Parallel Programming

Parallel Execution in CUDA (Part 2)

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

Overview

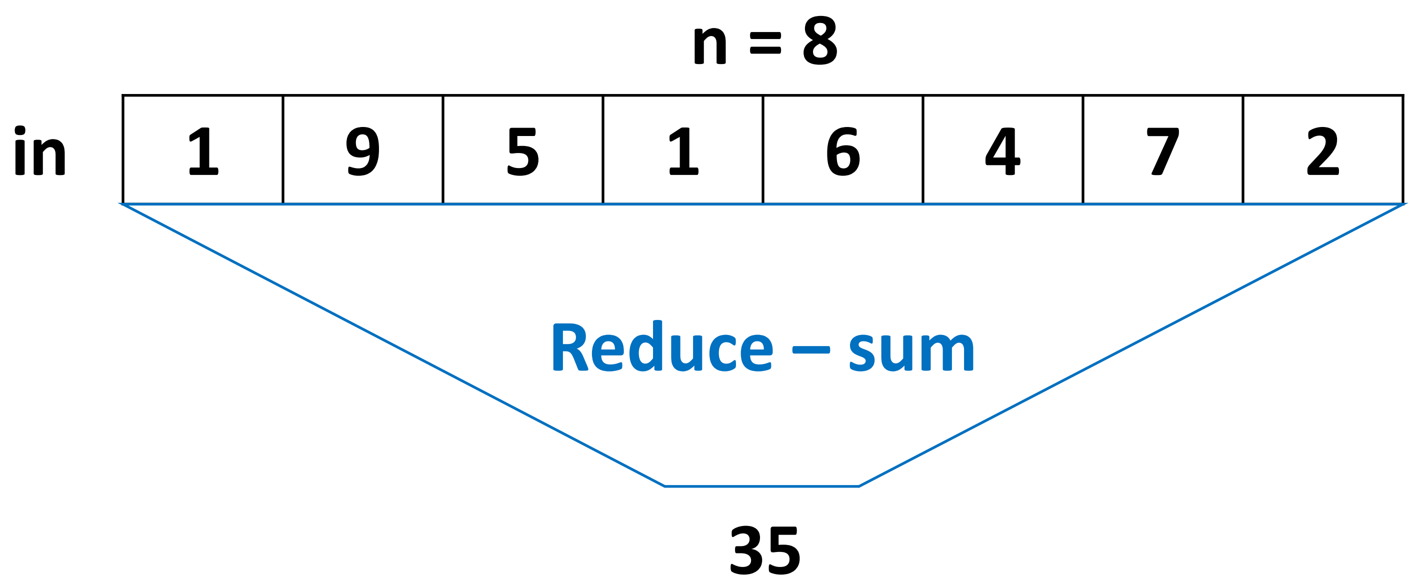
Apply knowledge of parallel execution in CUDA to write a fast CUDA program doing “reduction”

- The “reduction” task
- Sequential implementation
- Parallel implementation
  - Kernel function – 1st version
  - Kernel function – 2nd version: reduce warp divergence
  - Kernel function – 3rd version: reduce warp divergence + …

The “reduction” task

Input: an array **in** of **n** numbers

Output: sum (or product, max, min, …) of these numbers



Reduction algorithm

- Sum reduction sequential:

```
sum = 0;
for (int i = 0;i < N;i++){
    sum += input[i];
}
```

