Parallel Computing Comp 534
Swanand Sanjay Khonde - sk218

Introduction:

1) Background for this assignment:

In today's time the hunt for finding clean and sustainable energy resources is one of the most challenging tasks. Among different processes to achieve the above task, one process which is the most promising is a process that powers our sun and could potentially provide an almost limitless source of energy. Achieving controlled thermonuclear fusion on Earth requires the confinement of plasma, a hot, charged state of matter consisting of hydrogen nuclei (isotopes), within a magnetic field. One of the leading technologies for achieving such confinement is the tokamak, a doughnut-shaped fusion device.

Simulating this behavior of plasma in the tokamak is the critical aspects of fusion research. These simulations are important for understanding and predicting plasma dynamics, which are influenced by various factors like magnetic confinement, plasma turbulence, and the interactions of plasma with the reactor's walls. High-performance computing (HPC) plays an important role in this study.

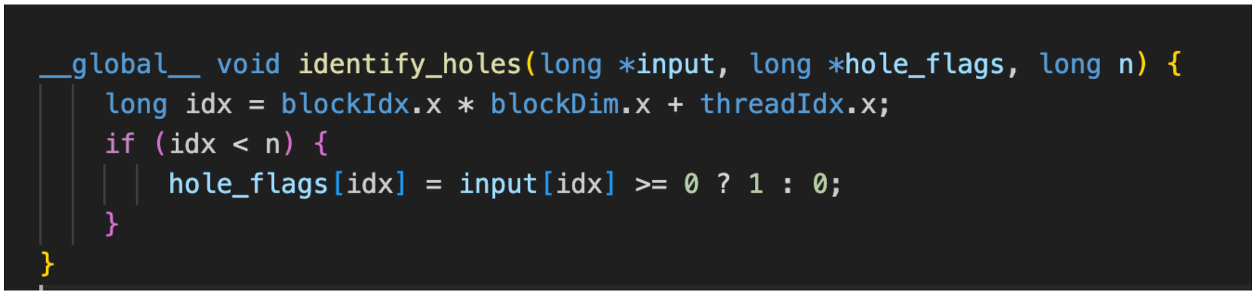
As plasma circulates within the magnetic field of a tokamak, some particles may transition between different regions of the simulation, corresponding to different sections of the tokamak or different computational resources (like GPUs). This movement results in what are termed "holes" in the data array that tracks these particles. These holes are essentially slots in the array from which particles (represented by positive numbers) have moved, leaving behind negative values to mark their departure.

The hole compaction process involves reorganizing the array to fill the holes (negative values). In this assignment, we will write a data parallel algorithm designed to efficiently compact holes within an array of particle data in a plasma simulation.

The parallel code involves 3 steps:

1) Marking the holes and non-holes in the array.
2) Performing prefix sum scan on the array which has the holes and non-hole values.
3) And finally backfill the holes.

