Major Project: Gaussian Process Regression (GPR)

Mian Qin, UIN: 725006574

1. ***Develop parallel code to determine the hyper-parameters l1 and l2 that minimize the MSE.***

In this project, I implemented both OpenMP-based shared memory code or a GPU code for parallelization. For OpenMP implementation, there are two version of optimizations (OPT1 & OPT2). The first implementation (OPT1) is serialized issue GPR function for each set of hyper-parameters and parallels the Cholesky Factorization code which is developed by previous homework within each GPR call. The idea is that factorization takes most of the runtime. The second implementation (OPT2), however, parallels in a larger granularity, i.e. parallelly issue GPR calls for each thread. Ideally, the second optimization can achieve better utilization since it reduces the thread launch overhead.

For GPU implementation of determining the hyper-parameters that minimize the MSE, I used the idea of the OPT2 for OpenMP code. The GPU kernel compute the whole GPR function (including initializing K/k matrix, factorization, calculate the ftest results and calculate the error). The kernel launch uses multiple thread blocks and multiple threads per thread block. Each thread block executes a single GPR call for a set of hyper-parameters and parallels the possible subroutines using multiple threads, including initializing K/k matrix, cholesky factorization, matrix multiplications, etc. Some subroutines such as forward/backsubstitution have no parallelism and it's executed in serial. **The reason to choose implemented the whole GPR code in kernel rather than only offloading factorization is to minimize the overhead of transfer data between host and GPU** back and forth (K0 and U matrix which are both large). Instead, the whole program only need to transfer XY and f input data and final MSE matrix which are relatively small. Inside kernel it dynamic allocate global memory for intermediate data structures such as K, k U matrix.

The **main benefit** of GPU implementation compared to OpenMP implementation is that GPU has more hardware resources to parallel compute GPR calls. (for K20, it has 13 SMs, each SMs has hundreds of floating points units, multiple thread blocks can run concurrently on single SMs to maximize the memory throughput). However, CPU has less floating points units per core and limited the parallelism.

A) Code description.

OpenMP code:

The C++ implementation doesn't rely on any mathematical libraries. Details of the implementation can be referred to the source code.

The main GPR function prototype is

```
double **GPR(double *XY1, double *XY2, int n, double **f, int *itest, i
nt ntest, int *itrain, int ntrain, double t, double l1, double l2)
```

which takes the 2D grids XY matrix, observed data vector f, test and train data index and hyper-parameters <t, l1, l2> as input. The GPR returns the calculate ftest matrix.

The LU factorization function prototype is

```
void CholeskyFactorization(double **K, double **U, int n)
```

which takes a 2D double vector pointer K and matrix size n as input. The result is returned to another 2D double vector pointer U. (U need to be allocated by the called as same size as K and filled with zero using calloc).

The arbitrary size matrix multiplication function prototype is

```
void Mmult(double **A, int rA, int cA, double **B, int rB, int cB, doub
le **C, int rC, int cC)
```

which takes two 2D double vector pointer A and B and corresponded matrix size as input. The result is written in the another 2D double vector C.

Forward/Back substitution function prototype is

```
void ForwardSubstitution(double **L, int nL, double *f, int nf, double
*y, int ny)
void BackSubstitution(double **U, int nU, double *f, int nf, double *y,
 int ny)
```

which takes triangular matrix and a vector to calculate the \ operation (for example L\f).

GPU code:

The main GPR function prototype is

```
__global__ void GPR(double *XY, int n, double *f, int *itest, int ntest
, int *itrain, int ntrain, double t, double *l, int nl, double *MSE)
```

which is pretty much the same as OpenMP version. EXCEPT it allocate final MSE matrix directly and each thread block write to an element in MSE matrix.

B) Build & Run

Compile: The code is compiled as follow:

---

icpc -qopenmp -std=c++11 GPR.cc -DOPT1 -O3 -o GPR_omp_opt1
icpc -qopenmp -std=c++11 GPR.cc -DOPT2 -O3 -o GPR_ omp_opt1

nvcc -arch=compute_35 -code=sm_35 -ccbin=icc -Xcompiler="-std=c++11 -O3" -O3 -o GPR_gpu GPR.cu

---

Note: There are two optimization macro (OPT1, OPT2) for the OpenMP implementation as discussed above.

Run: For OpenMP implementation, there are four program arguments for setting the grid size and parameter l vector (start, step, lenght). Number of OpenMP threads is set through environment variable **OMP_NUM_THREADS**. Example as follow:

```
OMP_NUM_THREADS=16 OMP_PLACE=sockets ./GPR_icc_opt1 32 0.25 0.5 20
```

For GPU implementation, there are 1 more main program arguments set the number of threads per thread block. Example as follow:

```
./GPR_gpu 32 0.25 0.5 20 512
```

C) Note

The build script and job file for batch system is included in the homework submission.

