Resource partitioning and latency hiding

What is latency ?

number of clock cycles between instruction being issued and being completed

Arithmetic instruction latency
Memory operation latency

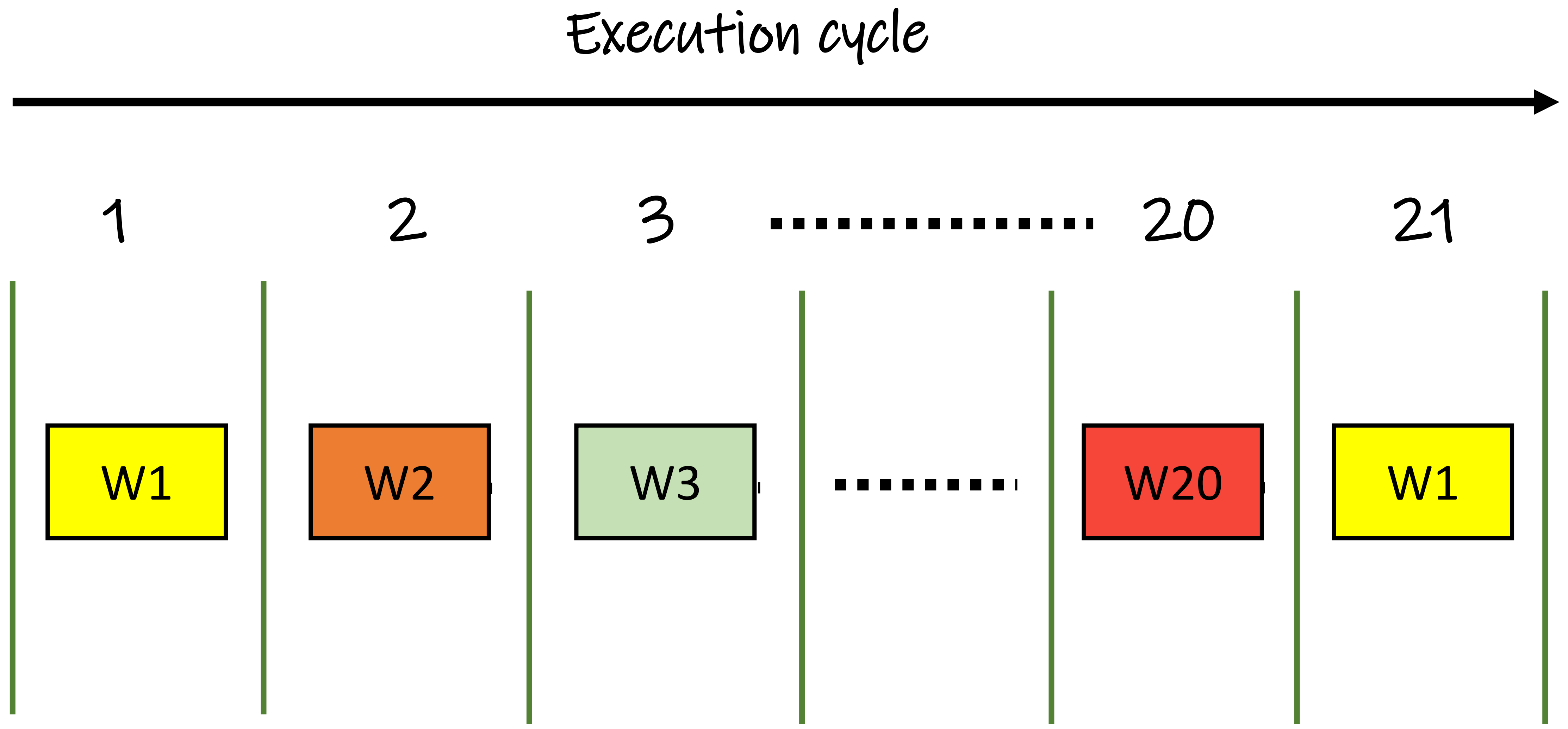
Operation	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
ADD/SUB	24	16	9	6
MAX/MIN	24	18	9	12
MAD	120	22	9	13
MUL	96	20	9	13
Div	608	286	141	210
Rem	728	280	138	202
AND, OR, XOR	24	16	9	6

Unit	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
Global & Local Memory				
L1 D$	×	45	30	×
L2 D$	×	310	175	194
DRAM	440	685	300	350
Shared Memory				
SMEM	38	50	33	28
Texture Memory				
Texture L1 D$	261	224	105	92
L2 D$	371	435	215	172
DRAM	×	791	348	330
Fixed-function pipeline	×	106	48	(-20)
Constant Memory				
Constant L1 D$	56	52	42	28
Constant L1.5 D$	129	165	104	79
L2 D$	268	375	215	184

Latency hiding

- The execution context of each warp processed by and SM are maintained on-chip during the entire lifetime of the warp.

- Therefore switching from one execution context to another has no cost.


Execution cycle
1
2
3
20
21
W1
W2
W3
W20
W1

1 SM -> 128 cores

can execute 4 warps parallelly in one SM

4 x 20= 80

To hide the latency of per SM

13 * 80 = 1040

To hide the latency of per device

How about memory latency

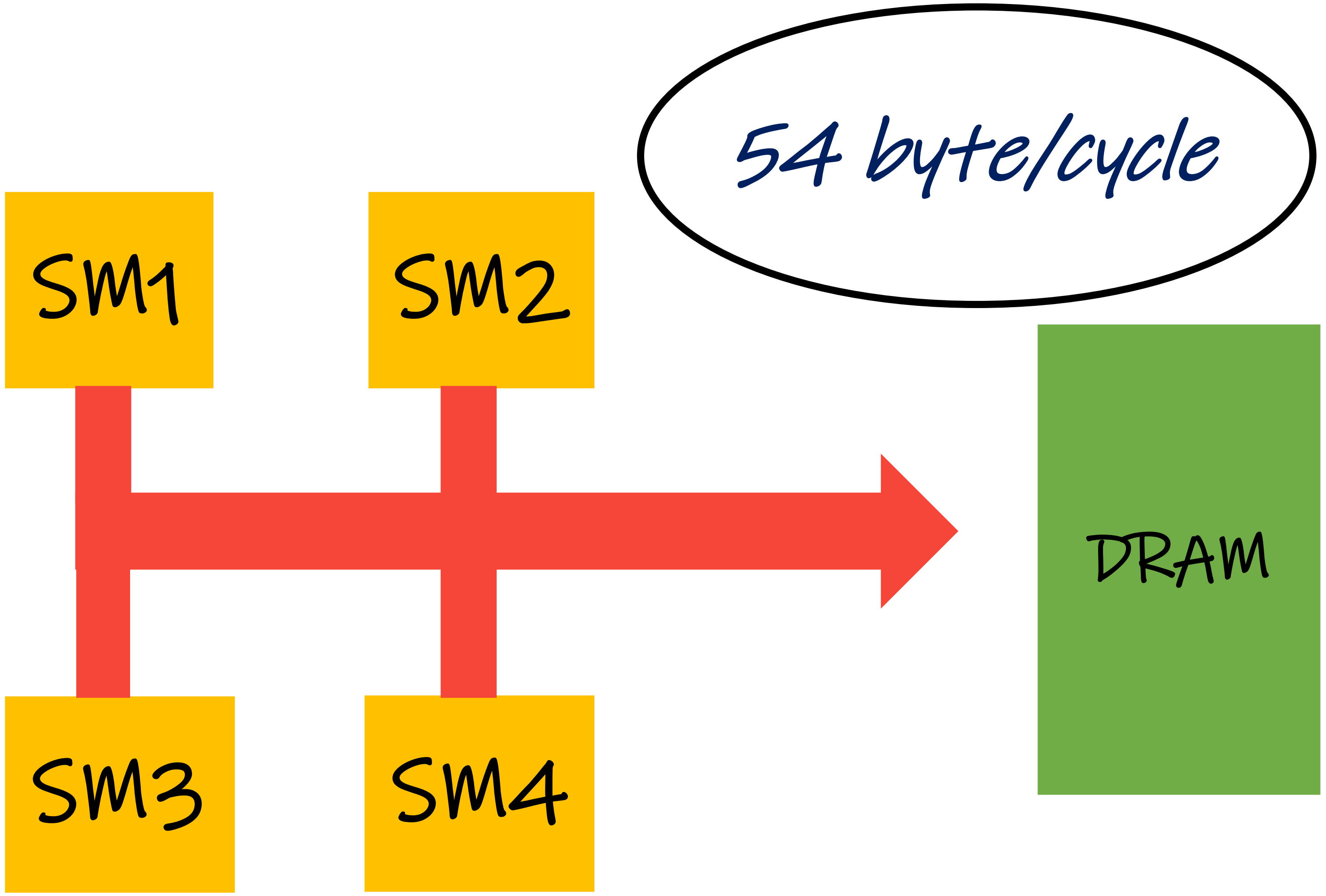
- Lets consider DRAM latency of Maxwell architecture as 350 cycles .

GTX 970 have bandwidth of 196 GB/s

Nvidia-smi -a -q -d CLOCK

3.6 GHz memory clock

196 / 3.6 = 54 Bytes/cycle


54 byte/cycle
SM1
SM2
SM3
SM4
DRAM

54 * 350 = 18900 Bytes

18900 / 4 = 4725 threads

4725/ 32 = 148 warps

148 / 13 -> 12 warps per SM

Categorizing CUDA applications

- Bandwidth bound applications
- Computation bound applications

Occupancy

Occupancy is the ratio of active warps to maximum number of warps, per SM.

$$\text{Occupancy} = \frac{\text{Active warps}}{\text{maximum warps}}$$

Depends on device and resource usage of kernel

Documentation or query the device

Our kernel use 48 registers per thread and 4096 bytes of Smem per block. And block size is 128

Reg per warp = 48 * 32 = 1536

For GTX 970 device – 65536 regs per SM

Allowed warps = 65536 / 1536 = 42.67

-> 40

For GTX 970 device – 98304 regs per SM

Avtive blocks= 98304 / 4096= 24

acvtive warp = 24 * 4 = 96

Active warp count does not limit by smem usage

GTX 970 -> max warps per SM is 64

$$\text{Occupancy} = \frac{\text{Active warps}}{\text{maximum warps}}$$

$$\frac{40}{64} = 63\ \%$$

Occupancy calculator

The CUDA Toolkit includes a spreadsheet, called the CUDA Occupancy Calculator, which assists you in selecting grid and block dimensions to maximize occupancy for a kernel.

