Parallel Algorithms Patterns: Map

COMPUTAÇÃO DE ALTO DESEMPENHO 2018/2019

HERVÉ PAULINO

SLIDES ADAPTED FROM

"STRUCTURED PARALLEL PROGRAMMING - CIS 410/510, UNIVERSITY OF OREGON"

REZAUL A. CHOWDHURY - CSE 613: PARALLEL PROGRAMMING

Bibliography

Sections 3.5 to 3.7 and Chapter 4 of Structured Parallel Programming, Michael McCool, Arch D. Robison and James Reinders. Morgan Kaufmann (2012)

Parallel Control Patterns

Serial control patterns

- Loop, recursion, and more

Parallel control patterns extend serial control patterns

Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns

Parallel control patterns: fork-join, map, stencil, reduction, scan, recurrence

Parallel Control Patterns: Fork-Join

Fork-join: allows control flow to fork into multiple parallel flows, then rejoin later

CUDA implements this when executing a kernel

A “join” is different than a “barrier

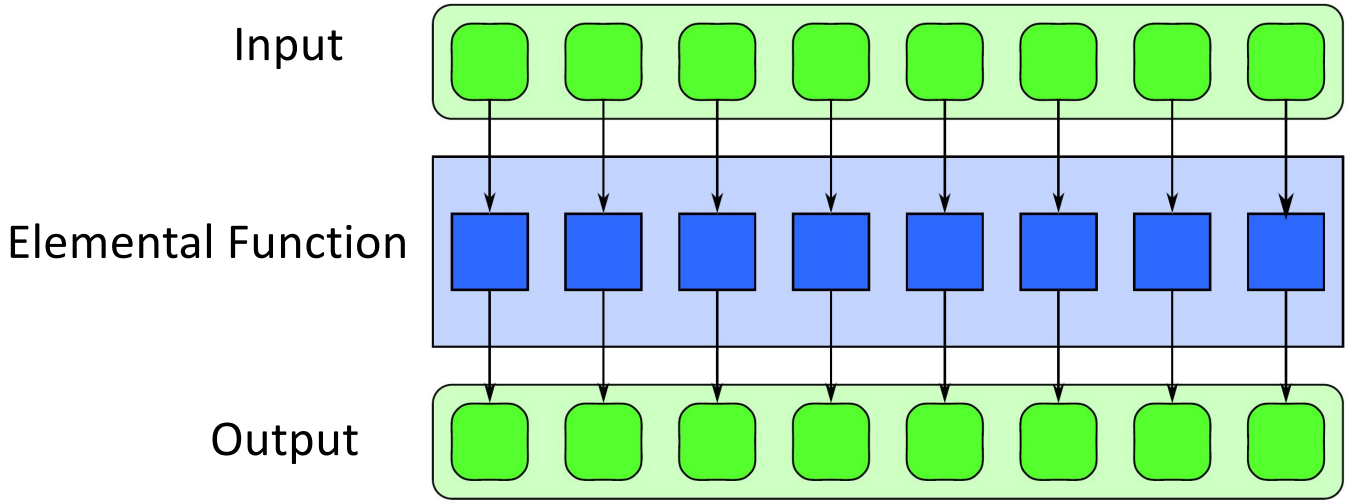
- After join only one thread continues
- Barrier – all threads continue

Parallel Control Patterns: Map

Map: performs a function over every element of a collection

Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection

The replicated function is referred to as an “elemental function”



