GPUs and CUDA 3 Intrinsics

CS121 Parallel Computing

Fall 2021

Race conditions

- Race condition: Different outcomes depending on execution order.
- Race conditions can occur in any concurrent system, including GPUs.
  - Code should print a =1,000,000.
  - But actually printed a = 88.
- GPUs have atomic intrinsics for simple atomic operations.
- Hard to implement general mutex in CUDA.
  - Codes using critical sections don't perform well on GPUs anyway.

```
__global__ void raceKernel(int *d_a) {
   *d_a += 1;
}

void main() {
   int a = 0; *d_a;

   cudaMalloc((void **) &d_a,
   sizeof(int));
   cudaMemcpy(d_a, &a, sizeof(int),
     cudaMemcpyHostToDevice);

   raceKernel<<<1000, 1000>>>(d_a);

   cudaMemcpy(&a, d_a, sizeof(int),
     cudaMemcpyDeviceToHost);
   printf("a = &d\n", a);
   cudaFree(d_a);
}
```

CUDA atomics

- Can perform atomics on global or shared memory variables.
- `int atomicInc(int *addr)`
  - Reads value at addr, increments it, returns old value.
  - Hardware ensures all 3 instructions happen without interruption from any other thread.
- `int atomicAdd(int *addr, int val)`
  - Reads value at addr, adds val to it, returns old value.
- `int atomicMax(int *addr, int val)`
  - Reads value at addr, sets it to max of current value and val, returns old value.
- `int atomicExch(int *addr1, int val)`
  - Sets val at addr to val, returns old value at val.
- `int atomicCAS(int *addr, old, new)`
  - “Compare and swap”, a conditional atomic.
  - Reads value at addr. If value equals old, sets value to new. Else does nothing.
  - Indicates whether state changed, i.e. if your view is up to date.
  - Universal operation, i.e. can be used to perform any other kind of synchronization.

Finding max of array

- A grid of threads computes the max of an array in global memory.

```
__global__ void max(int *vals, int* global_max) {
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   int val = vals[i];
   atomicMax(global_max, val);
}
```

- Works correctly, no race conditions.
- Ex Thread 1 tries to set global_max to 4, thread 2 tries to set it to 5.
  - If thread 1 goes, then thread 2, global_max becomes 4 then 5.
  - If thread 2 goes, then thread 1, global_max becomes 5, stays 5.

Atomics performance

```
__global__ void max(int *vals, int* global_max) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = vals[i];
    atomicMax(&global_max, val);
}
```

- Problem is this code is really slow.
- When multiple threads perform atomic operation on same variable, they get serialized.
  - Many threads may update `global_max` at same time. Execution becomes sequential instead of parallel.
  - Even with no contention, atomic operation somewhat slower than regular operation.

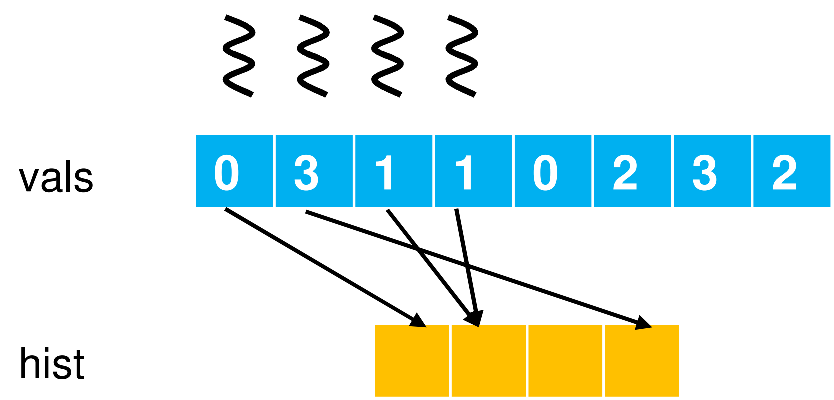
Improving performance

```
__global__ void max(int *vals, int* global_max, int
    *local_max, int num_locals) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = vals[i];
    int locali = i % num_locals;
    int old_max = atomicMax(&local_max[locali], val);
    if (old_max < val) {
        atomicMax(&global_max, val);
    }
}
```

- Split the single global max into `num_locals` number of local max values.
- Thread i atomically maxes with its local max, `local_max[locali]`.
- Only if it increased the local max does thread try to increase the global max.
  - Local max usually won't increase, so rarely need to update global max.
- Spread out contention to the local maxes, reduce frequency of updates to global max.
- Still doesn't perform that well. A reduction tree is more efficient.