If a kernel is not bandwidth-bound or computation-bound, then increasing occupancy will not necessarily increase performance. In fact, making changes just to increase occupancy can have other effects, such as additional instructions, more register spills to local memory which is an off-chip memory, more divergent branches.

Guide line for grid and block size

- Keep the number of threads per block a multiple of warp size (32).

- Avoid small block sizes: Start with at least 128 or 256 threads per block.

- Adjust block size up or down according to kernel resource requirements.

- Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism to your device.

- Conduct experiments to discover the best execution configuration and resource usage.

Profiling with nvprof

Nvprof

- The nvprof profiling tool enables you to collect and view profiling data from the command-line.
  - kernel executions.
  - memory transfers.
  - CUDA API calls
  - events or metrics for CUDA kernels.

Nvprof Profile modes

-Summary mode

-GPU and API trace mode

-Event metrics summery mode

-Event, metrics trace mode

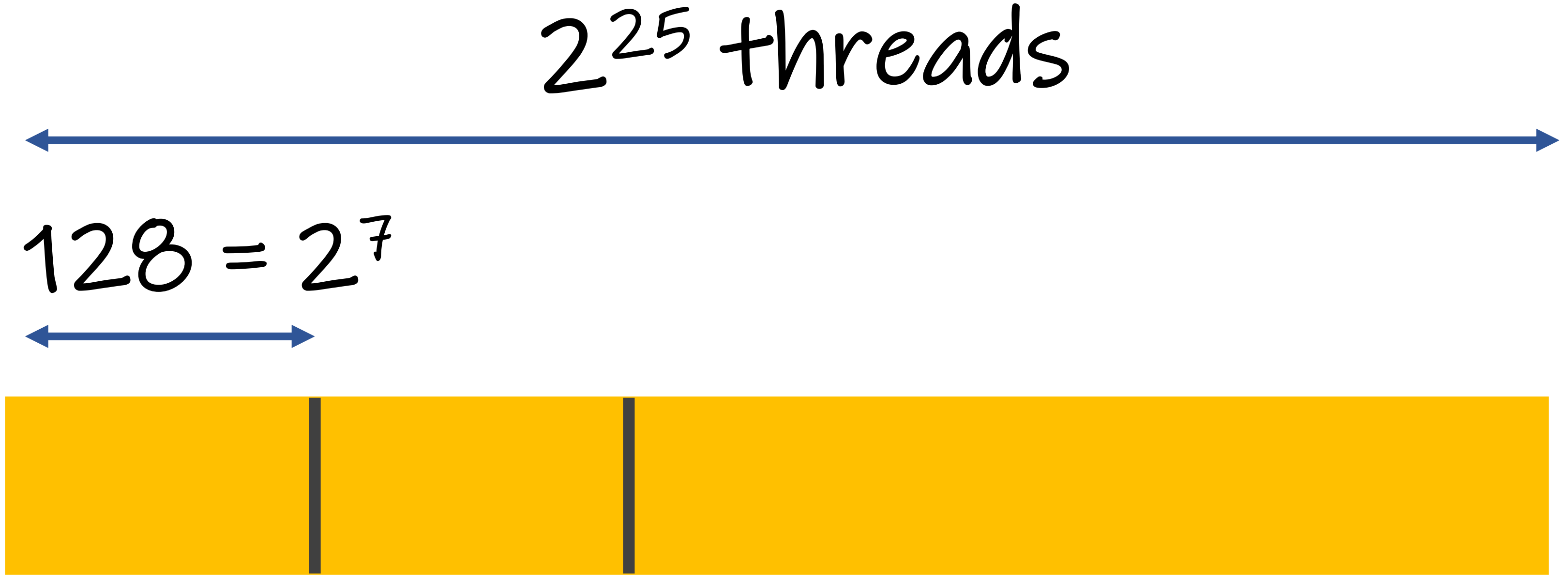
- nvprof [options]
  [application]
  [application-arguments]

- --events
- --metrics

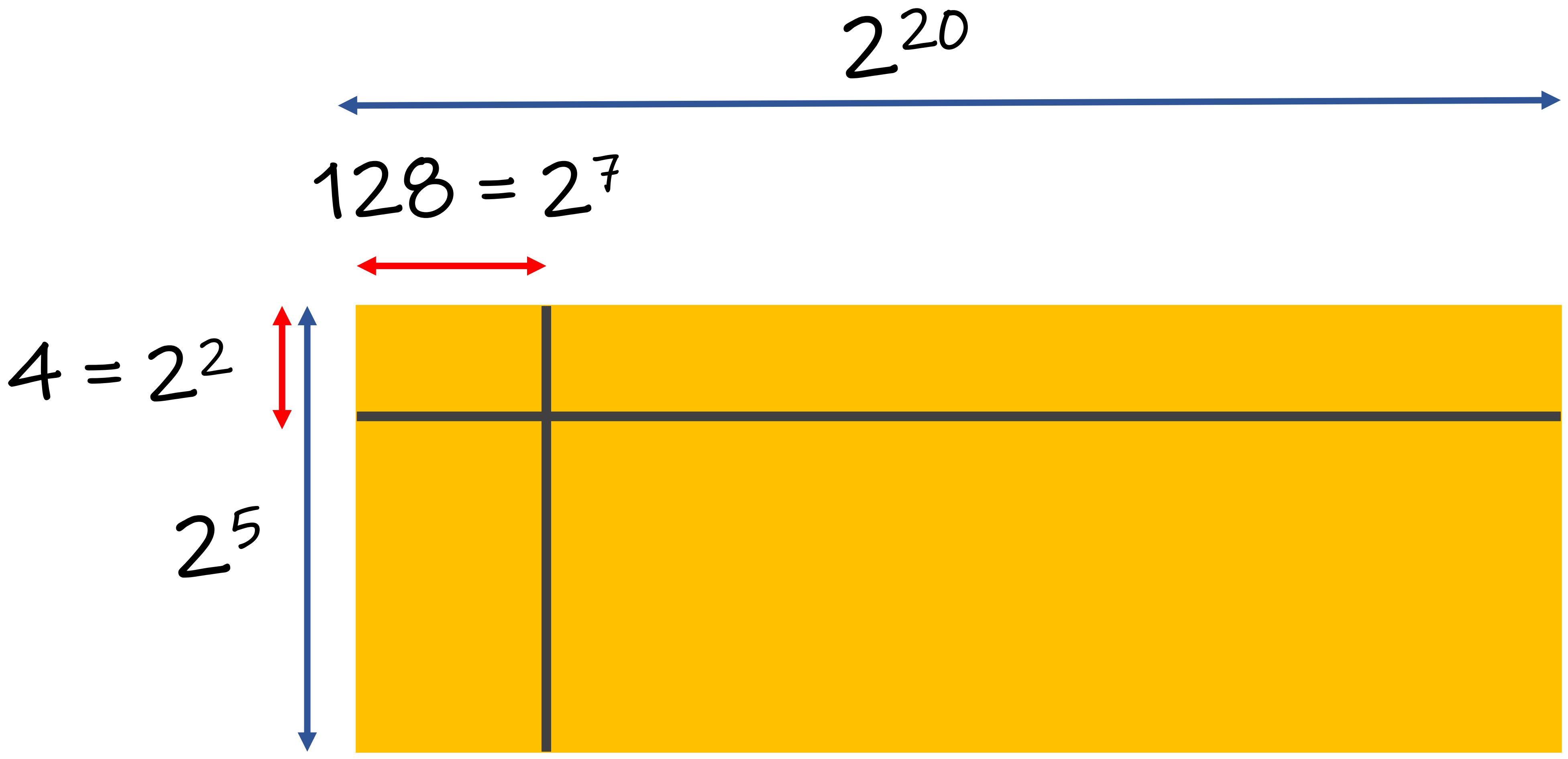
Some metrics

- sm_efficiency
- Achieved_occupancy
- Branch_efficiency
- Gld_efficiency
- Gld_throuput
- Dram_read_throughput
- Inst_per_warp
- Stall_sync

