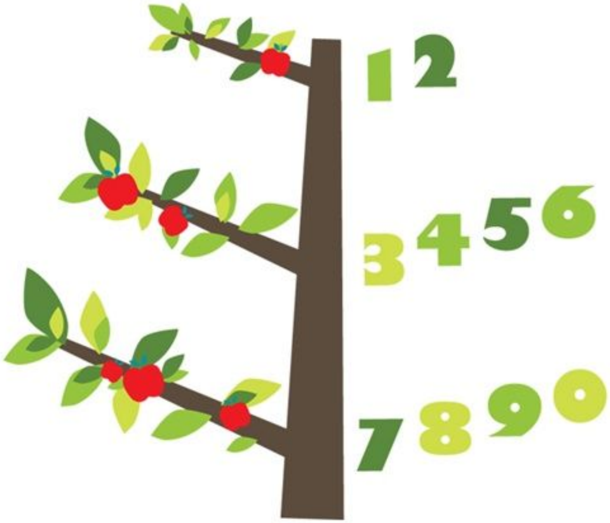
4. Case Study: Sum Reduction



Sum Reduction Problem

- **Sum Reduction:** Adding up the elements of an array
- Sequential approach:

```
float sum = 0;
for (int i = 0; i < n; i++) {
  sum += array[i];
}
```

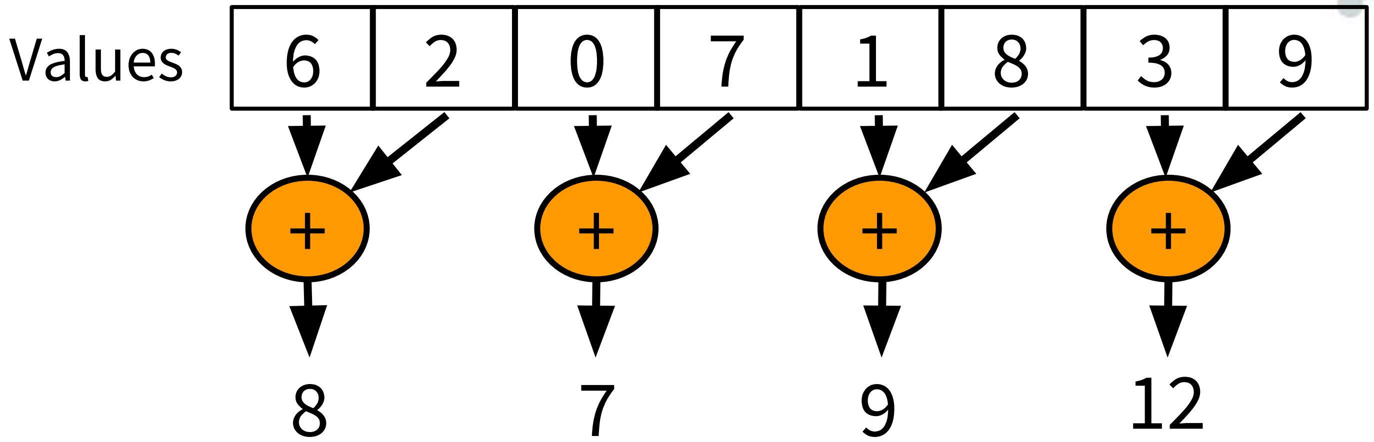
Sequential CPU Reduction

- With $n = 2^{30}$:

Approach	Throughput (MFLOPS)
CPU	**504**

A GPU Algorithm

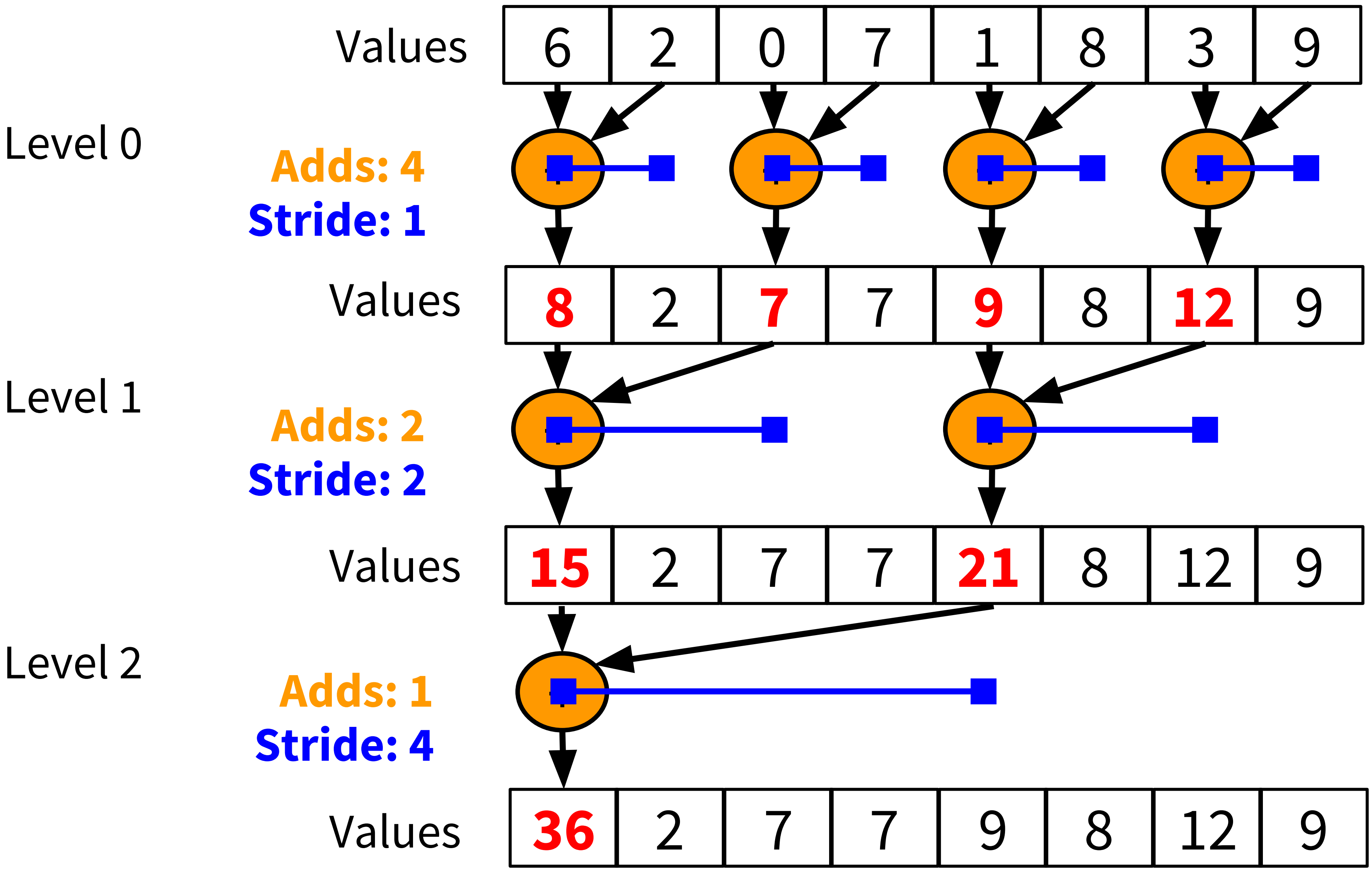
- How can we do this in parallel using threads?