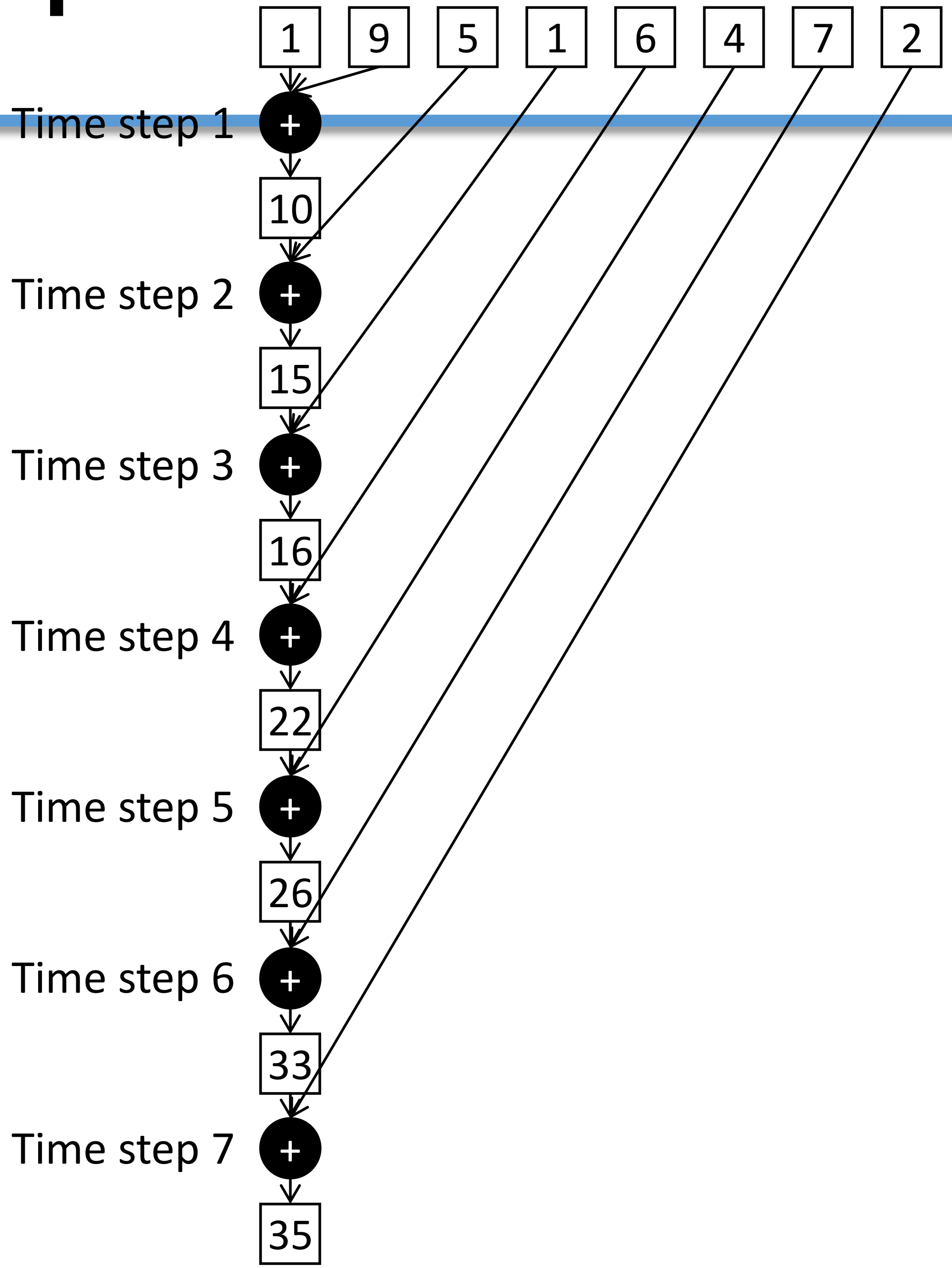
- General form of a reduction sequential:

```
acc = IDENTITY;
for (int i = 0;i < N;i++){
    acc = Operator(acc, input[i]);
}
```

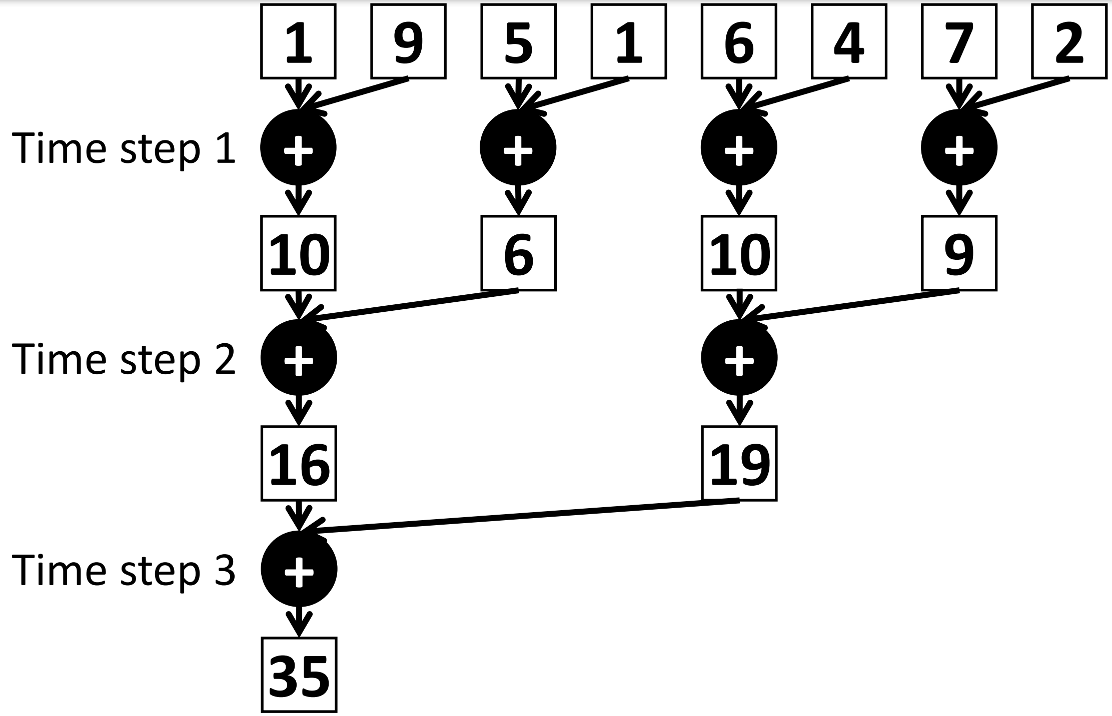
Sequential implementation

```
int reduceOnHost(int *in, int n)
{
    int s = in[0];
    for (int i = 1; i < n; i++)
        s += in[i];
    return s;
}
```

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



Parallel implementation – idea



Time: ?
Work: ?

