NYU New York University

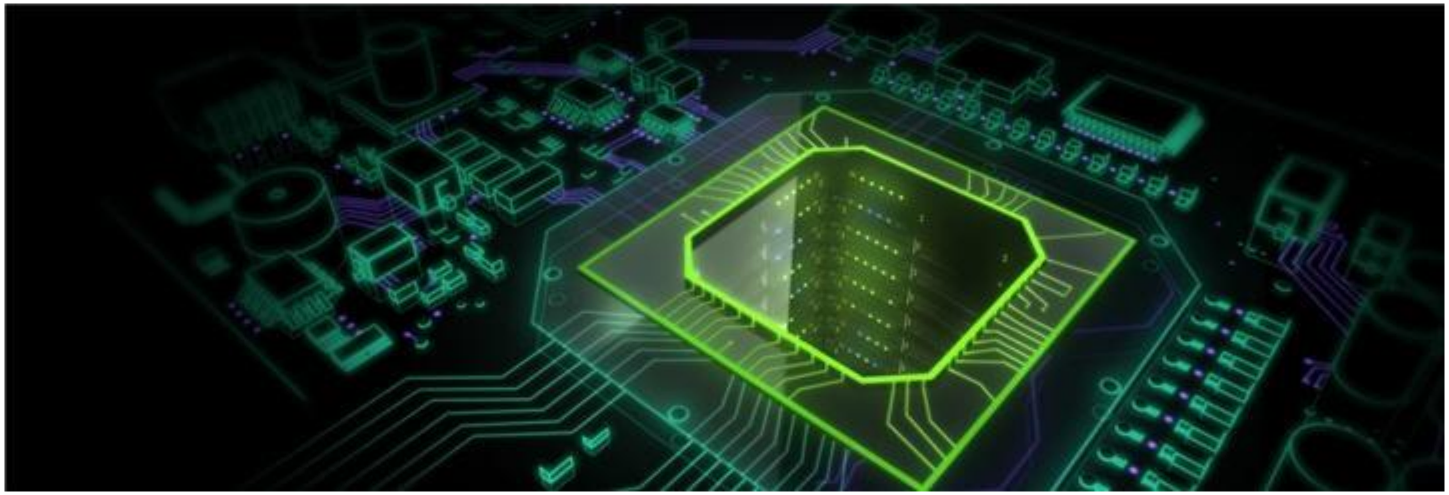
CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA

Advanced Techniques 3

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



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.

Throughput

Latency

Some Insights About Performance

Occupancy

Utilization

It is a common belief that …

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

Computational memory

But is it always true?

