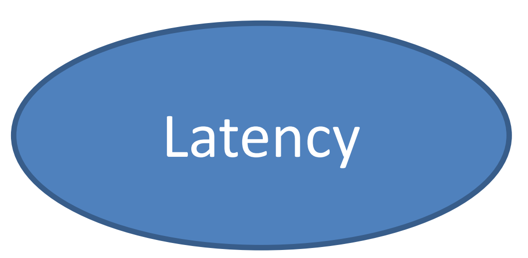
In This Lecture …

- More about performance
- Parallel Patterns
- Error Handling

More About Performance

Hardware configuration can be safely ignored when designing a software for correctness but must be considered in the code structure when designing for peak performance.





Some Insights About Performance

Occupancy

It is a common belief that …

- More threads is better
  - because it needs more threads to hide latency

Computational　　memory

But is it always true?

CUDA Basic Linear Algebra Subroutines

Multiplication of two large matrices, single precision (SGEMM):

	CUBLAS 1.1	CUBLAS 2.0	
Threads per block	512	64	8x smaller thread blocks
Occupancy (G80)	67%	33%	2x lower occupancy
Performance (G80)	128 Gflop/s	204 Gflop/s	1.6x higher performance

Batch of 1024-point complex-to-complex FFTs, single precision:

	CUFFT 2.2	CUFFT 2.3	
Threads per block	256	64	4x smaller thread blocks
Occupancy (G80)	33%	17%	2x lower occupancy
Performance (G80)	45 Gflop/s	93 Gflop/s	2x higher performance

Latency Vs Throughput

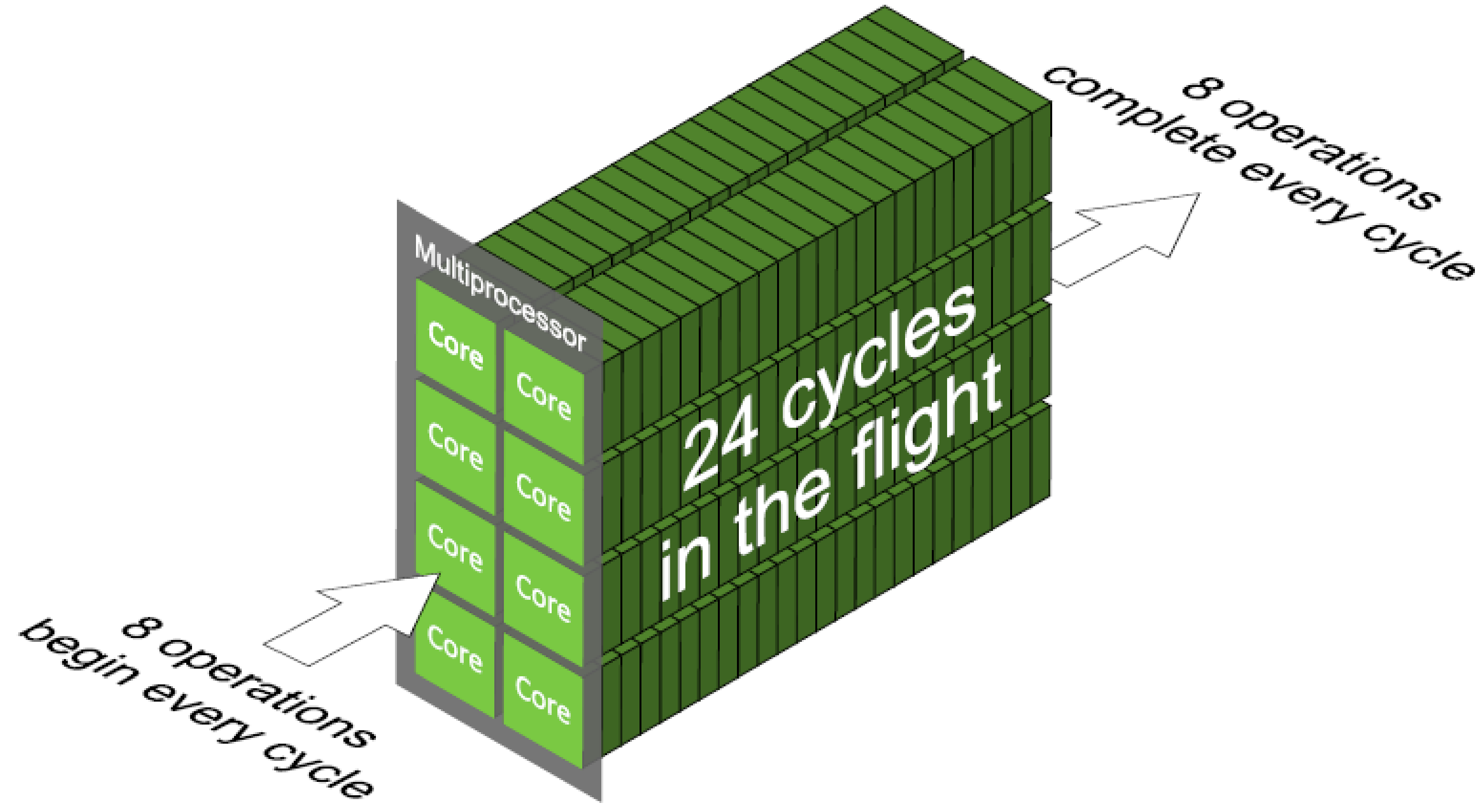
- Latency (how much time) is time
  - instruction takes 4 cycles per warp
  - memory takes 400 cycles
- Throughput (how many operations per cycle or second) is rate
  - Arithmetic: 1.3 Tflop/s = 480 ops/cycle (op=multiply-add)
  - Memory: 177 GB/s ≈ 32 ops/cycle (op=32-bit load)

Hide Latency is …

- Doing other operations while waiting
- This will make the kernel runs faster
- But not at the peak performance

What can we do??

Little's Law



Needed parallelism = **Latency** x **Throughput**

Examples from GPU

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (operations/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864

Average latency of a computational operation

Less operations means idle cycle

So …

- Higher performance does not mean more threads but higher utilization
- Utilization is related to parallelism
- We can increase utilization by
  - increasing throughput
    - Instruction level parallelism
    - Thread level parallelism
  - decreasing latency

Occupancy is not utilization, but one of the contributing factors.

Occupancy Calculator API

```
__global__ void MyKernel(int *d, int *a, int *b) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d[idx] = a[idx] * b[idx]; }

int main() {
    int numBlocks;
    int blockSize = 32;
    int device;
    cudaDeviceProp prop;
    int activeWarps;
    int maxWarps;

    cudaGetDevice(&device);
    cudaGetDeviceProperties(&prop, device);

    cudaOccupancyMaxActiveBlocksPerMultiprocessor(
            &numBlocks,
            MyKernel,
            blockSize,
            0);

    activeWarps = numBlocks * blockSize / prop.warpSize;
    maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;
}
```

cudaOccupancyMaxActiveBlocksPerMultiprocessor

- From CUDA 6.5
- Produces an occupancy prediction based on:
  - the block size
  - shared memory usage of a kernel
- Reports occupancy in terms of the number of concurrent thread blocks per multiprocessor
- Don't forget: it is just a prediction!
- Arguments:
  1. pointer to an integer (where #blocks will be reported)
  2. kernel
  3. block size
  4. dynamic shared memory per block in bytes

How about memory?

maximizing overall memory throughput for the application

=

minimize data transfers with low bandwidth

host ←→ device

Global mem access

This means …Typically

1. Load data from device memory to shared memory.
2. Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads.
3. Process the data in shared memory.
4. Synchronize again if necessary to make sure that shared memory has been updated with the results.
5. Write the results back to device memory.

But accessing global memory is a necessary evil … So:

- Can we apply the same technique (i.e. Little's law) to memory?

Needed parallelism = **Latency** x **Throughput**

	Latency	Throughput	Parallelism
Arithmetic	≈18 cycles	32 ops/SM/cycle	576 ops/SM
Memory	< 800 cycles (?)	< 177 GB/s	< 100 KB

This means that to hide memory latency you need to keep 100KB in flight.
But less if the kernel is compute bound!

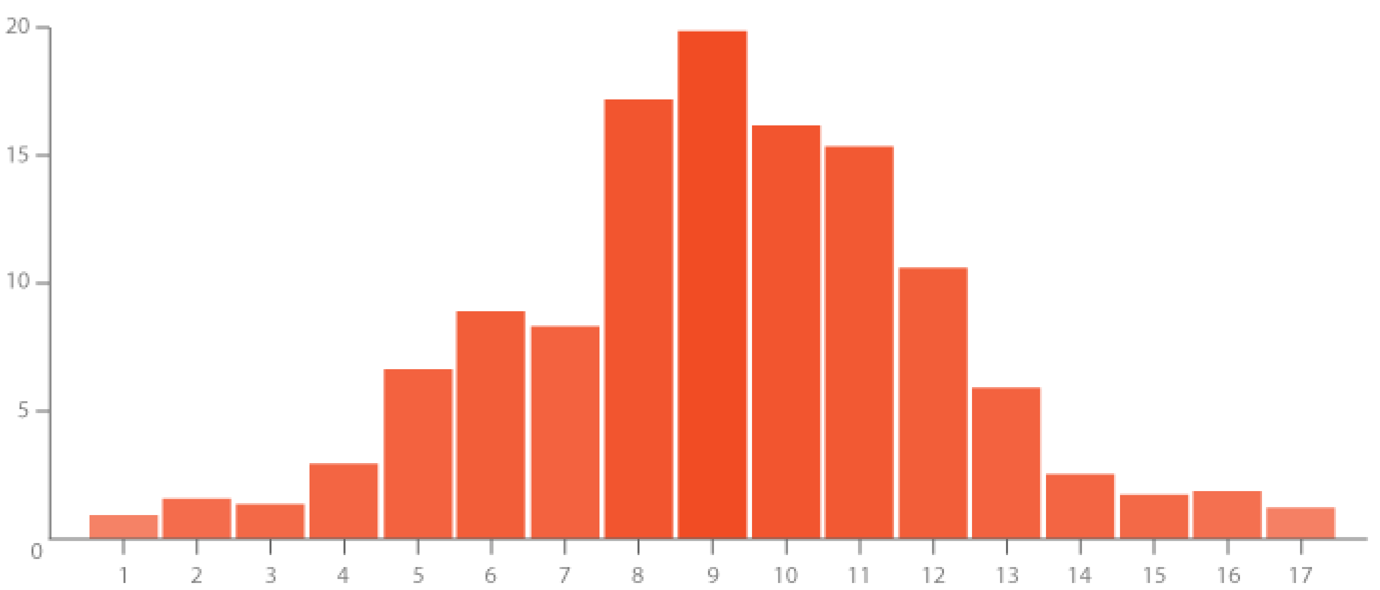
How Can You Get 100KB From Threads?

