Case Study: Reduction

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Reduction (simplest)

An algorithmic pattern where entries in an array are combined together to produce a scalar

```
double v[SIZE];
double res=INITIAL_VALUE

for (i=0; i<SIZE; i++)
   res = OP(res,v[i]);
```

where `OP` is a binary operation

Example:

$$s = \sum_i v(i)$$

This example is based on a similar one by Mark Harris, NVIDIA; see http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Why parallel reduction?

- It is an important parallel primitive (native in OpenMP and MPI), appears as a building block in many algorithms;
- It is extremely easy to describe (i.e. a simple implementation is immediate);
- It is surprisingly difficult to optimize (hence good case study).

Parallel Reduction

Why parallel reduction?

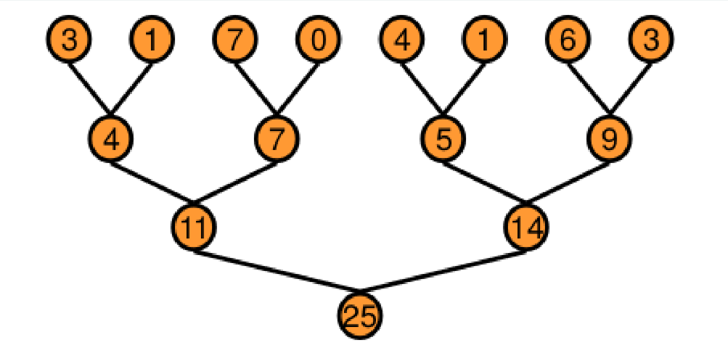
- It is an important parallel primitive (native in OpenMP and MPI), appears as a building block in many algorithms;
- It is extremely easy to describe (i.e. a simple implementation is immediate);
- It is surprisingly difficult to optimize (hence good case study).

Parallel and serial reductions will not give bit-identical results

To parallelize the reduction, we will have to (somehow) split the sum among the various processors; but this means that we are changing the order of summation. Thus, the results will be the same only for operators that are strictly associative; integer arithmetic is associative, but *floating-point arithmetic is not!*.

How to parallelize?

Use tree data flow structure



- Need to use multiple thread blocks: Very large arrays, multiple SMs on device;
- Need to communicate among thread bocks;

Global Synchronization

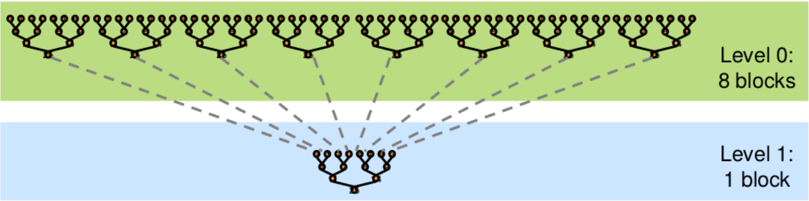
Synchronization anyone?

- We need synchronization: each block of threads does its own thing, once all are done, proceed recursively;
- Only, there's no synch among blocks:
  - Too expensive in hardware;
  - Would restrict number of blocks (occupancy);

Solution

Use kernel launch as implicit synchronization

Decompose into multiple kernel invocations



Code is the same at all levels (recursion).

How fast can we go?

Build a *roofline* model: where does the limitation on performance come from?

Compute-bound or memory-bound?

The big alternative:

- For compute-bound problems, the metric is GFLOPS;
- For memory-bound problems, Bandwidth is arguably a better metric (although people will often use GFLOPS anyway).

Our reduction performs one arithmetic operation per load, hence
$\Rightarrow$ Use bandwidth.

