CUDA Programming

Addition of 2 Arrays

Outline

- ❑Mapping
- ❑Addition on the device
    - ❑Moving to parallel using blocks
    - ❑Moving to parallel using threads
    - ❑Combining blocks and threads

Mapping

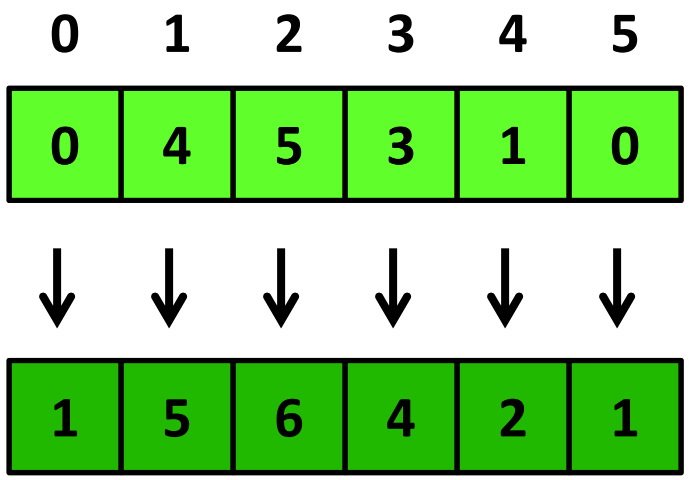
- “Do the same thing many times”

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

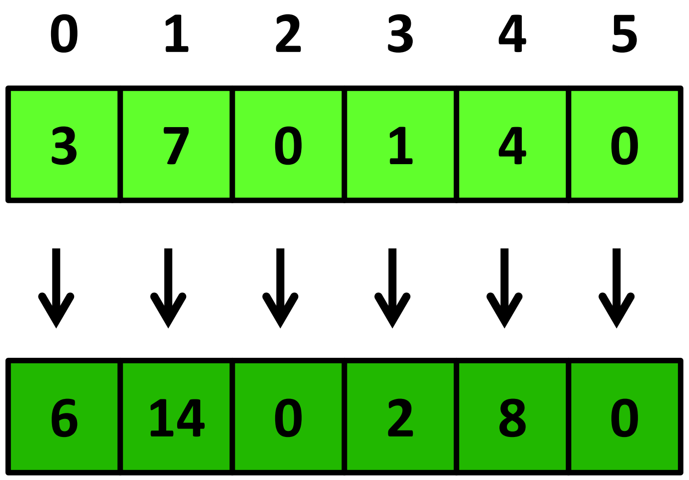
- Well-known higher order function in languages like ML, Haskell, Scala
  - applies a function on each element in a list and returns a list of results

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

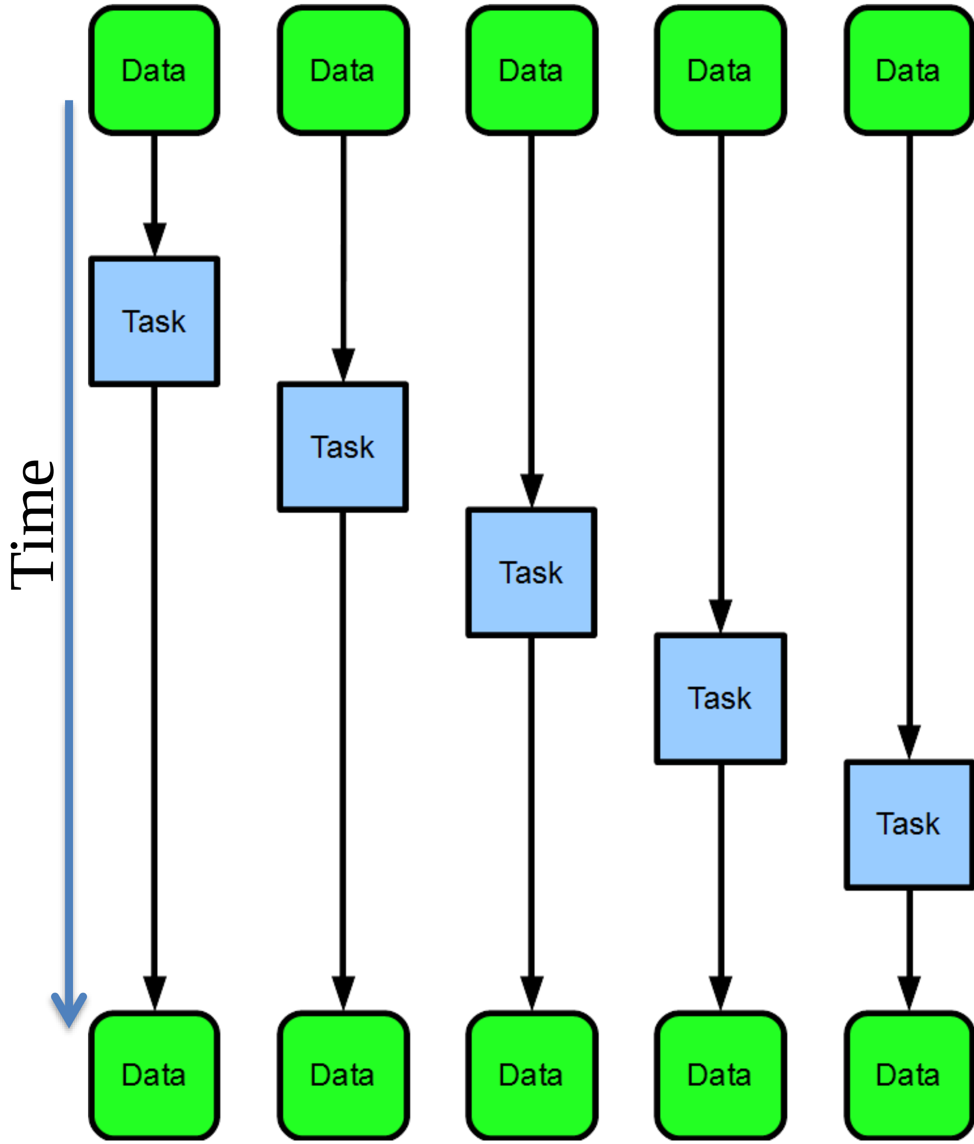
- Map is a "foreach loop" where each iteration is independent

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups!

