Homework 5

Parallel Distributed Num Algorithms

Name: Saurabh Kumar
UIN: 926009924

1. Code submission

Ans: The parallelized code using CUDA has been submitted on eCampus. The name of the file is main.cu

1.1 Steps to Compile & Execute

Ans: The submission contains the following files:

a. **main.cu :** The parallelized source code using CUDA to implement Gaussian Process Regression.

b. **grid.job :** Job file for submission of the parallel job to ada cluster. The job file has commands to run the parallel code for 1, 2, 4, 10 & 20 threads automatically.

c. **run.sh :** A bash script that compiles the CUDA code and submits it to the ada grid. It has commands to set the environment in the shell for CUDA compilation and job submission.

d. **Command to run :** ***./run.sh*** (This command will automatically submit the job for the following values of m:

m = 4, m=8 & m=16

1.2 Command to Compile & Run

-> **./run.sh**

2. Describe your strategy to parallelize the algorithm for a single multiprocessor of the GPU. Discuss any design choices you made to improve the parallel performance of the code.

Ans:

Strategy

My strategy to parallelize the algorithm for a single multiprocessor of the GPU is as follows:

- I just used the thread Id while implementing the CUDA functions instead of also taking block dimensions and grid dimensions into account.
- The initialization of grid points and observed data values using equations (1) and (2) have not been parallelized as per requirement. These values were calculated on the host.
- The output of initializations are then copied onto the device memory using CUDA functions.
- The matrix K' is then computed on the device.
- The above step is followed by the Cholesky factorization of K' on the device itself.
- All the parallel processes have been scheduled to be executed onto the threads such that if the number of threads available are t, the first t processes would be run in parallel and then the next t threads and so on.
- In the Cholesky factorization function, all the threads are synchronized using the block level synchronization barrier __syncthreads() at every stage.
- Then the solver function computes the solution of K'z=f on the device using the factors computed in the Cholesky step.
- Finally, the device_predict_value function computes the final predicted value computing k (n x 1 vector) and using the function $f(x,y) = k^T z$
- Inside every global function executed on the device, the device memory (even for 2 Dimensional matrices) was manipulated in the form of 1 D array.

Design Choices

Some of my design choices for improved efficiency are as follows:

- I combined the 2 steps in the computation of K' = (tI + K) where t is the noise parameter into a single step. I achieved this by adding 0.01 to the diagonal elements of the matrix i.e in cases where x & y co-ordinates of the matrix were the same.

- I used the Cholesky factorization technique to compute the solution instead of LU factorization which is more efficient for symmetric matrices.
- The 2D matrices on the device were stored in row major order manipulated as a 1D array. This was done to gain some efficiency via locality of reference considering the pattern of memory reads.
- Used shared memory in a function using the construct __shared__ . Loaded data from device memory to shared memory as the threads access shared memory much faster than device memory.
- I tracked down the execution time for Factorization, solver and overall process and tried to minimize it in various stages of development.

3. Compute the flop rate you achieve in the factorization routine and in the solver routine. Compare this value with the peak flop rate achievable on the processor, and estimate the speedup obtained over one core and the corresponding efficiency/utilization of the cores on the device. You may choose appropriate values for the grid size to study the features of your implementation.