Mapping

"Do the same thing many times"

```
foreach i in foo:
    do something
```

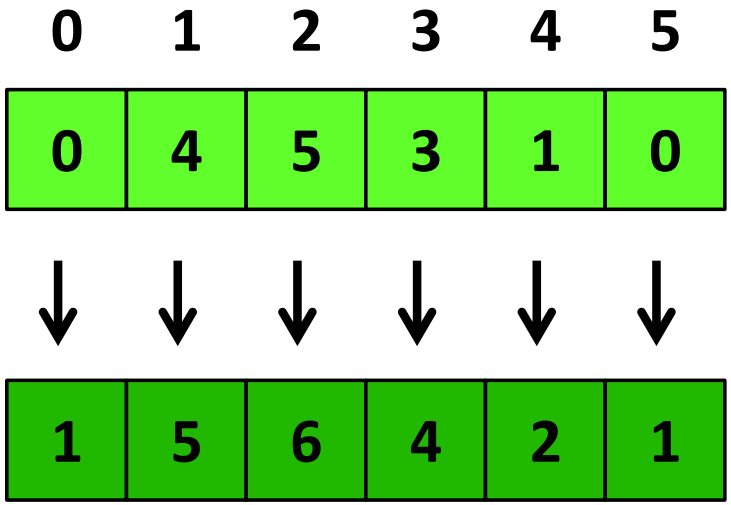
Well-known higher order function in languages like ML, Haskell, Scala

map: $\forall ab.(a \rightarrow b) List\langle a \rangle \rightarrow List\langle b \rangle$

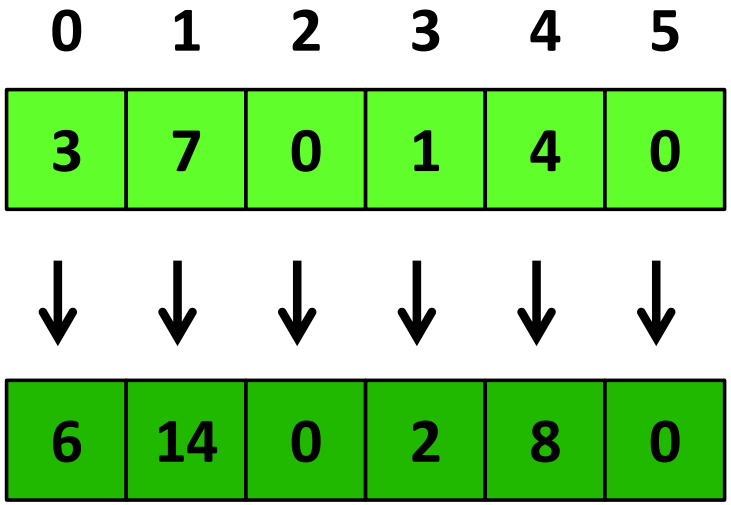
applies a function each element in a list and returns a list of results

Example Maps

Add 1 to every item in an array



Double every item in an array



Key Point: An operation is a map if it can be applied to each element without knowledge of neighbors.

