

PMPP 2015/16

PRAM (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT





(Preliminary) Course Schedule

- 12.10.2015 Introduction to PMPP
- 13.10.2015 Lecture CUDA Programming 1
- 19.10.2015 Lecture CUDA Programming 2
- 20.10.2015 Lecture CUDA Programming 3
- 26.10.2015 Lecture Parallel Basics, [Exercise 1 assigned](#)
- 27.10.2015 Questions and Answers (Q&A), S3|19, Room 2.8
- 2.11.2015 [Intro Final Proj.](#), [Ex. 1 due](#), [Ex. 2 assigned](#), Lecture PRAM
- 3.11.2015 Lecture PRAM (2)
- 9.11.2015 [Final Projects assigned](#), Lecture, [Exercise 2 due](#)
- 10.11.2015 Lecture
- 16.11.2015 Questions and Answers (Q&A)
- 17.11.2015 Questions and Answers (Q&A)
- 23.11.2015 [1st Status Presentation Final Projects](#)
- 24.11.2015 [1st Status Presentation Final Projects \(continued\)](#)

you are here



Exercise 2 – Gaussian Image Filter

Programming Massively Parallel Processors Exercise 2

TU Darmstadt, WS 2015

Daniel Thul, Max von Buelow <pmpp2015@gris.informatik.tu-darmstadt.de>

Introduction

Your task is to implement a Gaussian filter to blur images. The implementation will consist of several CUDA kernels in order to get accustomed with the performance difference of the GPU's different memory types and caches.

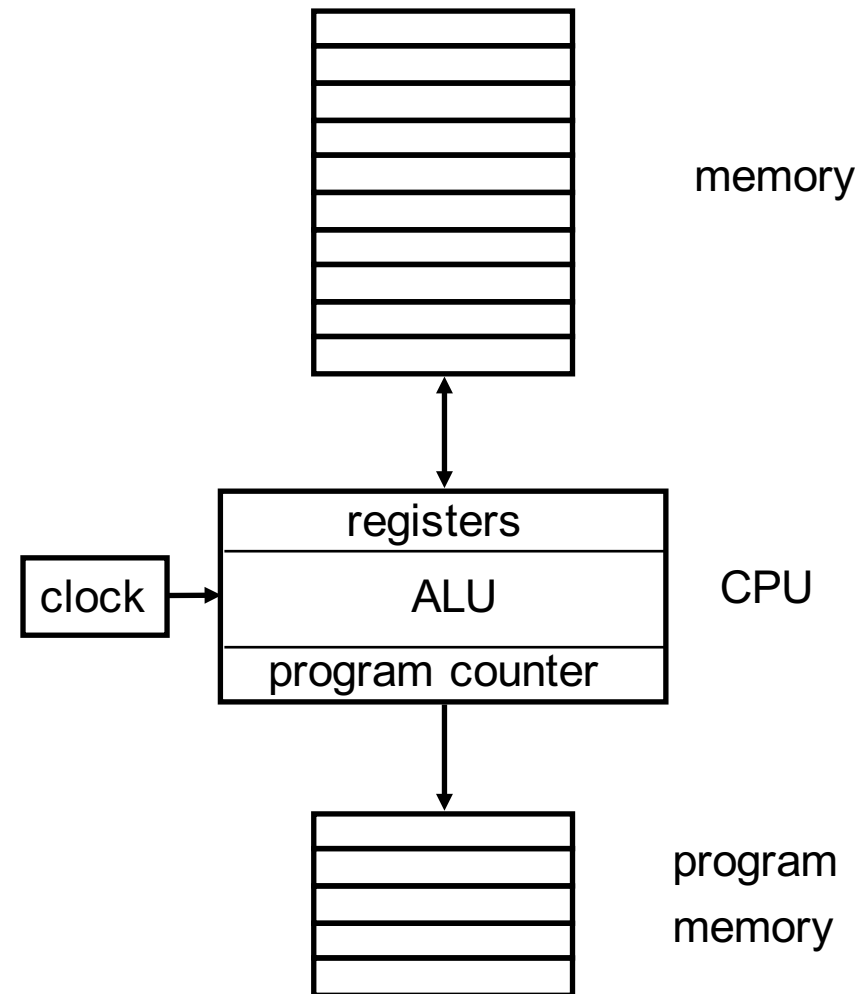
Prerequisites

All coding will be done and graded on the HHLR.

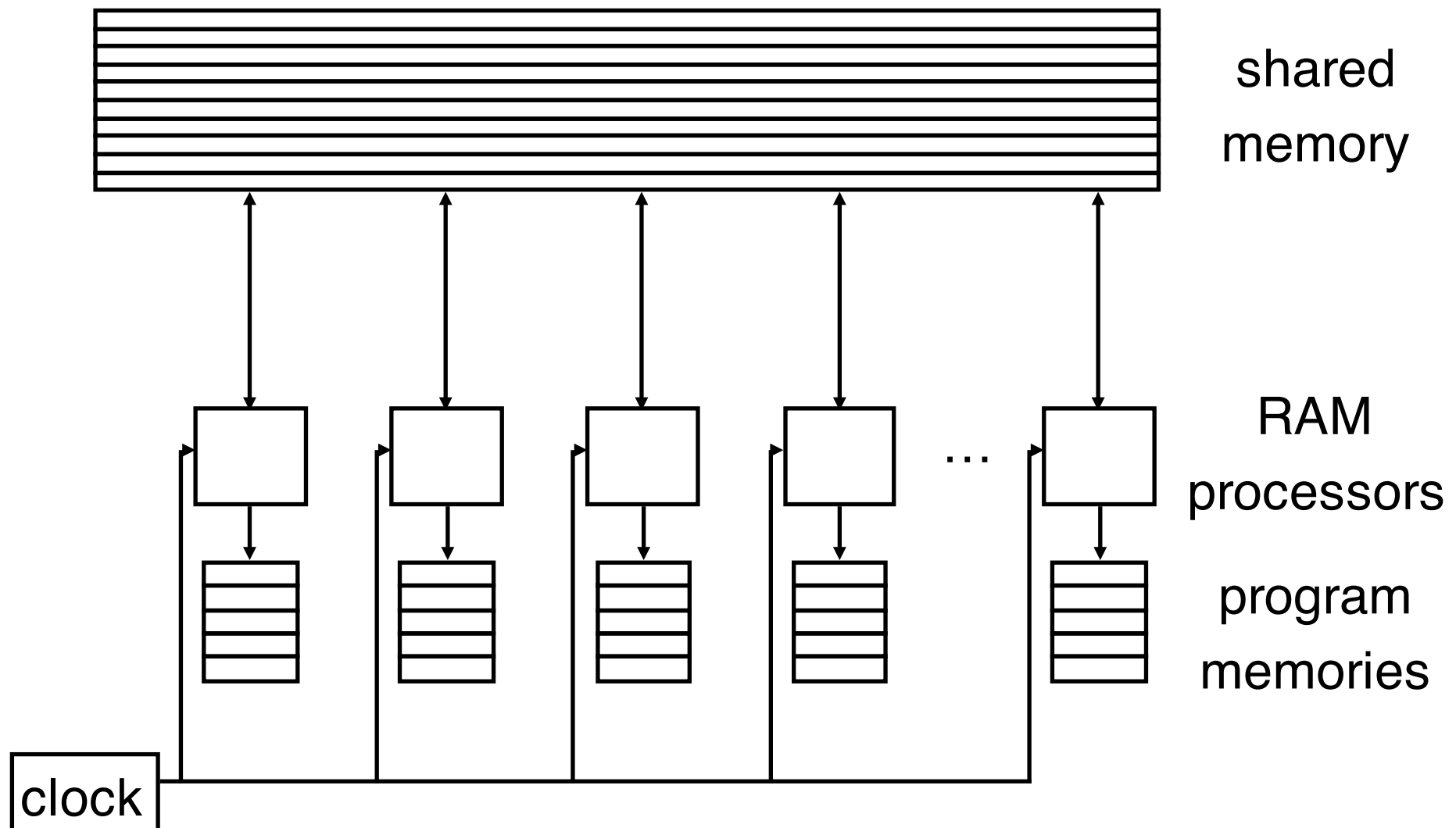
If you want to work with your own computer, you will need a CUDA-capable graphics card and the following

Random Access Machine (RAM)

- von Neumann model
 - program is sequence of instructions
 - memory access instructions
 - arithmetic logical instructions
 - conditional instructions
- sequential computation
- time complexity
 - number of operations as function of size of input (in bits, words, ...)