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

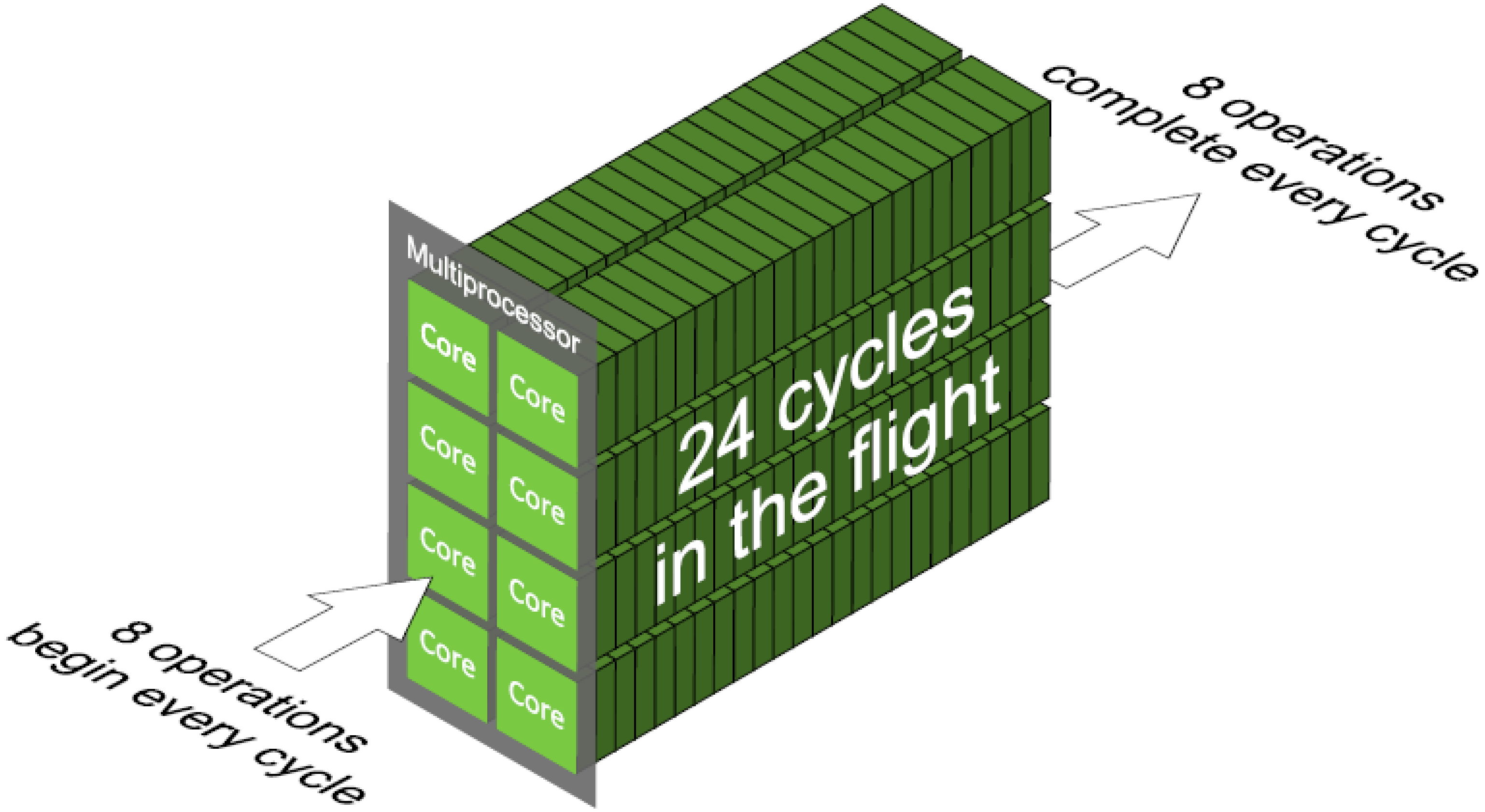
Latency Vs Throughput

- 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 patters

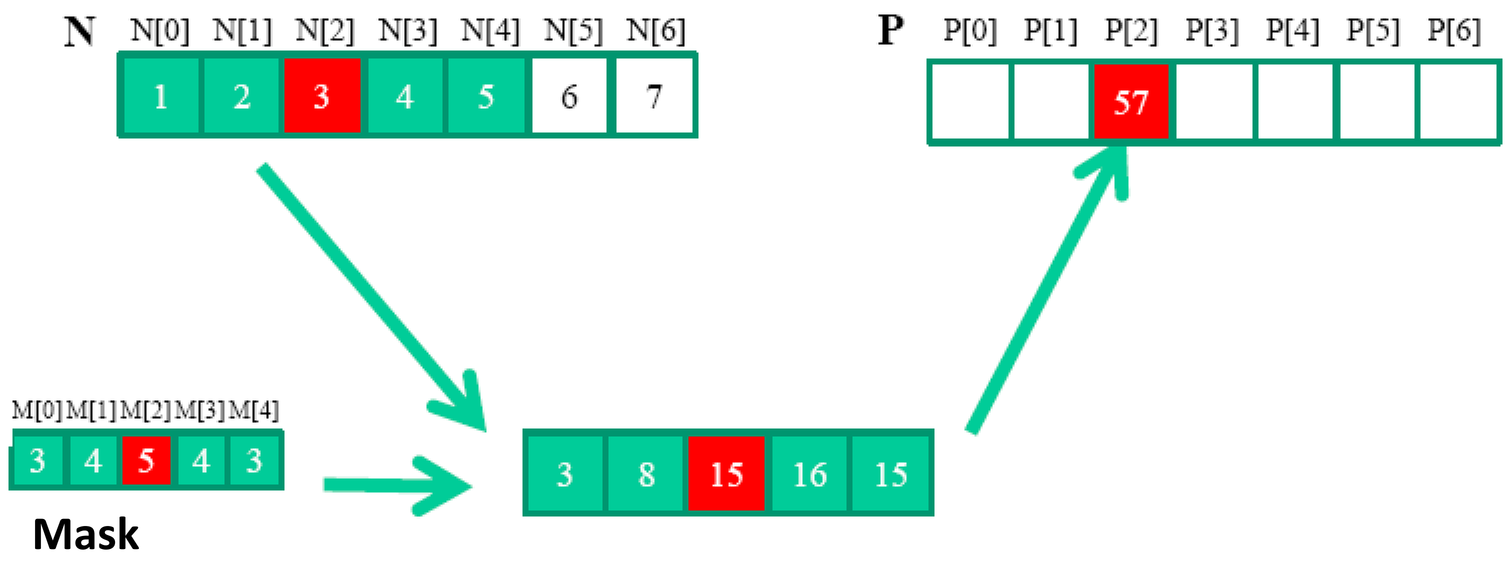
- Convolution
- Reduction tree
- Prefix some

Pattern: Convolution

Convolution

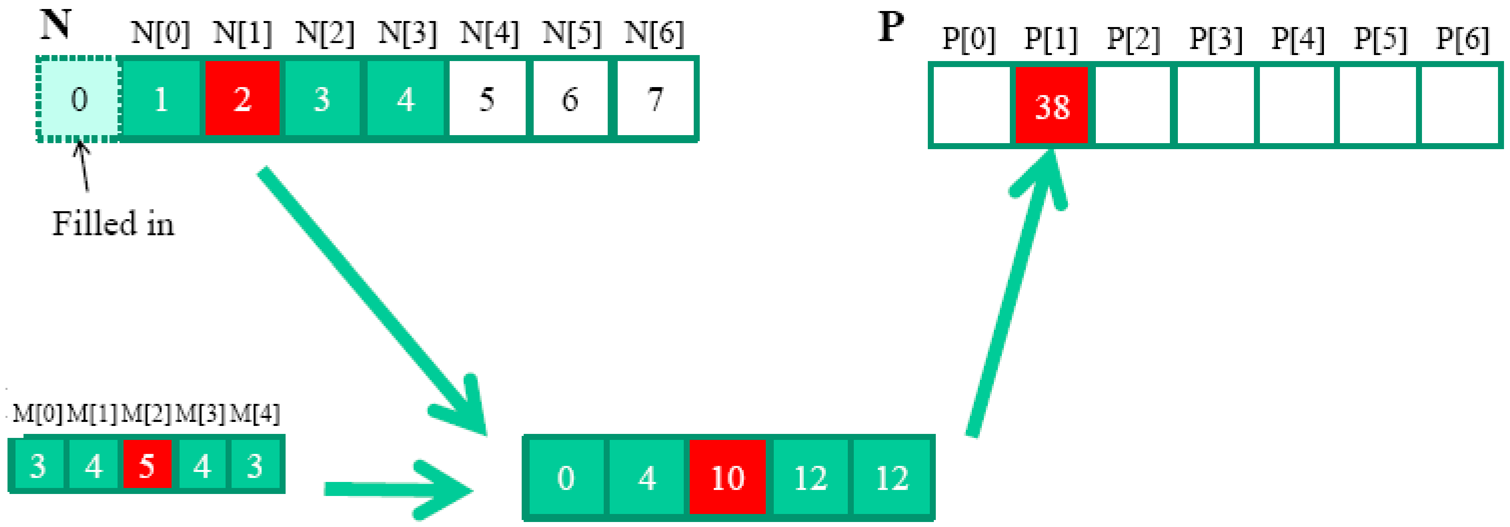
- An Array operation
