Parallel Programming

Prefix sum (scan)

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

Overview

- The “scan” task
- Sequential implementation
- Parallel implementation
  - Kernel 1: work - inefficient
  - Kernel 2: work - efficient

The “scan” task

in	**1**	**9**	**5**	**1**	**6**	**4**	**7**	**2**
out								

Scan trong

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2

out	1							

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2
out	1	10						

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2
out	1	10	15					

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2

out	1	10	15	16				

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	**1**	**9**	**5**	**1**	**6**	**4**	**7**	**2**
out	**1**	**10**	**15**	**16**	**22**			

- **In**clusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The "scan" task

in	1	9	5	1	6	4	7	2
out	1	10	15	16	22	26		

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2
out	1	10	15	16	22	26	33	

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The “scan” task

in	1	9	5	1	6	4	7	2
out	1	10	15	16	22	26	33	35

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

The "scan" task

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

in	**1**	**9**	**5**	**1**	**6**	**4**	**7**	**2**
out	**1**	**10**	**15**	**16**	**22**	**26**	**33**	**35**

Scan loại trừ

- **Exclusive scan**: $out[0] = 0, out[i] = \sum_{j=0}^{i-1} in[j]\ \forall i > 0$

in	**1**	**9**	**5**	**1**	**6**	**4**	**7**	**2**
out								

The "scan" task

- **Inclusive scan**: $out[i] = \sum_{j=0}^{i} in[j]$

in	1	9	5	1	6	4	7	2
out	1	10	15	16	22	26	33	35

- **Exclusive scan**: $out[0] = 0, out[i] = \sum_{j=0}^{i-1} in[j]\ \forall i > 0$

in	1	9	5	1	6	4	7	2
out	0	1	10	15	16	22	26	33

Identity

Phần tử đơn vị

- In addition to plus operation, it can be applied for product, max, min, …

Tập trung vào bài toán này

- Here we will focus on inclusive scan with plus operation

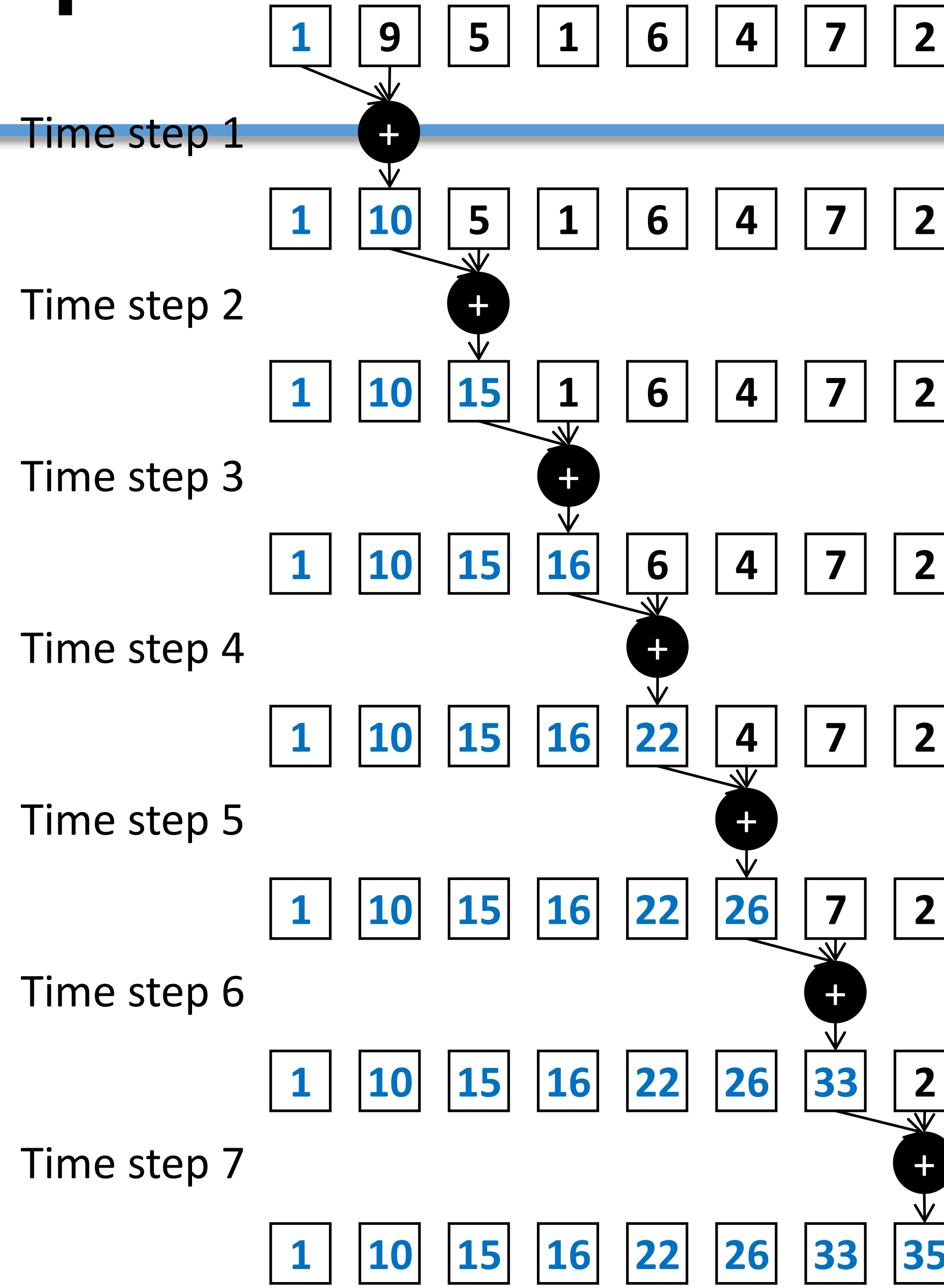
Introduction

- **Parallel scan** is used to parallelize có vẻ ***seemingly sequential operations***:
  - Resource allocation, work assignment, and polynomial evaluation
- Cơ sở A key primitive in many parallel algorithms to convert serial computation (recursion) into parallel computation
  - Radix sort, quick sort, histogram, string comparison,…
- Work efficiency in parallel code/algorithms
  - Parallel algorithms have higher complexity than a sequential algorithm

Sequential implementation

```
void scanOnHost(int *in, int *out, int n)
{
    out[0] = in[0];
    for (int i = 1; i < n; i++)
    {
            out[i] = out[i - 1] + in[i];
    }
}
```

Time (# time steps):
Work (# pluses):

