High-Performance Computing May 20, 2020
University of California, Irvine EECS 224

Homework3: GPU/CUDA

Due: 5 P.M. on Thursday, June 4, 2020

In this assignment, you will implement a parallel reduction and transpose on a GPU and learn some practical techniques to make it run faster.

For this assignment, there may be a lot of competition for time on the cluster. As such, we **strongly recommend that you start early**.

1 Getting the scaffolding code

We've written some skeleton code for you to use for this homework. The baseline code for this assignment is available at this URL: `https://github.com/UCI-HPC/EECS224-hw3.git`. To get it, use the same git-clone procedure from homework 1 and 2. If it worked, you'll have a new directory called **EECS224-hw3**.

2 Parallel Reduction

2.1 Naive Parallel Reduction

There are 6 CUDA program files (`.cu` files). Let's begin with `naive.cu` – this file contains a naive implementation of parallel reduction. This implementation suffers from several inefficiencies, which you will fix.

Before you start compiling `naive.cu`, first type the following commands to load the appropriate CUDA and gcc compilers.

```
$ module load  cuda/8.0
$ module load  gcc/4.9.0
```

Then, you can compile the code as follows:

```
$ nvcc naive.cu timer.c -o naive
```

which produces `naive` executable. Try running this on the login node. You should get an error, because the login node does *not* have a GPU.

```
$ ./naive      # <--- Should report an error when run on hpc-login
```

Now try instead to submit a job to the GPU node using the provided `cuda.sh` file. It should work, and report some statistics on its performance.

Start by opening `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.

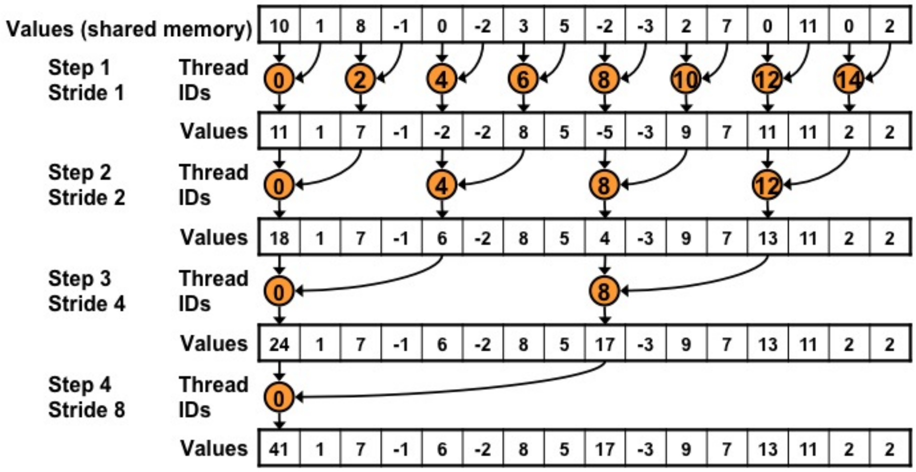
Lines 6 and 7 compute a global ID for the thread. This ID is stored in i.

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 ouput. 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).

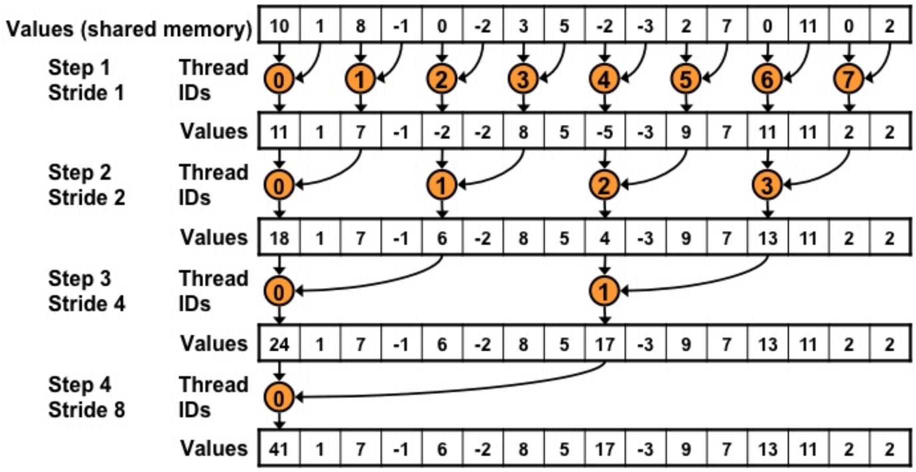
1. Compile and run this code, which reports the input vector size, the time to execute the kernel, and an effective bandwidth. Record these data. Explain how the effective bandwidth is being calculated.



2.2 Strided Access by Consecutive Threads

The performance of the naive code suffers from divergent warps. In 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.



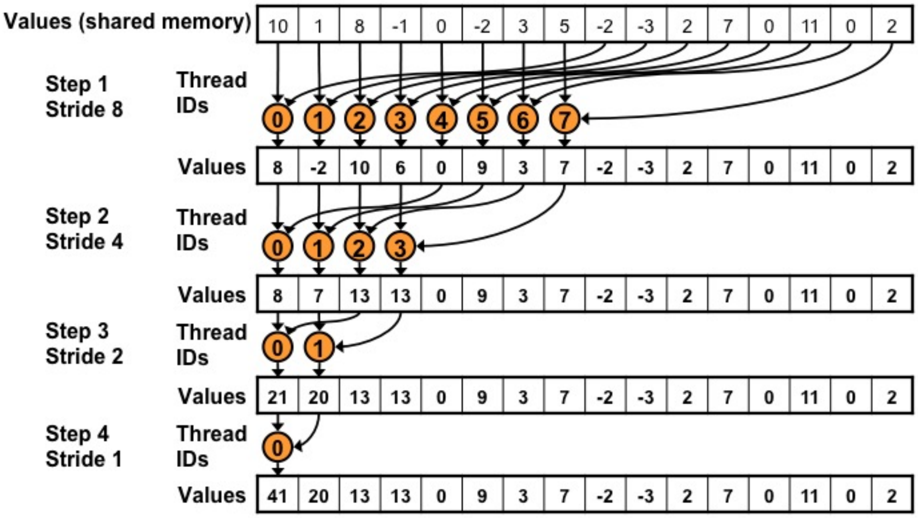
2. Implement this scheme in `kernel1` of `stride.cu`. Measure and record the resulting performance. How much faster than the initial code is this version?

Hint: The kernel is very similar to the naive kernel. You should only need to modify Lines 17-19 of the naive kernel.

2.3 Sequential Access by Consecutive Threads

Both of the previous kernels suffer from another inefficiency, known as *bank conflicts*. *For more information on bank conflicts, refer to the CUDA Programming Guide.*

The mapping of threads to shared memory indices that fixes bank conflicts in parallel reduction is shown in the following figure.

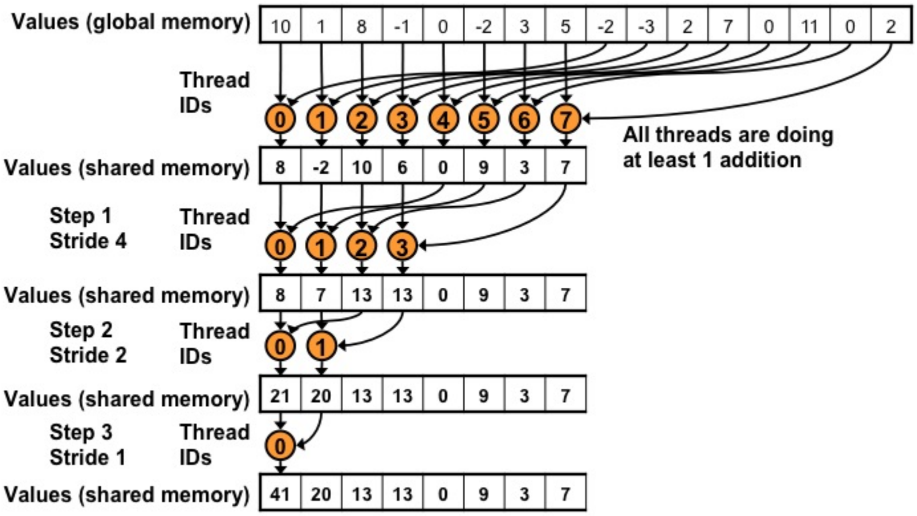


3. Implement this scheme in `kernel2` of `sequential.cu`. Record the new effective bandwidth.

2.4 First Add Before Reduce

All of the above implementations can be further improved. Observe 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. Therefore, to improve the number of threads doing useful work, we can use half the number of threads we did before and have each thread load and sum 2 elements from the global array instead.

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



4. Implement this scheme in `kernel3` of `first_add.cu` and report the effective bandwidth.

2.5 Unroll the Last Warp

During the GPU reduction lecture, we saw an optimization technique which lets us unroll the last 6 iterations of the inner loop. This is based on exploiting the fact that instructions are SIMD synchronous within a warp.

5. Implement this scheme in `kernel4` of `unroll.cu` and report the effective bandwidth.

2.6 Multiple

In this final optimization, 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. This technique is referred to as *algorithm cascading*.

6. Implement the algorithm cascading scheme in `multiple.cu` and report the effective bandwidth.

Note: 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.

3 Matrix transpose

The repo we've provided includes another file, `transpose.cu` which contains the skeleton code for transposing a matrix on the GPU. Only the code for creating the input matrix on the host has been given. This means that you'll have to write the code needed for allocating memory for the matrix on the GPU, as well as communication to and fro. More specifically, edit the functions `void gpuTranspose` and `__global__ void matTrans` without changing any of the provided code.

Although you have been given freedom to write your own code for this part, you CANNOT change the data structure of the matrix on the host. You may, however, pad the array to ensure that each row starts from an aligned memory address.

In addition to submitting your code, briefly describe how your algorithm works and optimizations attempted. Also, present performance results and a brief discussion of the results.

4 Submission

When you've written up answers to all of the above questions, turn in your write-up and zip/-tarball of your code by uploading it to Canvas.

Take care and have fun programming GPUs!