- Use more threads
- Use more instructions per thread
- Use more data per thread

Now for some commonly used parallel patterns

- Histogram
- Convolution
- Reduction tree
- Prefix sum

Histogram

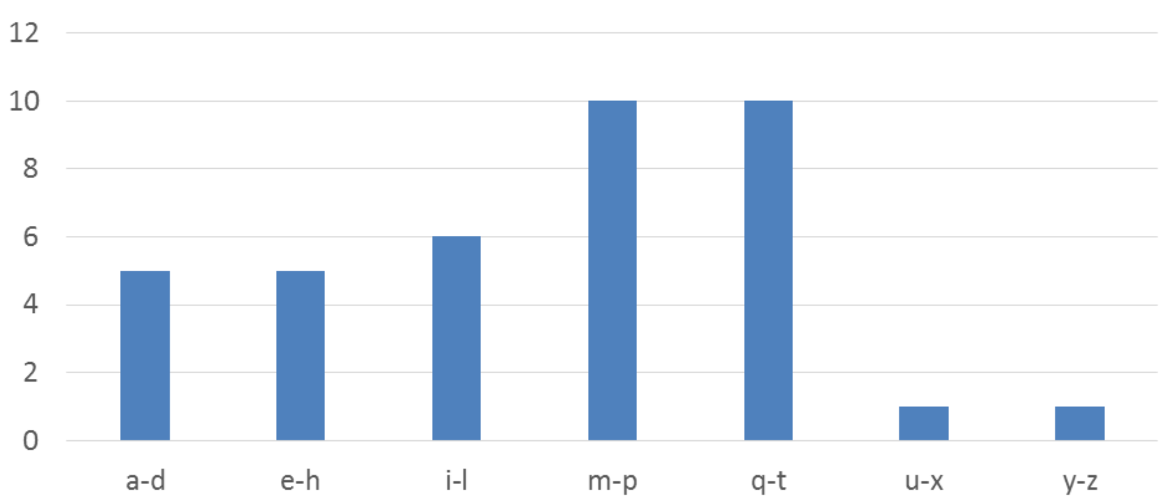


In a Nutshell

- Important and very useful computation:
  - For each element in the data set, use the value to identify a "bin counter" to increment.
- A good example for understanding output interference in parallel computation

Example

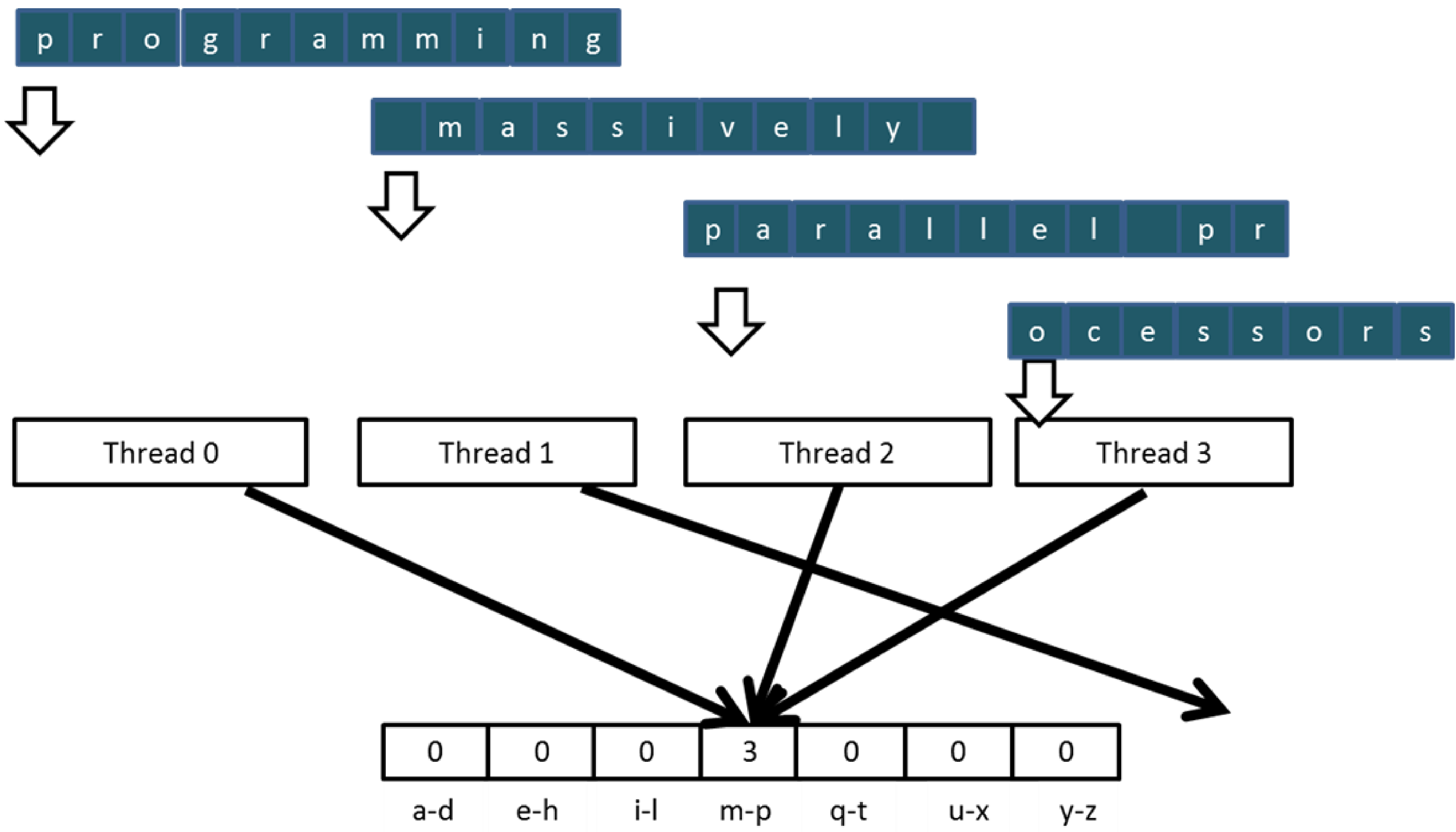
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, …
- For each character in an input string, increment the appropriate bin counter.
- In the phrase "Programming Massively Parallel Processors" the output histogram is shown below:



Implementation 1

- Partition the input into sections
- Have each thread to take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter

Implementation 1

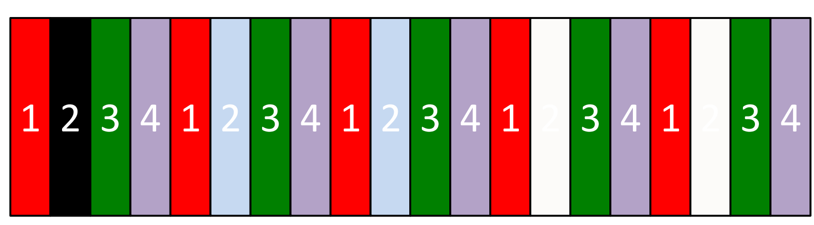


Evaluation of Implementation 1

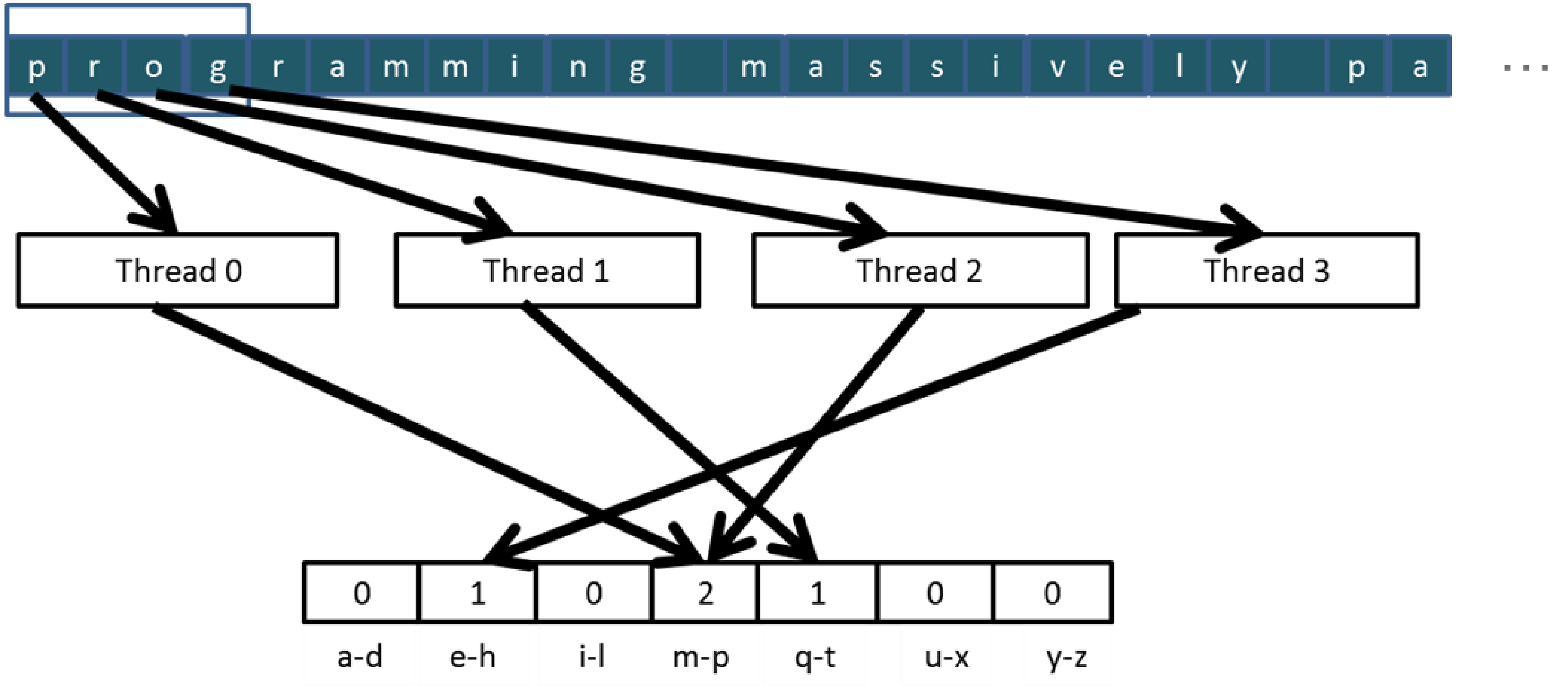
- Possible collision
- Poor memory access efficiency:
  – Adjacent threads do not access adjacent memory locations
  – Accesses are not coalesced
  – DRAM bandwidth is poorly utilized