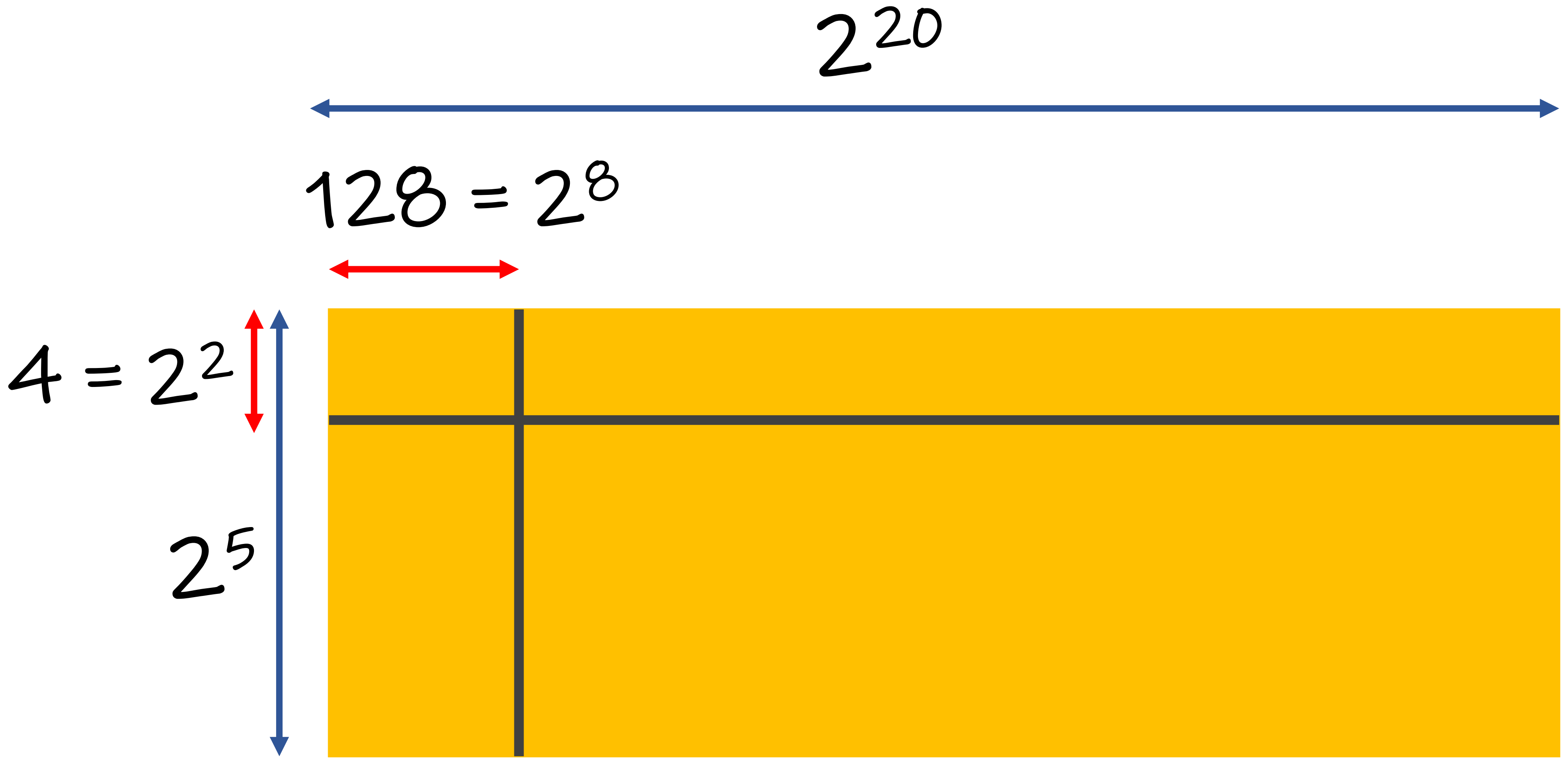
$32 \text{ Mb} = 2^{25}$ Bytes



Arguments : 0 25 0 7



Arguments : 1 25 20 7 2



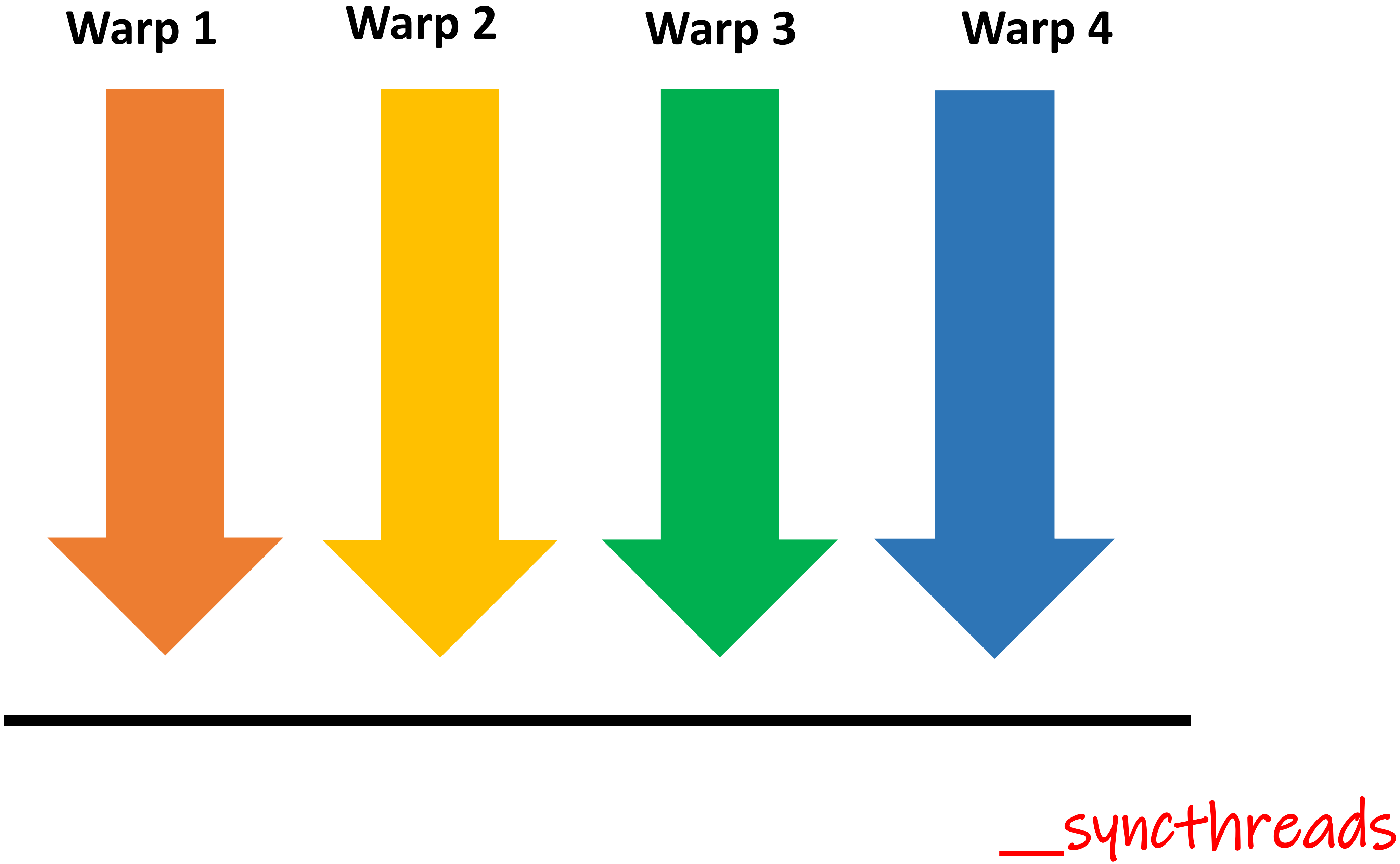
Arguments : 1 25 20 8 2

Synchronization

- cudaDeviceSynchronize
- __syncthreads

Introduce a
global synchronize
point in host code

Synchronization with
in a block


Warp 1
Warp 2
Warp 3
Warp 4
__syncthreads

Parallel reduction

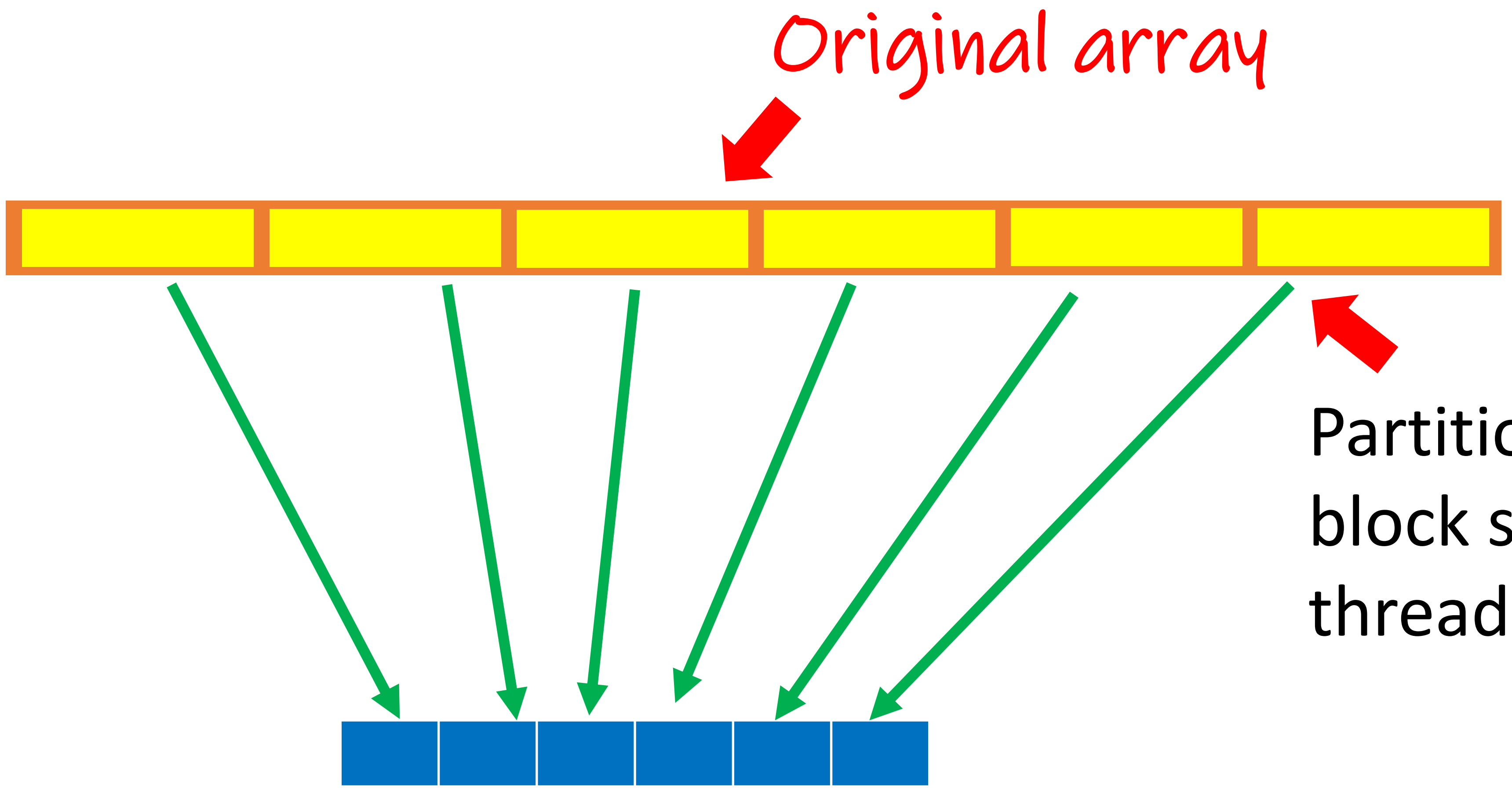
General problem of performing commutative and associative operation across vector is known as the reduction problem

Sequential reduction

```
int sum =0 ;
For (int I =0; I < size ; I ++)
{
        sum + = array[i];
}
```

Our approach

- Partition the input vector in to smaller chunks.
- And each chunk will be summed up separately.
- add these partial results from each chunk in to a final sum