Key Idea

Map is a “foreach loop” where each iteration is **independent**

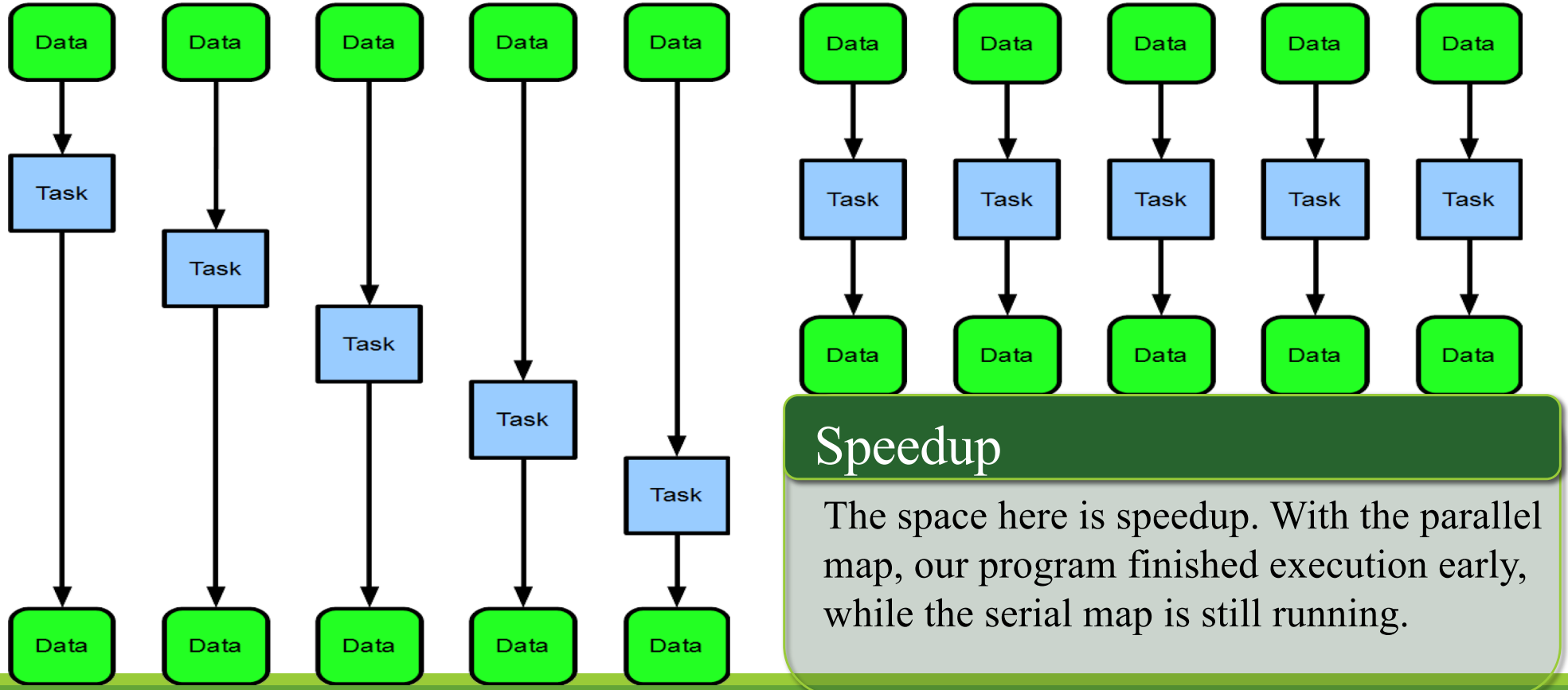
Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(\infty)$ is O(1) plus implementation overhead

Comparing Maps

SERIAL MAP

PARALLEL MAP



Speedup

The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.

Independence

The key to (embarrasing) parallelism is independence

Warning: No shared state!

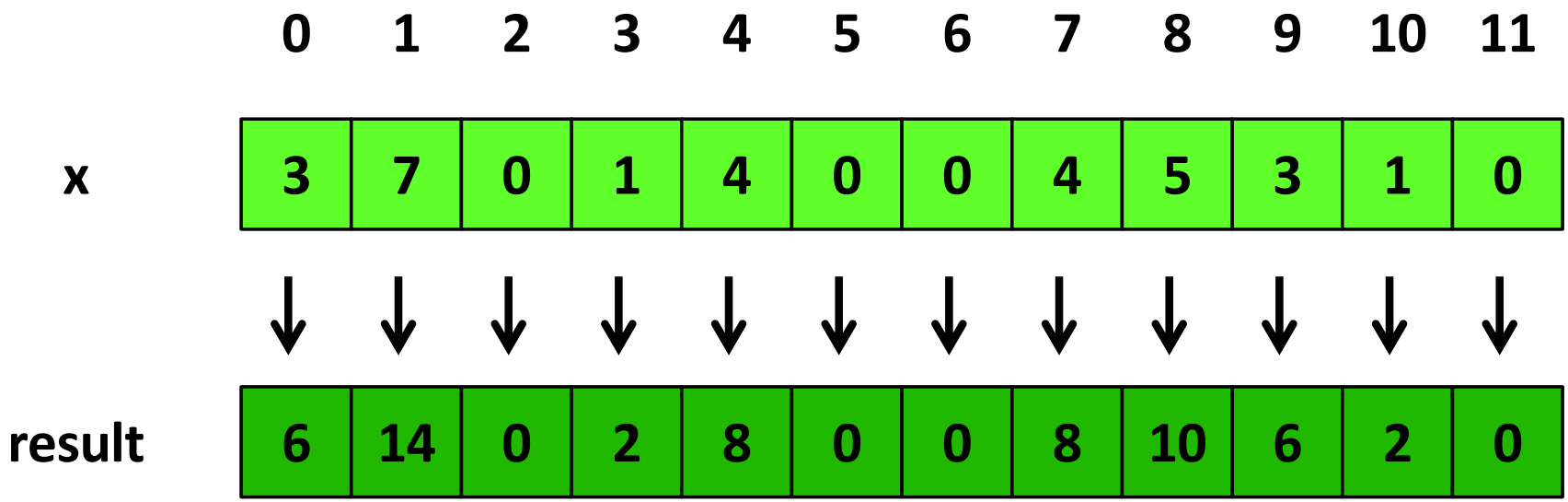
Map function should be “pure” (or “pure-ish”) and should not modify shared states