Implementation 2

- Interleaved partitioning:
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat

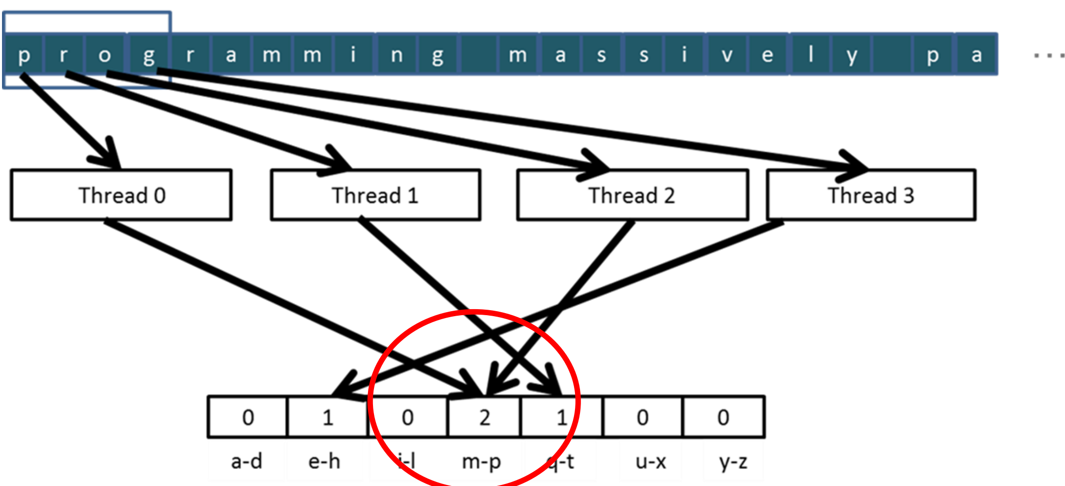


Implementation 2



Evaluation of Implementation 2

+ Better memory access patterns
- Still possibility of collision due to data races



Implementation 3

- We need to deal with data races:
  - read-modify-write operations

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Implementation 3

- We need atomic operation for read-modify-write.
- A read-modify-write operation performed by a single hardware 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 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 typically be held in a queue
  - All threads perform their atomic operations serially on the same location

Implementation 3

- Atomic operations in CUDA
  - Atomic add, sub, inc, dec, min, max, exch (exchange)
  - CAS (compare and swap)
    - 3 args: address, compare, val
    - reads a value from address (old value)
    - computer: Old = compare ? val : old
- Example:
  - int atomicAdd(int* address, int val);
  - unsigned int atomicAdd(unsigned int* address, unsigned int val);
  - atomicSub, atomicExch, atomicMin, … atomicAnd, AtomicOr, …
- atomicAdd for double precision floating-point numbers requires CC 6.0 or higher

Implementation 3

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

// stride is total number of threads
    int stride = blockDim.x * gridDim.x;

 // All threads handle blockDim.x * gridDim.x
 // consecutive elements
   while (i < size) {
      int alphabet_position = buffer[i] – "a";
      if (alphabet_position >= 0 && alpha_position < 26)
             atomicAdd(&(histo[alphabet_position/4]), 1);
       i += stride;
   }
}
```

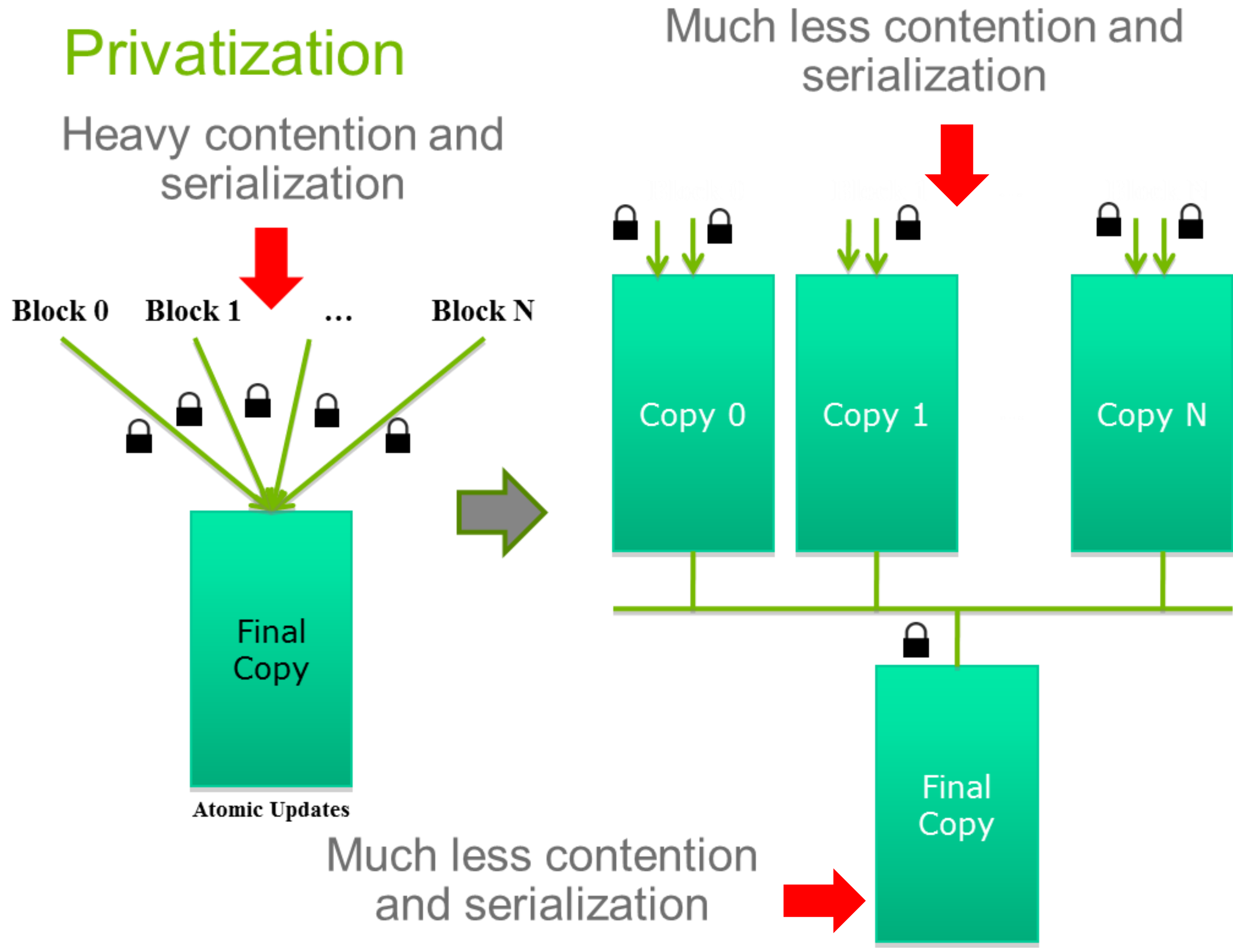
About Atomic Operations

- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles.
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles.
- During this whole time, no one else can access the location → serialization!
- Shorter latency is services from L2 cache.
- Much shorter latency if the operation is on shared memory (100x higher throughput than global and 10x than L2).

Evaluation of Implementation 3

+ No data race
+ Coalesced memory access
- Performance loss due to serialization

Implementation 4



Implementation 4

- Privatization: a very important use case for shared memory
- Cost
  - Overhead for creating and initializing private copies
  - Overhead for accumulating the contents of private copies into the final copy
- Benefit
  - 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

Implementation 4

- 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[7];
```

Implementation 4

- 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[7];

   if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
   __syncthreads();
```

Initialize the bin counters in the private copies of histo[]

Implementation 4

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(private_histo[buffer[i]/4), 1);
        i += stride;
    }
```

Build the private histogram

Implementation 4

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

  if (threadIdx.x < 7) {
        atomicAdd(&(histo[threadIdx.x]),
                              private_histo[threadIdx.x] );
  }