Parallel Random Access Machine (PRAM)



Parallel Random Access Machine (PRAM)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

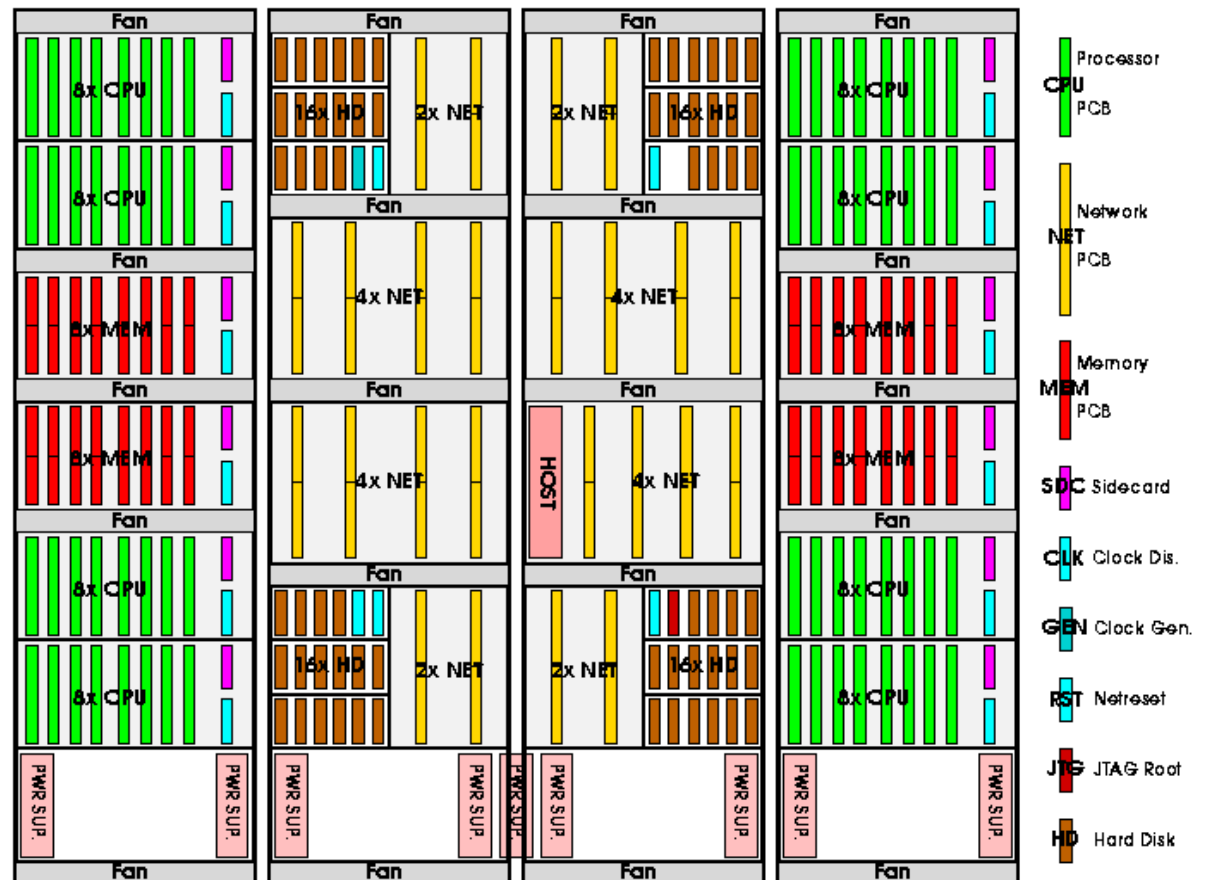
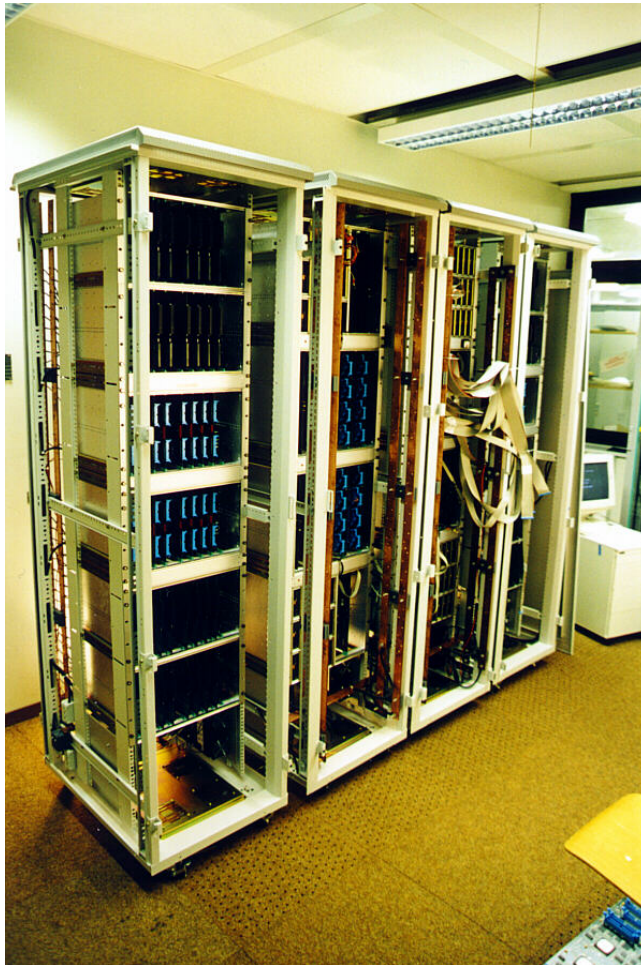


image from SB-PRAM web page



Parallel Random Access Machine (PRAM)

Some hardware approximations of the PRAM do exist, and some experience with the implementation of PRAM algorithms has been gathered, but the exact practical value of the PRAM is still a subject of controversy.

Joerg Keller, Christoph Kessler, Jesper Larsson Traeff
Practical PRAM Programming, 2001

Parallel Work and Parallel Time

- parallel work of an algorithm A on an input with size n
 - maximum number of instructions performed by all processors during execution of A
 - as many parallel processors as needed available in each (parallel) time step
 - each step executed in constant time
 - denoted $w_A(n)$
- parallel time of an algorithm A on an input with size n
 - maximum number of parallel time steps required under same conditions as above
 - denoted $t_A(n)$

PRAM Algorithms

- finding the maximum
- prefix-sum computation
- list ranking

Finding the Maximum

- given an array of n numbers x_0, \dots, x_{n-1}
- goal: find the maximum, i.e., the index m so that $x_m \geq x_i$ for all i in $[0, \dots, n-1]$ on a PRAM

Finding the Maximum

- given an array of n numbers x_0, \dots, x_{n-1}
- goal: find the maximum, i.e., the index m so that $x_m \geq x_i$ for all i in $[0, \dots, n-1]$ on a PRAM **in constant time**
- general idea
 - use n^2 processors to compare each pair in parallel
 - write intermediate result in shared array
 - return index of one maximum (of possibly multiple maxima)
- time complexity $O(1)$
- relative speedup $O(n) = O(n)/O(1)$
- but bad efficiency $O(1/n)$





PRAM Code

```
int maximum (sh int x[], sh int n, sh int c[])
{
    int i, j;

    i = ID / n; j = ID % n; // compute i and j
    if (i == j) c[i] = 1;    // initialize c
    if (x[i] < x[j])
        c[i] = 0;           // i cannot be maximum
                                // common write

    if (i == j && c[i] == 1)
        m = i;              // write index of maximum
                                // arbitrary write

    return m;
}
```



Finding the Maximum (2)