- Output data element = weighted sum of a collection of neighboring input elements.
- The weights are defined by an input mask array.
- Usually used as filters to transform signals (or pixels or …) into more desirable form.

Convolution



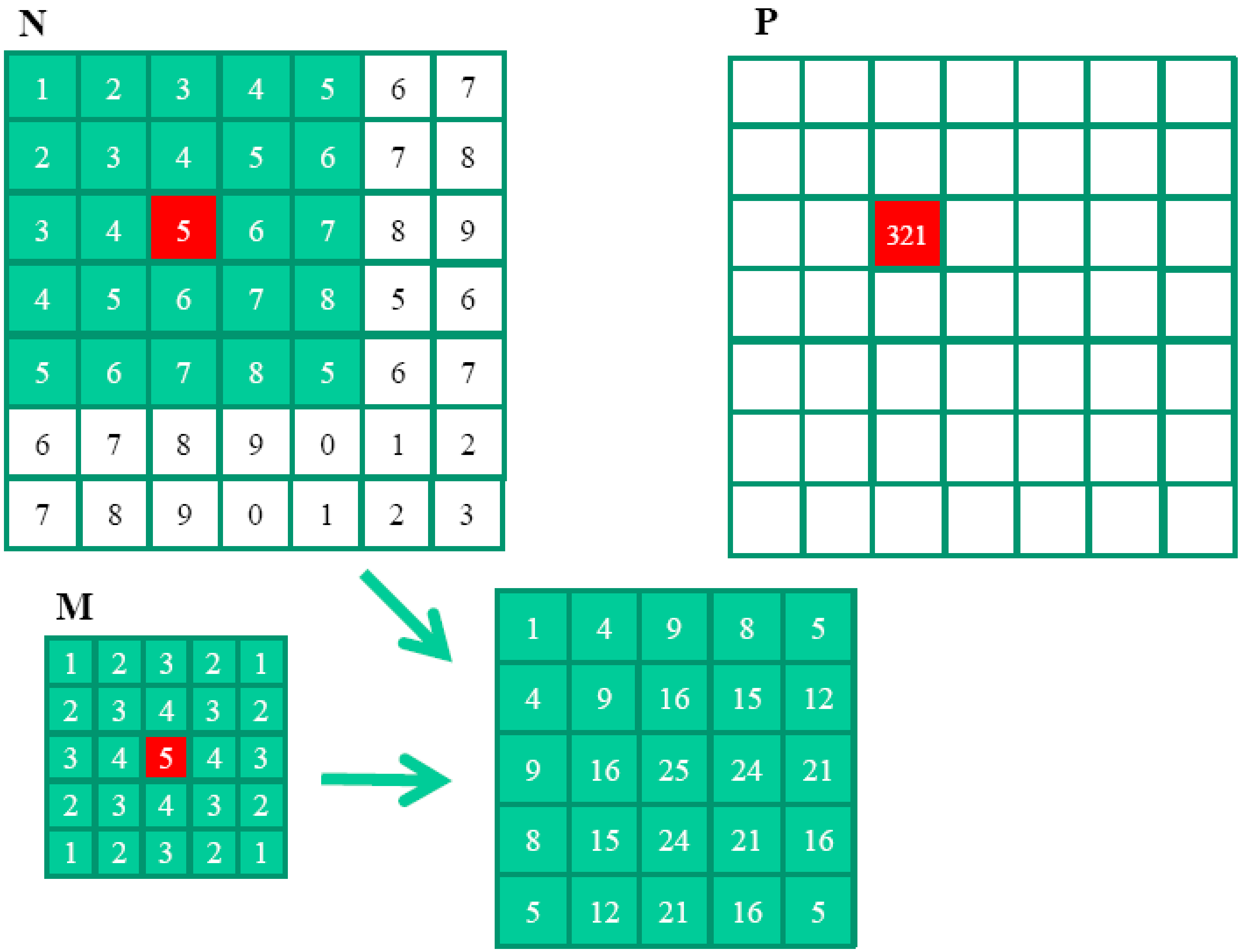
N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
57
M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
Mask
3 8 15 16 15

