Write the CR kernel for the version 1 that runs the GPU:

Complete code in the file: cr_cuda.cu (version 1)

Write the invocation of the kernel and the code where *A*, *B*, *C* and *D* are sent from the CPU to the GPU, and *X* is sent from the GPU to the CPU for the vesion 1:

Complete code in the file: cr_cuda.cu (version 1)

Compilation line and script to send to the queue system

```
nvcc -o CR_cuda CR_cuda.cu  -Xcompiler -fopenmp -lcudart
```

```
#!/bin/bash
#SBATCH -o %x-%J.out
#SBATCH -e %x-%J.error

#SBATCH -J CR_cuda       # Job name
#SBATCH -o CR_cuda.o%j   # Name of stdout output file(%j expands to jobId)
#SBATCH -e CR_cuda.o%j   # Name of stderr output file(%j expands to jobId)

#SBATCH --time=0-00:05:00 #requested time to run the job
#SBATCH -c 32 #(64 cores per job)
#SBATCH -t 00:25:00 #(25 min of execution time)
#SBATCH --mem=16GB #(4GB of memory)

#module --ignore-cache avail
module load cesga/2020 cuda-samples/11.2

# Run the CUDA program
#OMP_NUM_THREADS=8 ./CR_cuda

n_values=(8 16 32 64 128 256 512 1024 2048 4096)

# Array of GPU types to iterate over
gpu_types=("t4" "a100")

# Iterate over GPU types and n values
for gpu in "${gpu_types[@]}"; do
    export SLURM_GPUS_ON_NODE=$gpu  # Dynamically set GPU type for execution
    for n in "${n_values[@]}"; do
        B=$((2**24 / n)) # Calculate B as 2^24 / n
        echo "Running CR_cuda with n=$n, B=$B, and GPU=$gpu"
        OMP_NUM_THREADS=8 ./CR_cuda $n
    done
done
```

Complete this table using a Turing of FT-III for the version 1:

N	B	Shared Memory per block	Execution Time (only kernel) in seconds	Threads per block	Number of blocks	Speedup with respect to CPU version
8	209715	128b	0.000005	8	2097152	1.6
16	104857	256b	0.000008	16	1048576	1
32	524288	512b	0.000004	32	524288	2
64	262144	1024b	0.000006	64	262144	1.33333333
128	131072	2048b	0.000007	128	131072	1.14285714
256	65536	4096b	0.000006	256	65536	1.33333333
512	32768	8192b	0.000009	512	32768	0.88888888
1024	16384	16384b	0.000005	1024	16384	1.6
2048	8192	32768b	0.000008	2048	8192	1
4096	4096	65536b	0.000009	4096	4096	0.88888888

Complete this table using a Ampere of FT-III for the version 1:

N	B	Shared Memory per block	Execution Time (only kernel) in seconds	Threads per block	Number of blocks	Speedup with respect to CPU version
8	209715	128b	0.000007	8	2097152	1.14285714
16	104857	256b	0.000010	16	1048576	0.800000

32	524288	512b	0.000007	32	524288	1.1428571
64	262144	1024b	0.000009	64	262144	0.8888888
128	131072	2048b	0.000005	128	131072	1.6
256	65536	4096b	0.000005	256	65536	1.6
512	32768	8192b	0.000005	512	32768	1.6
1024	16384	16384b	0.000007	1024	16384	1.1428571
2048	8192	32768b	0.000003	2048	8192	2.6666666
4096	4096	65536b	0.000004	4096	4096	2

If any value of N cannot be executed in your version, indicate the reason why

All of then could be executed.

If you use a synchronization barrier, justify the reason for it:

Since each thread processes a part the arrays A, B, C and D and store in the shared memory, and afterwards the results of one thread may be required by other threads in subsequent iterations, synchronization ensures that all threads complete their updates to shared memory before any thread accesses those updates.

Complete this table using a Turing of FT-III for the version 2:

N	B	Shared Memory per block	Execution Time (only kernel) in seconds	Threads per block (x,y)	Number of blocks	Speedup with respect to CPU version
8						
16						
32						
64						
128						
256						
512						
1024						
2048						
4196						

Complete this table using a Ampere of FT-III for the version 2:

N	B	Shared Memory per block	Execution Time (only kernel) in seconds	Threads per block (x,y)	Number of blocks	Speedup with respect to CPU version
8						
16						
32						
64						
128						
256						
512						
1024						
2048						
4196						