D) Validation

To validate the correctness of the OpenMP and GPU version of the code. I compare the result generated by the Matlab example code and my C++ implementation with the same input. Since the random function of C++ (drand48_r) and Matlab (rand) generate different results, I print out the C++ implementation observed data vector f and test/train index and copied to Matlab code and make sure the input is identical and then compare the results.

Matlab result with same input:

```
MSE =

    0.0332    0.0149    0.0063    0.0036    0.0025    0.0020    0.0018    0.0018    0.0019    0.0020
    0.0162    0.0060    0.0023    0.0013    0.0009    0.0007    0.0007    0.0008    0.0009    0.0011
    0.0079    0.0027    0.0010    0.0006    0.0004    0.0003    0.0003    0.0004    0.0005    0.0007
    0.0051    0.0017    0.0007    0.0004    0.0003    0.0002    0.0002    0.0003    0.0004    0.0006
    0.0038    0.0013    0.0005    0.0003    0.0002    0.0002    0.0002    0.0003    0.0004    0.0006
    0.0031    0.0012    0.0005    0.0003    0.0002    0.0002    0.0002    0.0003    0.0004    0.0006
    0.0029    0.0011    0.0005    0.0003    0.0003    0.0002    0.0003    0.0003    0.0004    0.0006
    0.0028    0.0012    0.0006    0.0004    0.0003    0.0003    0.0003    0.0004    0.0005    0.0007
    0.0029    0.0014    0.0007    0.0005    0.0004    0.0004    0.0004    0.0005    0.0006    0.0007
    0.0030    0.0016    0.0009    0.0007    0.0006    0.0006    0.0006    0.0006    0.0007    0.0009

Minimum MSE 0.000197, Lparam_x 0.140625, Lparam_y 0.171875
```

C++ with OpenMP (20 threads) results (input argument 16 0.25 0.5 10) :

```
Finished in 39.52 miliseconds [Wall Clock]
MSE:
0.0326  0.0146  0.0062  0.0036  0.0025  0.0020  0.0018  0.0018  0.0018  0.0020
0.0159  0.0059  0.0022  0.0012  0.0008  0.0007  0.0007  0.0007  0.0009  0.0011
0.0078  0.0027  0.0010  0.0006  0.0004  0.0003  0.0003  0.0004  0.0005  0.0007
0.0050  0.0017  0.0006  0.0004  0.0003  0.0002  0.0002  0.0003  0.0004  0.0006
0.0037  0.0013  0.0005  0.0003  0.0002  0.0002  0.0002  0.0003  0.0004  0.0005
0.0031  0.0012  0.0005  0.0003  0.0002  0.0002  0.0002  0.0003  0.0004  0.0005
0.0028  0.0011  0.0005  0.0003  0.0003  0.0002  0.0003  0.0003  0.0004  0.0006
0.0027  0.0012  0.0006  0.0004  0.0003  0.0003  0.0003  0.0004  0.0005  0.0006
0.0028  0.0013  0.0007  0.0005  0.0004  0.0004  0.0004  0.0005  0.0006  0.0007
0.0029  0.0015  0.0009  0.0007  0.0006  0.0005  0.0005  0.0006  0.0007  0.0008

Minimum MSE 0.000192, Lparam_x 0.140625, Lparam_y 0.171875
```

C++ with GPU offloading results (input argument 16, 0.25 0.5 10 512) :

```
Finished in 287.79 miliseconds [Wall Clock]
MSE:
0.0326  0.0146  0.0062  0.0036  0.0025  0.0020  0.0018  0.0018  0.0018  0.0020
0.0159  0.0059  0.0022  0.0012  0.0008  0.0007  0.0007  0.0007  0.0009  0.0011
0.0078  0.0027  0.0010  0.0006  0.0004  0.0003  0.0003  0.0004  0.0005  0.0007
0.0050  0.0017  0.0006  0.0004  0.0003  0.0002  0.0002  0.0003  0.0004  0.0006
0.0037  0.0013  0.0005  0.0003  0.0002  0.0002  0.0002  0.0003  0.0004  0.0005
0.0031  0.0012  0.0005  0.0003  0.0002  0.0002  0.0002  0.0003  0.0004  0.0005
0.0028  0.0011  0.0005  0.0003  0.0003  0.0002  0.0003  0.0003  0.0004  0.0006
0.0027  0.0012  0.0006  0.0004  0.0003  0.0003  0.0003  0.0004  0.0005  0.0006
0.0028  0.0013  0.0007  0.0005  0.0004  0.0004  0.0004  0.0005  0.0006  0.0007
0.0029  0.0015  0.0009  0.0007  0.0006  0.0005  0.0005  0.0006  0.0007  0.0008

Minimum MSE 0.000192, Lparam_x 0.140625, Lparam_y 0.171875
```

2. ***Describe your strategy to parallelize the algorithm. Discuss any design choices you made to improve the parallel performance of the code. Also report on the parallel performance of your code.***

Below are the optimization strategies for each part of the code.

