CS 1302, 2012: Lab 2: Parallel Reduction on a GPU

[Due Date: 15th July 2012]
[Submission through github]

Code tarball: lab2.tgz

In this hands-on lab, you will implement an optimized parallel reduction code on a GPU.

- Reduction slides: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

We will ask you to modify several `.cu` (CUDA) files;

Part 1 : Naive Parallel Reduction

There are 5 CUDA program files (`.cu` files) in the downloaded tarball.

- `lab2_naive.cu`: This file contains a naive implementation of parallel reduction. This implementation suffers from several inefficiencies.
- `lab2_stride.cu`: You will implement an optimized implementation where consecutive threads are working on strides of the input array elements. Refer to Part 2 for more information on how to implement this part.
- `lab2_sequential.cu`: You will implement an optimized implementation where consecutive threads are working on consecutive elements of the input array. Refer to Part 3 for more information on how to implement this part.

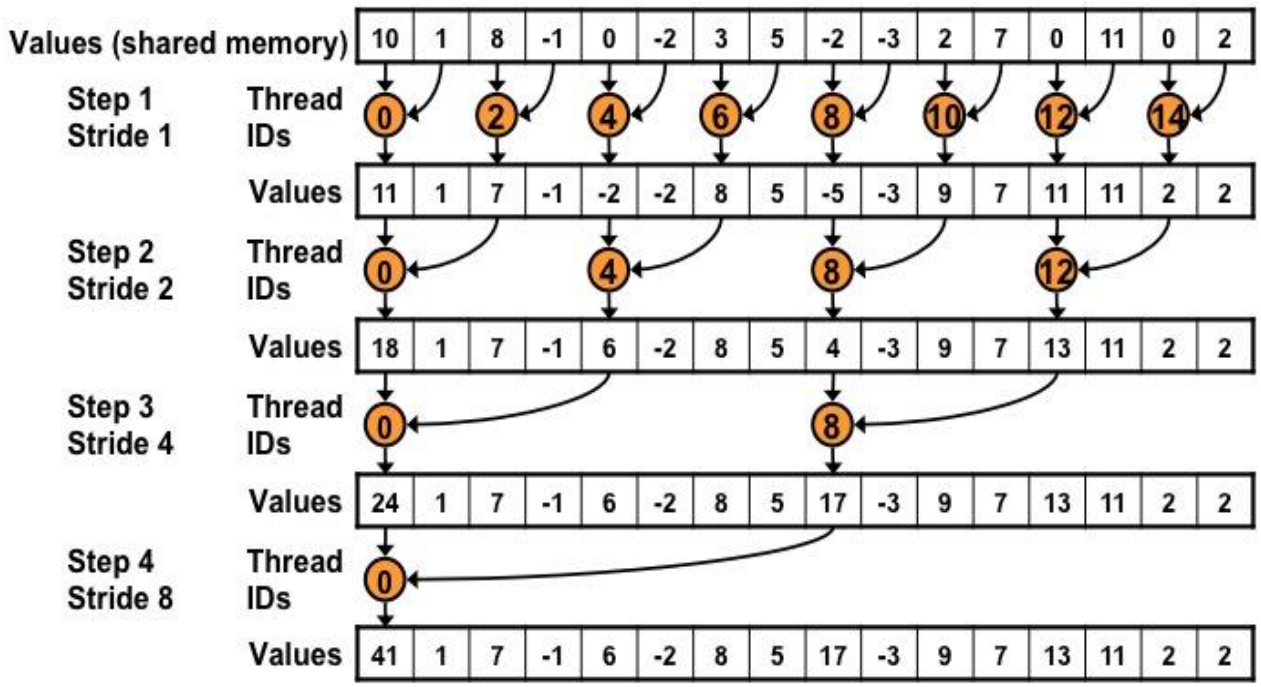
First of all, open `lab2_naive.cu` and find the function `kernel0`. From its `__global__` specifier, we know that this is a GPU kernel. The content of this function is shown below. The arguments on Line 2 point to the beginning of the `input` and `output` arrays. The variable `n` is the number of input elements.

```
1  __global__ void
2  kernel0 (dtype *input, dtype *output, unsigned int n)
3  {
4    __shared__  dtype scratch[MAX_THREADS];
5
6    unsigned int bid = gridDim.x * blockIdx.y + blockIdx.x;
7    unsigned int i = bid * blockDim.x + threadIdx.x;
8
9    if(i < n) {
10     scratch[threadIdx.x] = input[i];
11   } else {
12     scratch[threadIdx.x] = 0;
13   }
```

```
14    __syncthreads ();
15
16    for(unsigned int s = 1; s < blockDim.x; s = s << 1) {
17      if((threadIdx.x % (2 * s)) == 0) {
18        scratch[threadIdx.x] += scratch[threadIdx.x + s];
19      }
20      __syncthreads ();
21    }
22
23    if(threadIdx.x == 0) {
24      output[bid] = scratch[0];
25    }
26  }
```

Line 4 declares an array, `scratch`, of *shared memory*. Recall that "shared memory" in CUDA parlance refers to the scratchpad memory that is shared by all threads in a thread block. The constant `MAX_THREADS` is defined to be the number of threads in each block; thus, declaring `scratch` to be of this size implies 1 word of shared memory per thread.