- given an array of n numbers x_0, \dots, x_{n-1}
- goal: find the maximum, i.e., the index m so that $x_m \geq x_i$ for all i in $[0, \dots, n-1]$
- alternative solution: reduction
 - always compute maximum of 2 array elements
 - tree with logarithmic depth
- time complexity $O(\log n)$
- relative speedup $O(n)/O(\log n)$
- efficiency $O(1/\log n)$

Finding the Maximum using Reduction

- problem with reduction
 - number of active processors reduced by half in each step
 - soon many idle processors

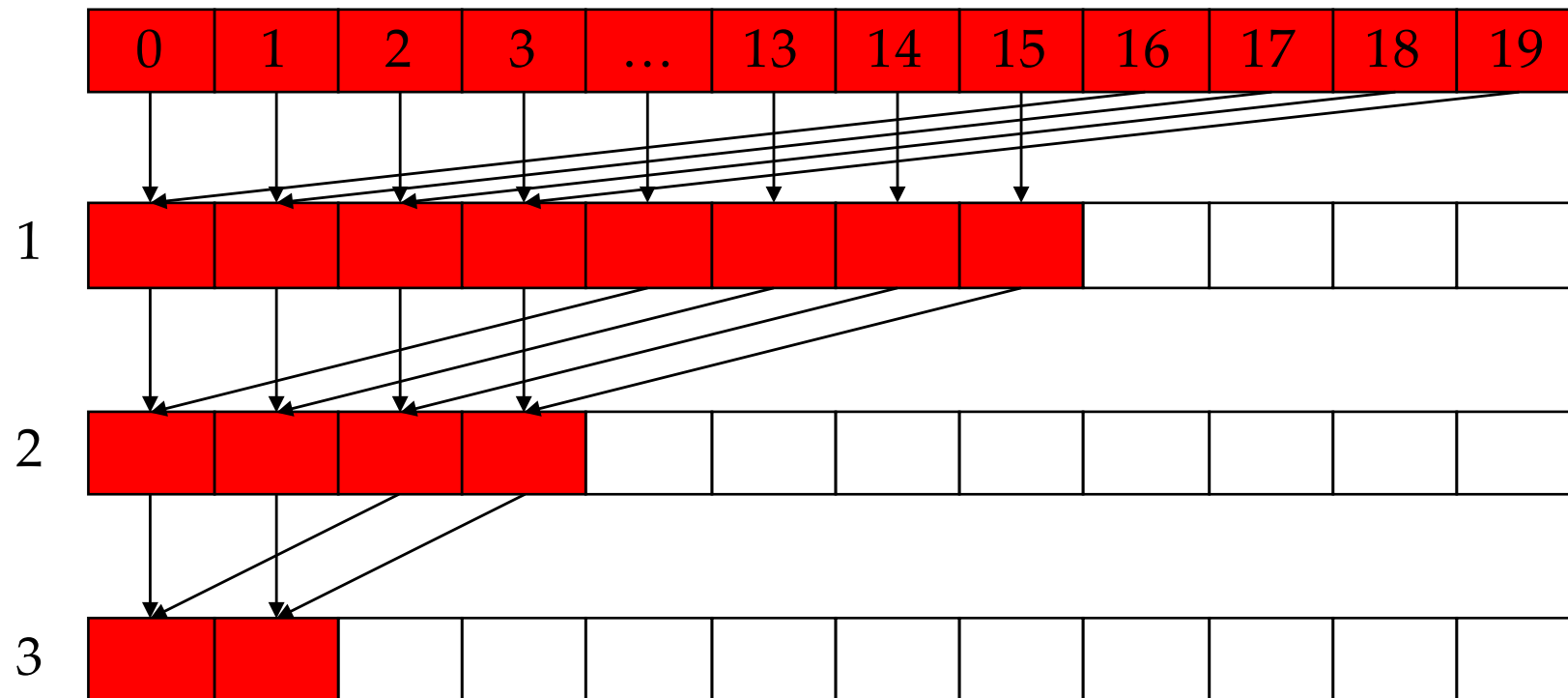


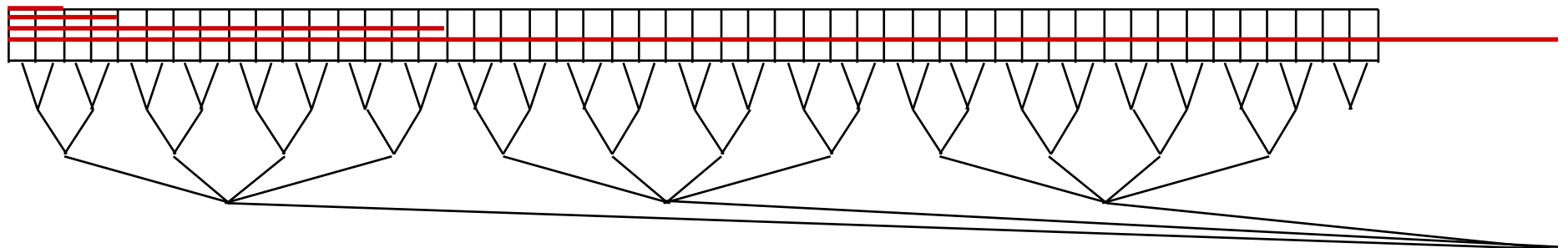
image source: NVIDIA

Finding the Maximum (3)

- $O(\log \log n)$ time complexity approach
 - Shiloach and Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. J. Algorithms, 2:88-102, 1981.
- general idea: accelerated cascading
 - use a cost optimal parallel algorithm to reduce the problem in size
 - use the fast but non-optimal algorithm to compute the solution from the reduced problem
 - use a cost-optimal algorithm to compute a solution to the original problem from the solution to the reduced problem

Finding the Maximum (3)

- start with 1 processor per array element
- standard reduction in iteration 1 and 2
 - combine pairs of array elements
- find maximum of 4 elements in iteration 3
 - $4 \cdot 2 \cdot 2 = 16 = 4^2$ processors available
- find maximum of 16 elements in iteration 4
 - $16 \cdot 4 \cdot 2 \cdot 2 = 256 = 16^2$ processors available
- yields doubly-logarithmic tree
- uses all processors starting in iteration 3



Parallel Prefix-Sum (Scan)

- Definition:

The all-prefix-sums operation takes a binary associative operator \oplus with identity I , and an array of n elements

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})].$$

- Example:

if \oplus is addition, then scan on the set

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

returns the set

$$[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$$

from Blelloch, 1990, “Prefix Sums and Their Applications”



Applications of Scan

- scan is a simple and useful parallel building block

- Convert recurrences from sequential :

```
for (j = 1; j < n; j++)  
    out[j] = out[j - 1] + f(j);
```

- into parallel:

```
forall(j) { temp[j] = f(j); }  
scan(out, tmp);
```

- useful for many parallel algorithms:

- radix sort
 - quicksort
 - string comparison
 - lexical analysis
 - stream compaction
 - ...