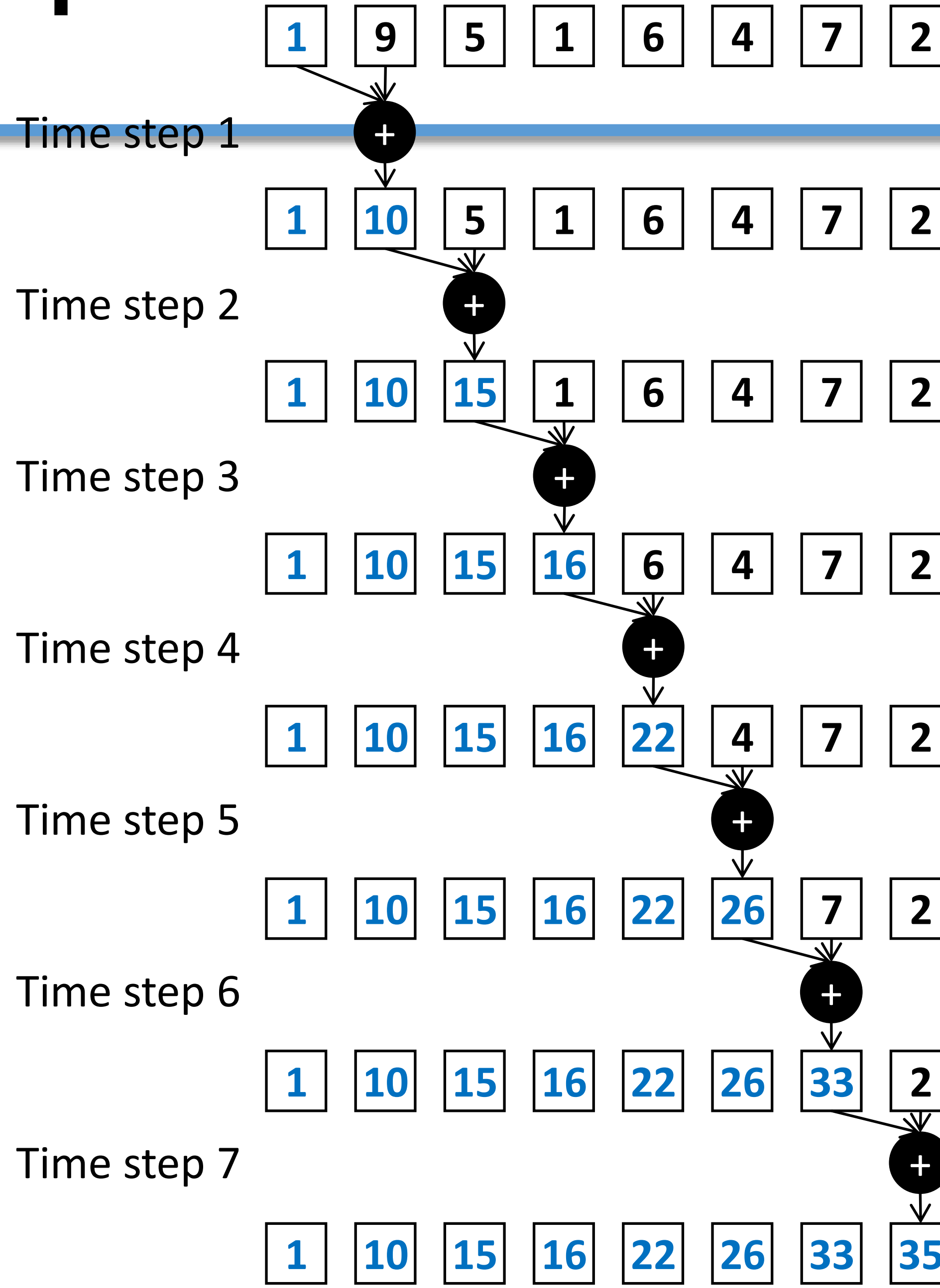


Phần tử cuối cùng là tổng các phần tử trong mảng

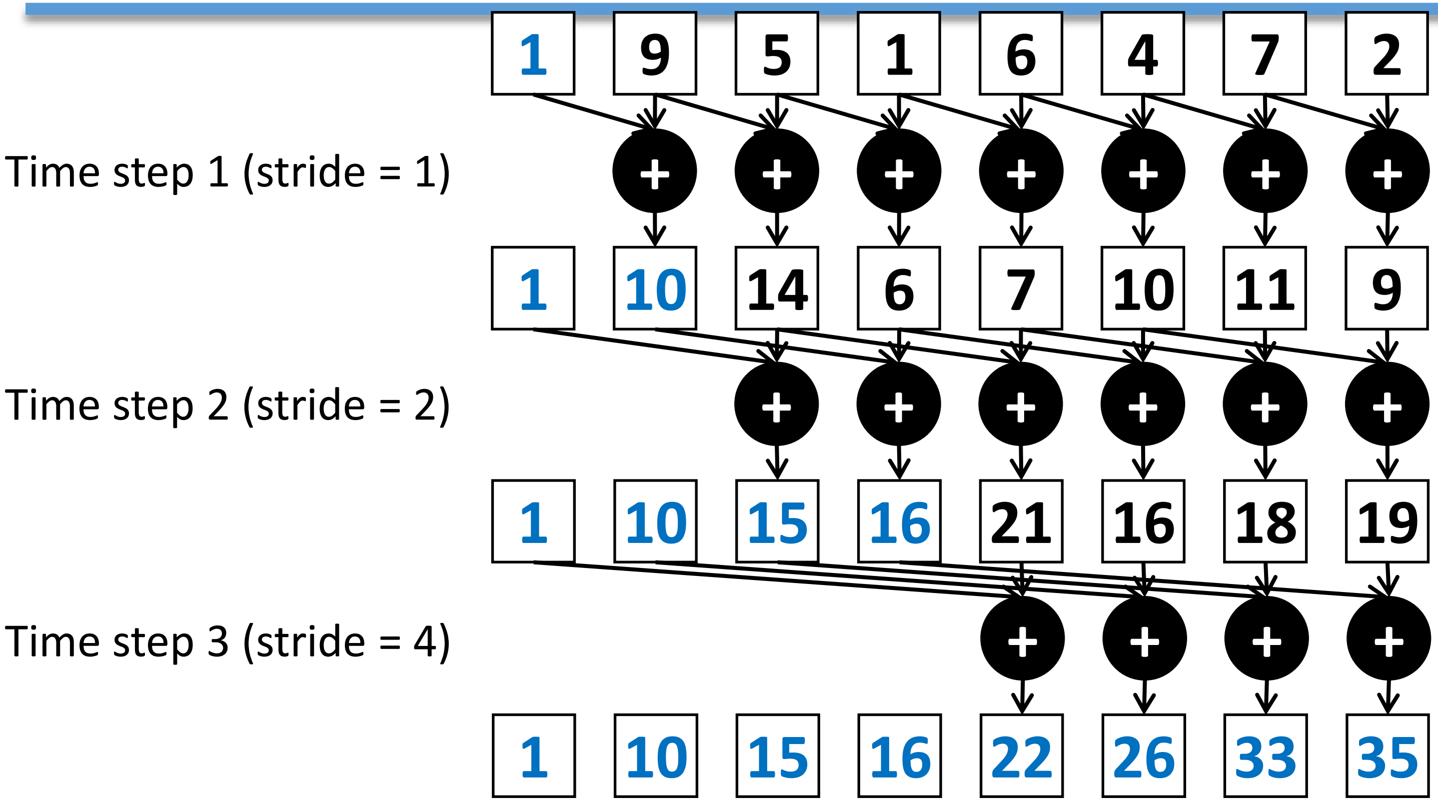
Sequential implementation

```c
void scanOnHost(int *in, int *out, int n)
{
   out[0] = in[0];
   for (int i = 1; i < n; i++)
   {
            out[i] = out[i - 1] + in[i];
   }
}
```

Time (# time steps): 7 = n-1 = O(n)
Work (# pluses): 7 = n-1 = O(n)

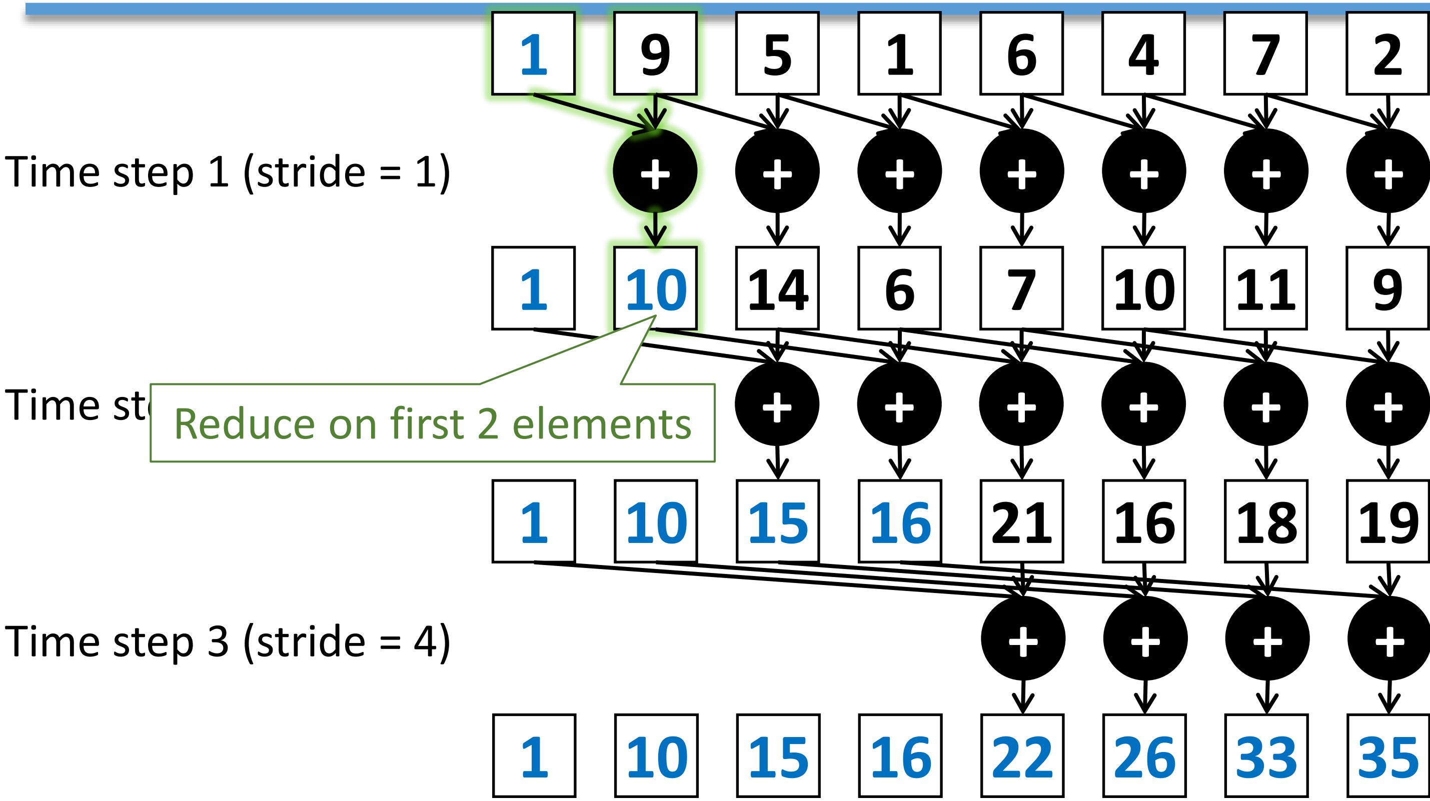


Parallel implementation

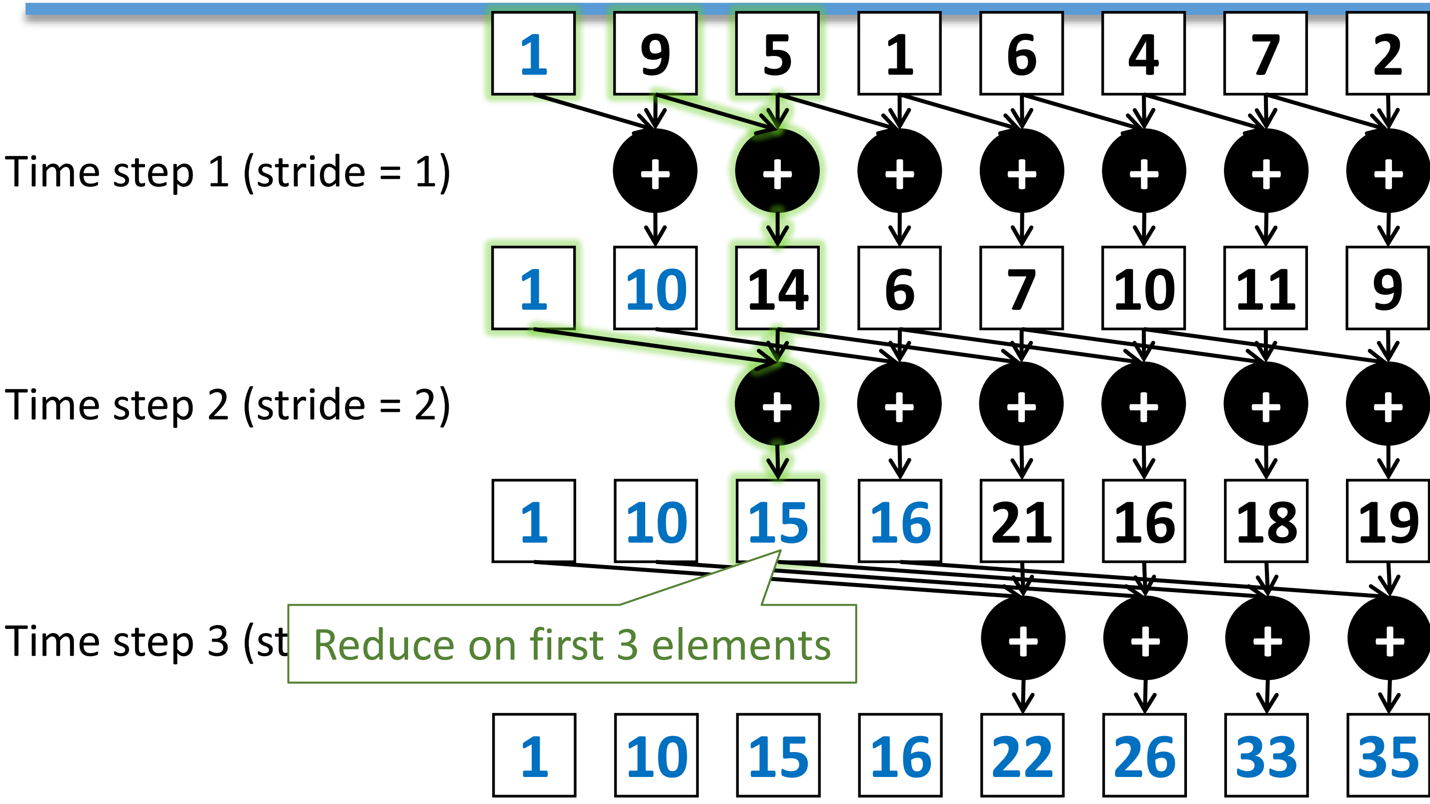


Giá trị của stride cho biết đến bước đó có bao nhiêu phần tử sẽ có đủ giá trị (chỉ tính trong bước đó)