Lines 6 and 7 compute a global ID for the thread. This ID is stored in `i`. This kind of calculation is hopefully familiar to you from the previous lab.
In lines 9–14, each thread loads an element from the global array into the shared memory. In this case, `i` serves as both a global thread ID *and* the index of the input element, `input[i]`, assigned to this thread. The call to `__syncthreads()` is a barrier for all threads within the same thread block. Here, it is used to ensure that all thread loads have completed prior to any computation on these data.
Lines 16–21 perform the reduction within the thread block. The figure below shows the mapping of threads to shared memory array indices.



Lines 23–25 show that only thread 0 in each block writes the sum back to the output array. Notice that the index into the output array is `bid`. That is, only thread 0 from *each* block produces a reduced output. By indexing the output array this way, we ensure that the result resides consecutively in memory. This is done so that during the next phase of reduction, data will again be accessed consecutively (coalesced memory access).

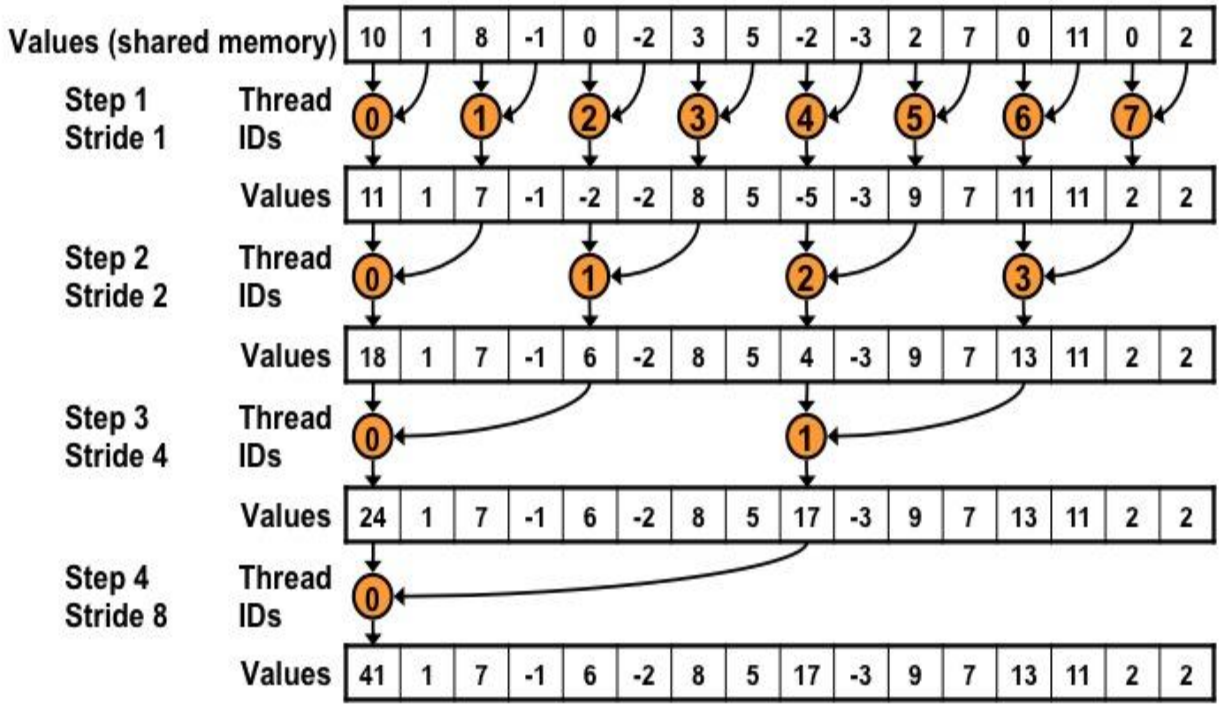
1. i. Compile and run this code, which reports the input vector size, the time to execute the kernel, and an "**effective bandwidth**." Record these data. Explain how the effective bandwidth is being calculated.
   ii. Refer CUDA Best Practices Guide (http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf) section 5.2 from page 23
   iii. Observe the functioning of the "reduce_cpu" function. Why the reduction is not straight forward ? (http://en.wikipedia.org/wiki/Kahan_summation_algorithm)
   You are recommended to refresh your memory on Floating point presentation and arithmetic: http://www.cs.cmu.edu/~213/lectures/04-floats.pdf
   iv. Is this code compute bound or memory bound or latency bound ? What is the maximum effective performance that the code can achieve?