Parallel implementation – idea

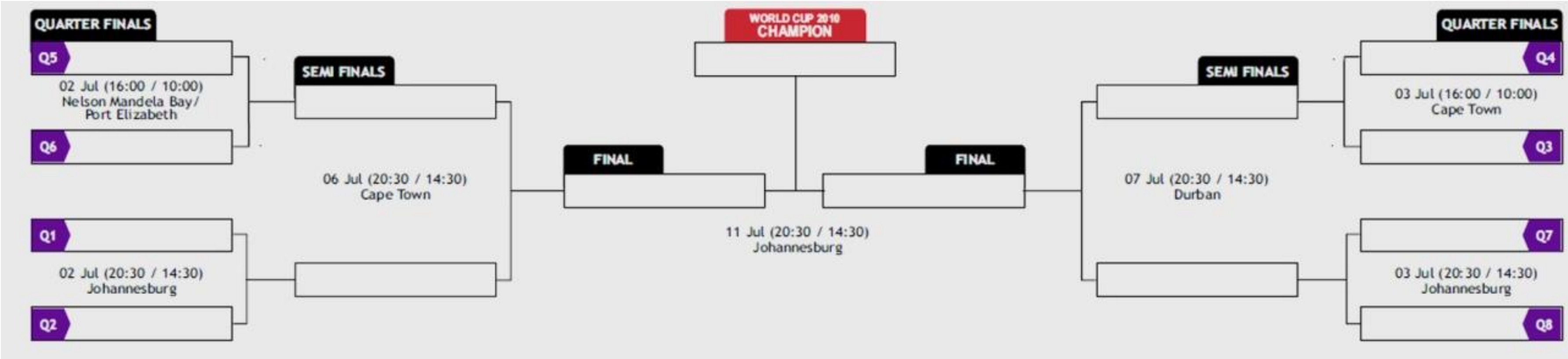
- For N input values, the reduction tree performs
  - $N/2 + N/4 + N/8 + ... + 1 = N - 1$ operations
  - In $\text{Log}\,(N)$ steps – $1{,}000{,}000$ input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism $(N\text{-}1)/\text{Log}(N))$
    - For $N = 1{,}000{,}000$, average parallelism is $50{,}000$
    - However, peak resource requirement is $500{,}000$
    - This is not resource efficient
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to the an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

Reduction trees

- Order of performing the operations will be changed (sequential ➔ parallel)
  - Operator must be associative
  - Serial
    - ((((((3 max 1) max 7) max 0) max 4) max 1) max 6) max 3
  - Paralell
    - ((3 max 1) max (7 max 0)) max ((4 max 1) max (6 max 3))
- We also need rearranges the order of the operands
  - Operator to be commutative

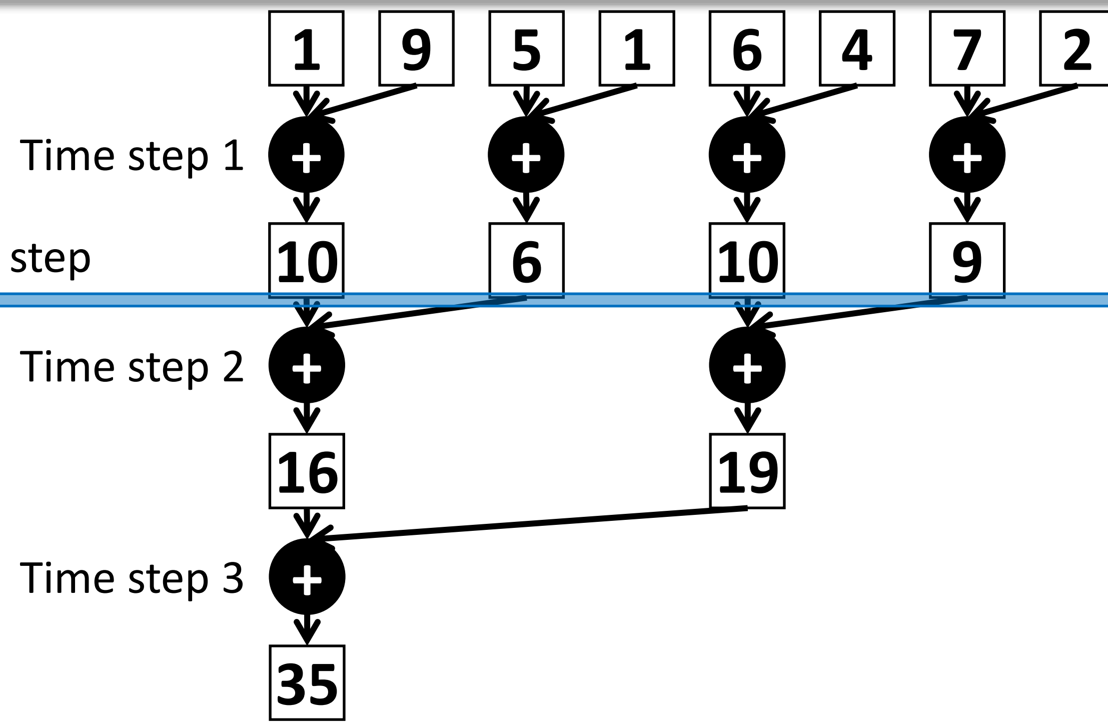
Parallel Reduction in Real life

- Sports & Competitions: Max reduction



- Also use to process large input data sets (Google and Hadoop MapReduce frameworks)
  - There is no required order of processing elements in a data set (associative and commutative)
  - 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

Parallel implementation – idea



Need **synchronization** before next step

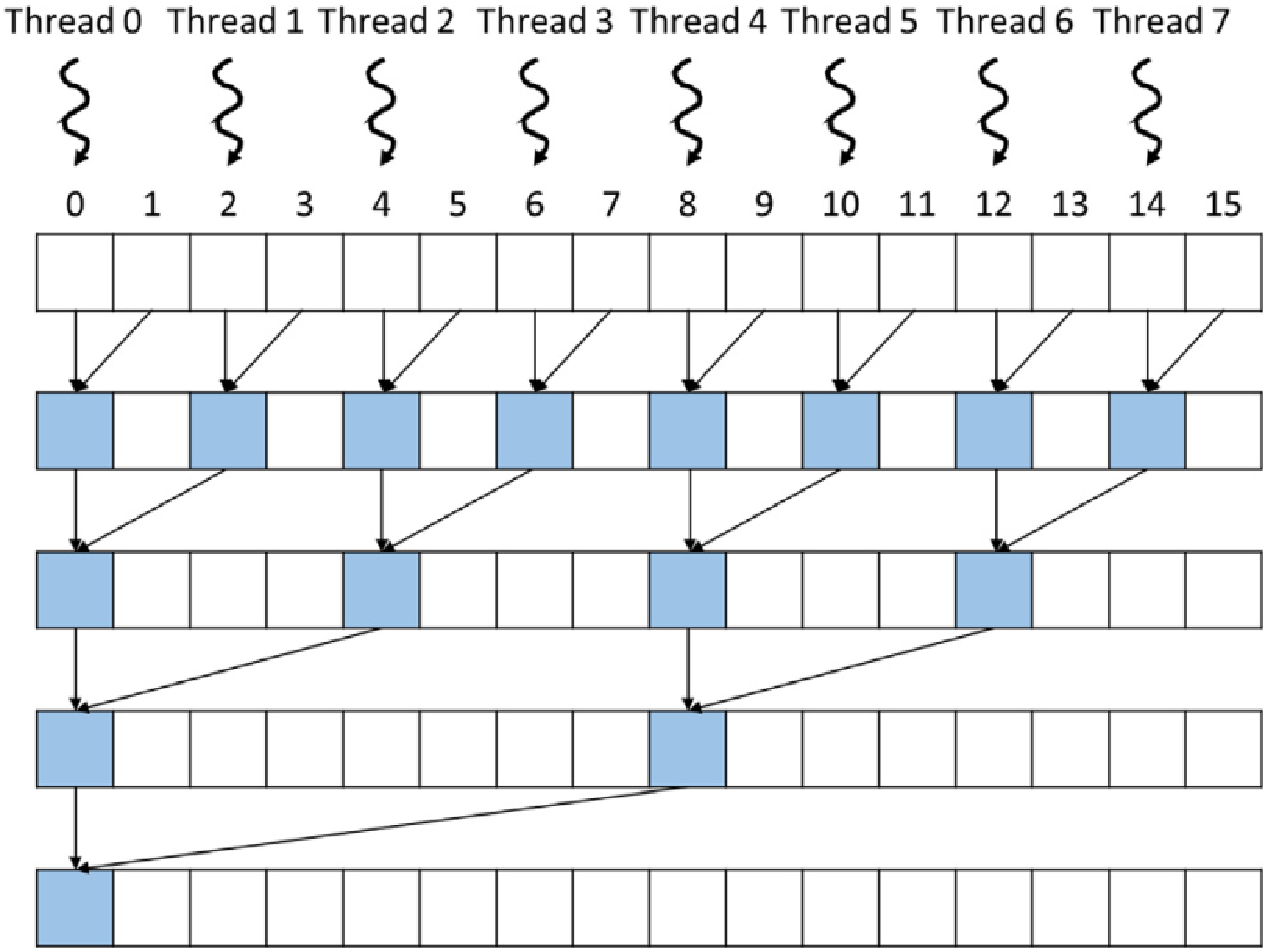
But: in a kernel function, we can only synchronize threads in the same block