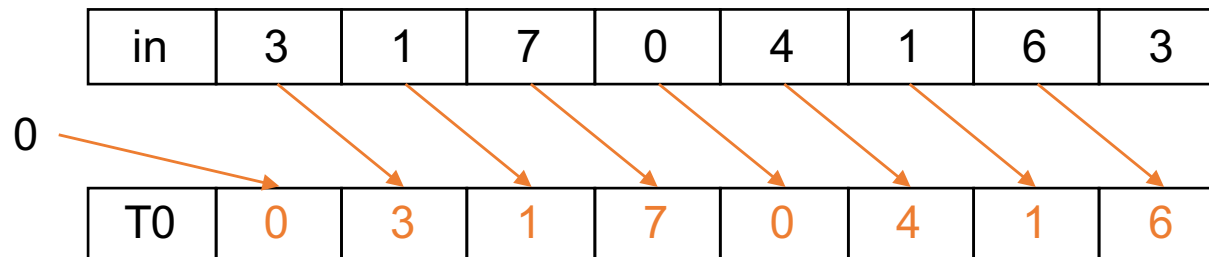


Prefix-Sum on the CPU

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

- add each element to the sum of the elements before it
- trivial but sequential
- exactly n adds: optimal in terms of work efficiency

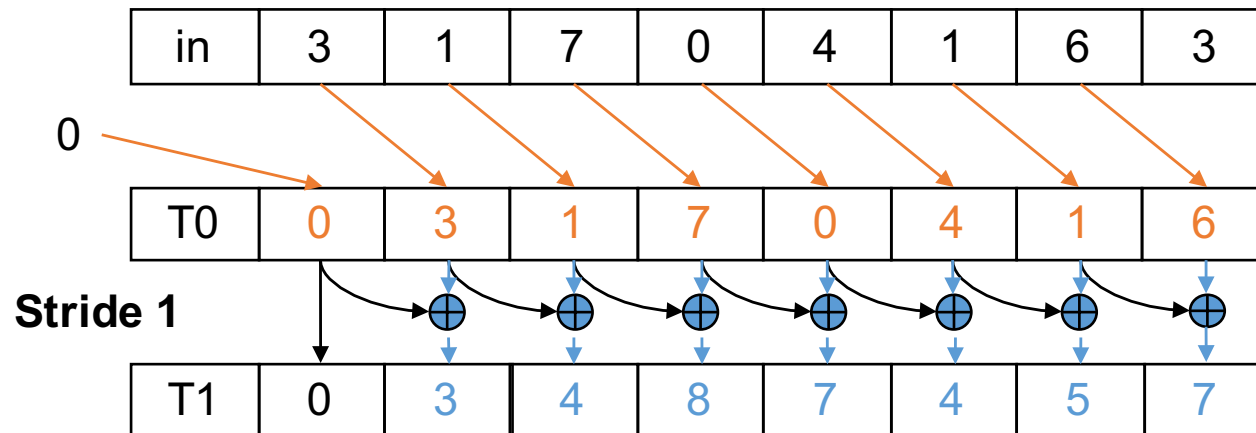
Simple Parallel Prefix-Sum



Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

Simple Parallel Prefix-Sum

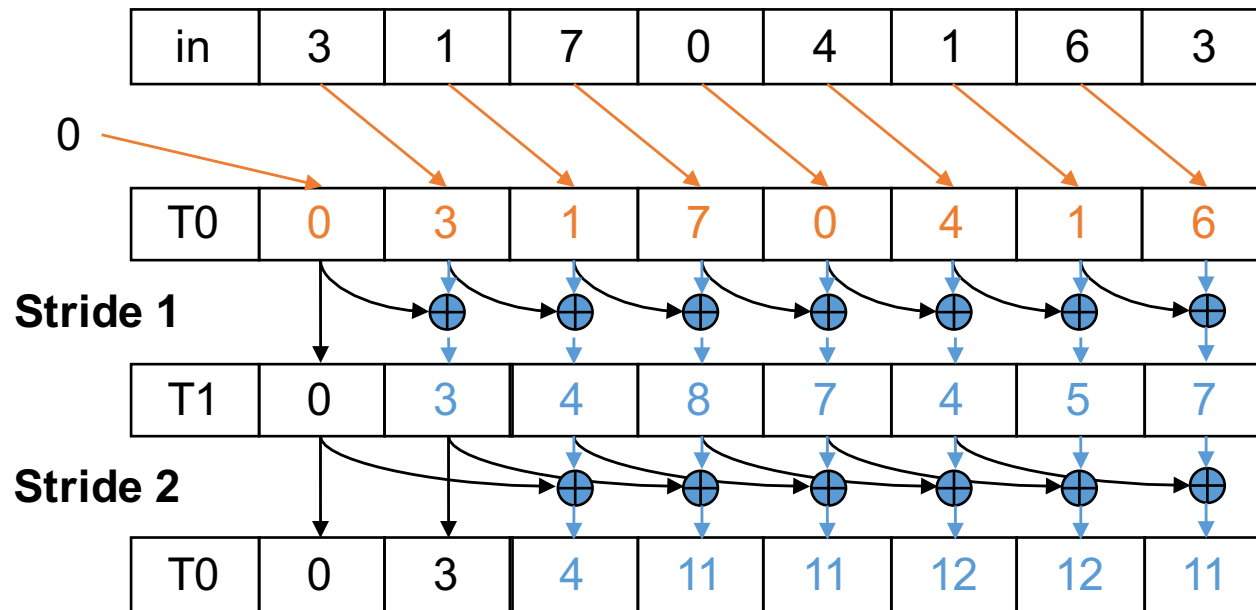


- Active threads: *stride* to $n-1$ (n -*stride* threads)
- Thread j adds elements j and j -*stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

Iteration #1
Stride = 1

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* each iteration. (note: must double buffer shared mem arrays)

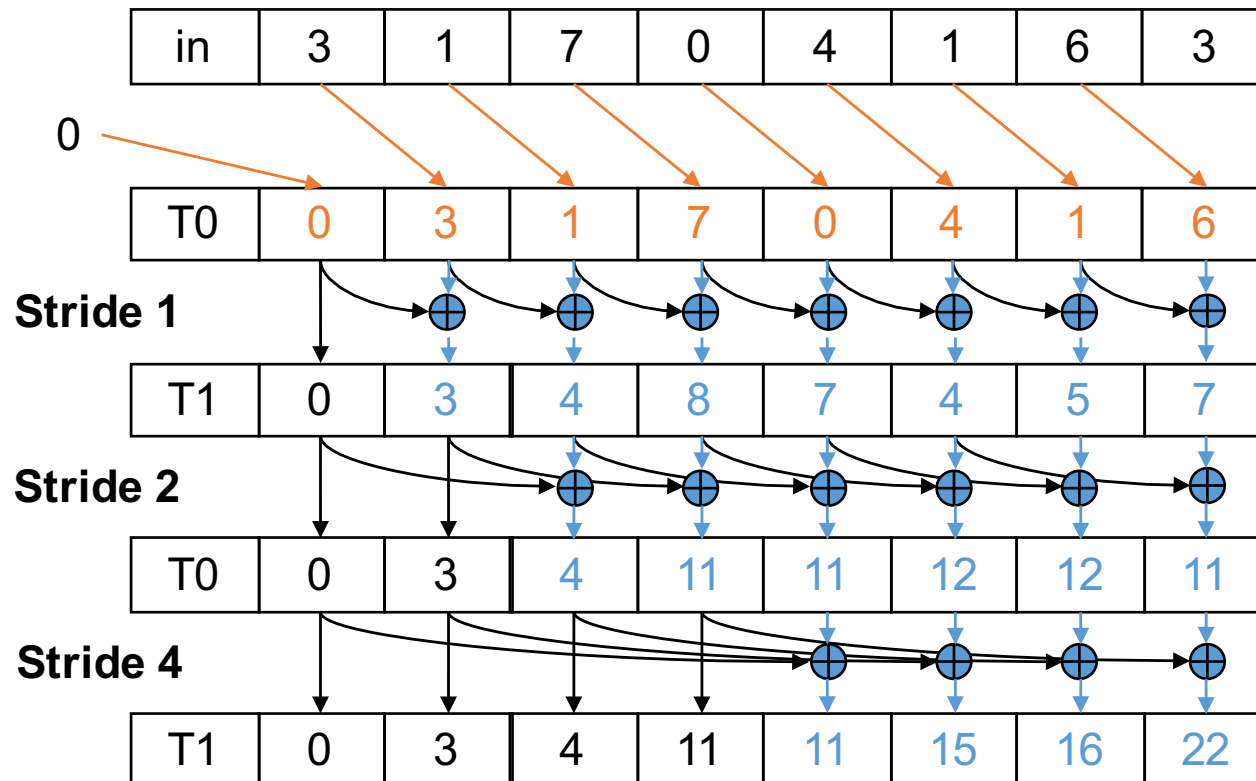
Simple Parallel Prefix-Sum



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* each iteration. (note: must double buffer shared mem arrays)