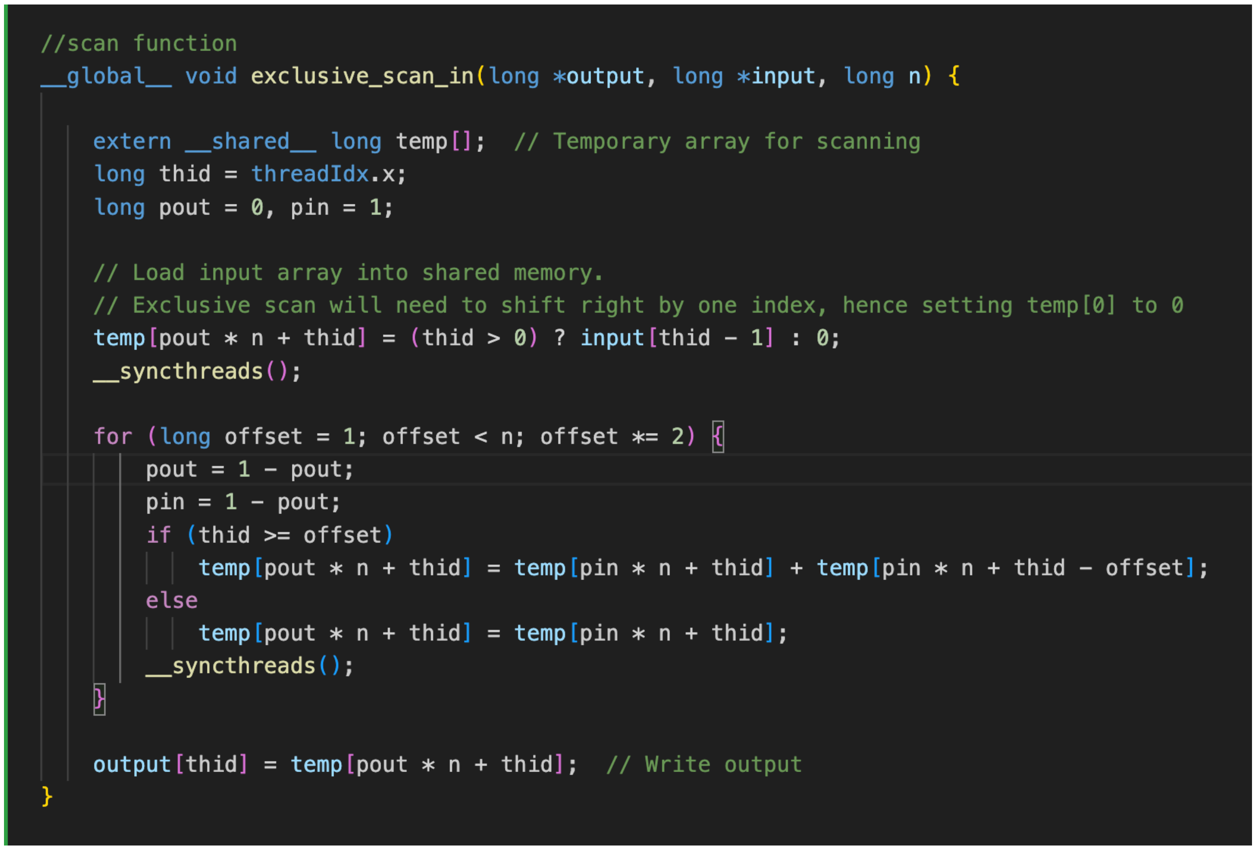
For all of these 3 steps we will write kernels in cuda.



```
__global__ void identify_holes(long *input, long *hole_flags, long n) {
    long idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        hole_flags[idx] = input[idx] >= 0 ? 1 : 0;
    }
}
```

This will create a temporary array hole_flags which will have 0 values at the indices where there is a negative number in the input array.

Scan function:

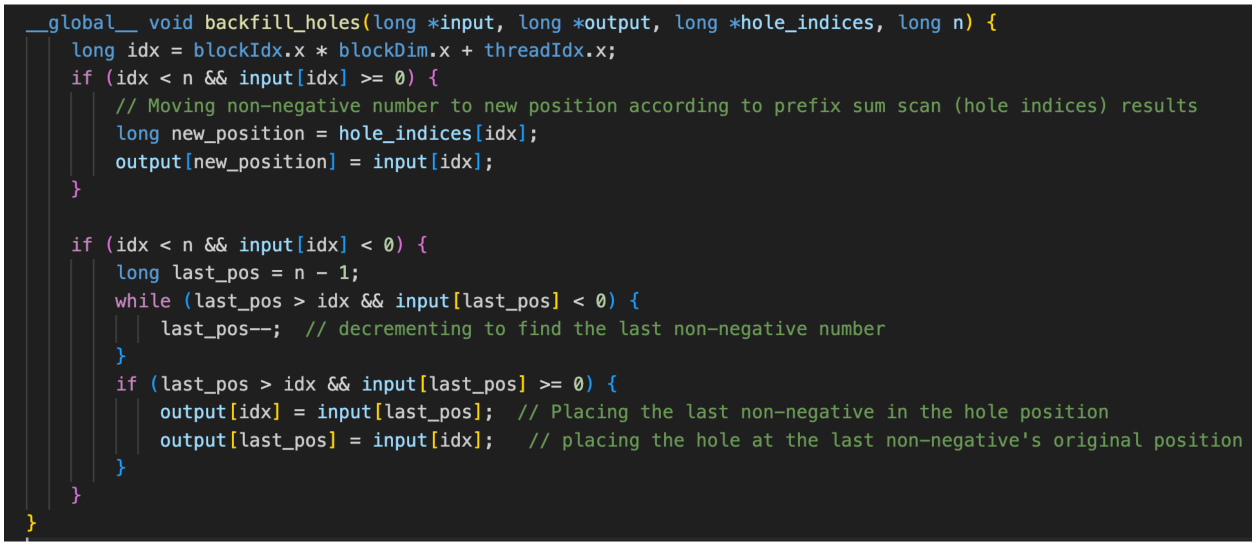


```
//scan function
__global__ void exclusive_scan_in(long *output, long *input, long n) {

    extern __shared__ long temp[];  // Temporary array for scanning
    long thid = threadIdx.x;
    long pout = 0, pin = 1;

    // Load input array into shared memory.
    // Exclusive scan will need to shift right by one index, hence setting temp[0] to 0
    temp[pout * n + thid] = (thid > 0) ? input[thid - 1] : 0;
    __syncthreads();

    for (long offset = 1; offset < n; offset *= 2) {
        pout = 1 - pout;
        pin = 1 - pout;
        if (thid >= offset)
            temp[pout * n + thid] = temp[pin * n + thid] + temp[pin * n + thid - offset];
        else
            temp[pout * n + thid] = temp[pin * n + thid];
        __syncthreads();
    }

    output[thid] = temp[pout * n + thid];  // Write output
}
```