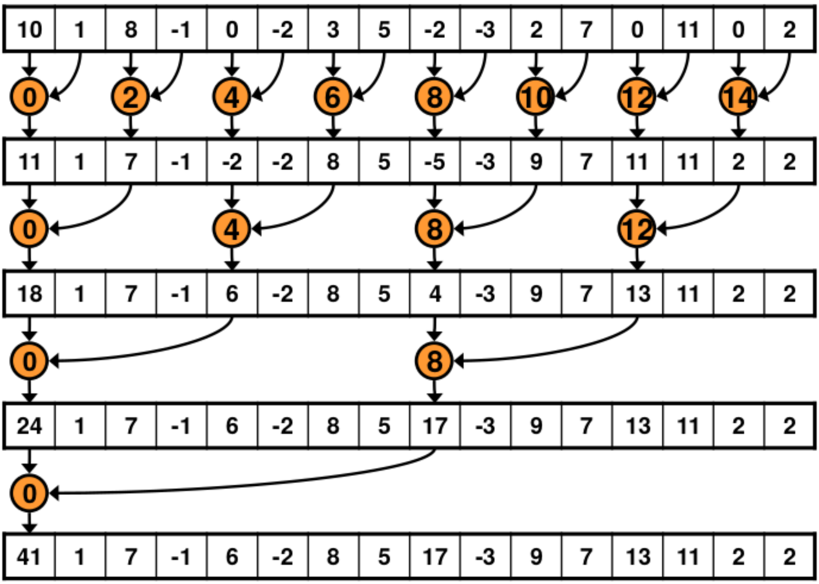
Numbers on K40: 384-bit interface, 3000 MHz DDR, hence
$(384 * 2 * 3000)/8 = 288GB/s$ theoretical peak bandwidth

Numbers on RTX5000: 256-bit interface, 7000 MHz DDR, hence
$(256 * 2 * 7000)/8 = 448GB/s$ theoretical peak bandwidth

Version 1: interleaved addressing

```
__global__ void reduce1(int n, double *g_idata,
                        double *g_odata)       {
  extern __shared__ double sdata[];
  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = 0.0;
  if (i<n) sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] += sdata[0];
}
```

Version 1: interleaved addressing



Version 1: interleaved addressing

		Performance	
Version	GPU Time	Bandwidth	Bandwidth
		K40 CUDA 10	RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s

Versikon 2: strided index

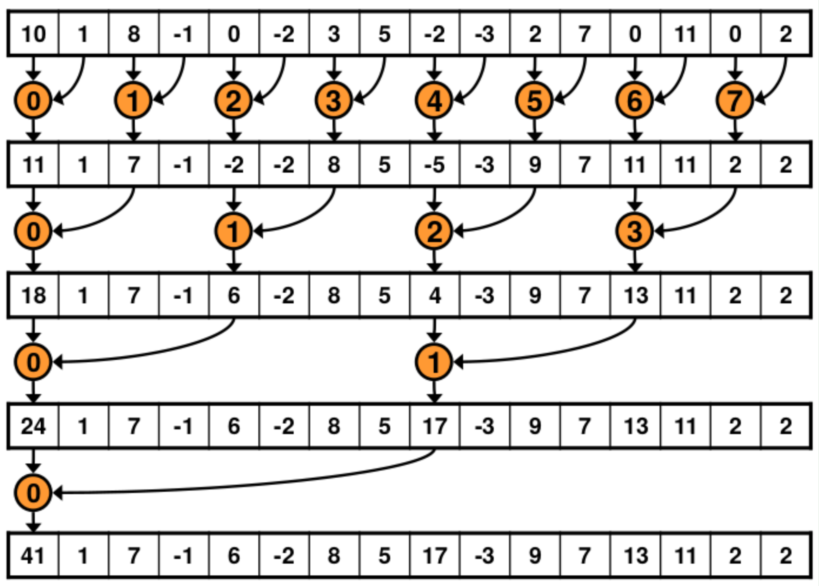
Replace divergent branch

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

with strided index

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2*s*tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

Version 2: strided index



Version	GPU Time	Performance Bandwidth K40 CUDA 10	Bandwidth RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s