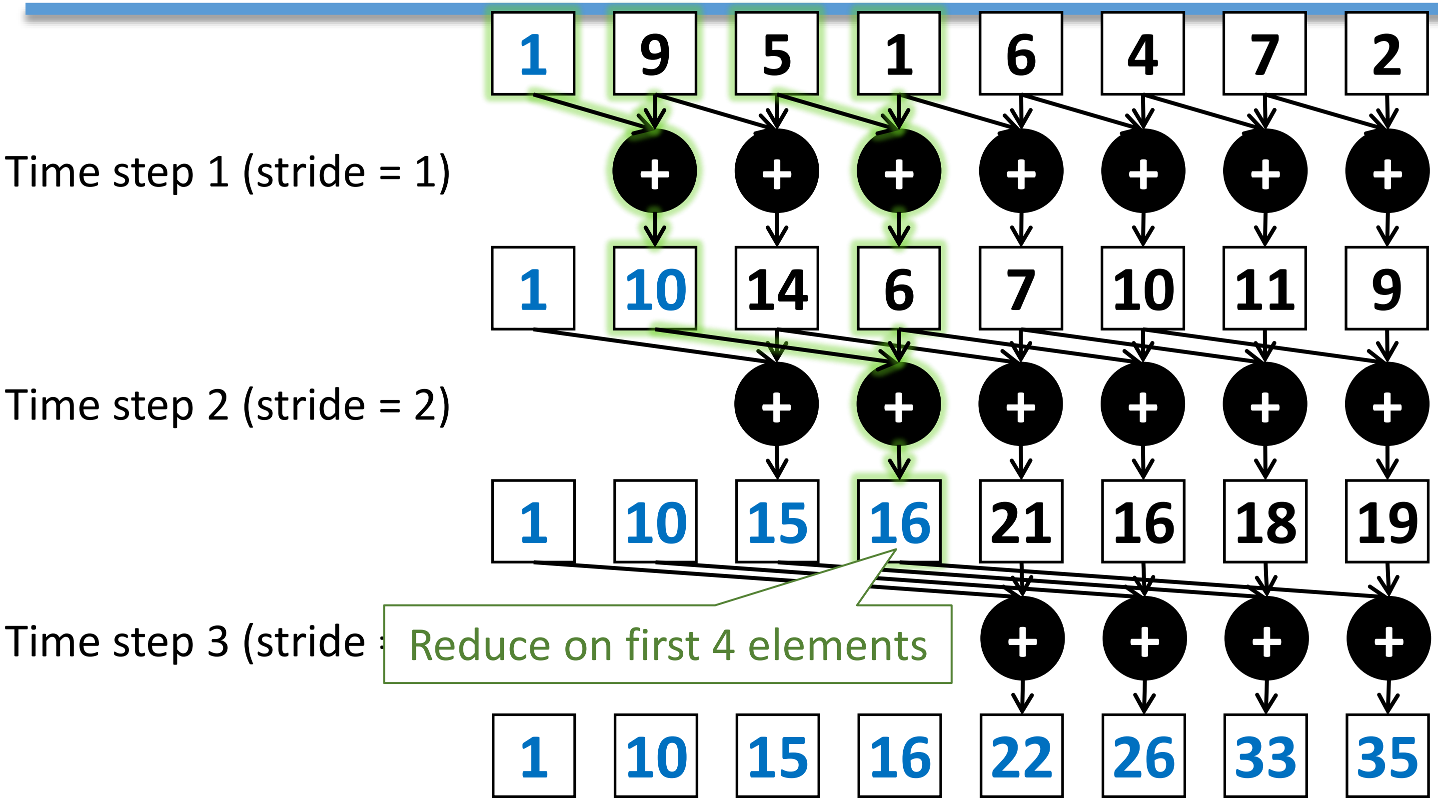
Parallel implementation



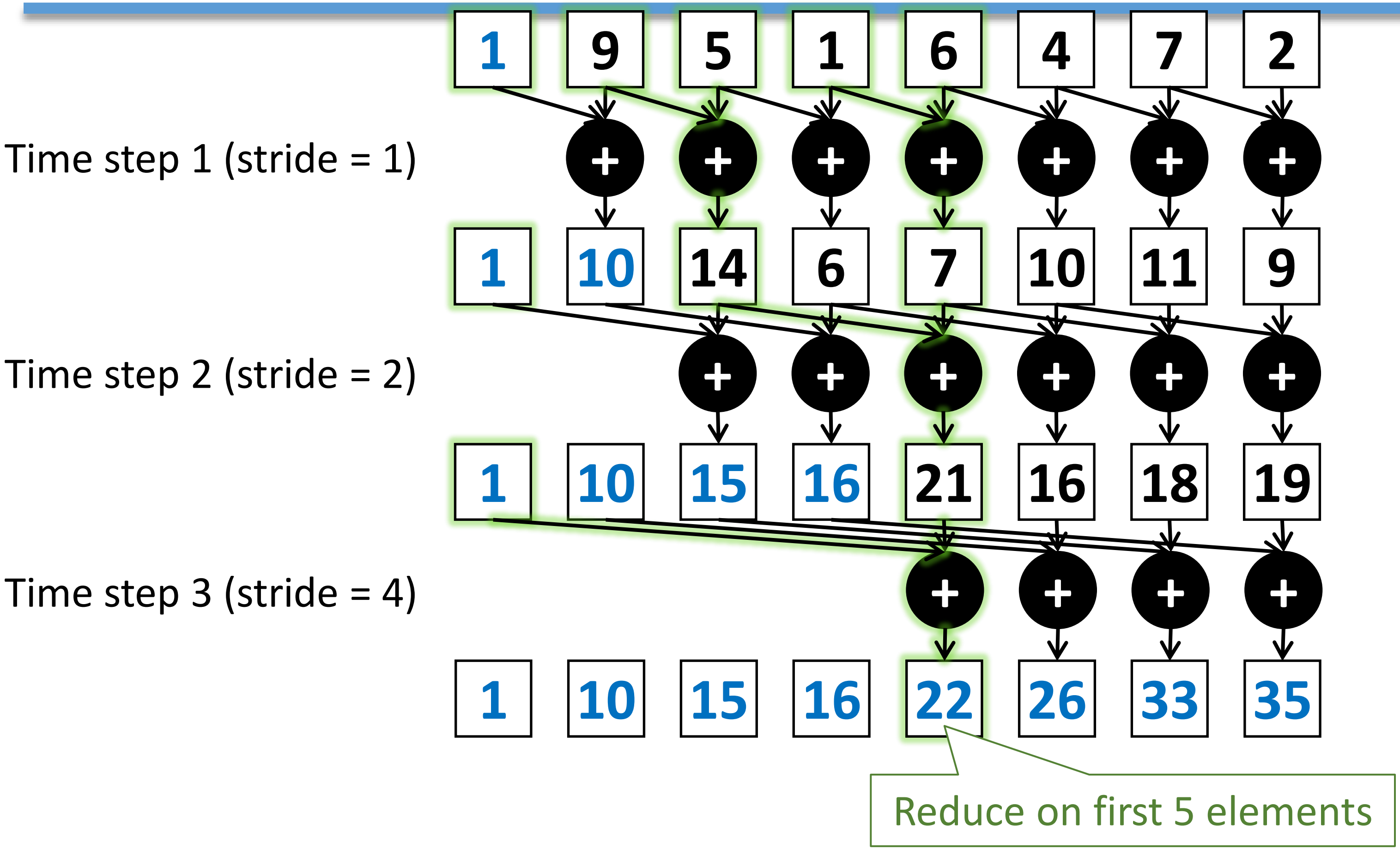
Parallel implementation



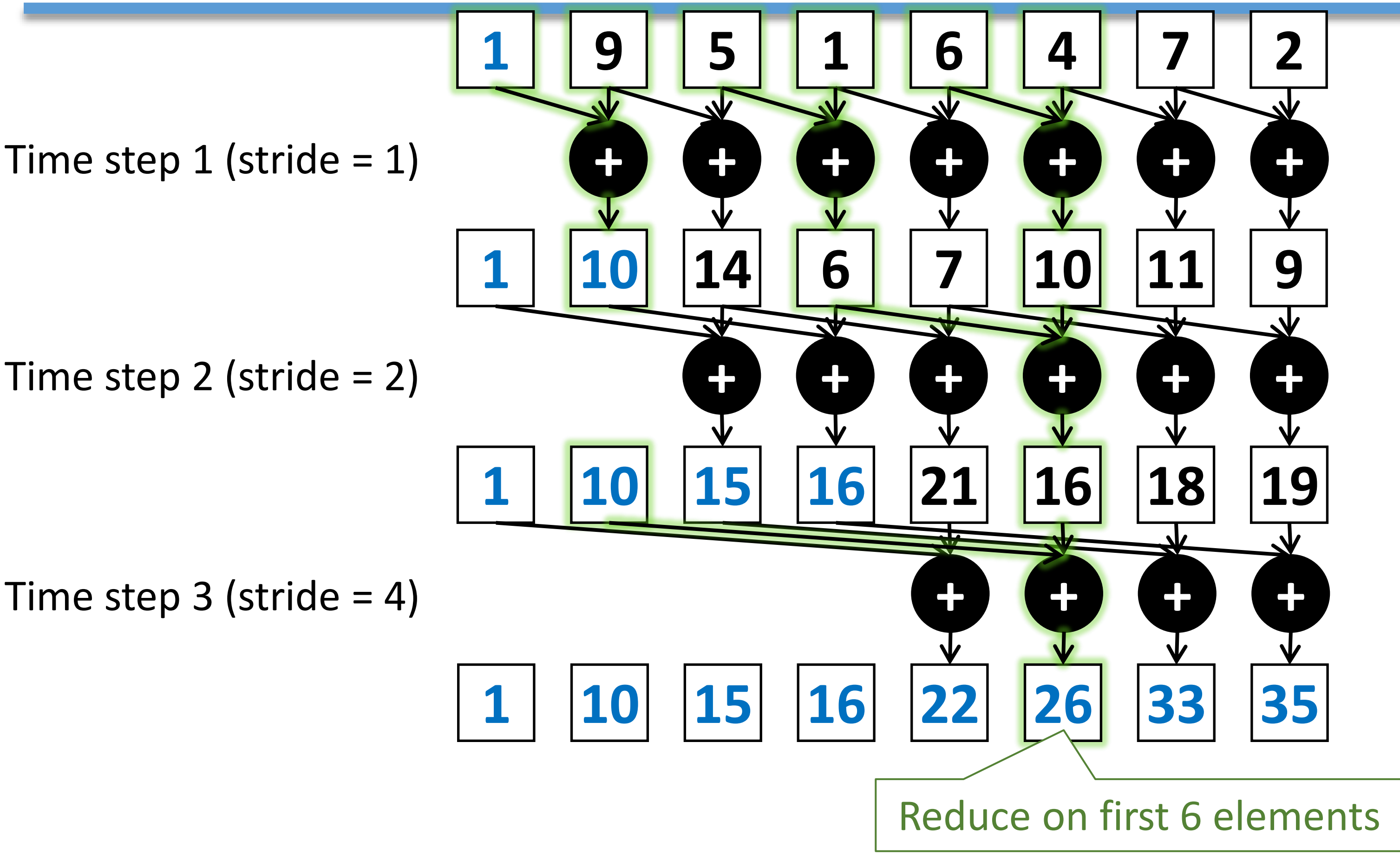
Parallel implementation



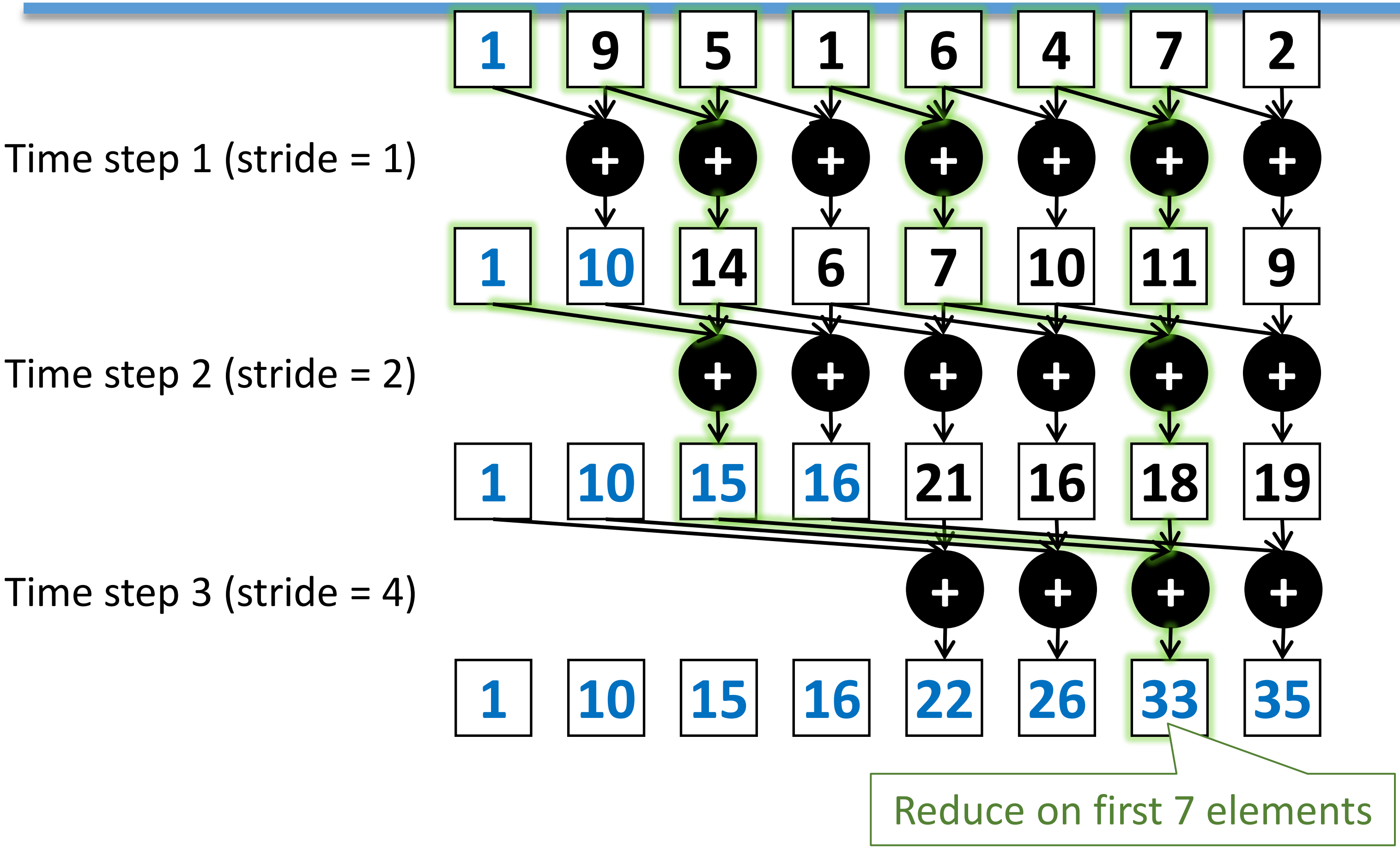
Parallel implementation



