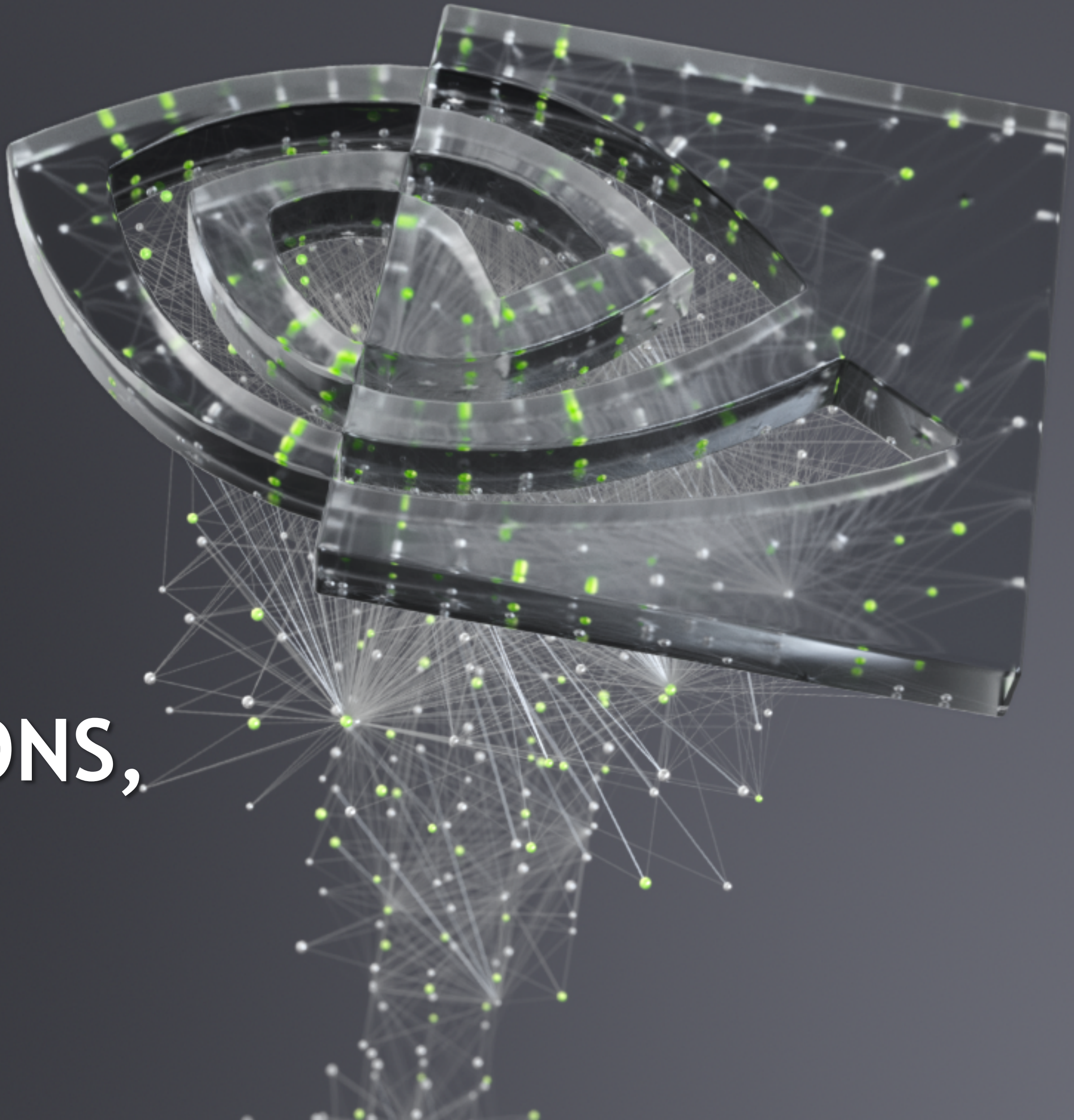
ATOMICS, REDUCTIONS, WARP SHUFFLE

Bob Crovella, 5/13/2020



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
- Other Warp Shuffle Uses
- Further Study
- Homework