Ans: Here is a sample profiling output (using nvprof) from my program for m=10 and r^*(x=3 , y=3):

```
==32327== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.15%  71.120ms         1  71.120ms  71.120ms  71.120ms  device_run_chol(int, double*, double*, double*)
                    4.98%  3.7652ms         1  3.7652ms  3.7652ms  3.7652ms  device_run_solver(int, double*, double*, double*, double*, double*)
                    0.80%  607.40us         1  607.40us  607.40us  607.40us  device_compute_K(int, int, double*, XY*)
                    0.06%  42.849us         1  42.849us  42.849us  42.849us  device_predict_value(int, int, double, double, double*, double*, double*, XY*)
                    0.00%  3.2640us         2  1.6320us  1.6000us  1.6640us  [CUDA memcpy HtoD]
                    0.00%  2.3040us         1  2.3040us  2.3040us  2.3040us  [CUDA memcpy DtoH]
      API calls:   76.41%  248.86ms         8  31.107ms  7.1440us  248.49ms  cudaMalloc
                   23.18%  75.484ms         3  25.161ms  45.320us  71.668ms  cudaEventSynchronize
                    0.13%  439.62us        94  4.6760us     176ns  173.58us  cuDeviceGetAttribute
                    0.13%  428.97us         1  428.97us  428.97us  428.97us  cudaGetDeviceProperties
                    0.04%  135.30us         4  33.825us  16.888us  79.560us  cudaLaunch
                    0.03%  113.06us         1  113.06us  113.06us  113.06us  cuDeviceTotalMem
                    0.02%  50.826us         2  25.413us  15.953us  34.873us  cudaMemcpy
                    0.01%  41.447us         1  41.447us  41.447us  41.447us  cuDeviceGetName
                    0.01%  39.745us         1  39.745us  39.745us  39.745us  cudaMemcpyFromSymbol
                    0.01%  37.319us         6  6.2190us  4.0870us  14.372us  cudaEventRecord
                    0.01%  20.905us         3  6.9680us  2.0450us  16.563us  cudaEventElapsedTime
                    0.01%  19.815us         6  3.3020us     828ns  13.234us  cudaEventCreate
                    0.00%  12.366us        22     562ns     167ns  7.2940us  cudaSetupArgument
                    0.00%  5.9140us         6     985ns     732ns  2.1010us  cudaEventDestroy
                    0.00%  4.8760us         3  1.6250us     281ns  3.4940us  cuDeviceGetCount
                    0.00%  3.3170us         4     829ns     479ns  1.7980us  cudaConfigureCall
                    0.00%  1.4990us         1  1.4990us  1.4990us  1.4990us  cudaGetDeviceCount
                    0.00%  1.2580us         2     629ns     379ns     879ns  cuDeviceGet
```

The profiling report with the flop counts for individual functions is as follows:

```
==1723== Profiling result:
==1723== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "Tesla K20m (0)"
    Kernel: device_run_solver(int, double*, double*, double*, double*, double*)
          1                            flop_count_sp   Floating Point Operations(Single Precision)         400         400         400
    Kernel: device_predict_value(int, int, double, double, double*, double*, double*, XY*)
          1                            flop_count_sp   Floating Point Operations(Single Precision)           0           0           0
    Kernel: device_compute_K(int, int, double*, XY*)
          1                            flop_count_sp   Floating Point Operations(Single Precision)           0           0           0
    Kernel: device_run_chol(int, double*, double*, double*)
          1                            flop_count_sp   Floating Point Operations(Single Precision)        9900        9900        9900
```

As evident from the above table, the runtime statistics are as follows:

- Flop count in Cholesky (Factorization) routine: ~10,000
- Flop count in Solver routine: 400
- Time Taken by Cholesky (Factorization) = 94.993759 ms
- Time Taken by Solver = 20.005983 ms

Therefore,

- Flop rate achieved in factorization: 105.26 KFLOPS
- Flop rate achieved in Solver: 20 KFLOPS

Peak Floating-point operation per second calculation

Device Details: The device I was using to run my program was Nvidia Tesla K20m. The device has the following properties:

- GPU Clock rate: 706 MHz (0.71 GHz)
- Multiprocessors: 13
- CUDA Cores/MP: 192
- Total No. Of Cores (Multiprocessors * Cores): 2496 CUDA cores
- CPU instruction per cycle: 2

The peak flop rate can be calculated using the following formula:

Node performance in GFlops = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node)

In our scenario:

- Clock Rate is 706 MHz
- We were required to use just one Multiprocessor in our implementation. I used an entire CPU as a single logical block So the number of CPU cores to be used in our calculation is 192.
- CPU instruction per cycle: 2 (from online sources)
- Number of CPU's per node: 1 (We just used 1 out of 13 available CPU's)

Peak Floating point performance = 706 MHz * 192 * 2 * 1= 271 GFLOPS

- Comparing the Peak Floating Point Performance with our achieved results, I think that the flop rates are acceptable but there is a lot of scope for improvement.

Speed-Up & Efficiency Calculations

The table below shows speedup & Efficiency calculations for various values of m(where m is the number of points in a 2-dimensional unit square along an axis) in dedicated batch submission mode. The calculations are made w.r.t the runtime of Cholesky factorization routine as it consumed the largest runtime among all other functions.

m	**Time Taken with 1 Core (ms)**	**Time taken with 192 cores (ms)**	**Speed Up**	**Efficiency** (Calculation based on the

				total number of threads available. Doesn't matter because not all the threads were used due to small size of the matrix)
4	.85	0.67	1.27	.66%
8	25.07	21.11	1.19	.62%
16	1450.17	1016.49	1.43	.74%