- Idea:

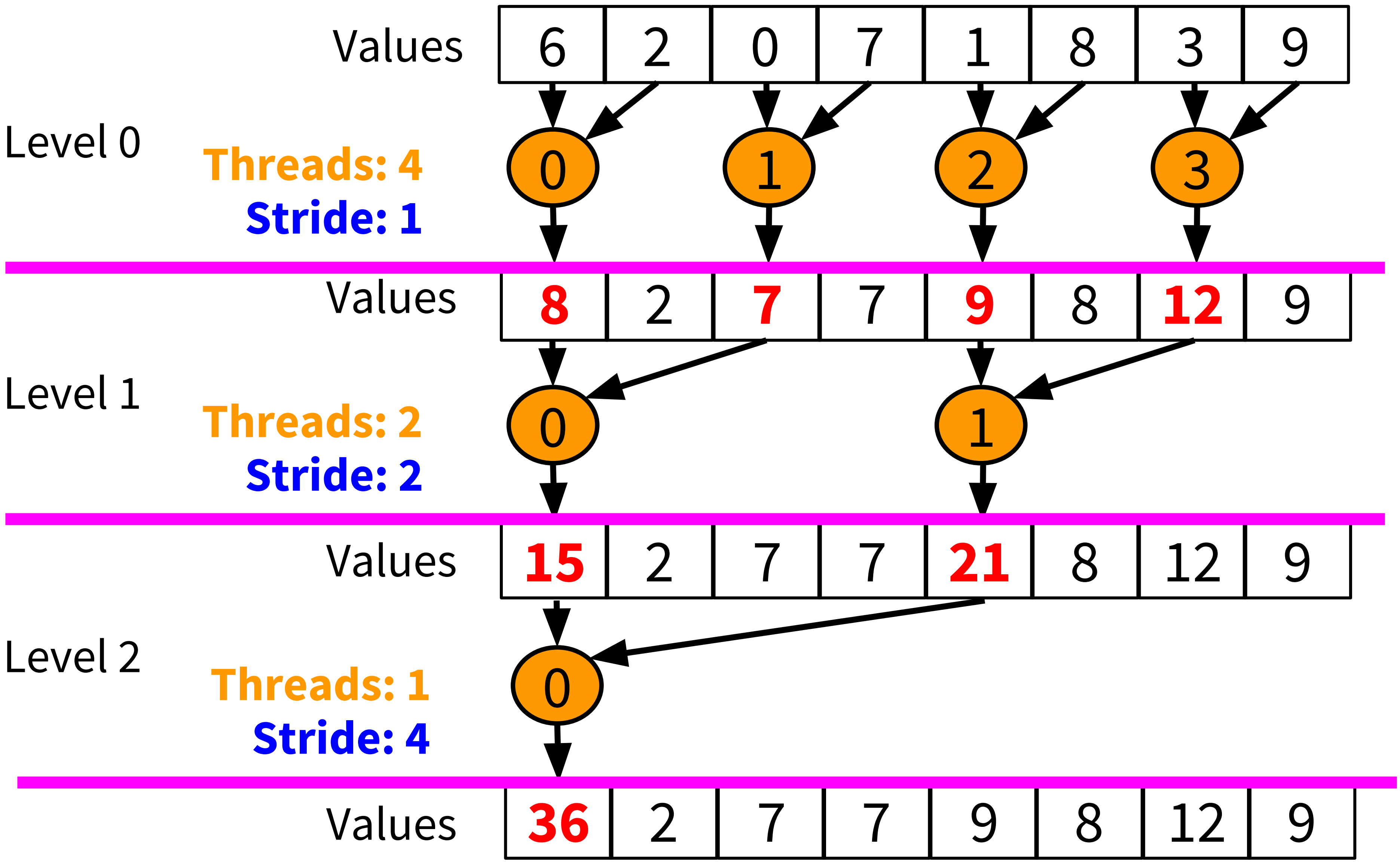
1. Create one thread for every pair of elements
2. All threads add in parallel
   a. This gives us a bunch of partial sums
3. Repeat from (1) using the partial sums
   a. Until we're left with a single value