Modifying shared state breaks perfect independence

Results of accidentally violating independence:

- non-determinism
- data-races
- undefined behavior
- segfaults

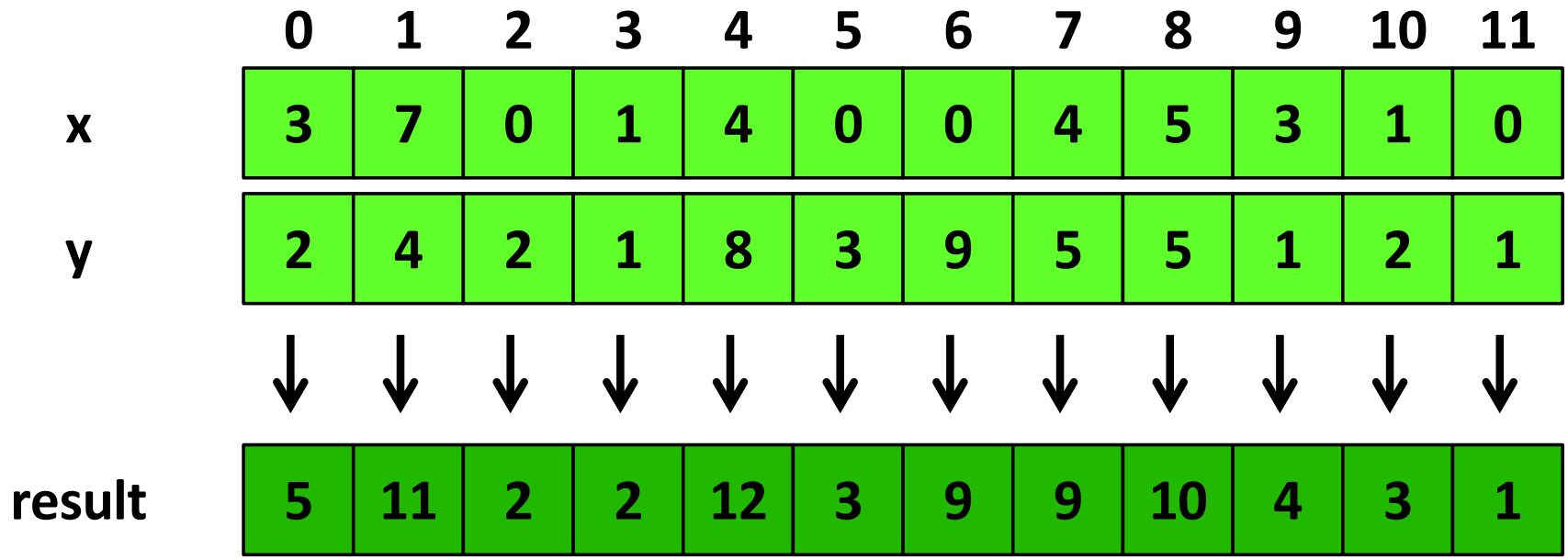
Unary Map: Map with 1 Input, 1 Output



CUDA implementation

```
__global__ void mult2(int* c, int n) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < n)
    c[index] = c[index] * 2;
}
```