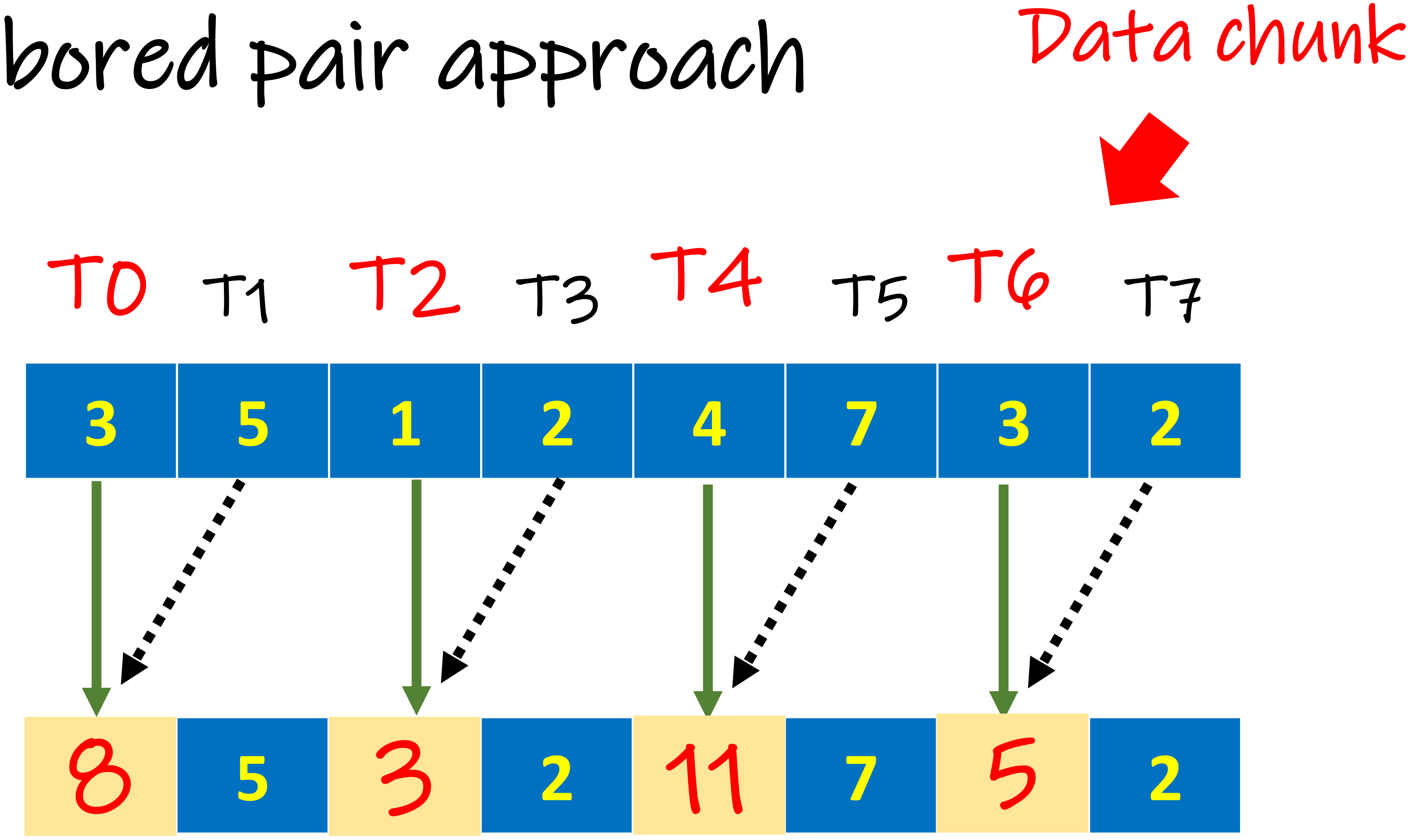


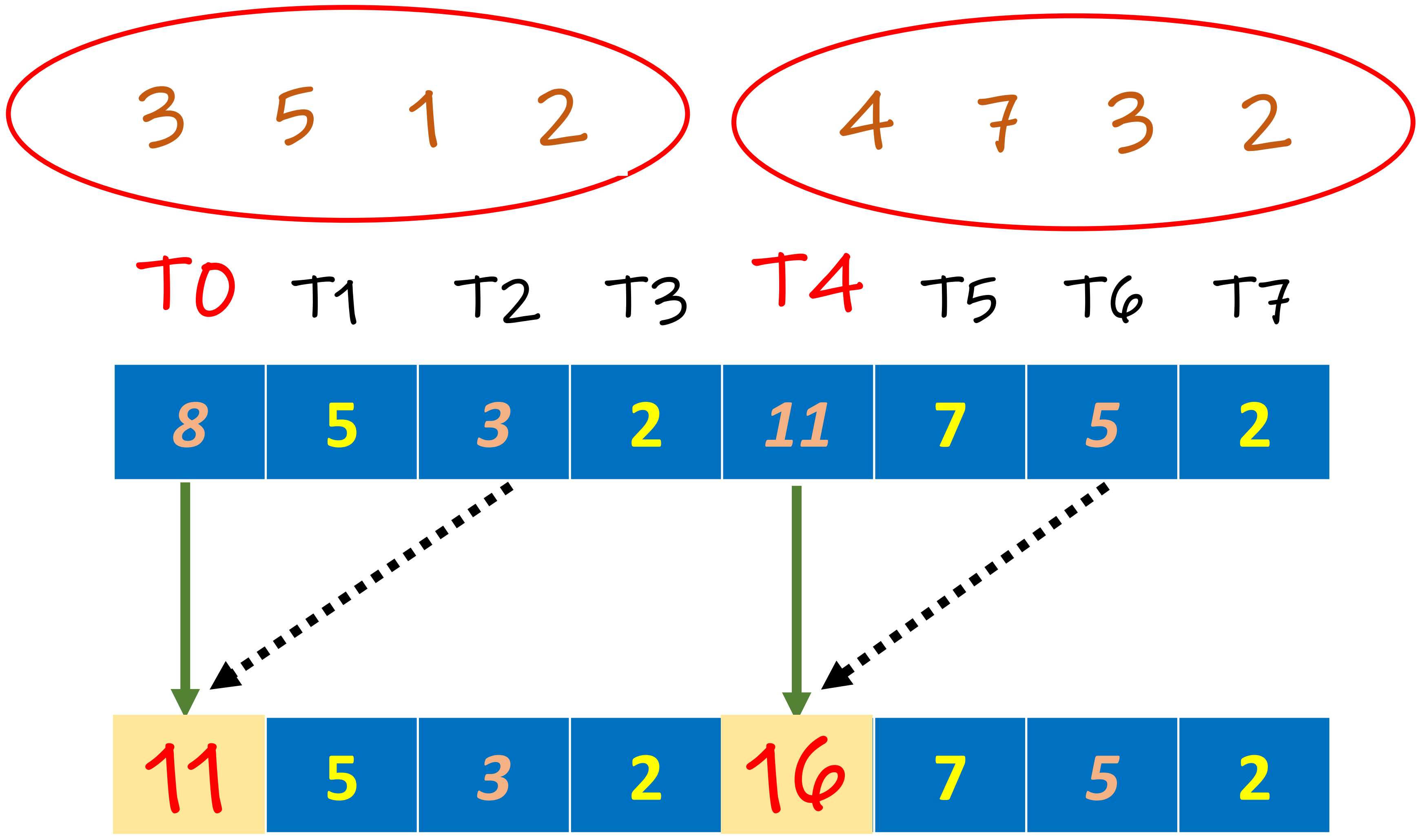
Sum of the each block is going to store in to a partial sum array

Neighbored pair approach

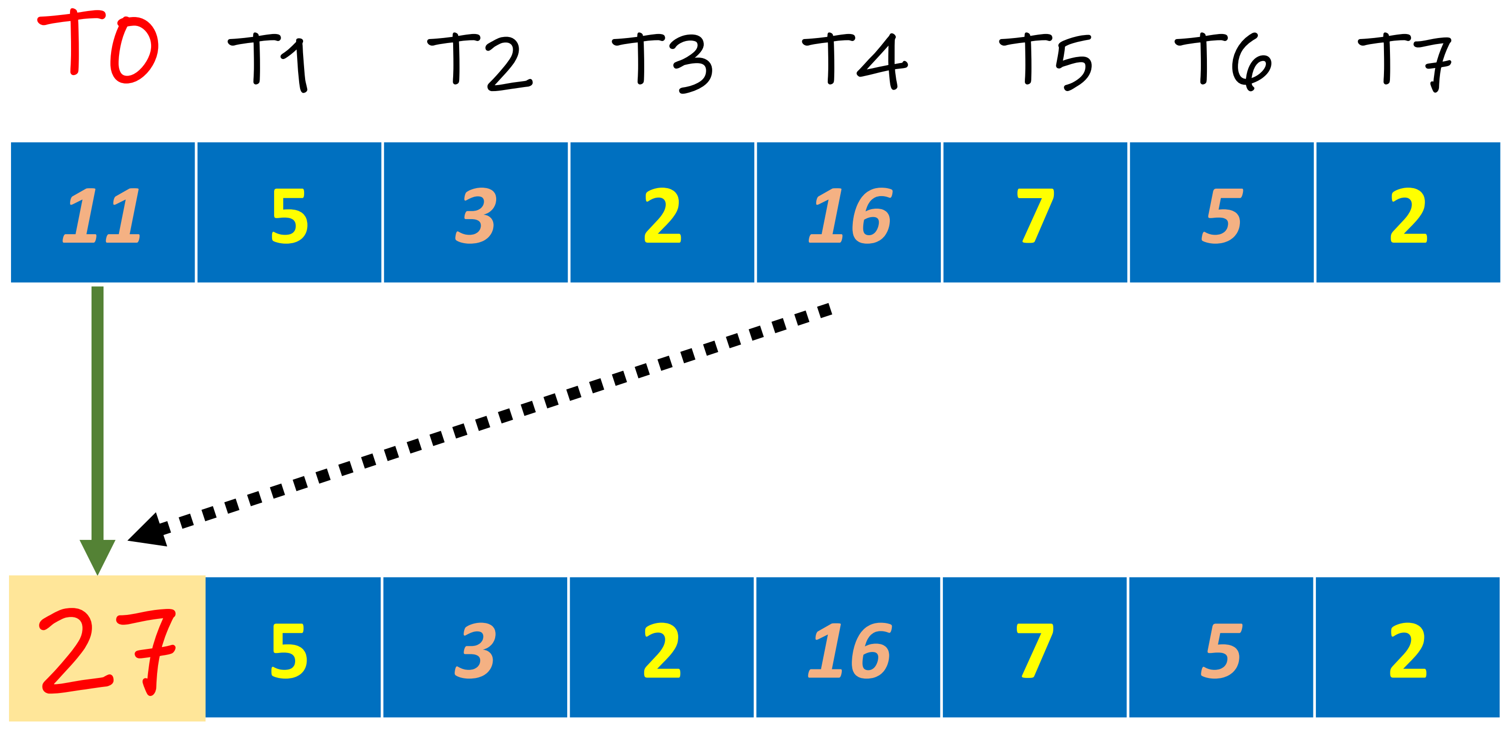
- we are going to calculate sum of the block in iterative manner and in each iteration selected elements are paired with their neighbor from given offset
- For the first iteration we are going to set 1 as the offset and in each iteration, this offset will be multiplied by two
- And number of threads which are going to do any effective work will be divide by this offset value.