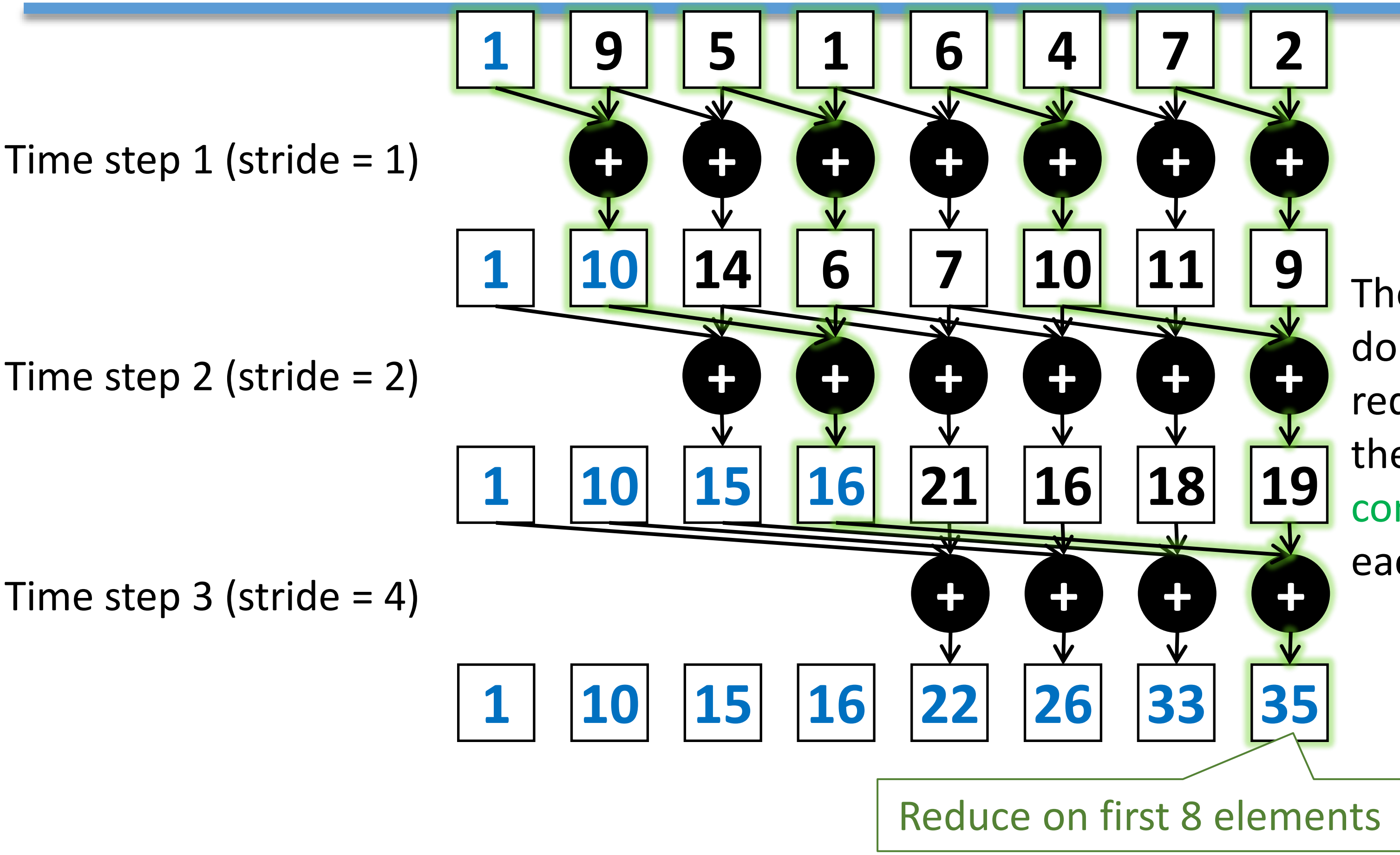
Parallel implementation



Parallel implementation

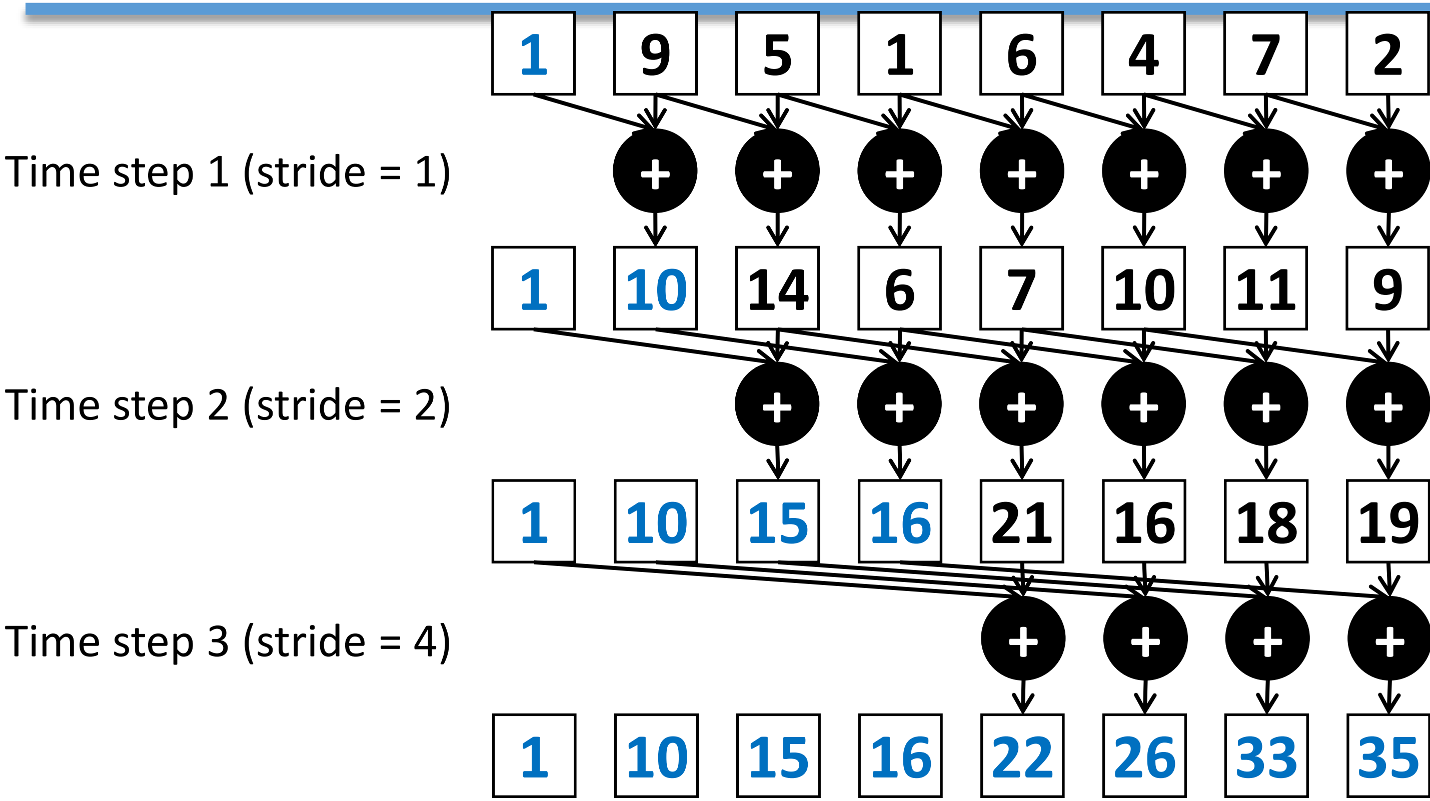


Parallel implementation



The efficiency is we don't use 8 separate reduction trees; these trees share computations with each other

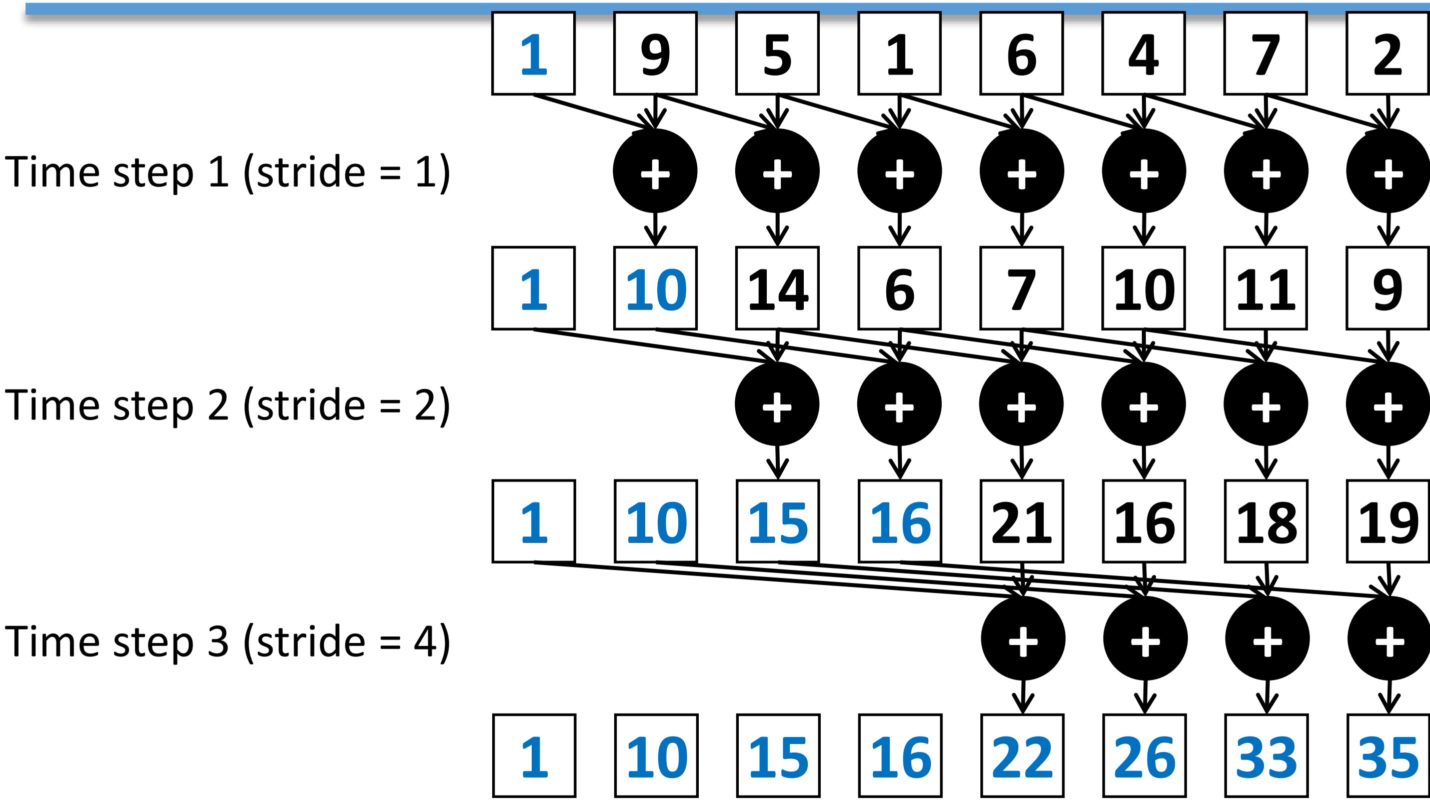
Parallel implementation



Time:

Work:

Parallel implementation



Time: $3 = \log_2 n = O(\log_2 n)$

Work: $17 = (n-1) + (n-2) + (n-4) + \ldots + (n-n/2)$

$= n\log_2 n - \underbrace{(1 + 2 + \ldots + n/2)}_{n-1} = O(n\log_2 n)$

Work-inefficient

Parallel implementation

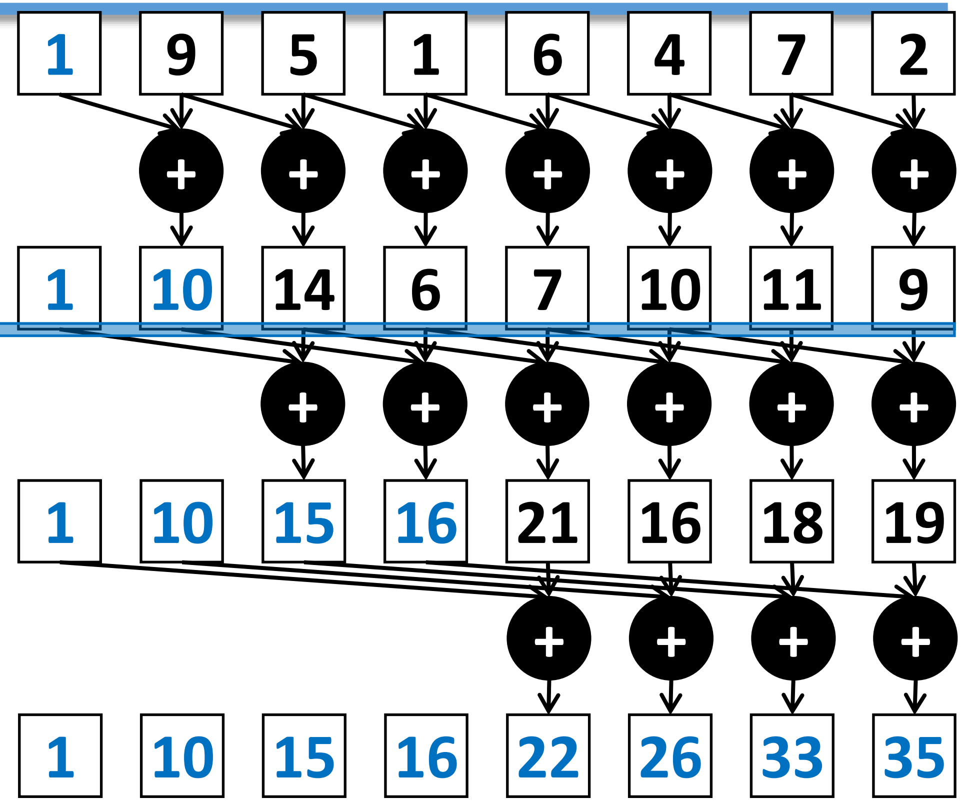
Cần synchronize trước khi sang bước tiếp theo

Need **synchronize** before next step?

But we can only synchronize threads in the same block

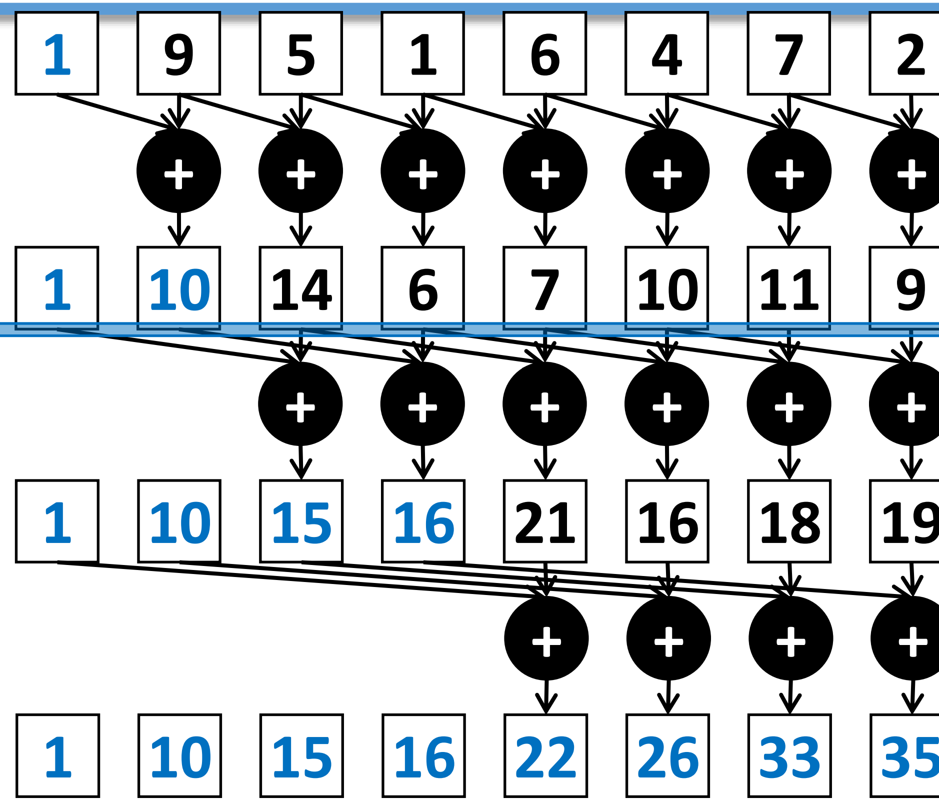
If n ≤ block-data-size, we can use a kernel with one block

If n > block-data-size, what will we do?



Parallel implementation

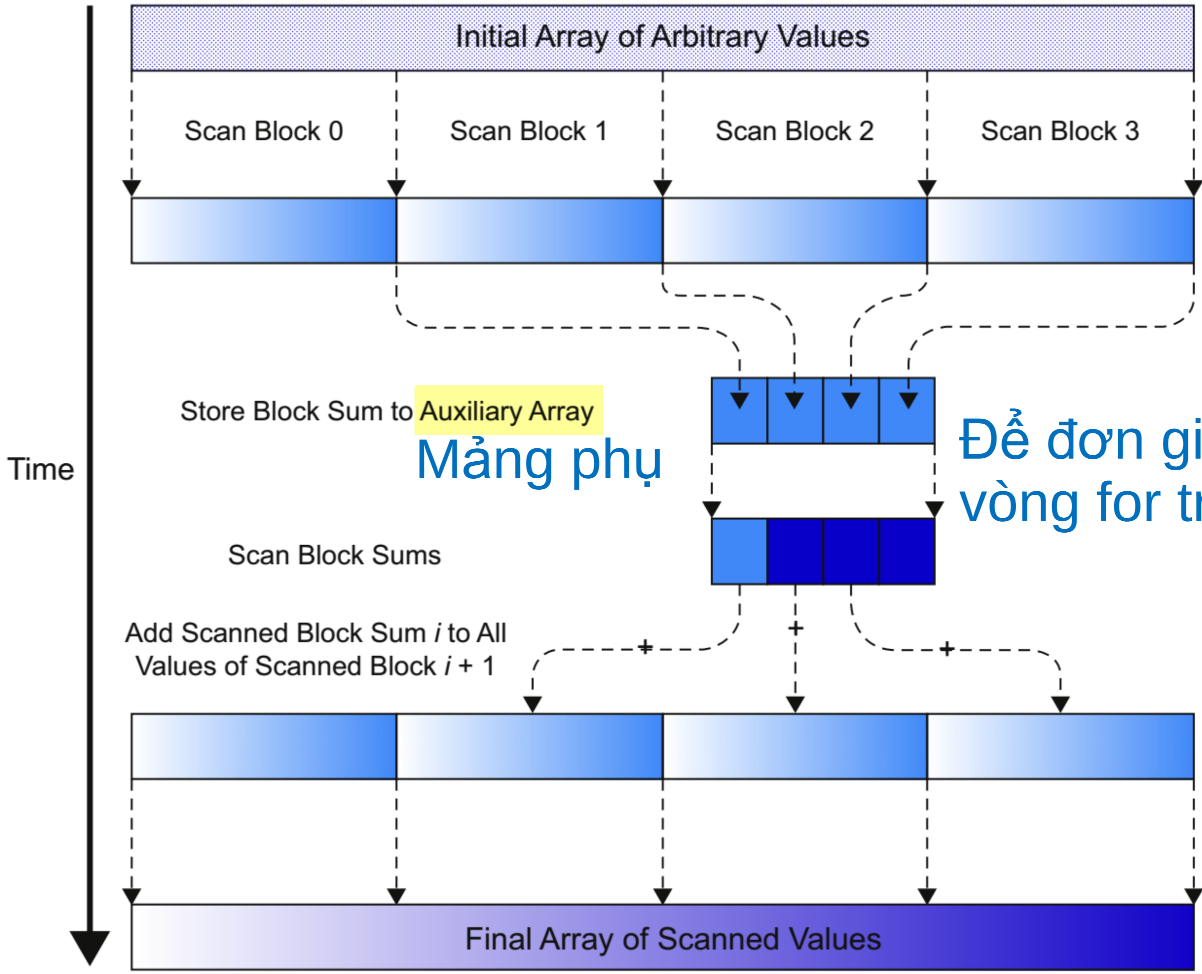
During every iteration, each thread can overwrite the input of another thread



- Barrier synchronization to ensure all inputs have been properly generated
- All threads secure input operand that can be overwritten by another thread
- Barrier synchronization is required to ensure that all threads have secured their inputs
- All threads perform addition and write output

Parallel implementation

If n > block-data-size, what will we do?



Để đơn giản thì dùng vòng for trên host

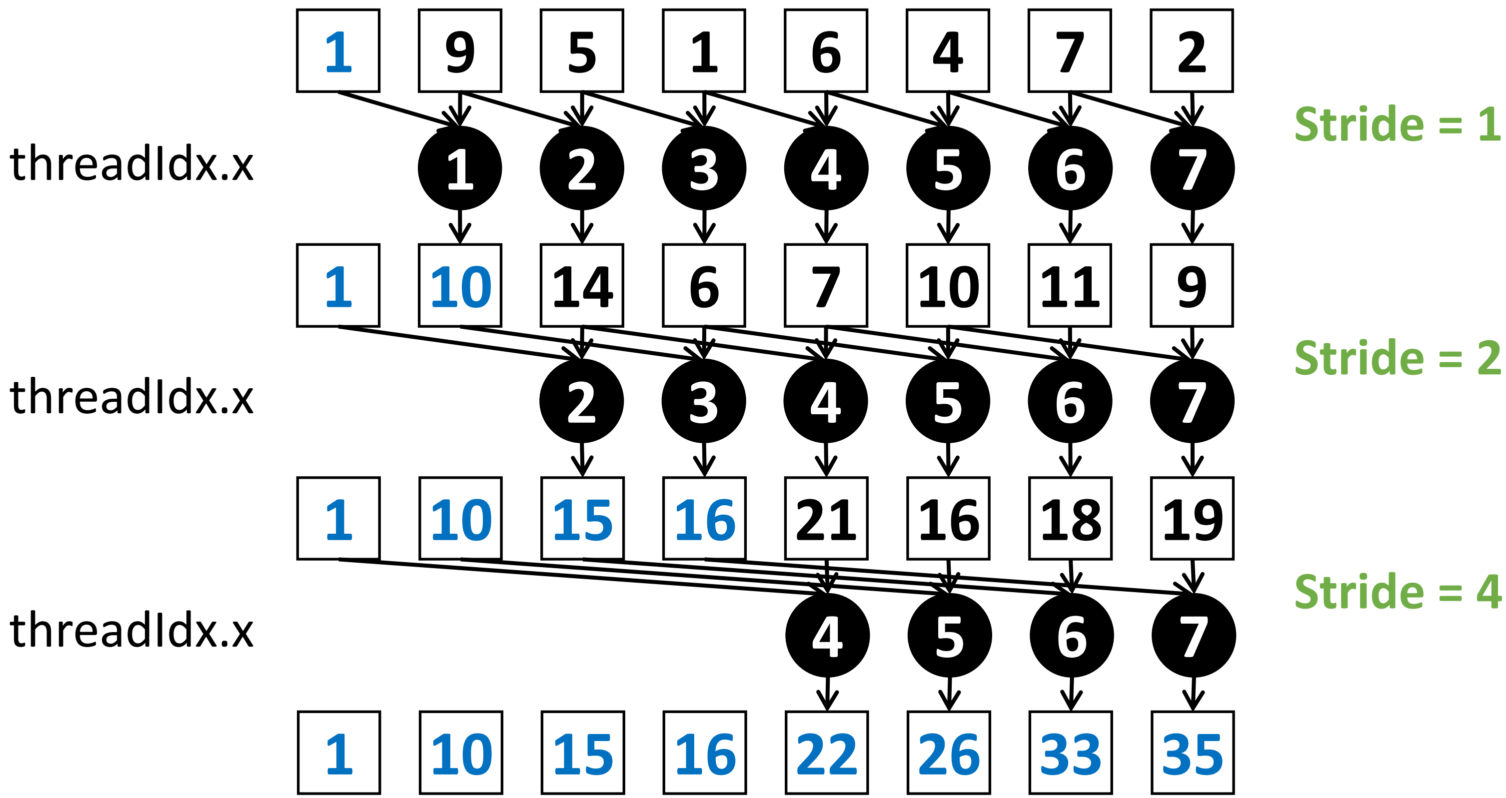
Giai đoạn này chạy trên device

Image source: David B. Kirk et al. Programming Massively Parallel Processors

Scan in each block

Consider a block of 8 threads

1. Block reads data from GMEM to SMEM
2. Block scans with data on SMEM



3. Block writes result from SMEM to GMEM

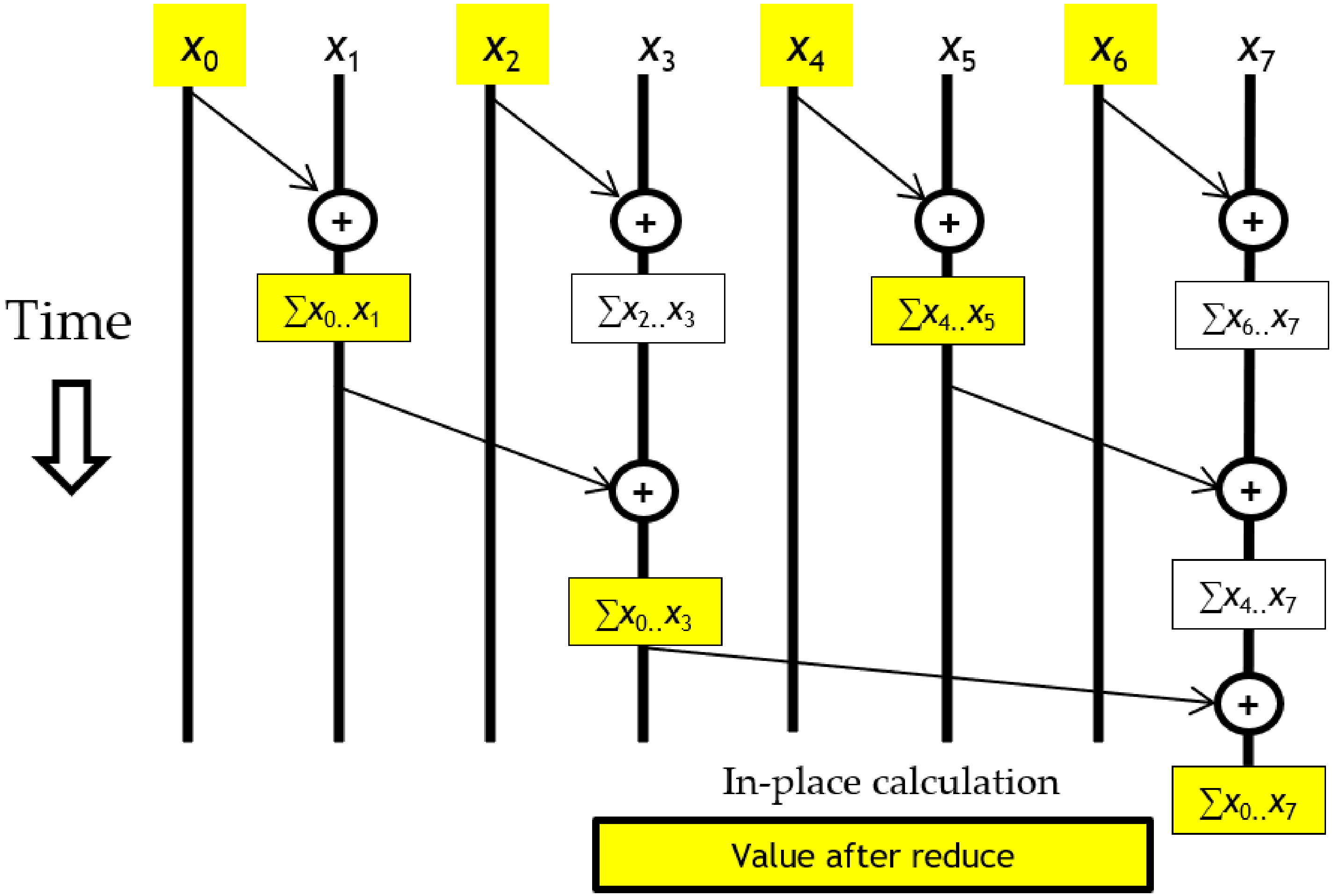
Live coding

Kernel 2 – “work-efficient”

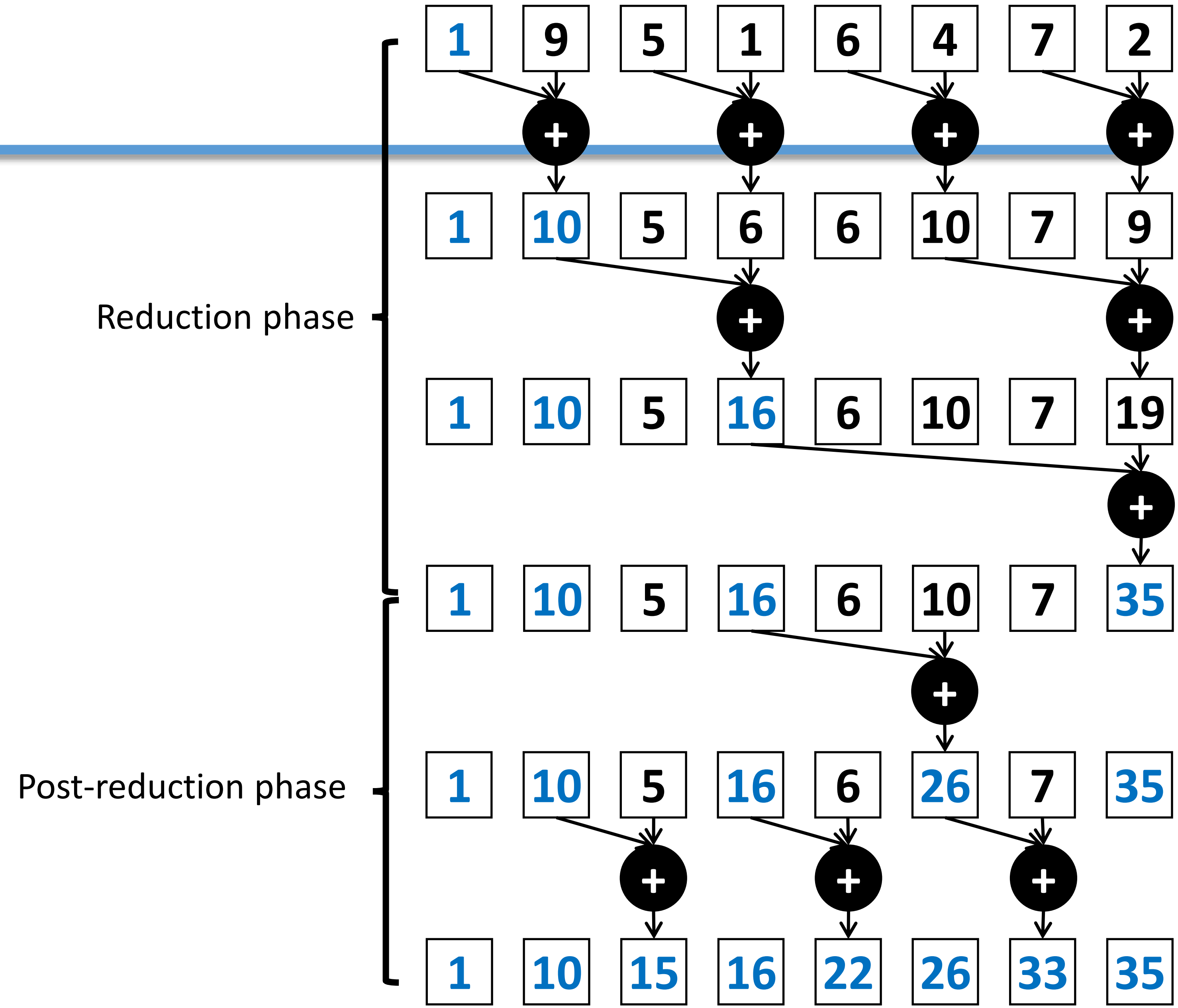
Idea: reduce # pluses by reusing results more

- Balanced Trees Xây dựng cây cân bằng (nhị phân)
  - Form 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 the root building partial sums at internal nodes in the tree
  - The root holds the sum of all leaves
  - Traverse back up the tree building the output from the partial sums

Kernel 2 – “work-efficient”



Time:
Work:

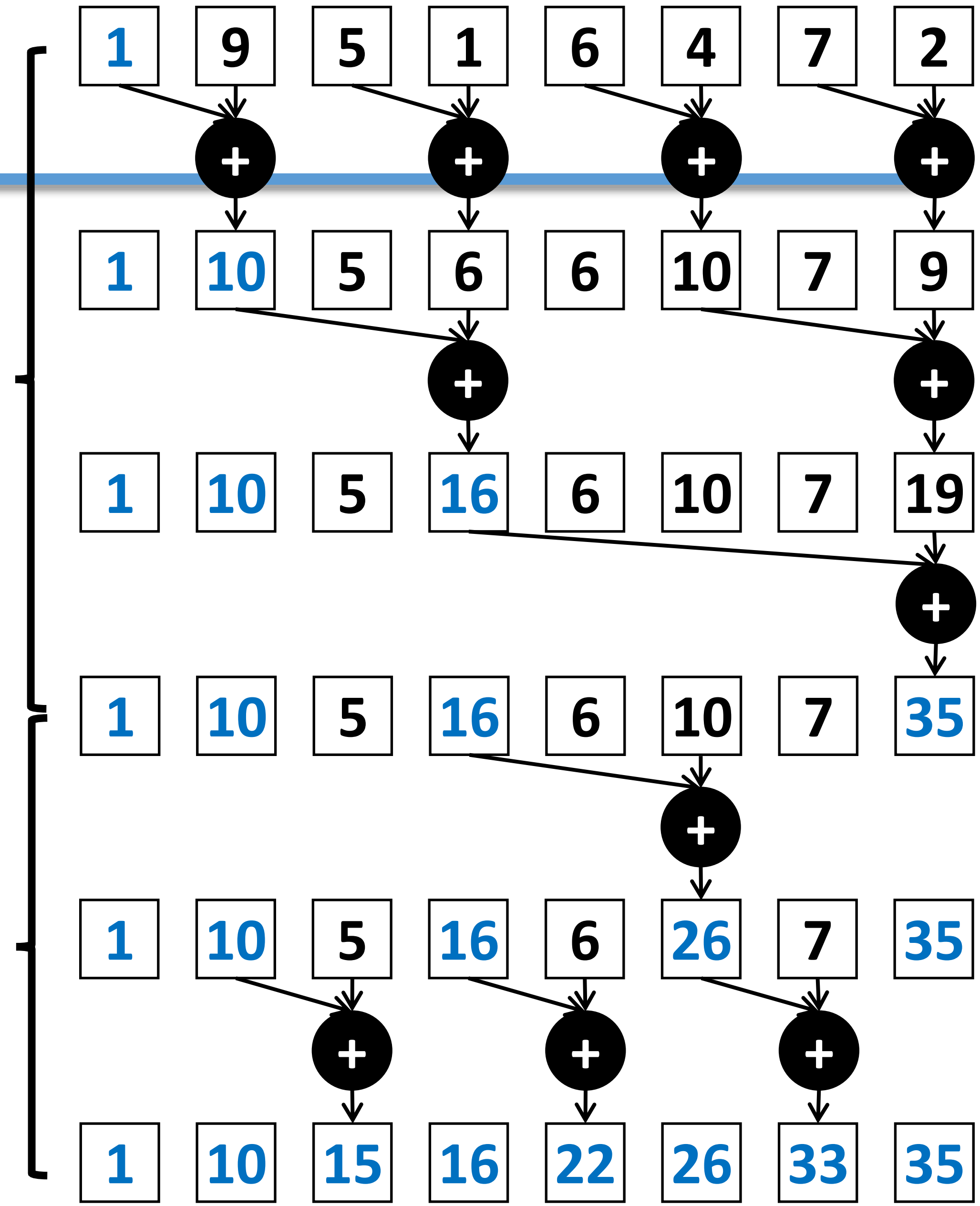

1 9 5 1 6 4 7 2
1 10 5 6 6 10 7 9
1 10 5 16 6 10 7 19
1 10 5 16 6 10 7 35
1 10 5 16 6 26 7 35
1 10 15 16 22 26 33 35
Reduction phase
Post-reduction phase

Time: $5 \approx 2 * \log_2 n = O(\log_2 n)$
Work: $11 \approx 2*(n-1) = O(n)$

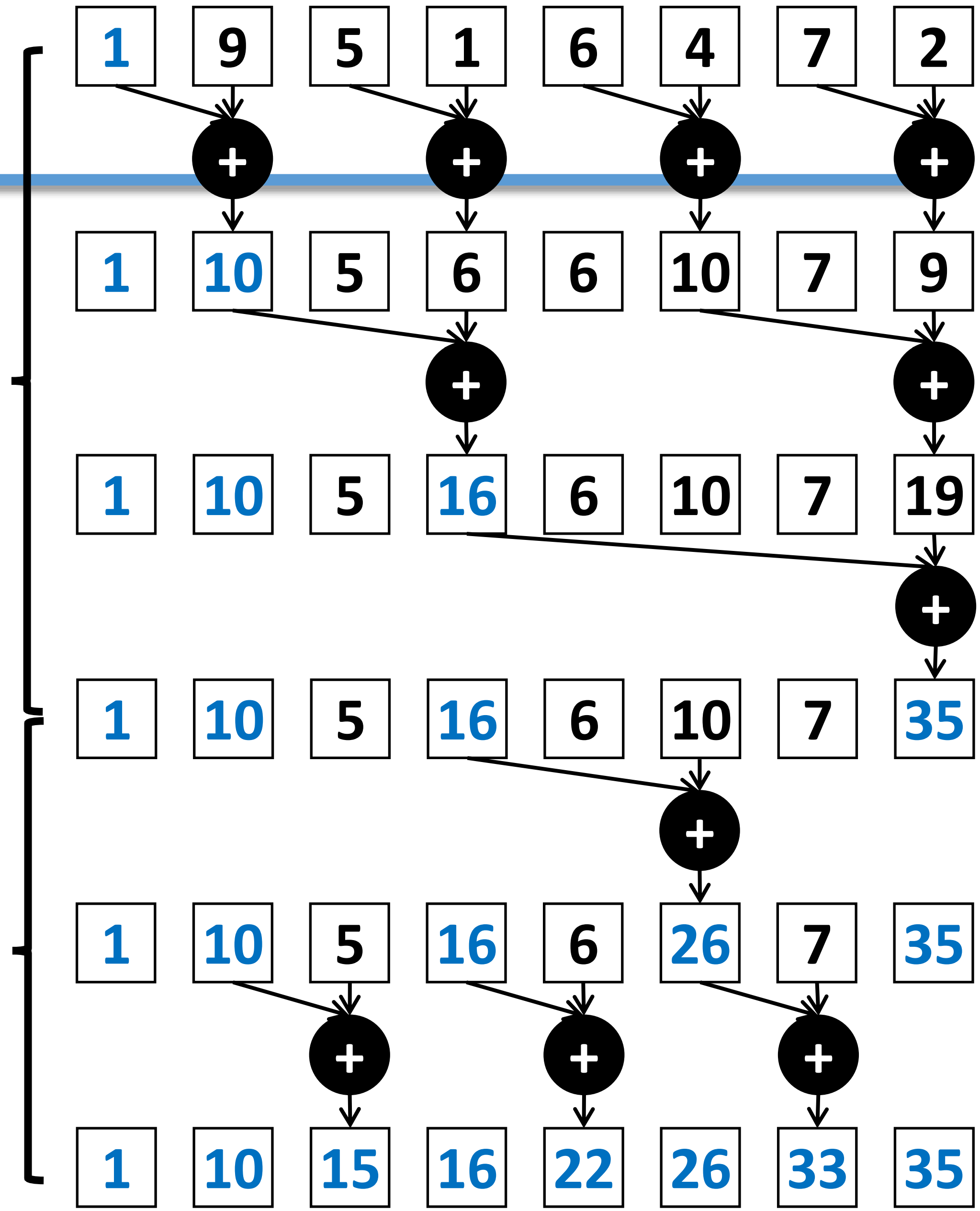
Work-efficient

Reduction phase

Post-reduction phase



Reduction phase



Warp divergence

Solution: let active threads be adjacent threads (similar to kernel 2 in reduction)