}
```

Build the final histogram

About Privatization

- Privatization is a powerful and frequently used technique for parallelizing applications
- The operation needs to be associative and commutative
- The private histogram size needs to be small
- Fits into shared memory
- What if the histogram is too large to privatize?
  - Sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared memory

What we learned from the histogram example

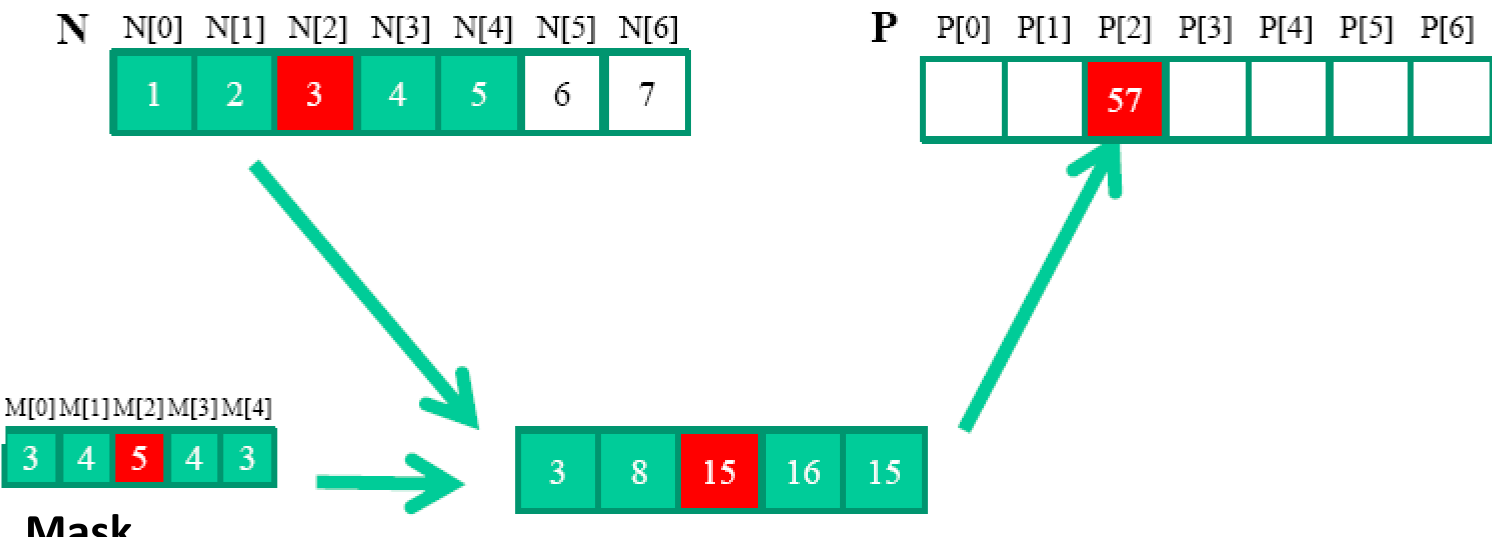
- Atomic operations may be needed → sacrificing some performance for correctness
- Privatization can sometimes reduce the performance loss due to serialization caused by atomic operations.

Pattern: Convolution

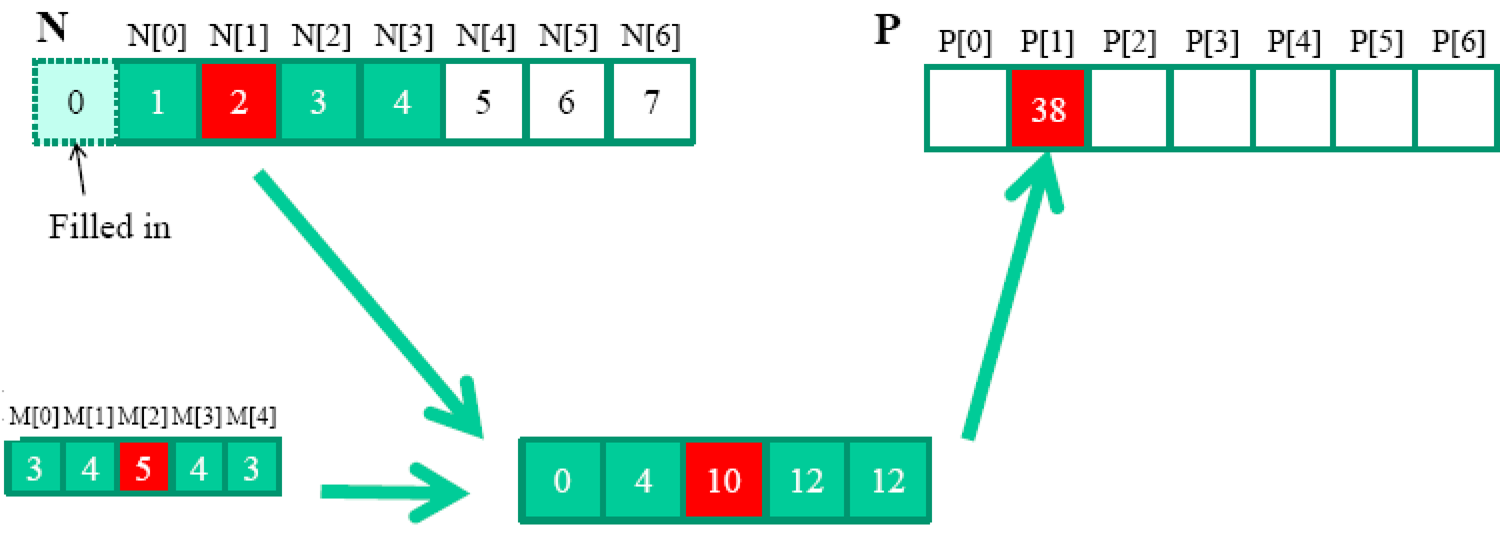
Convolution

- An Array operation
- Output data element = weighted sum of a collection of neighboring input elements.
- The weights are defined by an input mask array.
- Usually used as filters to transform signals (or pixels or …) into more desirable form.

Convolution



Convolution



Convolution can also be 2D.

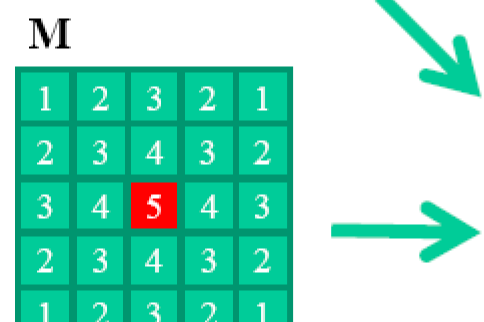
Convolution

N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

		321				



1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

Convolution

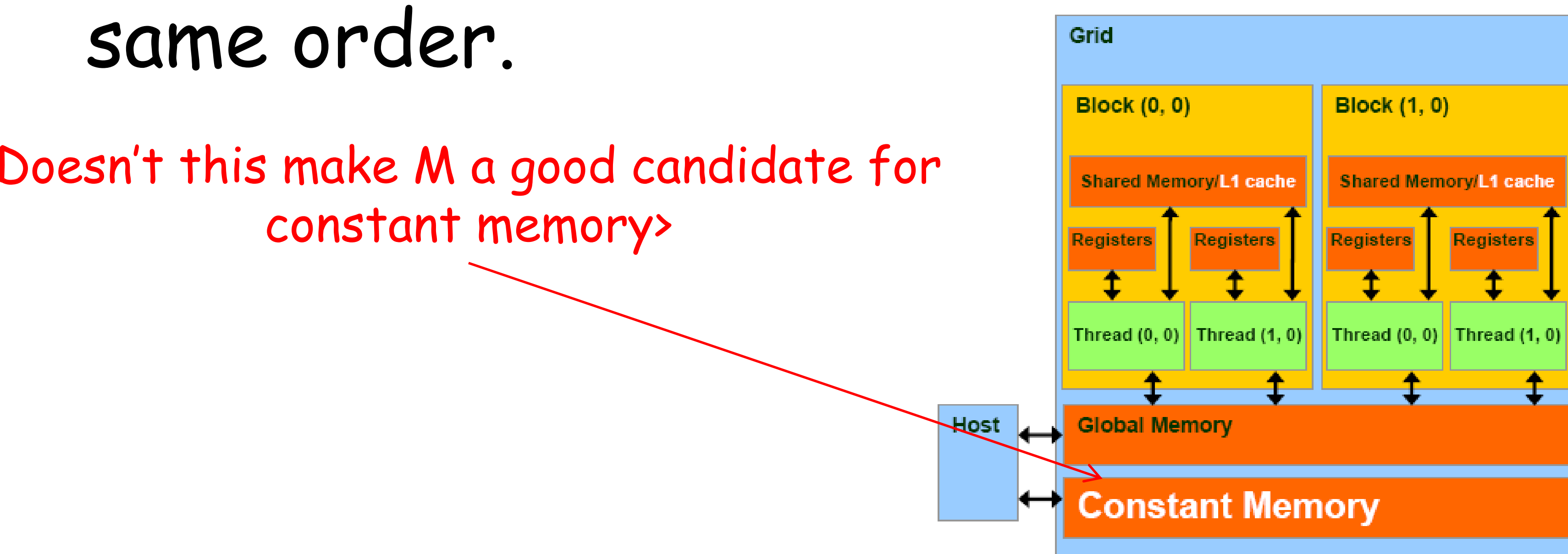
```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

The 1D Version

- Thread organized as 1D grid.
- Pvalue allows intermediate values to be accumulated in registers to save DRAM bw.
- We assume *ghost* values are 0.
- There will be control flow divergence (due to ghost elements).
- Ratio of floating point arithmetic calculation to global memory access is ~ 1.0 → What can we do??

Regarding Mask M

- Size of M is typically small.
- The contents of M do not change during execution.
- All threads need to access M and in the same order.



Constant Memory

- Constant memory variables are visible to all thread blocks.
- Constant memory variables cannot be changed during kernel execution.
- The size of constant memory can vary from device to device.

Mask M and Constant Memory

- In host:
  - *#define MASK_WIDTH 10*
    __constant__ float M[MASK_WIDTH]
  - Allocate and initialize a mask h_M
  - cudaMemcpyToSymbol(M, h_M, MASK_WIDTH * sizeof(float),
    offset, kind);
- Kernel functions
  - access constant memory variables as global variables → no need to pass pointers of these variables to the kernel as parameter.

Question: Isn't the constant memory also in DRAM? Why is it assumed faster than global memory?

Answer:

- CUDA runtime knows that constant memory variables are not modified.
- It directs the hardware to aggressively cache them during kernel execution.

Pattern: Reduction Tree

What is it?