Iteration #2
Stride = 2

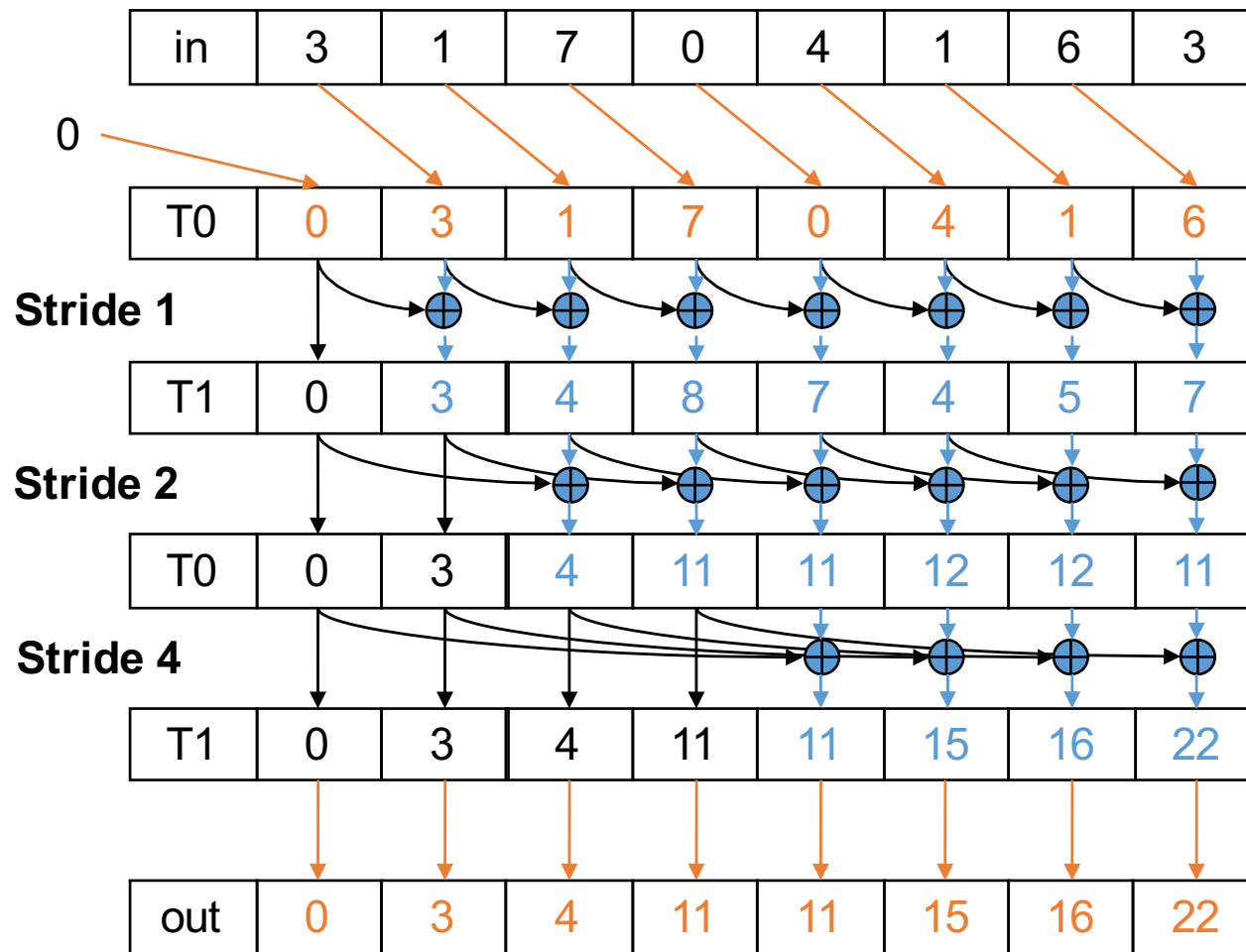
Simple Parallel Prefix-Sum



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* each iteration. (note: must double buffer shared mem arrays)

Simple Parallel Prefix-Sum



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* each iteration. (note: must double buffer shared mem arrays)
3. Write output to device memory.

Work Efficiency Considerations

- the first-attempt parallel prefix-sum executes $\log(n)$ parallel iterations
 - the steps do $(n - 1, n - 2, n - 4, \dots, n - 2^{(\text{stride} - 1)})$ adds each
 - total adds: $n * (\log(n) - 1) + 1 \rightarrow O(n * \log(n))$ work
 - this algorithm is not very work efficient
 - sequential prefix-sum algorithm does n adds
 - factor of $\log(n)$ hurts: 20x for 10^6 elements!
- ➔ parallel algorithm can be slow when execution resources are saturated due to low work efficiency

Improving Efficiency

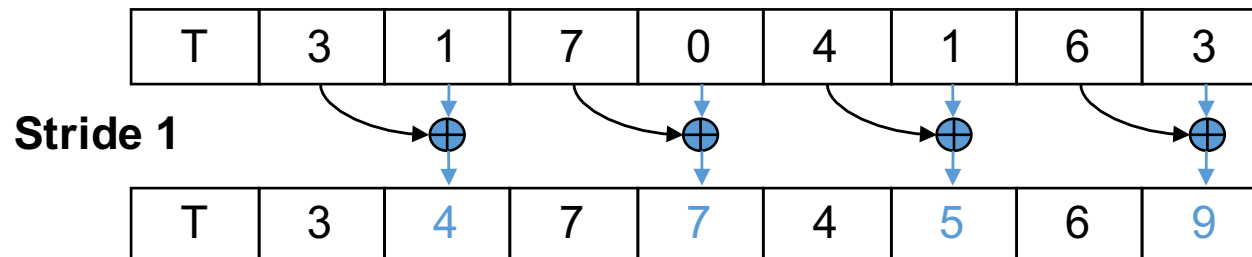
- common parallel algorithm pattern: balanced trees
 - build a balanced binary tree on the input data and sweep it to and from the root
 - tree is not an actual data structure, but a concept to determine what each thread does at each step
- for prefix-sum
 - traverse down from leaves to root building 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

Build the Sum Tree


T	3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---	---

Assume array T is already in shared memory.

