DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: Final Exam Problem 3: Stream Compaction

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 3/25/2017

DATE DUE: 3/25/2017

Assignment description:

This assignment involved designing the device side GPU code to implement compact streaming on a given input array. The provided scan function will be able to take a computed flag array as input, and output the corresponding scanned flag array that will be necessary in determining the indices of the desired array elements. The host side function will also need to be created in order to allocate the input and output arrays, as well as copy the input array to the device side's input array.

Parallelization approach:

The parallelization approach for this assignment was, that assuming the number of elements is 1024, each thread within a thread block will perform operations on the given ith index of each array, whether it be the input, output, flag, or scanned flag arrays. The scan function uses the "ping-ponging" approach to switch between reading and writing to separated areas within the shared buffer. The kernel function will first get each thread to index a certain area within the input array and check whether that element is positive or negative, and denote this by supplying into a flag array, a 0 or 1 to the corresponding index. Once the flag has been created, the flag will be supplied to the scan function where it will output an array that details the index positions of the desired value that should be stored. By checking the same index value of the flag and seeing the index of the scanned flag array, each thread can then store the corresponding input element into the output element, completing the desired streaming compaction objective.

Host Side Function:

The host-side function shown below allocates GPU memory for the input data, the result vector, flag vector, and the vector for scanning procedure output. The input data is copied to the GPU. The subsequent lines setup grid and thread blocks in a usual way. The kernel is launched, and the results vector is copied back from the GPU to the host side.

```
// Use the GPU to compact the h_data stream
void compact_stream_on_device(float *result_d, float *h_data, unsigned int num_elements)
{
    //allocate h_data on device
    float* d_h_data;
    float * d_result_d;
    int * flag;
    int * scan_output;

    cudaMalloc((void**)&d_h_data, num_elements*sizeof(float));
    cudaMalloc((void**)&d_result_d, num_elements*sizeof(float));
    cudaMalloc((void**)&flag, num_elements*sizeof(int));
    cudaMalloc((void**)&scan_output, num_elements*sizeof(int));


    cudaMemcpy(d_h_data, h_data, num_elements*sizeof(float), cudaMemcpyHostToDevice);

    dim3 thread_block(THREAD_BLOCK_SIZE, 1, 1);
    int num_thread_blocks = ceil((float)num_elements/(float)THREAD_BLOCK_SIZE);
    dim3 grid(num_thread_blocks, 1, 1);

    compact_stream_kernel<<<grid, thread_block>>>(d_h_data, d_result_d, flag, scan_output, num_elements);

    cudaThreadSynchronize();

    //fix num elements
    cudaMemcpy(result_d, d_result_d, num_elements*sizeof(float), cudaMemcpyDeviceToHost);
}
```

Device Side Function:

We utilized two kernels. The first one was provided by Dr. Kandasamy (shown below). It performs an inclusive scan operation on the g_idata vector and outputs the g_odata vector. The data types for these two vectors were changed to **int** because for our application, the only possible inputs into this scan are integer arrays. The function uses the ping-pong approach to the scanning algorithm, and it utilizes a shared array **temp** for this purpose. This array's size is twice the block size, which allows implementation of the ping-pong.

```
__device__ void scan_naive(int *g_odata, int *g_idata, int n)
{
    // Dynamically allocated shared memory for scan kernels

    //extern   __shared__  float temp[100];
    __shared__  float temp[2*THREAD_BLOCK_SIZE];
        int thid = threadIdx.x;
    int pout = 0;
    int pin = 1;

    // Cache the computational window in shared memory
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;

    for (int offset = 1; offset < n; offset *= 2)
    {
        pout = 1 - pout;
        pin  = 1 - pout;
        __syncthreads();

        temp[pout*n+thid] = temp[pin*n+thid];

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
    }

    __syncthreads();

    g_odata[thid] = temp[pout*n+thid];
}
```

In the kernel code shown below for stream compaction, given the input, output, flag, scanned flag, and number of element values, this will be the primary function to scan the array and compact the stream. The ith index was calculated by taking the block index and multiplying it by the block dimension, and adding the offset which is the thread index. The flag array is then calculated by each thread by checking whether the current element is greater than 0.0, and if so, supplying 1 to the corresponding flag index, otherwise it is a 0 meaning it's negative. __syncthreads() ensures all threads have completed calculating the flag array, where then the kernel can proceed to scanning the flag array and output the sequence that delimits where the index of the desired value is stored. The last if conditional statement checks whether the flag is 1 by each thread, and then indexes the scanned_output array which stores the index for the output array. This array gets stored the corresponding input array at the index location i.

```
// Add additional kernels here
__global__ void compact_stream_kernel(float* input, float* output, int* flag, int * scan_output, unsigned int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    //check whether each element is positive or negative
    flag[i] = (input[i] > 0.0) ? 1 : 0;

    __syncthreads();

    scan_naive(scan_output, flag, n);
    __syncthreads();


    if (flag[i] == 1) {

        output[scan_output[i] ] = input[i];
    }

}
```

Result:

The assignment description says that it is assumed that the default stream size is 1024. The following timings were obtained when testing the program with 1024 elements:

Time Serial (s)	Time Parallel (s)	Speedup
0.000013	0.000058	0.224

As the table above shows, the parallel program is slower than the serial program. However, Dr. Kandasamy informed students during the last lecture that we may not see a speedup in the parallel implementation for this particular problem, which is why the lack of speedup is not seen as a problem in our implementation. It is possible that an implementation that is flexible and can account for larger stream sizes would be able to provide speedup without any changes in the procedure.