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set
  - 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
- Google and Hadoop MapReduce frameworks are examples of this pattern

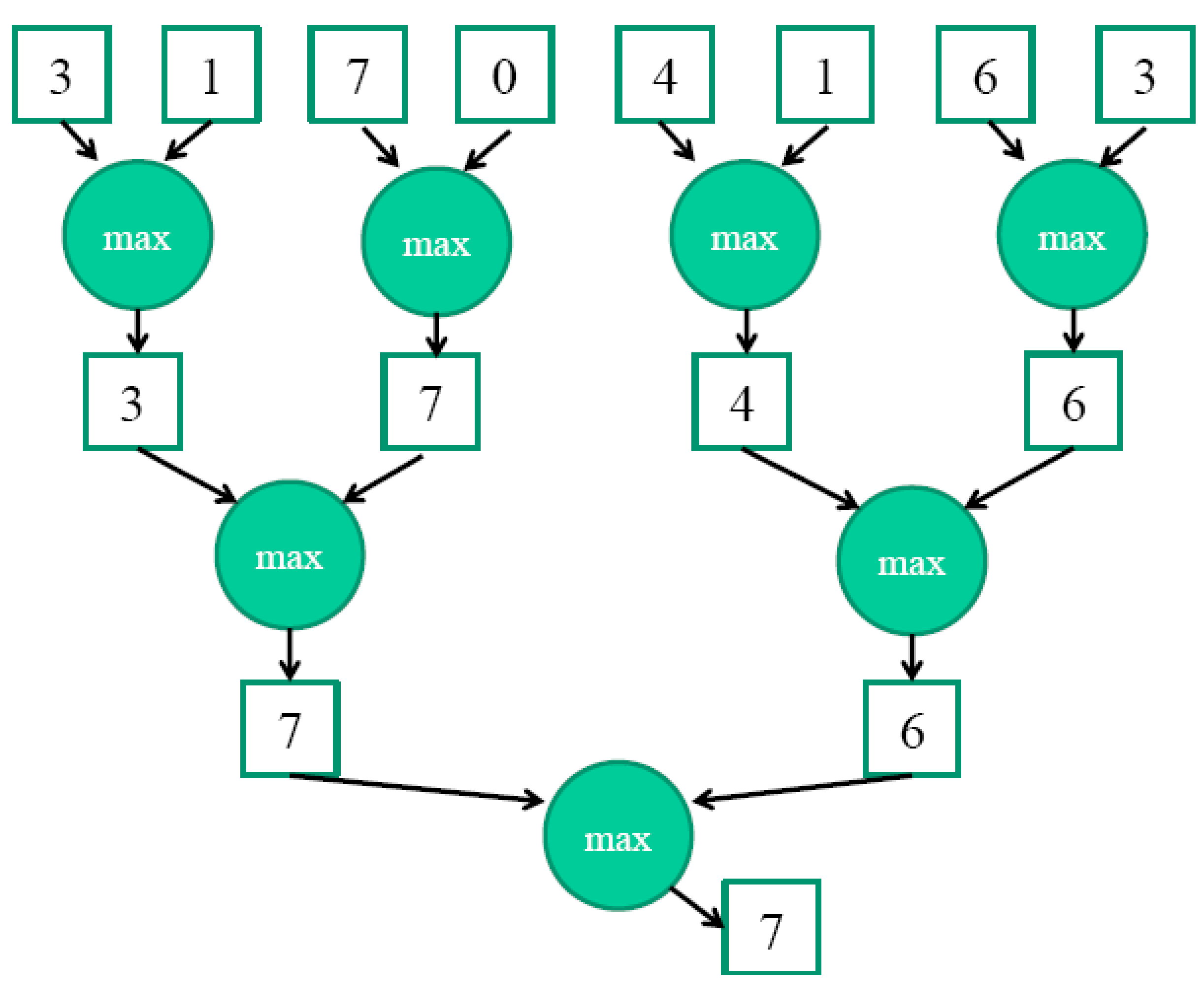
What is it?

- 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 in 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



Straightforward Implementation

- 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

First Step: Block Design

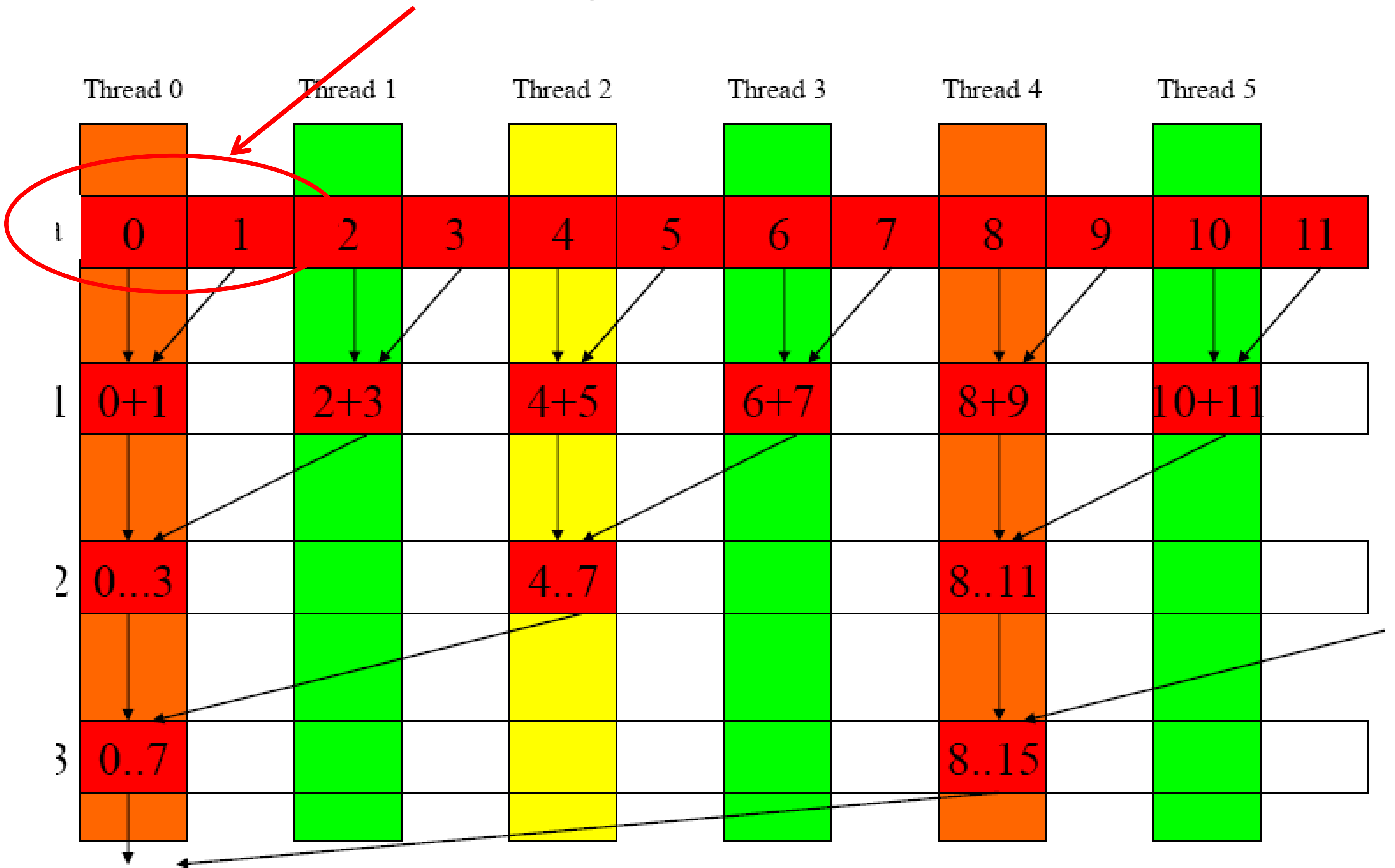
- Each thread block takes 2*BlockDim.x input elements
- Each thread loads 2 elements into shared memory

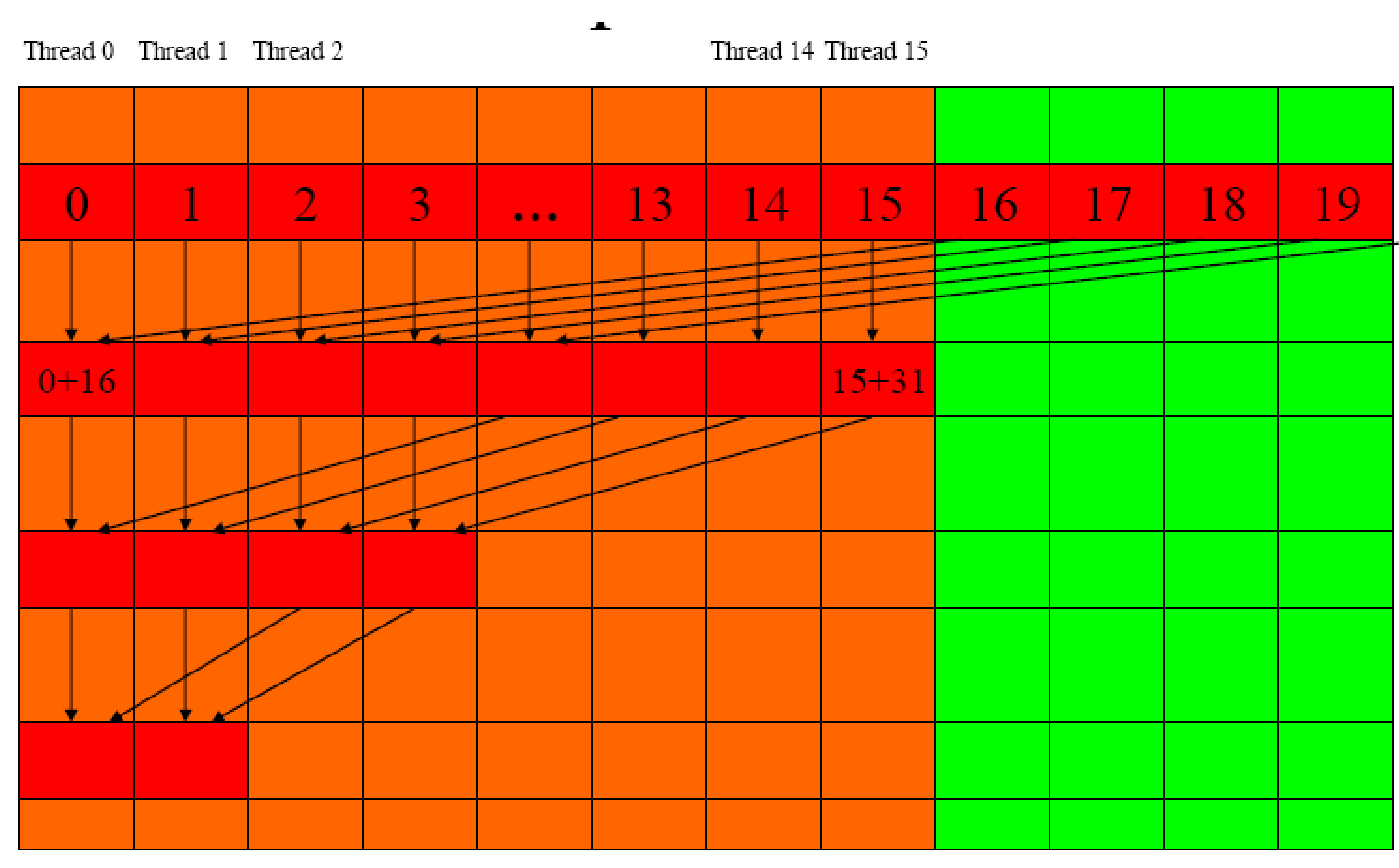
```
__shared__ float partialSum[2*BLOCK_SIZE];

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];
```

Step 2: Reduction

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

A lot of branch divergence.