Version 3: sequential addressing

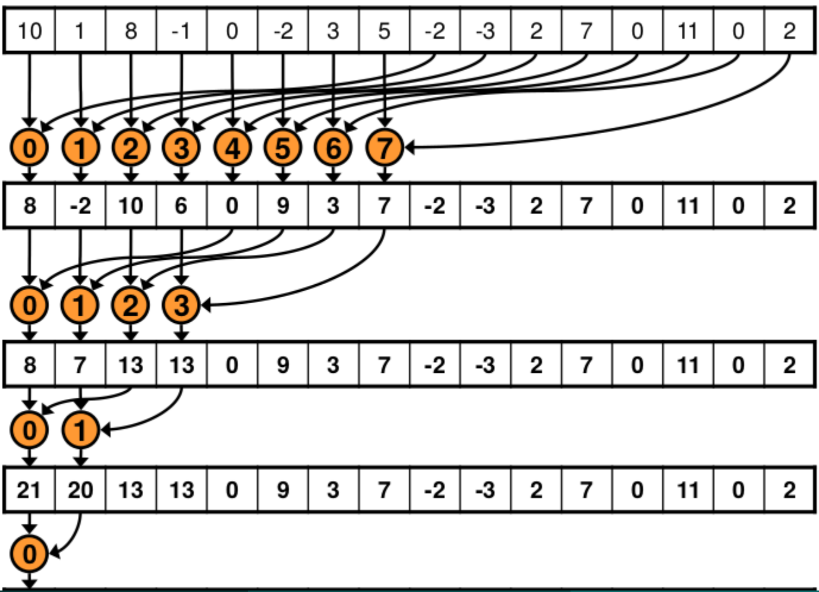
Replace strided index

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2*s*tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

with reversed loop and thread-based indexing:

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

Version 3: sequential addressing



Version 3

Version	GPU Time	Performance Performance Bandwidth K40 CUDA 10	Bandwidth RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s

Idle threads

The inner loop starts at half the block size

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

Hence, half the threads are always idle!

Version 4: Load & Add

Go from

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = 0.0;
if (i<n)
  sdata[tid] = g_idata[i];
__syncthreads();
```

to blocks with double-sized footprint

```
// each thread loads and sums two elements
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = 0.0;
if (i<n)
  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Version 4

Version	GPU Time	Performance Performance Bandwidth K40 CUDA 10	Bandwidth RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s
4	1.19600 ms	56 GB/s	112 GB/s

Why so slow?

- We are still very far from bandwidth, but
- We know the FLOPs cannot be the limit; hence
- Consider auxiliary operations: address arithmetic, loop overheads, `__syncthreads`

$\Rightarrow$ Modify and unroll loops

Version 5: Save the last warp

Number of active threads decreases with `s`:

- At some point *There can be only one* warp active;
- Threads in a warp work in SIMD (lockstep);
- No need for `__syncthreads`

Version 5: Save the last warp

```
// do reduction in shared mem
for(unsigned int s=blockDim.x>>1; s>=32; s >>=1) {
  if (tid  < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
for(unsigned int s=32>>1; s>0; s >>=1) {
  sdata[tid] += sdata[tid + s];
}
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] += sdata[0];
```

Version	GPU Time size 8388608	Performance Performance Bandwidth K40 CUDA 10 size 16777216	Bandwidth RTX5000 CUDA 11 size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s
4	1.19600 ms	56 GB/s	112 GB/s
5	0.85900 ms	82 GB/s	166 GB/s

Version 6: Unroll the last warp

We've come so far, we now have only one warp active:

Why keep around the loop structure?

- Unroll the loop;
- But, need `volatile` to be safe

Version 6: Unroll the last warp

Do away with the loop

```
__device__ void warp_reduce(volatile double *sdata,
                            int tid) {
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid +  8];
  sdata[tid] += sdata[tid +  4];
  sdata[tid] += sdata[tid +  2];
  sdata[tid] += sdata[tid +  1];
}
```

Version 6: Unroll the last warp

Invoke as

```
// do reduction in shared mem
for(unsigned int s=blockDim.x>>1; s>=32; s >>=1) {
  if (tid  < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
if (tid < 32) warp_reduce(sdata,tid);
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] += sdata[0];
```

Version 6

		Performance	
Version	GPU Time	Bandwidth	Bandwidth
		K40 CUDA 10	RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s
4	1.19600 ms	56 GB/s	112 GB/s
5	0.85900 ms	82 GB/s	166 GB/s
6	0.82600 ms	83 GB/s	166 GB/s

Version 7: Unroll the outer loop

Take out the outer loop structure

- We *cannot* avoid synchronization;
- But will save loop overhead;
- We want to keep flexible with respect to number of threads per block;
- Unroll in source code;