An In-Place Array Implementation



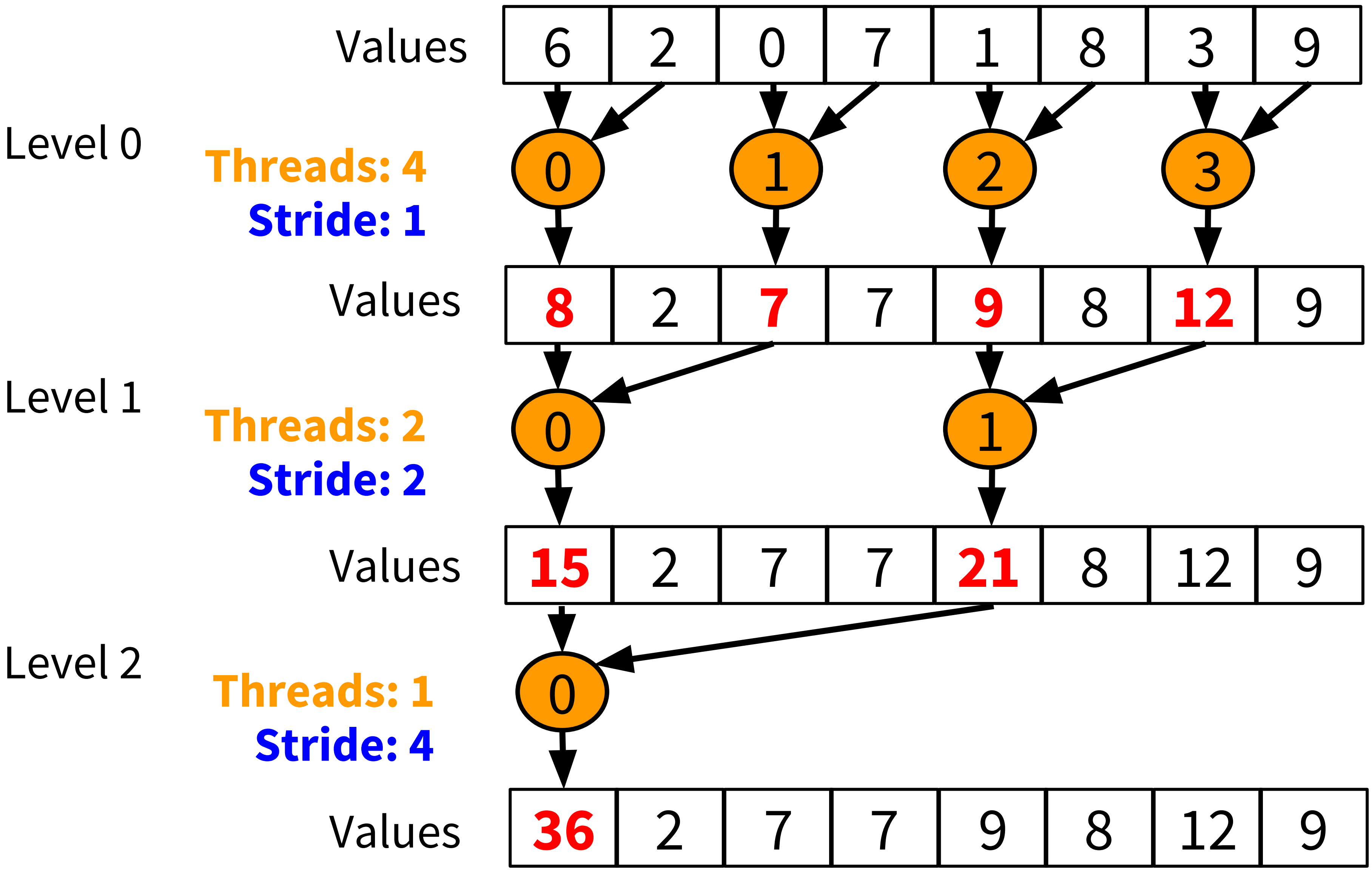
After each level: **Adds** /= 2 , **Stride** *= 2

Synchronization



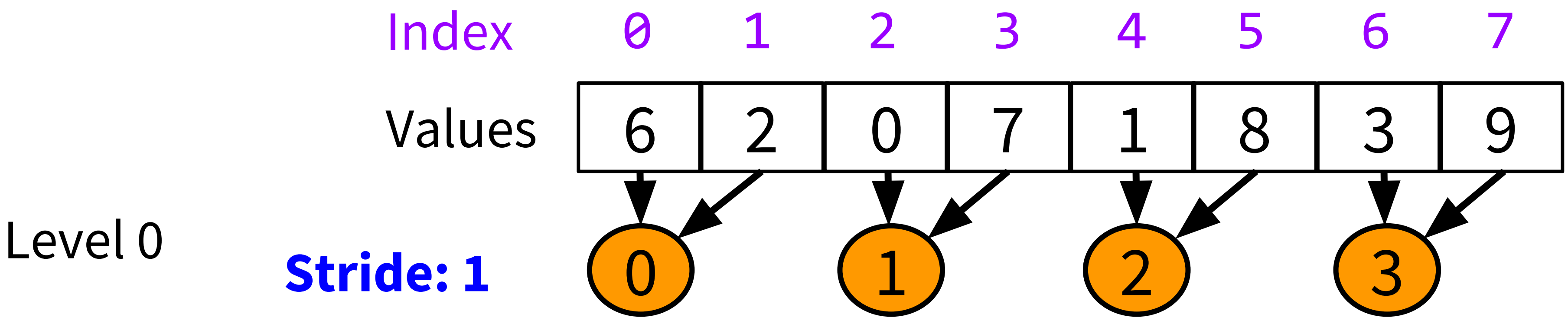
Must sync after each level.

Looping through Levels

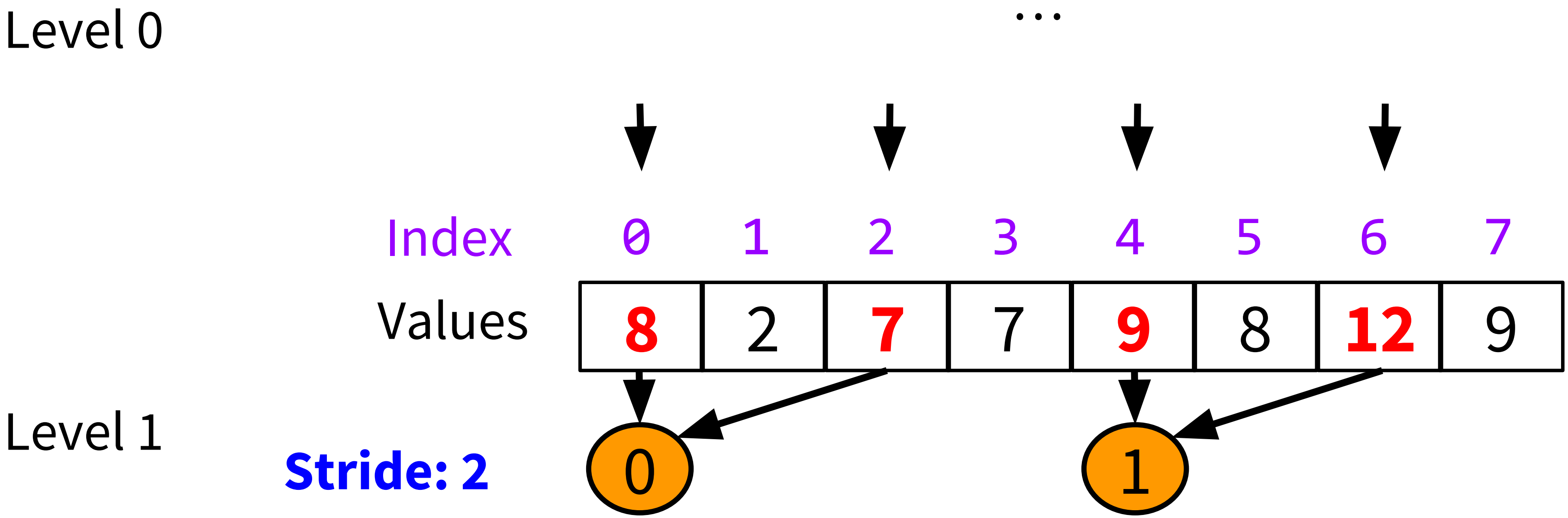



```
for (stride = 1; stride < n; stride *= 2)
```

Calculating Indices



- What's the index of each left number?
  - `0, 2, 4, 6`
- Say we're thread 1. How can we calculate our left index from our id?
  - `Multiply by 2`
- Generalizing, if we're thread id:
  - `left = id * 2`

