R. N. Navagamuwa
120418H

Menaka L. Sirisena
120628C

CS4532 - Concurrent Programming
Lab 2

1.

a. **Quad-Core processor**

Latency = 50 ns (Since latency is the amount of time to complete a one task)

Throughput = Number of Elements/Total Time
= 200 (elements)/ (200 x 50) (ns)
= 0.08 elements per ns

Assumptions:

All the elements take the same amount of processing time to process.Context switching time was negligible or not happening during the task execution.

b. **GPU with 440 cores**

Latency = 50 ns * 1.6
= 80 ns

Throughput = Number of Elements/Total Time
= 200 (elements)/ 80 (ns)
= 2.5 elements per ns

Assumptions:

All the elements take the same amount of processing time to process.All 200 elements take 80 ns for process, since GPU kernel can uses 200 cores out of 440 cores.

2.

a. **blockIdx**

Gives the block index of each block.

b. **cudaMemcpy( )**

CPU copies input data from CPU to GPU.

c. **cudaMalloc( )**

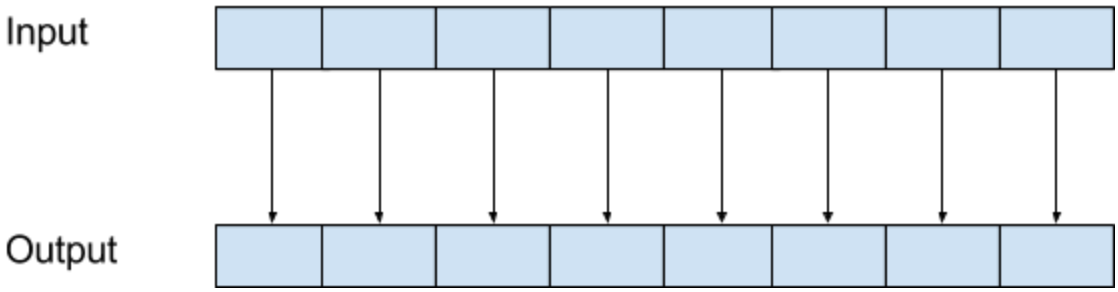
CPU allocates storage on GPU.

d. **blockDim( )**

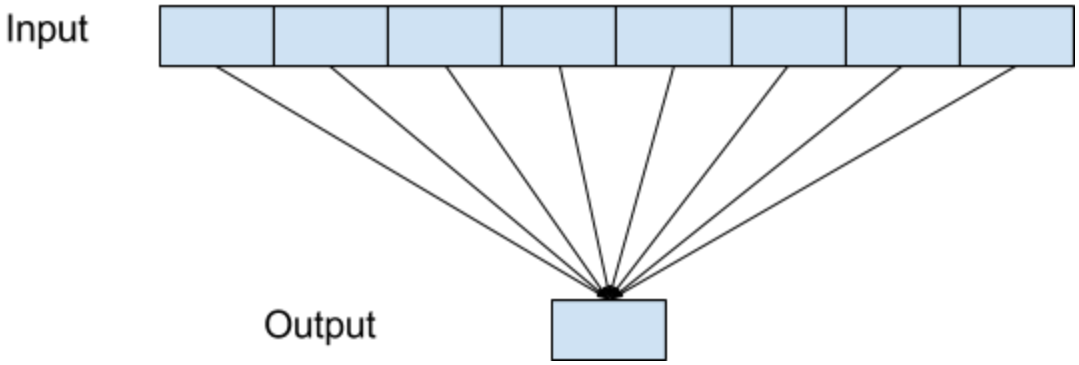
Gives the number of threads in a block, in the particular direction.

