A3Q2

Name: Viswa Nihar Nukala

UB Name: viswanih

UB Number: 50414392

This code is to run a Gaussian Kernel on an array using CUDA.

First, we copy the data to the devices and then we start retrieving parts of data from the memory and load it into the shared memory. This is done by all the threads in the block accessing one element and hence 1024 elements at once. We then compute Gaussian Kernel on this data. This happens it iteratively (block by block) till all the elements are read.

Following is the make file for the code to run:

```
CXX=nvcc
CXXFLAGS=-std=c++14 -O3 -o a3

all: a3

a3: a3.cu a3.hpp
    $(CXX) $(CXXFLAGS) a3.cu

clean:
    rm -rf a3
```

Following is the snapshot of my slurm script to run the code:

```
#!/bin/bash

#SBATCH --output=%j.stdout
#SBATCH --error=%j.stderr
#SBATCH --job-name="changeme"
#SBATCH --gres=gpu:tesla_v100-pcie-16gb:1
#SBATCH --nodes=1
```

```
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:45:00

module load gcc/6.3.0
module load cuda

make clean

make

./a3 100000 0.2
./a3 200000 0.2
./a3 400000 0.2
./a3 800000 0.2
./a3 1600000 0.2
./a3 3200000 0.2
./a3 6400000 0.2

make clean
```

Here I am increasing the size of n to see the time it takes to complete the computations.

Following are the GPU specifications:

Device 0: "Tesla V100-PCIE-16GB"

CUDA Driver Version / Runtime Version	11.4 / 9.2
CUDA Capability Major/Minor version number:	7.0
Total amount of global memory:	16160 MBytes (16945512448 bytes)
(080) Multiprocessors, (064) CUDA Cores/MP:	5120 CUDA Cores
GPU Max Clock rate:	1380 MHz (1.38 GHz)
Memory Clock rate:	877 Mhz
Memory Bus Width:	4096-bit
L2 Cache Size:	6291456 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers	1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total shared memory per multiprocessor:	98304 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 7 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Device supports Managed Memory:	Yes
Device supports Compute Preemption:	Yes
Supports Cooperative Kernel Launch:	Yes
Supports MultiDevice Co-op Kernel Launch:	Yes

Device PCI Domain ID / Bus ID / location ID: 0 / 59 / 0

Compute Mode:

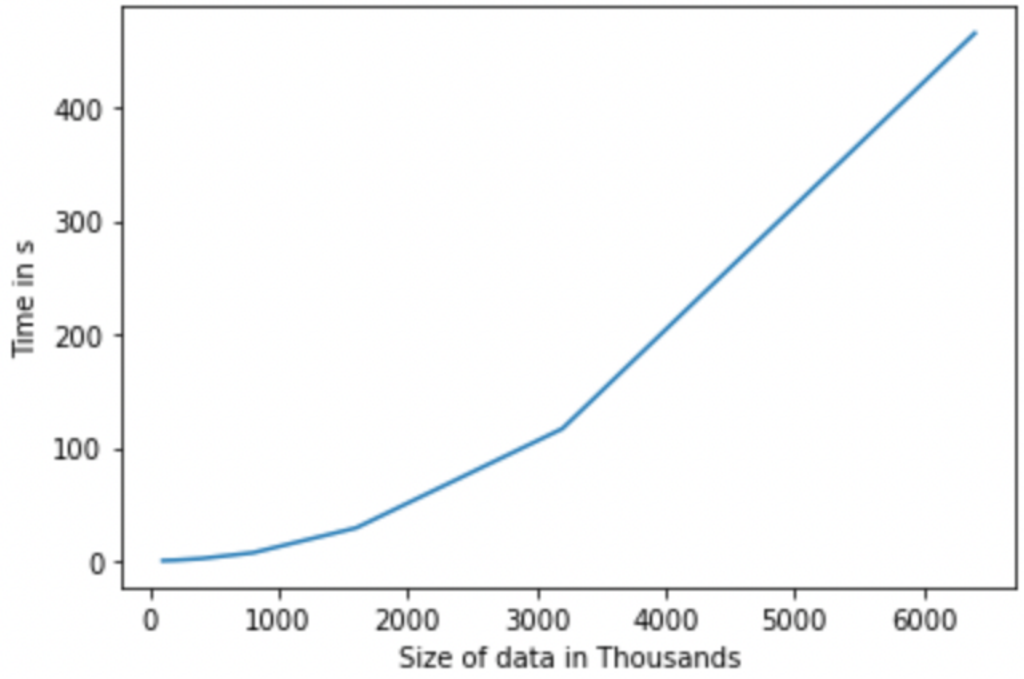
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Following are the readings of my data with h=0.2:

Size of n	h	Time 1	Time 2	Trial 3	h	Time
100000	0.2	0.497994s	0.53256s	0.362592s	0.001	0.390385s
200000	0.2	0.867765s	0.92738s	0.863137s	0.001	0.915779s
400000	0.2	2.31611s	2.30731s	2.31454s	0.001	2.35921s
800000	0.2	7.45076s	7.41008s	7.42244s	0.001	7.4546s
1600000	0.2	29.3518s	29.2282s	29.3084s	0.001	29.2867s
3200000	0.2	116.823s	116.829s	116.654s	0.001	116.614s
6400000	0.2	465.633s	465.513s	465.509s	0.001	465.559