Neighbored pair approach





3 5 1 2
4 7 3 2
T0
T1
T2
T3
T4
T5
T6
T7
8
5
3
2
11
7
5
2
11
5
3
2
16
7
5
2


T0
T1
T2
T3
T4
T5
T6
T7
11
5
3
2
16
7
5
2
27
5
3
2
16
7
5
2

Code-segment

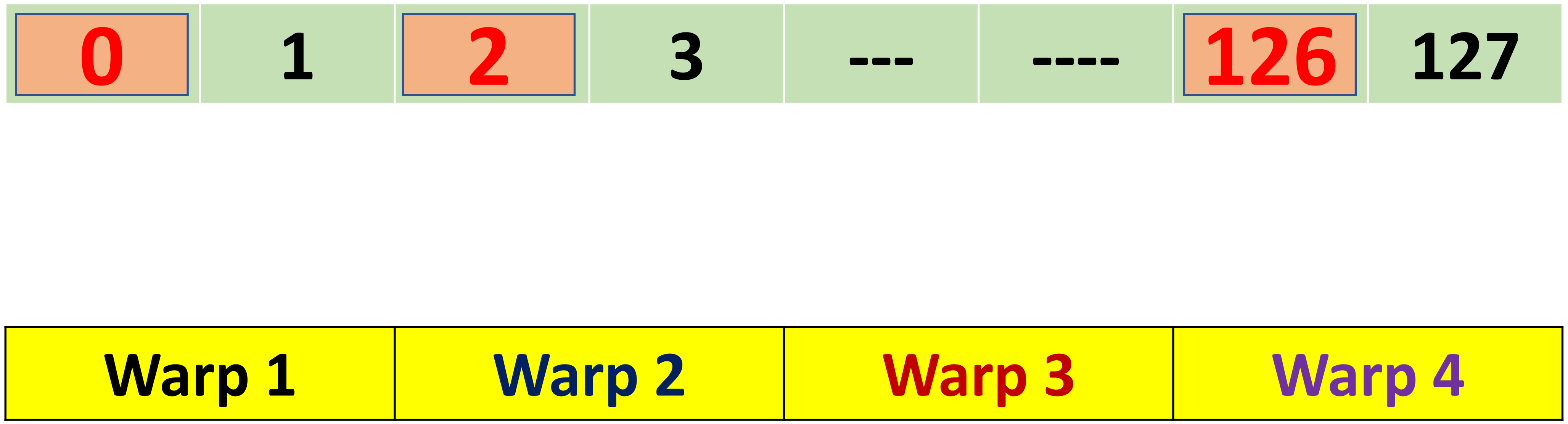
```
For(int offset =1 ; offset < blockdim.x; offset *=2)
{
        if(  tid % (2* offset)==0  )
        {
                input[ tid ] += input[ tid + offset ];
        }
        __syncthreads()
}
```

Careful.......... Careful..........

- Be mind full when using __syncthreads() function call inside the condition check.

Paradox

Divergence in reduction algorithm


0
1
2
3

126
127
Warp 1
Warp 2
Warp 3
Warp 4

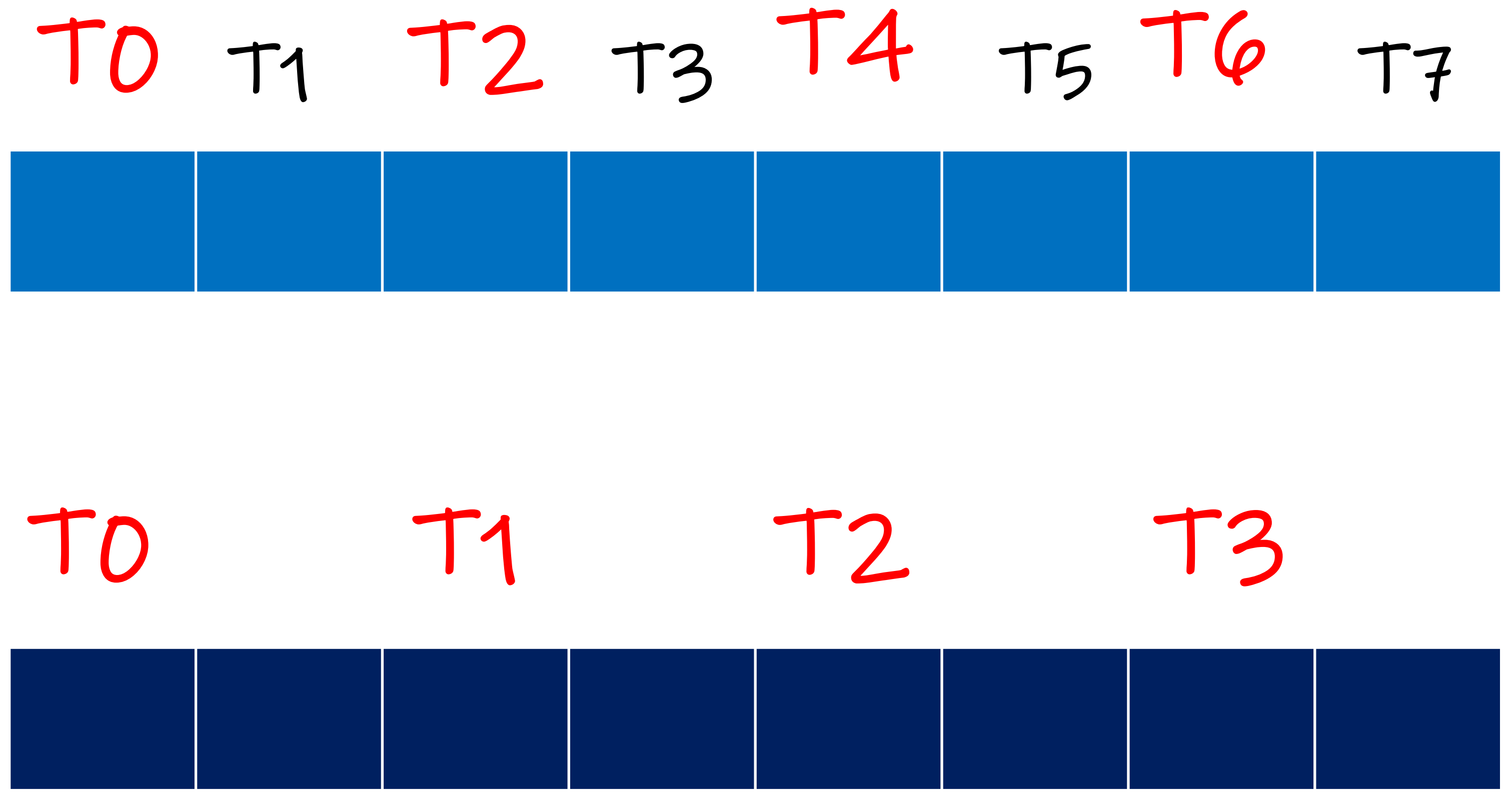
0	1	2	3	4	----	126	127

Warp 1	Warp 2	Warp 3	Warp 4

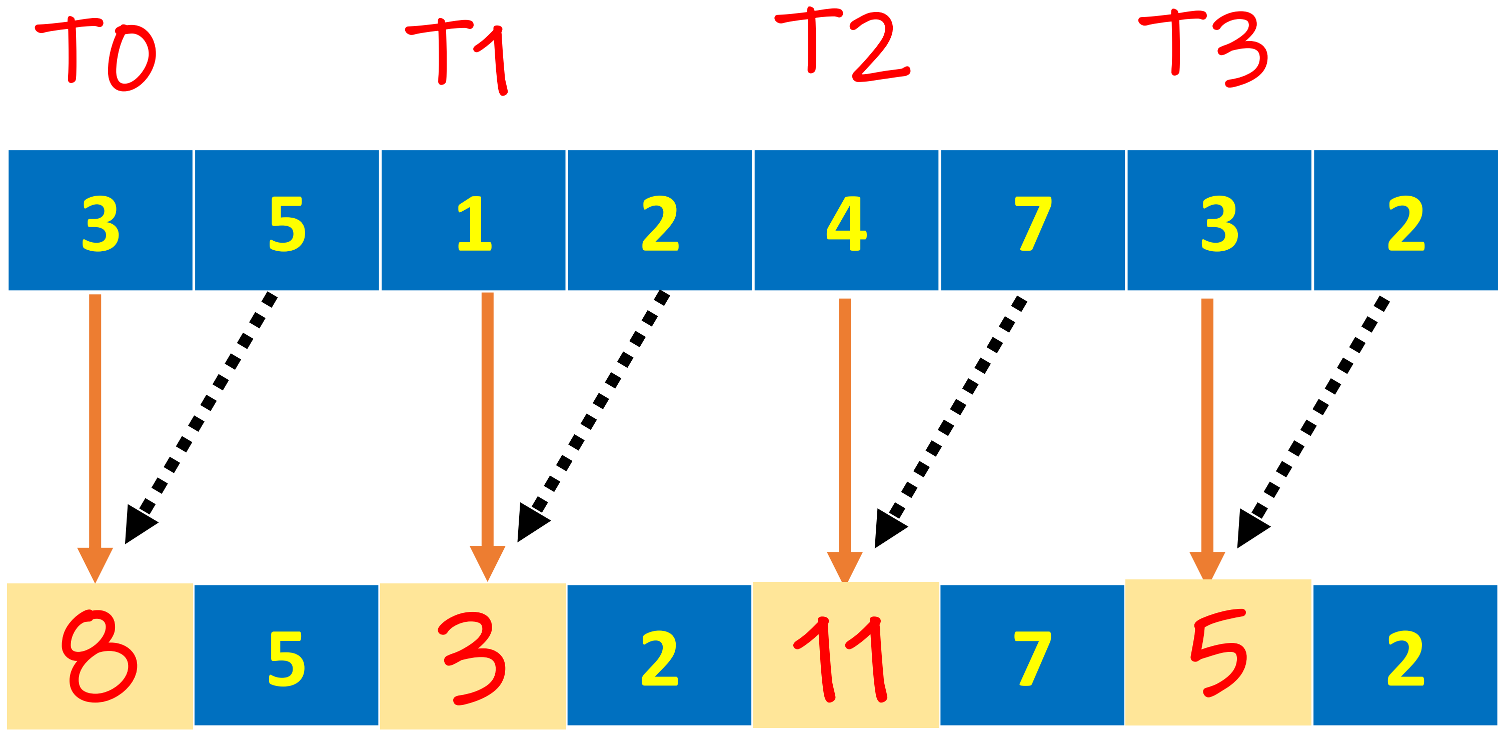
1. Force neighboring threads to perform summation

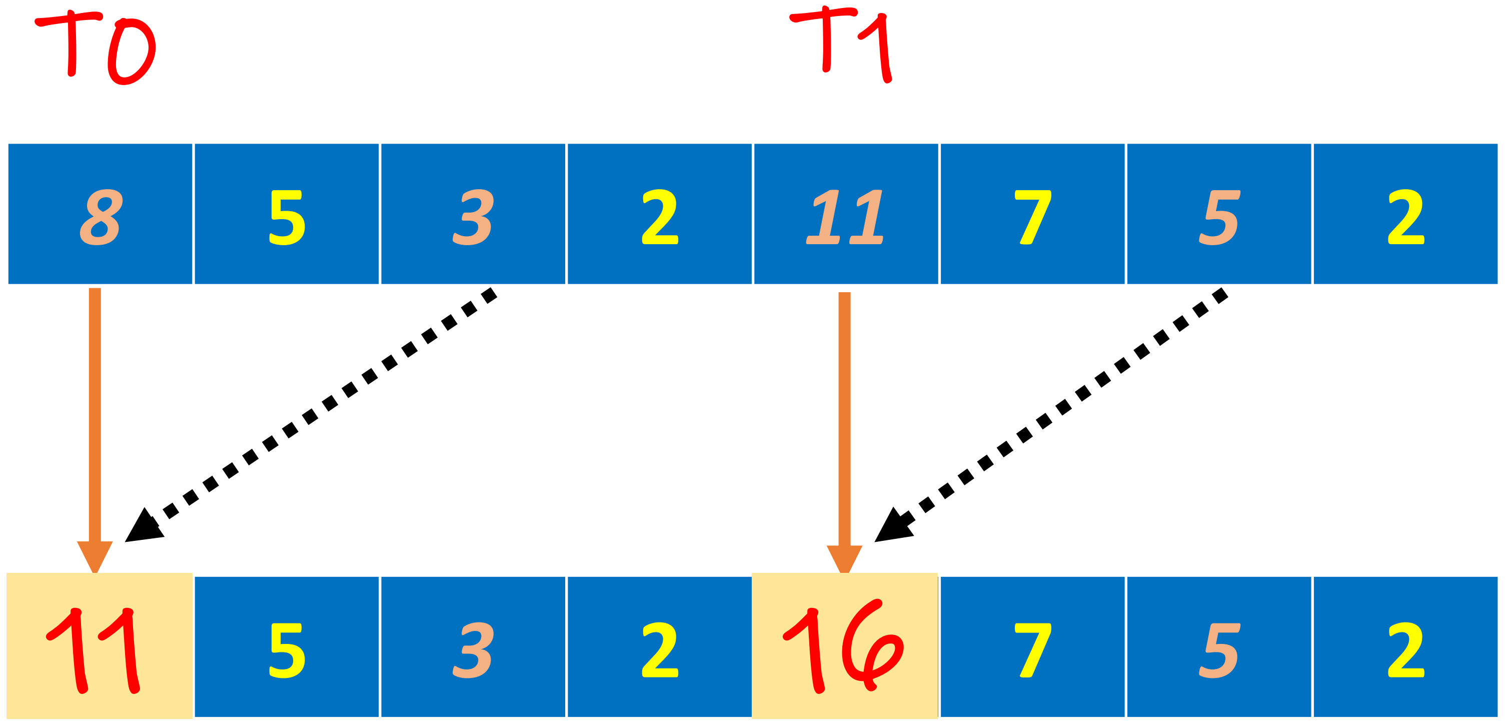
2. Interleaved pairs

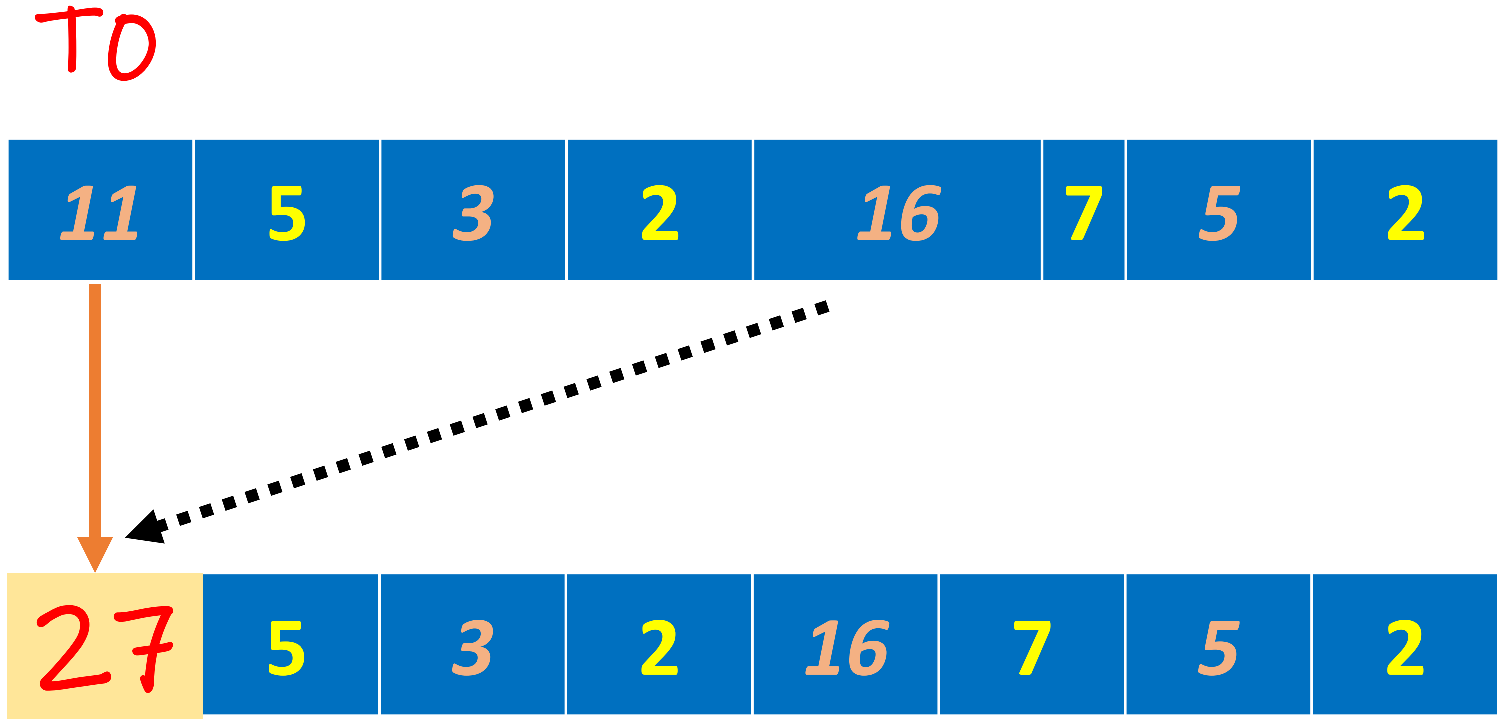
Rearranging thread index

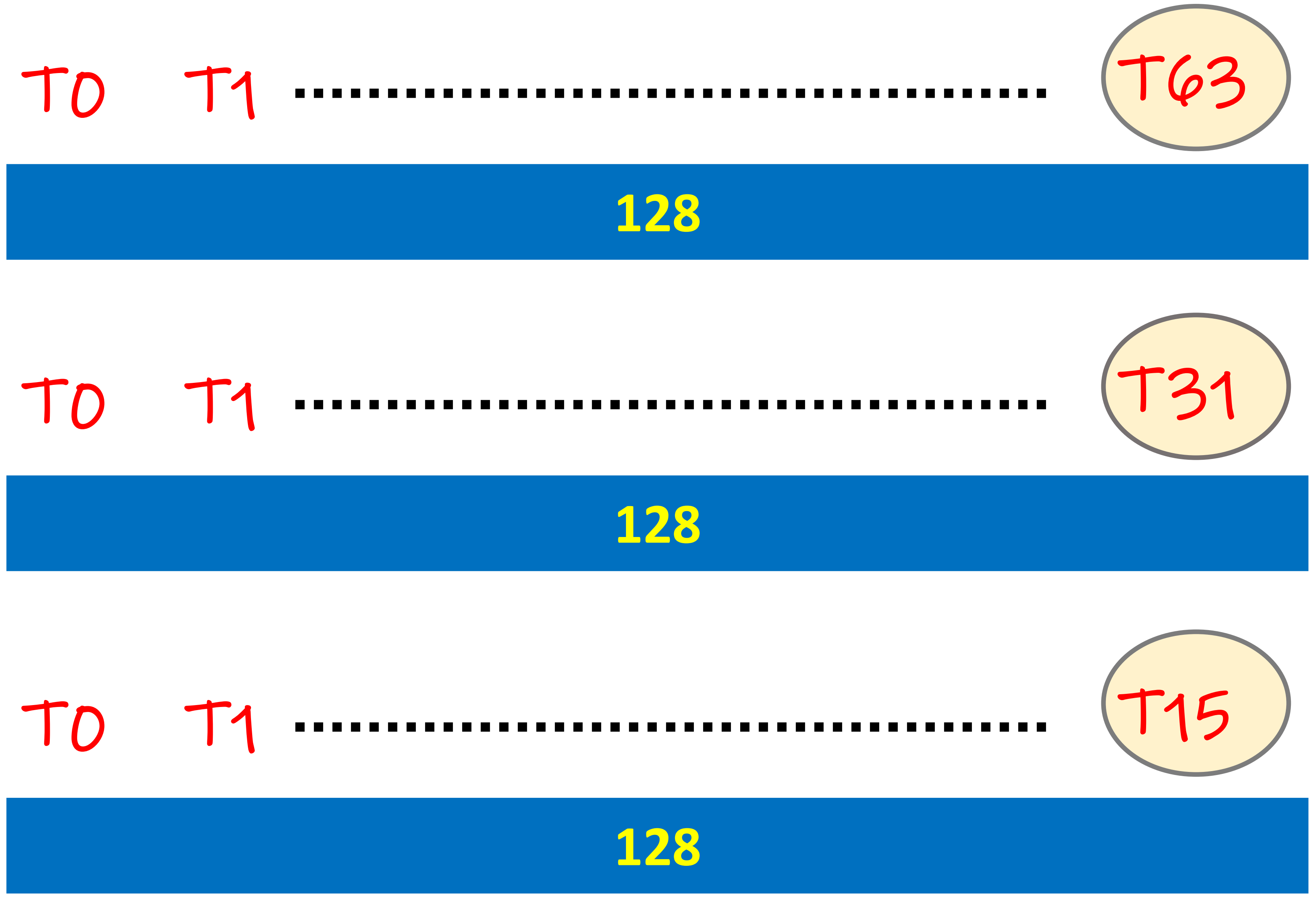


New approach




T0
T1
8
5
3
2
11
7
5
2
11
5
3
2
16
7
5
2


T0
11
5
3
2
16
7
5
2
27
5
3
2
16
7
5
2

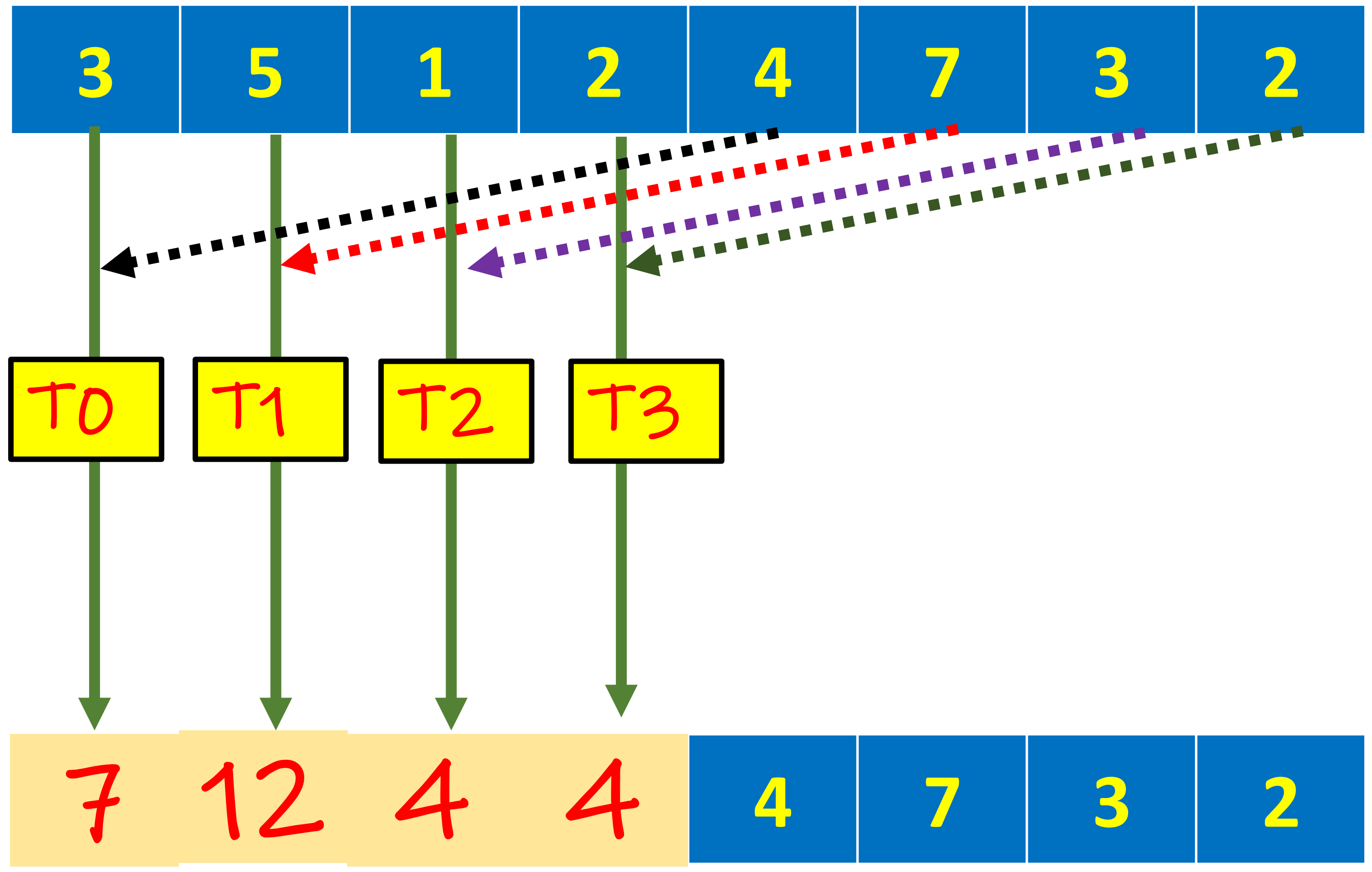

T0 T1 .... T63
128
T0 T1 .... T31
128