Part 2: Strided Access by Consecutive Threads

The naive implementation has inefficiencies arising from *divergent warps*. In each successive iterations of the loop, the number of *active* threads per warp is halved, causing two problems: (a) divergent control flow between active threads and inactive threads, and

(b) under-utilization of the threads in each warp, since every warp in the thread block needs to execute even though the number of active threads in each warp is decreasing.

Therefore, we would like to change the code so that at each level of the reduction only consecutively numbered threads remain active even as the number of active threads decreases. The following figure illustrates this technique.



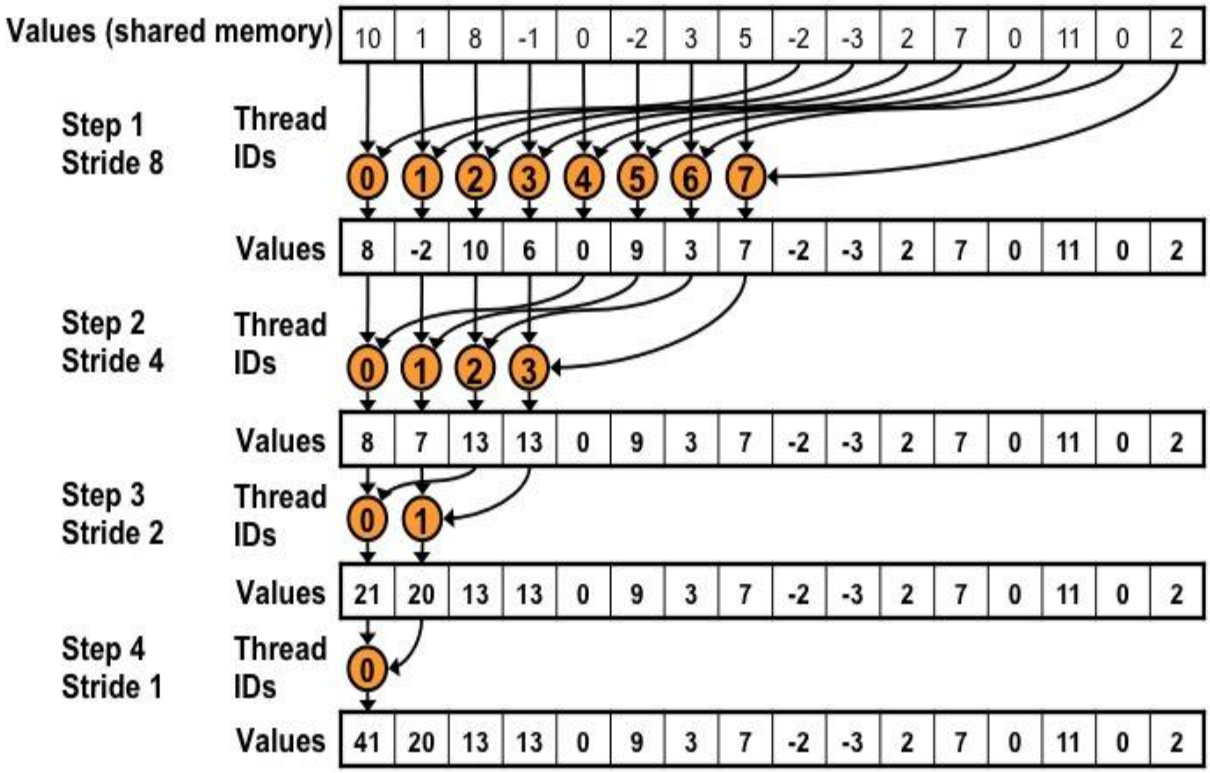
2. Implement this scheme in `kernel1` of `lab2_stride.cu`. Measure and record the resulting performance. How much faster than the initial code is this version?
Hint: The kernel is very similar to the naive kernel. You should only need to modify Lines 17–19 of the naive kernel.

Part 3 : Sequential Access by Consecutive Threads

Both of the previous kernels suffer from another inefficiency, known as *bank conflicts*. Without going into the gory details, bank conflicts essentially occur when the threads of a warp make *strided* accesses to *shared memory*. Thus, having a warp's threads access consecutive words in global memory is also a good policy for shared memory. The most common way to prevent bank conflicts is to ensure that the threads of a warp access contiguous words in shared memory.