If n $\leq$ 2$\times$block-size, we can use a kernel with one block

If n $>$ 2$\times$block-size, what should we do?

Time: 3 = $\log_2 n$ = $O(\log_2 n)$
Work: 7 = n-1 = O(n) = work of the sequential version
(Later, we will see tasks in which parallel implementations need to do more work than sequential)

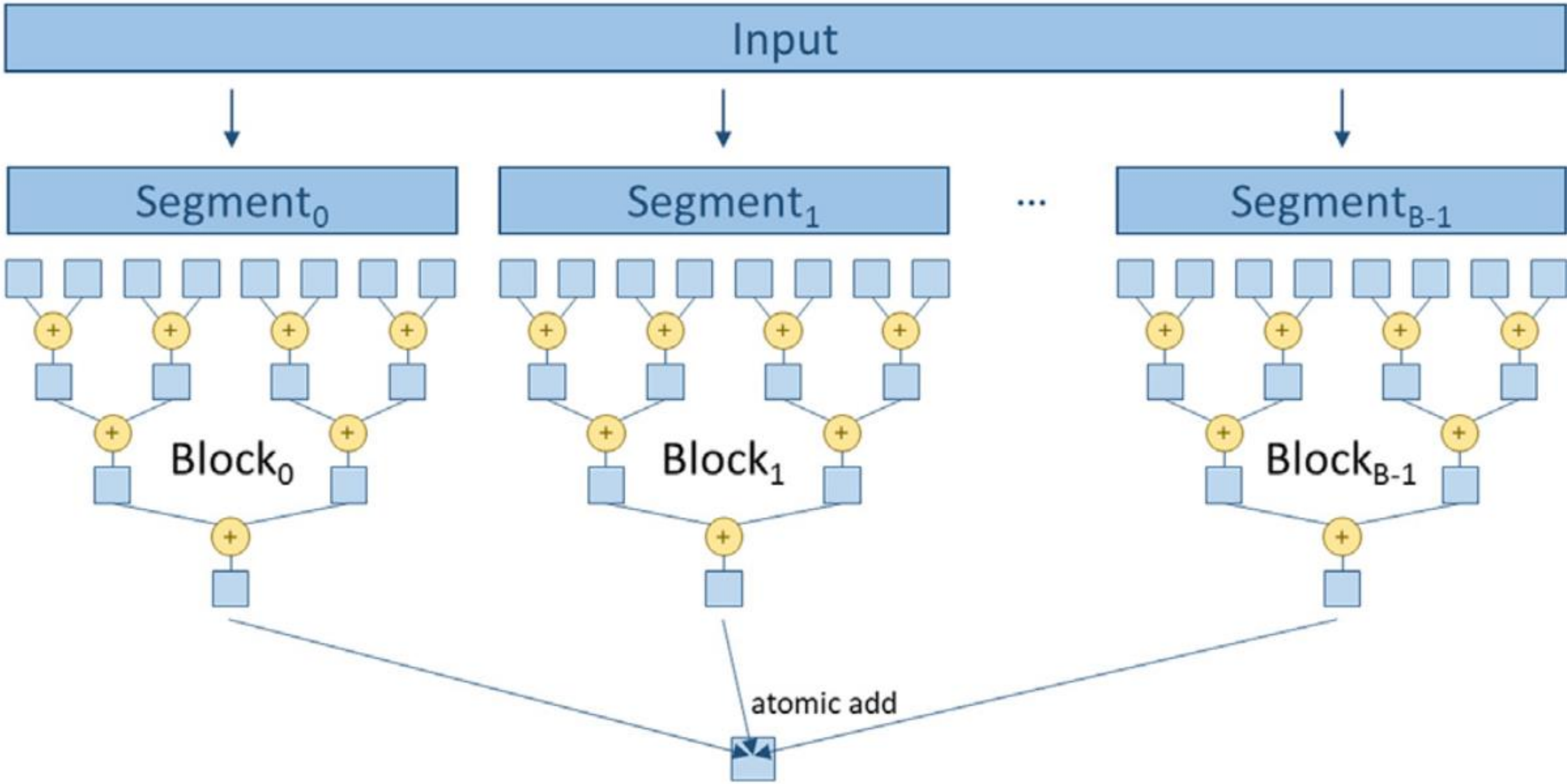
A simple reduction kernel



A simple reduction kernel

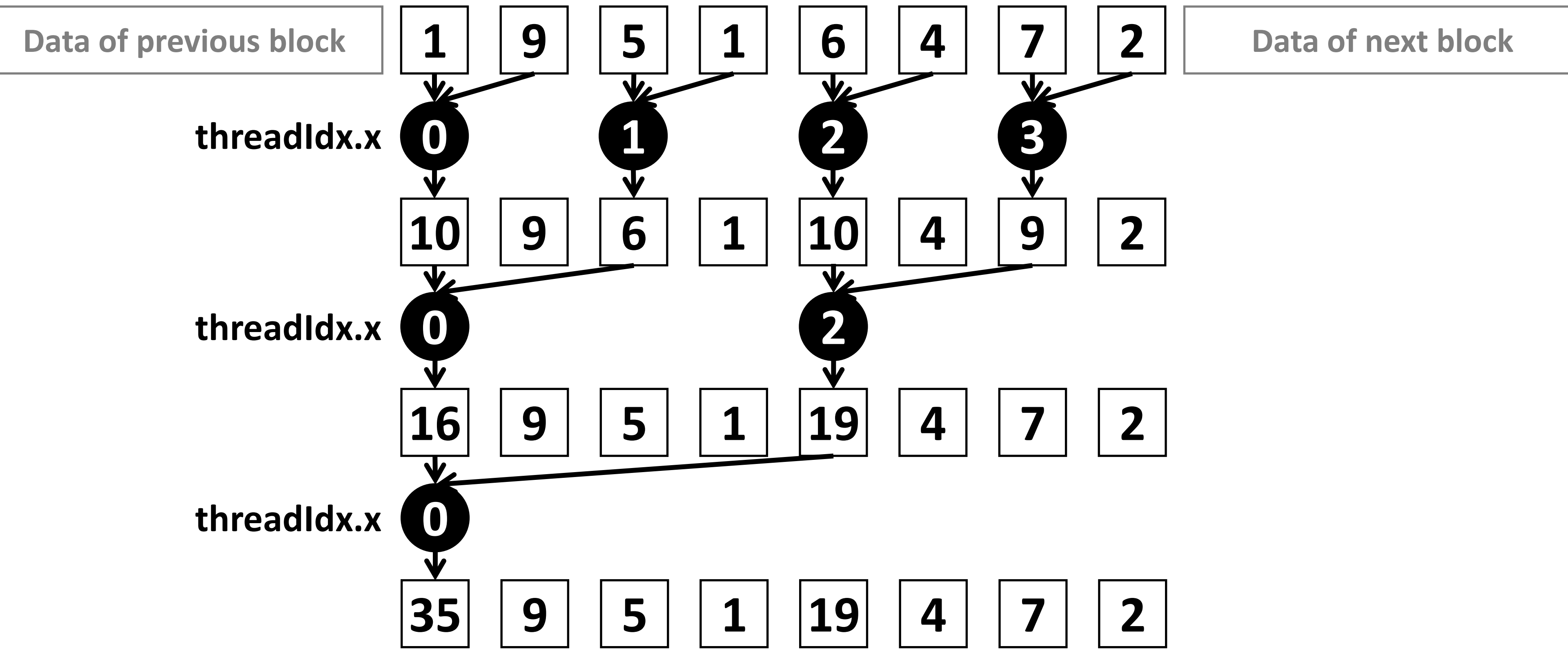
```
__global__ void reduceBlksKernel0(int* in, int* out, int n) {
    int i = 2 * threadIdx.x;
    for (int stride = 1; stride <= blockDim.x; stride *= 2) {
        if (threadIdx.x % stride == 0)
            in[i] += in[i + stride];
        __syncthreads();
    }
    if (threadIdx.x == 0)
        *out = in[0];
}
```

Hierarchical reduction for bigger input



Parallel implementation
– idea to reduce within each block

Consider a block of 4 threads



Hierarchical reduction for arbitrary input length

```
__global__ void reduceBlksKernel1(int* in, int* out, int n){
    int i = blockIdx.x * 2 * blockDim.x + 2 * threadIdx.x;
    for (int stride = 1; stride <= blockDim.x; stride *= 2){
        if (threadIdx.x % stride == 0)
            if (i + stride < n)
                in[i] += in[i + stride];
        __syncthreads(); // Synchronize within each block
    }
    if (threadIdx.x == 0)
        atomicAdd(out, in[blockIdx.x * 2 * blockDim.x]);
}
```

Assume: 2×block-size = 2^k

In each block, how many diverged warps? (not consider blocks in the edge)

- **Stride = 1:**

  All threads are "on"

  → No diverged warp

- **Stride = 2:**

  Only threads with `threadIdx.x % 2 == 0` are "on"

  → All warps are diverged

- **Stride = 4, 8, …, 32:**

  All warps are diverged

- **Stride = 64, 128, …:**

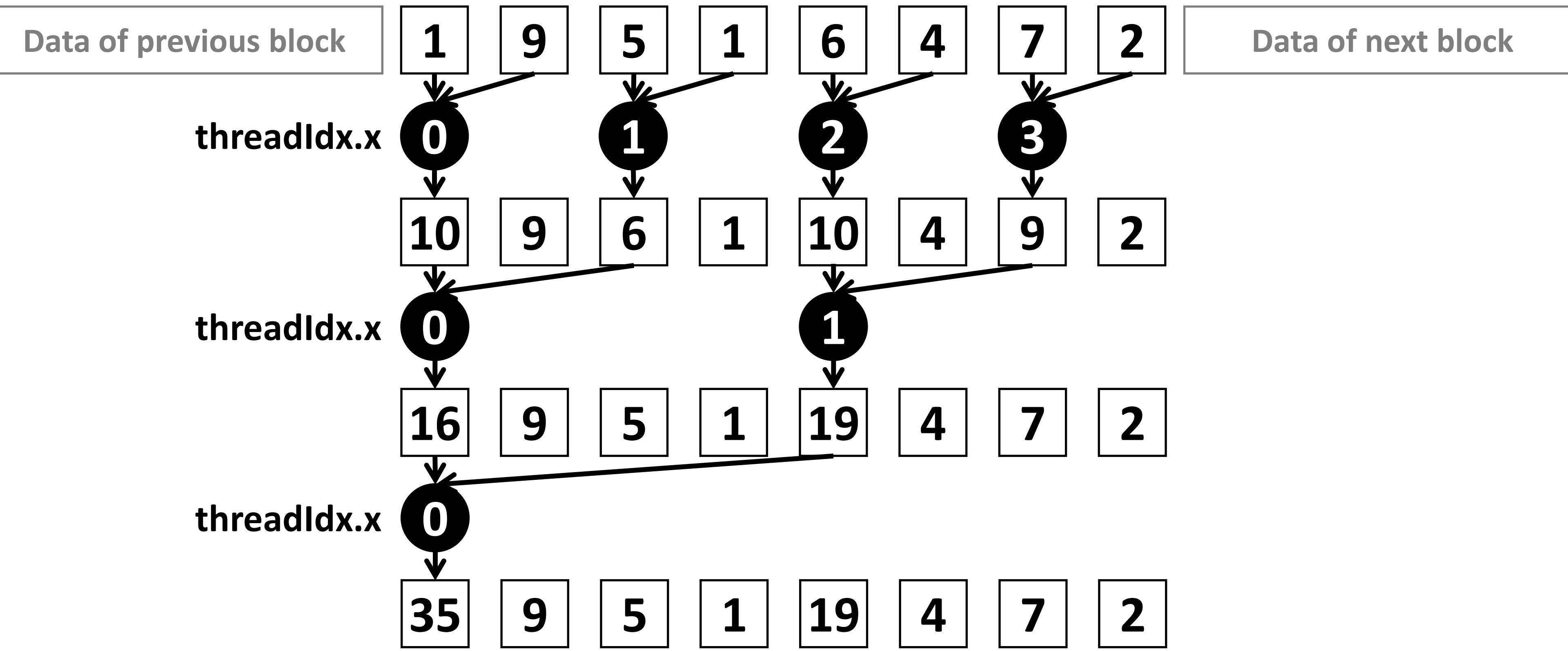
  # diverged warps decrease to 1

Kernel function – 2^{nd} version: reduce warp divergence

- **Idea:** reduce # diverged warps in each step by rearranging threads so that "on" threads are first adjacent threads
- **Example:** consider a block of 128 threads
  - Stride = 1: All 128 threads are "on"
  - Stride = 2: First 64 threads are "on", the rest are "off"
  - Stride = 4: First 32 threads are "on", the rest are "off"
  - …

Kernel function - 2nd version: reduce warp divergence

Consider a block of 4 threads



Kernel function – 2nd version: reduce warp divergence

```
__global__ void reduceOnDevice2(int *in, int *out, int n)
{
    int numElemsBeforeBlk = blockIdx.x * blockDim.x * 2;

    for (int stride = 1; stride < 2 * blockDim.x; stride *= 2)
    {
        int i = numElemsBeforeBlk + ...;
        if (threadIdx.x ...)
            if (i + stride < n)
                in[i] += in[i + stride];
        __syncthreads(); // Synchronize within each block
    }

    if (threadIdx.x == 0)
        out[blockIdx.x] = in[numElemsBeforeBlk];
}
```

Kernel function – 2nd version: reduce warp divergence

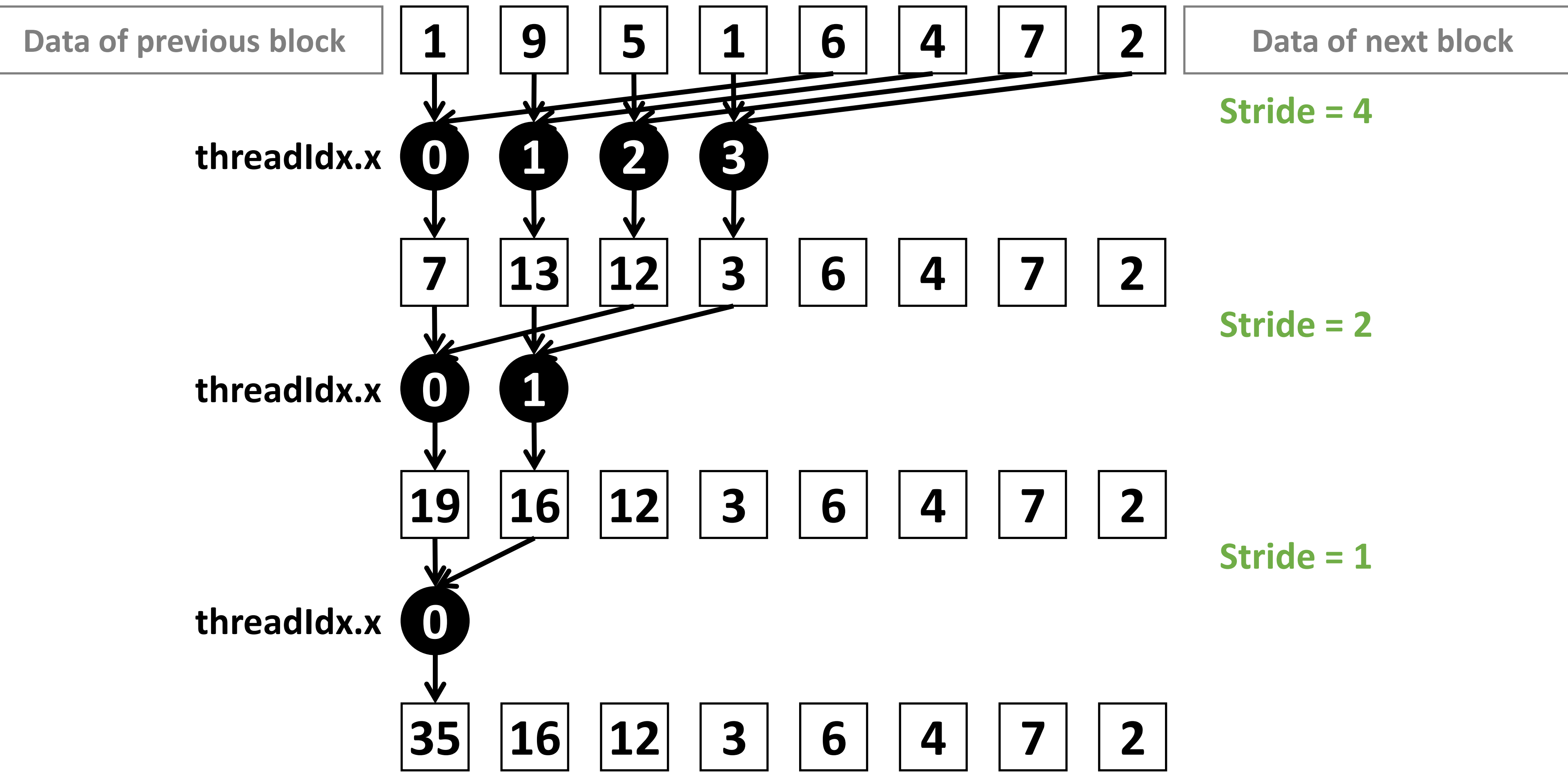
```
__global__ void reduceOnDevice2(int *in, int *out, int n)
{
    int numElemsBeforeBlk = blockIdx.x * blockDim.x * 2;

    for (int stride = 1; stride < 2 * blockDim.x; stride *= 2)
    {
        int i = numElemsBeforeBlk + threadIdx.x * 2 * stride;
        if (threadIdx.x < blockDim.x / stride)
            if (i + stride < n)
                in[i] += in[i + stride];
        __syncthreads(); // Synchronize within each block
    }

    if (threadIdx.x == 0)
        out[blockIdx.x] = in[numElemsBeforeBlk];
}
```

Kernel function - 3rd version: reduce warp divergence + ?

Consider a block of 4 threads



Kernel function - 3rd version: reduce warp divergence + ?

Code: your next homework ;-)

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

THE END