NVIDIA

ATOMICS, REDUCTIONS, WARP SHUFFLE

Guillaume Barnier, Solutions Architect, Energy
gbarnier@nvidia.com

AGENDA

- Transformations vs. reductions, thread strategy
- Atomics, atomic reductions
- Atomic tips and tricks
- Classical parallel reduction
- Parallel reduction + atomics
- Warp shuffle, reduction with warp shuffle

ATOMICS

MOTIVATING EXAMPLE

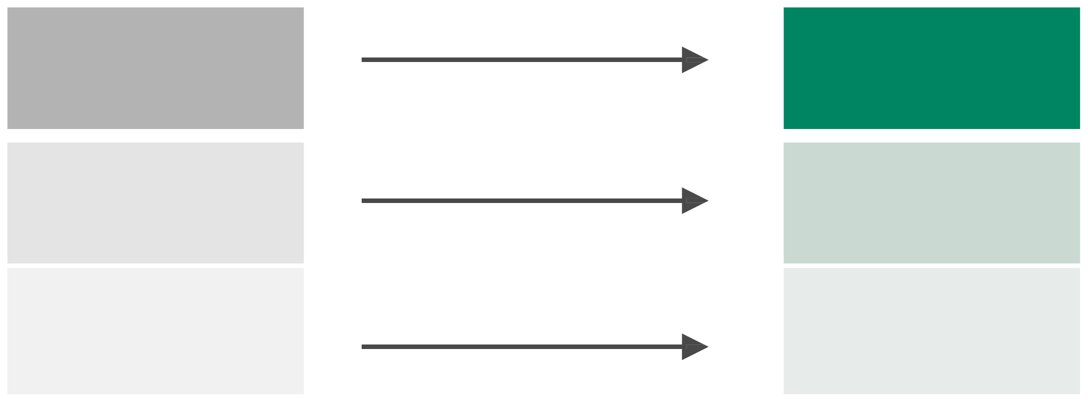
Sum - Reduction

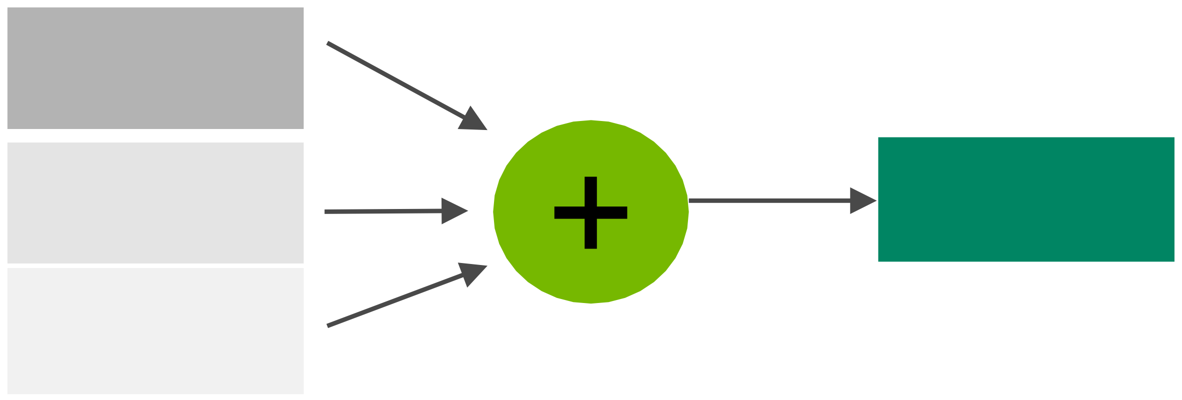
```
const int size = 100000;
float a[size] = {...};
float sum = 0;
for (int i = 0; i < size; i++)
    sum += a[i];
```

Goal: sum variable contains the sum of all the elements of array a

TRANSFORMATION VS. REDUCTION

May guide the thread strategy: what will each thread do?





Transformation:

c[i] = a[i] + 10;

Thread strategy: one thread per output point

Reduction:

$c = \Sigma\, a[i]$

Thread strategy: ?