Build the Sum Tree

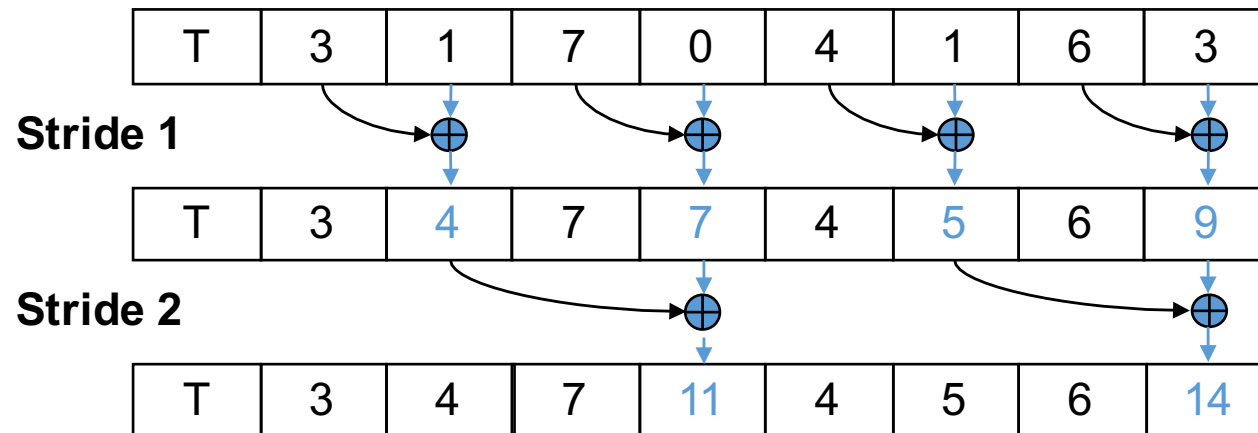


Iteration 1, $n/2$ threads


Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree

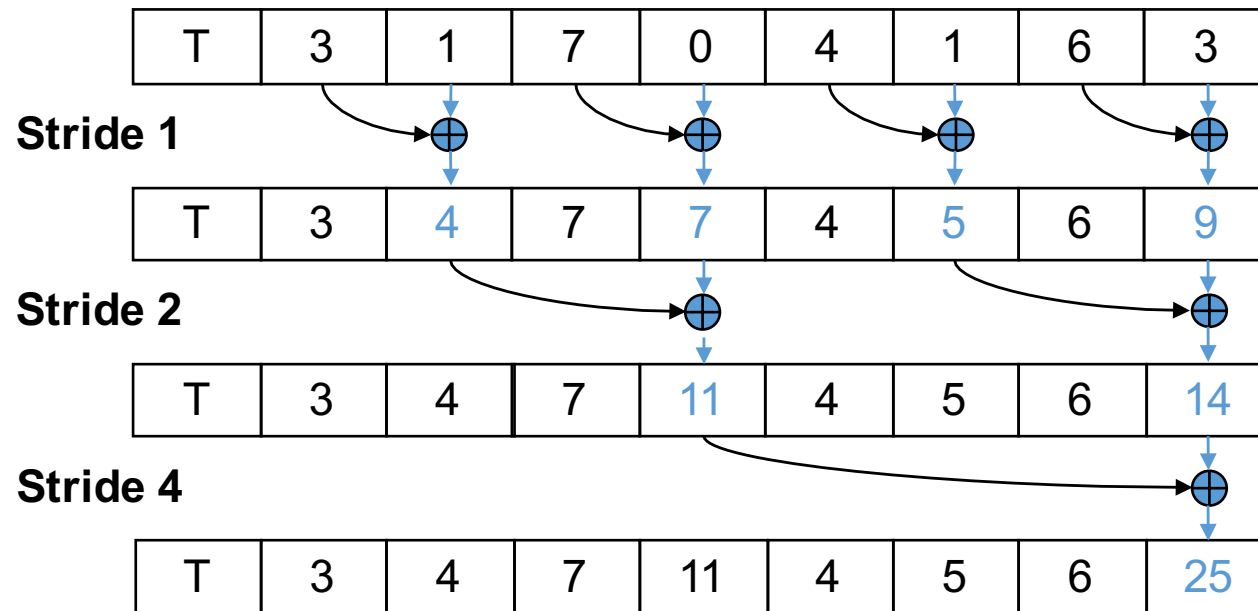


Iteration 2, $n/4$ threads


Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree



Iteration $\log(n)$, 1 thread

Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Prefix-Sum From Partial Sums

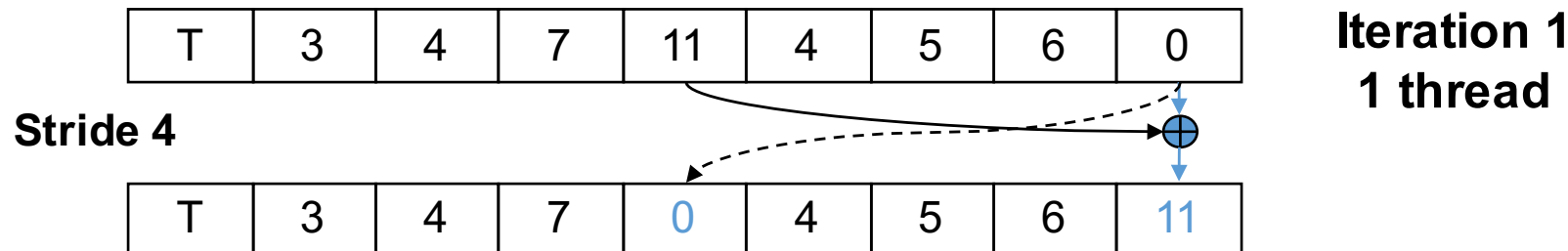


TECHNISCHE
UNIVERSITÄT
DARMSTADT

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---



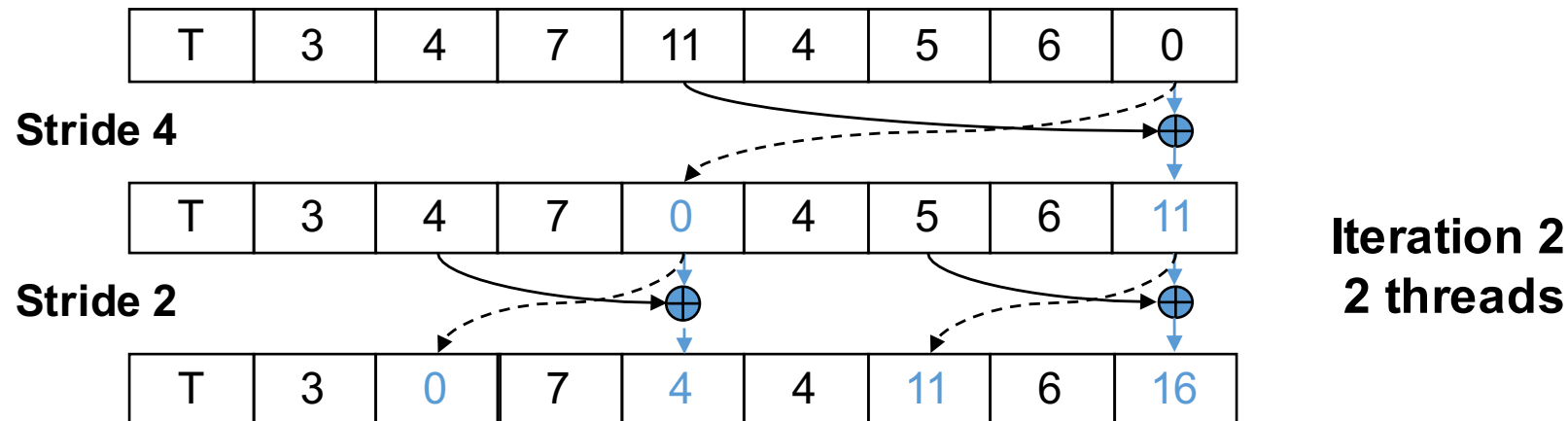
Build Prefix-Sum From Partial Sums




Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

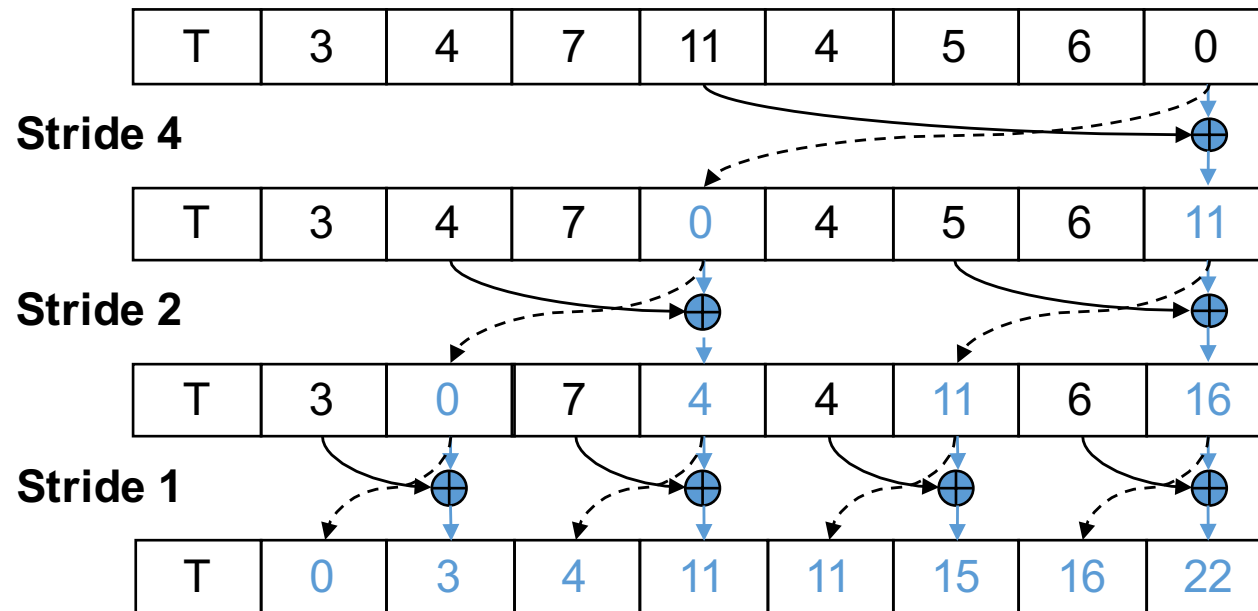
Build Prefix-Sum From Partial Sums




Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Prefix-Sum From Partial Sums



Iteration $\log(n)$
 $n/2$ threads

Each  corresponds to a single thread.