N-Array Map: Map with N Inputs, 1 Output



CUDA implementation

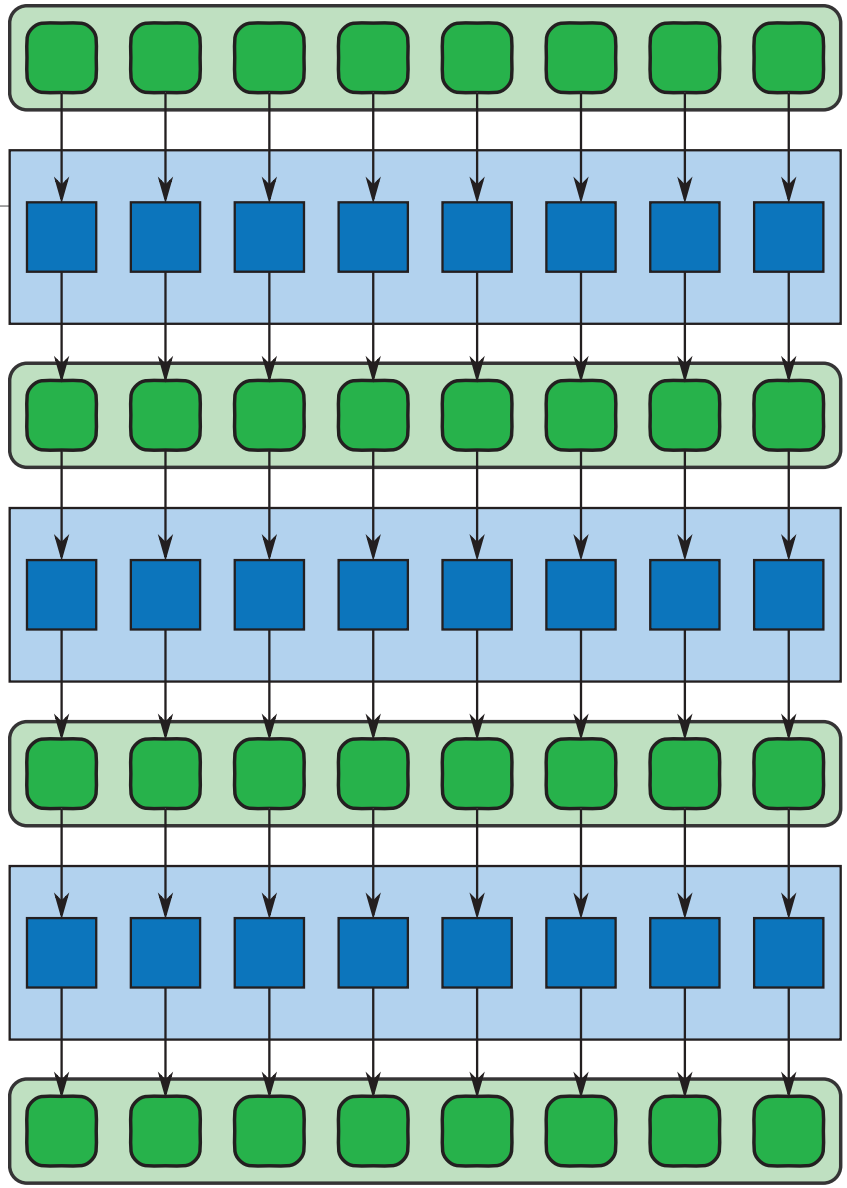
```
__global__ void add(int* a, int* b, int* c, int n) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < n)
    c[index] = a[index] + b[index];
}
```

Optimization Sequences of Maps

Often several map operations occur in sequence

- Vector math consists of many small operations such as additions and multiplications applied as maps

