

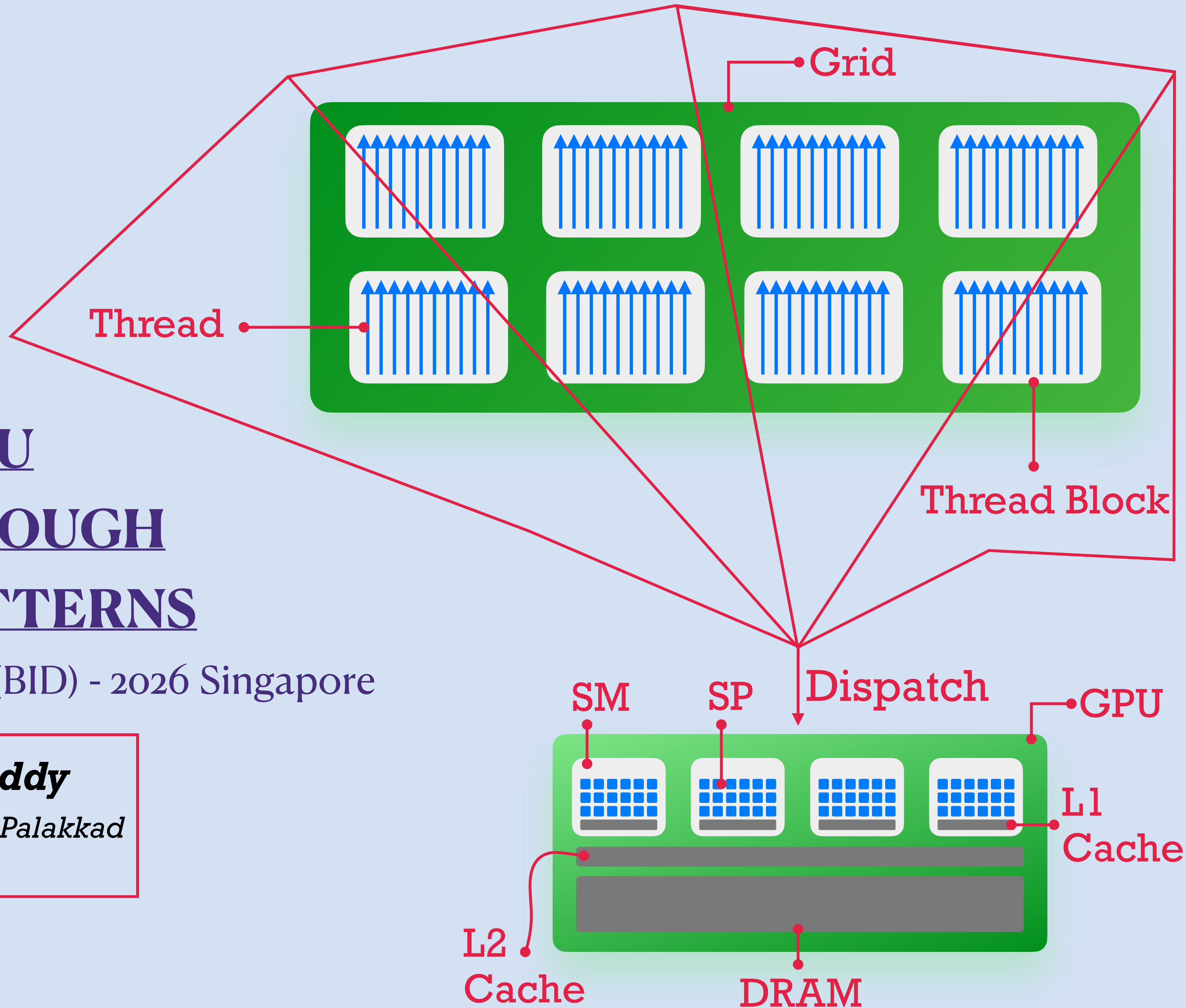
ENHANCING GPU PERFORMANCE THROUGH OPTIMIZED LOOP PATTERNS

Benchmarking in the Data Center (BID) - 2026 Singapore

Chekkala Sandeep Reddy

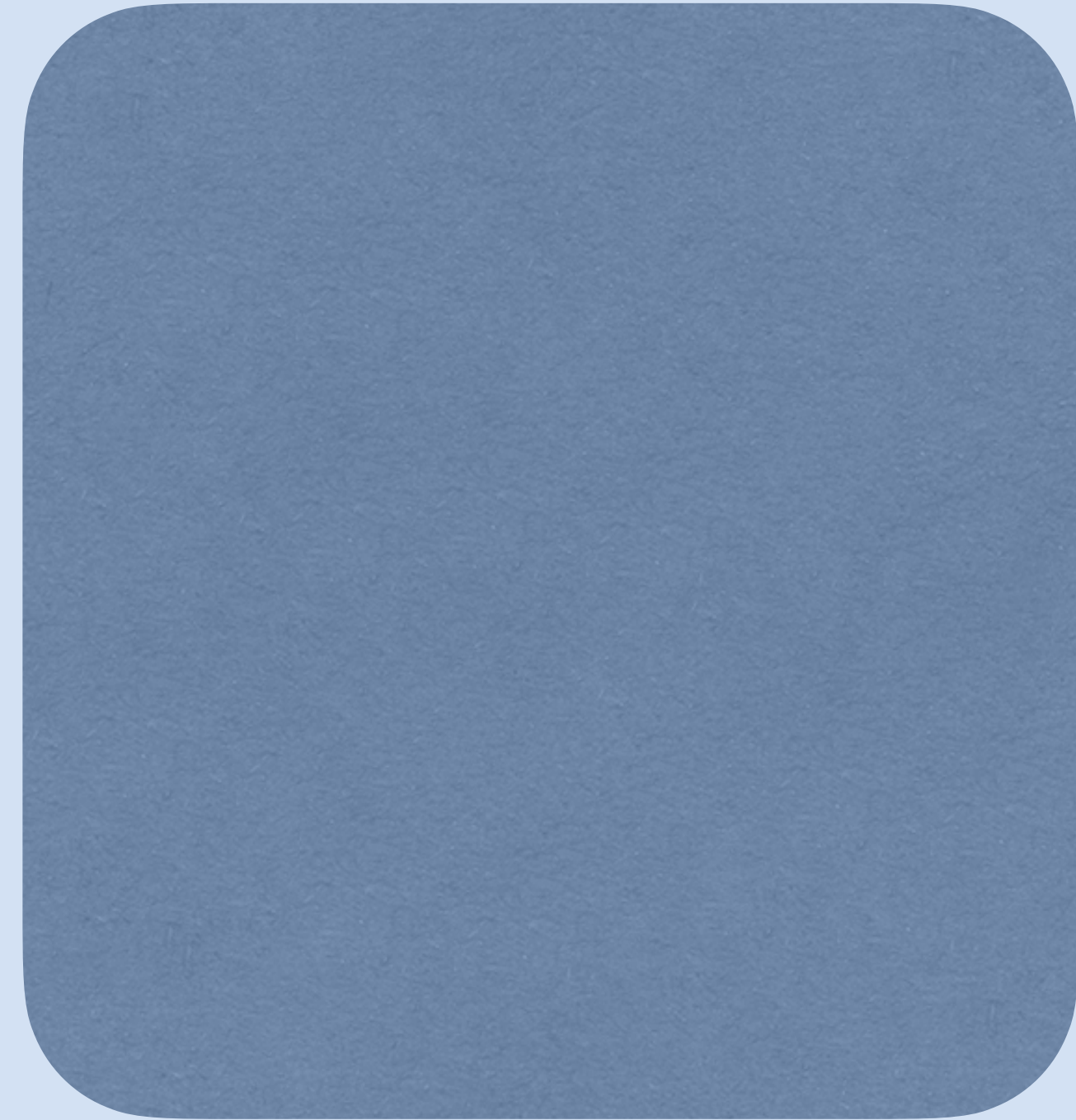
Indian Institute of Technology Palakkad

sandeep.chekkala.wh@gmail.com

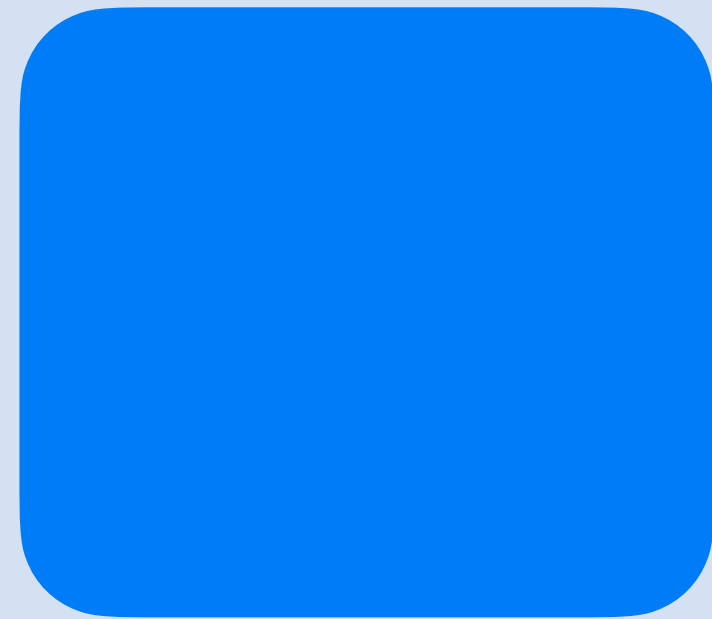


PROCESSING UNIT

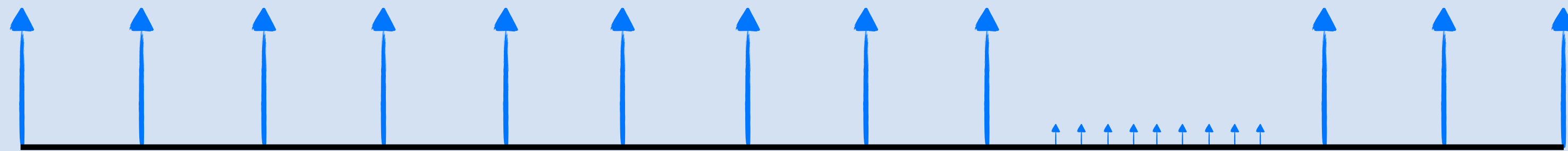
CPU core



Streaming Processor (SP)



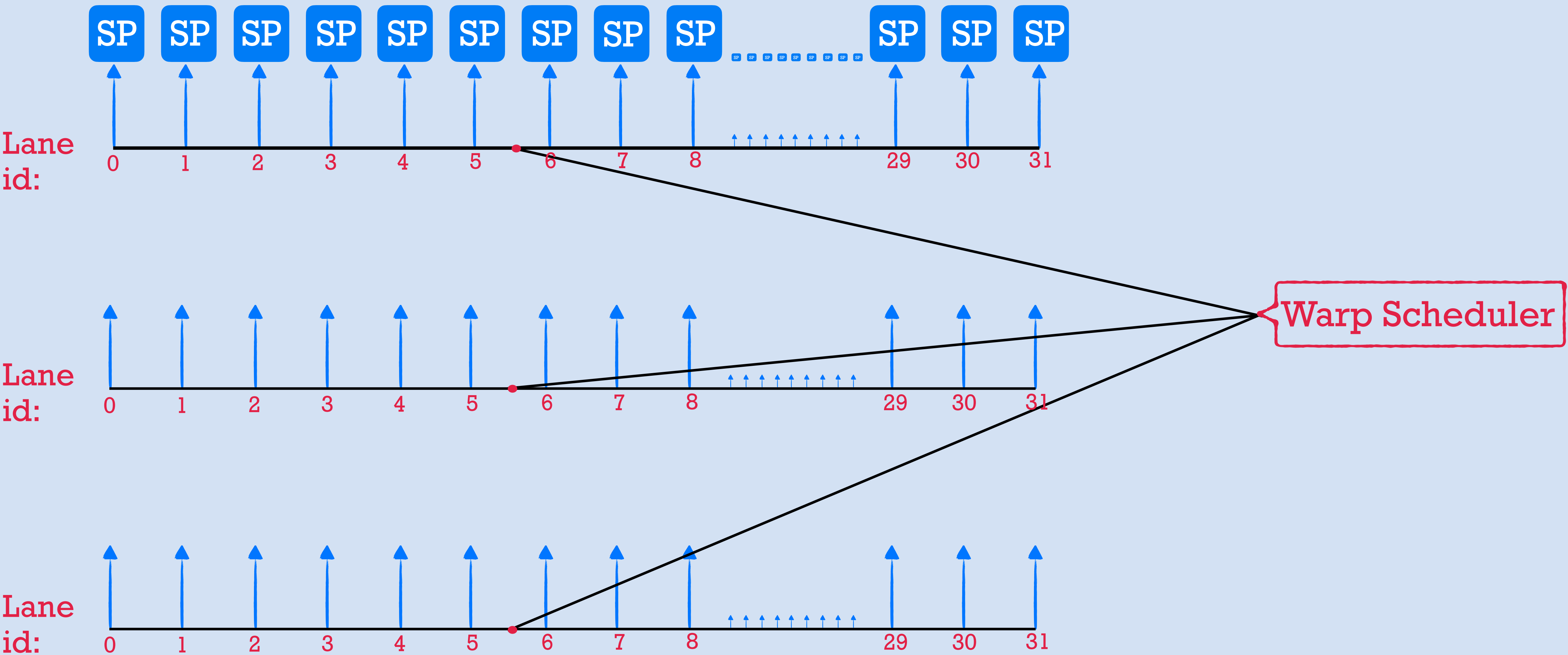
WARP EXECUTION MODEL (SIMT ARCHITECTURE)



The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called ***warps***.