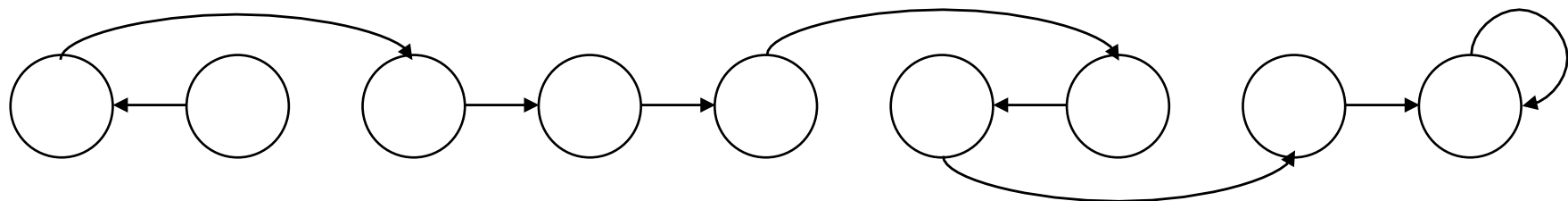
Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 * \log(n)$.

Total work: $2 * (n-1)$ adds = $O(n)$ **Work Efficient!**

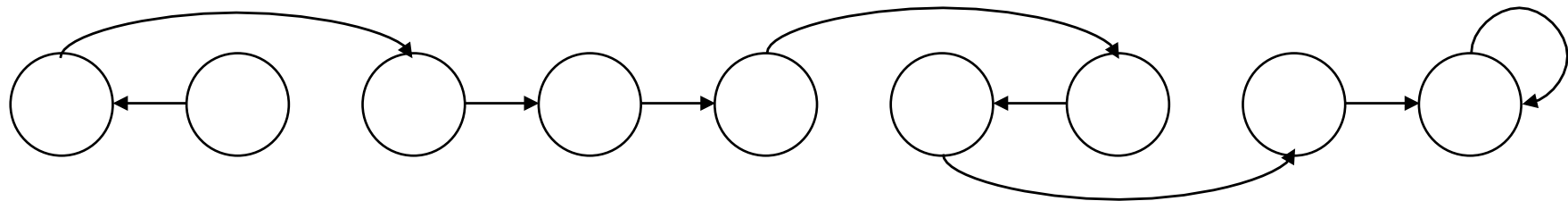
List Ranking

- data stored in a list of elements instead of an array
 - list itself can be stored in an array of elements
 - each element contains pointer to next element $N[i]$ (plus optionally some data)
 - order not predefined by position in the array
- can we compute the prefix-sum on this list?



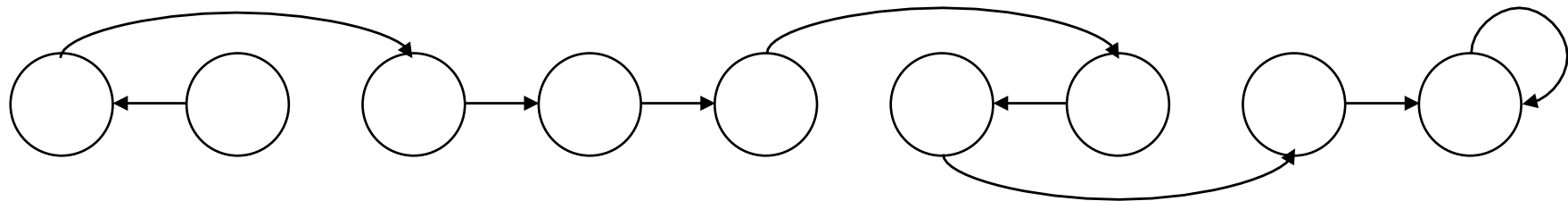
List Ranking

- rank of a list element = distance from the last element
- general approach for prefix-sum on lists
 - compute rank of each element
 - copy elements to an array with element i at position i
 - perform standard prefix-sum calculation
- problem: how to compute rank



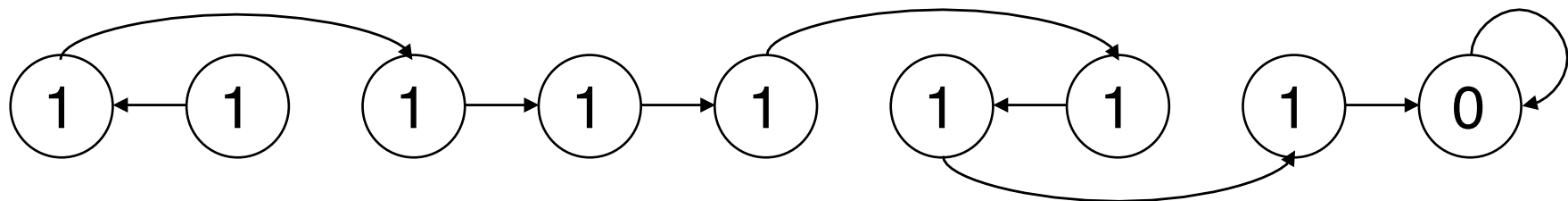
Naïve List Ranking

- start at each list element and follow pointers to end of list
 - maximum of n steps for n list elements
 - slow and work-inefficient