REDUCTION: NAÏVE THREAD STRATEGY

One thread per input point

Doesn't work

c += a[i];

LD R2, a[i]	(thread independent)
LD R1, c	(READ)
ADD R3, R1, R2	(MODIFY)
ST c, R3	(WRITE)

SASS instructions

- But **every thread** is trying to do this, potentially at the same time
- The CUDA programming model does not enforce any order of thread execution

ATOMIC TO THE RESCUE

"Indivisible"

ATOMIC TO THE RESCUE

Indivisible READ-MODIFY-WRITE

atomicAdd(&c, a[i]);

c += a[i];

LD R2, a[i] (thread independent)

LD R1, c (READ)

ADD R3, R1, R2 (MODIFY)

ST c, R3 (WRITE)

These 3 instructions become one indivisible operation/instruction

RED.E.ADD.F32.FTZ.RN [c], R2;

- Facilitated by special hardware in the L2 cache
- May have performance implications

OTHER ATOMICS

- atomicMax/Min – choose the max (or min)
- atomicAdd/Sub – add to (or subtract from)
- atomicInc/Dec – increment (or decrement) and account for rollover/underflow
- atomicExch/CAS – swap values, or conditionally swap values
- atomicAnd/Or/Xor – bitwise ops
- atomics have different datatypes they can work on (e.g. int, unsigned, float, etc.)
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

ATOMIC TIPS AND TRICKS

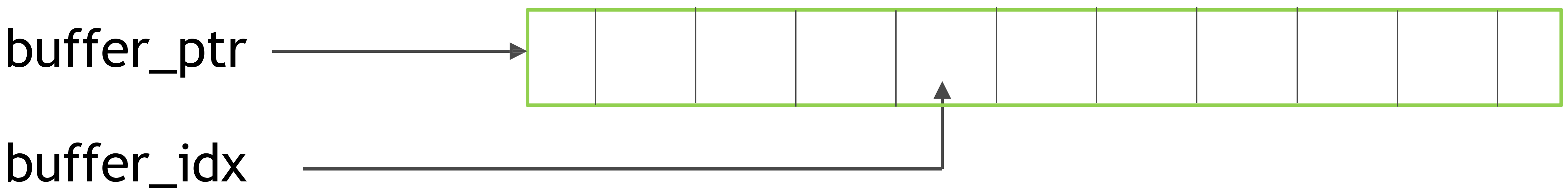
Determine my place in an order

- Could be used to determine next work item, queue slot, etc.
- int my_position = atomicAdd(order, 1);
- Most atomics return a value that is the “old” value that was in the location receiving the atomic update

ATOMIC TIPS AND TRICKS

Reserve space in a buffer

Each thread in my kernel may produce a variable amount of data. How to collect all of this in one buffer, in parallel?



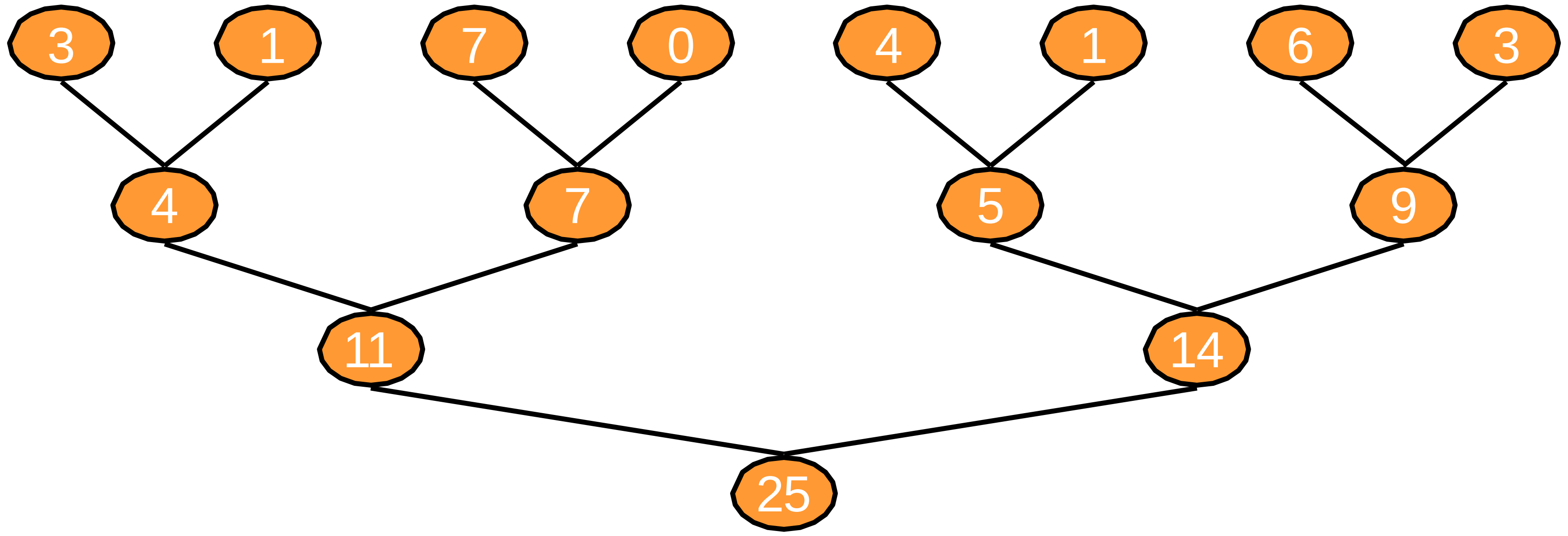

```
int my_length = var; // Length of array

float local_buffer[my_length] = {…}; // Allocate and initialize array

int my_offset = atomicAdd(buffer_idx, my_length); // Increment buffer_idx value, save position of first element

// buffer_ptr + my_offset now points to the first reserved location, of length “my_length”

memcpy(buffer_ptr + my_offset, local_buffer, my_length * sizeof(float));
```

CLASSICAL PARALLEL REDUCTION

THE CLASSICAL PARALLEL REDUCTION

Atomics don’t run at full memory bandwidth

- We would like a reduction method that is not limited by atomic throughput
- We would like to effectively use all threads, as much as possible
- Parallel reduction is a common and important data parallel primitive Naïve
- implementations will often run into bottlenecks
- Basic methodology is a tree-based approach



PROBLEM: GLOBAL SYNCHRONIZATION

If we could synchronize across all thread blocks, could easily reduce very large arrays, right?

- Global sync after each block produces its result
- Once all blocks reach sync, continue recursively

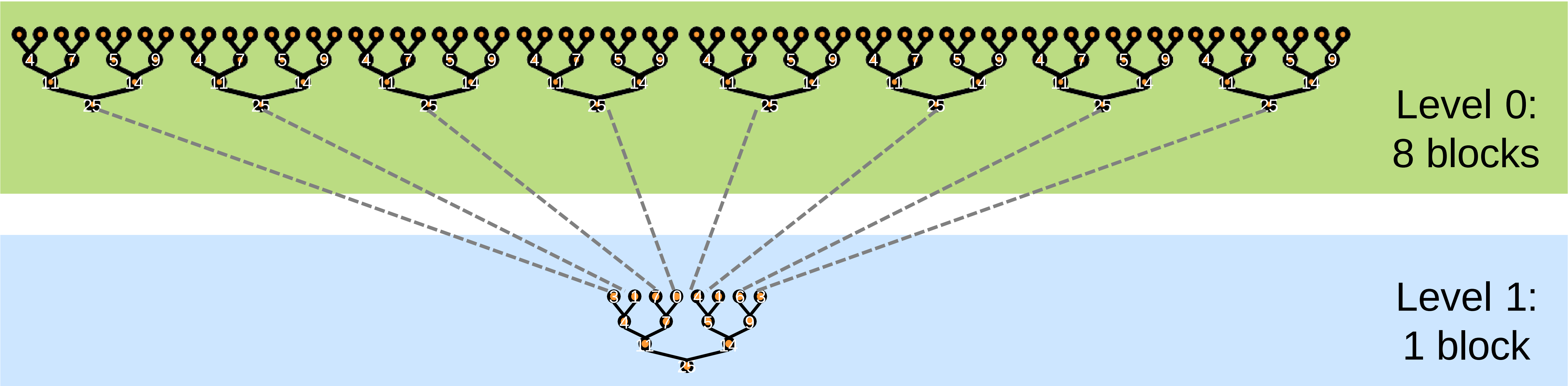
One possible solution: decompose into multiple kernels

- Kernel launch serves as a global synchronization point
- Kernel launch has low overhead (but not zero)

Other possible solutions:

- Use atomics at the end of threadblock-level reduction
- Use a threadblock-draining approach (see threadFenceReduction sample code)
- Use cooperative groups – cooperative kernel launch

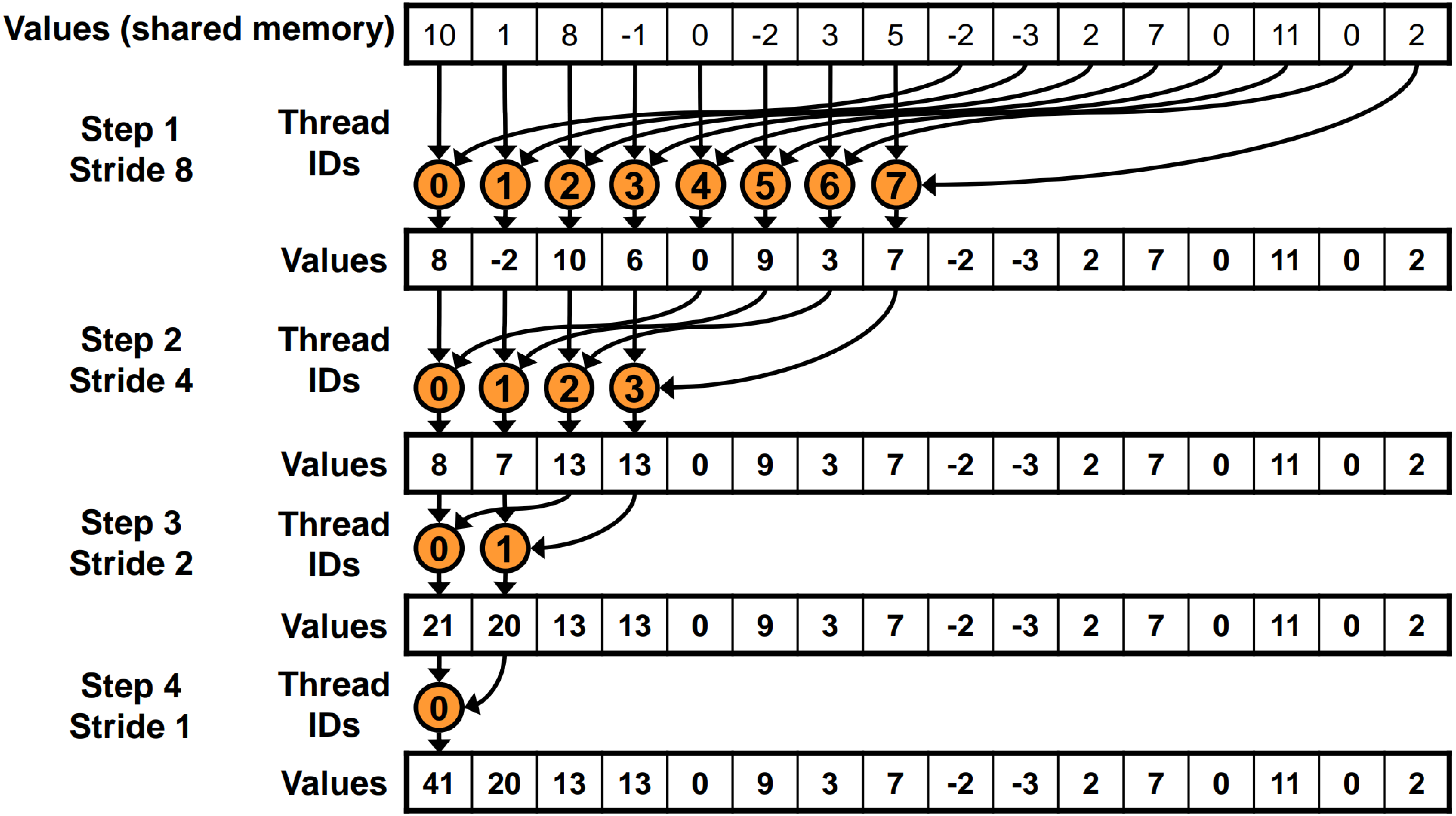
SOLUTION: KERNEL DECOMPOSITION



- Create global sync by decomposing computations into multiple kernel invocations
- In the case of reductions, code for all levels (invocations) is the same

SEQUENTIAL ADDRESSING

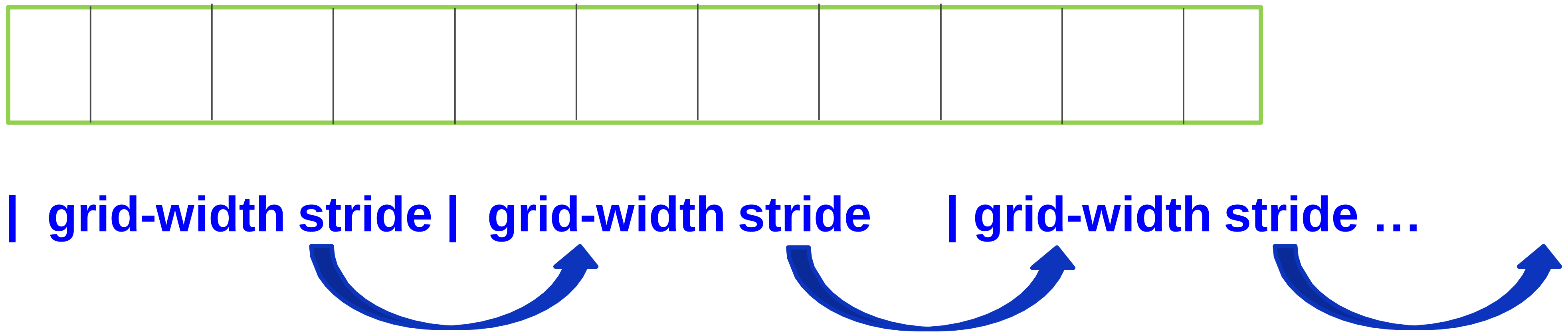
```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s]; }
  __syncthreads();  // outside the if-statement
}
```



Sequential addressing is bank-conflict free

DETOUR: GRID-STRIDE LOOPS

- We’d like to be able to design kernels that load and operate on arbitrary data sizes efficiently
- Want to maintain coalesced loads/stores, efficient use of shared memory
- Can also be used for ninja-level tuning – choose number of blocks sized to the GPU
- **gdata**[0..N-1]:




```
int idx = threadIdx.x+blockDim.x*blockIdx.x;
float sum = 0.f;
while (idx < N) {
   sum += gdata[idx];
   idx += gridDim.x*blockDim.x;  // grid width
   }
   sdata[threadIdx.x] = sum;
```

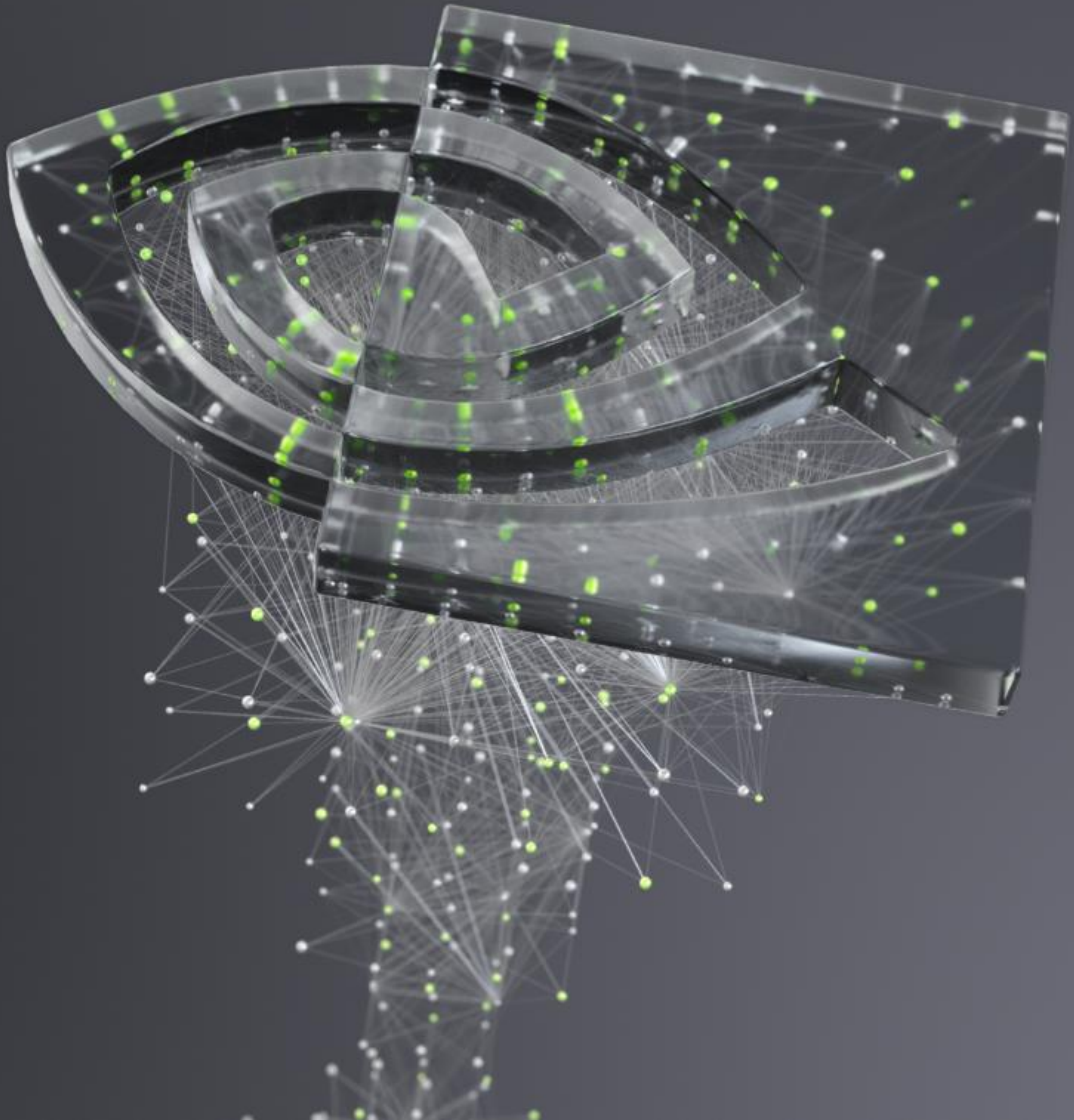
PUTTING IT ALL TOGETHER

```
__global__ void reduce(float *gdata, float *out)
{
    __shared__ float sdata[BLOCK_SIZE];
    int tid = threadIdx.x;
    size_t idx = threadIdx.x + blockDim.x * blockIdx.x;

    // Grid stride loop to load data into shared memory
    float sum = 0.f;
    while (idx < N) {
        sum += gdata[idx];
        idx += gridDim.x * blockDim.x;
        }
     sdata[tid] = sum; // Load temporary sum into share mem

    // Parallel sweep reduction using shared memory
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        __syncthreads();
        if (tid < s) sdata[tid] += sdata[tid + s];
        }

    // Thread 0 writes total sum
    if (tid == 0)
         out[blockIdx.x] = sdata[0];
 }
```

GETTING RID OF 2ND KERNEL LAUNCH

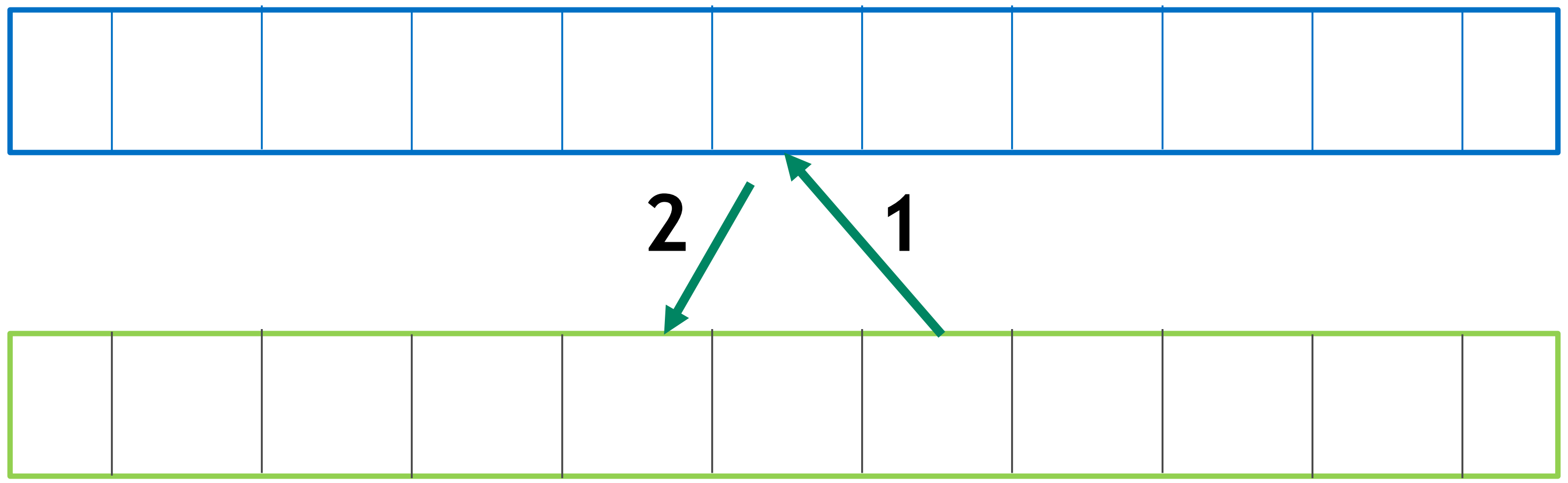
```
__global__ void reduce(float *gdata, float *out)
{
    __shared__ float sdata[BLOCK_SIZE];
    int tid = threadIdx.x;
    size_t idx = threadIdx.x + blockDim.x * blockIdx.x;

    // Grid stride loop to load data into shared memory
    float sum = 0.f;
    while (idx < N) {
       sum += gdata[idx];
       idx += gridDim.x * blockDim.x;
       }
     sdata[tid] = sum; // Load temporary sum into share mem

    // Parallel sweep reduction using shared memory
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
       __syncthreads();
       if (tid < s) sdata[tid] += sdata[tid + s];
       }

    // Thread 0 writes total sum
    if (tid == 0)
        atomicAdd(out, sdata[0]);
 }
```

WARP SHUFFLE



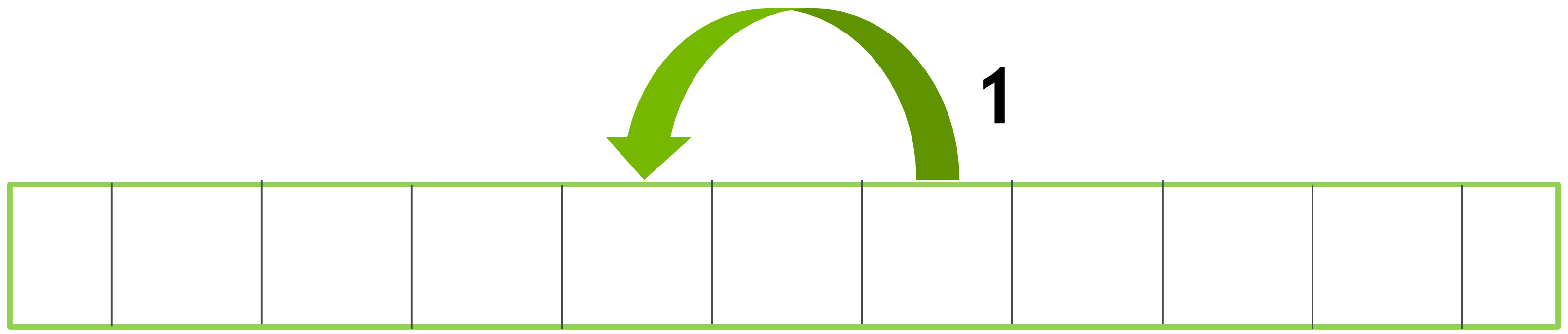
INTER-THREAD COMMUNICATION: SO FAR

- Using **shared memory:**



- **Threads:**

- Wouldn’t this be convenient:

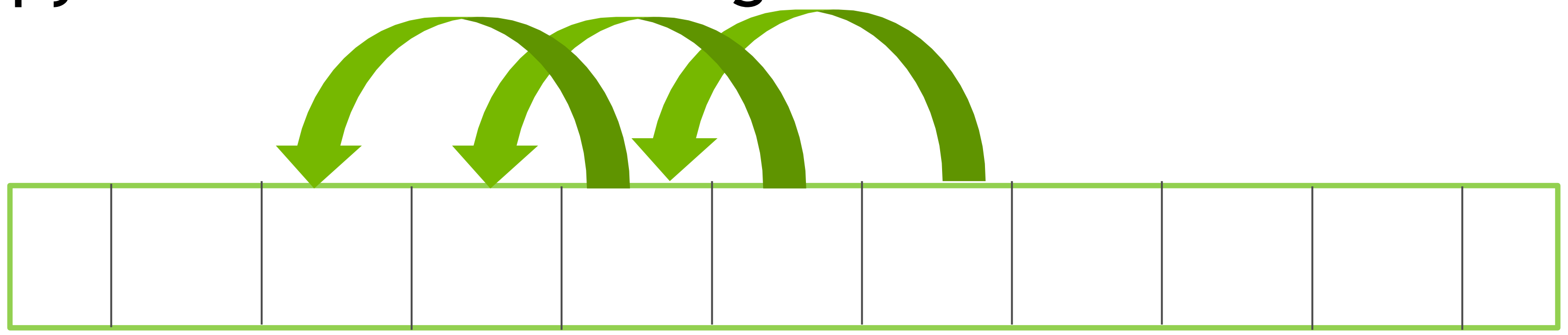


- **Threads:**

NVIDIA.

INTRODUCING WARP SHUFFLE

- Allows for intra-warp communication
- Various supported movement patterns:
  - __shfl_sync(): copy from lane ID (arbitrary pattern)
  - __shfl_xor_sync(): copy from calculated lane ID (calculated pattern)
  - __shfl_up_sync(): copy from delta/offset lower lane
  - __shfl_down_sync(): copy from delta/offset higher lane:



- Both source and destination threads in the warp must “participate”
- Sync “mask” used to identify and reconverge needed threads

```
__global__ void reduce (float *gdata, float *out)
{
  __shared__ float sdata[32]; // Allocate shared memory
  int tid = threadIdx.x;
  sdata[tid] = 0.f;
  size_t idx = threadIdx.x + blockDim.x * blockIdx.x;
  float val = 0.f;
  unsigned int mask = ~0;
  int lane = threadIdx.x % warpSize;
  int warpID = threadIdx.x / warpSize;

  // Each thread computes sums over grid stride loops
  while (idx < N) {
    val += gdata[idx];
    idx += gridDim.x*blockDim.x;
  }

  // First warp-shuffle
  for (int offset = warpSize/2; offset > 0; offset >>= 1)
        val += __shfl_down_sync(mask, val, offset);

  // Only thread with lane = 0 write their partial sum to shared memory
  if (lane == 0)
        sdata[warpID] = val;
   __syncthreads();

   // Only warp 0 is working
   if (warpID == 0) {
     val = (tid < blockDim.x / warpSize) ? sdata[lane] : 0.f;

     // Second warp-shuffle
     for (int offset = warpSize/2; offset > 0; offset >>= 1)
         val += __shfl_down_sync(mask, val, offset);

     // Only thread with lane = 0 write their partial sum to shared memory
     if (threadIdx.x == 0)
         atomicAdd(out, val);
   }

} // End kernel
```

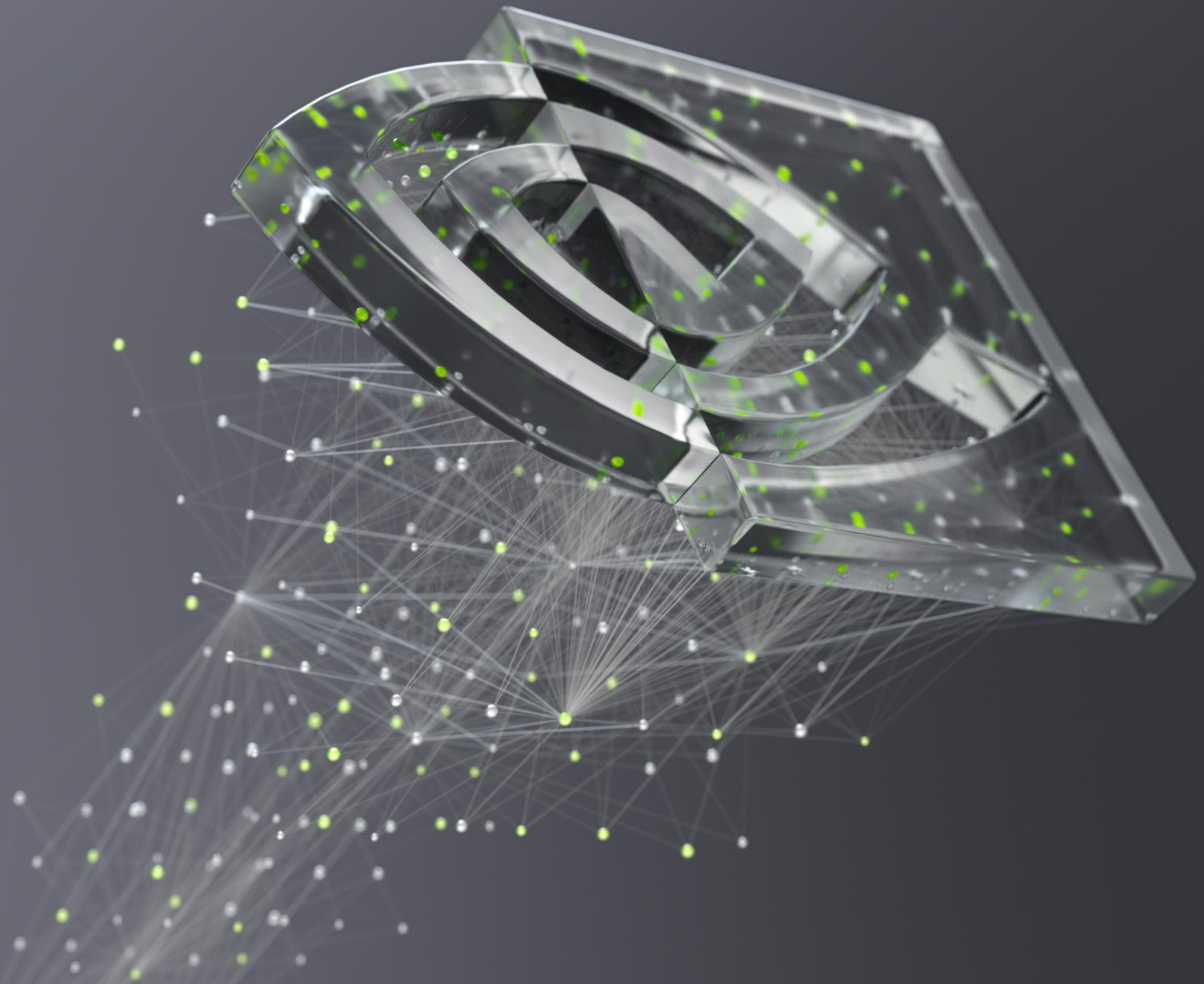
WARP SHUFFLE BENEFIT

- Reduce or eliminate shared memory usage
- Single instruction vs. 2 or more instructions
- Reduce level of explicit synchronization

WARP SHUFFLE TIPS AND TRICKS

What else can we do with it?

- Broadcast a value to all threads in the warp in a single instruction
- Perform a warp-level prefix sum
- Atomic aggregation



NVIDIA

MANAGED MEMORY

- Good for prototyping or for porting applications quickly
- Page fault at the page level
- Managed memory -> you rely on page fault to move the data
- Page fault are expensive, and you only move one page
- One load/pagefault, then next piece of data, page fault, etc.
- The code may be slow if you rely a lot on these page faults
- We have functions such as prefetch to help reducing page faults
- Bigger pages -> fewer page faults
- Recommendation on x86: cudaMalloc and explicitly copy data