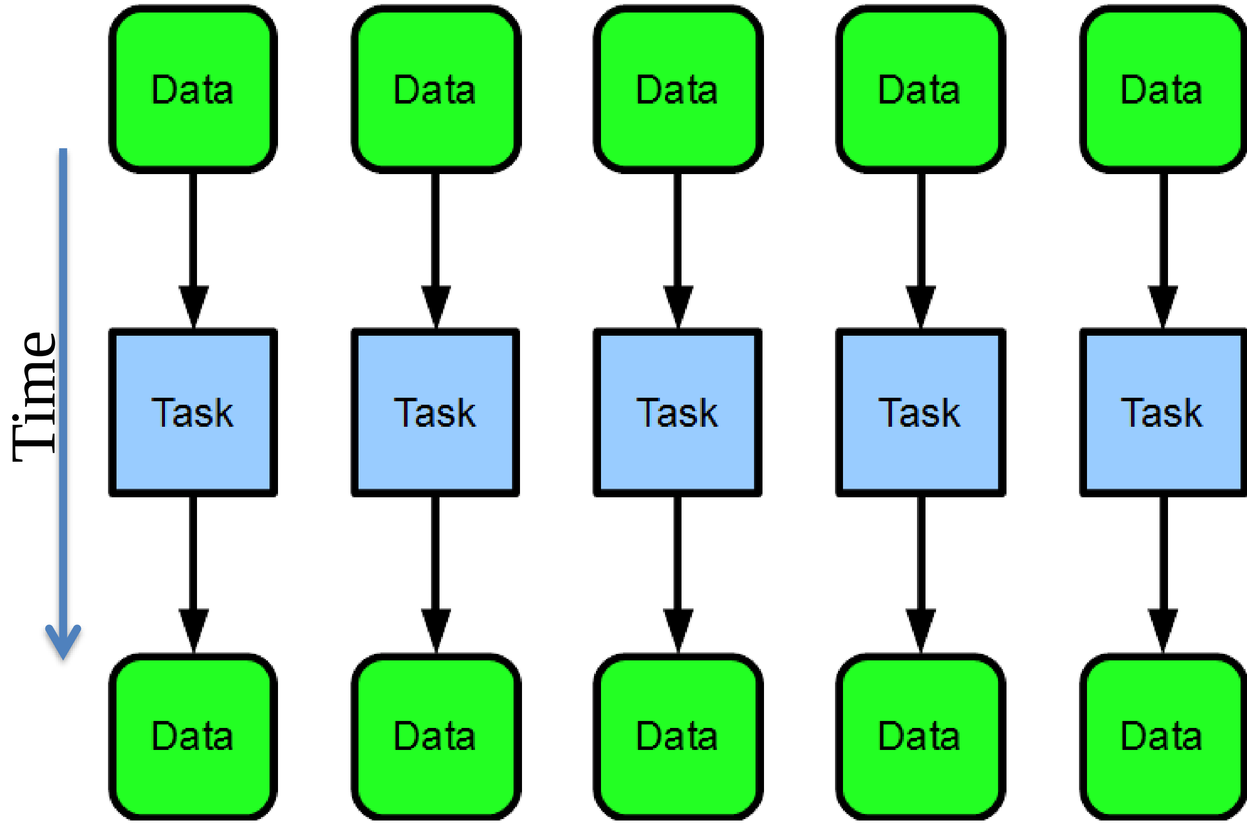
Sequential Map

```
for(int n=0;
       n< array.length;
       ++n){

      process(array[n]);
}
```



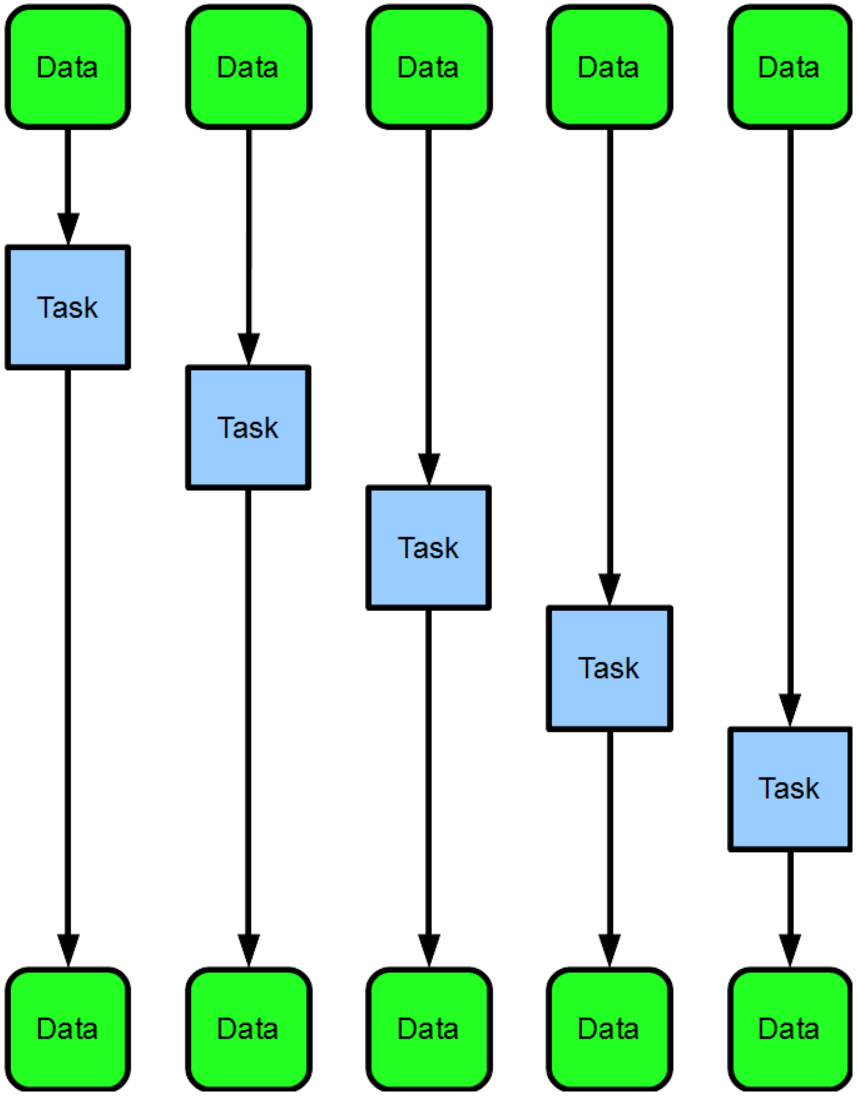
Parallel Map

```
parallel_for_each(
      x in array){

      process(x);
}
```