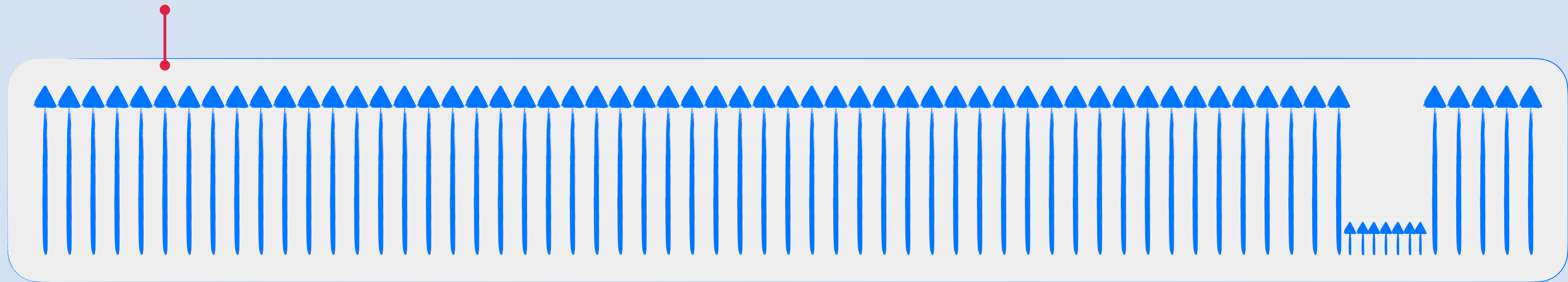
WARP EXECUTION MODEL (SIMT ARCHITECTURE)

At every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads



THREAD BLOCK

Thread Block



ThreadId: 0 1 2 255

Block Size = 256

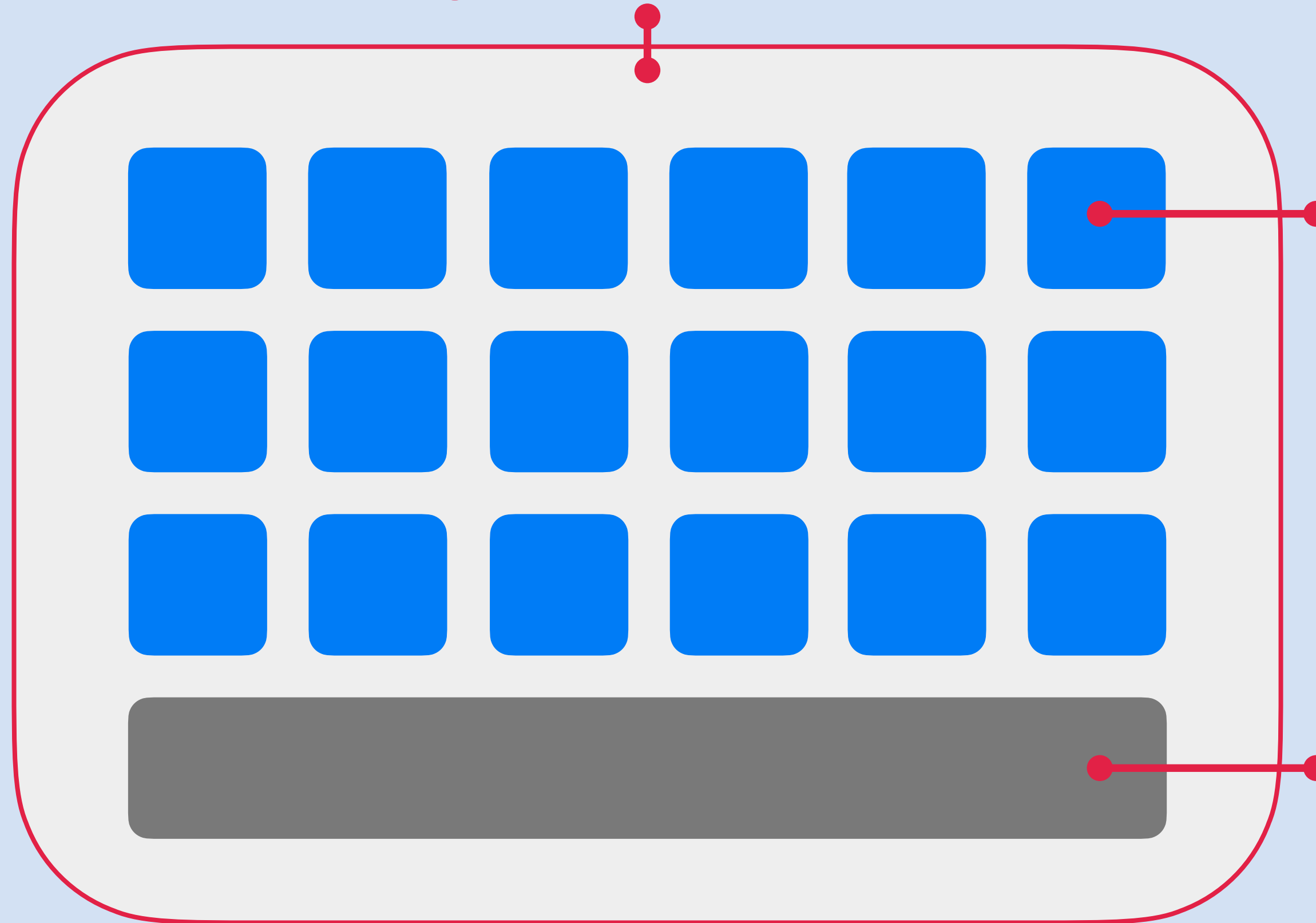
8 Warps

STREAMING MULTI-PROCESSOR

Example:

NVIDIA GeForce RTX
2080 Ti
(Turing Architecture)

Streaming Multi-Processor (SM)



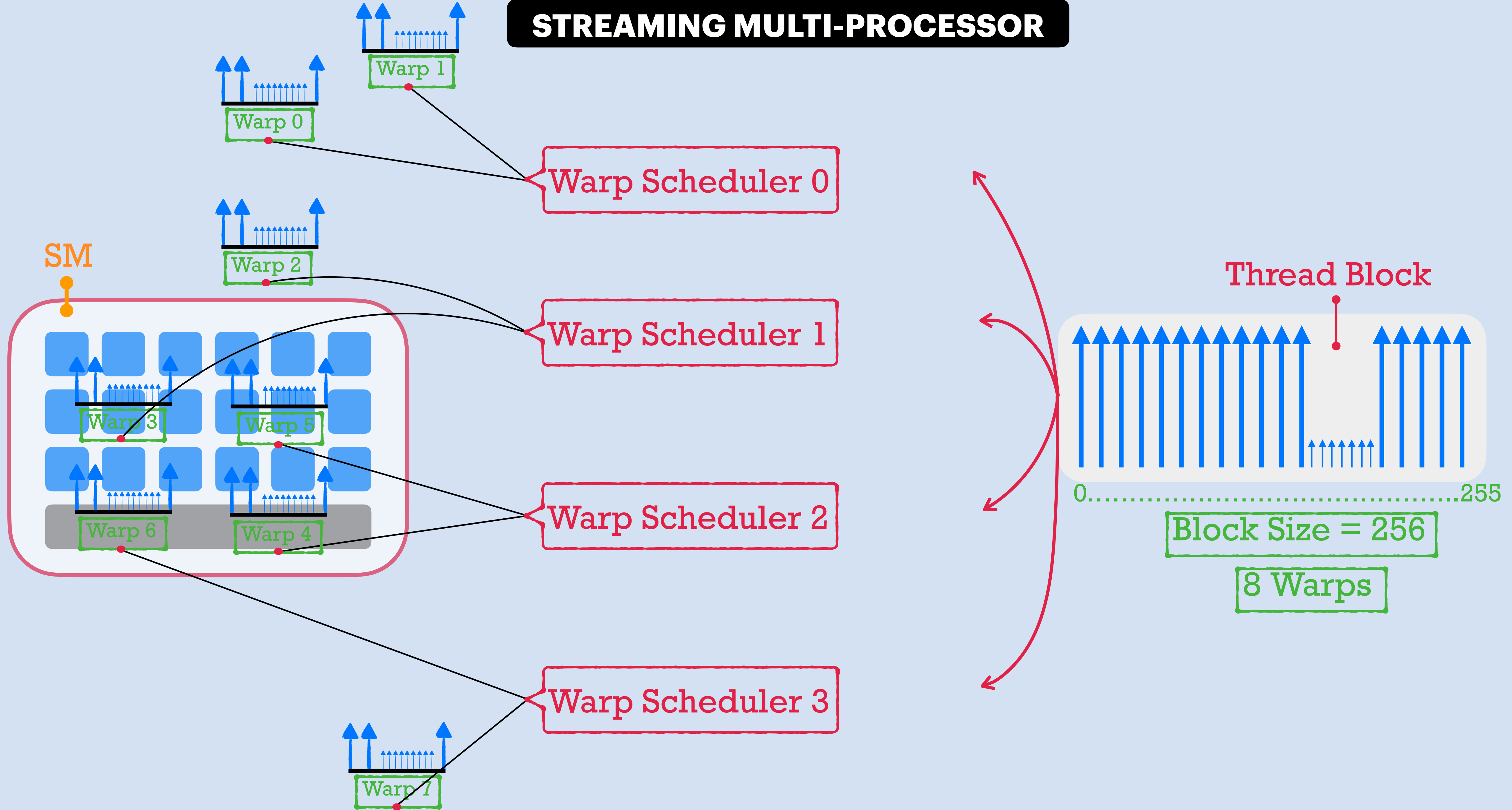
1
Streaming
Processor (SP)
128

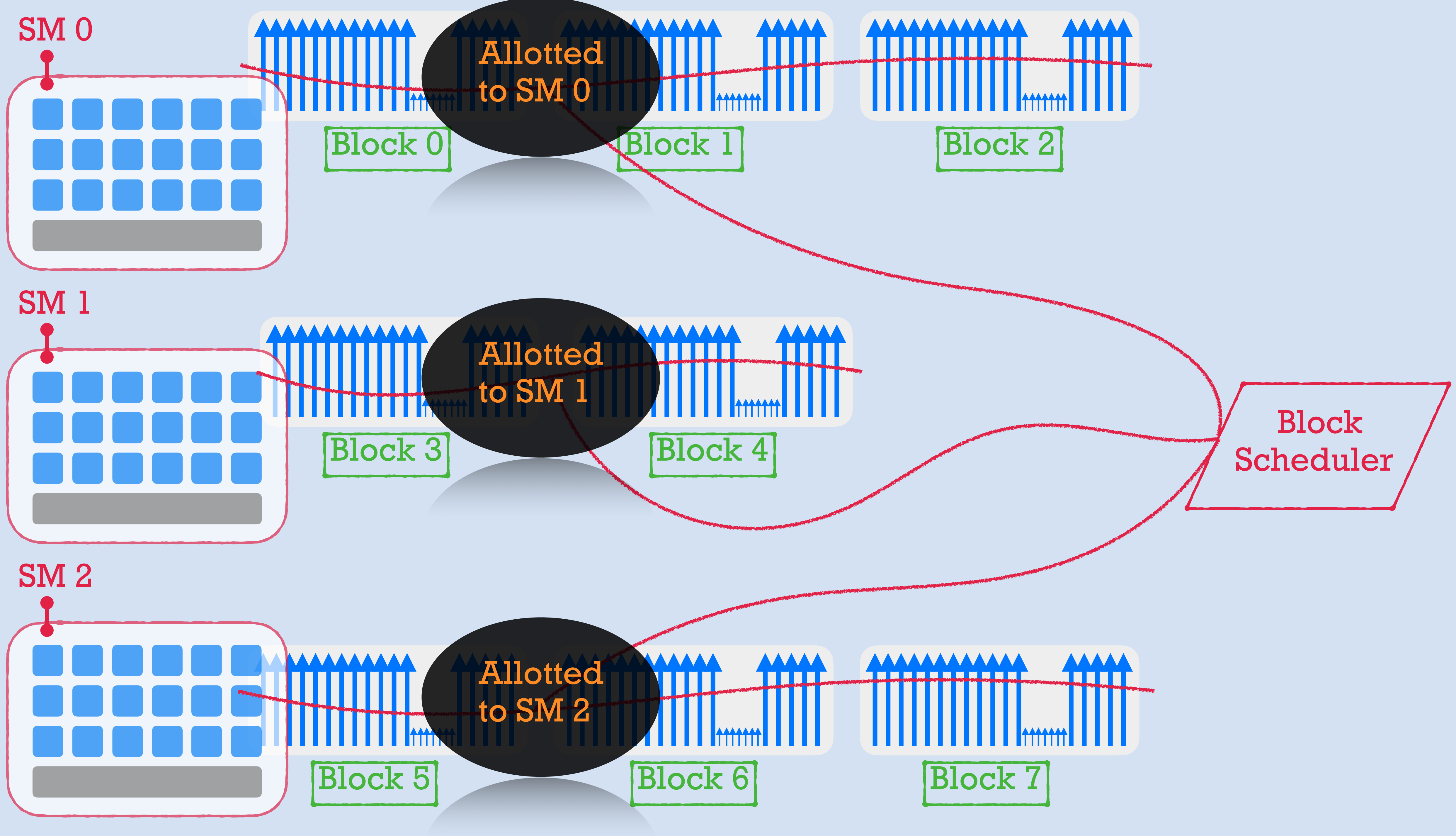
2
Unified Data
Cache = L1 Cache + Shared
Memory
96 KB

3 Warp Schedulers 4

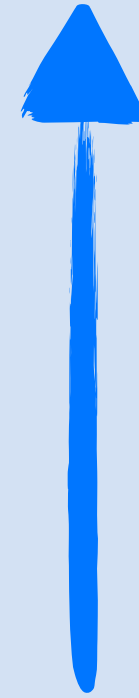
When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution.