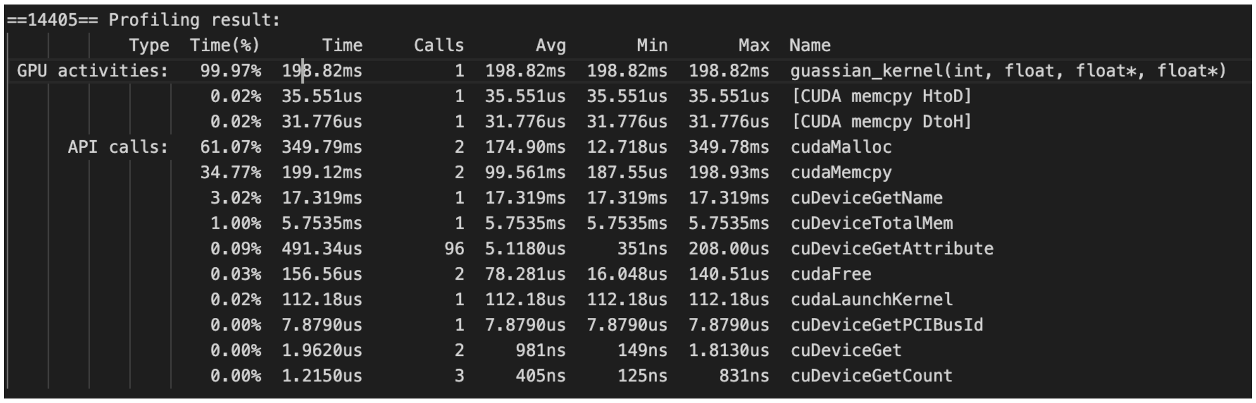
We can see that as we increase the value of n, the time keeps increasing.

Following is the graph for the above data with h=0.2:



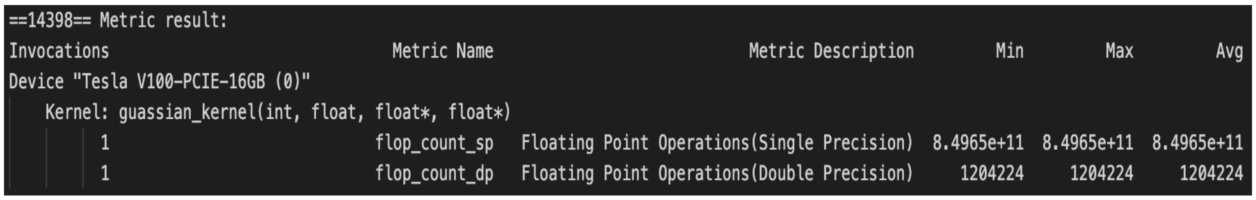
I have used nvprof to gather the following data:

The following is when nvprof is used without any attributes:



```
==14405== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.97%  198.82ms         1  198.82ms  198.82ms  198.82ms  guassian_kernel(int, float, float*, float*)
                    0.02%  35.551us         1  35.551us  35.551us  35.551us  [CUDA memcpy HtoD]
                    0.02%  31.776us         1  31.776us  31.776us  31.776us  [CUDA memcpy DtoH]
      API calls:   61.07%  349.79ms         2  174.90ms  12.718us  349.78ms  cudaMalloc
                   34.77%  199.12ms         2  99.561ms  187.55us  198.93ms  cudaMemcpy
                    3.02%  17.319ms         1  17.319ms  17.319ms  17.319ms  cuDeviceGetName
                    1.00%  5.7535ms         1  5.7535ms  5.7535ms  5.7535ms  cuDeviceTotalMem
                    0.09%  491.34us        96  5.1180us     351ns  208.00us  cuDeviceGetAttribute
                    0.03%  156.56us         2  78.281us  16.048us  140.51us  cudaFree
                    0.02%  112.18us         1  112.18us  112.18us  112.18us  cudaLaunchKernel
                    0.00%  7.8790us         1  7.8790us  7.8790us  7.8790us  cuDeviceGetPCIBusId
                    0.00%  1.9620us         2     981ns     149ns  1.8130us  cuDeviceGet
                    0.00%  1.2150us         3     405ns     125ns     831ns  cuDeviceGetCount
```

We can see that in the GPU activities, most of the time is taken by the guassian_kernel function.
Following are the flops (Floating Point Operations) counts for both single-precision and double-precision:



```
==14398== Metric result:
Invocations                               Metric Name                            Metric Description         Min         Max         Avg
Device "Tesla V100-PCIE-16GB (0)"
    Kernel: guassian_kernel(int, float, float*, float*)
          1                             flop_count_sp   Floating Point Operations(Single Precision)  8.4965e+11  8.4965e+11  8.4965e+11
          1                             flop_count_dp   Floating Point Operations(Double Precision)     1204224     1204224     1204224
```

We can calculate the performance from the above data using:

Performance = FLOPS (SP) / Time (in s)

Hence the performance (of the code) is around 4.5 TFLOPS.

The performance given by the Tesla V100 for single point precision is around 14 FLOPS.

Comparing it to the performance of the code, it is around 35% efficient.

Conclusion:

The code isn’t efficient as the number of FLOPS is almost only 35% which can be improved.