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				

M

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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;

}
```

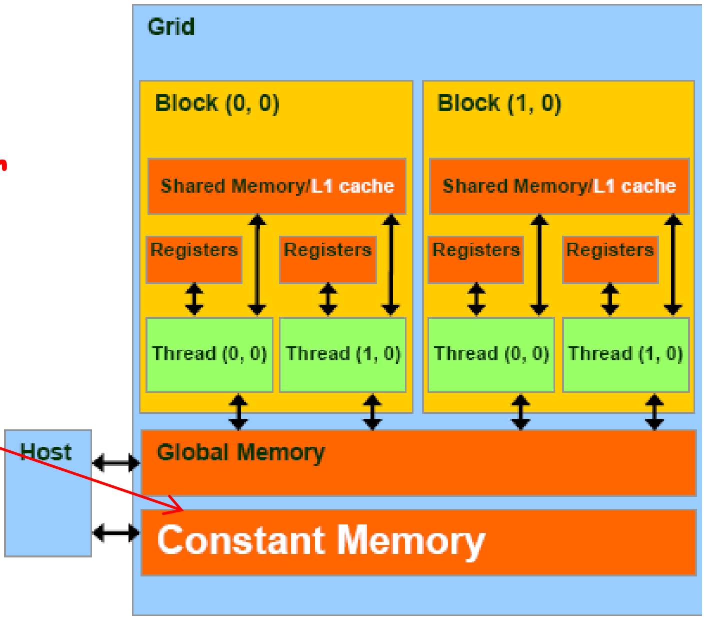
The 1D Version

- 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.

Doesn't this make M a good candidate for constant memory>



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

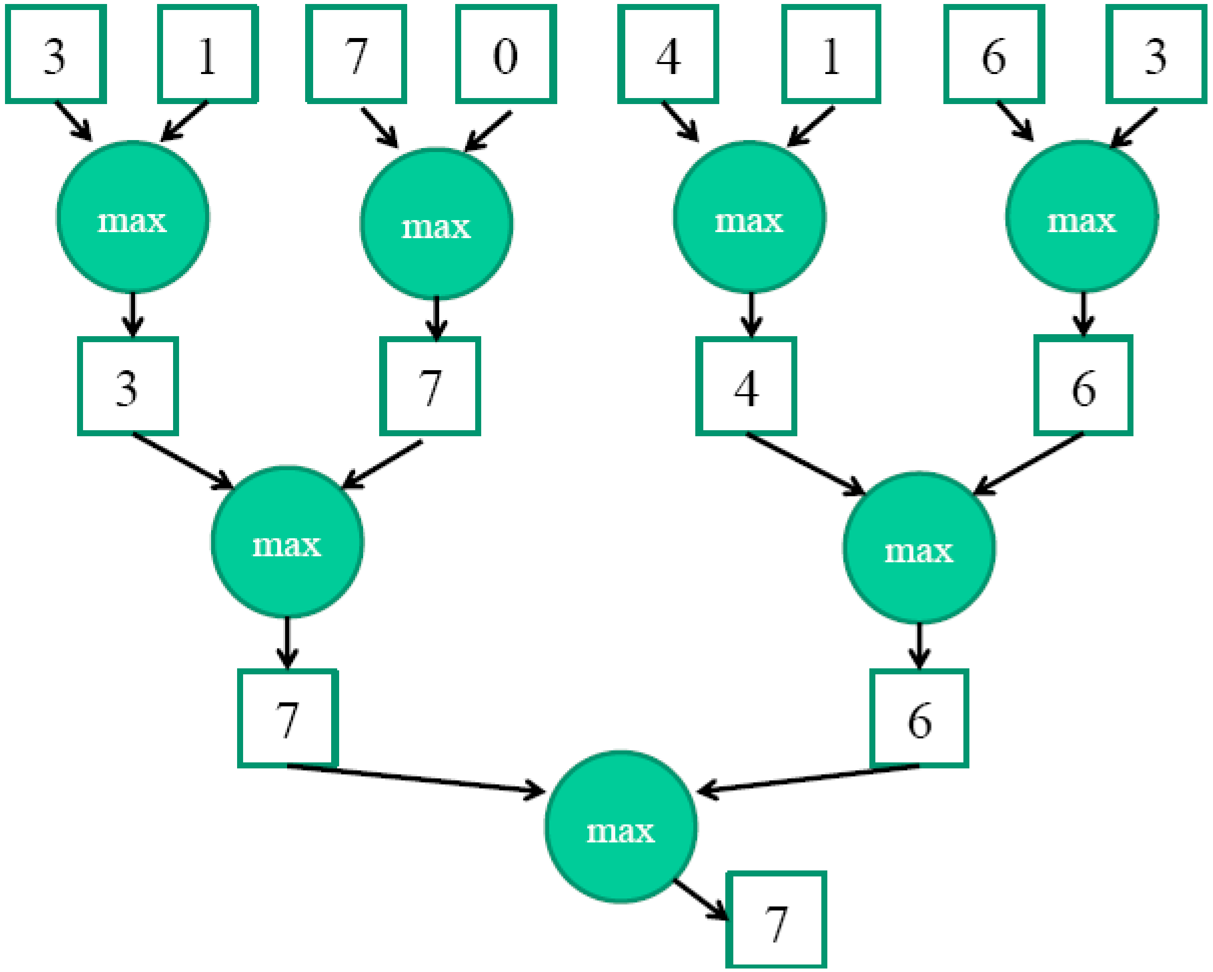
What is it?

- 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

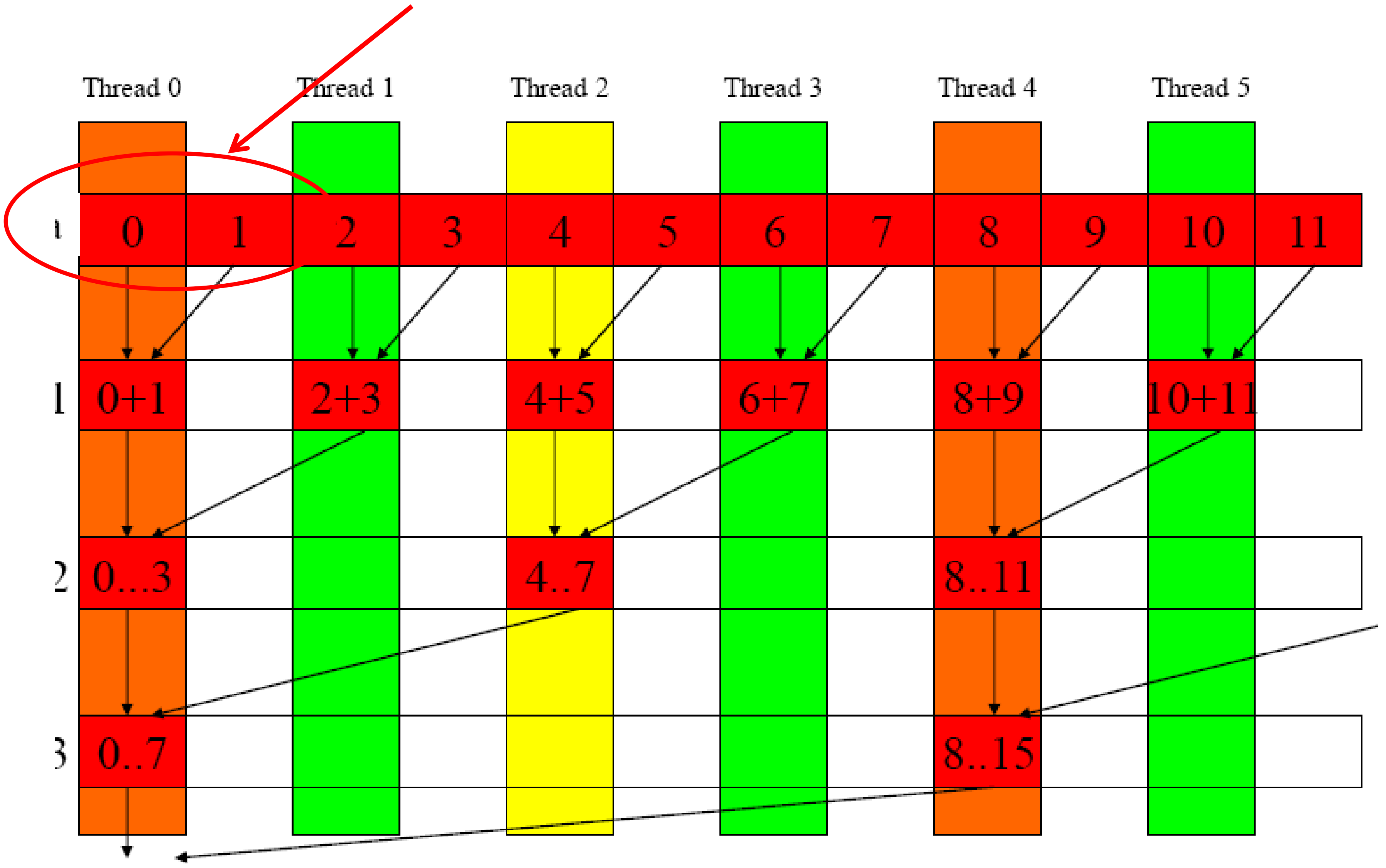
A parallel reduction tree algorithm performs N-1 Operations in log(N) steps