A Better Version

Better Reduction Step

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

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
  - All threads in each warp either all active or all inactive
- The final 5 steps will still have divergence

Be Careful!

- 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.

Pattern: Prefix Sum (Scan)

Scan / Parallel Prefix Sum

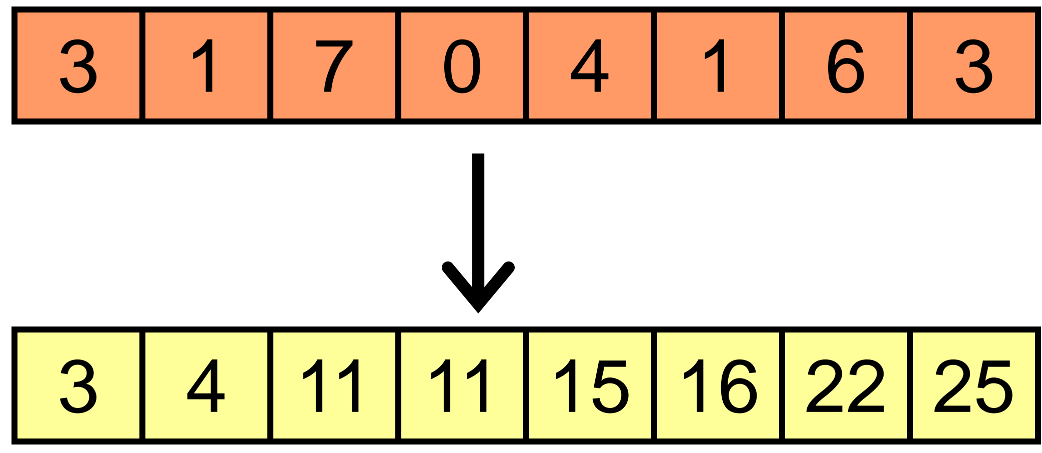
3	1	7	0	4	1	6	3

↓

0	3	4	11	11	15	16	22

- Given an array **A = [a0, a1, …, an-1]**
  and a binary associative operator @ with identity I
  - **scan (A) = [I, a0, (a0 @ a1), …, (a0 @ a1 @ … @ an-2)]**
- This is the **exclusive scan**

Inclusive Scan



- Given an array **A = [a0, a1, …, an-1]** and a binary associative operator @ with identity I
  - scan (A) = [a0, (a0 @ a1), …, (a0 @ a1 @ … @ an-1)]

Why?

- Scan is used as a building block for many parallel algorithms, especially those involving **parallel working assignment and resource allocation**
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Run-length encoding
  - Histograms
  - Etc.

A Inclusive Scan Application Example

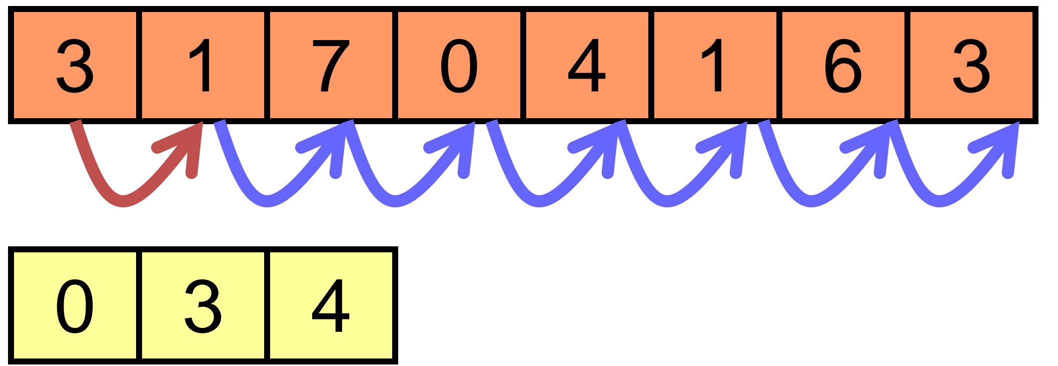
- Assume that we have a 100-inch sausage to feed 10
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sausage quickly?
- How much will be left

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Other Examples

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- …

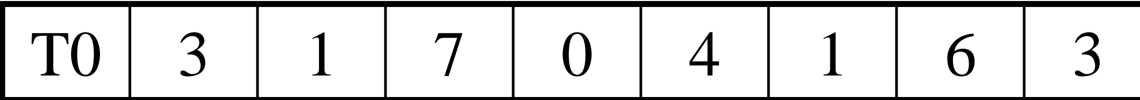
Sequential algorithm




```
void scan( float* output, float* input, int length)
{
    output[0] = 0;
    for(int j = 1; j < length; ++j)
    {
        output[j] = input[j-1] + output[j-1];
    }
}
```

- *N* additions
- Use a guide:
  - Want parallel to be *work efficient*
  - Does similar amount of work

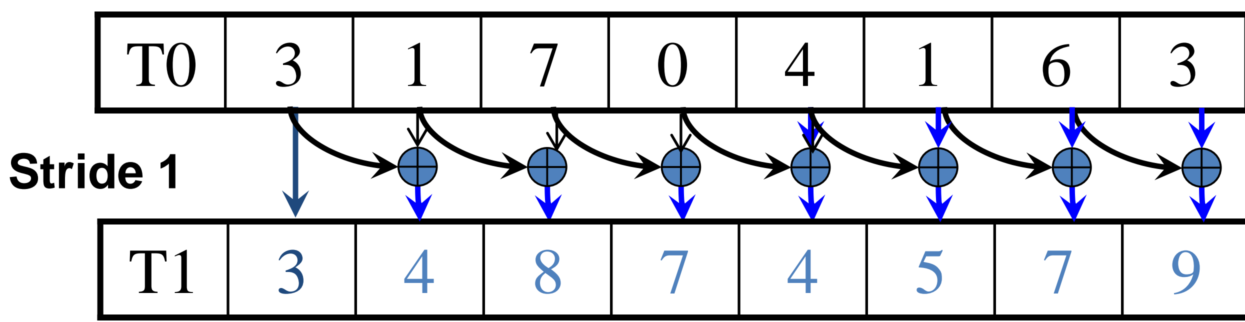
A Parallel Inclusive Scan Algorithm



1. Read input from device memory to shared memory

Each thread reads one value from the input array in device memory into shared memory array.

A Parallel Scan Algorithm

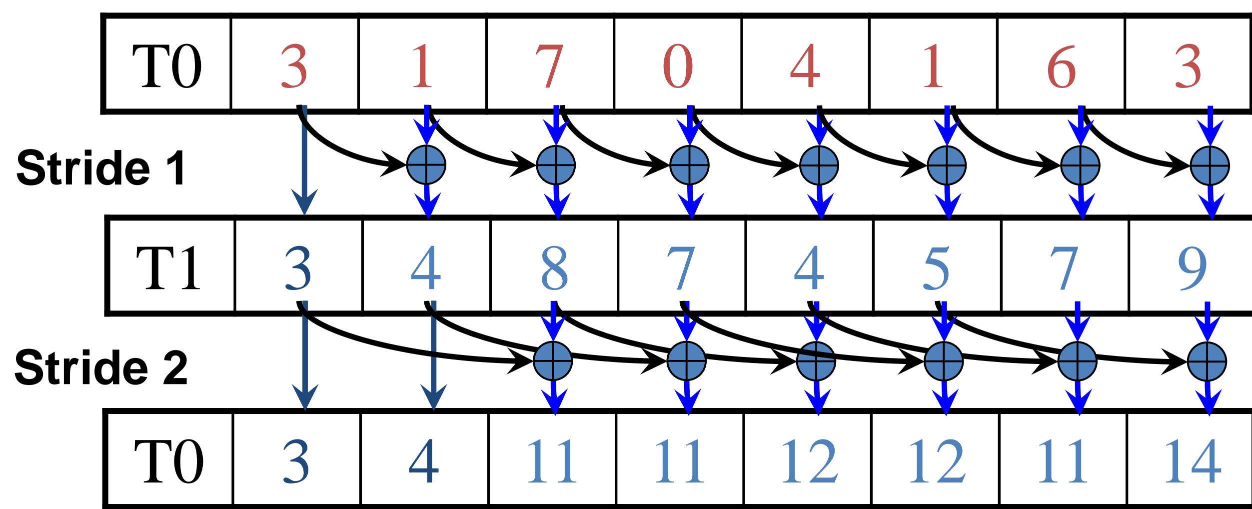


1. (previous slide)
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements s*tride* elements apart. Double *stride* at each iteration.

Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

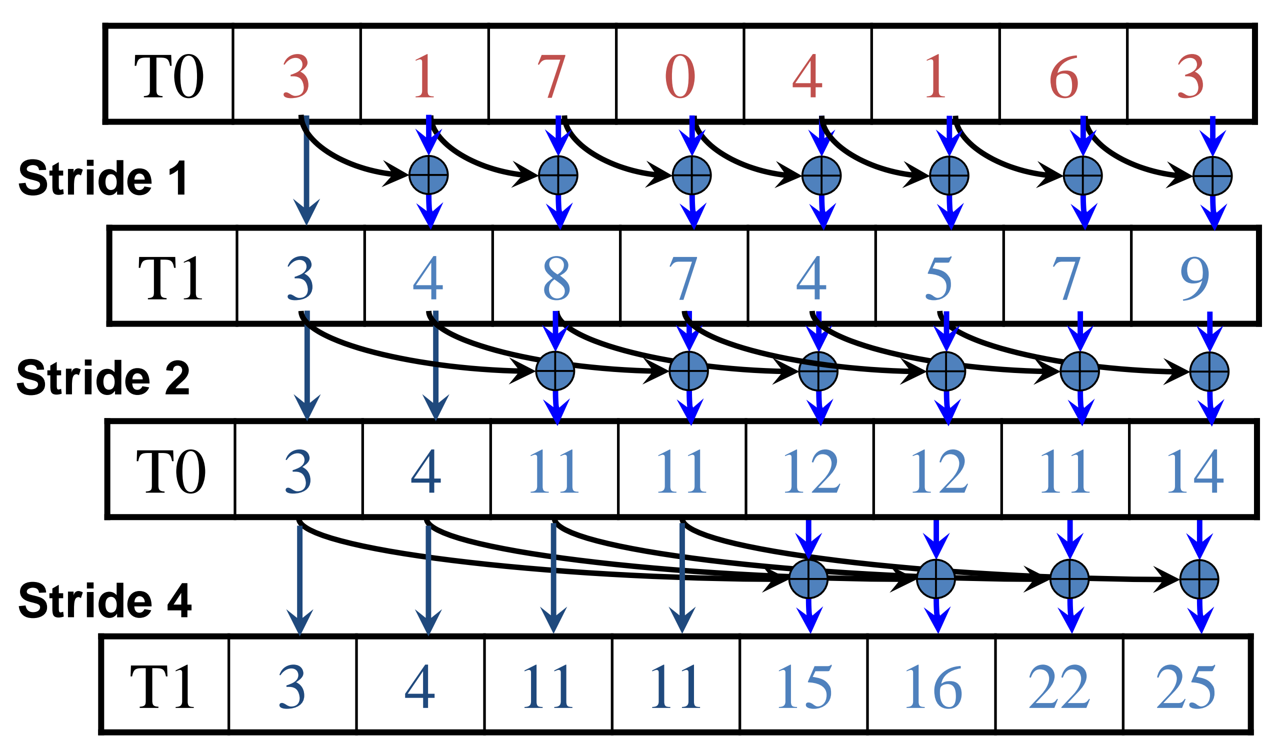
A Parallel Scan Algorithm



1. Read input from device memory to shared memory.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements s*tride* elements apart. Double *stride* at each iteration.

Iteration #2
Stride = 2

A Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements s*tride* elements apart. Double *stride* at each iteration.
3. Write output from shared memory to device memory

How to handle dependencies in that implementation?

- During every iteration, each thread can overwrite the input of another thread
- Barrier synchronization to ensure all inputs have been properly generated
- Barrier synchronization is required to ensure that all threads have secured their inputs
- All threads perform addition and write output

Possible Implementation

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int InputSize) {
  __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 <= threadIdx.x; stride *= 2) {
      __syncthreads();
         float in1 = XY[threadIdx.x - stride];
         __syncthreads();
         XY[threadIdx.x] += in1;
      }
    __ syncthreads();