This will calculate the prefix sum of the array as mentioned in the Nvidia cuda chapter 39 documentation.

Finally

Backfilling logic:



```
__global__ void backfill_holes(long *input, long *output, long *hole_indices, long n) {
    long idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n && input[idx] >= 0) {
        // Moving non-negative number to new position according to prefix sum scan (hole indices) results
        long new_position = hole_indices[idx];
        output[new_position] = input[idx];
    }

    if (idx < n && input[idx] < 0) {
        long last_pos = n - 1;
        while (last_pos > idx && input[last_pos] < 0) {
            last_pos--;  // decrementing to find the last non-negative number
        }
        if (last_pos > idx && input[last_pos] >= 0) {
            output[idx] = input[last_pos];  // Placing the last non-negative in the hole position
            output[last_pos] = input[idx];   // placing the hole at the last non-negative's original position
        }
    }
}
```

This kernel checks for two conditions one for non-negative numbers and one for negative numbers. When it is a non-negative number just add it to the output array on the new index which we get from the hole indices prefix scan array.

When it is a non-negative number, we check for the last position of the input array. If it is a negative number, then we decrement the last position until we get a positive number. Once we get a positive number, we place the last non-negative in the hole position and then place the hole at the last non-negatives original position.

Compare the time to backfill a sequence of 400 million elements with 10% holes serially on the CPU with the time to do so on a GPU.

Results:
Time for serial compaction on 10 percent of N: **178 ms**

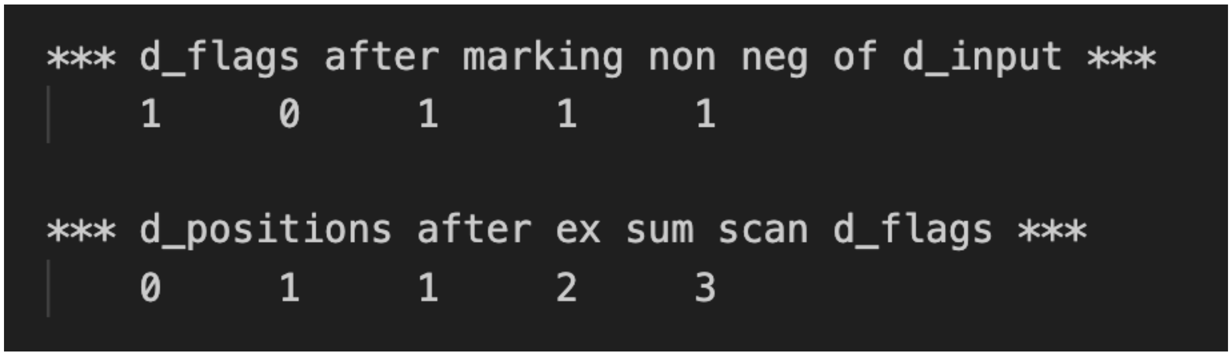
Time for parallel compaction on full N: **14 ms**

Explaining Scan Operation

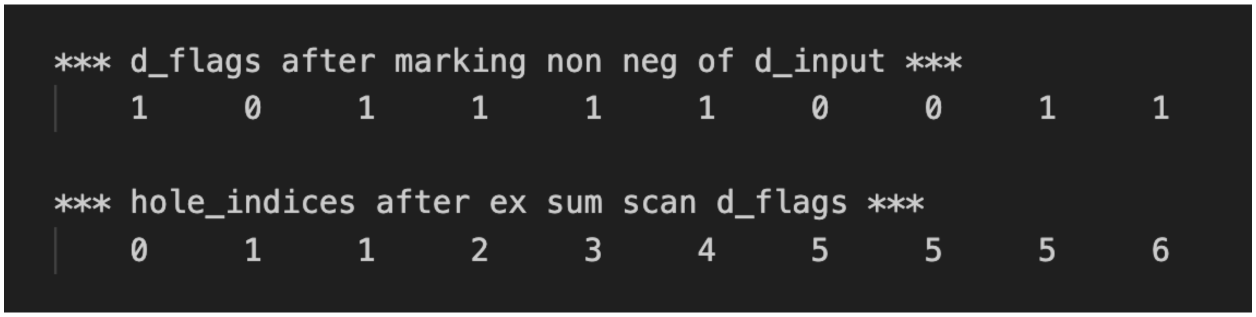
The scan kernel performs an "exclusive scan" on an array. The scan generates a new array where each element at index ‘i’ is the sum of all elements from index 0 to ‘i-1’ of the original array. Each thread in this kernel loads an element from the input array into the temp shared memory. The first element of temp is explicitly set to 0 because an exclusive scan does not include the element at the current index in its sum according to Nvidia documentation. After initializing temp, the threads within the block synchronize with each other using __syncthreads(). This ensures that all threads have completed their memory writes to temp before any reading begins, preventing data corruption. We use a for-loop to iteratively calculate the sum of elements. The variable offset doubles each iteration, representing the stride of indices each thread will add from temp. Within the loop, threads use two alternating indexes (pout and pin) to read from and write to temp without interfering with each other. if the thread's index is greater than or equal to offset, it adds the value from a previous position in temp to its current position. This accumulates the sum of elements up to that index. After completing all iterations of the loop, each thread writes its final result from temp to the output array. This result is the cumulative sum of elements excluding the current element's value.

Verification that the scan works:

As per professors’ comments on piazza, I ran my scan operation on a small array to see if it works.



```
*** d_flags after marking non neg of d_input ***
    1     0     1     1     1

*** d_positions after ex sum scan d_flags ***
    0     1     1     2     3
```



```
*** d_flags after marking non neg of d_input ***
    1     0     1     1     1     1     0     0     1     1

*** hole_indices after ex sum scan d_flags ***
    0     1     1     2     3     4     5     5     5     6
```

Scan performance:

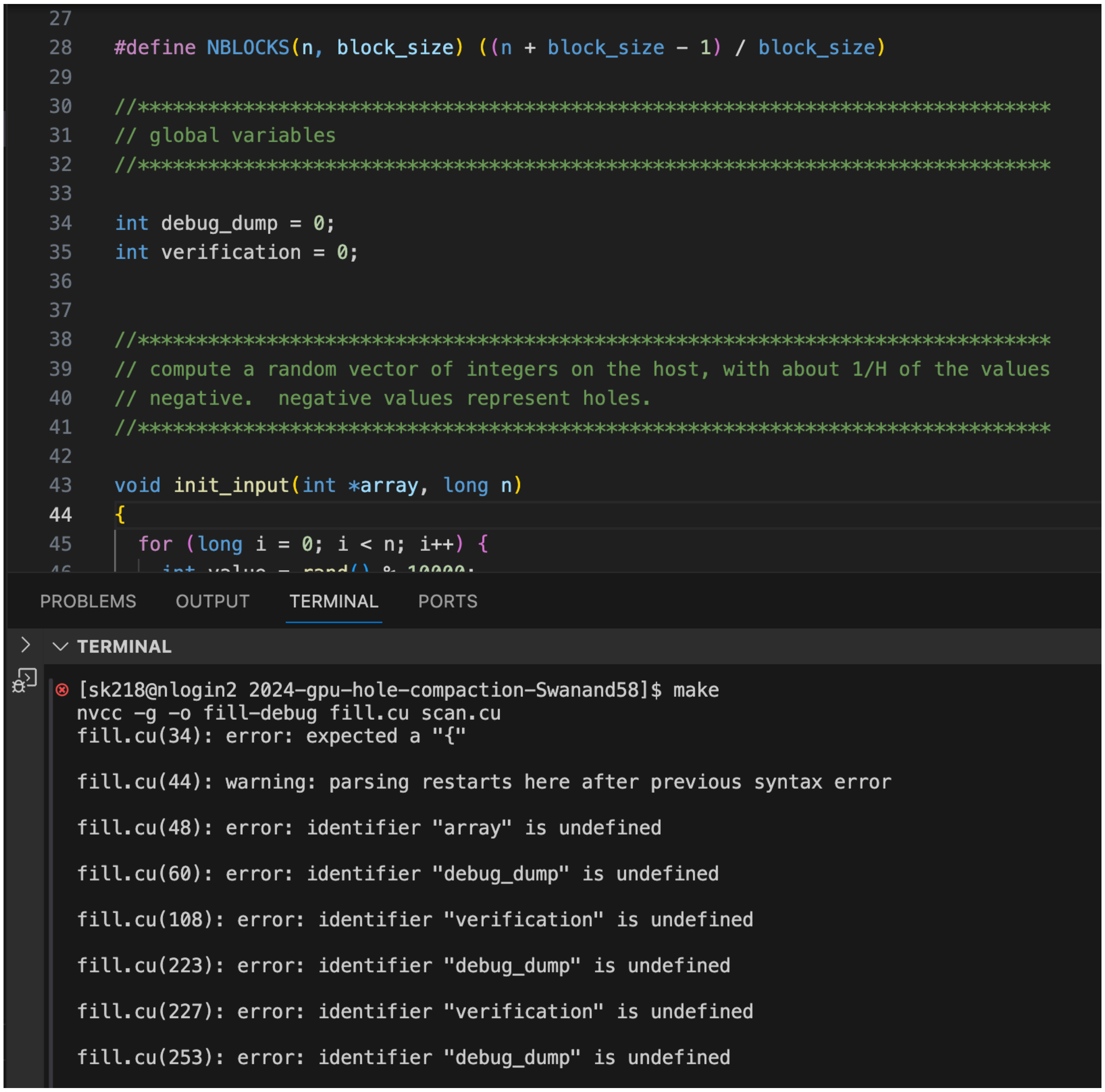
scan time provided by the professor for CUB:

scan time = 0.004624 seconds

scan time for my sum scan function:

could not able to generate the time for my scan as the code did not run for 400 million elements. It kept giving me errors.(cgroup out-of-memory handler.)

I was getting weird errors while development which I posted on piazza. Had to clone the github repo again and again.



```
27
28  #define NBLOCKS(n, block_size) ((n + block_size - 1) / block_size)
29
30  //****************************************************************************
31  // global variables
32  //****************************************************************************
33
34  int debug_dump = 0;
35  int verification = 0;
36
37
38  //****************************************************************************
39  // compute a random vector of integers on the host, with about 1/H of the values
40  // negative.  negative values represent holes.
41  //****************************************************************************
42
43  void init_input(int *array, long n)
44  {
45    for (long i = 0; i < n; i++) {
46      int value = rand() % 10000;
```

```
[sk218@nlogin2 2024-gpu-hole-compaction-Swanand58]$ make
nvcc -g -o fill-debug fill.cu scan.cu
fill.cu(34): error: expected a "{"

fill.cu(44): warning: parsing restarts here after previous syntax error

fill.cu(48): error: identifier "array" is undefined

fill.cu(60): error: identifier "debug_dump" is undefined

fill.cu(108): error: identifier "verification" is undefined

fill.cu(223): error: identifier "debug_dump" is undefined

fill.cu(227): error: identifier "verification" is undefined

fill.cu(253): error: identifier "debug_dump" is undefined
```

Did not get enough time to develop the actual kernel logic. Also faced several issues with running the code on gpu.

When I ran my code I got this kind of output

```
/var/spool/slurm/job7928966/slurm_script: line 9:  7633 Killed             ./fill-opt -n
400000000 -v

real    0m25.700s
user    0m21.697s
sys     0m3.816s
```

```
slurmstepd: error: Detected 1 oom-kill event(s) in step 7928966.batch cgroup. Some of
your processes may have been killed by the cgroup out-of-memory handler.
```

I saw similar post on piazza updated the sbatch file but still got the same error. Couldn’t test properly. ☹. The above result with the time is without verification and dump.