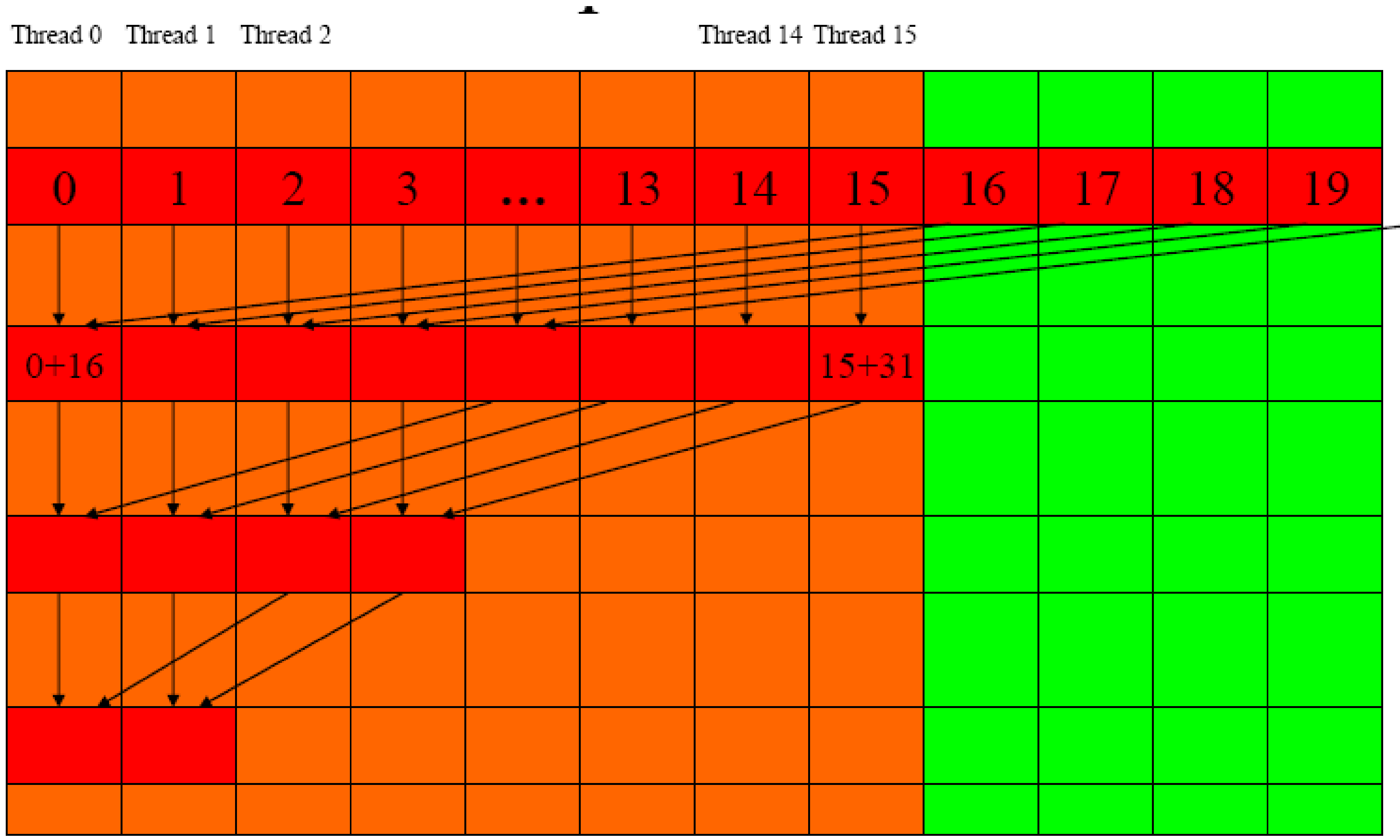


Straightforward Implementation

- 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

A lot of branch divergence.





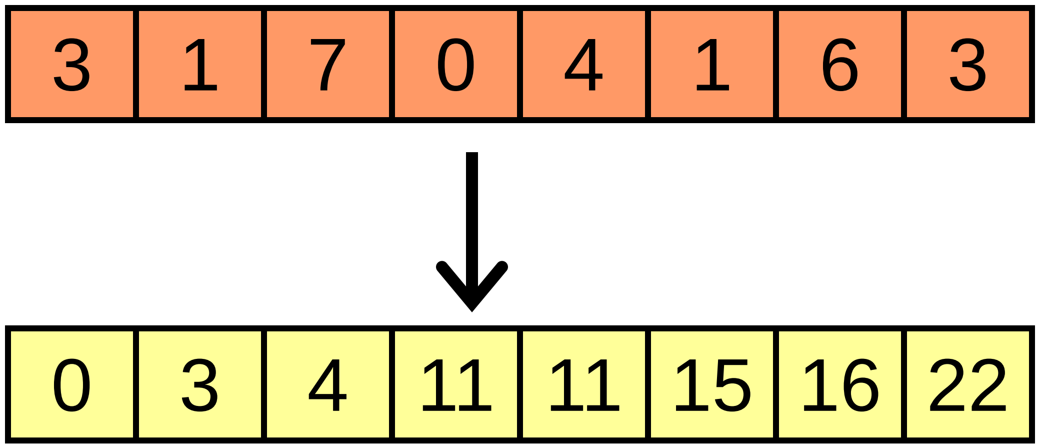
A Better Version

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.

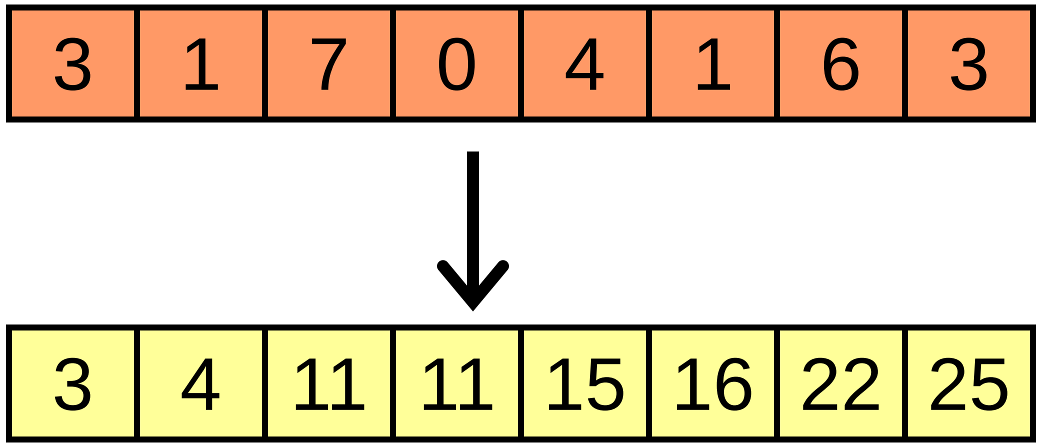
Pattern: Prefix Sum (Scan)

Scan / Parallel Prefix Sum



- 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
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Run-length encoding
  - Histograms
  - Etc.

A Inclusive Scan Application Example

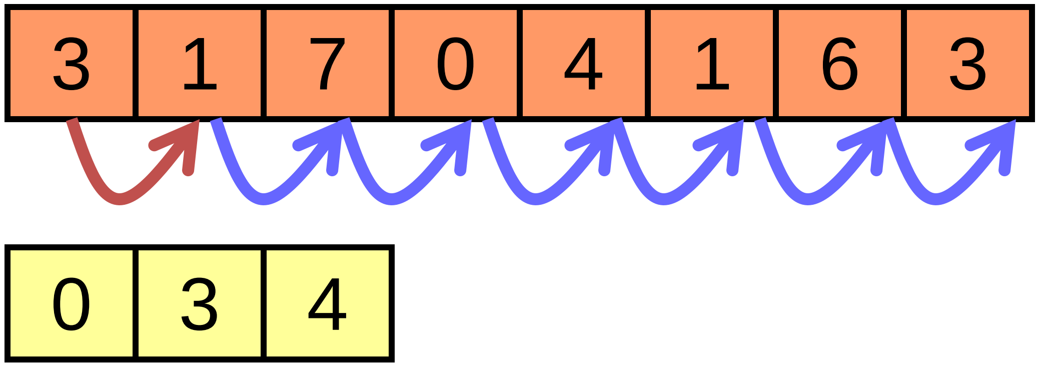