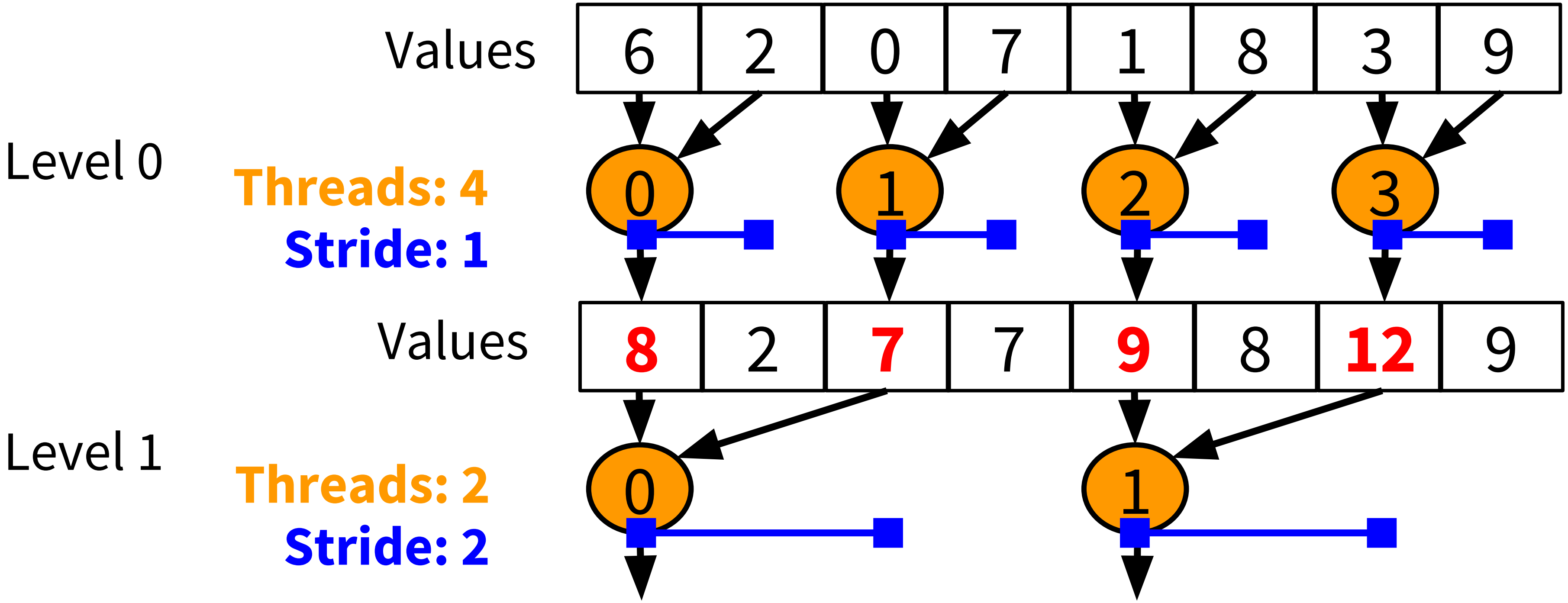


- What are the left indices at level 1?
  - `0, 4`
- Our pattern is `left = id * 2`. Does it work here?
  - No.
  - What should the pattern be here?
  - `left = id * 4`

A General Formula

- We have:
  - level 0: `left = id * 2`
  - level 1: `left = id * 4`
- What changes between levels?
  - Stride:
    - At level 0, stride = 1
    - At level 1, stride = 2

```
left = id * (stride * 2)
```



- So we have:

```
left = id * (stride * 2)
```

- What about a formula for the right index?
  - If we know `left`, then:

```
right = left + stride
```

Writing a Kernel

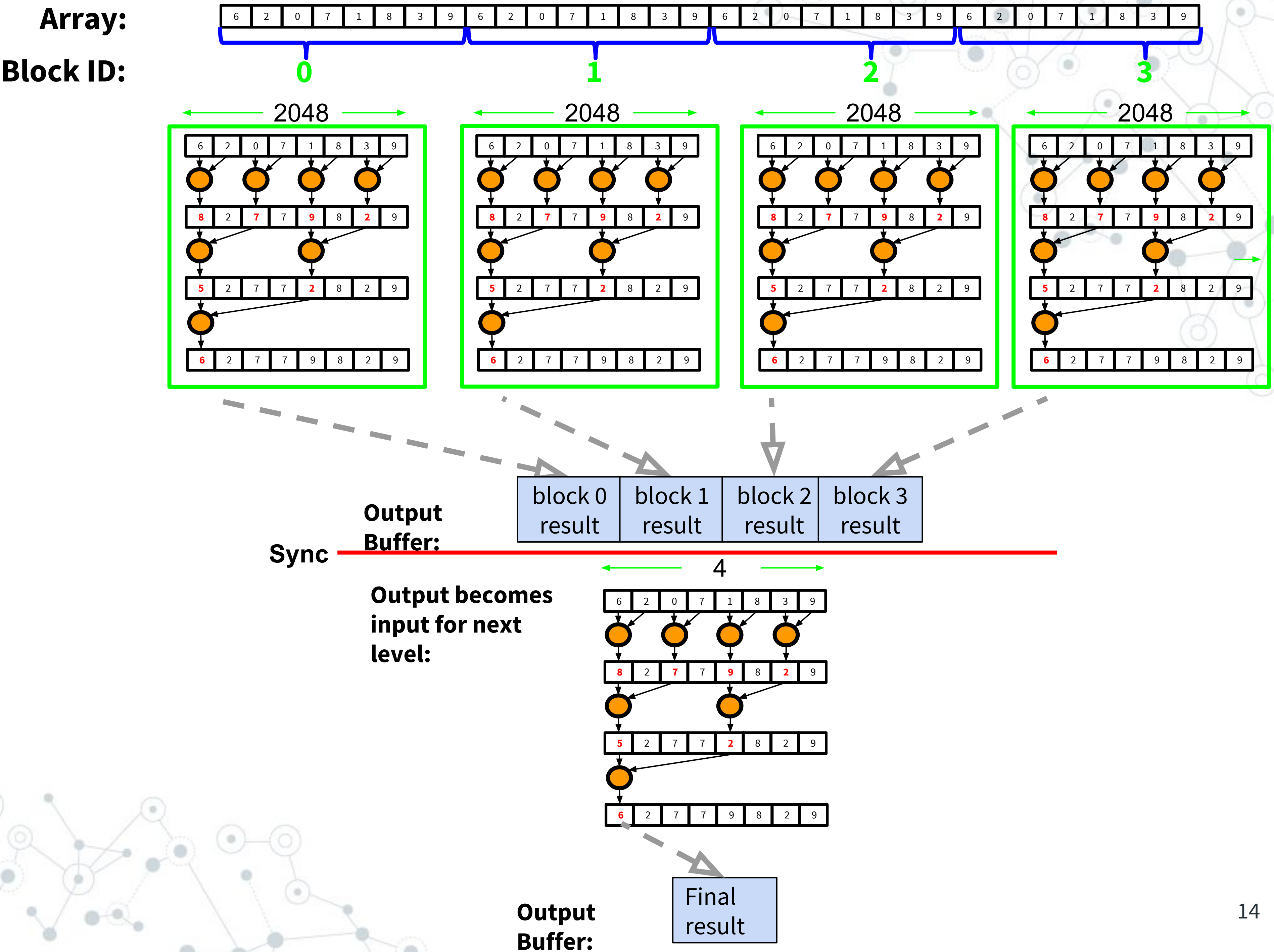
```
__global__ void reduce(float *array, int n) {
  int id = threadIdx.x;
  int threads;
  int stride;
  int left, right;
  threads = n / 2;
  for (stride = 1; stride < n; stride *= 2, threads /= 2) {
     if (id < threads) {
        left = id * (stride * 2);
        right = left + stride;
        array[left] = array[left] + array[right];
     }
     __syncthreads();
  }
}
```

Limitations to this approach

- Our kernel assumes we have enough threads to run level 0 in a single block
    - n / 2 threads

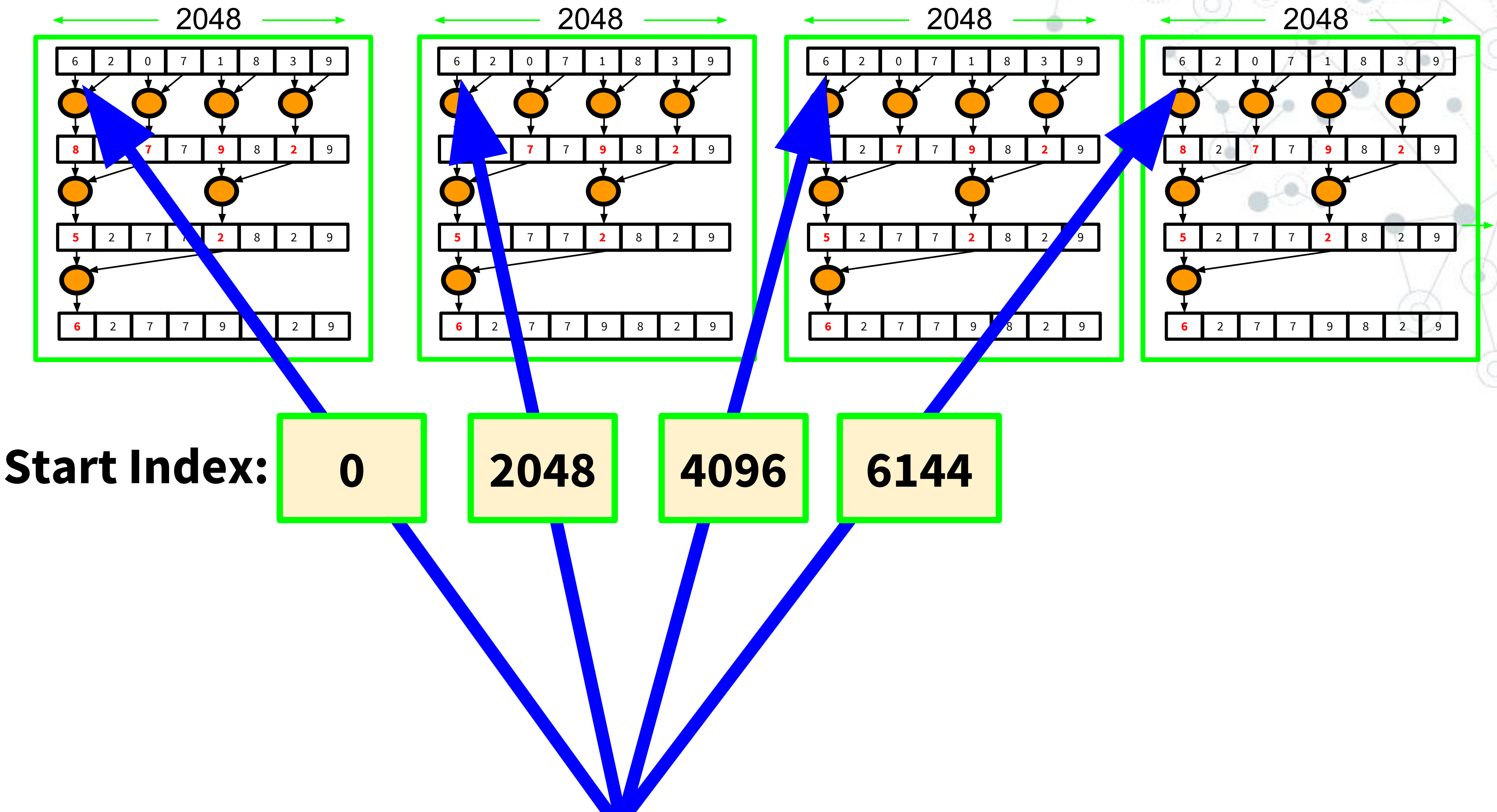
- **Problem:** Max block size on our GPU is 1024 threads
    - At level 0, each threads adds 2 elements
    - With 1024 threads, we can add a 2048 element array
    - any more, and we need to use another block

- Can we split the summation across multiple blocks?



Array:
Block ID:
0
1
2
3
2048
2048
2048
2048
block 0 result
block 1 result
block 2 result
block 3 result
Output Buffer:
Sync
4
Output becomes input for next level:
Final result
Output Buffer:

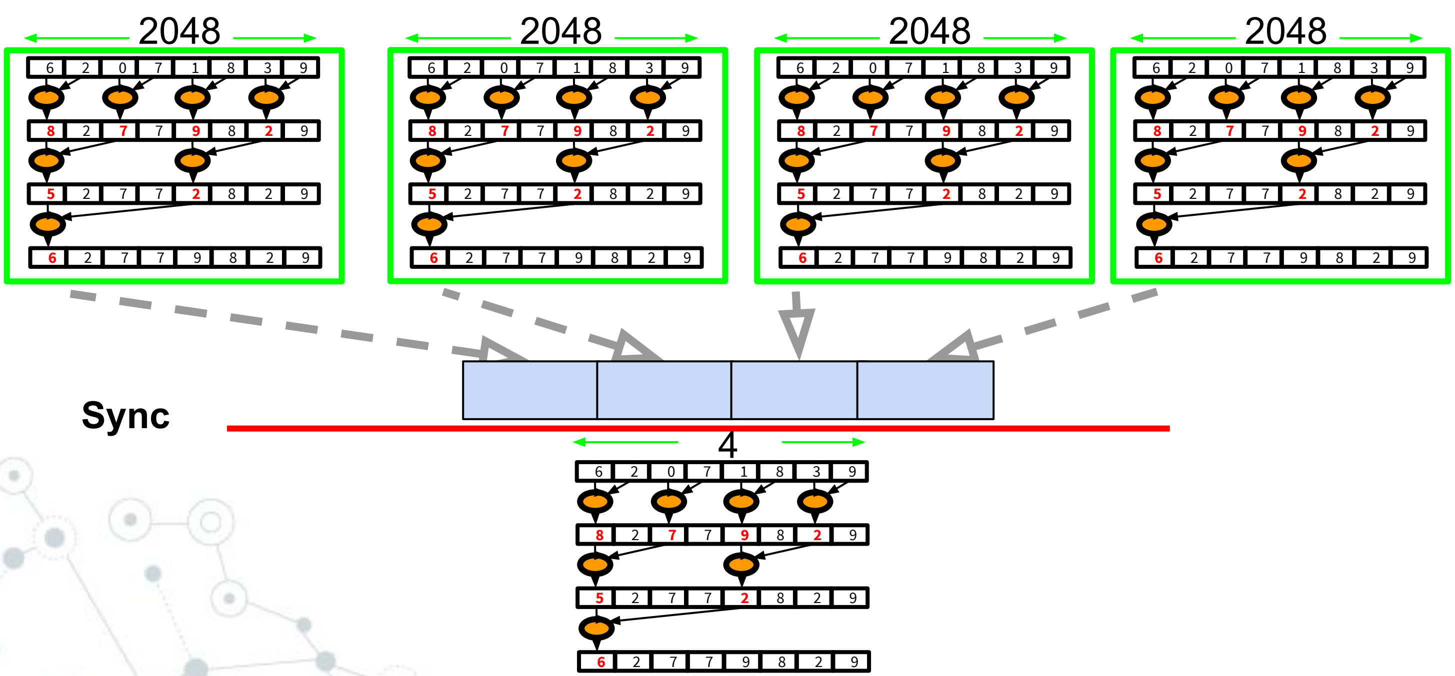
Kernel Changes




```
int block_start = block_id * 2048;
left = block_start + id * (stride * 2);
right = left + stride;
```

Synchronizing Blocks

- **Problem:** need to sync after each block-level
  - Can’t synchronize thread blocks in CUDA
  - Except by returning control to host…
- **Solution:** launch kernel multiple times
  - Use a loop on the host
  - One iteration for each block-level
  - Data stays in global memory between launches



Host code

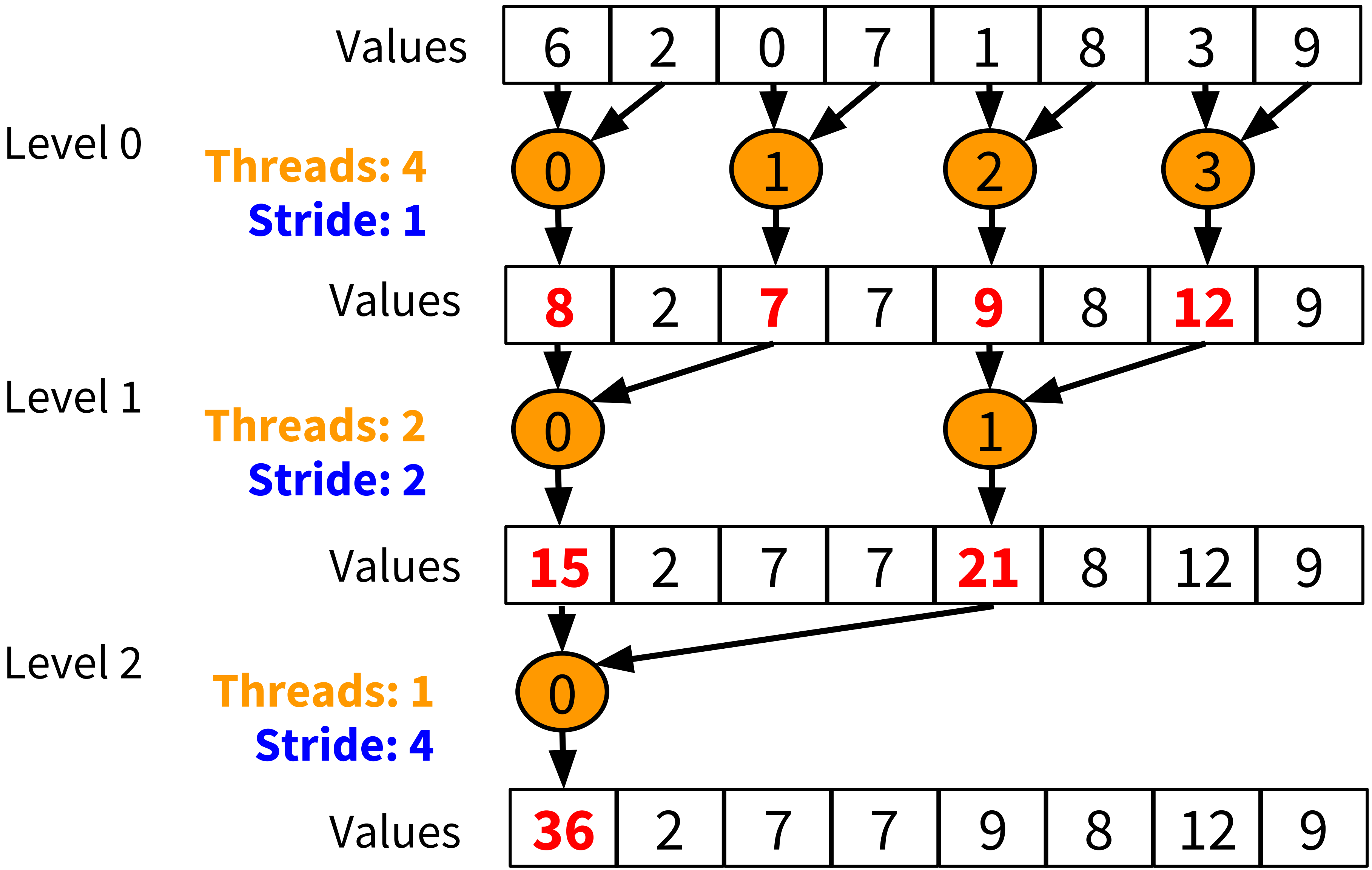
```
int threads = n / 2; // total (across all blocks)
int bsize = 1024; // block size
int blocks = threads / bsize + (threads % bsize > 0 ? 1 : 0);
int remaining = n; // total number of values left to add
while (remaining > 1) {
    // launch kernel
    reduce<<<blocks, bsize>>>(input_buf, output_buf, bsize);
    remaining = blocks;
    // if we'll do another iteration
    if (remaining > 1) {
        // recalculate num threads & blocks for next iteration
        threads = remaining / 2;
        blocks = threads / bsize + (threads % bsize > 0 ? 1 : 0);

        // output from last level becomes input for the next
        float *temp = input_buf;
        input_buf = output_buf;
        output_buf = temp;
    }
}
```

0. Initial Approach - Results

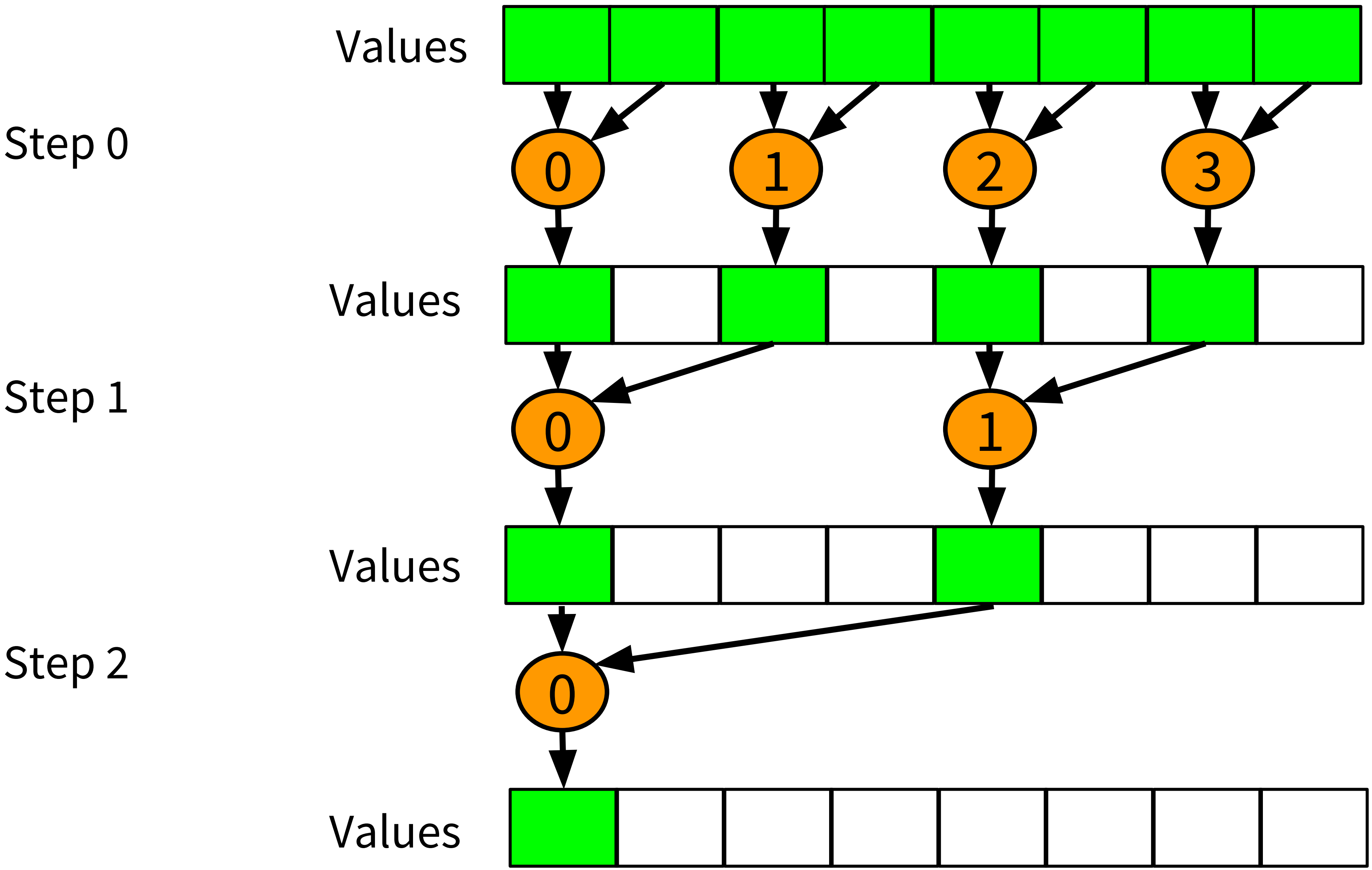
Approach	Throughput (MFLOPS)	Improvement
CPU	504	
0. Initial Approach	**1911**	**1407**

- Ok, but max device throughput is ~8228 **G**FLOPS! How can we improve?
  - We know memory access patterns matter
  - What do our kernel’s look like?

Access Pattern



Reads & Writes: Active Locations



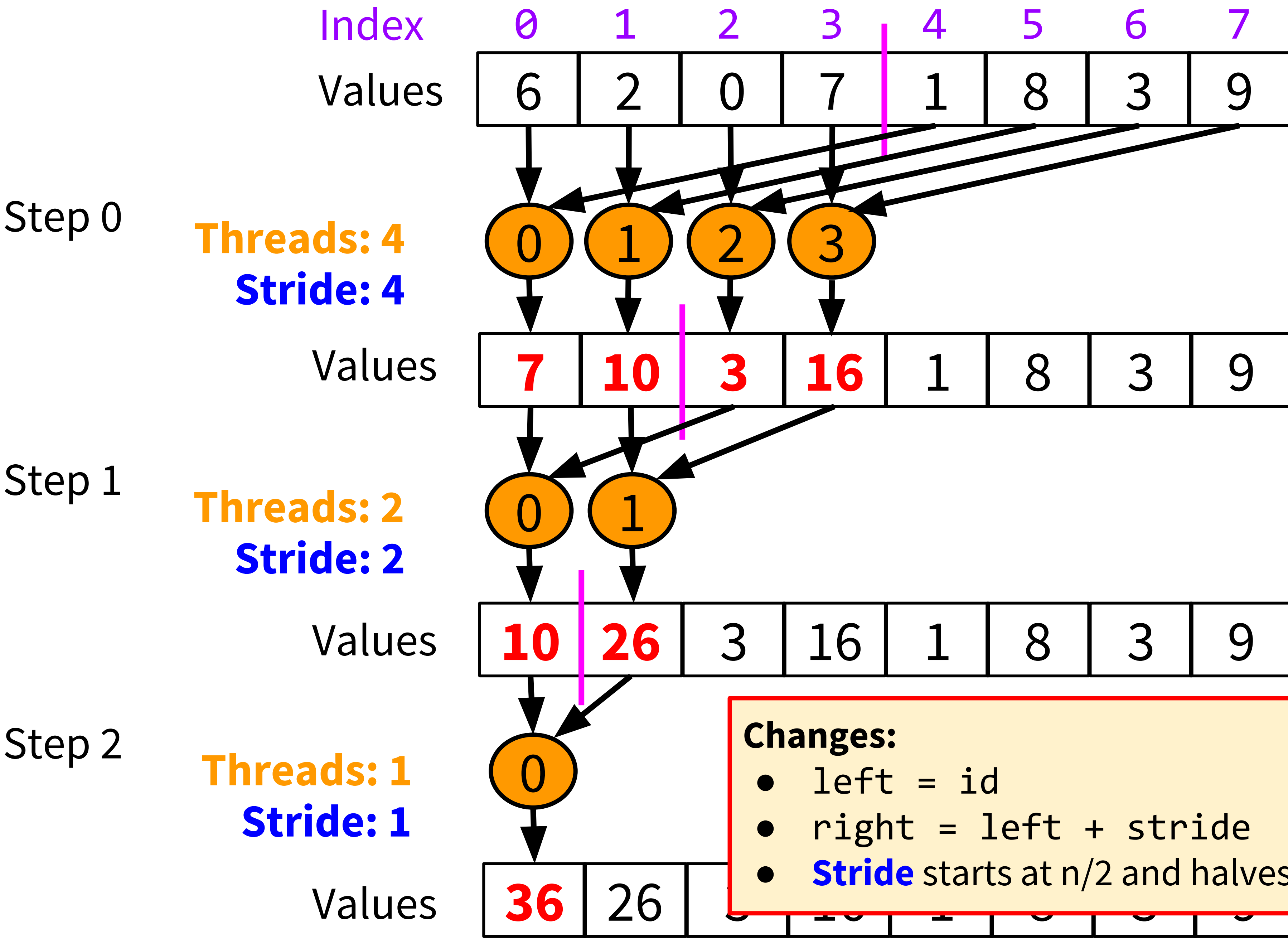
Wide gaps! Leads to poor global memory performance!

How can we fix it?

- Observation: addition is *commutative*
  - Order doesn't matter…
- We can choose which elements we add first
  - Can we eliminate the gaps?

$$x + y = y + x$$

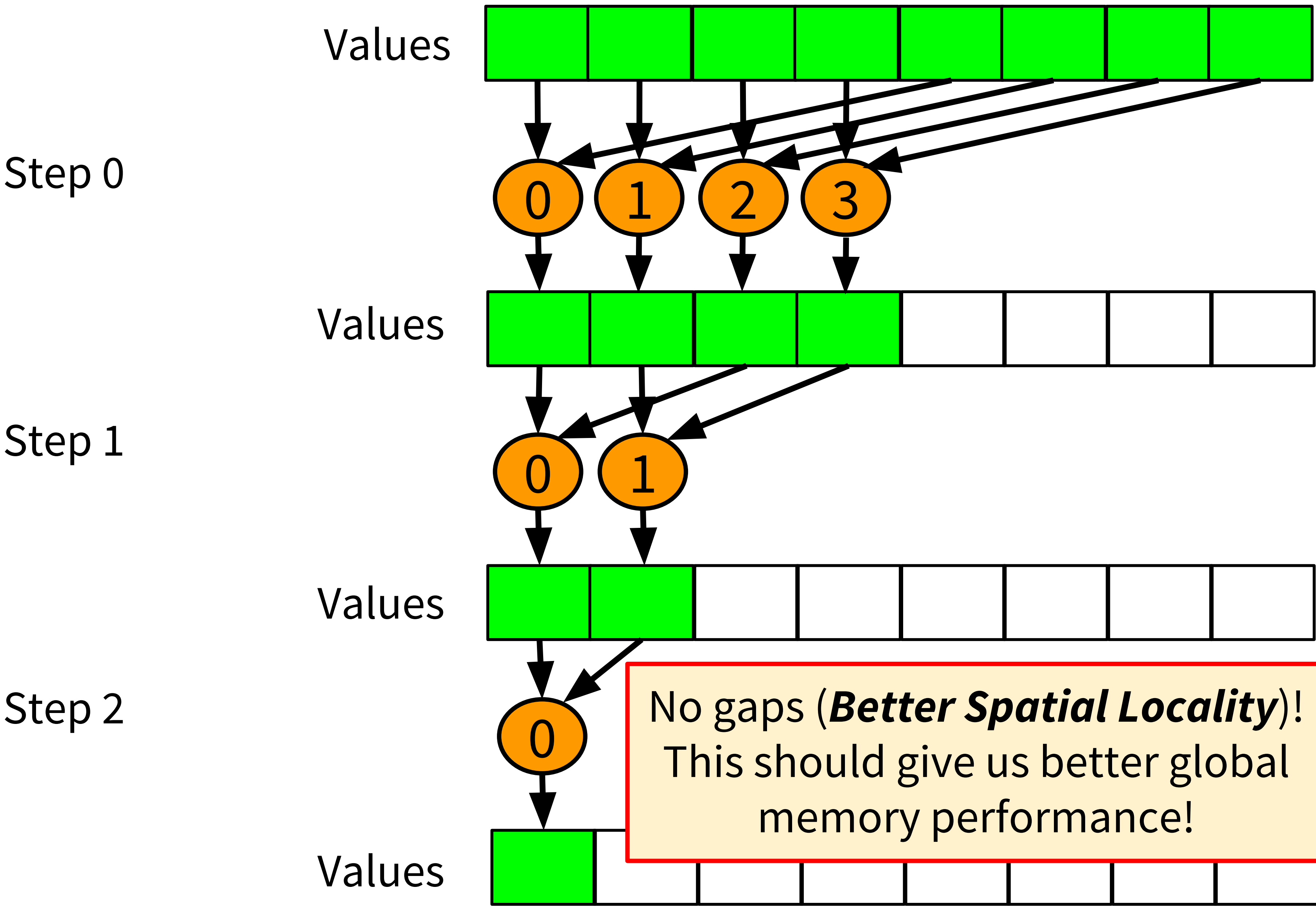
1. Global Memory Coalescing



Changes:

- `left = id`
- `right = left + stride`
- **Stride** starts at n/2 and halves

Reads & Writes: Active Locations



1. Global Memory Coalescing - Results

Approach	Throughput (MFLOPS)	Improvement
CPU	504	
0. Initial Approach	1911	1407
1. Global Memory Coalescing	**1978**	**67**

- 67 Million more adds/sec!
- Can we do more?
  - **Useful question:** ***How is our execution time being used?***