ATOMICS

MOTIVATING EXAMPLE

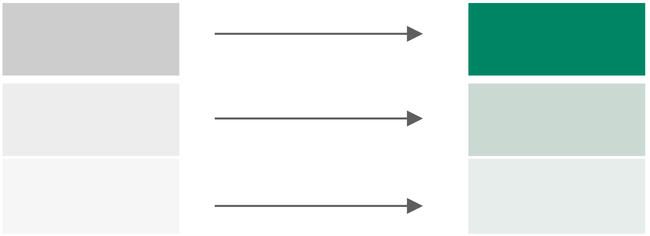
Sum - reduction

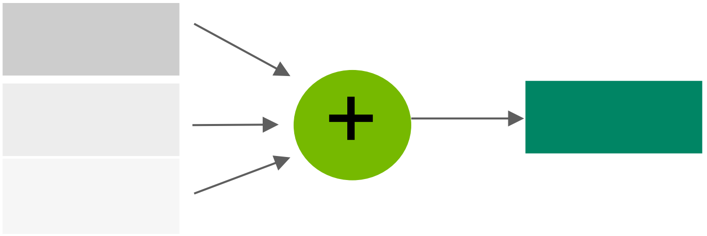
```
const int size = 100000;
float a[size] = {…};
float sum = 0;
for (int i = 0; i < size; i++) sum += a[i];
```

-> 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:

e.g. c[i] = a[i] + 10;

Thread strategy: one thread per output point

Reduction:

e.g. *c = Σ a[i]

Thread strategy: ??

REDUCTION: NAÏVE THREAD STRATEGY

One thread per input point

```
*c += a[i];
```

(Doesn’t work.) Actual code the GPU executes:

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

But **every thread** is trying to do this, potentially at the same time

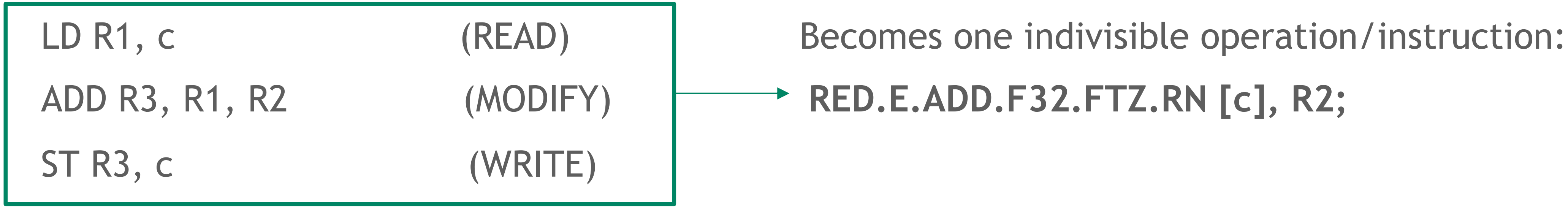
The CUDA programming model does not enforce any order of thread execution

ATOMICS TO THE RESCUE

indivisible READ-MODIFY-WRITE

atomicAdd(&c, a[i]); https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

LD R2, a[i] (thread independent)



LD R1, c (READ)

ADD R3, R1, R2 (MODIFY)

ST R3, c (WRITE)

Becomes 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

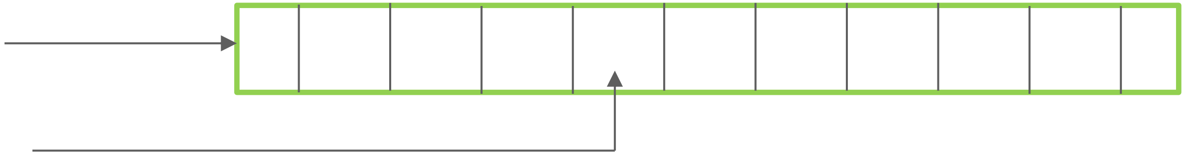
Determine my place in an order

- 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?
- buffer_ptr:
- buffer_idx