- Assume that we have a 100-inch sausage to feed 10
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sausage quickly?
- How much will be left

- 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

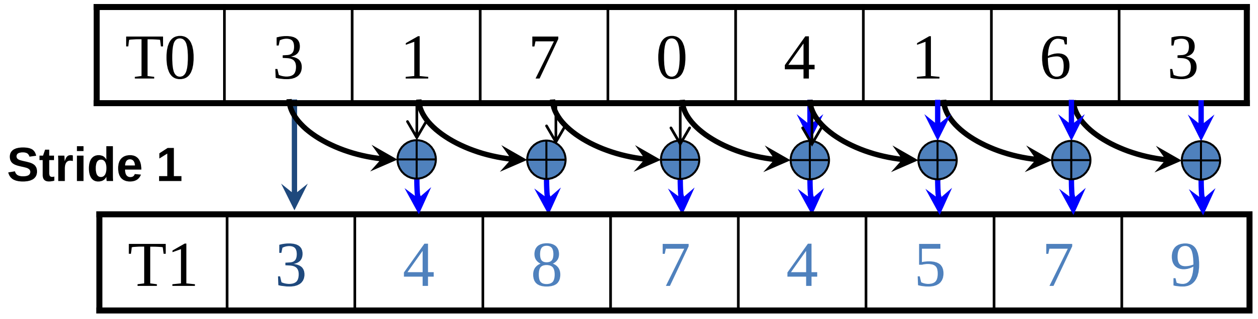
A Parallel Inclusive Scan Algorithm

T0	3	1	7	0	4	1	6	3

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



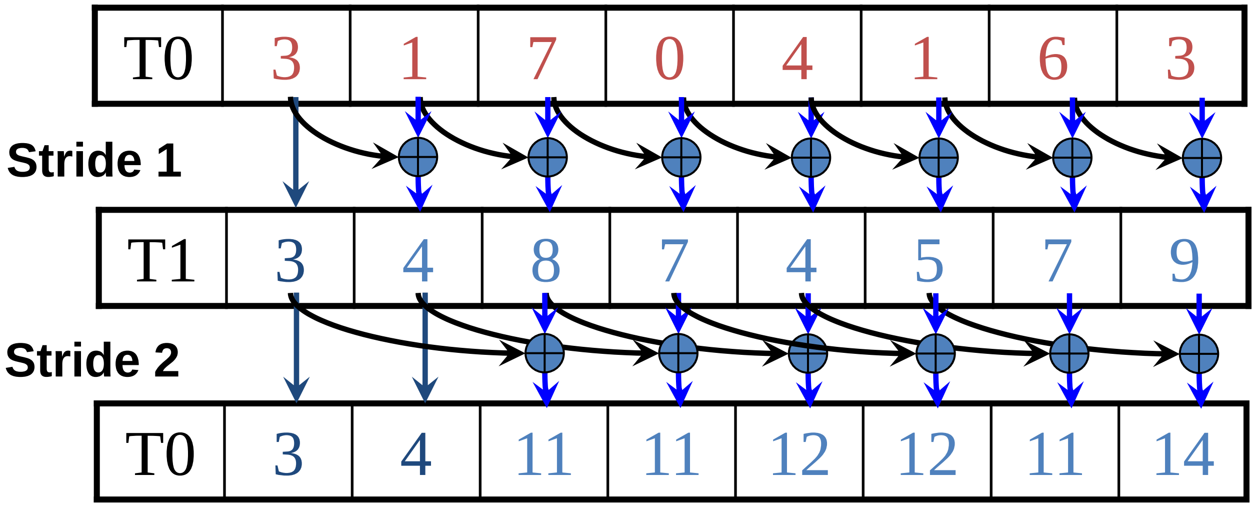
1. (previous slide)

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

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

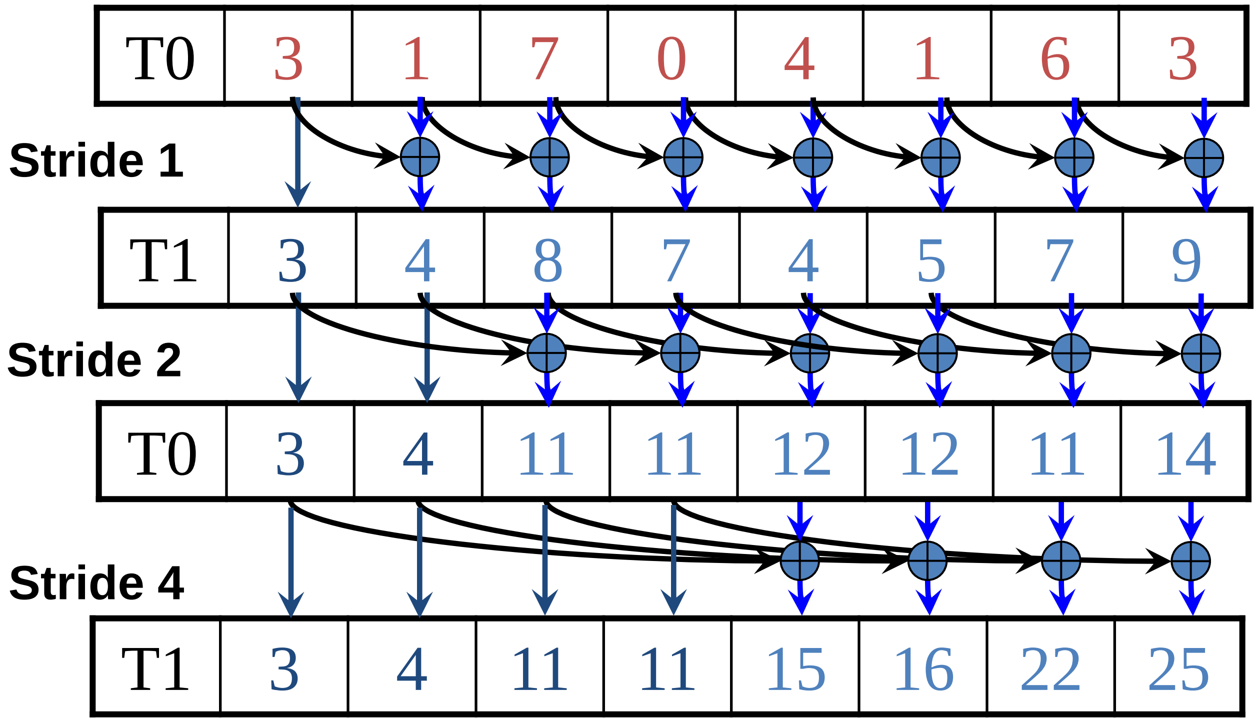


1. Read input from device memory to shared memory.