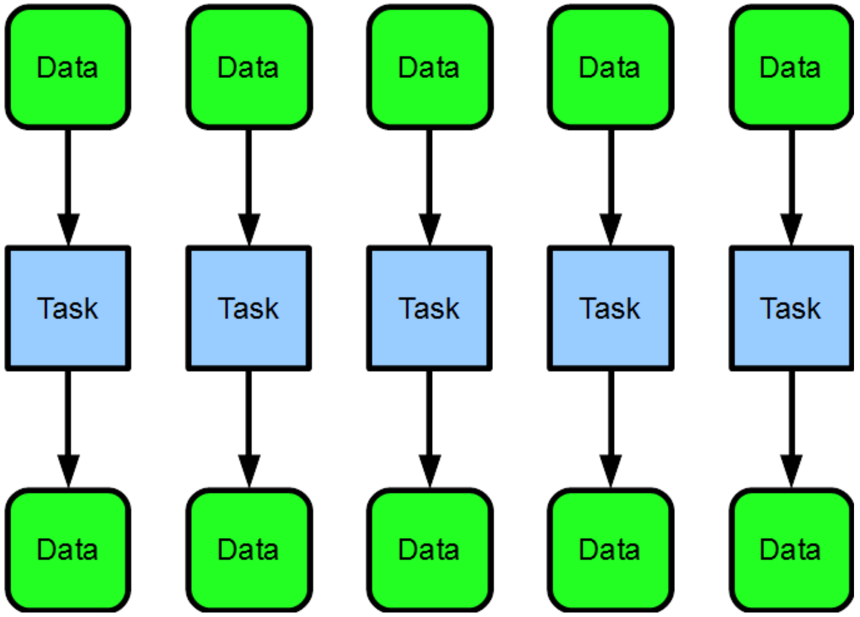


Comparing Maps

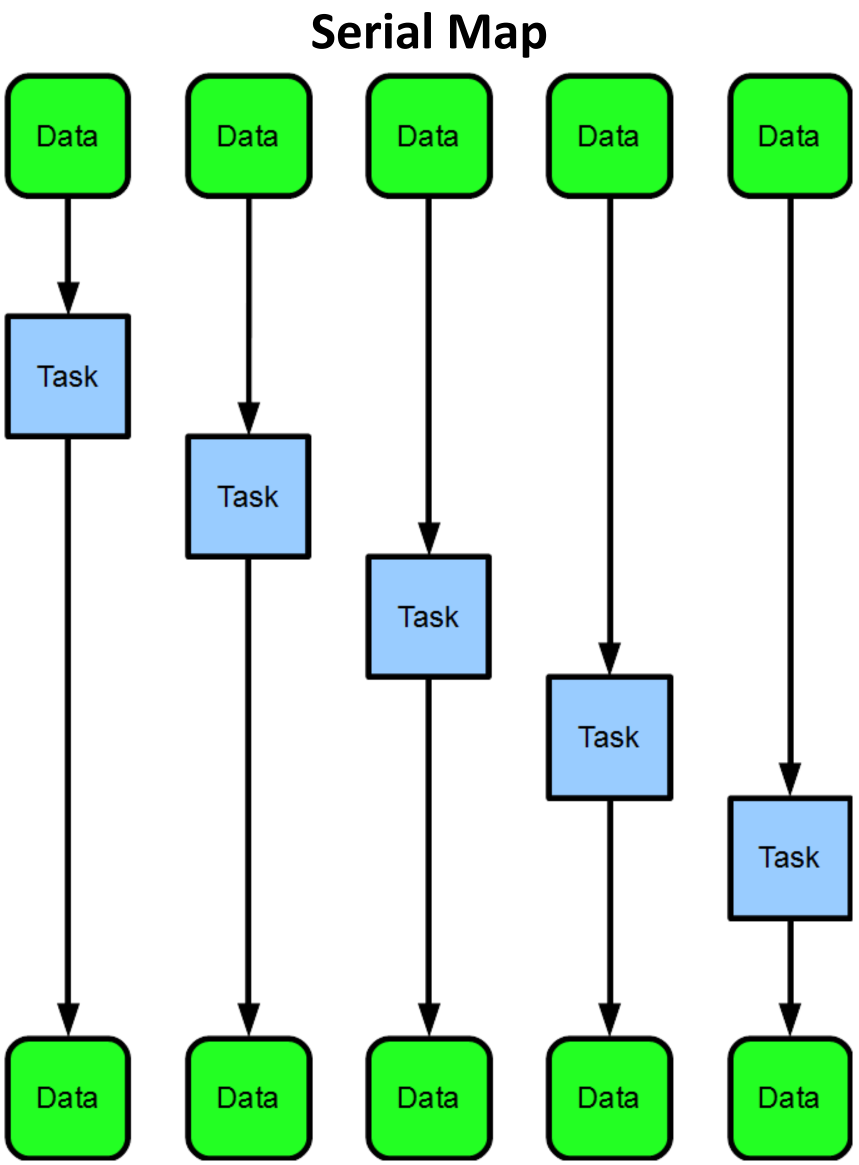
Serial Map

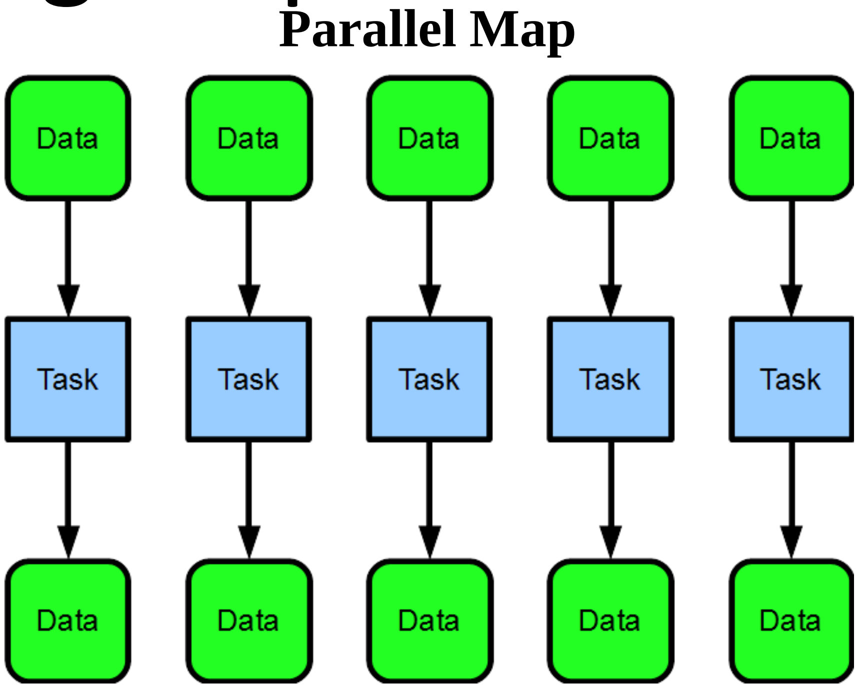


Parallel Map



Comparing Maps





Speedup

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

Independence

Warning: No shared state!

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

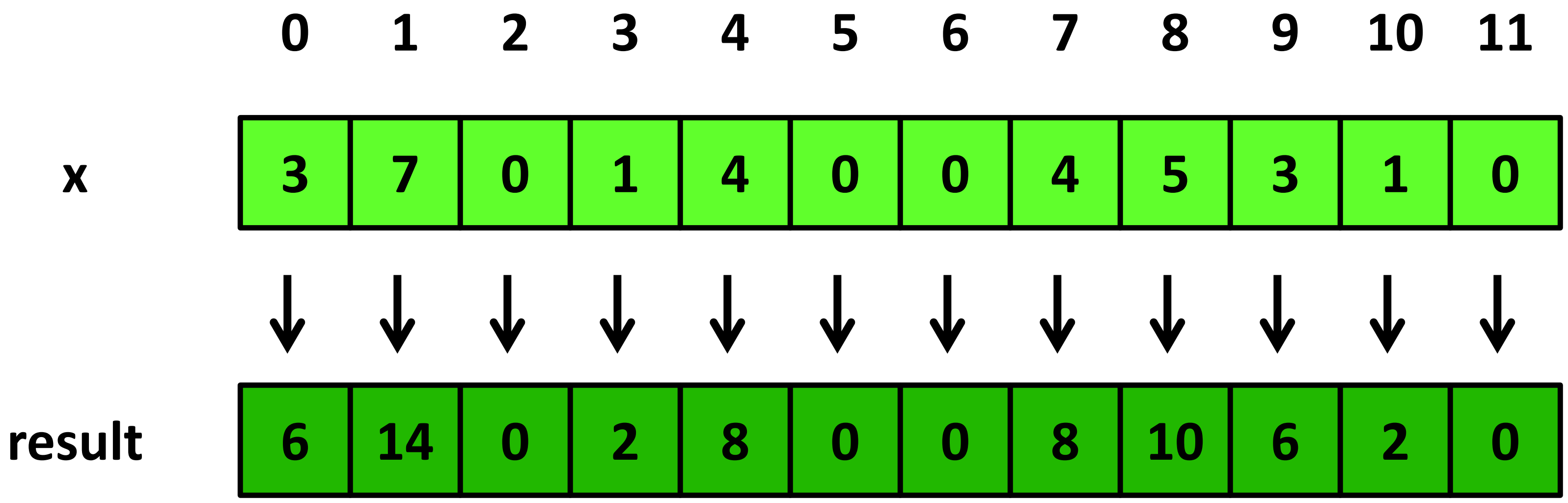
- Modifying shared state breaks perfect independence
- Results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults

Unary Maps

Unary Maps

So far we have only dealt with mapping over a single collection…

Map with 1 Input, 1 Output



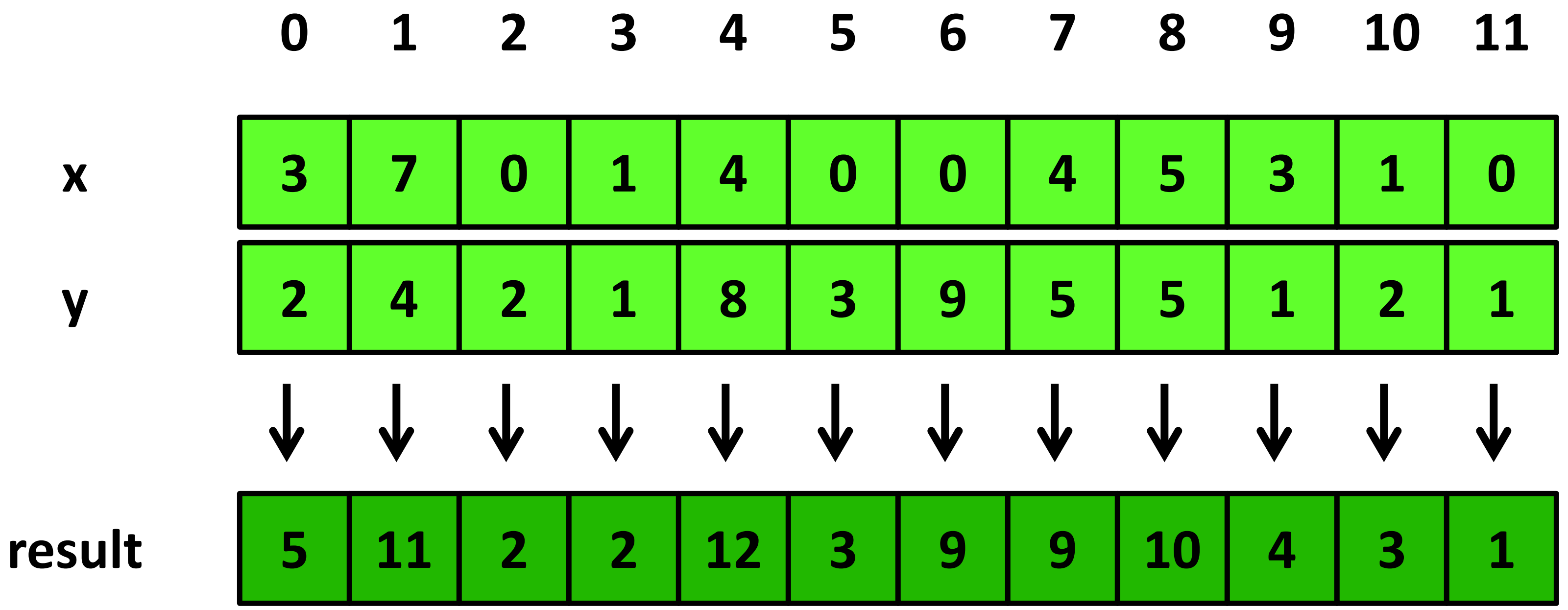
```
int  oneToOne ( int x[11] ) {
        return x*2;
}
```

N-ary Maps

N-ary Maps

But, sometimes it makes sense to map over multiple collections at once…

Map with 2 Inputs, 1 Output



```
int  twoToOne ( int x[11], int y[11] ) {
        return x+y;
}
```

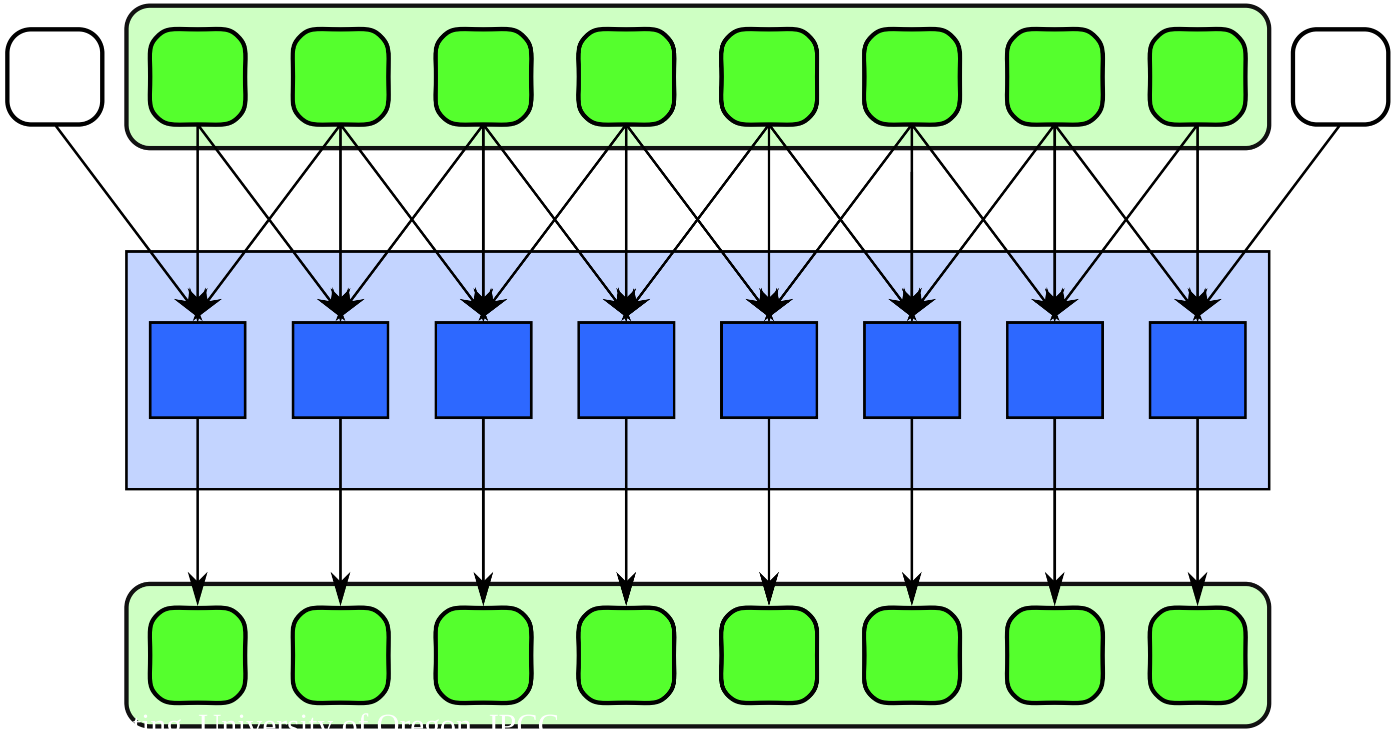
Related Patterns

Two patterns related to map are discussed here:

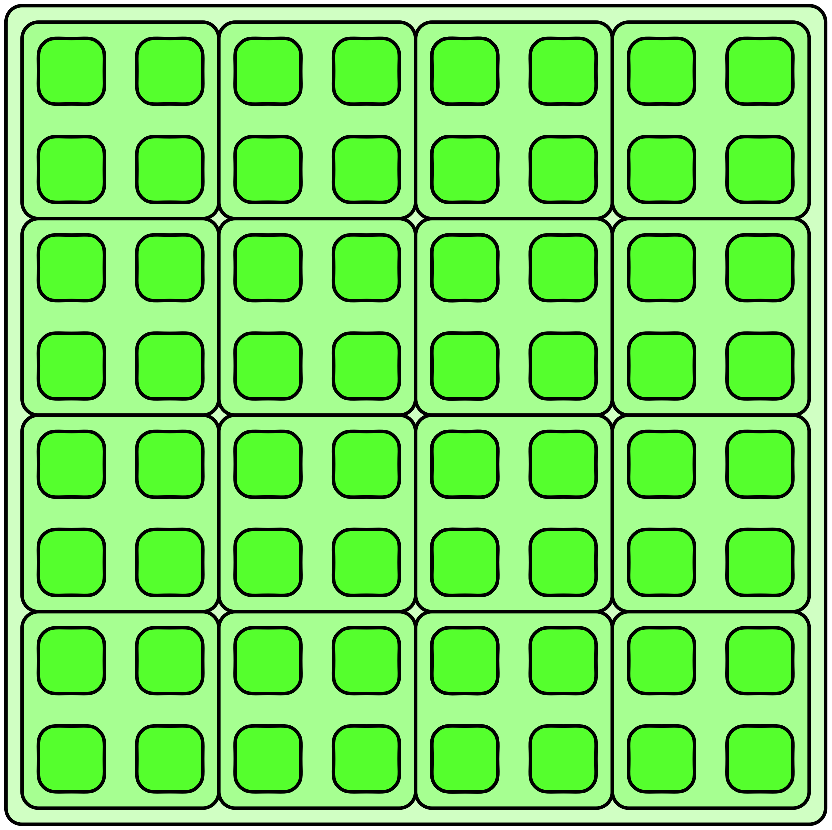
- Stencil
- Divide-and-Conquer

Stencil

- Each instance of the map function accesses neighbors of its input, offset from its usual input



Divide-and-Conquer



- Applies if a problem can be divided into smaller sub-problems recursively until a base case is reached that can be solved serially

Example: Scaled Vector Addition

- $y \neg \; ax + y$
  - Scales vector x by a and adds it to vector y
  - Result is stored in input vector y
- Comes from the BLAS (Basic Linear Algebra Subprograms) library
- **Every element in vector x and vector y are independent**

What does $y \neg \; ax + y$ look like?

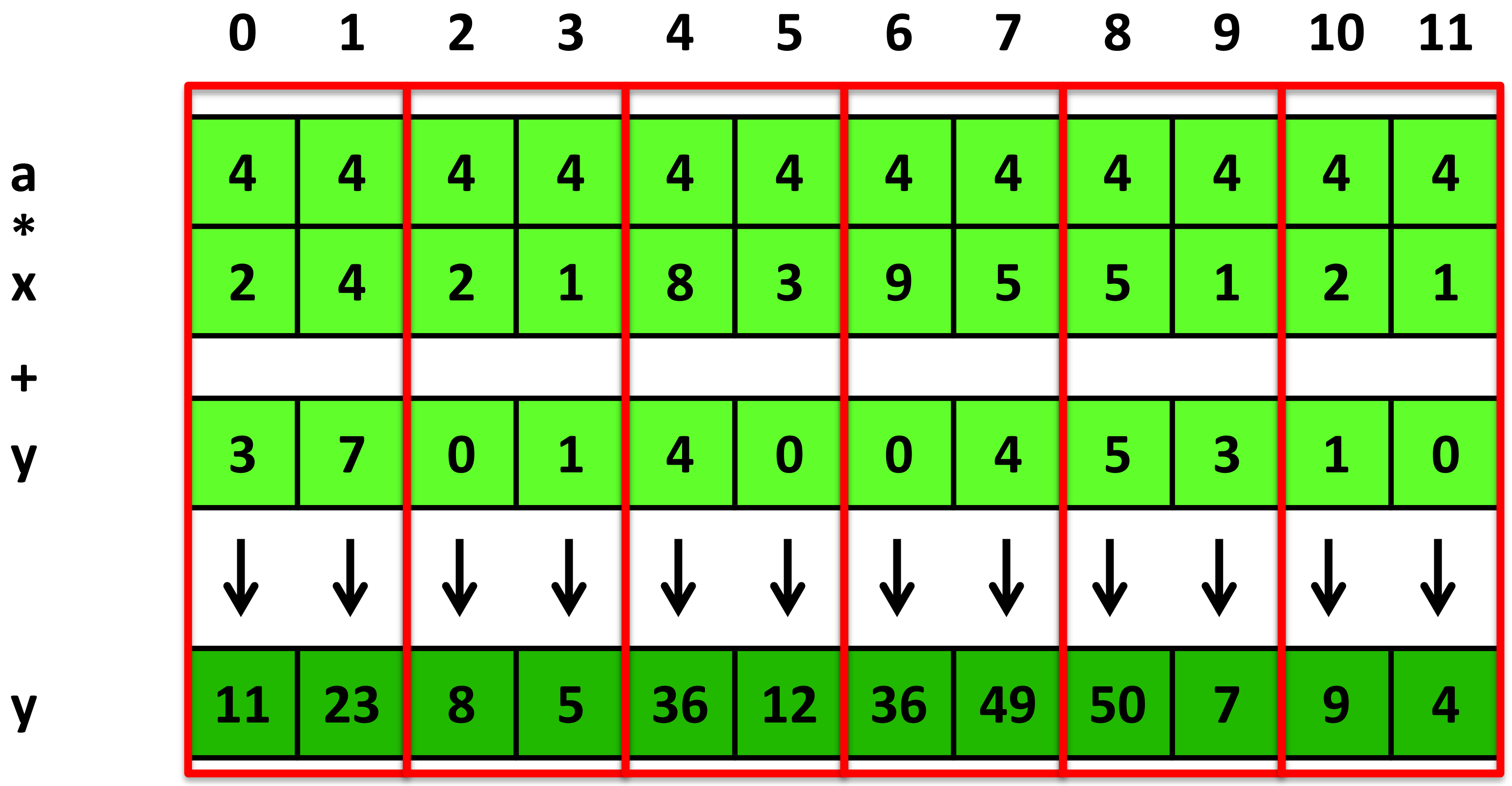
	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Visual: $y \neg \quad ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Twelve processors used → one for each element in the vector

