**CSCE 659 Fall 2017**

**HW5: GPU Programming**

**UIN:122006286**

**Haiping Xue**

**Problem 1**

For this program, I used CUDA library and implemented a program using Gaussian

Process Regression to predict the values of a function a point from observations at other points in the domain.

Since this code is optimized to certain size of the gird, the program will only produce correct answer for certain input grid size. The tested right results is 64 and 32. Other result is not tested yet. It’s not promised they are correct. My experiment is based grid size 64 in which the LU factorization matrix’s size is 4096.

To verify the code, the program will generate 4 matrices for checking the computation result. They are LU\_Matrix which is LU matrix after LU factorization, origin\_Matrix which is (tI+K) in the HW5 description, fVec is the f(x,y) and fRes is kT. Put all the program generated in the same folder with check.m. Run check.m to see vecNorm and nor to check the correctness of the result.

**Use the following command to compile the code on ADA:**

module load intel/2017A CUDA

nvcc -o cuda cuda.cu

**Run the program using:**

./cuda 64 0.5 0.5 1024

./cuda 32 0.5 0.5 1024

64 is the grid size m in the HW5 requirement, 0.5 0.5 is the point for prediction and 1024 is the thread number. To keep the program’s correctness, use tested size to run the program.

**For job submission using the following file:**

#BSUB -J test #Set the job name to "ExampleJob2"

#BSUB -L /bin/bash #Uses the bash login shell to initialize the job's execution environment.

#BSUB -W 0:10 #Set the wall clock limit to 6hr and 30min

#BSUB -n 20 #Request n cores

#BSUB -n 20 -R "span[ptile=20]" -R "select[gpu256gb]"

#BSUB -R "rusage[mem=2560]" #Request 2560MB per process (CPU) for the job

#BSUB -M 2560 #Set the per process enforceable memory limit to 2560MB.

#BSUB -o testOut.%J #Send stdout and stderr to "Example2Out.[jobID]"

module load intel/2017A CUDA

./cuda 64 0.5 0.5 1024

**Problem 2**

To parallel this program, I separate the main program into following parts:

1. computePoints(xi, yi, h, m) and computeF(xi, yi, f, n) which create all points in the grid and compute the f(xi, yi) on the host.
2. computeK<<<1, threadNum>>>(dXi, dYi, dK, n) will compute (tI+K) on the device. Each thread will control certain number of rows and finish the computation.
3. LU<<<1, threadNum >>>(dK, n, blockSize) is computed using cyclic row partition. This strategy will keep most threads busy when the dynamic matrix’s size is decreasing. The big for loop will keep running and the inner for loops are paralleled. At the each of two inner for loops end, a syncthreads will enforced to avoid getting wrong result.
4. Solver are divided into two parts: solveL and solveU. Each thread will control certain number of rows and finish the computation.
5. computeKT<<<n,1>>>(dXi, dYi, DkTrans, n, xp, yp) is done on the device by using n=m\*m size 1D blocks.
6. The result is computed with computeRes(kt, res, n).

To improve the performance of LU, I used cyclic row partition instead of normal row partition. The reason is I found the size of the dynamic matrix is decreasing and fewer threads are doing useful work. Cyclic partition will solve this problem. The performance improvement is around 7 seconds for input size 64 which is a 4096\*4096 matrix. Also, using double and float will influence the result. Double type will give higher accuracy and float type will give better performance.

**Problem 3**

The test for this program is

Time for LU factors routines: 184.9884796143 s

Time for solver routine is: 0.0870275795 s

In theory,

The flop counts for LU in theory is = 45812984490.7

The flops rate for LU is flop count/time = 45812984490.7/184.9884796143 = 247653175.95 flops/s = 0.25 Gflop/s

The flop counts for solve L and solve U in theory is = 33554432

The flops rate for solve L and solve U is

flop count/time 33554432/0.0870275795 = 385560901.415 flops/s = 0.39 Gflop/s

The peak flop rate for the GPU is 3.52 TFLOPS which is 3520 Gflop/s. The estimated peak flop rate for each multiprocessor is 3520/13 = 270.77 Gflop/s. The max threads in one block is 1024. The max threads on one multiprocessor is 2048. Therefore, the peak flop rate for one block is 135.385 Gflop/s.

The utilization of cores for LU is program flop rate/peak flop rate on one block = 0.25/135.385 = 0.00184658566

The utilization of cores for solver is program flop rate/peak flop rate on one block = 0.39/135.385 = 0.00288067363

The speedup of the program for an input 32 which is 1024\*1024 matrix is 120/3 = 40

However, when I am just looking at the utilization of flop, I found that the speedup for flops maybe very low. The true speedup comes from the parallel program’s algorithm. Not the capability of flops on the GPU.