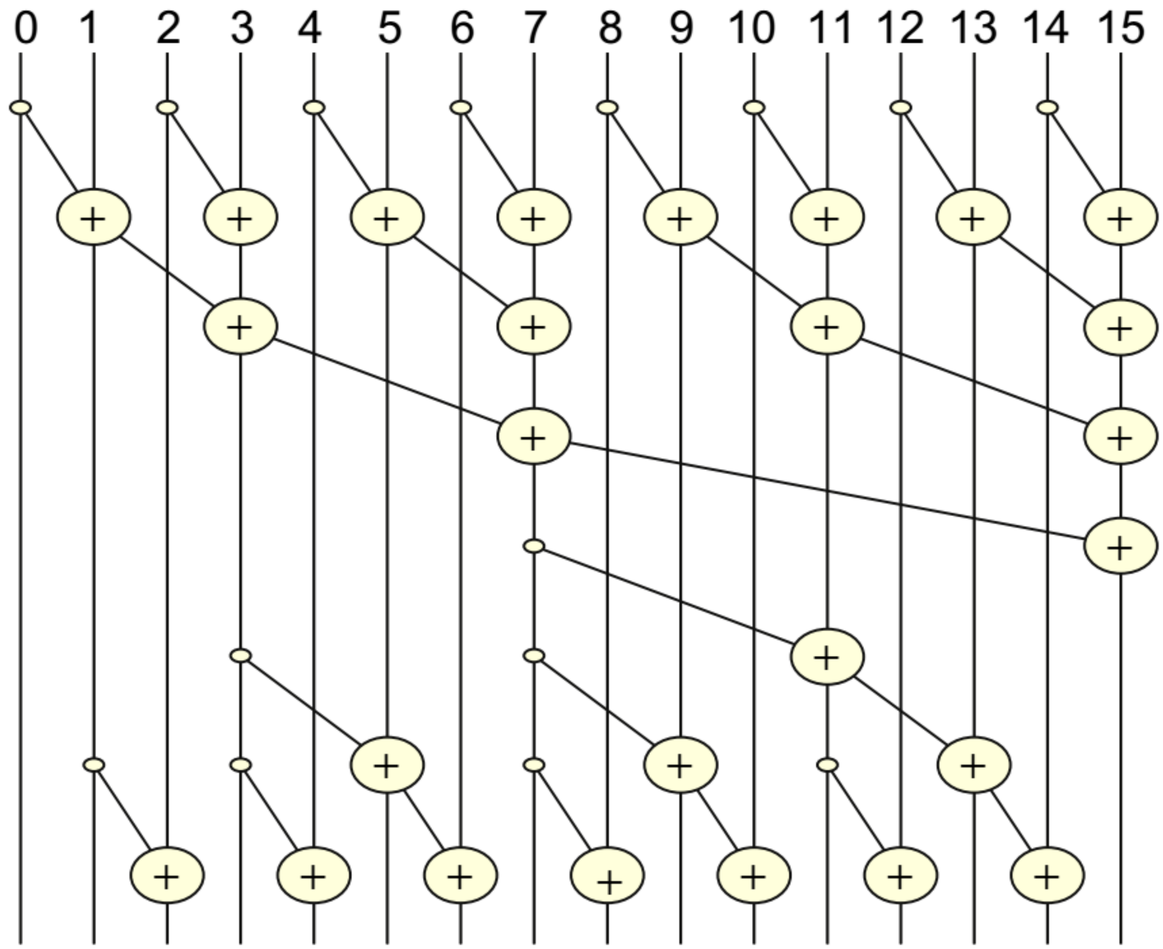
Inefficient GMEM access

Solution: use SMEM

Post-reduction phase

Illustration for 16 elements

Time:
log2(n)



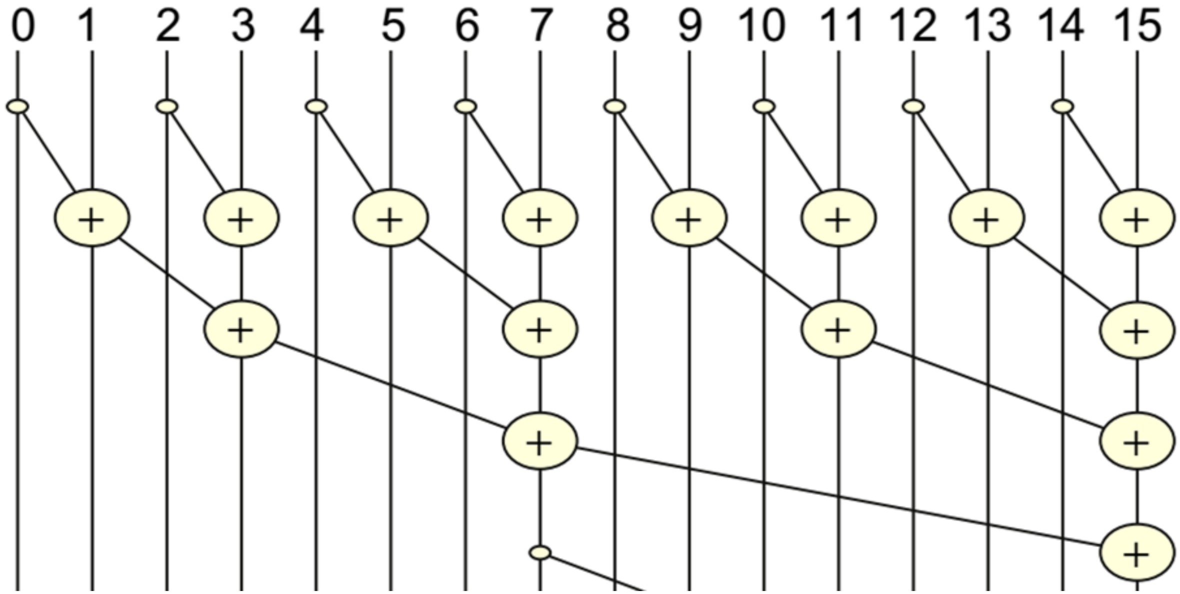
Time:
log2(n) - 1

Image source: David B. Kirk et al. Programming Massively Parallel Processors

Reducion phase

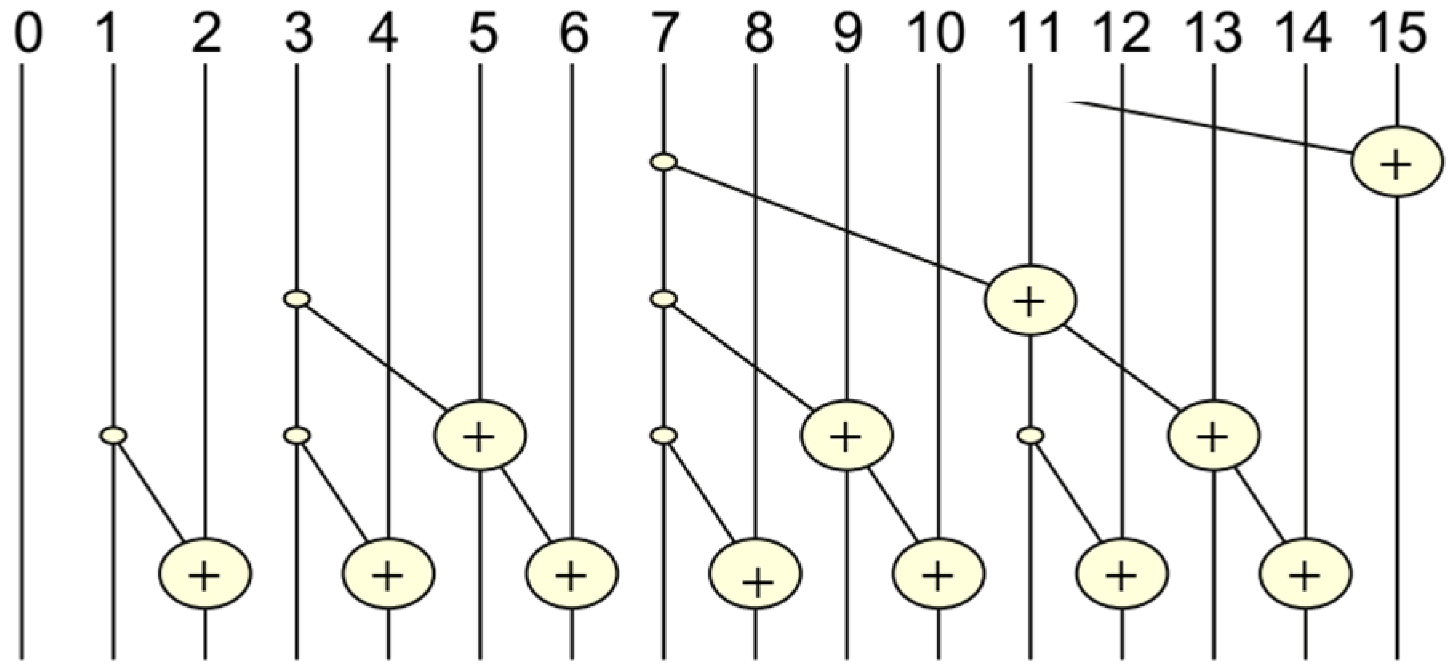
Kích thước mảng = 2* blockDim.x

```
for (int stride = 1; stride < 2 * blockDim.x; stride *= 2)
{
        int s_dataIdx = (threadIdx.x + 1) * 2 * stride - 1;
        if (s_dataIdx < 2 * blockDim.x)
                s_data[s_dataIdx] += s_data[s_dataIdx - stride];
        __syncthreads();
}
```



Post-reduction phase

```
for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
{
	int s_dataIdx = (threadIdx.x + 1) * 2 * stride - 1 + stride;
	if (s_dataIdx < 2 * blockDim.x)
		s_data[s_dataIdx] += s_data[s_dataIdx - stride];
	__syncthreads();
}
```



Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022
- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- [3] Illinois GPU course

https://wiki.illinois.edu/wiki/display/ECE408/ECE408+Home

THE END