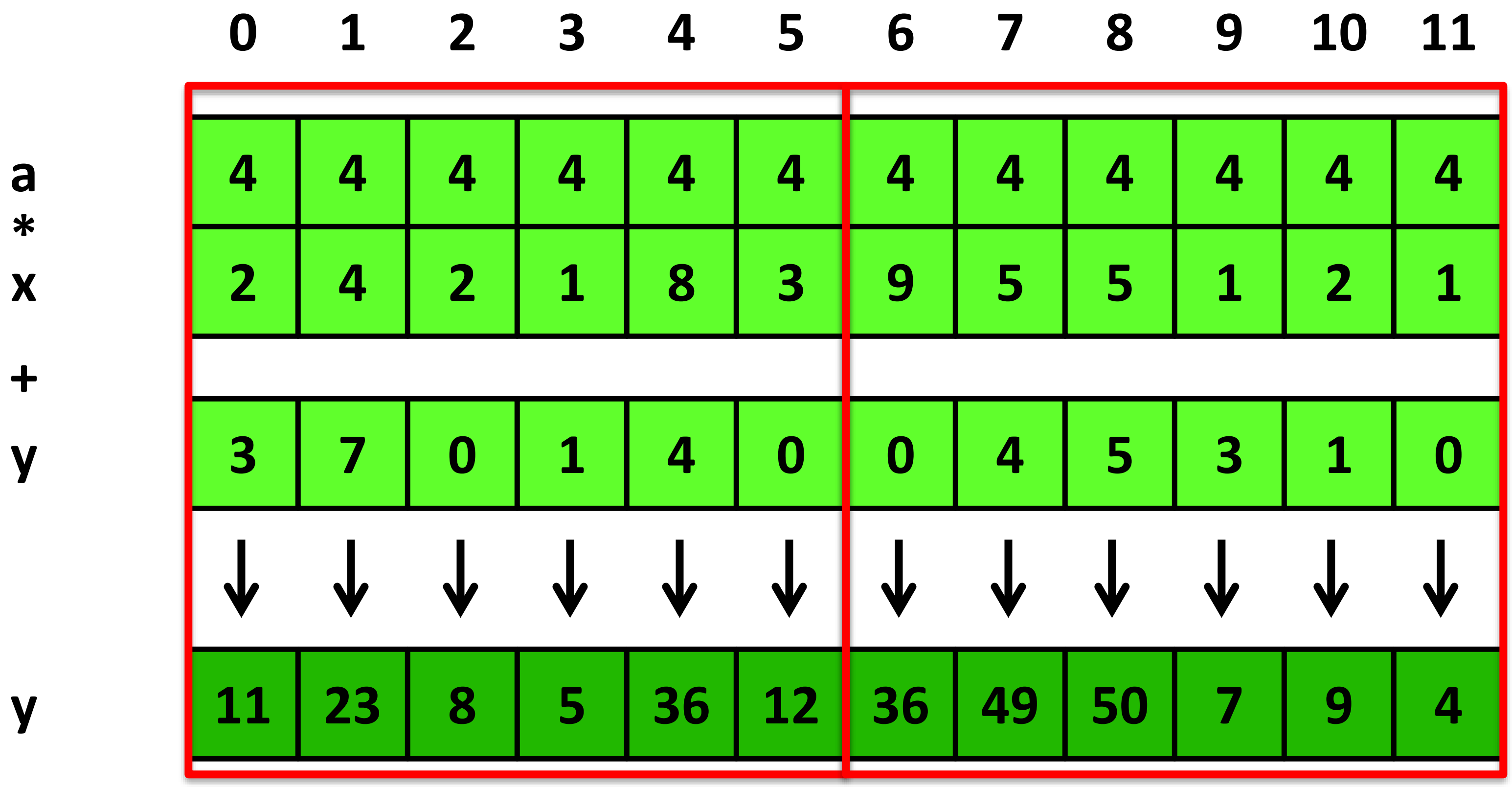
Visual: $y \neg\ ax + y$



	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Six processors used → one for every two elements in the vector

Visual: $y \neg \; ax + y$

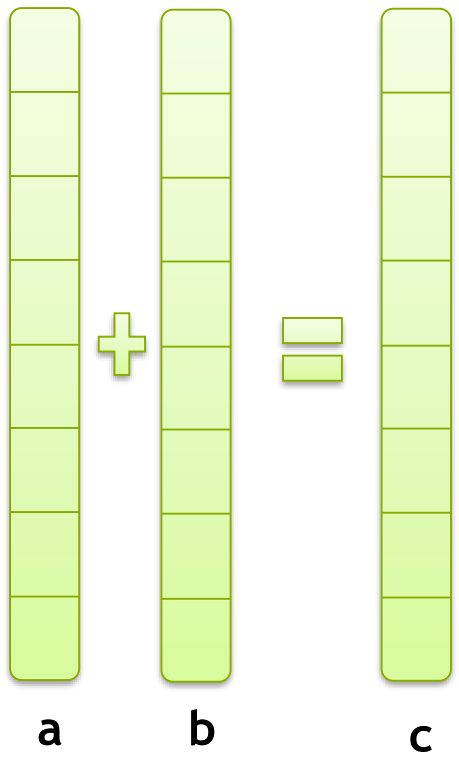


	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Two processors used → one for every six elements in the vector

Parallel Programming in CUDA C/C++

- GPU computing is about massive parallelism!
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
      *c = *a + *b;
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- **`add()`** runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
      *c = *a + *b;
}
```

- Let’s take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

```
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

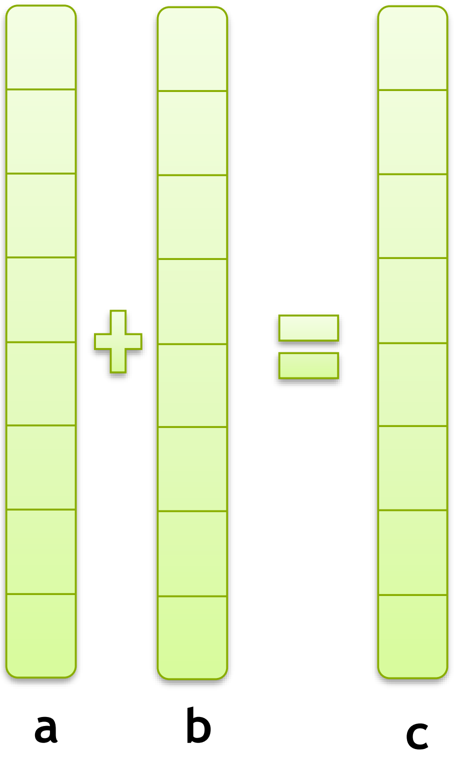
Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```

$\Downarrow$

```
add<<< N, 1 >>>();
```



- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

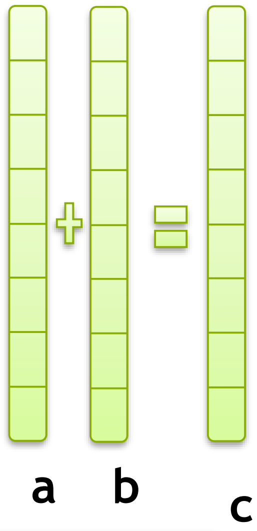
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:



Block 0

```
c[0]   = a[0] + b[0];
```

Block 1

```
c[1]   = a[1] + b[1];
```

Block 2

```
c[2]   = a[2] + b[2];
```

Block 3

```
c[3]   = a[3] + b[3];
```

Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA Threads

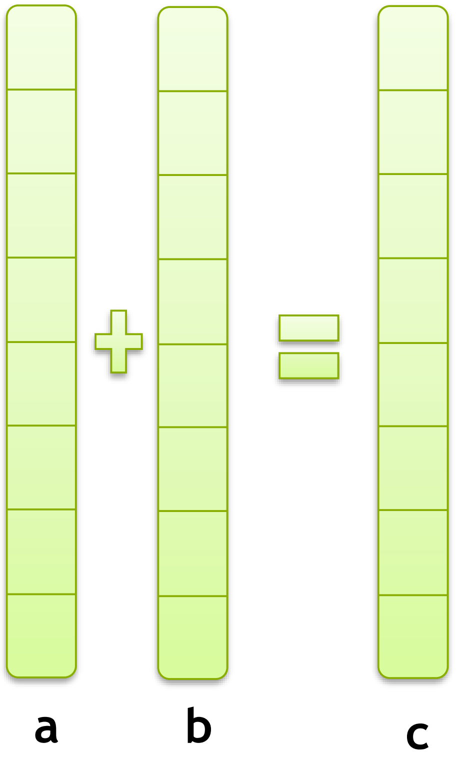
- Terminology: a block can be split into parallel threads

Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c)

}
```

- Need to make one change in `main()`…

```
add<<< 1, 1 >>>();
          ↓
add<<< 1, N >>>();
```



Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a thread
  - Each invocation can refer to its thread index using `threadIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- By using `threadIdx.x` to index into the array, each thread handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- On the device, each thread can execute in parallel:

Thread 0

```
c[0]  = a[0] + b[0];
```

Thread 1

```
c[1]  = a[1] + b[1];
```

Thread 2

```
c[2]  = a[2] + b[2];
```

Thread 3

```
c[3]  = a[3] + b[3];
```

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

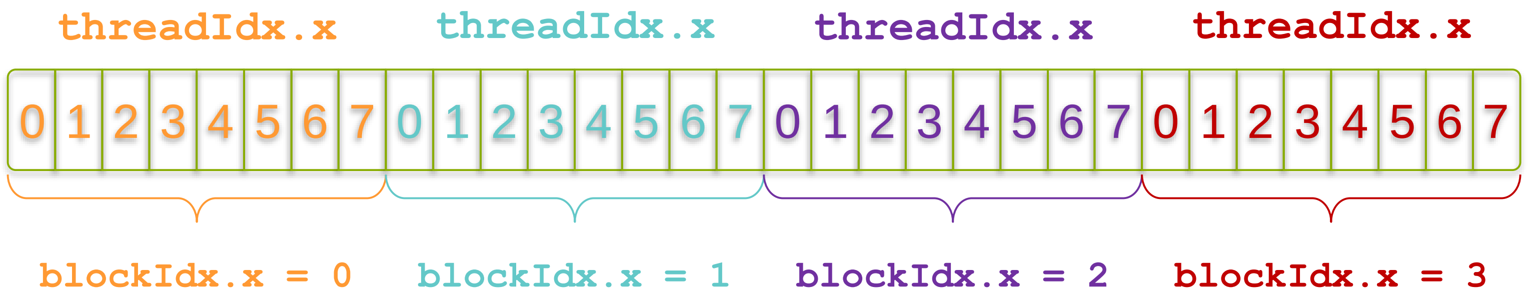
```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<1, N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Combining Blocks and Threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let’s adapt vector addition to use both blocks and threads
- Why? We’ll come to that…
- First let’s discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

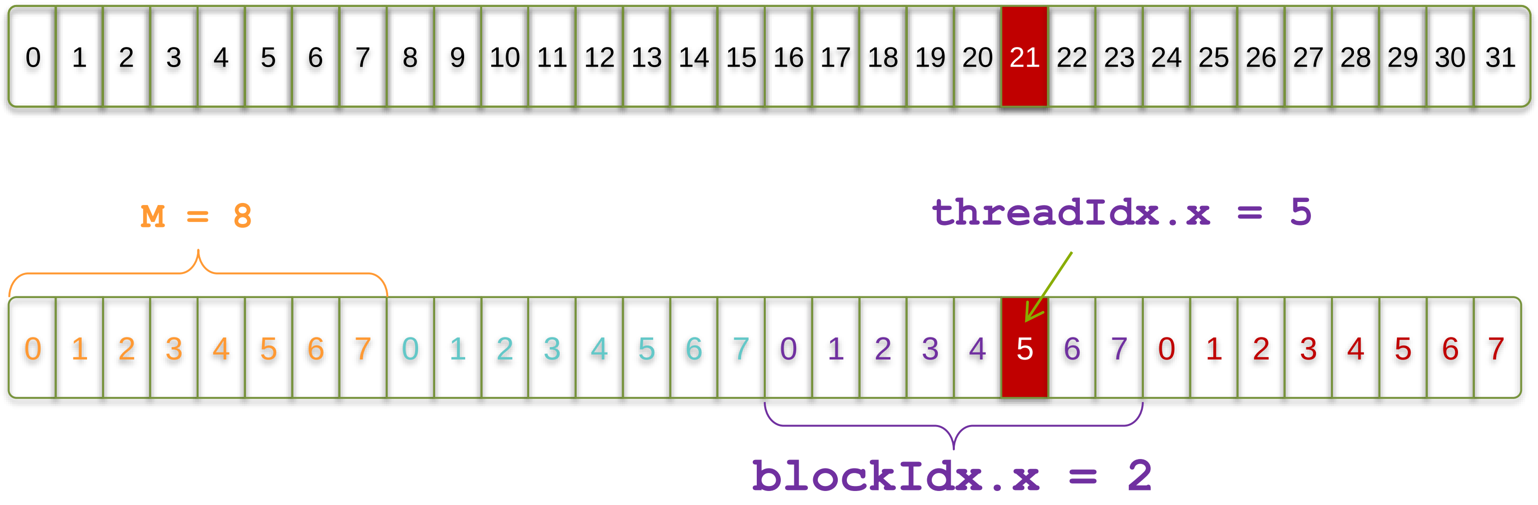


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =         5   +       2     * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                       // host copies of a, b, c
    int *d_a, *d_b, *d_c;            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

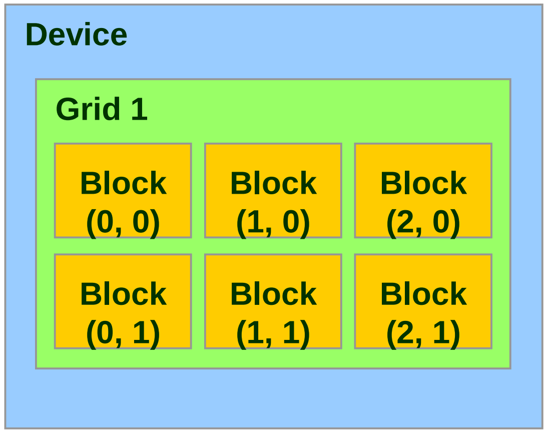
```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

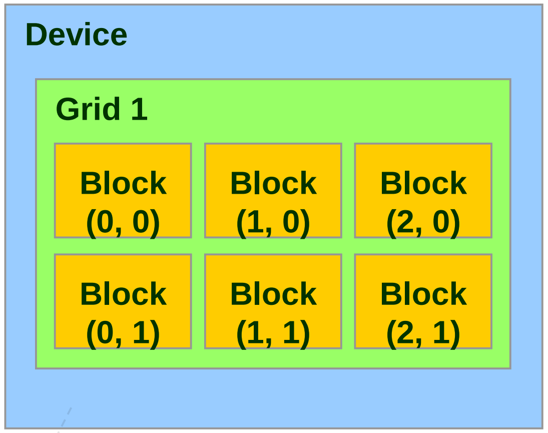
Formatting the grid as a Matrix

- dim3 grid(3,2);
- kernel<<grid, 1>>(…);



Formatting the grid as a Matrix

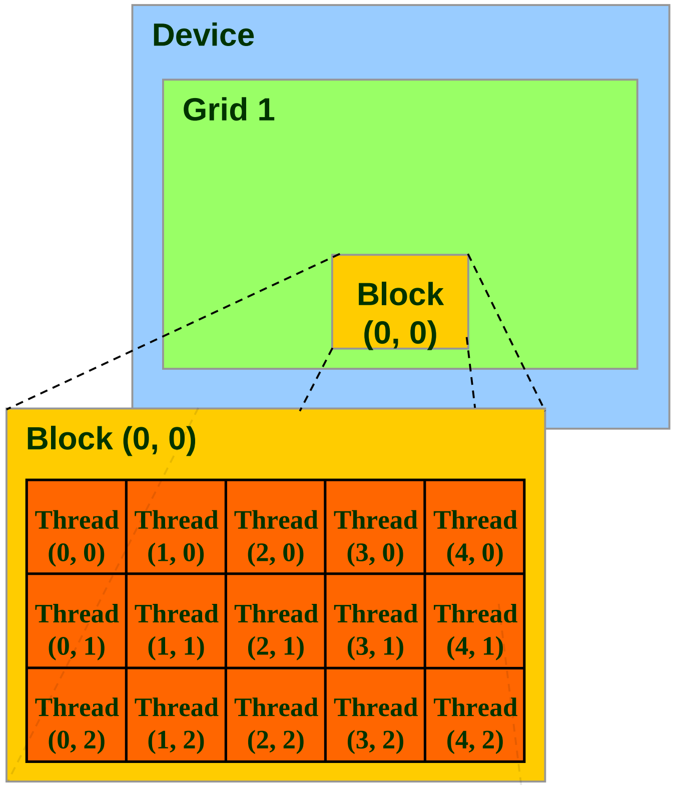
- dim3 grid (3,2);
- kernel<<<grid, 1>>>(…);



- `int index = blockIdx.x + blockIdx.y * gridDim.x;`

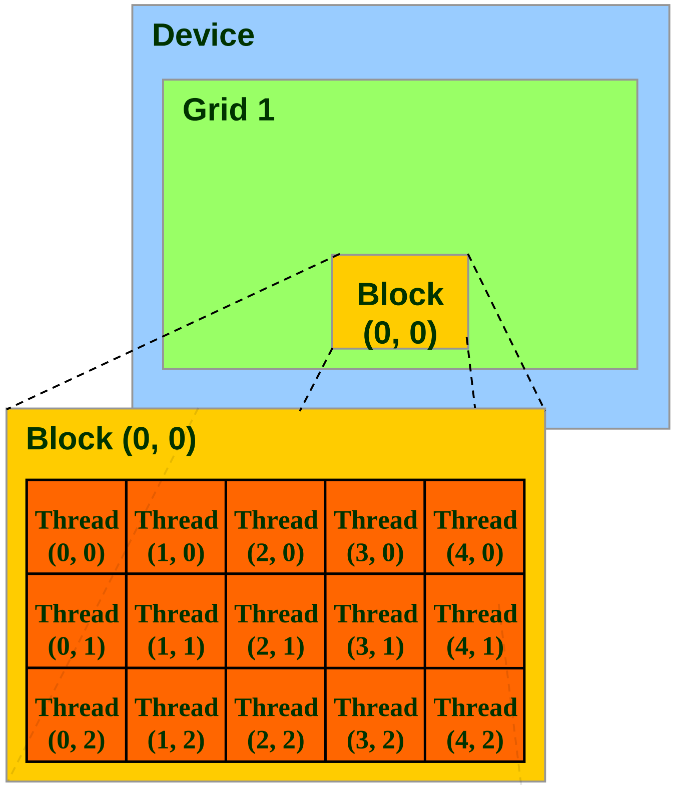
Formatting the grid as a Matrix

- dim3 threads(5,3);
- kernel<<<1, threads>>>(…);



Formatting the grid as a Matrix

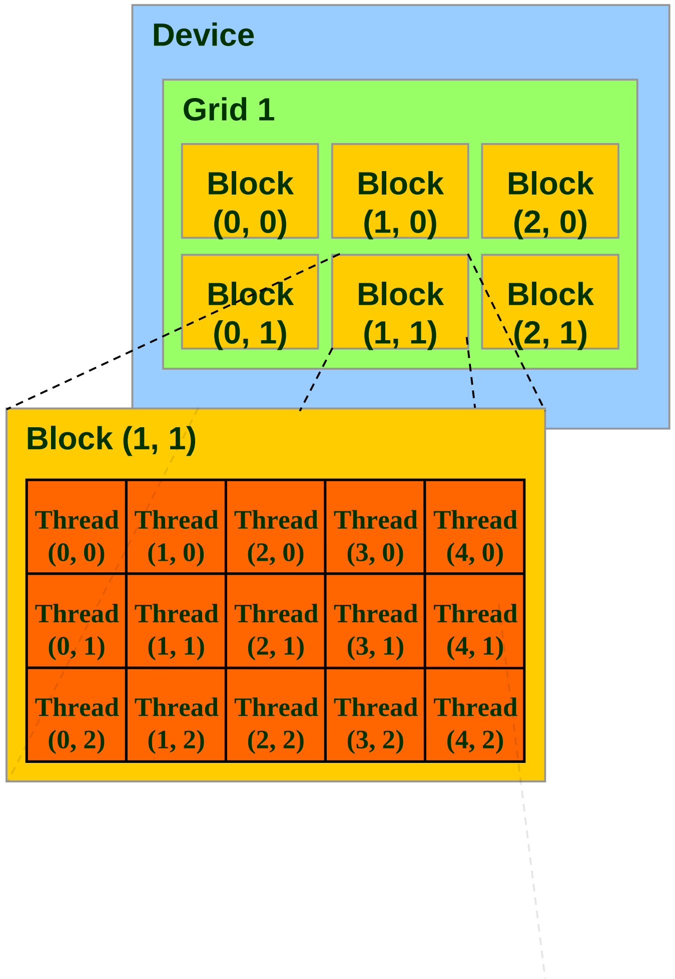
- dim3 threads(5,3);
- kernel<<1, threads>>(…);



- `int index = threadIdx.x + threadIdx.y * blockDim.x;`

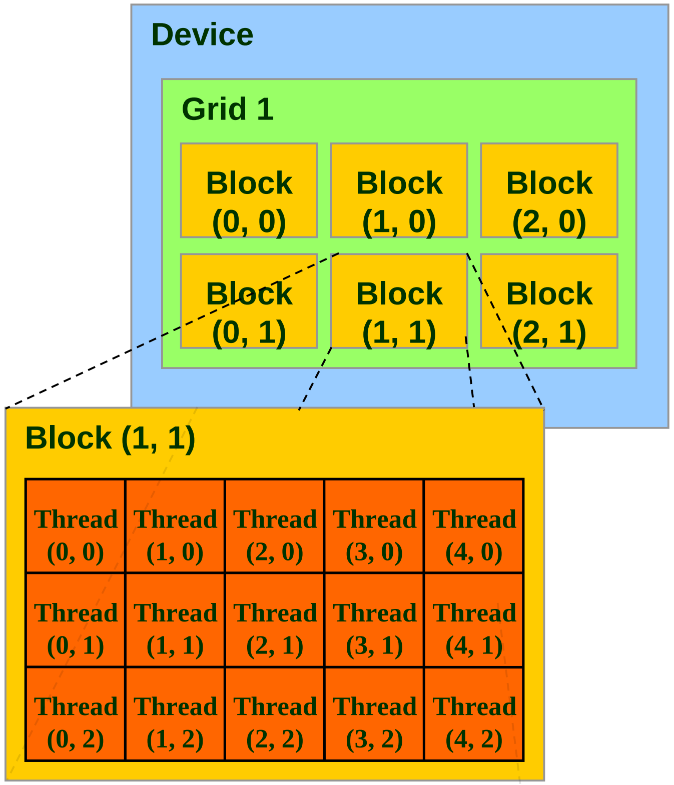
Formatting the grid as a Matrix

- dim3 grid (3,2);
- dim3 block(5,3);
- kernel<<<grid, block>>>(…);



Formatting the grid as a Matrix

- dim3 grid(3,2);
- dim3 block(5,3);
- kernel<<<grid, block>>>(…);



```
•   int index  = (blockIdx.y * gridDim.x + blockIdx.x )
               * (blockDim.x * blockDim.y)
               + threadIdx.y * blockDim.x + threadIdx.x;
```