For more information on bank conflicts, refer to the CUDA Programming Guide.

This mapping of threads to shared memory indices is shown in the following figure.

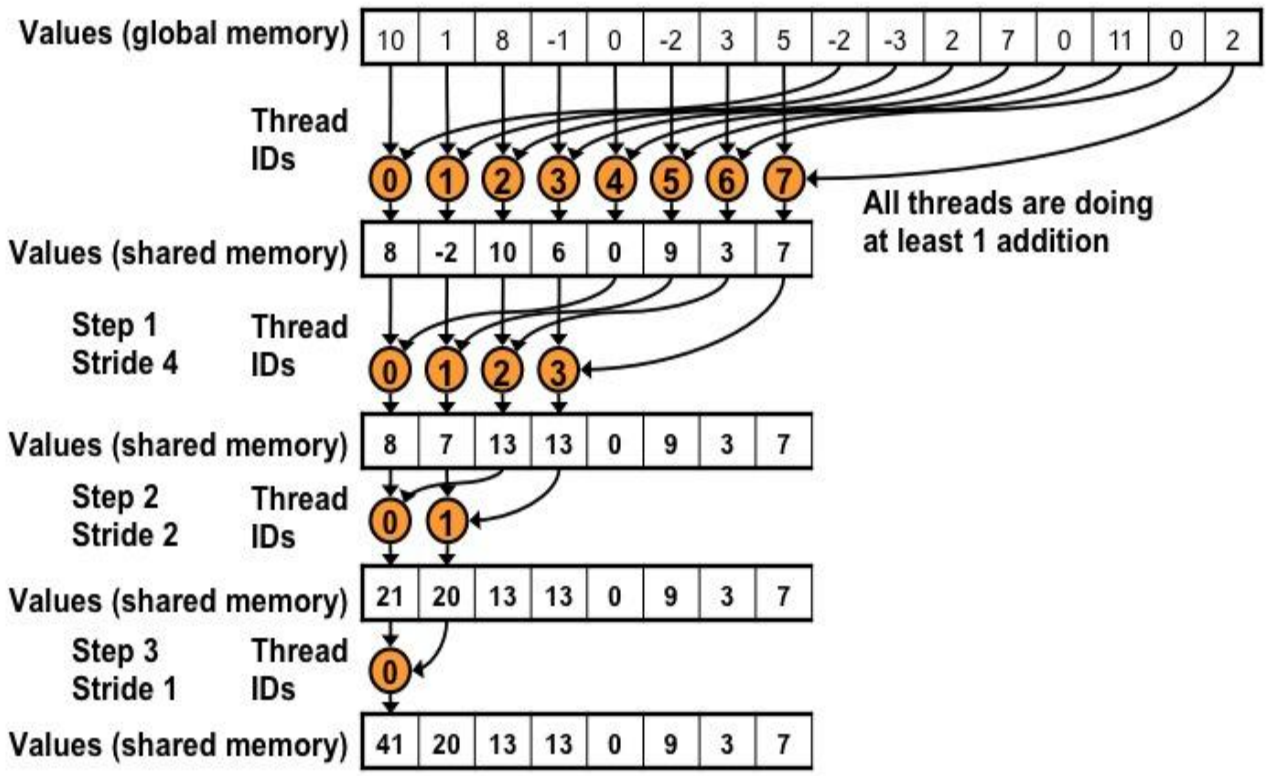


3. Implement this scheme in `kernel2` of `lab2_sequential.cu`. Record the new effective bandwidth.

Part 4 : First Add Before Reduce

The above implementations all suffer from the fact that after each thread loads its global array element into the shared memory, half of them immediately become inactive and take no part in the actual computation. The kernel can be improved by using half the number of threads as before and having each thread loading 2 elements from the global array instead, and then summing them together before writing the result into the shared memory.

This mapping of threads to shared memory indices is shown in the following figure.



4. Implement this scheme and report the effective bandwidth.

Part 5: Algorithm cascading

In this part of the lab, instead of having each thread load 2 elements from the global array, have them load multiple elements and then sum them all up before placing the result into the shared memory.

The reduction tutorial slides referred to this technique as "algorithm cascading." We gave a theoretical reason for this in terms of *parallel efficiency* during one of the classes.

In the main function, we have restricted the maximum number of threads to 256 AND the maximum number of thread blocks to 64. This means that there are at most 16384 threads. If the input size is 8388608 elements, then each thread will have to sum up 512 elements from the global array before storing the sum into the shared memory.

5. Implement this scheme and report the effective bandwidth.