Computing a histogram

- Given an array of values from 0 to k, count the number of each value.
- Ex [0,0,1,0,1] $\Rightarrow$ [3,2], i.e. three 0's and two 2's.




```
__global__ void hist(unsigned char *vals, int size, int *hist) {
    int i = threadIdx.x+ blockIdx.x* blockDim.x;
    int stride = blockDim.x* gridDim.x;
    while (i< size) {
        atomicAdd(&(hist[vals[i]]), 1);
        i += stride;
}}
```

Improving performance

- Hist array stored in global memory, so access is slow.
- Since all threads write to same array, contention can be high.
- Better to make local histogram in shared memory for each thread block, add result to global histogram at the end.
  - Atomic ops on shared memory faster.
  - Fewer threads per block so less contention on local_hist.
  - Only one atomic op on global histogram per value instead of many increments.

```
__global__ void hist(unsigned char *vals,
     int size, int *hist) {

// Assume 256 different vals
__shared__ int local_hist[256];
if (threadIdx.x < 256)
     local_hist[threadIdx.x] = 0;
__syncthreads();

int i = threadIdx.x+ blockIdx.x*
     blockDim.x;
int stride = blockDim.x* gridDim.x;
while (i< size) {
     atomicAdd(&(local_hist[vals[i]]), 1);
     i += stride;
}
__syncthreads();
if (threadIdx.x < 256)
     atomicAdd(&(hist[threadIdx.x]),
     local_hist[threadIdx.x]);
}
```

Aggregation and atomics

- Want threads to aggregate data into a shared structure.
- Ex Filtering. Given a source array `src` with n elements, and a predicate, copy all elements of `src` satisfying predicate into destination `dst`.
  - In our examples, we copy all positive values.
- Consider four methods
  - Global memory atomicAdd.
  - Shared memory atomicAdd.
  - Scan based filtering using Thrust's `copy_if()`.
  - Warp-aggregated filtering.

```
int filter(int *dst, const int *src,
     int n) {
  int nres = 0;
  for (int i = 0; i < n; i++)
    if (src[i] > 0)
      dst[nres++] = src[i];
  return nres; }
```

Sequential

```
__global__
void filter_k(int *dst, int *nres,
     const int *src, int n) {
  int i = threadIdx.x + blockIdx.x *
     blockDim.x;
  if(i < n && src[i] > 0)
    dst[atomicAdd(nres, 1)] = src[i]; }
```

Global memory atomics

Source: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

Aggregation and atomics

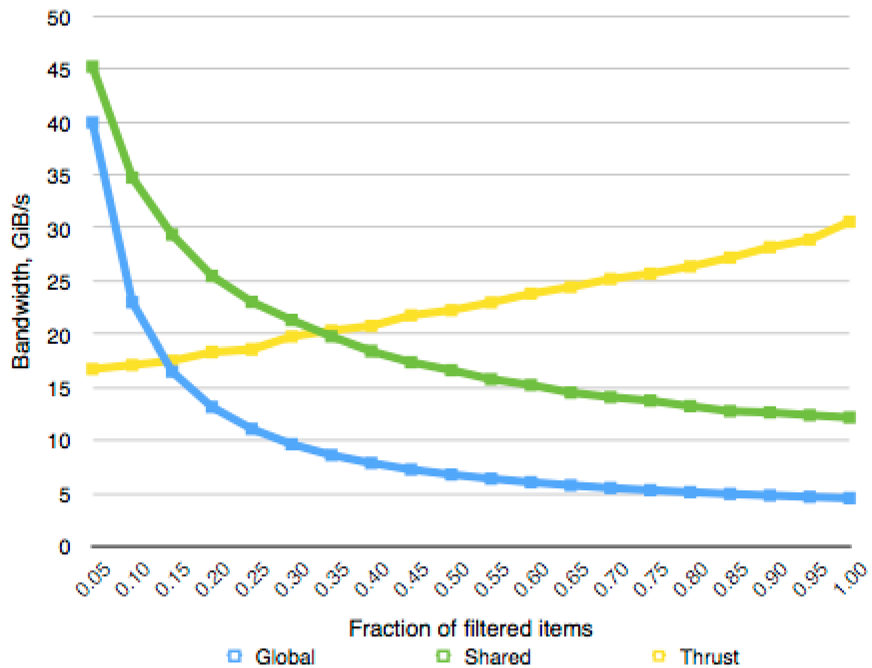
```
__global__
void filter_shared_k(int *dst, int
     *nres, const int* src, int n) {
  __shared__ int l_n;
  int i = blockIdx.x * (NPER_THREAD *
     blockDim.x) + threadIdx.x;

  for (int iter = 0; iter < NPER_THREAD;
     iter++) {
    // zero the counter
    if (threadIdx.x == 0)
      l_n = 0;
    __syncthreads();

    // get the value, evaluate the
    // predicate, and increment the
    // counter if needed
    int d, pos;

    if (i < n) {
      d = src[i];
      if(d > 0)
        pos = atomicAdd(&l_n, 1);
    }
```

```
    __syncthreads();

    // leader increments the global
    // counter
    if (threadIdx.x == 0)
      l_n = atomicAdd(nres, l_n);
      __syncthreads();

    // threads with true predicates
    // write their elements
    if (i < n && d > 0) {
      pos += l_n; // increment local pos
                  // by global counter
      dst[pos] = d;
    }
    __syncthreads();

    i += blockDim.x;
  }
}
```