Use a template

```
const int shmem_size    = THREAD_BLOCK*sizeof(double);
int    nblocks = ((n + THREAD_BLOCK - 1) / THREAD_BLOCK);

reduce<THREAD_BLOCK><<<nblocks,THREAD_BLOCK,
                        shmem_size,0>>>(n,g_idata,g_odata);
return;
```

Define unrolling with template

```
template <unsigned int THD> __global__ void reduce(int n,
                  double *g_idata, double *g_odata) {
  extern __shared__ double sdata[];
  // each thread loads and sums two elements
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  sdata[tid] = 0.0;
  if (i<n) sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();
  if (THD >= 1024){ if (tid < 512) { sdata[tid] += sdata[tid + 512]; }
                    __syncthreads();  }
  if (THD >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
                    __syncthreads();  }
  if (THD >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
                    __syncthreads();  }
  if (THD >= 128) { if (tid < 64)  { sdata[tid] += sdata[tid +  64]; }
                    __syncthreads();  }
  if (THD >=  64) { if (tid < 32)  { sdata[tid] += sdata[tid +  32]; }
                    __syncthreads();  }
  if (tid < 32) warpReduce(sdata,tid);
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] += sdata[0];
```

		Performance	
Version	GPU Time	Bandwidth	Bandwidth
		K40 CUDA 10	RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s
4	1.19600 ms	56 GB/s	112 GB/s
5	0.85900 ms	82 GB/s	166 GB/s
6	0.82600 ms	83 GB/s	166 GB/s
7	0.75600 ms	88 GB/s	176 GB/s

Version 8/9/10

One major, one minor issue

- With most threads we are still doing only a few FLOPs;
- We should be able to choose the block size at runtime;

The second issue can be solved with templated structure

```
void do_gpu_reduce(int n, double *g_idata, double *g_odata)
{ const int shmem_size   = thread_block*sizeof(double);
  int   nblocks = ((n + thread_block - 1) / thread_block);
  if (nblocks > max_blocks) nblocks = max_blocks;

  switch(thread_block) {
  case 1024:
    reduce<1024><<<nblocks,1024,shmem_size,0>>>(n,g_idata,g_odata); break;
    .....
  case 64:
    reduce<64><<<nblocks,64,shmem_size,0>>>(n,g_idata,g_odata); break;
  default:
    fprintf(sdterr,"thread_block must be a power of 2 between 64 and 1024");
  }
  return;
}
```

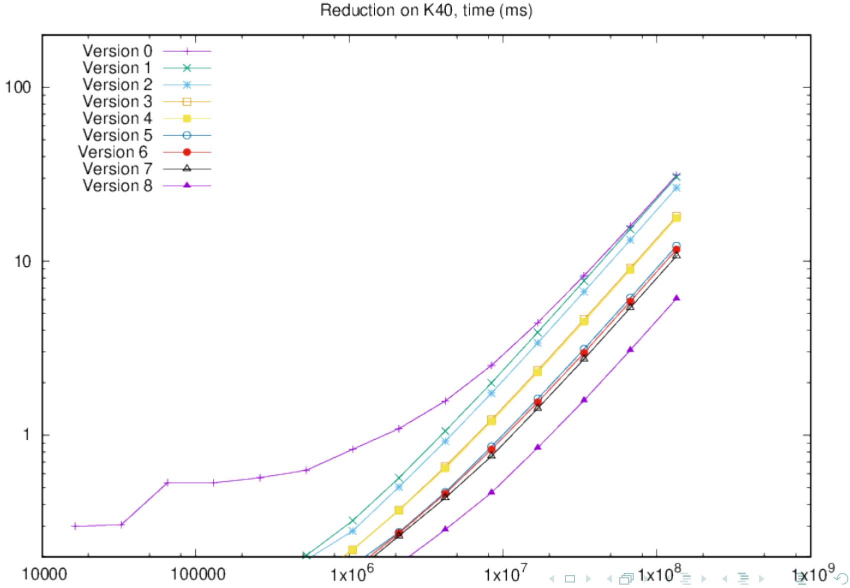
Version 8/9/10

For the first issue, the way to go is to let each thread work on multiple entries of the input vector, by using the grid as a sliding window over the input vector:

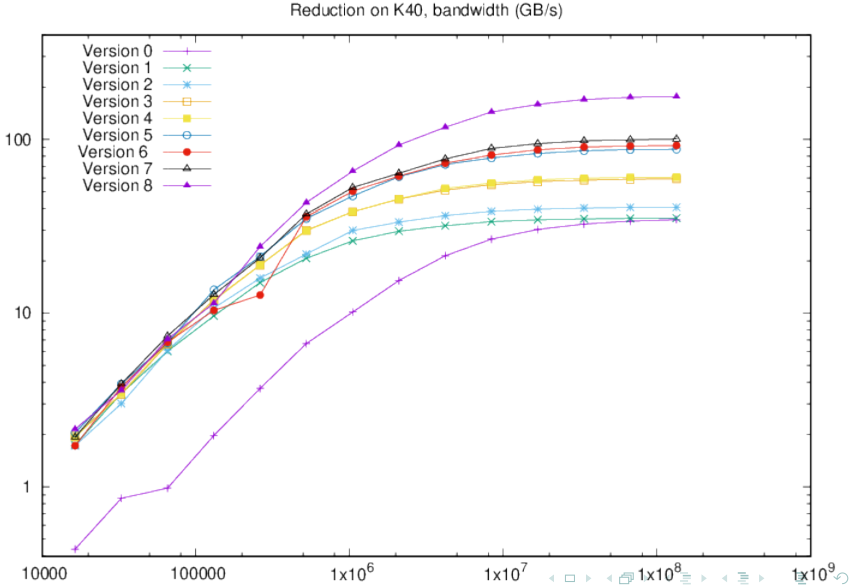
```
template <unsigned int THD> __global__ void reduce(int n,
                         double *g_idata, double *g_odata) {
  extern __shared__ double sdata[];
  // each thread loads and sums multiple elements
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int gridSize = blockDim.x * gridDim.x;
  sdata[tid] = 0.0;
  while (i<n) {
    sdata[tid] += g_idata[i] ;
    i += gridSize;
  }
  __syncthreads();
  // do reduction in shared mem
  if (THD >= 1024){ if (tid < 512) { sdata[tid] += sdata[tid + 512]; }
                                      __syncthreads();  }
  .....
```

		Performance	
Version	GPU Time	Bandwidth	Bandwidth
		K40 CUDA 10	RTX5000 CUDA 11
	size 8388608	size 16777216	size 400000000
1	1.99800 ms	36 GB/s	96 GB/s
2	1.74200 ms	40 GB/s	117 GB/s
3	1.22500 ms	57 GB/s	122 GB/s
4	1.19600 ms	56 GB/s	112 GB/s
5	0.85900 ms	82 GB/s	166 GB/s
6	0.82600 ms	83 GB/s	166 GB/s
7	0.75600 ms	88 GB/s	176 GB/s
8	0.46700 ms	153 GB/s	404 GB/s
9		159 GB/s	404 GB/s
10		154 GB/s	404 GB/s

Performance summary



Performance summary



Further issues

- Can we assume that we do not need `__syncthreads()` within a warp?
- Would be nice to exchange data without going to shared memory

CUDA warns you that the absolute lock-step execution of threads in a warp could be relaxed in the future. Solution?

Further issues

- Can we assume that we do not need `__syncthreads()` within a warp?
- Would be nice to exchange data without going to shared memory

CUDA warns you that the absolute lock-step execution of threads in a warp could be relaxed in the future. Solution?

shfl

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width);
```

They are only available from CUDA 9. In CUDA 10 there are some API changes in this area.

Version 10

```
// Fully unrolled
__device__ double warpReduce(double term) {
  double sum=term; const int width=32;

  for (unsigned int i=width>>1; i>0; i >>=1) {
    double value = __shfl_down_sync(0xffffffff, sum, i, width);
    sum += value;
  }
  return(sum);
}


 ....
  if (THD >= 64)  { if (tid < 32)  {
      term=sdata[tid] + sdata[tid + 32];}  __syncthreads();
  }
  if (tid < 32) sum=warpReduce(term);
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] += sum;
```

Exercises

1. Reimplement all of the above, starting from the available initial version(s);
2. Test performance;
3. Can you do any better than this?
4. Reuse the same techniques to implement a dot product;
5. Reuse the same techniques to implement the 2-norm of a vector;
6. Compare performance of the dot product and/or 2-norm with CUBLAS.