3. Number of blocks = $3 \times 4 = 12$
   Number of threads/block = $4 \times 8 = 32$
   Number of total blocks = $32 \times 12 = 384$

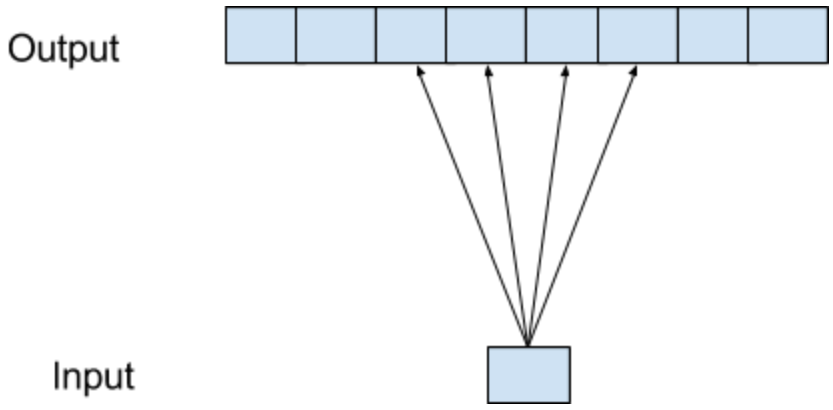
4.
   a. Map
      This is a parallel communication pattern which has one to one correspondence between input and output. Therefore Map provide an output for each input.



   b. Gather
      This is a parallel communication pattern which there is many to one correspondence between input and output. It takes inputs from different locations and create an output element.

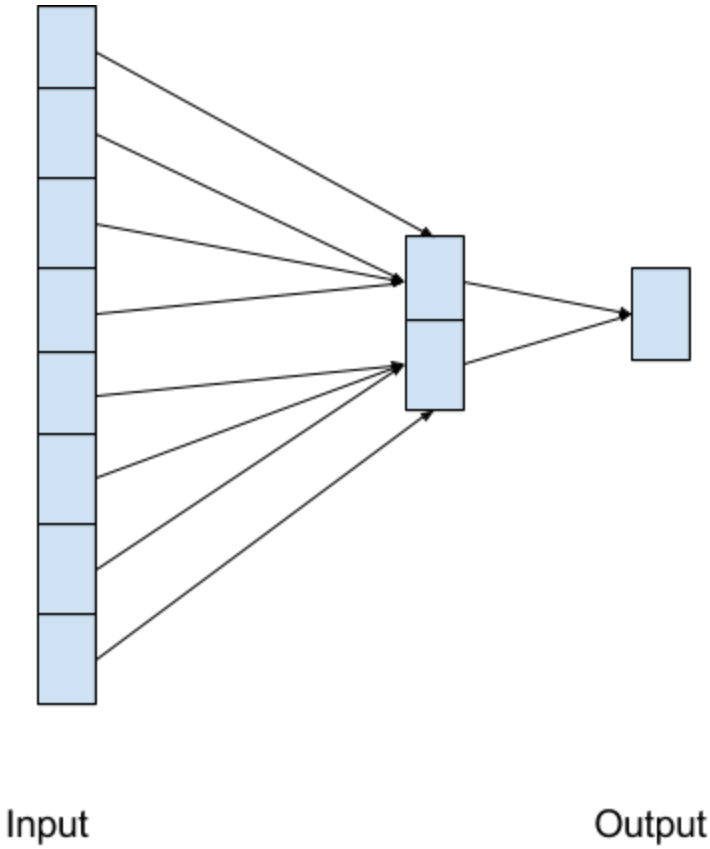


   c. Scatter
      In contrast to Gather, Scatter is a communication pattern which there is one to many correspondences between input and output. (One input can scattered though the memory)



d. Reduce

This is a parallel communication pattern which there is all to one correspondence between input and output. For example, adding up all the elements in one array and put input in one output.



5. **Pros:**
    - Because of the flexibility in allocating thread blocks, hardware can run the program efficiently.
    - Scalability – Because assumptions are not made on where a thread block will be executed or how many blocks may be running at the same time, the program can be scaled down to a GPU running with a single SM (Streaming Multiprocessor) or to a massive GPUs on a supercomputer. It also applies to future GPUs. Cuda code will scale to larger GPUs.

    Cons :
    - No assumptions on what block will run on what SM.
    - No communication between blocks. Can lead to “dead lock” in parallel computing. (Where one block requires the results of another block. But that block has already been executed and exited)