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2

A Parallel Scan Algorithm



T0 | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3
Stride 1
T1 | 3 | 4 | 8 | 7 | 4 | 5 | 7 | 9
Stride 2
T0 | 3 | 4 | 11 | 11 | 12 | 12 | 11 | 14
Stride 4
T1 | 3 | 4 | 11 | 11 | 15 | 16 | 22 | 25

Iteration #3
Stride = 4

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 *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared memory arrays)
3. Write output from shared memory to device memory

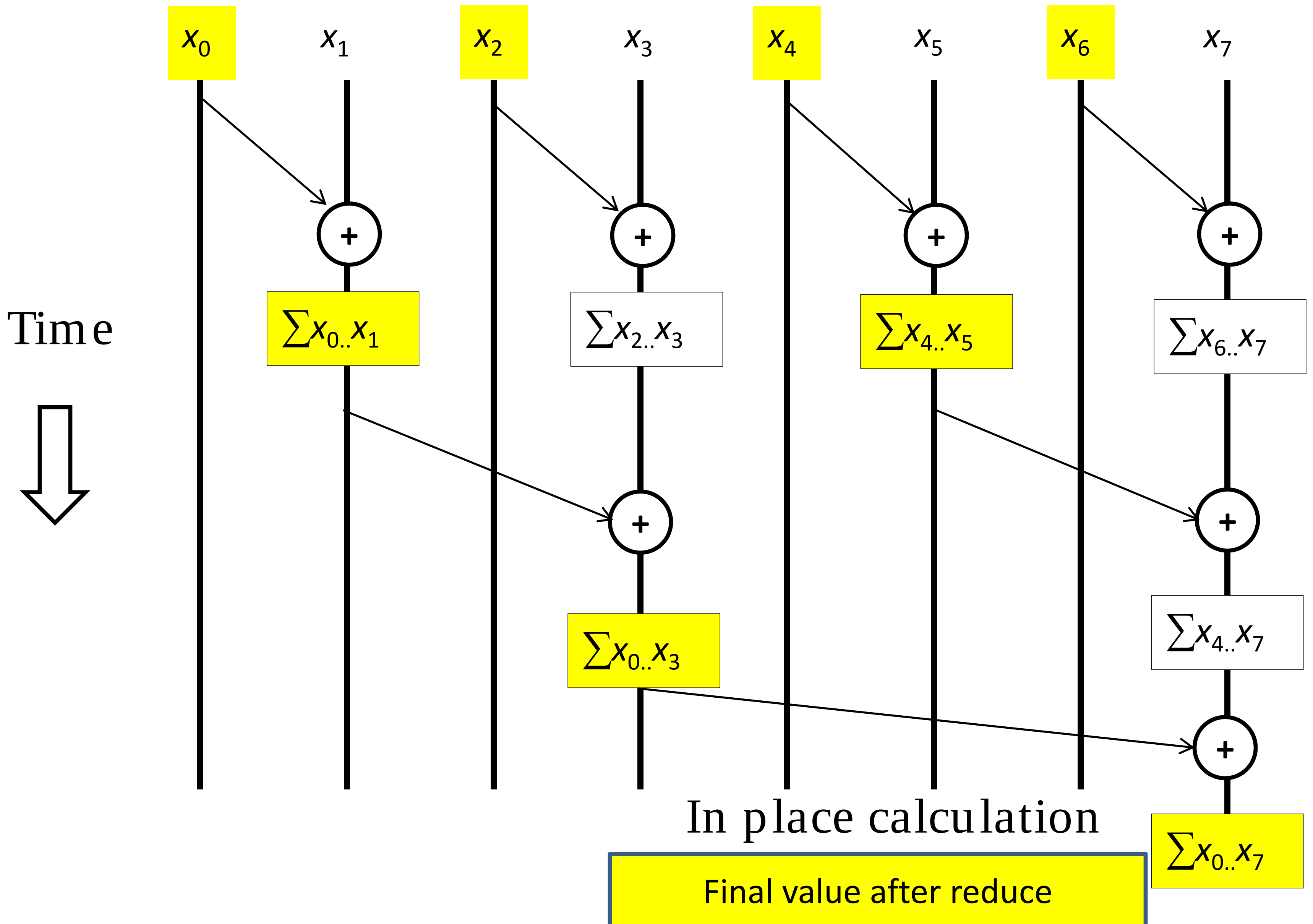
Work Efficiency Considerations

- 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