Shared memory atomics

Each block uses local index l_n, then merges with global counter nres.

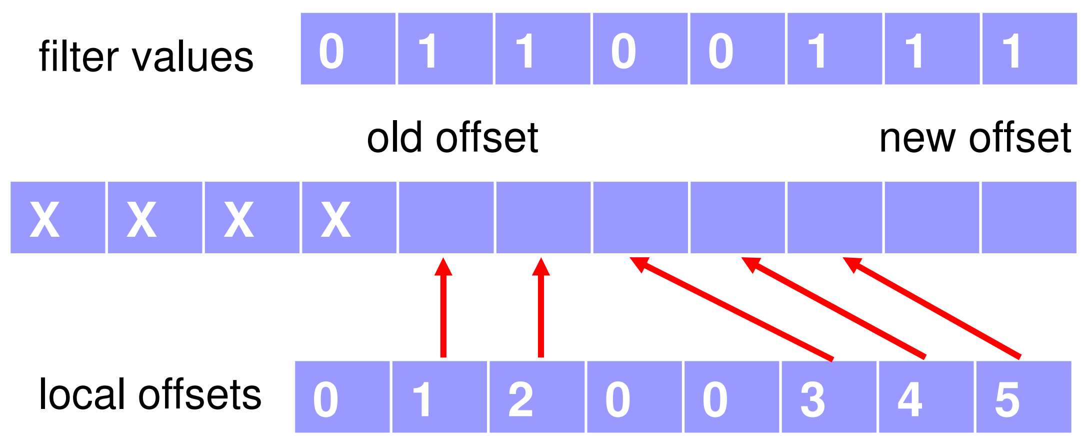
Aggregation and atomics



- Global memory atomics has very poor performance.
  - Performance degrades as more elements satisfy filter, due to conflicts.
- Shared memory over 2X better, but performance still degrades.
- Scan based filtering in Thrust has large up front cost.
  - But performance actually improves as filtered items increase, due to higher GPU utilization.

Warp-aggregated atomics

- Warp-based aggregation lets threads in a warp add their private values into a shared counter using fewer atomic adds.
  - Threads in the warp elect a leader.
  - Threads compute a total atomic increment for the warp.
  - Leader thread performs an atomic add, and gets the old global offset value.
  - Leader broadcasts global offset to all threads in warp.
  - Each thread adds its own local offset to global offset to get its final array index.