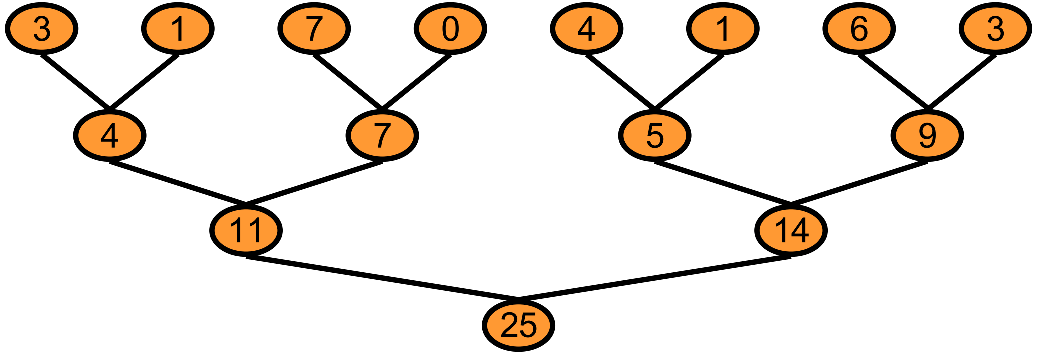
- int my_dsize = var;
- float local_buffer[my_dsize] = {…};
- int my_offset = atomicAdd(buffer_idx, my_dsize);
- // buffer_ptr+my_offset now points to the first reserved location, of length my_dsize
- memcpy(buffer_ptr+my_offset, local_buffer, my_dsize*sizeof(float));

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:

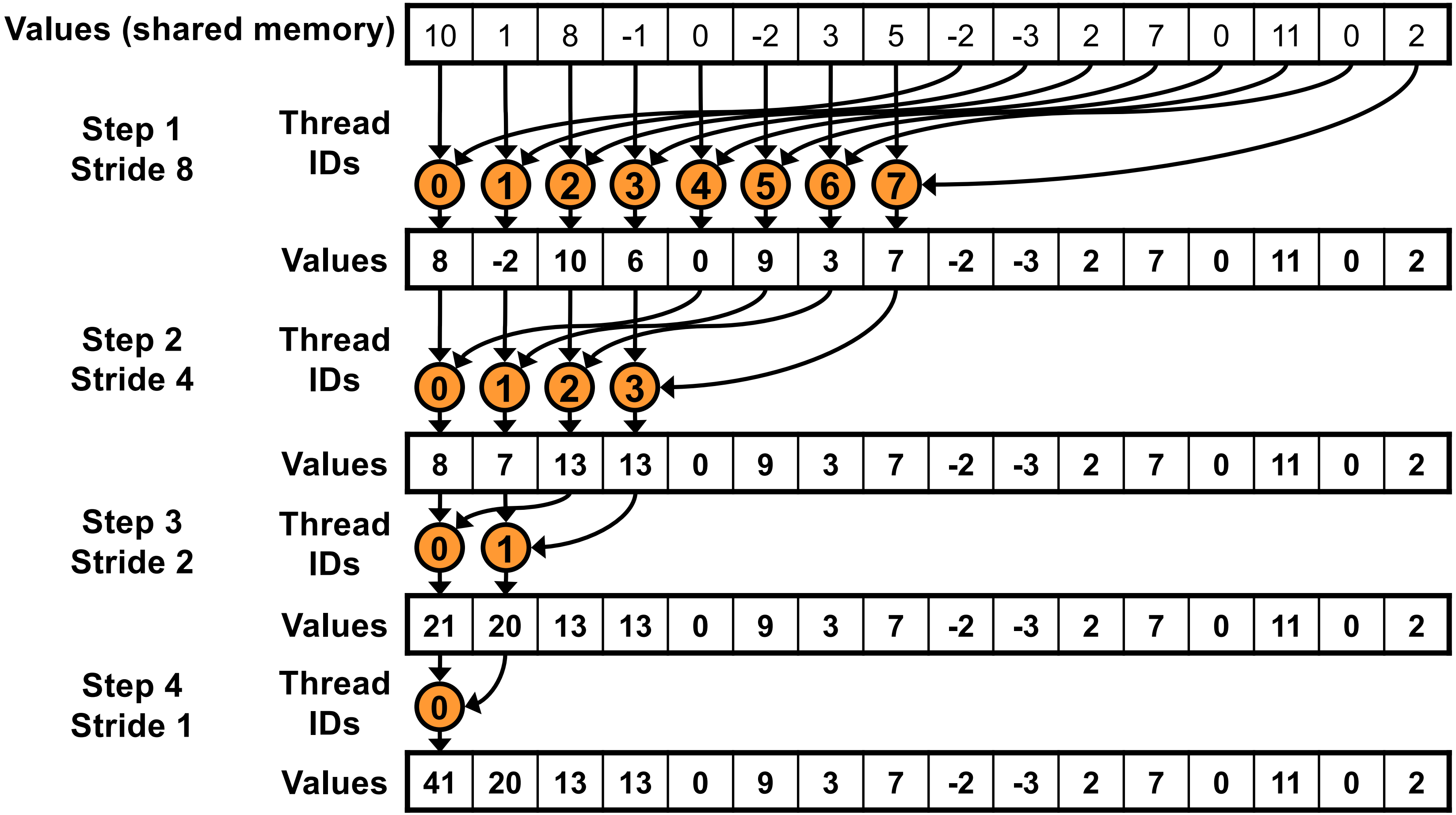


PROBLEM: GLOBAL SYNCHRONIZATION

- 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 SW 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

```
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

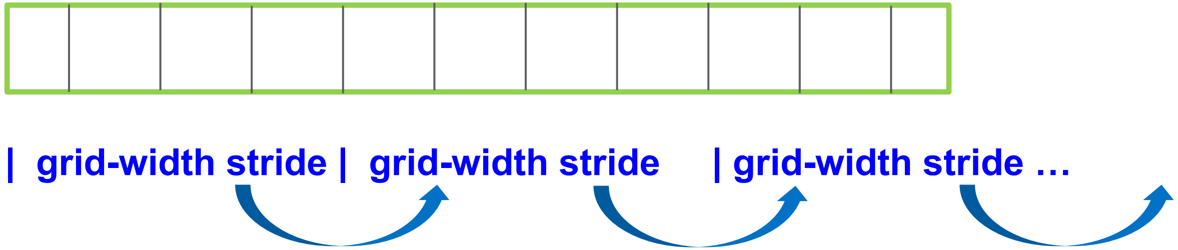


	Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 8	Thread IDs	0	1	2	3	4	5	6	7								
	Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 2 Stride 4	Thread IDs	0	1	2	3												
	Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 3 Stride 2	Thread IDs	0	1														
	Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 4 Stride 1	Thread IDs	0															
	Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

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]:


| grid-width stride | grid-width stride | grid-width stride …

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

PUTTING IT ALL TOGETHER

```
__global__ void reduce(float *gdata, float *out){
   __shared__ float sdata[BLOCK_SIZE];
   int tid = threadIdx.x;
   sdata[tid] = 0.0f;
   size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

   while (idx < N) {  // grid stride loop to load data
      sdata[tid] += gdata[idx];
      idx += gridDim.x*blockDim.x;
      }

   for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
      __syncthreads();
      if (tid < s)  // parallel sweep reduction
          sdata[tid] += sdata[tid + s];
      }
   if (tid == 0) out[blockIdx.x] = sdata[0];
}
```

GETTING RID OF THE 2ND KERNEL CALL

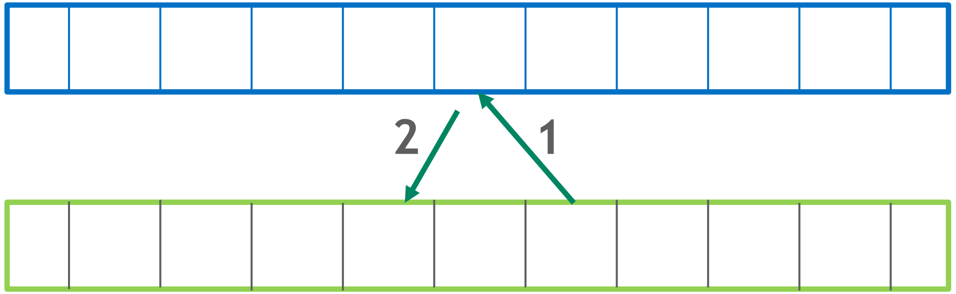
```
__global__ void reduce_a(float *gdata, float *out){
    __shared__ float sdata[BLOCK_SIZE];
    int tid = threadIdx.x;
    sdata[tid] = 0.0f;
    size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