STREAMING MULTI-PROCESSOR





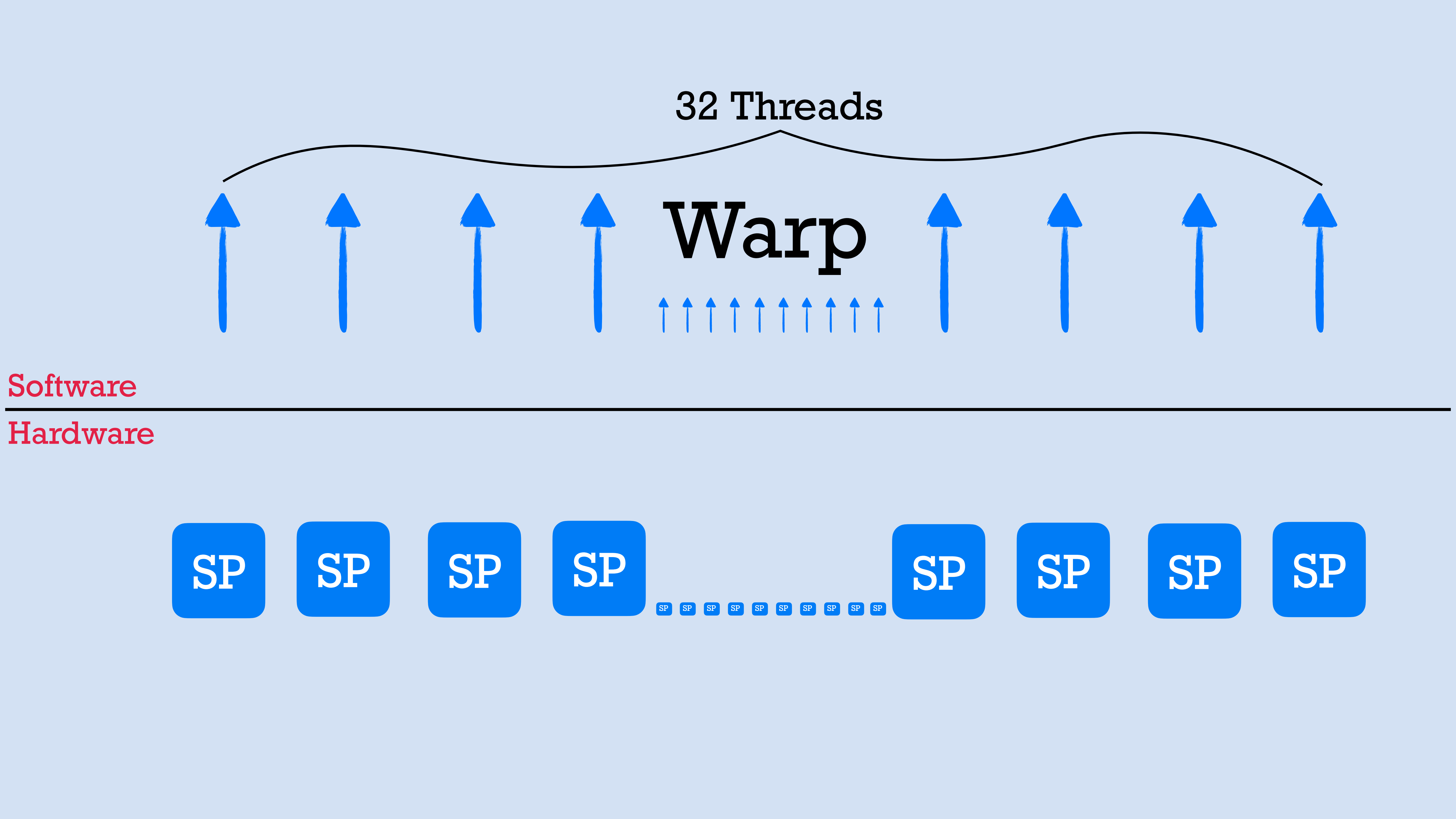
Thread



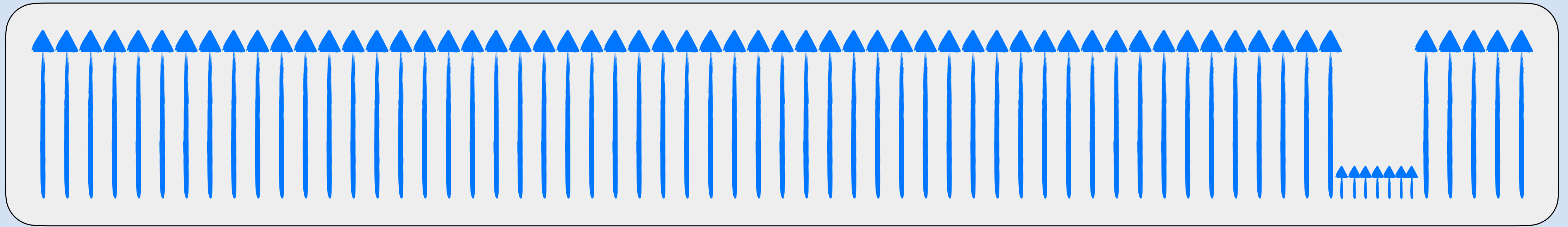
Software

Hardware

SP



Thread Block



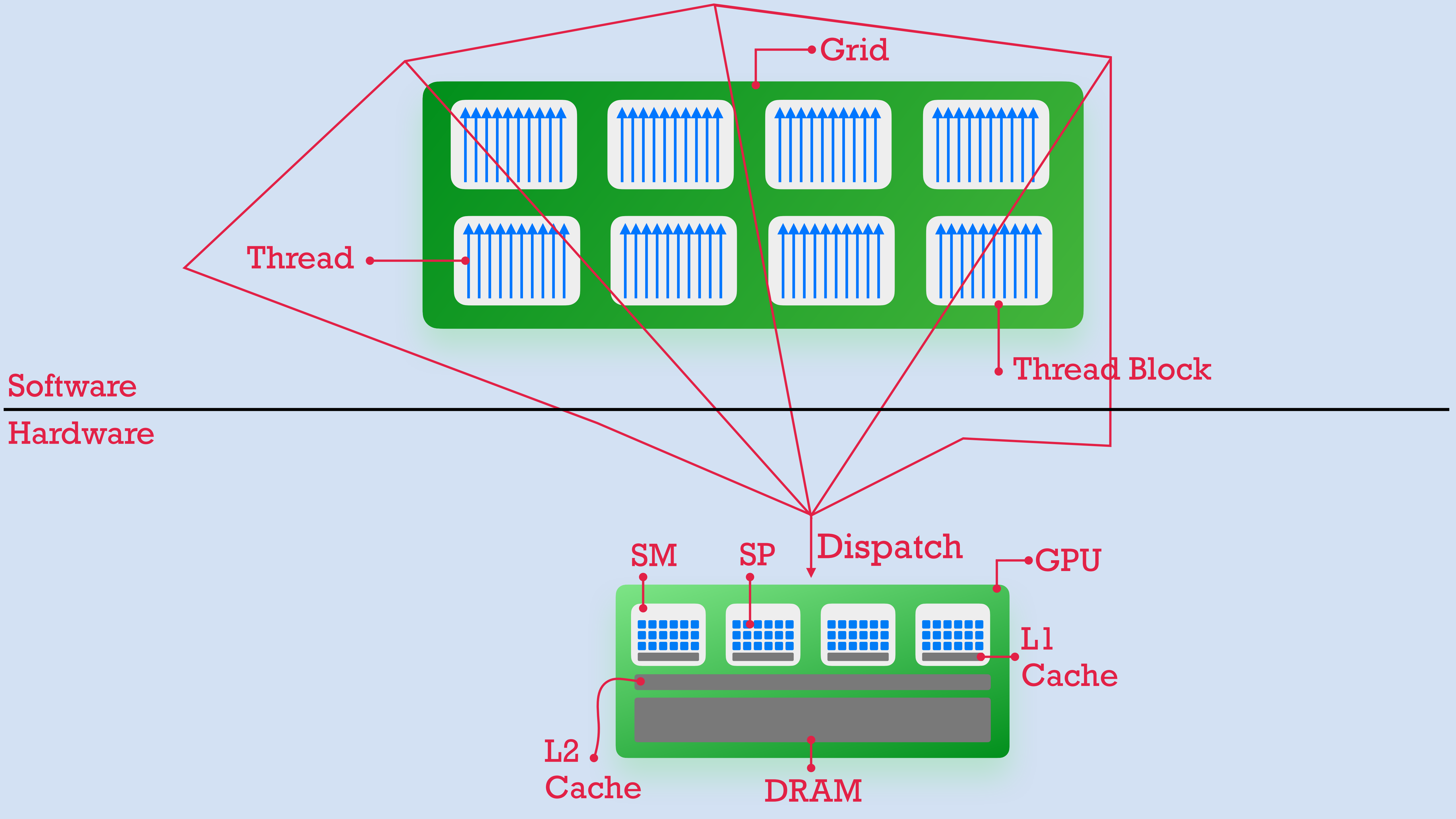
Software

Hardware

SM



L1
Cache



REDUCTION

Given an array A of n elements. For $i = 0$ to $n-1$ $A[i]$ represents the i th element.

The problem of reduction w.r.t binary associative operator $\oplus$ is reducing array A into single value $A[0] \oplus A[1] \oplus A[2] \dots \oplus A[n-1]$.

In our research we assume that $\oplus$ is commutative as well.

Let's delve into a specific example of addition. The efficient global memory access pattern kernels discussed in the following slides are written for addition but can be easily extended to a wide range of applications including but not limited to arithmetic associative operators, dot product, norms etc..

KERNEL INVOCATION STRATEGY

```
1 constexpr unsigned int min_work_per_thread = 16;
2
3 constexpr unsigned int block_size_256_power = 8;
4 constexpr unsigned int block_size_256      = (1 << block_size_256_power);
5
6 constexpr unsigned int block_size_512_power = 9;
7 constexpr unsigned int block_size_512      = (1 << block_size_512_power);
8
9 constexpr unsigned int num_blocks_256_power = 8;
10 constexpr unsigned int num_blocks_256      = (1 << num_blocks_256_power);
11
12 constexpr unsigned int warp_size_power     = 5;
13 constexpr unsigned int warp_size           = (1 << warp_size_power);
14
```

Compile time constants

We are defining a new parameter, to ensure that when a thread is created it handles at least a certain number of work items except for very small inputs

```
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29                 warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_512_power>
34             <<< 1, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36 }
```

Notice the power constants, which are useful for bitwise operations (which are faster than arithmetic)

KERNEL INVOCATION STRATEGY

```
12 constexpr unsigned int warp_size_power = 1;
13 constexpr unsigned int warp_size = 32;
14
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29                 warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_256_power>
34             <<< num_blocks_256, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36     else
37     {
38         treeReductionXStrideKernel <block_size_512_power>
39             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
40     }
41
42     cudaDeviceSynchronize();
43     return;
44 }
```

Launching with 1 block with 256 threads with shared memory of 32
x sizeof(long long int) = 32 x 8 bytes = 256 bytes

KERNEL INVOCATION STRATEGY

```
12 constexpr unsigned int warp_size_power = 16;
13 constexpr unsigned int warp_size = 16;
14
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29             warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_256_power>
34             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36     else
37     {
38         treeReductionXStrideKernel <block_size_512_power>
39             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
40     }
41
42     cudaDeviceSynchronize();
43     return;
44 }
```

Launching with 256 threads per block

Launching with less than 256 blocks

Each thread handles at least 16 work items

KERNEL INVOCATION STRATEGY

```
12 constexpr unsigned int warp_size_power = 16;
13 constexpr unsigned int warp_size = 16;
14
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29             warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_256_power>
34             <<< num_blocks_256, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36     else
37     {
38         treeReductionXStrideKernel <block_size_512_power>
39             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
40     }
41
42     cudaDeviceSynchronize();
43     return;
44 }
```

Launching with 256 threads per block

Launching with 256 blocks per grid

Each thread handles at least 16 work items

KERNEL INVOCATION STRATEGY

```
12 constexpr unsigned int warp_size_power = 16;
13 constexpr unsigned int warp_size = 16;
14
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29             warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_256_power>
34             <<< num_blocks_256, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36     else
37     {
38         treeReductionXStrideKernel <block_size_512_power>
39             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
40     }
41
42     cudaDeviceSynchronize();
43     return;
44 }
```

Launching with 512 threads per block

Launching with 256 blocks per grid

Each thread handles at least 16 work items

KERNEL INVOCATION STRATEGY

```
12 constexpr unsigned int warp_size_power = 1;
13 constexpr unsigned int warp_size = 32;
14
15 void computeReduction(const unsigned int N, const int* d_a, long long int* d_sum, long long int init)
16 {
17     cudaMemset(d_sum, init, sizeof(long long int));
18
19     // Kernel invocation
20     if (N < block_size_256 * min_work_per_thread)
21     {
22         treeReductionXStrideKernel <block_size_256_power>
23             <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
24     }
25     else if (N < num_blocks_256 * block_size_256 * min_work_per_thread)
26     {
27         treeReductionXStrideKernel <block_size_256_power>
28             <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256,
29             warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
30     }
31     else if (N < num_blocks_256 * block_size_512 * min_work_per_thread)
32     {
33         treeReductionXStrideKernel <block_size_256_power>
34             <<< num_blocks_256, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
35     }
36     else
37     {
38         treeReductionXStrideKernel <block_size_512_power>
39             <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);
40     }
41
42     cudaDeviceSynchronize();
43     return;
44 }
```

Invoke

```
graph LR
    A["treeReductionXStrideKernel <block_size_256_power> <<< 1, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);"] --> D[Invoke]
    B["treeReductionXStrideKernel <block_size_256_power> <<< (N >> block_size_256_power) / min_work_per_thread, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);"] --> D
    C["treeReductionXStrideKernel <block_size_256_power> <<< num_blocks_256, block_size_256, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);"] --> D
    E["treeReductionXStrideKernel <block_size_512_power> <<< num_blocks_256, block_size_512, warp_size * sizeof(long long int) >>> (N, d_a, d_sum);"] --> D
    D --> F[ ]
```


TREE REDUCTION KERNEL

```
1 template <const unsigned int block_size_power>
2 __global__ void treeReductionXStrideKernel (const unsigned int N,
3                                             const int* __restrict__ a,
4                                             long long int* __restrict__ total_sum)
5 {
6     long long int partial_sum;
7
8     unsigned int iter = threadIdx.x + (blockIdx.x << block_size_power);
9     const unsigned int grid_stride = blockDim.x << block_size_power;
10    // Grid-stride access
11    partial_sum = a[iter];
12    iter += grid_stride;
13    while(__builtin_expect(iter < N, 1)){    // Branch prediction: Always take
14        partial_sum += a[iter];
15        iter += grid_stride;
16    }
17
18    // Tree reduction
19    partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
20    partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
21    partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
22    partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
23    partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
24
25    extern __shared__ long long int warp_partial_sum[];
26    const uint8_t lane_id = (threadIdx.x & 31);
27    const uint8_t warp_id = (threadIdx.x >> 5);
28
```

Kernel signature

We shall replace X with appropriate word
in following slides

TREE REDUCTION KERNEL

```
2 __global__ void treeReductionXStrideKernel (const unsigned int N,
3
4 long long int* __restrict__ total_sum)
5 {
6     long long int partial_sum;
7
8     unsigned int iter = threadIdx.x + (blockIdx.x << block_size_power);
9     const unsigned int grid_stride = gridDim.x << block_size_power;
10    // Grid-stride access
11    partial_sum = a[iter];
12    iter += grid_stride;
13    while(__builtin_expect(iter < N, 1)){ // Branch prediction: Always take
14        partial_sum += a[iter];
15        iter += grid_stride;
16    }
17
18    // Tree reduction
19    partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
20    partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
21    partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
22    partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
23    partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
24
25    extern __shared__ long long int warp_partial_sum[];
26    const uint8_t lane_id = (threadIdx.x & 31);
27    const uint8_t warp_id = (threadIdx.x >> 5);
28
29    // Writing to shared memory
30    if (lane_id == 0) warp_partial_sum[warp_id] = partial_sum;
31    __syncthreads();
32
```


TREE REDUCTION KERNEL

```
16 }  
17  
18 // Tree reduction
```

```
19 partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);  
20 partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);  
21 partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);  
22 partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);  
23 partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);  
24
```

```
25 extern __shared__ long long int warp_partial_sum[];
```

```
26 const uint8_t lane_id = (threadIdx.x & 31);
```

```
27 const uint8_t warp_id = (threadIdx.x >> 5);
```

```
28  
29 // Writing to shared memory
```

```
30 if (lane_id == 0) warp_partial_sum[warp_id] = partial_sum;
```

```
31 __syncthreads();
```

```
32  
33 if (warp_id == 0){
```

```
34     partial_sum = warp_partial_sum[lane_id];
```

```
35  
36     if constexpr (block_size_power >= 10) partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
```

```
37     if constexpr (block_size_power >= 9) partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
```

```
38     if constexpr (block_size_power >= 8) partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
```

```
39     if constexpr (block_size_power >= 7) partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
```

```
40     if constexpr (block_size_power >= 6) partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
```

```
41  
42     if (lane_id == 0) atomicAdd(total_sum, partial_sum);
```

```
43 }
```

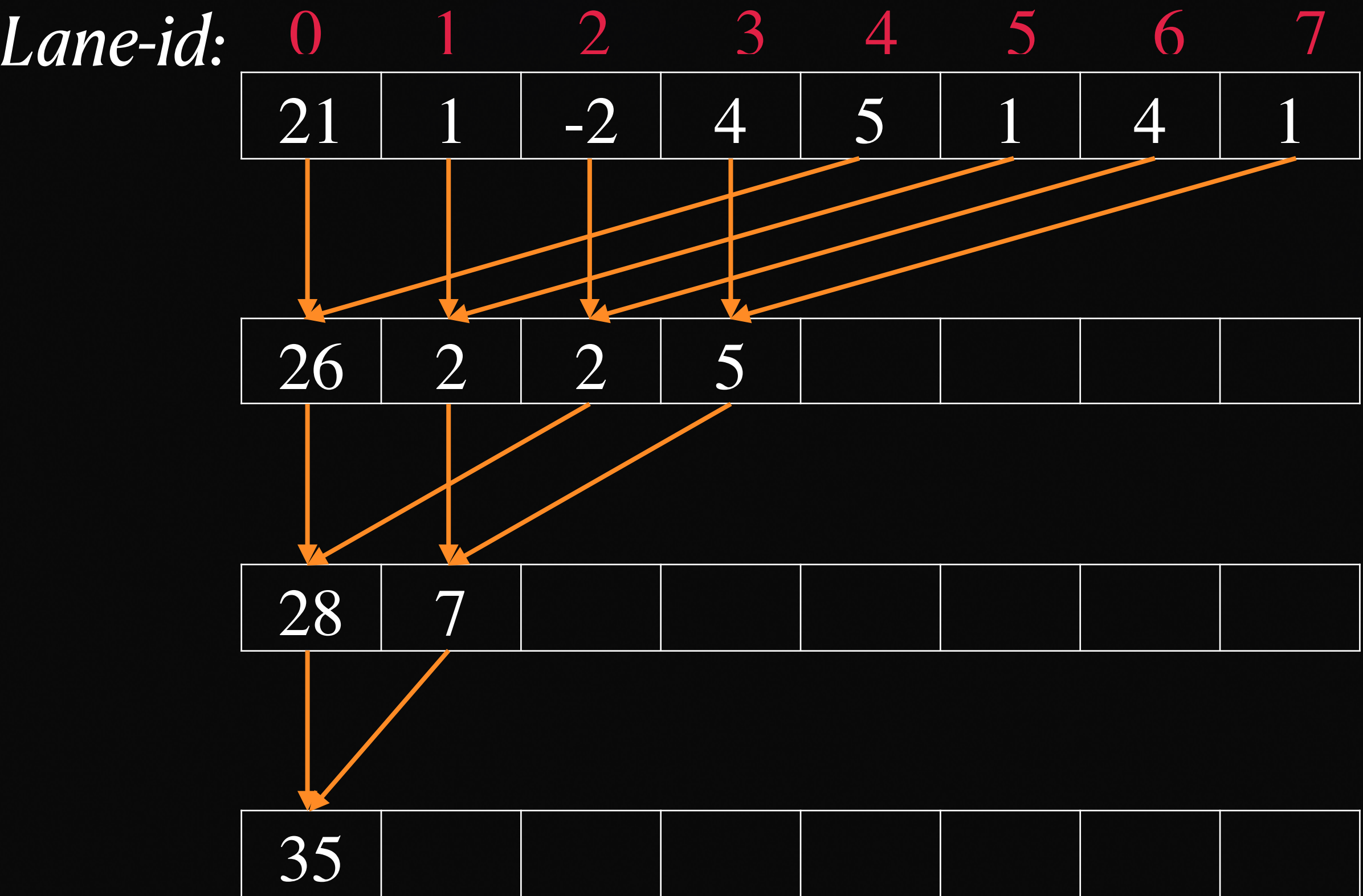
```
44 }
```


Warp level primitives

- Warp shuffle functions such as `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync` and `__shfl_xor_sync` exchange a variable between active threads within a warp as specified by mask without use of shared memory (via registers). The exchange occurs simultaneously for all active threads within the warp (and named in mask), moving 4 or 8 bytes of data per thread depending on the type.

```
int __shfl_down_sync(unsigned mask, int var, unsigned delta, int width=warpSize);
```

Tree reduction using warp shuffle functions:



```
val += __shfl_down_sync(0x000000FF, val, 4)
```

```
val += __shfl_down_sync(0x0000000F, val, 2)
```

```
val += __shfl_down_sync(0x00000003, val, 1)
```


TREE REDUCTION KERNEL

// Tree reduction

```
partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
```



TREE REDUCTION KERNEL

```
// Tree reduction
```

```
partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
partial_sum += __shfl_down_sync(0x000FFFFF, partial_sum, 8);
partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
```

```
const uint8_t lane_id = (threadIdx.x & 31);  
const uint8_t warp_id = (threadIdx.x >> 5);
```

```
// Writing to shared memory
```

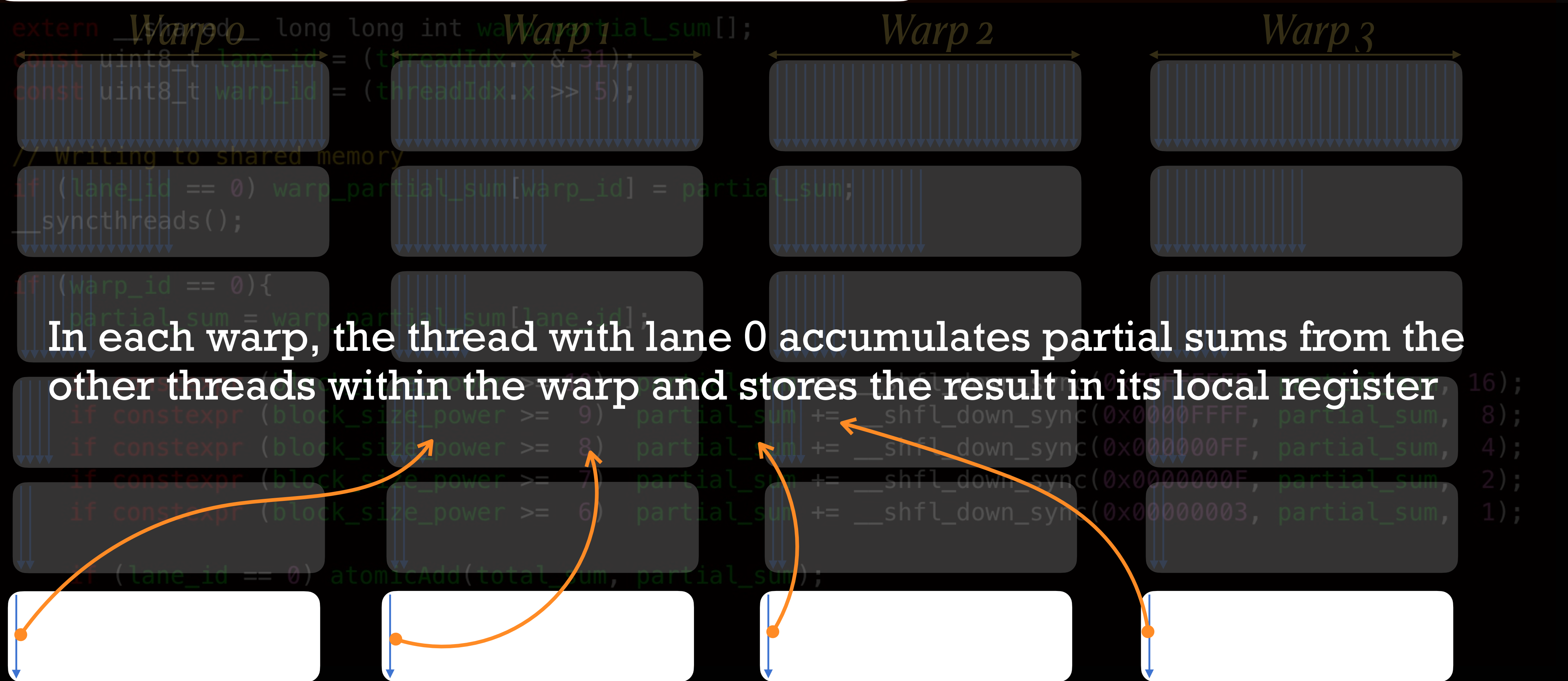
```
if (warp_id == 0){
    partial_sum = warp_partial_sum[lane_id];
```

other threads within the warp and stores the result in its local register

```
if constexpr (block_size_power >= 9) partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
if constexpr (block_size_power >= 8) partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
if constexpr (block_size_power >= 7) partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
if constexpr (block_size_power >= 6) partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
```

```
if (lane_id == 0) atomicAdd(&total_sum, partial_sum);
```

In each warp, the thread with lane 0 accumulates partial sums from the other threads within the warp and stores the result in its local register,



TREE REDUCTION KERNEL

```
16 }
17
18 // Tree reduction
19 partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
20 partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
21 partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
22 partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
23 partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
```

```
24
25 extern __shared__ long long int warp_partial_sum[]; —————→
```

```
26 const uint8_t lane_id = (threadIdx.x & 31);
27 const uint8_t warp_id = (threadIdx.x >> 5);
28
29 // Writing to shared memory
30 if (lane_id == 0) warp_partial_sum[warp_id] = partial_sum;
31 __syncthreads();
32
33 if (warp_id == 0){
34     partial_sum = warp_partial_sum[lane_id];
```

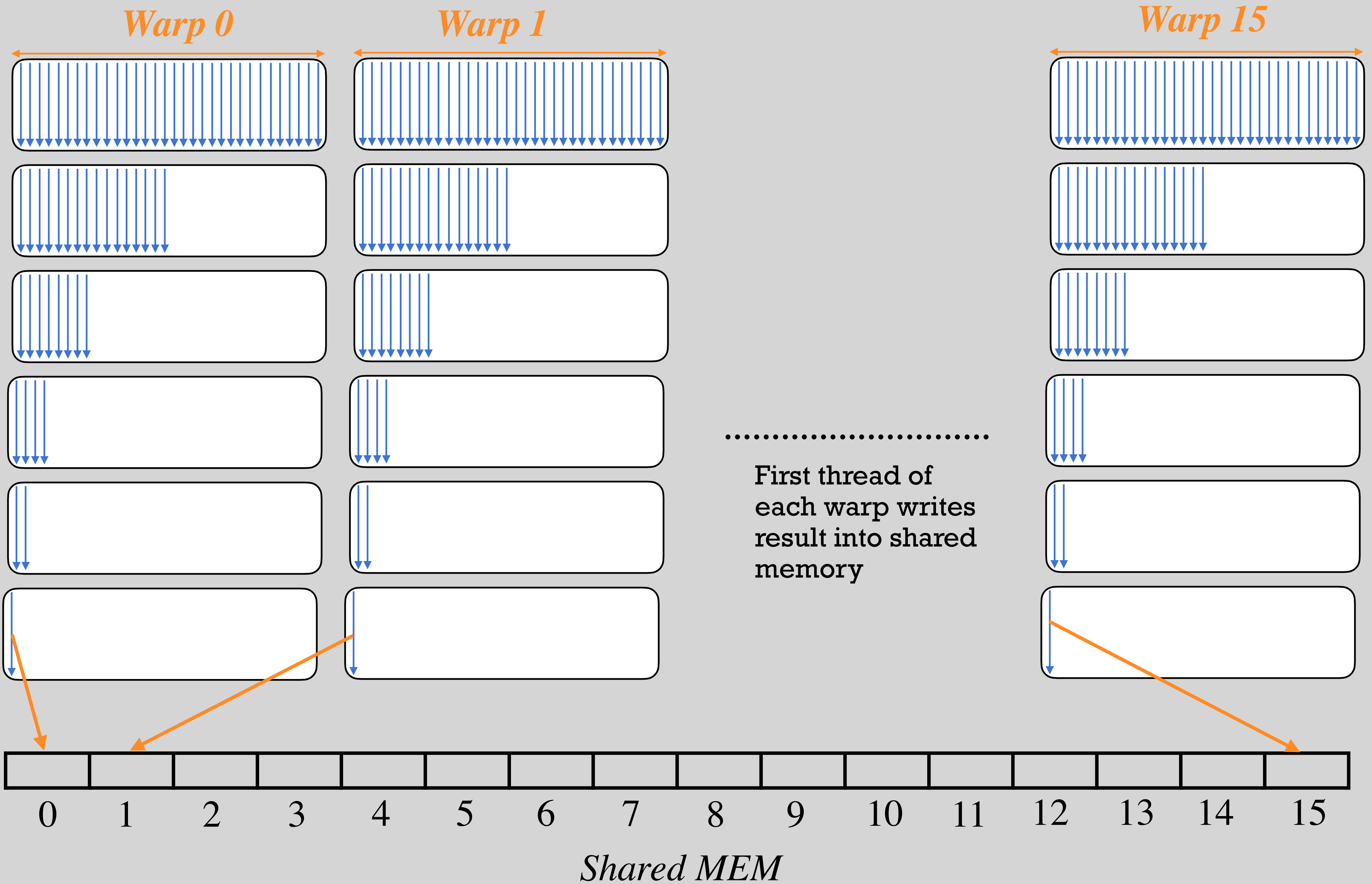
```
35
36     if constexpr (block_size_power >= 10) partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
37     if constexpr (block_size_power >= 9) partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 8);
38     if constexpr (block_size_power >= 8) partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 4);
39     if constexpr (block_size_power >= 7) partial_sum += __shfl_down_sync(0x0000000F, partial_sum, 2);
40     if constexpr (block_size_power >= 6) partial_sum += __shfl_down_sync(0x00000003, partial_sum, 1);
41
42     if (lane_id == 0) atomicAdd(total_sum, partial_sum);
43 }
```

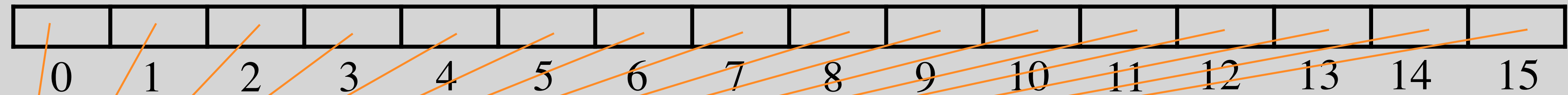
```
44 }
```

Shared Memory is a type of fast, user-managed memory that is shared among all threads within the same block in a CUDA kernel.