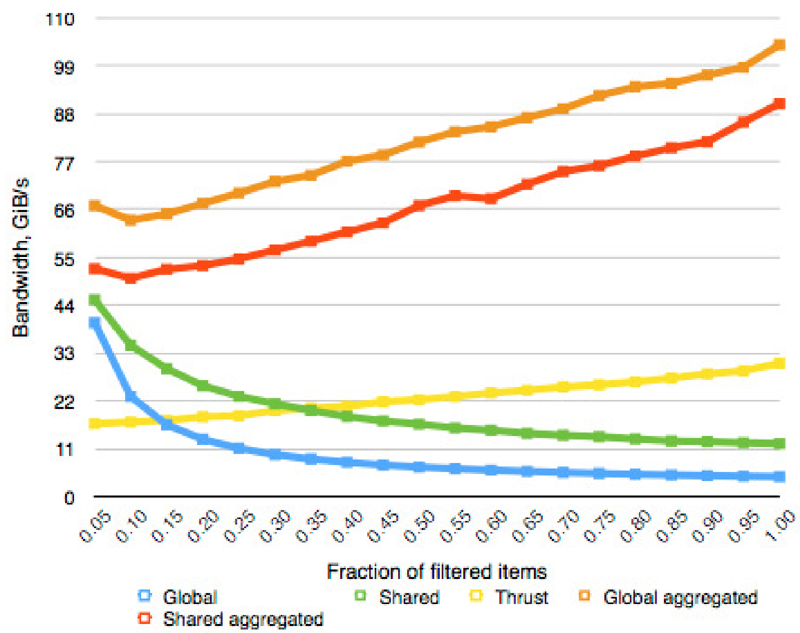
Warp-aggregated filtering

```
__device__
int atomicAggInc(int *ctr) {
  // mask of active lanes
  int mask = __ballot(1);

  // select the leader
  int leader = __ffs(mask) – 1;

  // leader does the update
  int res;
  if (lane_id == leader)
    res = atomicAdd(ctr, __popc(mask));

  // broadcast result
  res = __shfl(res, leader);

  // each thread computes its own value
  return res + __popc(mask & ((1 <<
      lane_id) – 1));
}
```

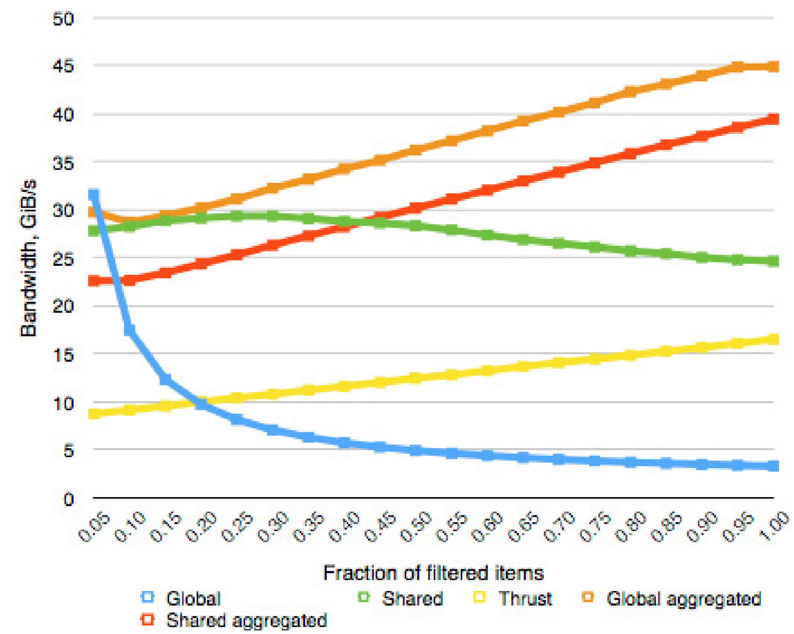
```
__global__ void filter_k(int *dst, const
      int *src, int n) {
  int i = threadIdx.x + blockIdx.x *
      blockDim.x;
  if (i >= n)
    return;
  if (src[i] > 0)
    dst[atomicAggInc(nres)] = src[i];
}
```

- Lane is a thread's index within the warp (0 to 31).
  - Compute as `threadIdx.x % 32`.
- Want lowest active lane (i.e. thread with positive src value) to be leader.
- `__ballot(v)` intrinsic returns a 32 bit mask indicating whether v is true at each lane.
  - So `__ballot(1)` selects the active lanes.
- `__ffs(mask)` intrinsic finds first set bit in `mask`.
- `__popc(mask)` intrinsic counts the number of set bits in `mask`.
- `__shfl(res, leader)` intrinsic lets `leader` broadcast `res` to all other threads in warp.
- `mask & ((1 << lane_id) – 1)` keeps only the set bits in `mask` before the `lane_id`'th bit.
  - Ex If `mask` = 10100110, `lane_id` = 4, then this sets mask to 0000110.
  - `__popc(mask & ((1 << lane_id) – 1))` counts number of threads satisfying predicate with lower ID.