T0 T1 .... T15
128

code

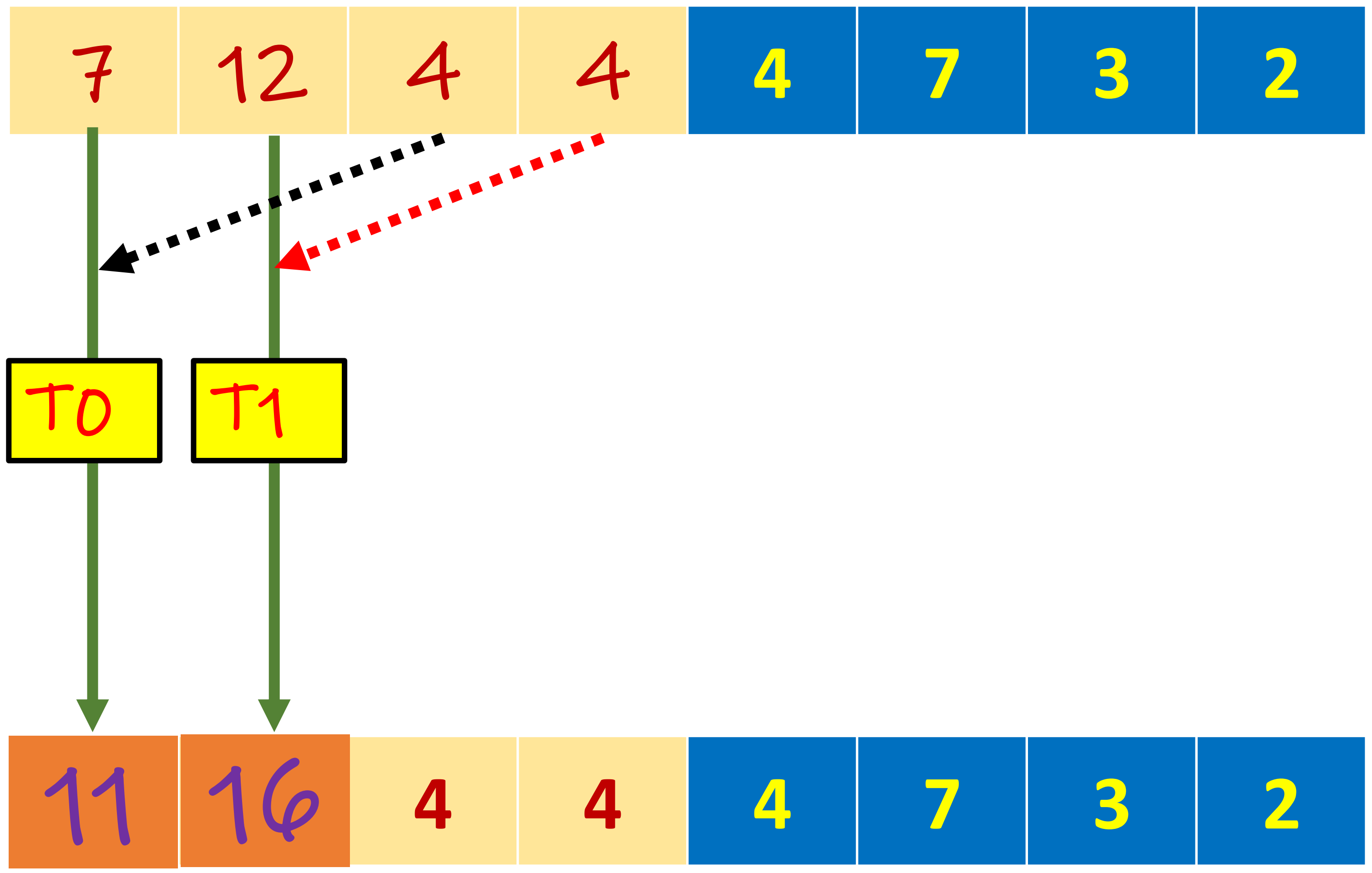
```
int * i_data = int_array +  blockDim.x * blockIdx.x;

for (int offset = 1; offset < blockDim.x; offset *= 2)
{
        int index = 2 * offset * tid;

        if (index < blockDim.x)
        {
                i_data[index] += i_data[index + offset];
        }
        __syncthreads();
}
```