Dynamic Shared Memory is a type of shared memory whose size is determined at kernel launch time, unlike static shared memory with a fixed size defined at compile time

*Thread
block of
Size 512*





Shared MEM

For thread block of Size 512

```
if (warp_id == 0){
    partial_sum = warp_partial_sum[lane_id];

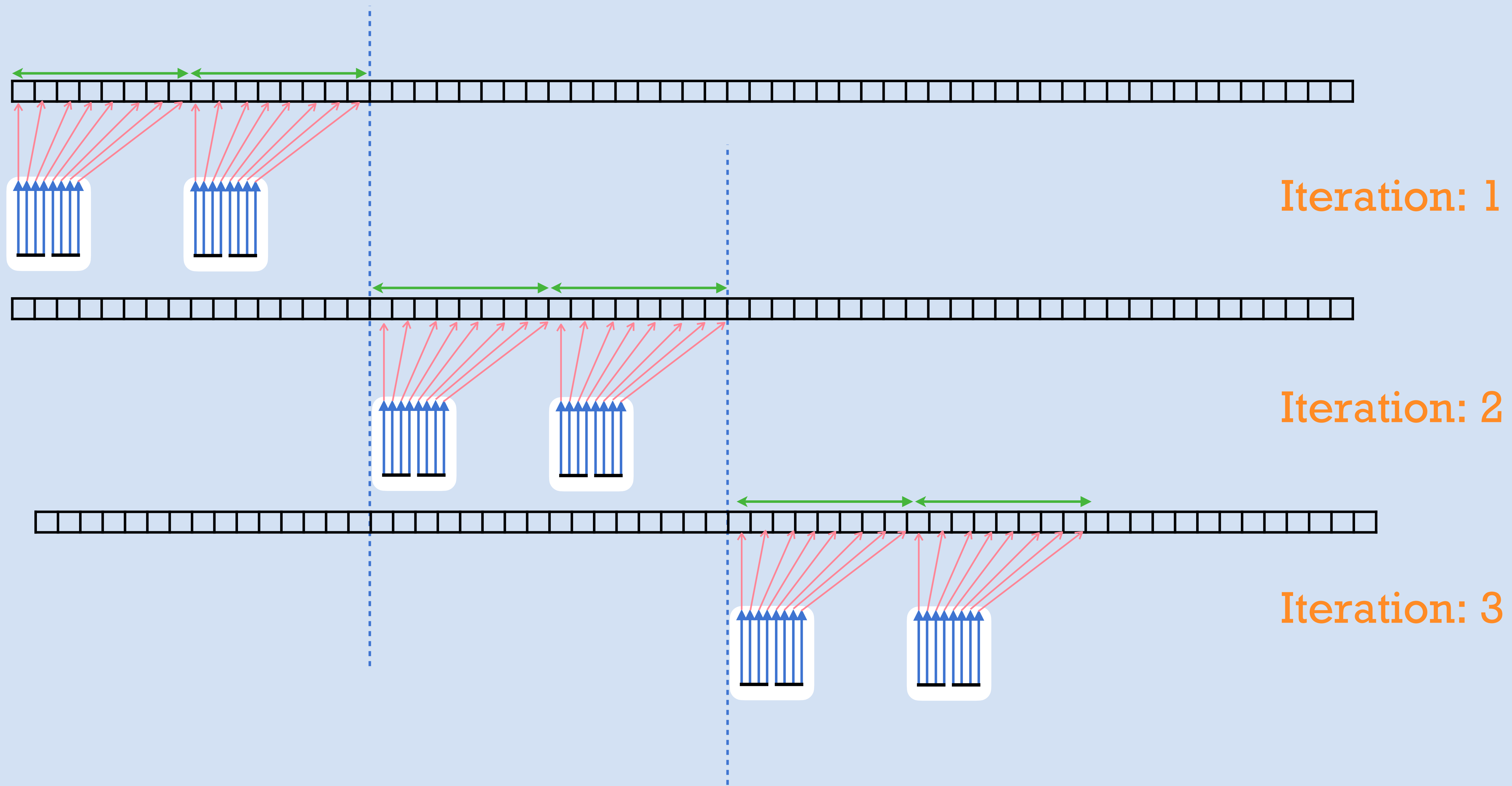
    if constexpr (block_size_power >= 10) partial_sum += __shfl_down_sync(0xFFFFFFFF, partial_sum, 16);
    if constexpr (block_size_power >= 9)  partial_sum += __shfl_down_sync(0x000FFFFF, partial_sum, 8);
    if constexpr (block_size_power >= 8)  partial_sum += __shfl_down_sync(0x0000FFFF, partial_sum, 4);
    if constexpr (block_size_power >= 7)  partial_sum += __shfl_down_sync(0x00000FFF, partial_sum, 2);
    if constexpr (block_size_power >= 6)  partial_sum += __shfl_down_sync(0x000000FF, partial_sum, 1);

    if (lane_id == 0) atomicAdd(total_sum, partial_sum);
}
```

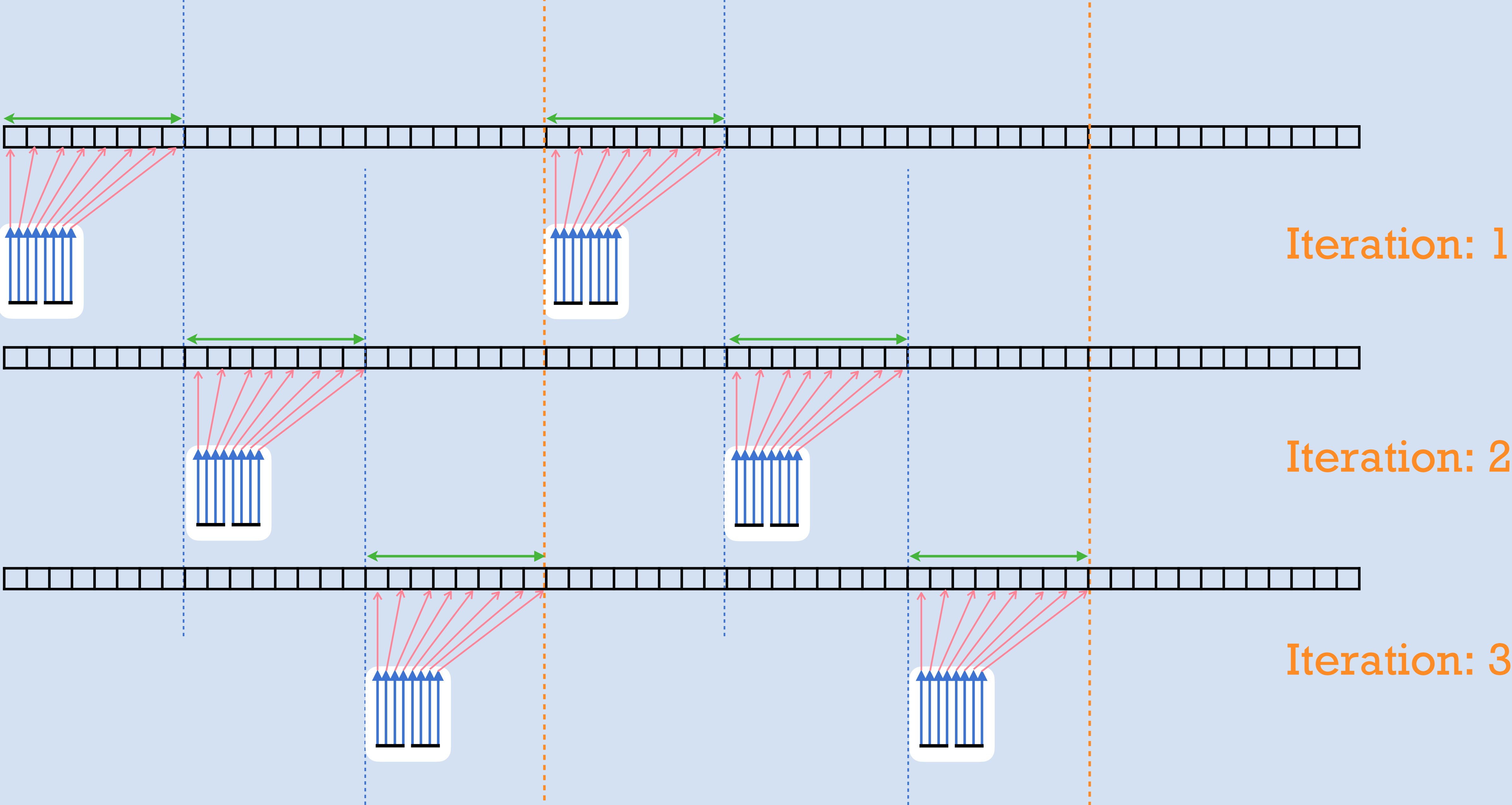
Warp 0

partial_sum —————→ *atomicAdd(total_sum, partial_sum)*

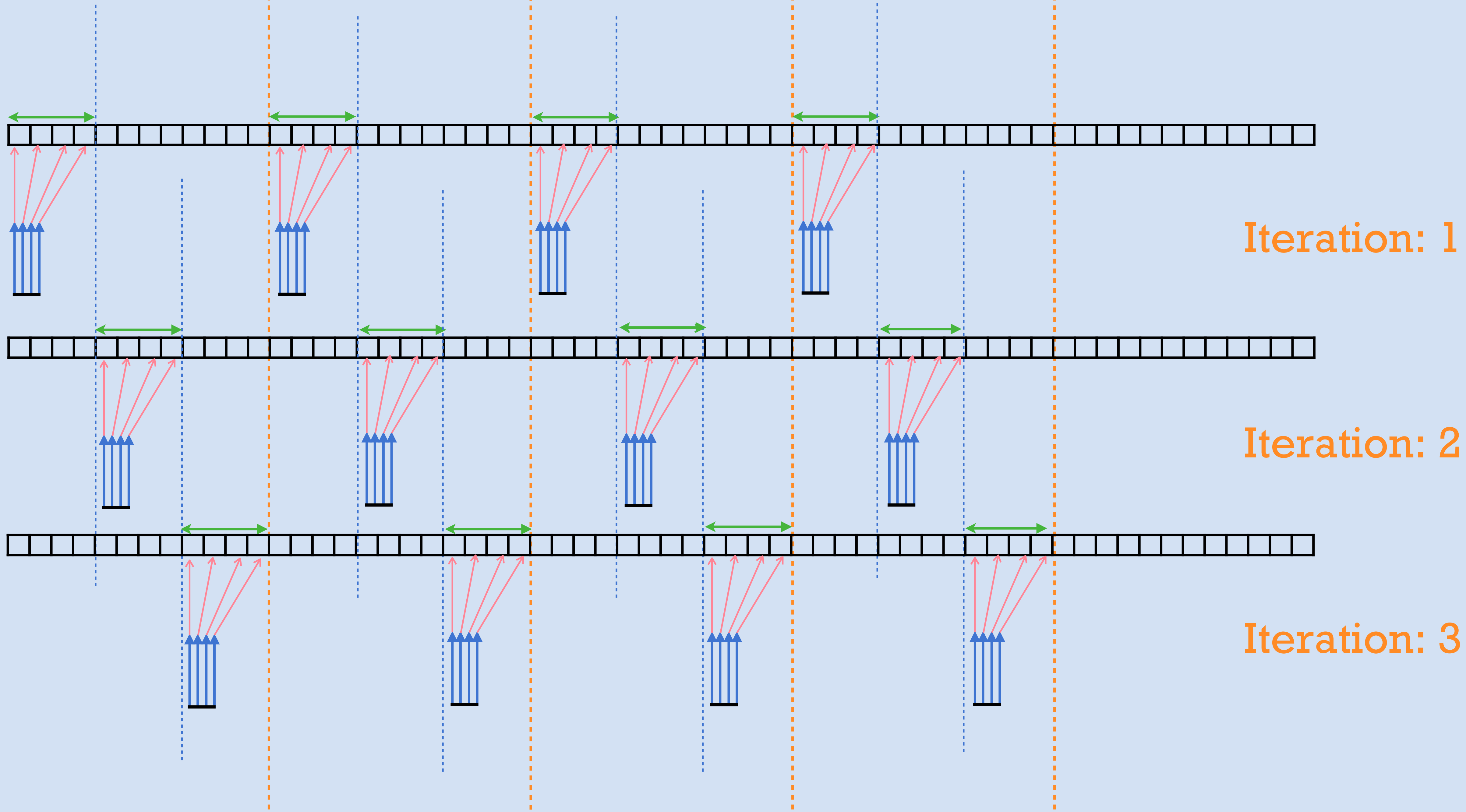
GRID STRIDE ACCESS PATTERN



BLOCK STRIDE ACCESS PATTERN

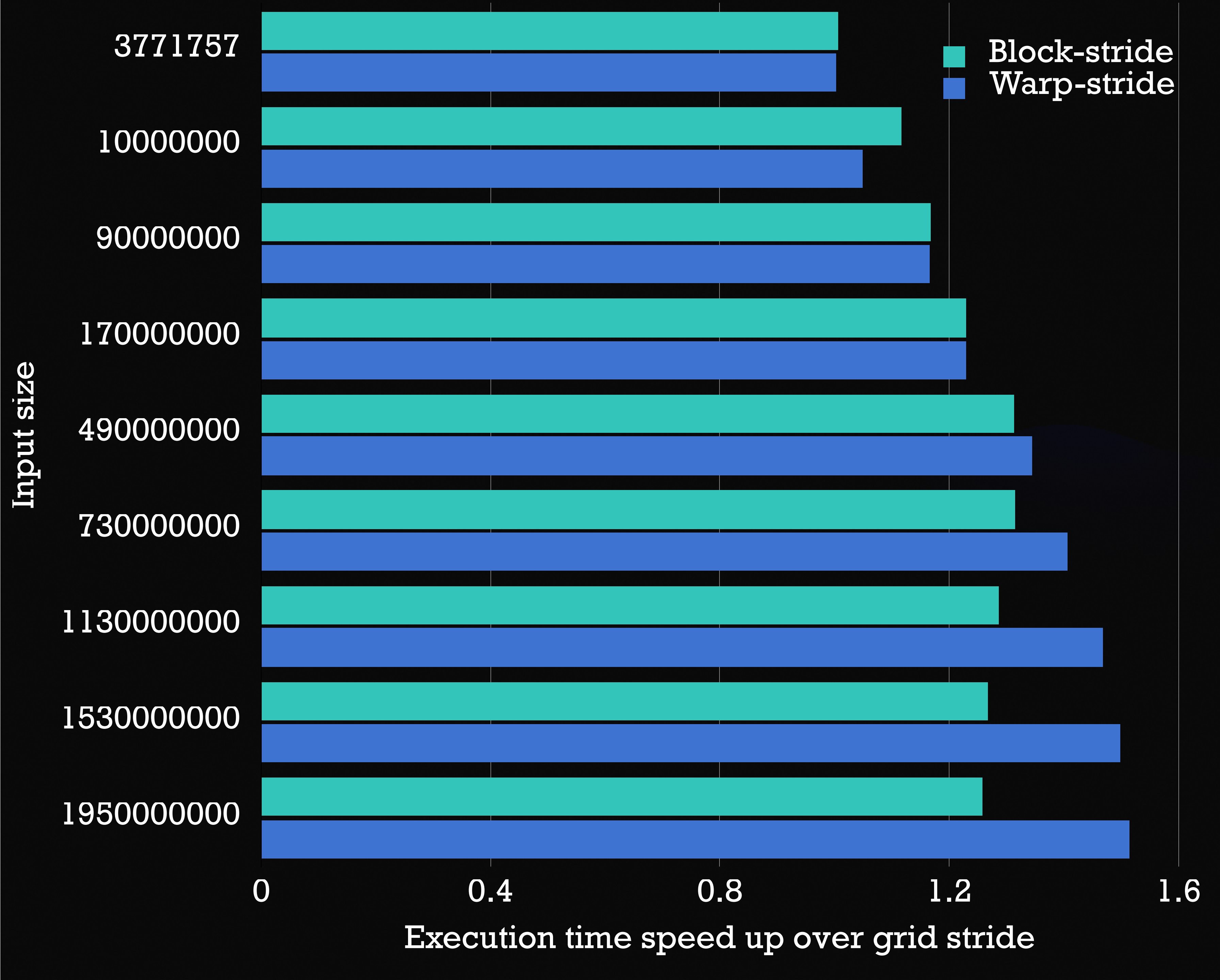


WARP STRIDE ACCESS PATTERN



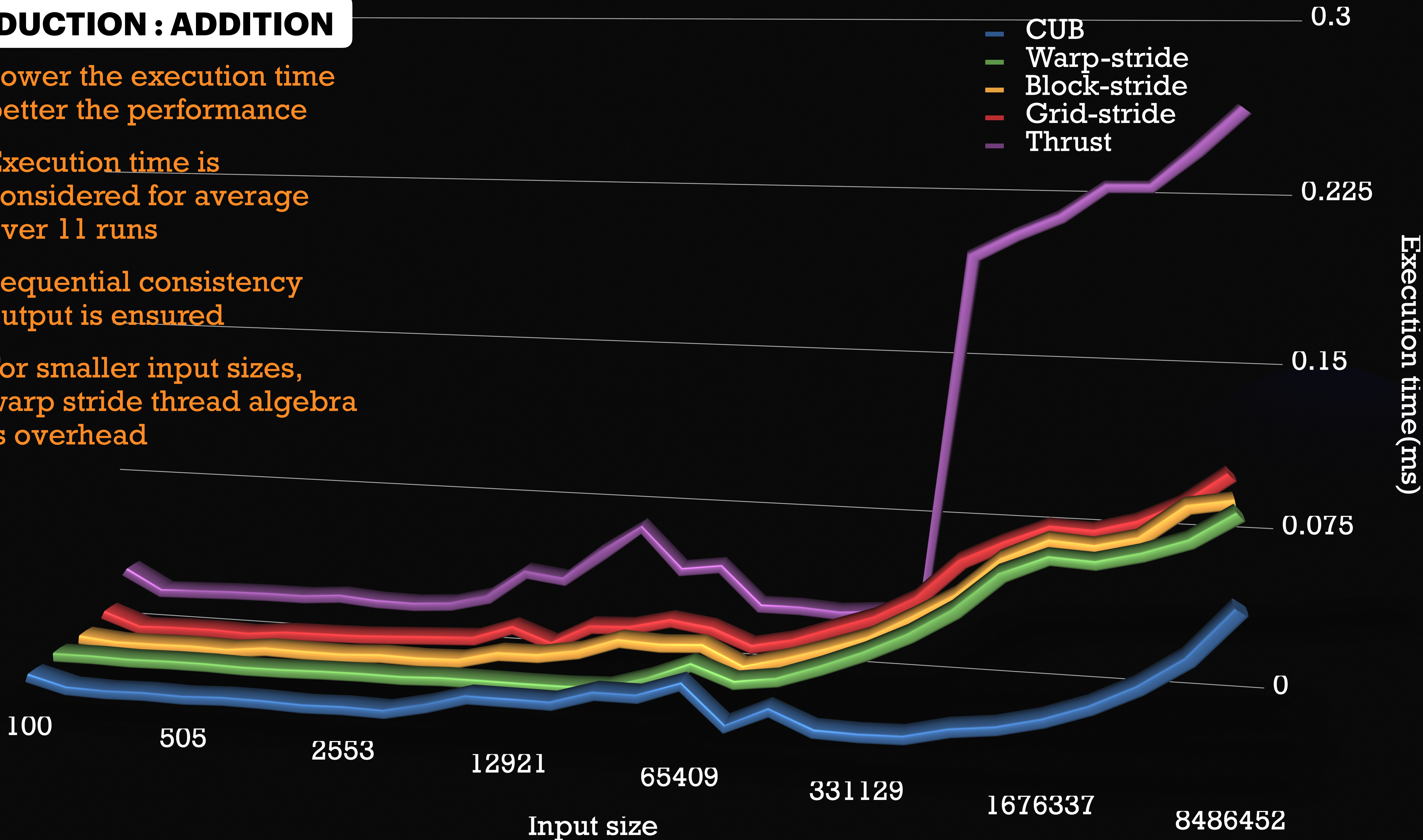
EXECUTION TIME SPEED OF WARP AND BLOCK STRIDE OVER GRID STRIDE FOR REDUCTION: ADDITION

- 1) Higher the speedup better the performance
- 2) Speedup of warp-stride or block-stride access is the ratio of their average execution time over 11 runs to that of grid-stride access
- 3) Sequential consistency of kernels output is ensured
- 4) Approximately 50 percent speed up is achieved for 2 billion input size



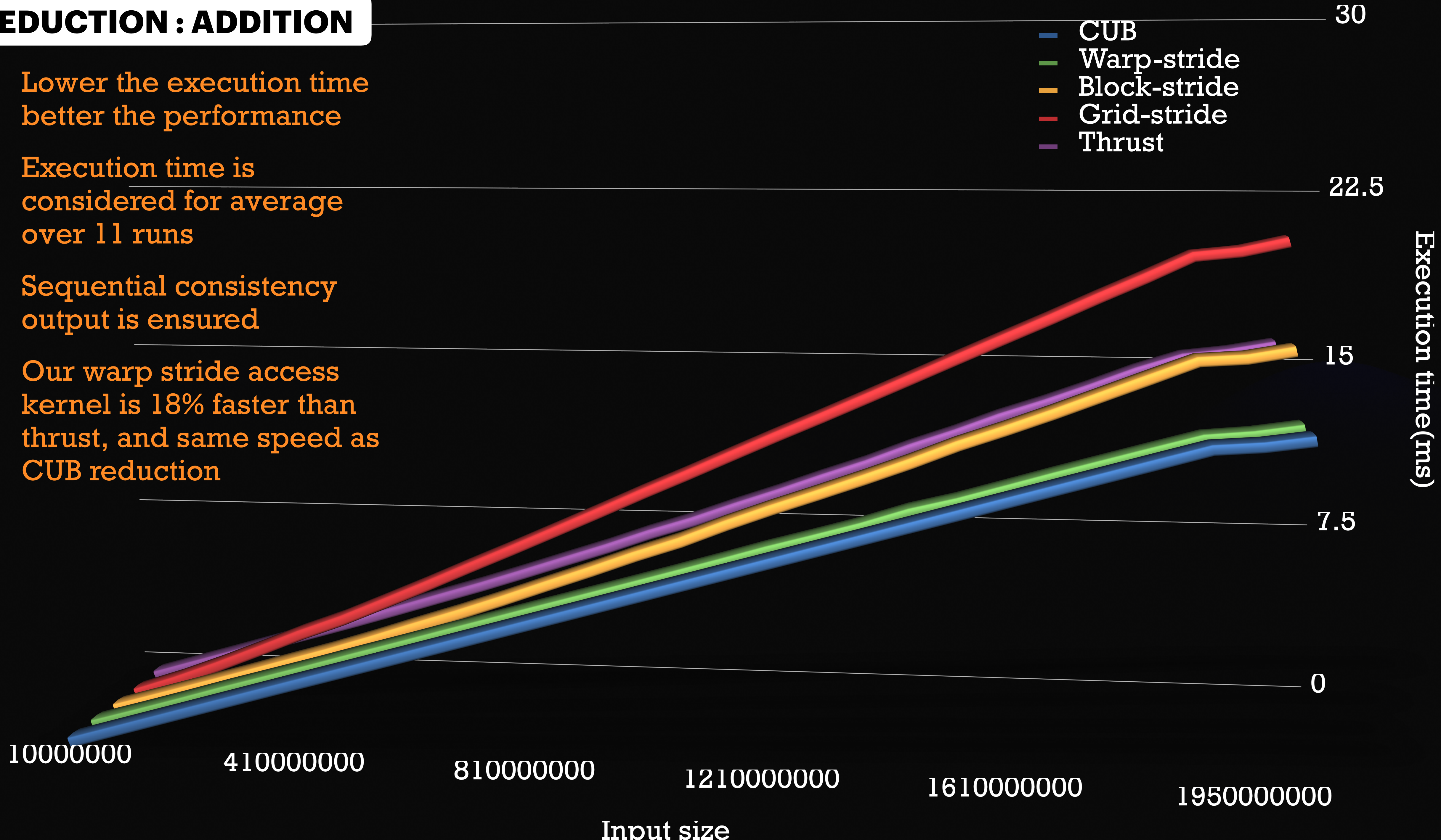
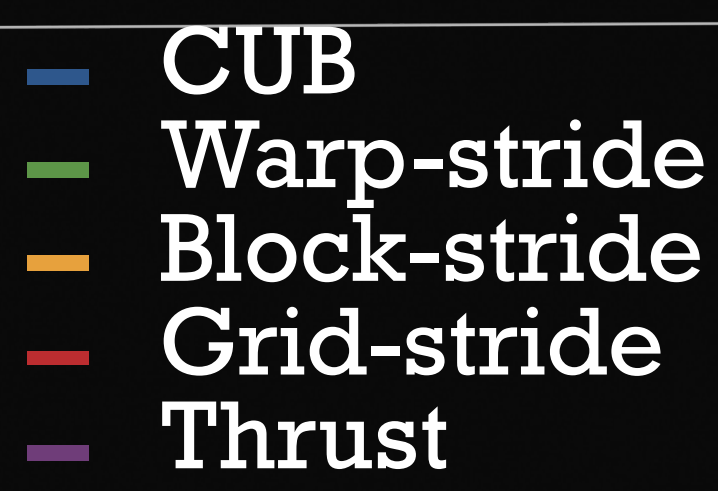
REDUCTION : ADDITION

- 1) Lower the execution time better the performance
- 2) Execution time is considered for average over 11 runs
- 3) Sequential consistency output is ensured
- 4) For smaller input sizes, warp stride thread algebra is overhead



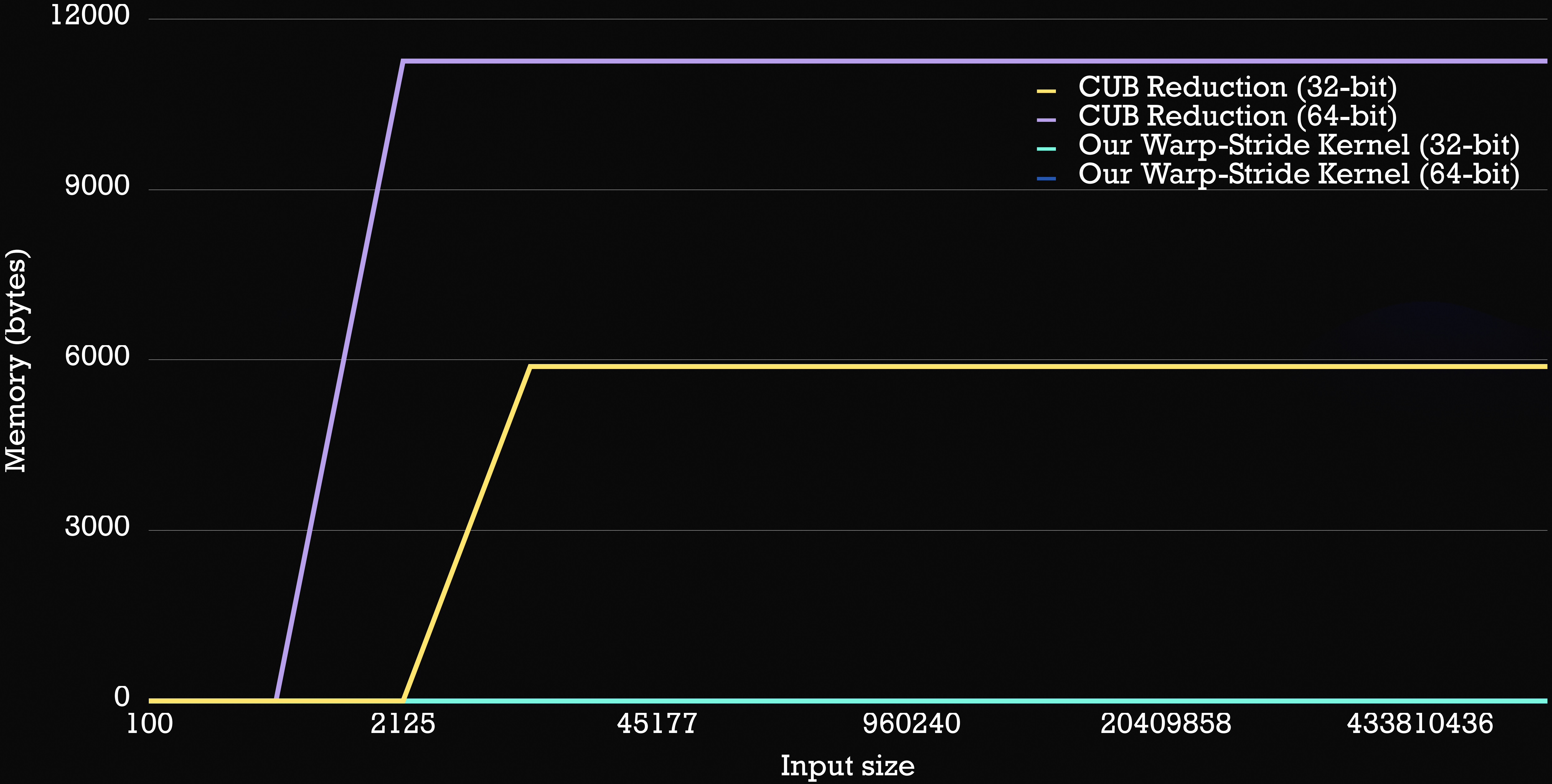
REDUCTION : ADDITION

- 1) Lower the execution time better the performance
- 2) Execution time is considered for average over 11 runs
- 3) Sequential consistency output is ensured
- 4) Our warp stride access kernel is 18% faster than thrust, and same speed as CUB reduction



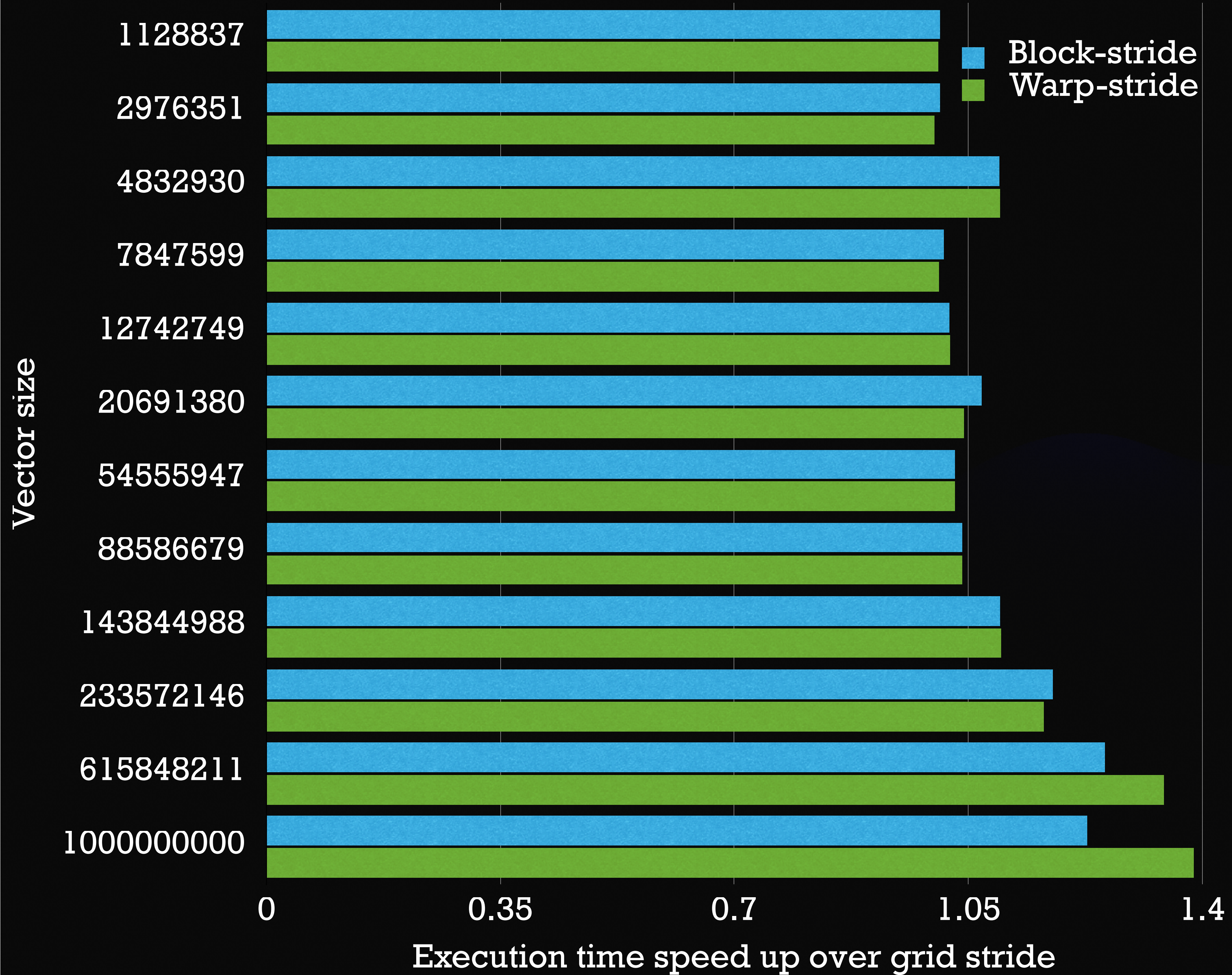
REDUCTION : ADDITION

Temporary storage required (without including output and input space)



EXECUTION TIME SPEED OF WARP AND BLOCK STRIDE OVER GRID STRIDE FOR DOT PRODUCT

- 1) Higher the speedup better the performance
- 2) Speedup of warp-stride or block-stride access is the ratio of their average execution time over 11 runs to that of grid-stride access
- 3) Sequential consistency of kernels output is ensured
- 4) Approximately 40% speed up is achieved for 1 billion vector size input



**EXECUTION TIME SPEEDUP
OF WARP AND BLOCK
STRIDE OVER GRID STRIDE
FOR VARIOUS NORM
DISTANCES**

- 1) Higher the speedup better the performance
- 2) Speedup of warp-stride or block-stride access is the ratio of their average execution time over 11 runs to that of grid-stride access
- 3) Sequential consistency of kernels output is ensured
- 4) 30-40% speed up is achieved for 1 billion vector size input

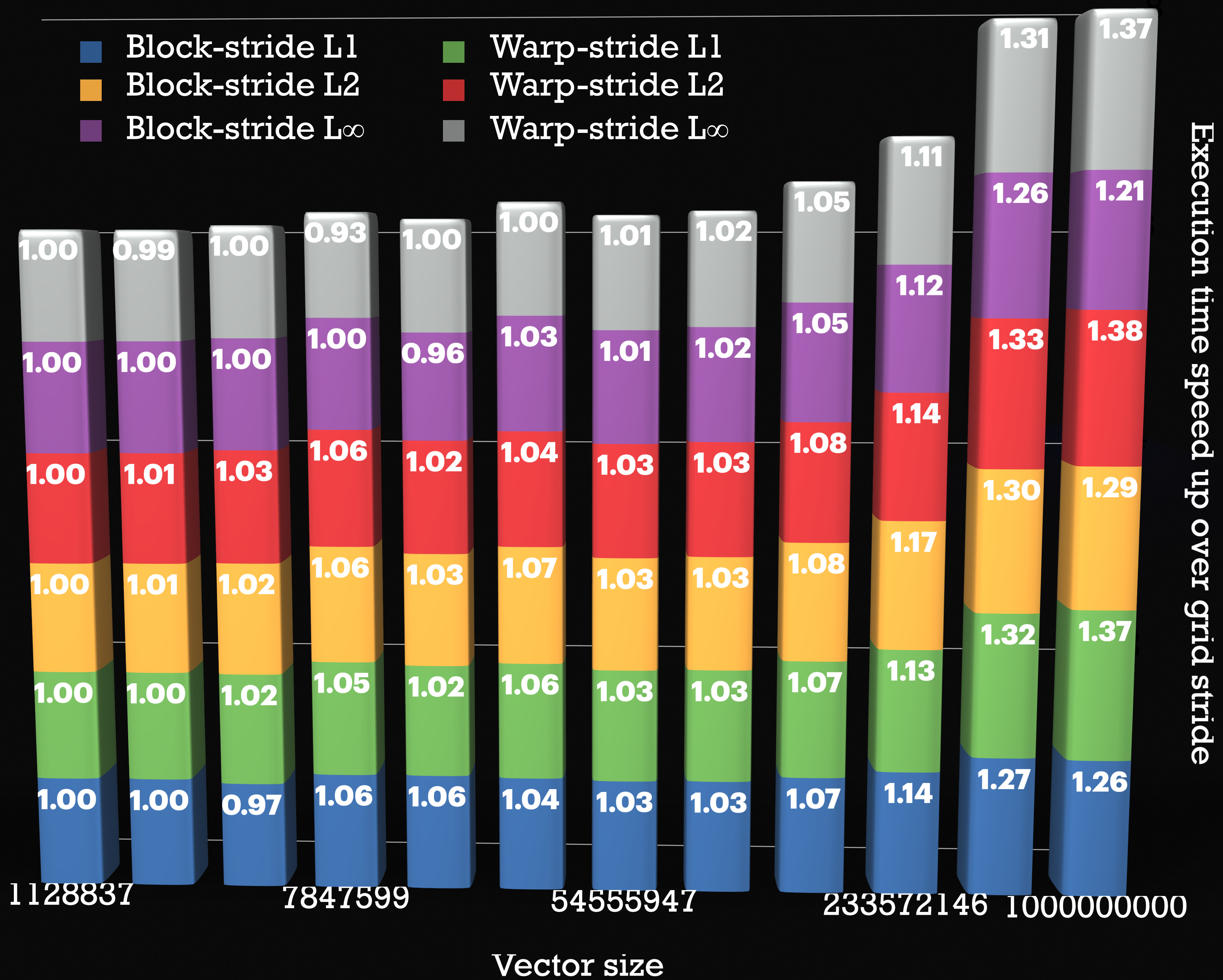
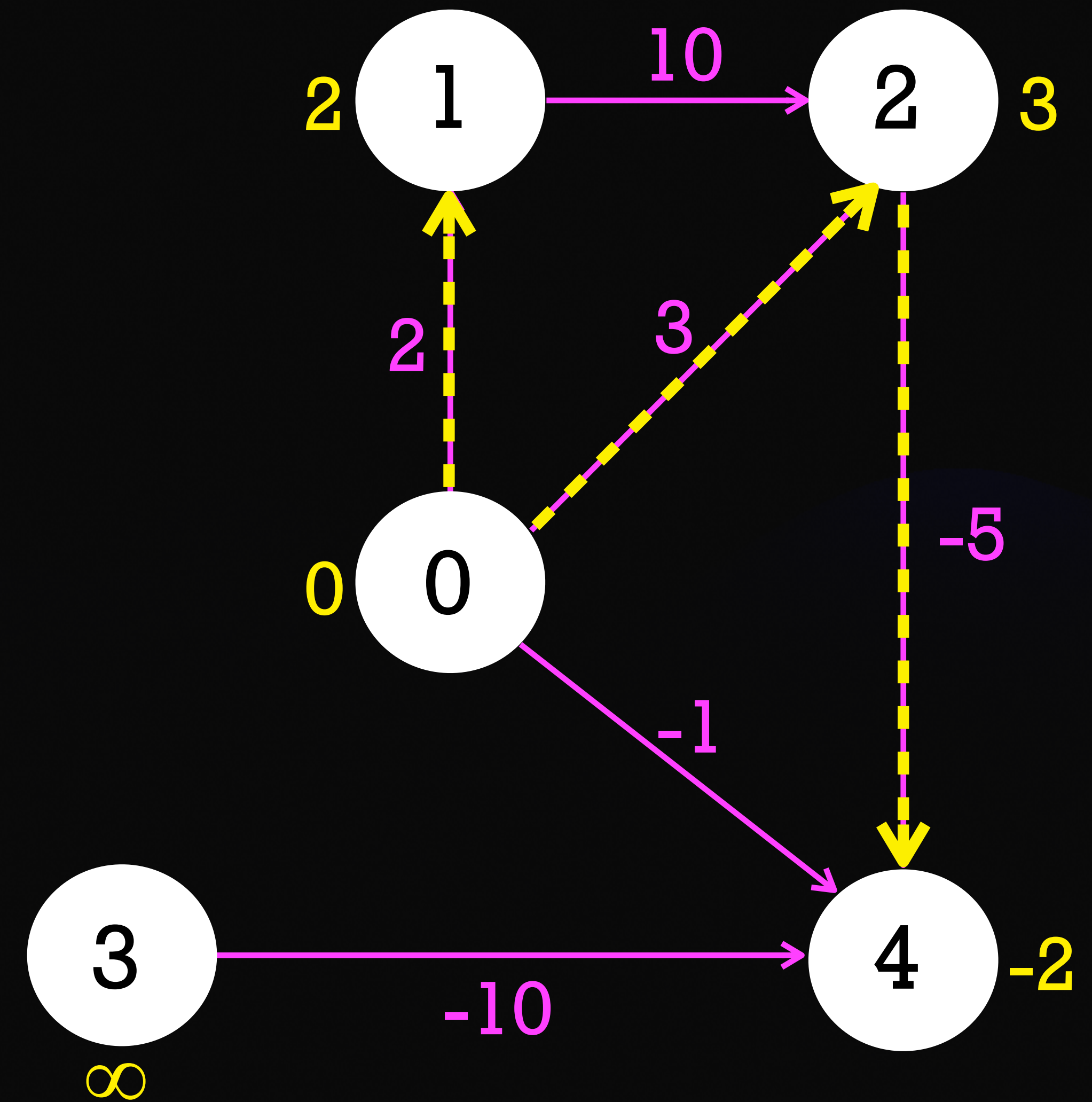


Table 1

Topic	Repository URL	Description
CUDA Efficient Memory Access Patterns	https://github.com/CodeCraftsmanSandeep/CUDA-EfficientMemoryAccessPatterns	Contains the full implementation of CUDA-based memory-efficient access patterns, focusing on optimizing memory throughput and minimizing latency for large-scale computations.

SSSP

- Single Source Shortest Path (SSSP) is a fundamental graph problem that involves finding the shortest path from a given source vertex to all other vertices in a graph.
- We are going to consider weighted directed graphs.
- Well known serial algorithms:
 1. Dijkstra's algorithm
 2. Bellman ford algorithm
- Well know parallel algorithms
 1. Delta stepping algorithm
 2. Parallel bellman ford



Source vertex: 0

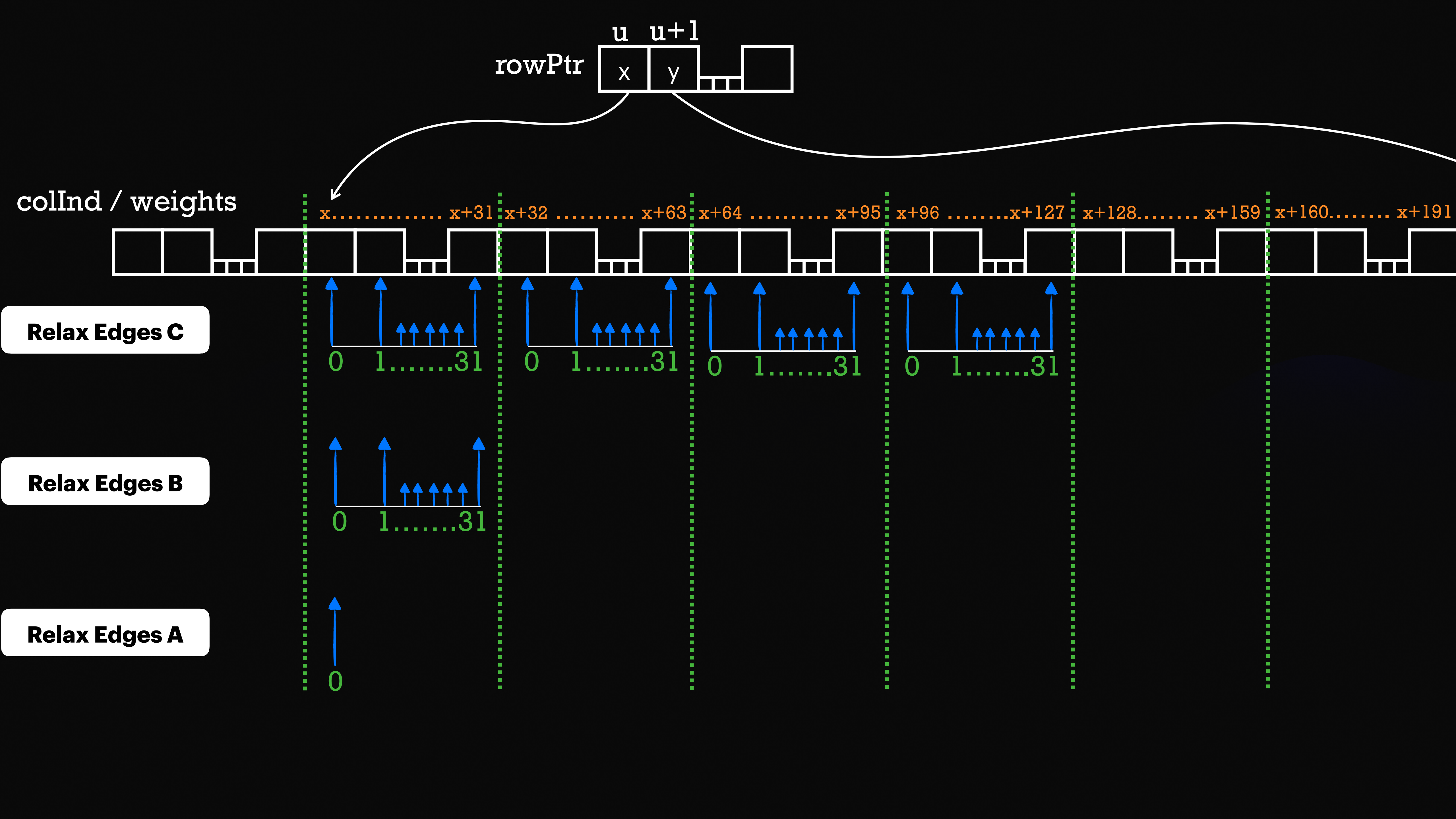
Bellman ford algo:

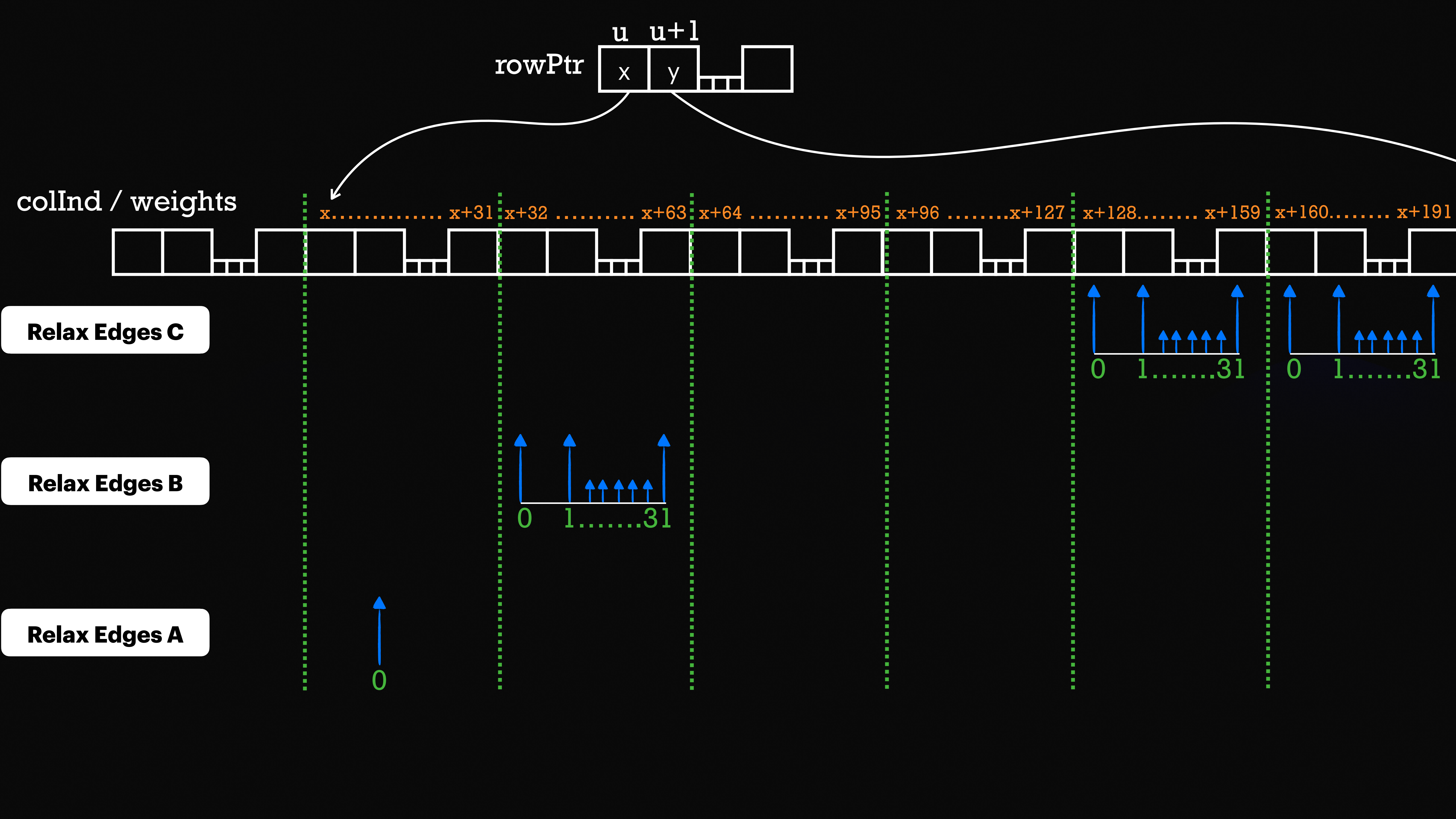
- Input: Directed weighted graph $G(V, E)$ n be the number of vertices and m be the number of edges,
 $V = \{u \in \mathbb{Z} \mid 0 \leq u < n\}$
 $E = \{(u_i, v_i, w_i) \mid i \in \mathbb{Z}, 0 \leq i < m\}$
 (theoretically)
- Relaxing an edge means trying to update the shortest distance to the destination vertex of that edge using the source vertex

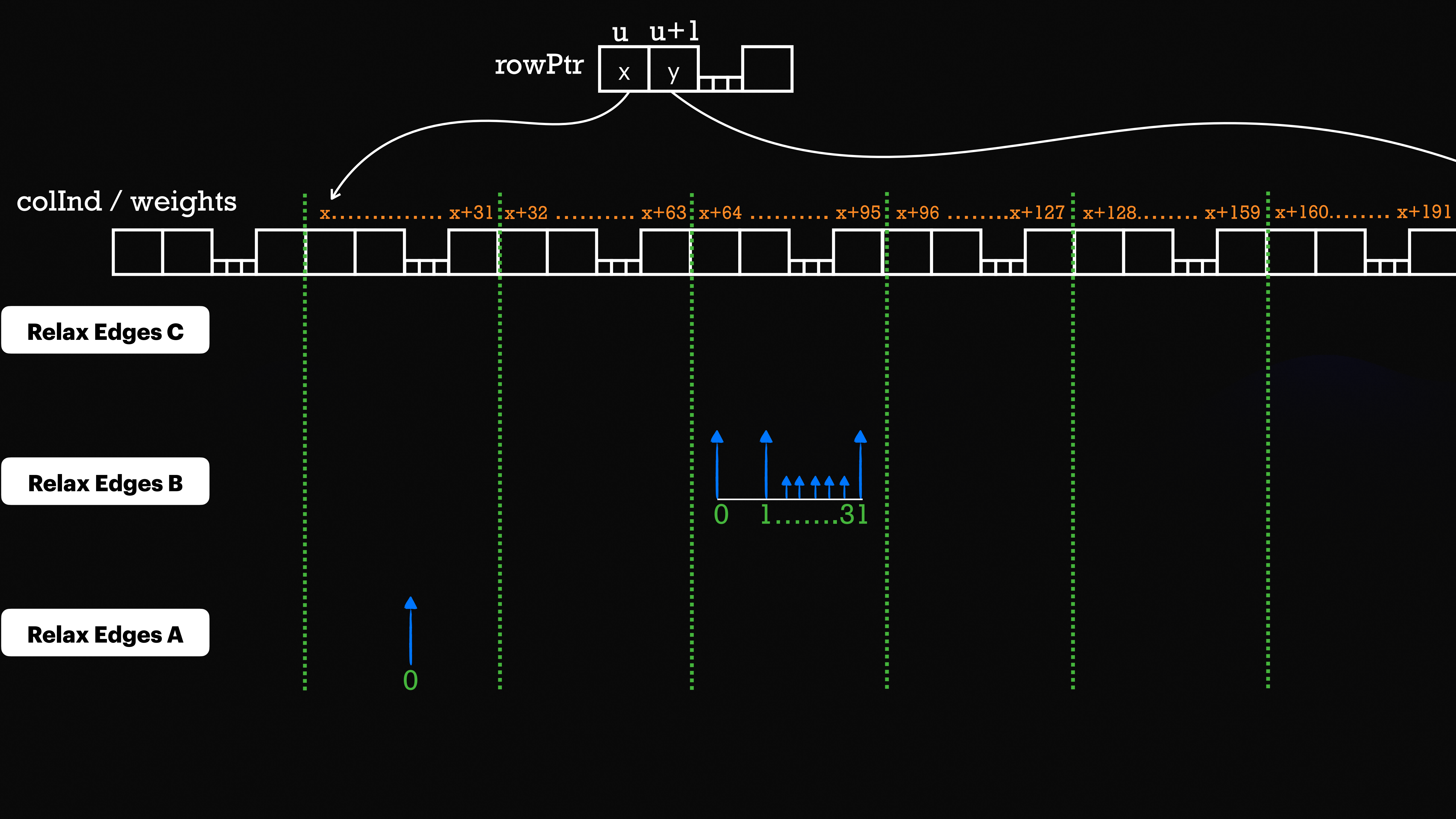
```

1  for(iter = 0; iter < numNodes - 1; iter++)
2  {
3      for(edge = 0; edge < numEdges; edge++)
4      {
5          u = edges[edge].sourceVertex;
6          v = edges[edge].destinationVertex;
7          w = edges[edge].weight;
8          if(dist[v] < dist[u] + w)
9          {
10             dist[v] = dist[u] + w;
11          }
12      }
13  }
14  for(edge = 0; edge < numEdges; edge++)
15  {
16      u = edges[edge].sourceVertex;
17      v = edges[edge].destinationVertex;
18      w = edges[edge].weight;
19      if(dist[v] < dist[u] + w)
20      {
21          negCycle = true;
22      }
23  }

```





GRAPH DATA SETS

Sparse graphs (Road networks)

S.No	Dataset Name	V	E	Description
1	roadNet-CA	19,65,206 (1.9 M)	55,33,214 (5.5 M)	Road network of California from SNAP
2	roadNet-PA	10,88,092 (1 M)	30,83,796 (3 M)	Road network of Pennsylvania from SNAP
3	roadNet-TX	13,79,917 (1.3 M)	38,43,320 (3.8 M)	Road network of Texas from SNAP
4	soc-LiveJournal1	48,47,571 (4.8 M)	6,89,93,773 (68.9 M)	Social network from LiveJournal blogging platform
5	soc-orkut-dir	30,72,441 (3 M)	11,71,85,083 (117.1 M)	Directed social network from Orkut community

Dense graphs (Social networks)

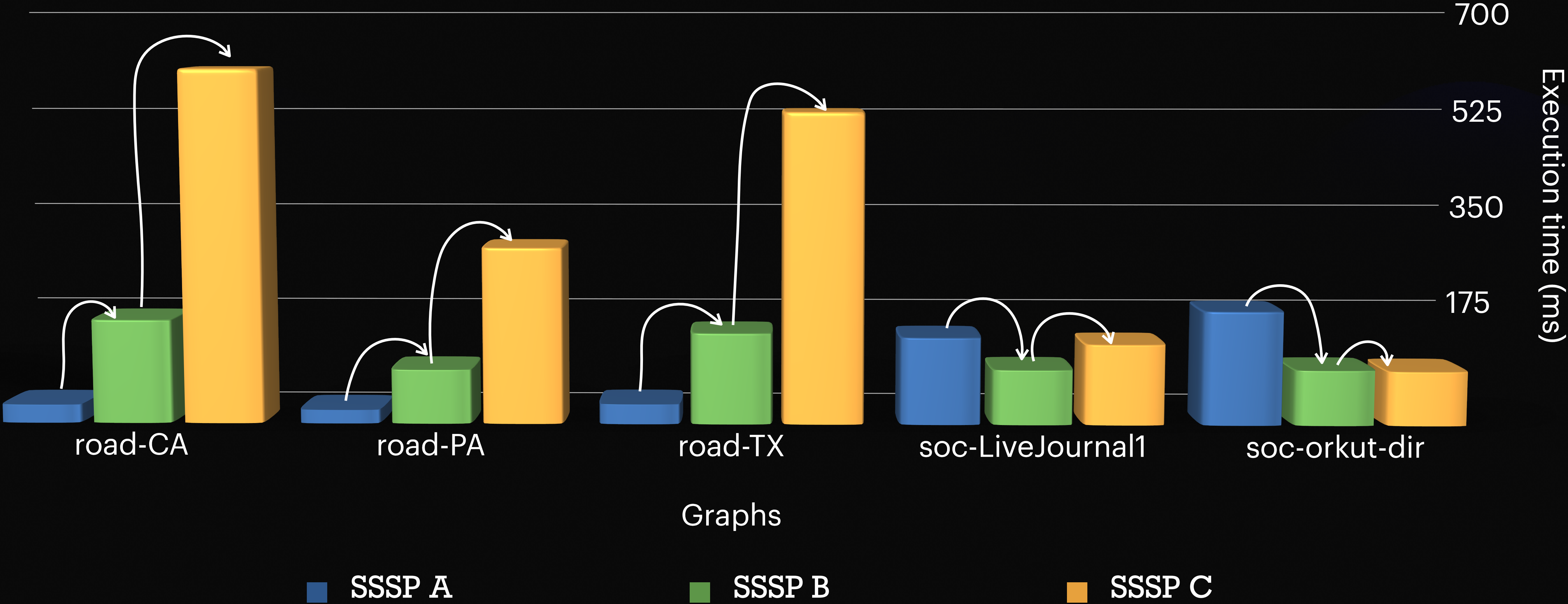
Plot information:

- 1. Smaller the execution time better the performance
- 2. The average time over 11 independent runs is used for comparison

Performance:

For sparse graphs: SSSP A > SSSP B > SSSP C

For dense graphs: SSSP A < SSSP B < SSSP C



ENHANCING GPU PERFORMANCE THROUGH OPTIMIZED LOOP PATTERNS

Benchmarking in the Data Center (BID) - 2026 Singapore

Chekkala Sandeep Reddy

Indian Institute of Technology Palakkad

sandeep.chekkala.wh@gmail.com

Questions?

Lets connect on
LinkedIn?



Up Coming —>

BP-MDS : Million Scale Approximate CVRP in Minutes via Parallel Divide-and-Conquer

01 October 2026 - ICPP 2026 Main Conference