Performance



Tesla K40 (Kepler)



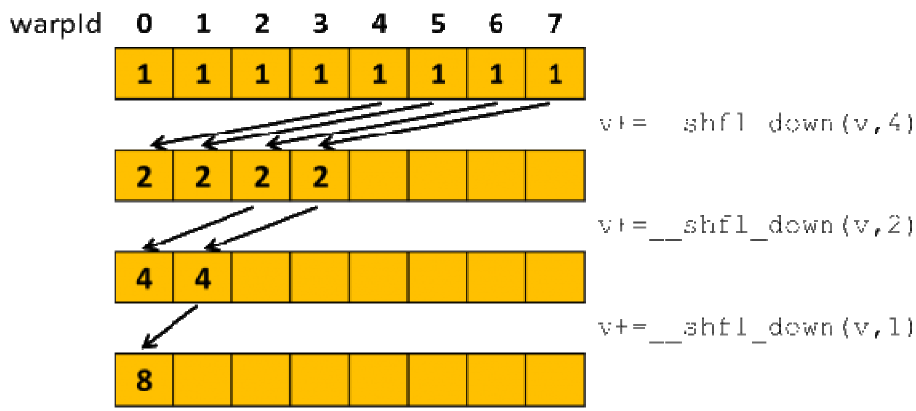
GeForce GTX 750 Ti (Maxwell)

- Up to 100 GB/s bandwidth on Kepler (simply copying has 180 GB/s).
  - Performance improves with filter ratio, because more elements written.
  - Global memory aggregation even faster than shared memory.
- Maxwell provides very efficient shared memory atomics, so the atomicAdd method is competitive for small filter ratios.
  - This particular Maxwell only had 5 SMs, hence the low overall performance.

Shuffle intrinsics

- Starting from Kepler, shuffle operations allow very fast moving of data between threads in a warp.
- Again, let lane = threadIdx.x % 32.
- Four shuffle operations.
  - `int __shfl_up(int var, int delta)`
    - `var` is a local register variable.
    - Copy value of `var` from a thread that's `delta` lanes lower than this thread.
      - Threads above lane `31-delta` don't do anything.
  - `int __shfl_down(int var, int delta)`
  - `int __shfl_xor(int var, int laneMask)`
    - Do XOR of `laneMask` with this thread's lane to determine lane to copy `var` from.
  - `int __shfl(int var, int srcLane)`
    - Copy `var` from lane `srcLane`.
- Can only read value from an active thread doing the shuffle instruction.
  - Copying from inactive thread leads to undefined behavior.
  - Be careful using shuffle on branching code.