A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache

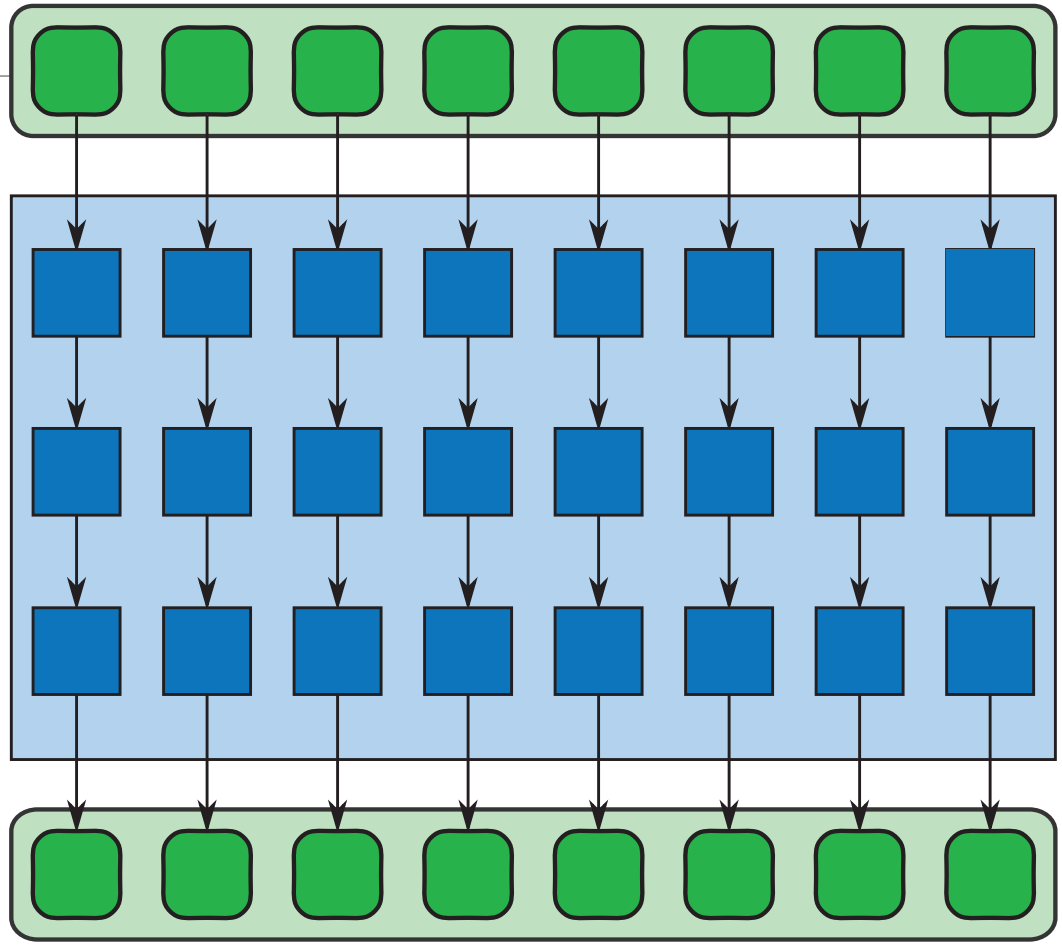


Optimization
Code Fusion

Can sometimes “fuse” together the operations to perform them at once

Adds arithmetic intensity, reduces memory/cache usage

Ideally, operations can be performed using registers alone



Optimization
Code Fusion

C = A + B

C = C - 1

```
__global__ void add(int* a, int* b,
                    int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                blockDim.x;
  if (index < n)
    c[index] = a[index] + b[index];
}

__global__ void sub1(int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                blockDim.x;
  if (index < n)
    c[index] = c[index] – 1;
}
```

C = A + B - 1

```
__global__ void addsub1(int* a, int* b,
                        int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                  blockDim.x;
  if (index < n)  {
    c[index] = a[index] + b[index];
    c[index] = c[index] – 1;
  }
}
```

Optimization
In GPU computing is not always possible

C = A + B

FORALL I>1 C[I] = C[I-1]

```
__global__ void add(int* a, int* b,
                    int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                blockDim.x;
  if (index < n)
    c[index] = a[index] + b[index];
}

__global__ void sub1(int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                blockDim.x;
  if (index > 1 && index < n)
    c[index] = c[índex-1];
}
```

C = A + B - 1

```
__global__ void addsub1(int* a, int* b,
                        int* c, int n) {
  int index = threadIdx.x + blockIdx.x *
                  blockDim.x;
  if (index < n)  {
    c[index] = a[index] + b[index];
    if (índex > 1)
       c[index] = c[índex-1];
  }
}
```

In the general case, it is not possible to know if c[index-1] has already been computed