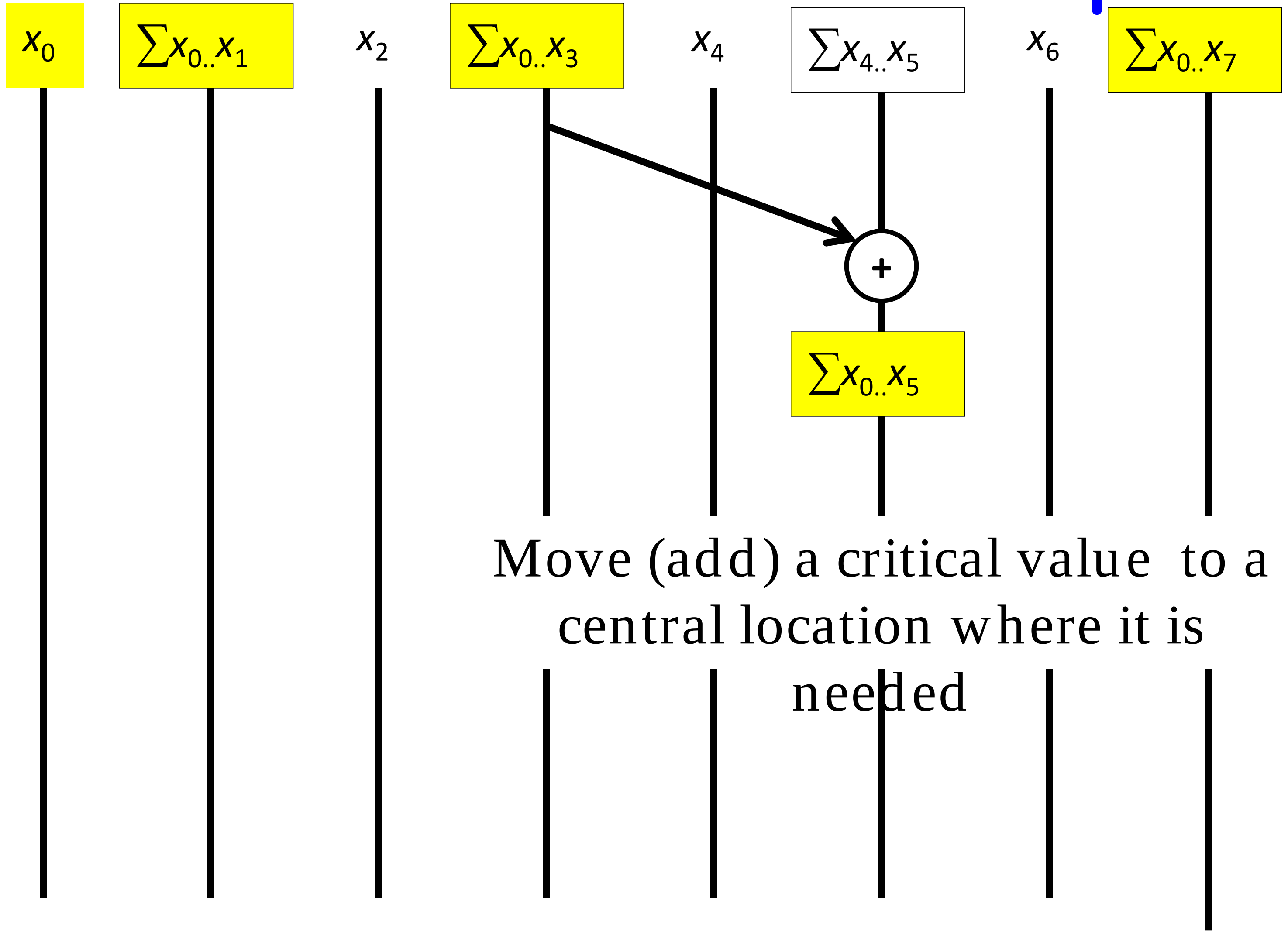
Parallel Scan - Reduction Step



In place calculation

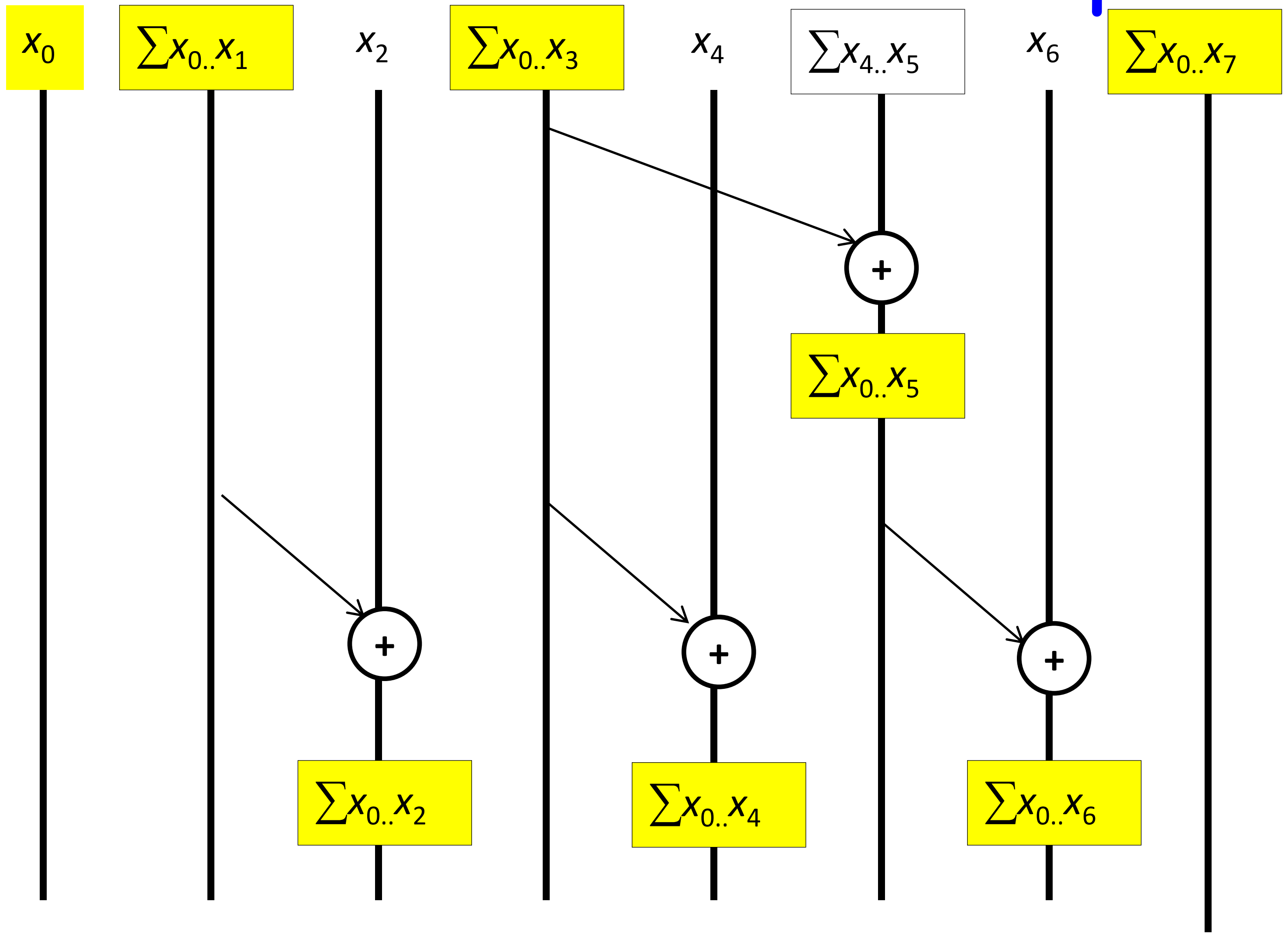
Final value after reduce

Inclusive Post Scan Step

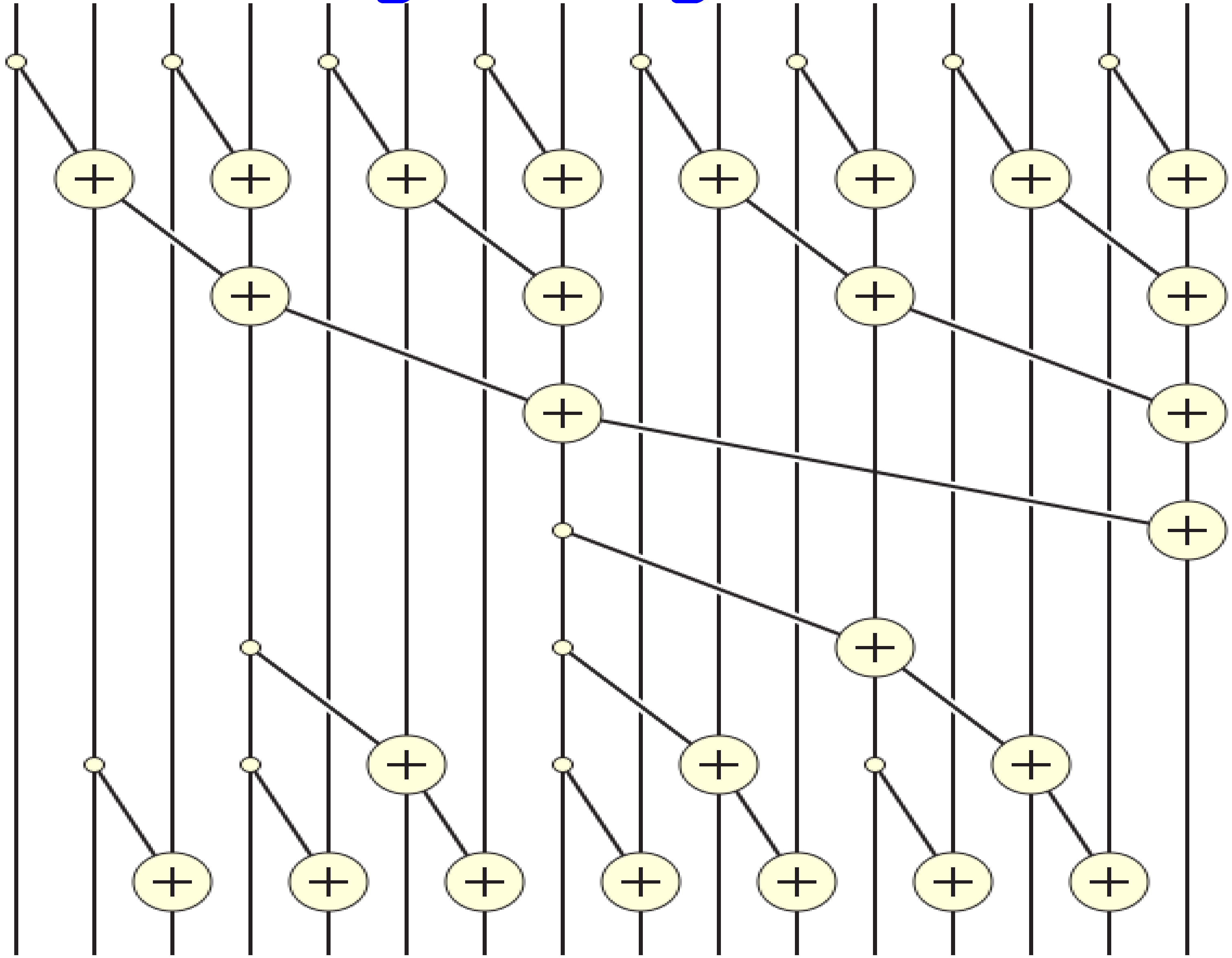


Move (add) a critical value to a central location where it is needed

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) $\rightarrow$ 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?

```
#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;
}
```

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

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.