   If (i < InputSize) {Y[i] = XY[threadIdx.x];}
}
```

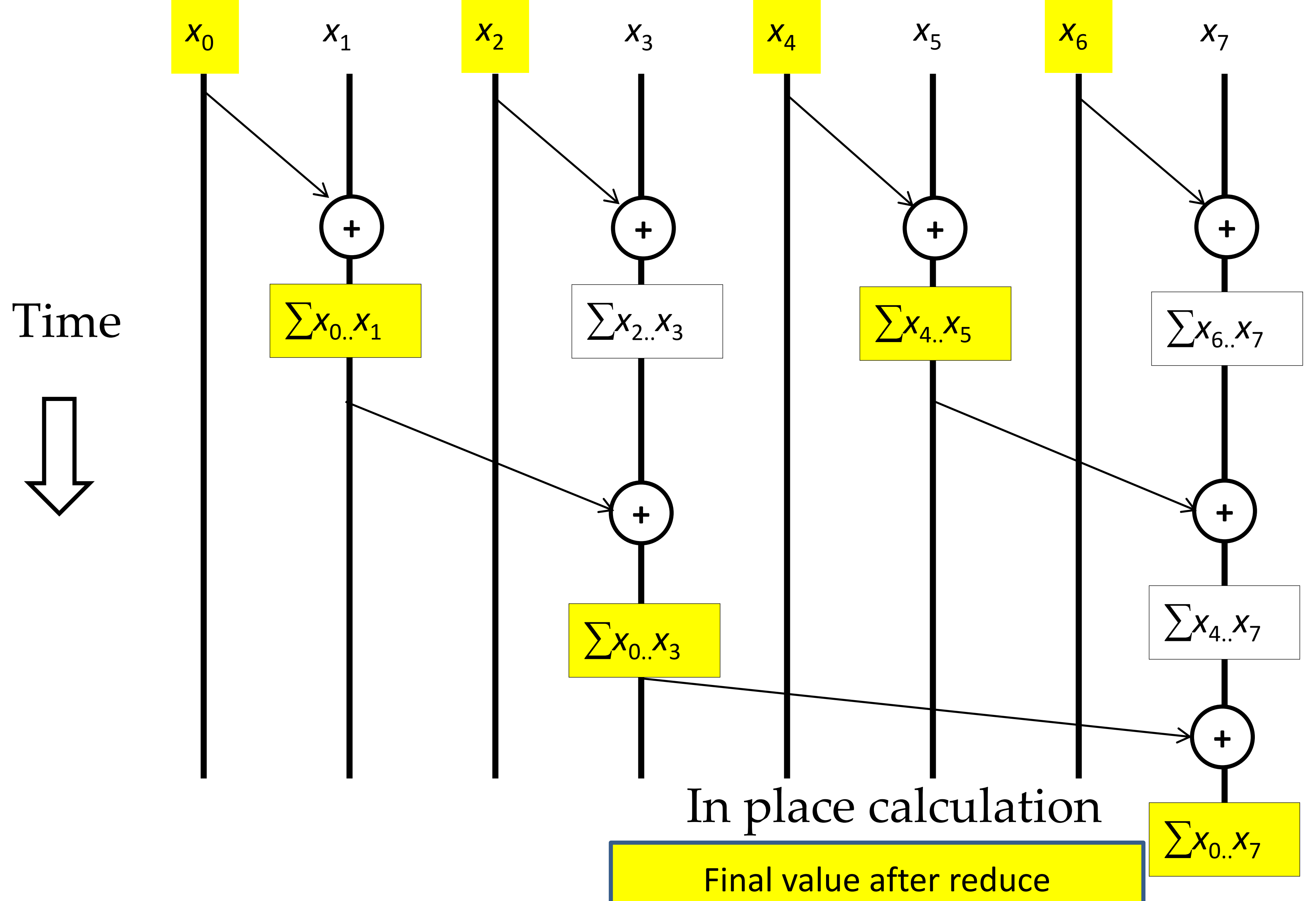
Work Efficiency Considerations

- The first-attempt Scan executes log(n) iterations
- This scan algorithm is not very work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

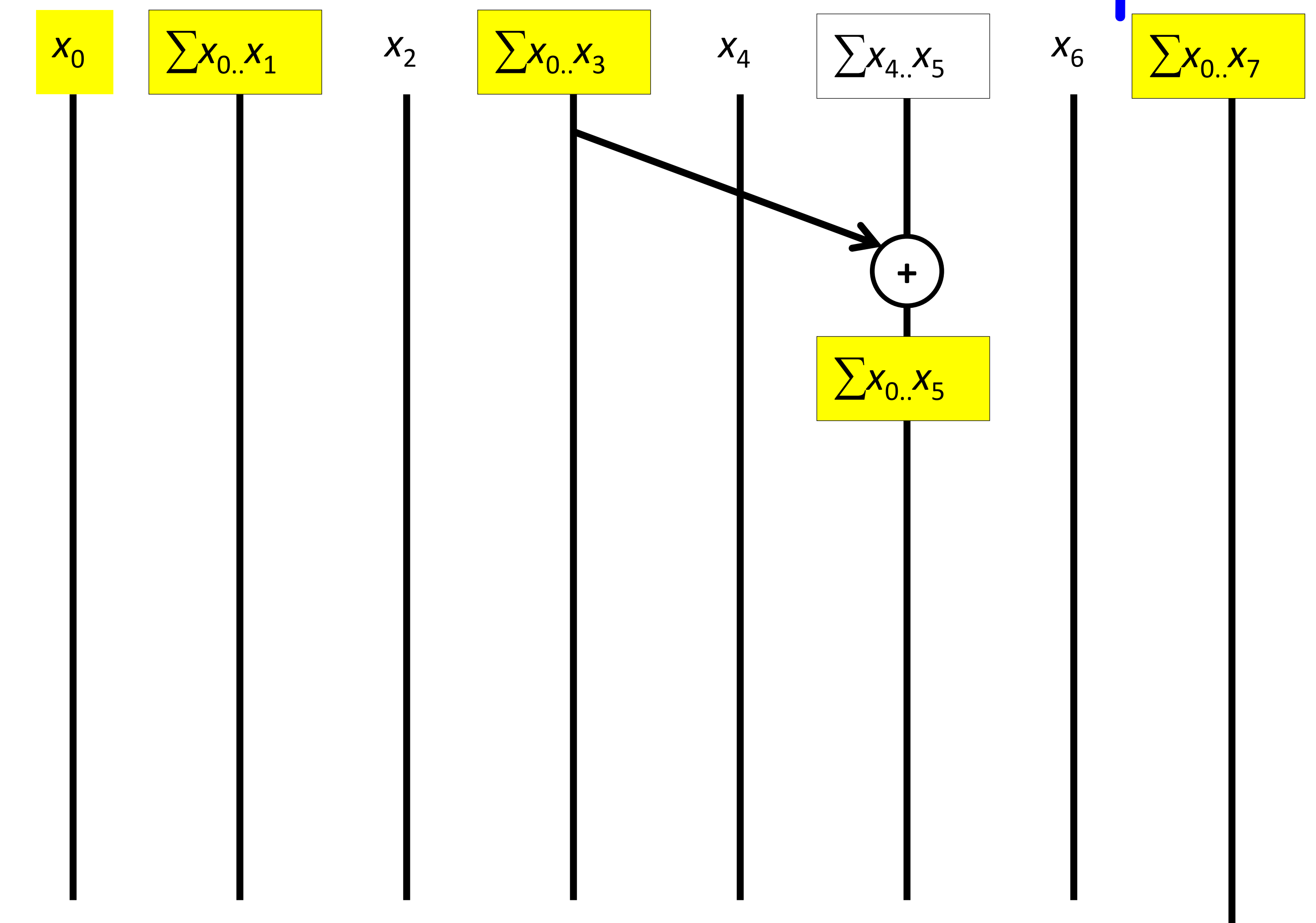
Improving Efficiency

- A common parallel algorithm pattern:
  Balanced Trees
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

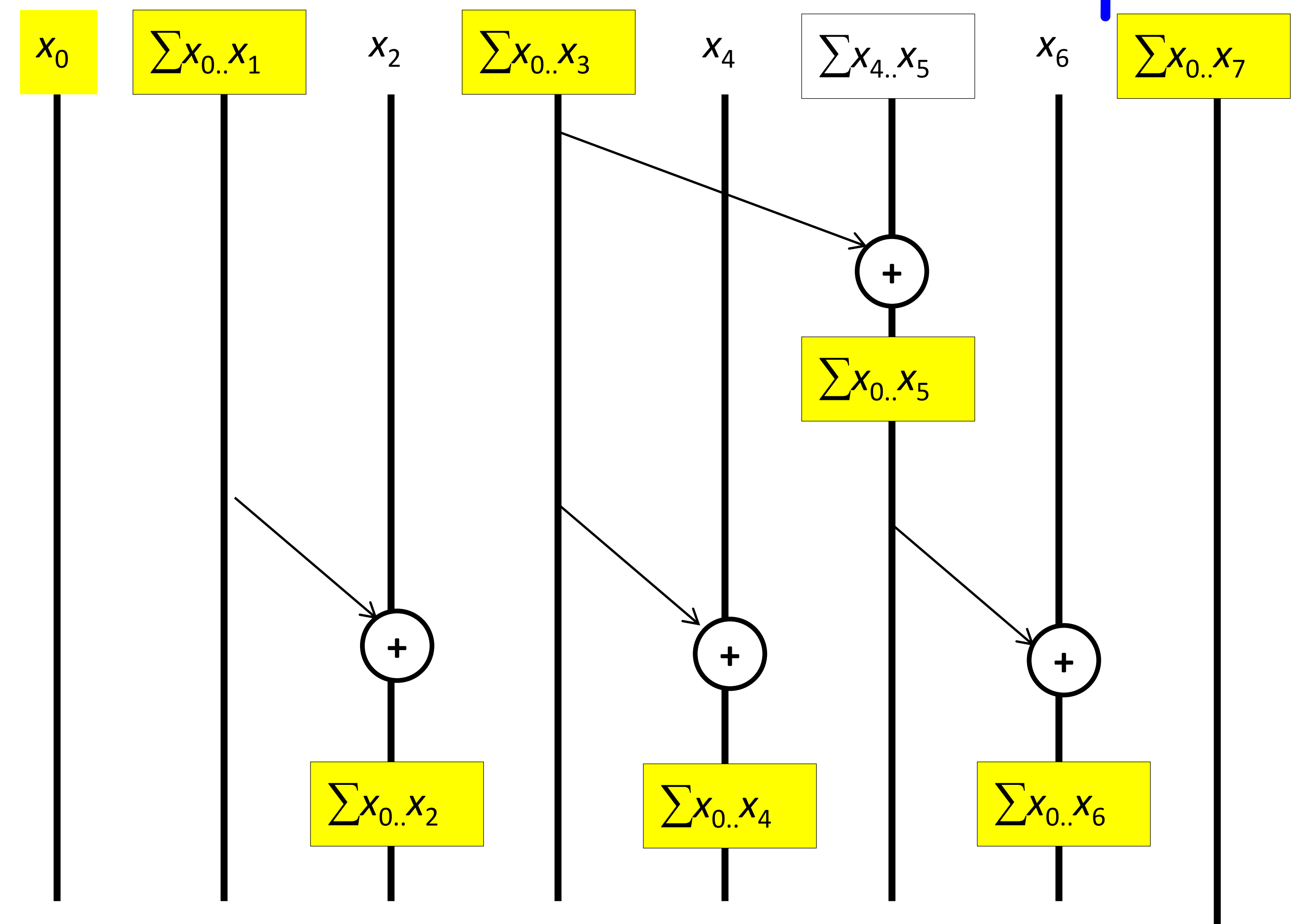
Parallel Scan - Reduction Step



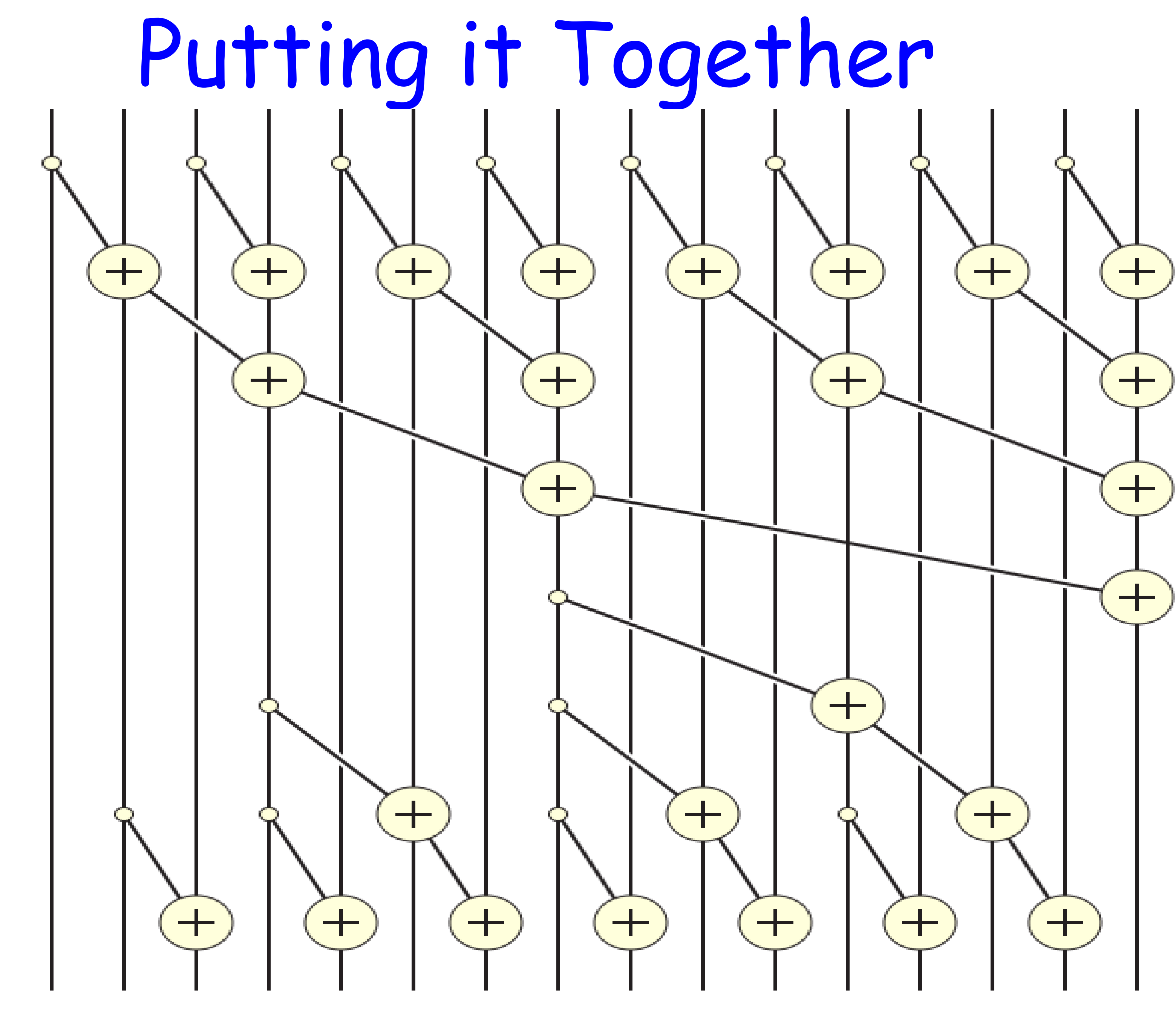
Inclusive Post Scan Step



Inclusive Post Scan Step



Putting it Together



Work Analysis

- The parallel Scan executes 2* log(n) parallel iterations
  - log(n) in reduction and log(n) in post scan
  - The iterations do n/2, n/4,..1, 1, …., n/4. n/2 adds
  - Total adds: 2* (n-1) → O(n) work
- The total number of adds is no more than twice of that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware

Error Handling in CUDA

```
__global__ void foo(int *ptr)
    {
      *ptr = 7;
    }

    int main(void)
    {
      foo<<<1,1>>>(0);
      return 0;
    }
```

What will happen when you compile and execute this piece of code?

Error Handling

- In a CUDA program, if we suspect an error has occurred during a kernel launch, then we must explicitly check for it after the kernel has executed.
- CUDA runtime will respond to questions … But won't talk without asked!

cudaError_t cudaGetLastError(void);

- Called by the host
- returns a value encoding the kind of the last error it has encountered
- check for the error only after we're sure a kernel has finished executing → don't forget kernel calls are async!
  - What will you do?

```
$ nvcc crash.cu -o crash
$ ./crash
CUDA error: unspecified launch failure
```

```
#include <stdio.h>
#include <stdlib.h>

__global__ void foo(int *ptr)
{
  *ptr = 7;
}

int main(void)
{
  foo<<<1,1>>>(0);

  // make the host block until the device is finished with foo
  cudaThreadSynchronize();

  // check for error
  cudaError_t error = cudaGetLastError();
  if(error != cudaSuccess)
  {
    // print the CUDA error message and exit
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }

  return 0;
}
```

Same Technique with Synchronous Calls

```
cudaError_t error = cudaMalloc((void**)&ptr,
                                        1000000000000);
 if(error != cudaSuccess)
 {
   // print the CUDA error message and exit
   printf("CUDA error: %s\n",
     cudaGetErrorString(error));
   exit(-1);
 }
```

The output will be:

CUDA error: out of memory

Rules of Thumb

- Do not use cudaThreadSynchronize() a lot in your code because it has a large performance penalty.
- You can enable it during debugging and disable it otherwise.

```
#ifdef DEBUG
  cudaThreadSynchronize();
  cudaError_t error = cudaGetLastError();
  if(error != cudaSuccess)
  {
   printf("CUDA error at %s:%i: %s\n", filename, line_number, cudaGetErrorString(error));
   exit(-1);
  }
#endif
```

If debugging, compile with:
$ nvcc **-DDEBUG** mycode.cu

Conclusions

- Performance is related to how you keep the GPU and its memory busy → does not necessarily mean higher occupancy.
- We looked at some of the common parallel patterns used in many GPU kernels. These are tools that you can use in your own kernels.