Shuffle warp reduce

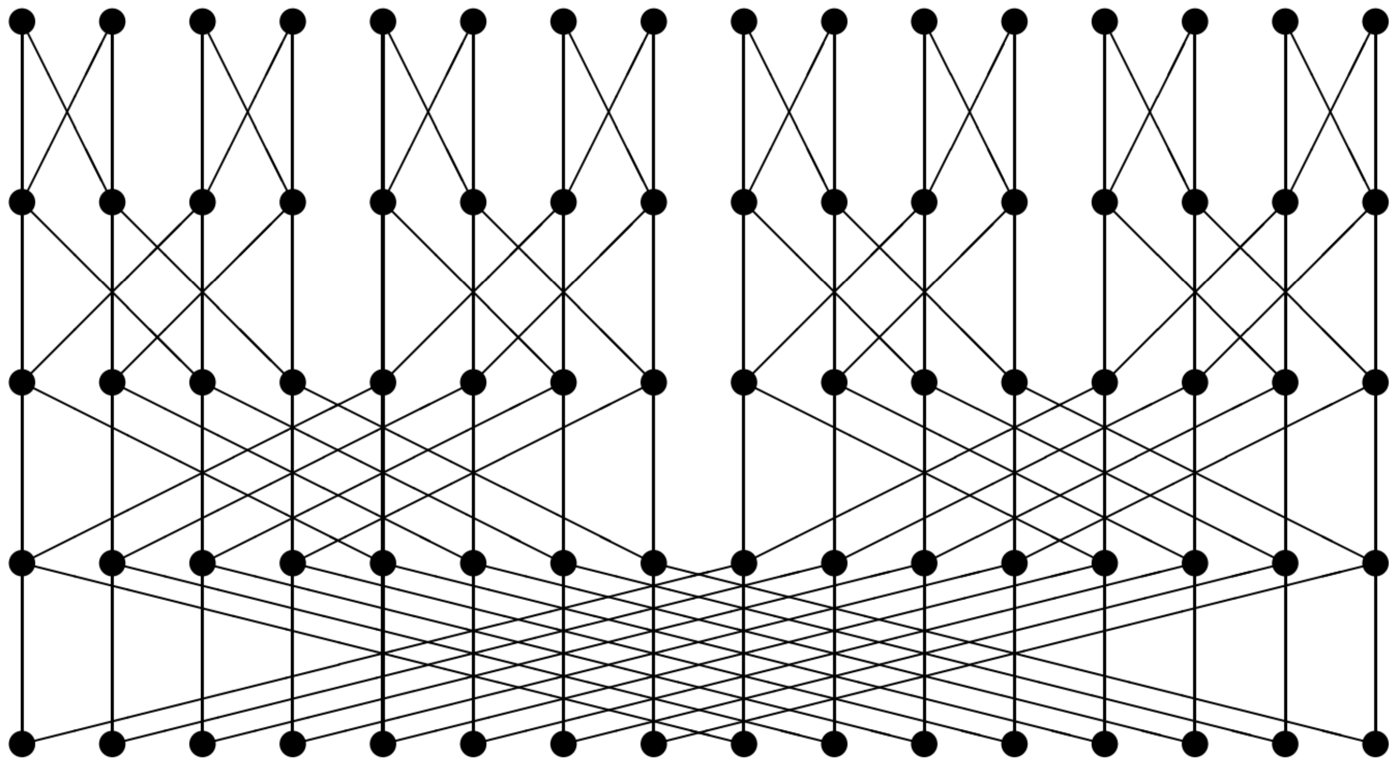


Source: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

- Can do reduce using shuffle instead of shared memory.
- Can be 2-3X faster than moving same data using shared memory.
- Also frees up shared memory for other uses.
- Since warp is SIMD, threads are automatically synchronized after shuffle, without need for __synchthreads().

Shuffle warp all-reduce

```
for (int i=1; i<32; i*=2)
  value += __shfl_xor(value, i);
```



Source: https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf

Block reduce

```
__inline__ __device__ int blockReduceSum(int val) {
  static __shared__ int shared[32];       // Shared mem for 32 partial sums
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize;

  val = warpReduceSum(val);               // Each warp performs partial reduction
  if (lane==0) shared[wid]=val;           // Write reduced value to shared memory
  __syncthreads();                        // Wait for all partial reductions

  //read from shared memory only if that warp existed
  val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;
  if (wid==0) val = warpReduceSum(val);   // Final reduce within first warp
  return val;
}
```

- Assume we have at most 1024 threads in the block.
- First do reduction in each warp.
- Then first thread in each warp writes value to a size 32 ($\geq$ block size / warp size) shared memory array.
- Then (a subset of) threads in the first warp read the array, then do another warp wide reduction.
  - This can reduce 32 x 32 = 1024 values.

Reducing larger arrays

```
__global__ void deviceReduceKernel(int *in,
     int* out, int N) {
  int sum = 0;
  //reduce multiple elements per thread
  for (int i = blockIdx.x * blockDim.x +
     threadIdx.x; i < N;
      i += blockDim.x * gridDim.x) {
    sum += in[i]; }
  sum = blockReduceSum(sum);
  if (threadIdx.x==0)
    out[blockIdx.x]=sum;
}
```

```
void deviceReduce(int *in, int* out, int N)
     {
  int threads = 512;
  int blocks = min((N + threads - 1) /
     threads, 1024);

  deviceReduceKernel<<<blocks,
     threads>>>(in, out, N);
  deviceReduceKernel<<<1, 1024>>>(out, out,
     blocks);
}
```

- Run two kernels, first in which every block produces a sum, and the second to add up the block sums.
  - Assume at most 1024 thread blocks.
- Each thread in grid first sums array elements in strides of the grid size.
  - Number of threads in the grid is T = blockDim.x * gridDim.x.
  - Each thread sums every T'th array element.
- First thread in each block writes block sum to array out.
- Run second kernel with one thread block to sum up the ≤ 1024 block sums.