    while (idx < N) {  // grid stride loop to load data
       sdata[tid] += gdata[idx];
       idx += gridDim.x*blockDim.x;
       }

    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
       __syncthreads();
       if (tid < s)  // parallel sweep reduction
           sdata[tid] += sdata[tid + s];
       }
    if (tid == 0) atomicAdd(out, sdata[0]);
}
```

WARP SHUFFLE

INTER-THREAD COMMUNICATION: SO FAR

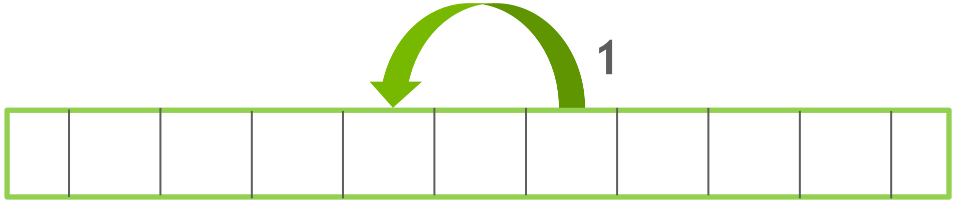
- Using **shared memory:**

- **Threads:**



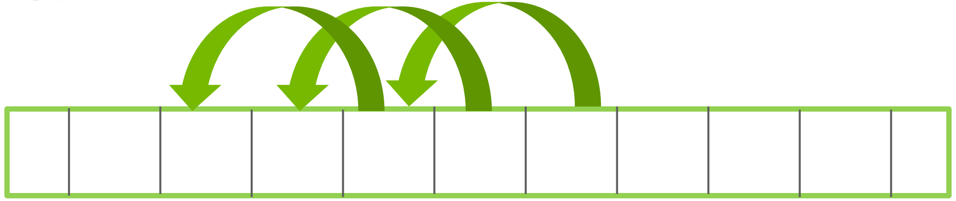
- Wouldn’t this be convenient:

- **Threads:**



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

WARP SHUFFLE REDUCTION

```
__global__ void reduce_ws(float *gdata, float *out){
   __shared__ float sdata[32];
   int tid = threadIdx.x;
   int idx = threadIdx.x+blockDim.x*blockIdx.x;
   float val = 0.0f;
   unsigned mask = 0xFFFFFFFFU;
   int lane = threadIdx.x % warpSize;
   int warpID = threadIdx.x / warpSize;
   while (idx < N) {  // grid stride loop to load
     val += gdata[idx];
     idx += gridDim.x*blockDim.x;
     }

// 1st warp-shuffle reduction
  for (int offset = warpSize/2; offset > 0; offset >>= 1)
    val += __shfl_down_sync(mask, val, offset);
  if (lane == 0) sdata[warpID] = val;
 __syncthreads(); // put warp results in shared mem
```

```
// hereafter, just warp 0
   if (warpID == 0){
 // reload val from shared mem if warp existed
     val = (tid < blockDim.x/warpSize)?sdata[lane]:0;