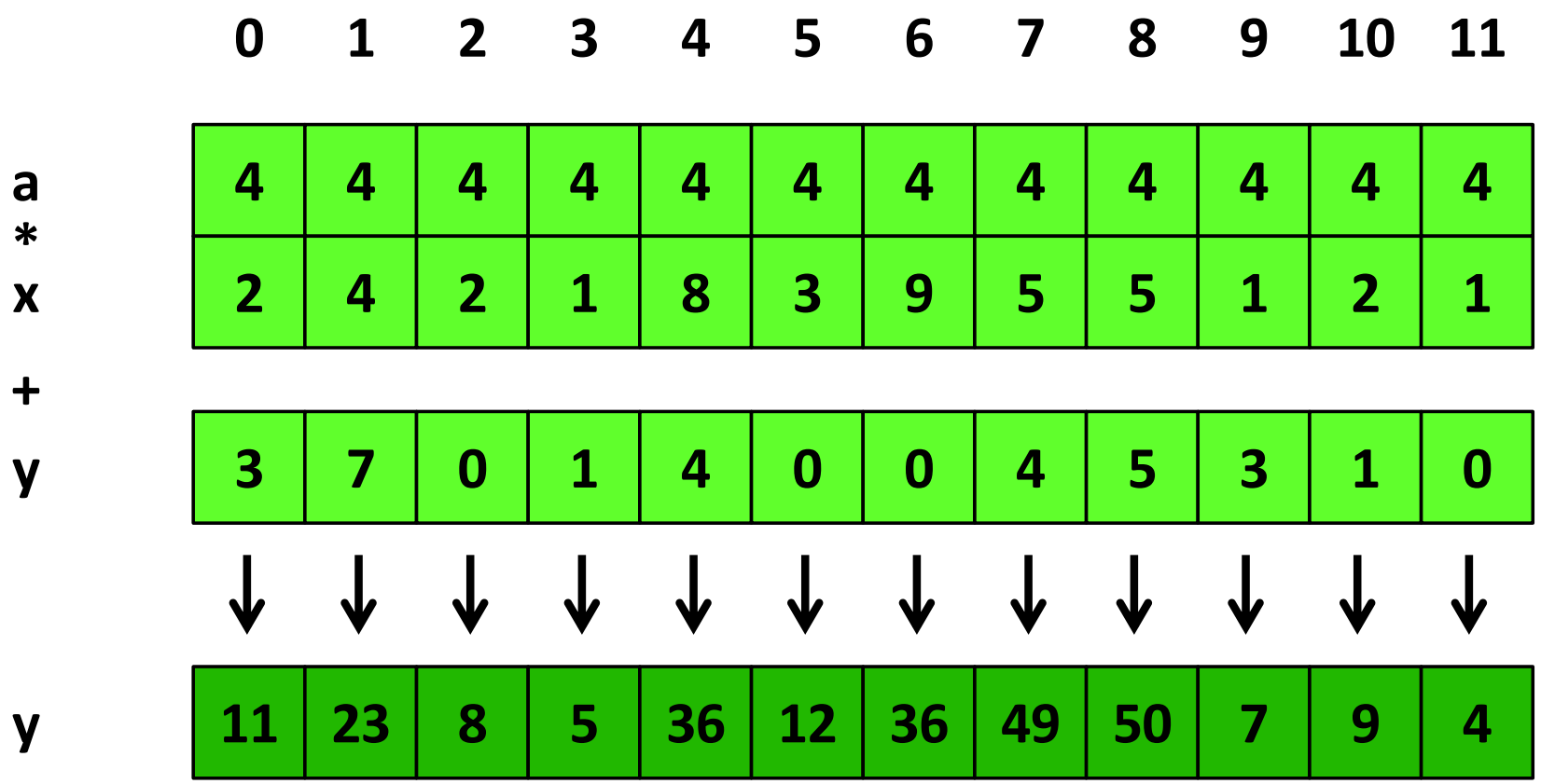
Related Patterns

Three patterns related to map are discussed here:

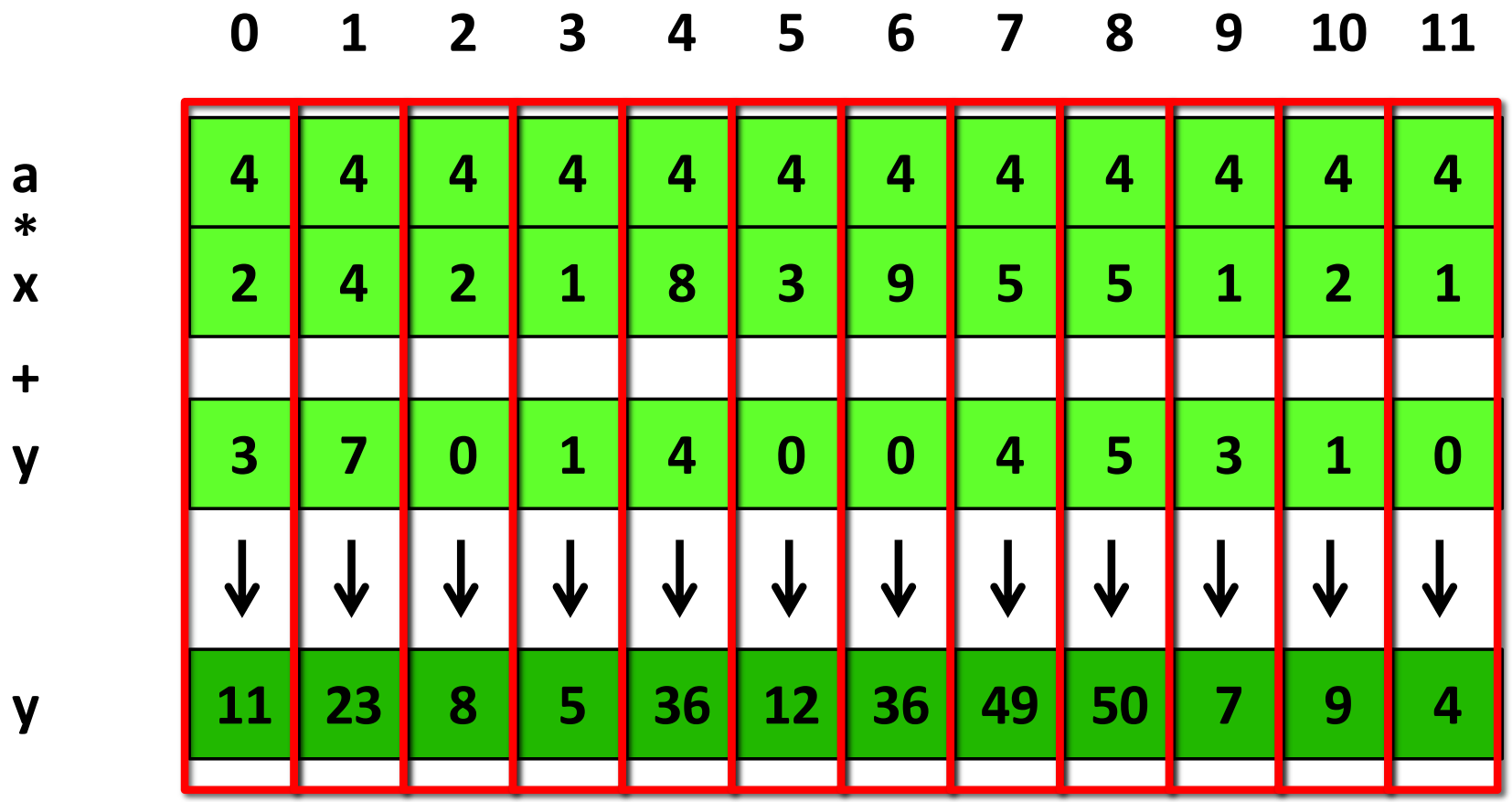
- Stencil
- Workpile
- Divide-and-Conquer

More detail presented in a later lecture

SAXPY: $y \leftarrow ax + y$



SAXPY: $y \leftarrow ax + y$



SAXPY: $y \leftarrow ax + y$