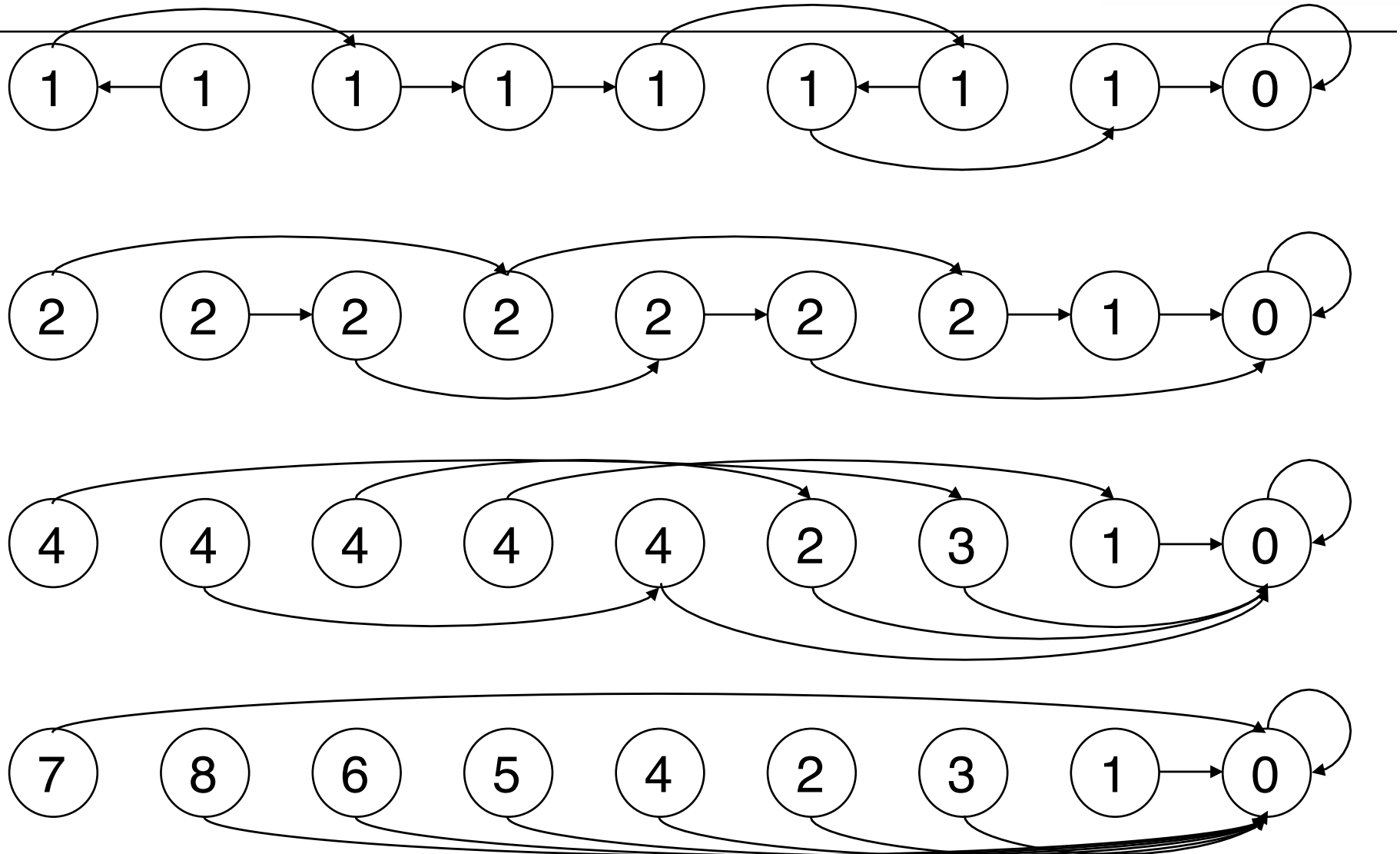


List Ranking with Pointer Jumping

- introduce local rank $R[i]$ for each list element
 - $\text{rank} = \text{local rank} + \text{rank of the element pointed to by } N[i]$
- initialization
 - $R[i] = 1$ for all elements except for the last element
 - $N[i]$ points to successor in list
 - $N[\text{last element}]$ points to itself



List Ranking with Pointer Jumping

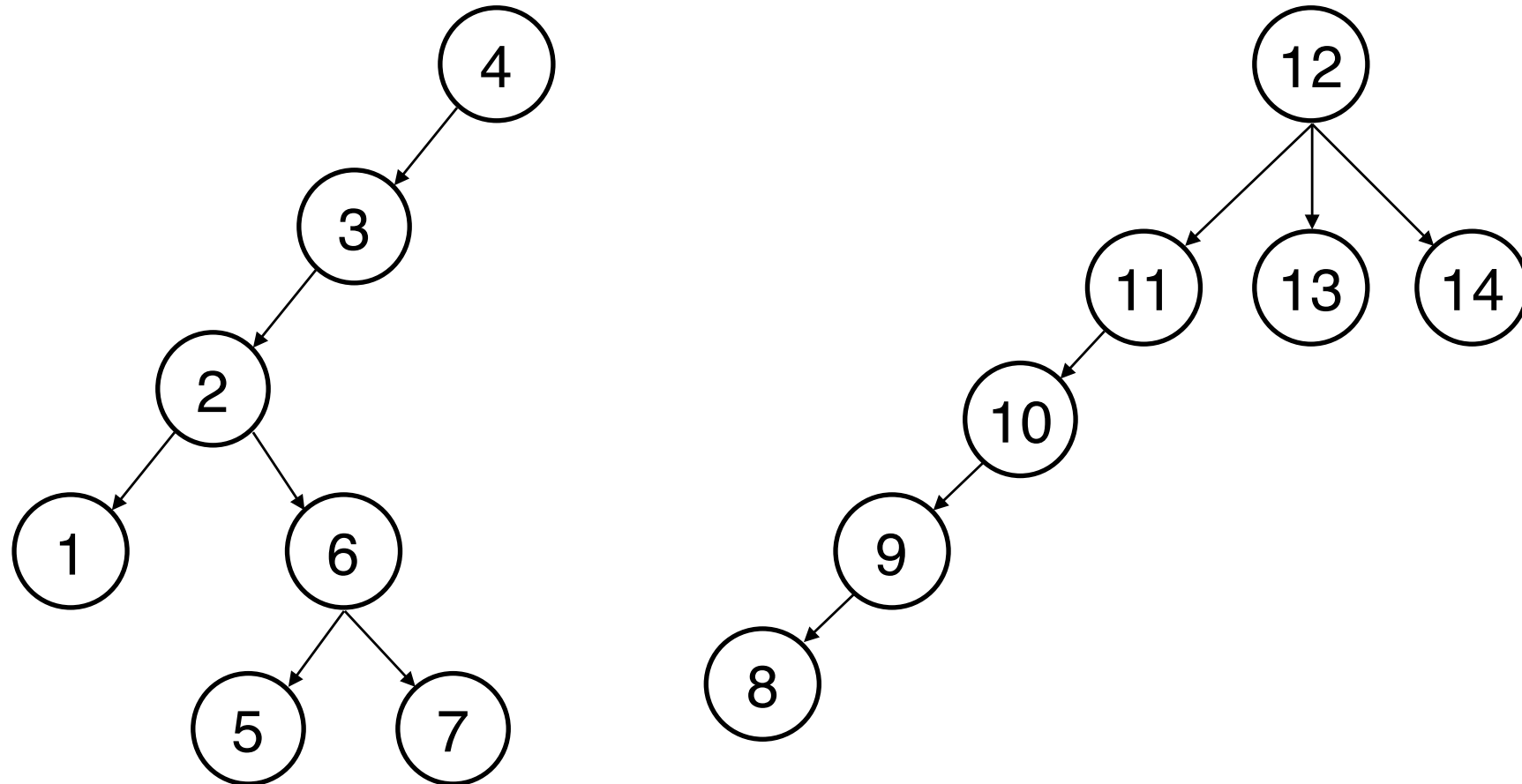


List Ranking with Pointer Jumping

- termination after $k = \lceil \log_2 n \rceil$ steps for a list with n elements
- Wyllie's algorithm after [Wyllie 1979]
- principal technique also called path doubling
 - applicable to other problems as well
- caveat
 - algorithm assumes that list elements are compactly stored in an array
 - otherwise list marking required to determine storage locations of list elements

Related Problem: Root Finding

- find the root for all trees in a forest of directed trees



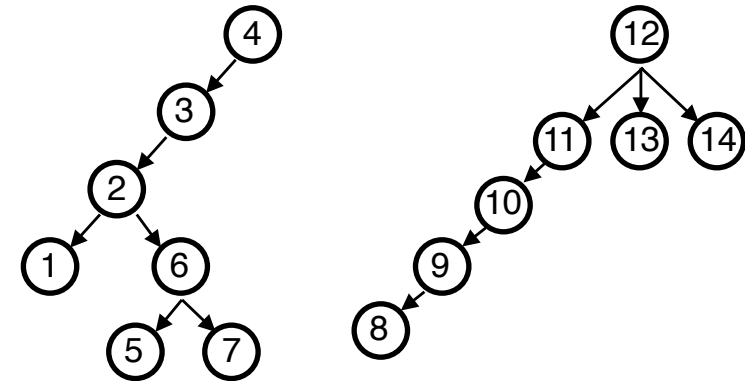
Related Problem: Root Finding

- find the root for all trees in a forest of directed trees

- inherently sequential operation
- follow path from the root to all nodes

- reformulation

- define a successor for each node
- initially parent
- ultimately root
- repeat until convergence



- increases complexity $O(N)$ to $O(N \log N)$
- reduces run time from $O(N)$ to $O(\log N)$

Literature on PRAM Algorithms

- Practical PRAM Programming
Jörg Keller, Christoph W. Kessler, Jesper Larsson Träff
Wiley Series on Parallel and Distributed Computing, 2001
 - unfortunately out of print
- Synthesis of Parallel Algorithms, J. H. Reif
Morgan Kaufmann Publishers, 1993
- An Introduction to Parallel Algorithms, J. Jàjà
Addison-Wesley, 1992

Recommended Reading

- presentation on parallel reduction using CUDA by Mark Harris
 - step-by-step discussion of optimizations to create a high-performance implementation of reduction
 - available in the doc directory of the reduction SDK example