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

     if  (tid == 0) atomicAdd(out, val);
   }
 }
```

WARP SHUFFLE BENEFITS

- 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

FUTURE SESSIONS

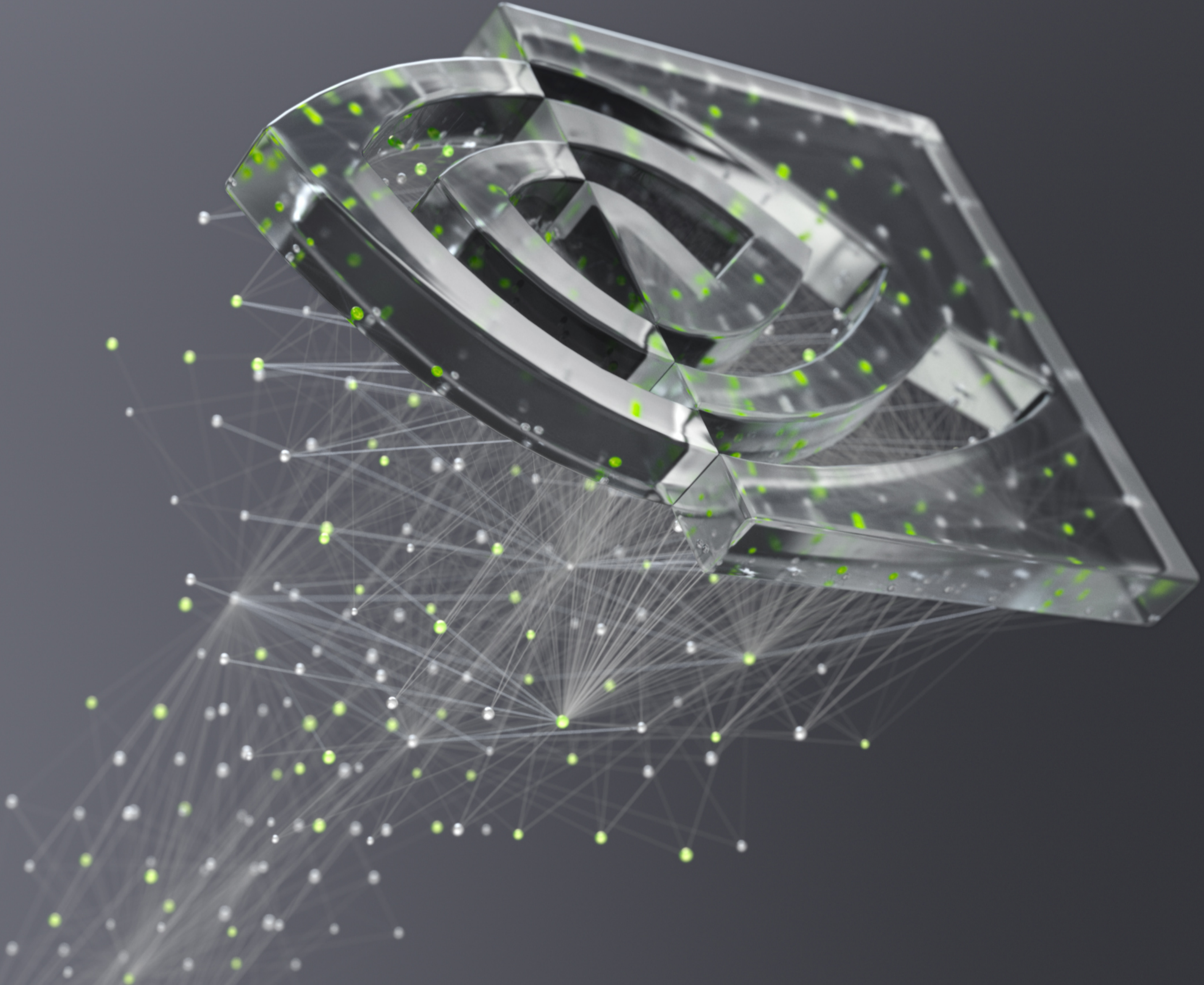
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- Parallel reduction:
  - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- Warp-shuffle and reduction:
  - https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
- CUDA Cooperative Groups:
  - https://devblogs.nvidia.com/cooperative-groups/
- Grid-stride loops:
  - https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
- Floating point:
  - https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
- CUDA Sample Codes:
  - Reduction, threadFenceReduction, reductionMultiBlockCG

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw5/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA