GPU Programming Basics

Northeastern University
NUCAR Laboratory

for all

Julian Gutierrez
David Kaeli

Hands-on Lab #3

Objective

- Test the advantage of using shared memory for a convolution filter.

Part 2: Shared Memory

First of all, we need to request an interactive node, we can use the following command. Once a resource is available, you will be connected automatically into the compute node. If you are having trouble with this, go back to Lab 1 and review part 1.

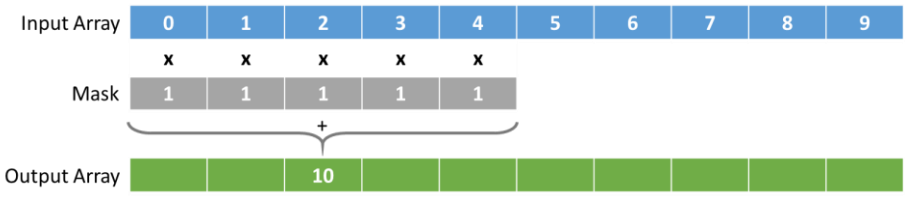
```
srun --pty --export=ALL --reservation=gpu-class --partition=gpu --task
s-per-node 1 --nodes 1 --mem=2Gb --time=02:00:00 /bin/bash # Reserve t
he node
```

Copy the following folder to your scratch directory (or folder of your choosing):

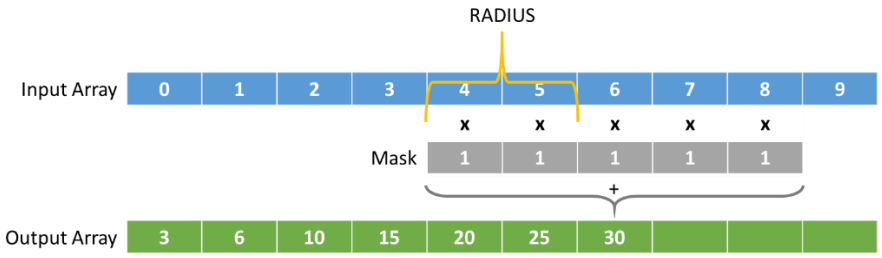
```
cd /scratch/`whoami`/GPUClass18
cp -r /scratch/gutierrez.jul/GPUClass18/HOL3/ .
cd HOL3/
```

In this part, we are going to focus on the usage of shared memory. As we saw in class, shared memory can be seen as a cache memory that you can control what is being stored in it. Given that it is close to the processing cores, the latency is low, and the bandwidth is high when comparing it to global memory. Overall, shared memory will present a great opportunity to improve the performance if used correctly, but as you will see, this will not be easy.

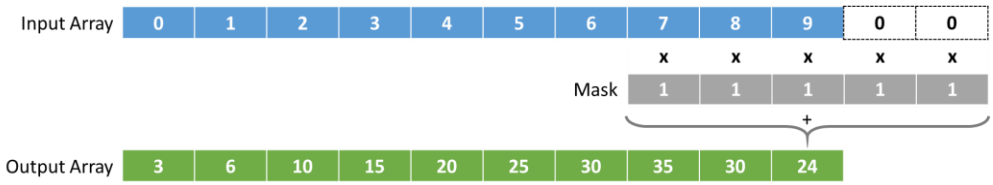
The problem we will be looking into in this section is a 1D-convolution filter. These algorithms are commonly used in signal and image processing. Mathematically, they apply a mask on top of the data (multiplier value) and apply a reduction summation as shown in the following figure.



Element 3 from the output array is calculated by applying the mask (multiplier) to the input data centered in the same index and then adding the resulting values together. Given that the mask we are using for this example is filled with ones, we can see the output as the summation of values surrounding the index by a radius of 2, as demonstrated in the following figure.



An important case to consider for this example is how we handle the **boundary conditions**. To simplify the problem, what we will do is to set the values to 0 where we meet the boundaries as demonstrated in the following figure.



In this example, to calculate the final element of the array, the mask is applied to elements that do not exist in the input array. These "ghost" cells are also known as halo cells. Given these elements don't exist, we can assign values to them depending on the algorithm we are implementing. Common approaches are giving them a fix value, or mirroring. For this example, we will fill them up with 0s.

The naïve way of implementing this algorithm on the CPU is shown in the following code snippet.

```
for (int i = 0; i < vector_size; i++) {
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
           out_cpu[i] += (i + offset >= 0 && i + offset < vector_size)
                                    ? in_cpu[i + offset] : 0.0;
}
```

For each element in the output array, we iterate from the elements to the left (starting at –RADIUS) to the elements to the right (ending at +RADIUS) and adding those values together. The "(X) ? Y : Z" combination is an if statement that allows us to handle the boundary conditions. If we add the offset to our index and we are still inside the input array, add that value, or else, add 0.0.

The idea is to enhance the performance of the algorithm through the use of shared memory, and any other optimization you can think of. Read the main function code and the basic kernel function, and think of ways to improve it.

Baseline code is the following file:

`./stencil.cu`

File you need to modify is:

`./stencil_shared.cu`

Please look at the `stencil.cu` file and understand how everything is working. The code works similarly to the one from our previous lab. If you wish to test the code out, you can do so by compiling it and executing it using the following commands:

```
nvcc -O3 -arch=sm_35 -lineinfo stencil.cu -o stencil

sbatch exec.bash
```

Once you have understood how the code works, we can work on implementing this algorithm using shared memory. Please feel free to ask any questions regarding the code at this moment.

Implementation with Shared Memory

1. The first thing we need to do is decide on how much shared memory space we require. In line 9 of the code, specify the size for the array considering all the data required to process each element.
    a. Remember that shared memory is memory accessible by all threads in the block. This means we want to copy all the data the block would need. This includes copying the halo cells to the left and right of our current section (block) of the array.
2. In line 20, copy the global data indexed by gindex into shared memory using lindex.
3. Now, that we have filled out the inner part of the array in shared memory, the tricky part is how do we fill out the boundaries.
    a. Remember, if it's a block in the middle of the array, we need to copy the values for this halo cells from the global array. If our block is in the boundaries of the vector, we can store 0s in them. We will do all of this in lines 25 and 26.
    b. We will use the first RADIUS threads in our block to copy the data that we need for either side of the block.
    c. First, let us copy the left halo cells. We index into them in shared memory using lindex – RADIUS. What value should we use from the global array to store in that position?
        i. Hint, to do this, we should add a check to make sure such index falls inside the array. If not, what should we store?
    d. Now, let us copy the right halo cells. We need to the same analysis but this time we index into shared memory using lindex + BLOCK_SIZE.
4. In line 30, why do we need to store 0.0 if the global index is larger than our vector size? In which scenario would this affect our result? Try visualizing it by drawing a case where this could happen. Hint: vector_size % block_size != 0
5. Now that we have moved the data to shared memory, we need to guarantee that all threads have completed before we start using the data. What command do we use for this and is it important? Add this command in line 33.
6. Now we can collect the result from applying the stencil and storing it into a temporary variable. We do this to avoid writing to global memory after each summation. Fill out line 39 with the appropriate code to implement the same functionality as the sequential version.
7. Finally, write the temporary result into global memory so that we can copy the result back to the CPU at line 42.

Results

Now that we have implemented the shared memory version, we can compile it using the following command:

```
nvcc -O3 -arch=sm_35 -lineinfo stencil_shared.cu -o stencil_shared
```

Make sure that the code is producing the correct results by running the application with different vector sizes. You can do this by running `sbatch exec_shared.bash`. Also, run cuda-memcheck using `sbatch memcheck_shared.bash`, as we learned before, to make sure we are not accessing any illegal memory addresses. Once this is done, proceed to fill out the following tables (which have been prefilled to speed up the recollection of data).

If time is limited, run the code using the same Block Size (512).

Radius Size	8			
Vector Size	**CPU (ms)**	Block Size	**Naïve GPU (ms)**	**Opt GPU (ms)**
1000	0.05	256	0.05	
		512	0.07	
		1024	0.06	
100000	4.08	256	0.10	
		512	0.12	
		1024	0.11	
1000000	40.82	256	0.50	
		512	0.48	
		1024	0.49	
10000000	414.45	256	4.29	
		512	4.29	
		1024	4.35	
100000000	3199.01	256	42.03	
		512	41.94	
		1024	42.90	

- What is the speed up for the Naïve GPU and the optimized GPU compared to the sequential version?
- Is it worth the development time?
- Run NVPROF using `sbatch nvprof.bash` and `sbatch nvprof_shared.bash` and investigate which metrics improved compared to the naïve implementation.

The next experiment will involve varying the radius size to see if shared memory is always worth it for this application. Remember, there is an overhead associated with shared memory, from having to synchronize between all threads, to adding more if statements. This overhead sometimes can be higher than the benefits from shared memory.

By increasing the radius of the stencil, we increase how much data is being shared for each calculation. If there is still time, fill out the tables with all the experiments. If not, just do a few of the cases (first and last should be ok).

To do this, modify the code define used to set the variable for radius and recompile the code, then run the command using `sbatch radius.bash`. You will have to change the code recompile and execute each time for each radius.

Vector Size	100000000			
Radius Size	**CPU (ms)**	Block Size	**Naïve GPU (ms)**	**Opt GPU (ms)**
0	135.66	512	8.37	
1	214.61	512	9.38	
2	451.66	512	12.76	
4	735.54	512	22.23	
8	3501.82	512	41.92	
16	6622.34	512	81.55	
32	15528.04	512	161.19	

- Look at the solution. There is an additional command added. Investigate what it does (line 70).
- What conclusions can you get from the performance from using shared memory? Is shared memory always useful?
- Extra homework: Do you think changing the data to floats would help improve the performance? Try to explain why yes or no (because of computation? Because of memory?).
    - You can run the experiment by using the following command to substitute all double words to floats in the code. I recommend creating a new file before doing this edition. Does the change affect the sequential version as well?

```
sed –i "s/double/float/g" <file>
```

Note: Please remember to ask as many questions as possible. I am here to help as much as we can.