Interleaved pairs approach

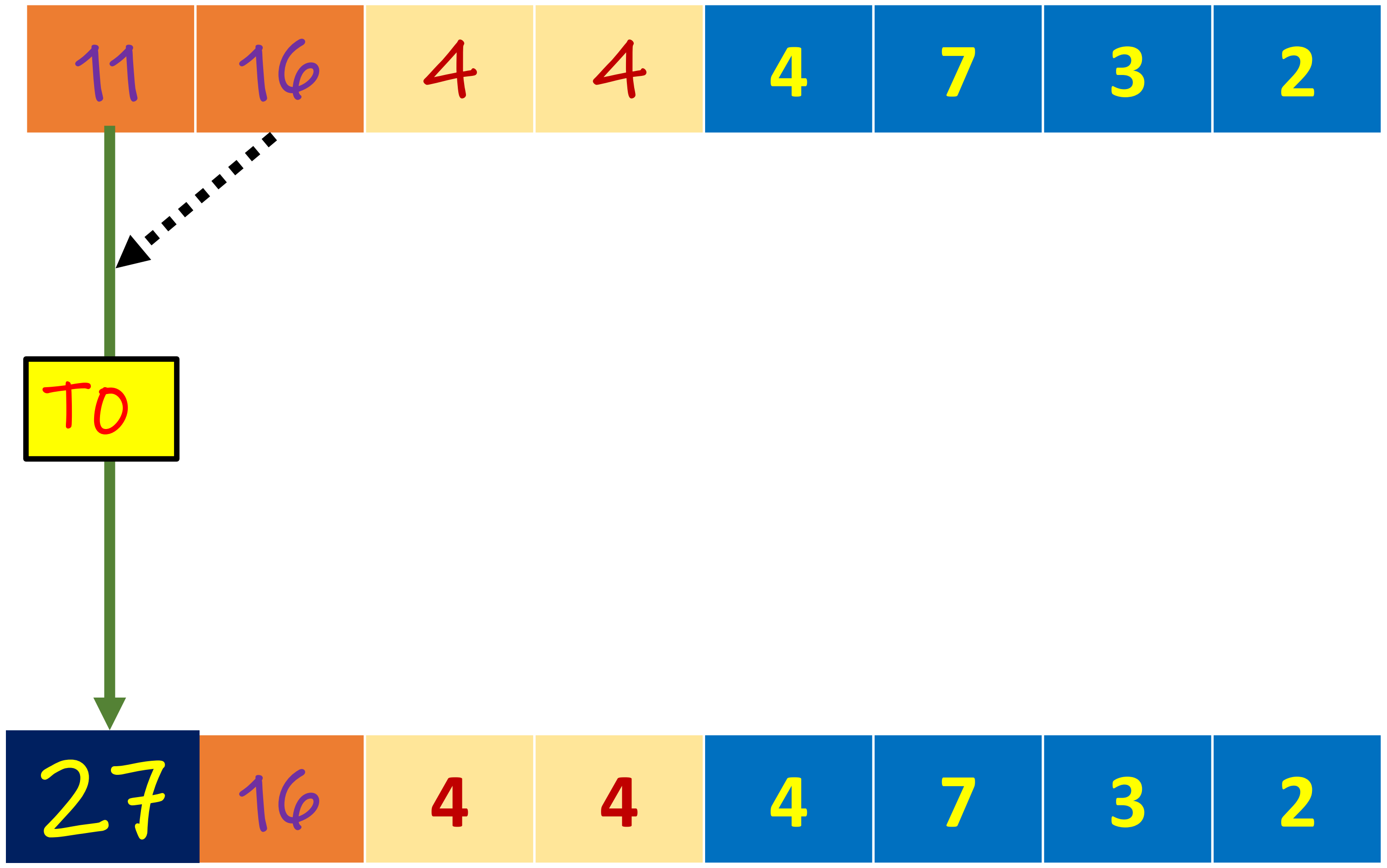
Interleaved pairs approach



Interleaved pairs approach



Interleaved pairs approach



code

```
int * i_data = int_array +  blockDim.x * blockIdx.x;

for (int offset = blockDim.x / 2 ; offset > 0 ; offset /= 2)
{
        if (tid < offset)
        {
                i_data[index] += i_data[index + offset];
        }
        __synchthreads();
}
```

unrolling

What is loop unrolling

- In loop unrolling, rather than writing the body of a loop once and using a loop to execute it repeatedly, the body is written in code multiple times.
- The number of copies made of the loop body is called the loop unrolling factor

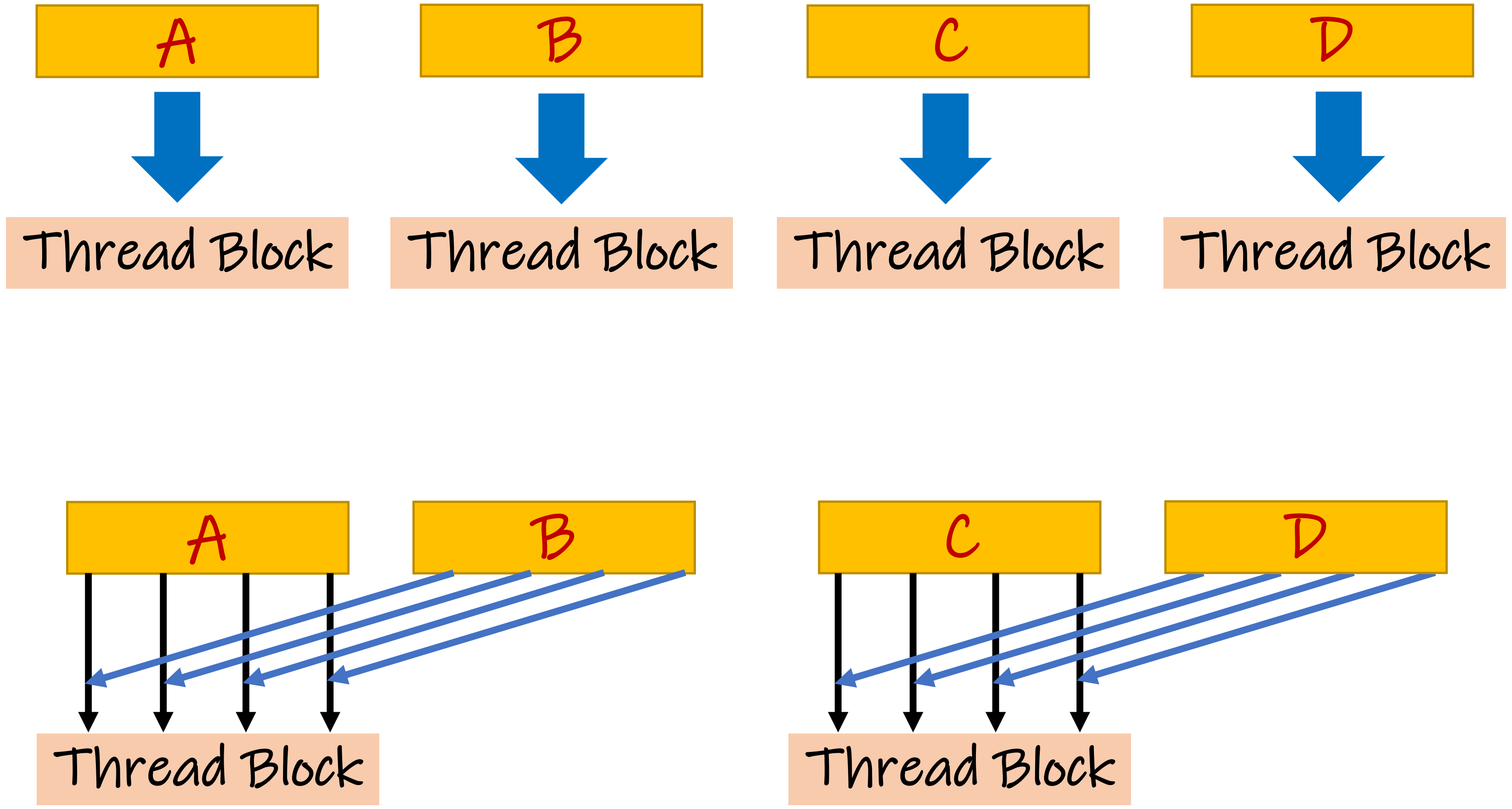
```
for ( int i  =  0;  i < 100 ;  i++ )
{
        sum += a[i];
}
```

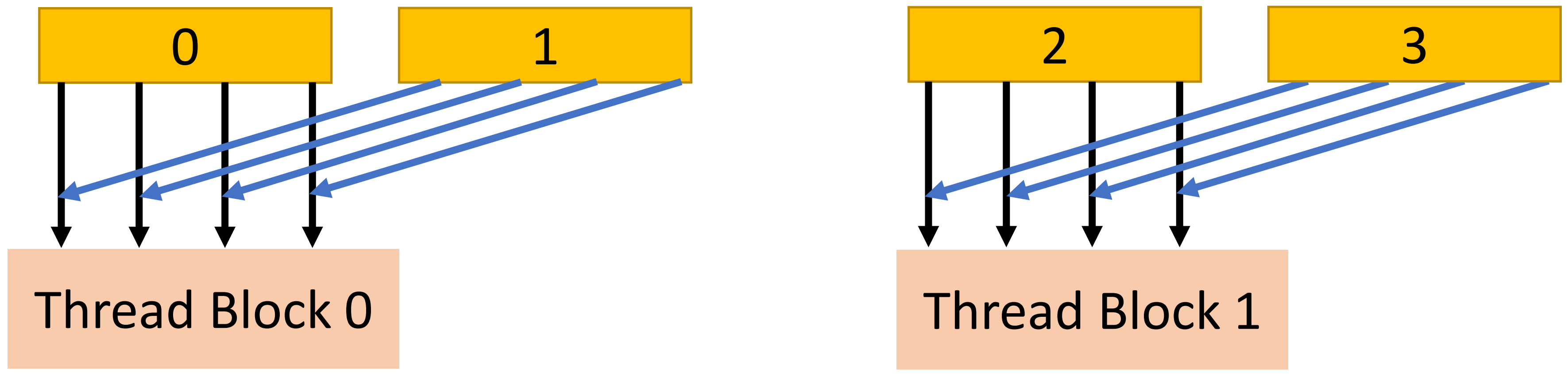
$$\Downarrow$$

```
for ( int  i  =  0 ;  i < 100;  i += 2 )
{
        sum  + =  input[i];
        sum  + =  input[i+1];
}
```

Thread blocks unrolling

Unrolling we are going to apply here is somewhat different than our loop unrolling example in sequential code. You can refer it as thread block unrolling. But the main purpose remains same, reduction of the instruction overheads.


A
B
C
D
Thread Block
Thread Block
Thread Block
Thread Block
A
B
C
D
Thread Block
Thread Block



0
1
2
3
Thread Block 0
Thread Block 1