1) **OpenMP optimization:**

A) **Parallel granularity**. The OpenMP-based parallelism is quite straightforward, since we have identified multiple key issues for OpenMP optimization in the previous homework (such as cutoff threshold for small size iterations). The main question in this project is the parallel granularity since we need to process multiple GPR calls. In each GPR call, factorization is still the most time-consuming part ($O(N^3)$ complexity). As we discuss in Section 1, we can choose to parallel factorization only and sequentially call GPR function in serial (OPT1). Or, we can directly let each thread pickup a GPR function call and run factorization in serial (OPT2). The latter has better efficiency due to it needs less thread launch and scheduling. (Results present in later part).

The third way is to use nested parallelization which is support by OpenMP. However, this approach adds even more thread launch and scheduling overhead and the performance is much worse than OPT1 & OPT2.

2) **GPU optimization:**

Compared to OpenMP-based implementation, GPU code is much difficult to implement and to debug. So, first I want to share some experience with GPU debugging. And then discuss about the GPU optimization techniques.

A) GPU debugging. Generally, we can debug GPU code in this sequence. First, use only 1 thread block and 1 thread and debug the serial code (using cuda-gdb) which is much easier to find serial logical bugs. Then, we can keep thread block as 1 and increase the thread count in the block. In this case cuda-gdb may not easily identify bugs. However, we can break into synchronization point to obverse each data point to find potential problems. Finally, we may add more thread blocks. Usually, in this stage the logical bugs are already cleared. It may have some higher level problems which requires more knowledge of the CUDA framework.

For example, in my implementation there is a weird problem that the program finish quickly with all zero results when the thread block count pass certain point (for example more than 4). Below that the results are correct. Using cuda-gdb I found that some matrix which is dynamically allocated in kernel code (to global memory) failed (by return 0x0) and cause kernel

unexpectedly exit. The problem is that CUDA has a very small heapsize limit (8MB). By increasing the heap limit using cudaDeviceSetLimit the problem got resolved.

B) Minimize data transfer. One important optimization for GPU code is to minimize the data transfer between host and GPU. In this project, each GPR call requires different K matrix (based on hyper-parameters). So, if we want to only offloading the factorization part (using HW5), it needs to transfer K and U matrix multiple times. In order to avoid that, I implement the whole GPR function in the kernel code, including initializing K/k matrix, factorization, calculate the ftest results and calculate the error. In this case, the whole program only needs to transfer 2D grid XY and the MSE matrix (from device to host).

C) Using multiple SMs. With in single GPR function, there are multiple parts that can be parallelized, such as extracting matrix, factorization and matrix multiplication. This can be parallelized by multiple threads in a single thread block. Given the fact that different GPR calls has no dependency, we can use multiple thread blocks and better utilize the multiple SMs in GPU. This will bring us more speedup.

D) Minimize heap usage. Another problem I met is shortage of GPU device memory. Since each GPR call needs heap memory for holding intermediate results (such as K0, K, factorization results, etc.), the GPU may run out of device memory and fails. (Also, CUDA will map more blocks on single SM, which will cost even more memory). So, in the code, I did some optimization to limit the heap usage by not allocating all the intermediate matrix at beginning and free them at the end of block life.

E) Device memory limitation. Due to the fact that GPU has limited device memory (K20 has 4GB onboard DRAM as global memory), very large dataset is not able to run on GPU directly. In order to make sure arbitrary large dataset can run successfully through the GPU code. I need to control each kernel grid size to make sure the heap usage (on device) doesn't exceed the maximum capacity. In my implementation, I will estimate the heap usage per thread block and calculate the optimal grid size and launch kernel multiple times to finish all GPR calls.

3) **Evaluation:**

For evaluation, I evaluated the three run cases (small: 32, 0.25 0.5 10, large: 48 0.25 0.5 20, extreme: 64 0.25 1.0 20). I ran each experiment three times to rule out some performance jitter. All evaluation is done in the **ADA cluster (with K20 GPU).**

Figure 1 illustrates the speedup efficiency results for first optimization of OpenMP-based parallel code. The baseline to calculate speedup is the runtime of single thread. For different data set the scale slope changes. For small case, the maximum speedup is around 7x at 18 threads/cores. For large and extreme test case, the maximum speedup is ~12x and ~16x respectively. This is because parallel the Cholesky factorization introduce more thread launch and scheduling overhead for smaller inputs. However, for larger inputs, since each GPR call takes long time (most cost on factorization), then the thread launch and scheduling overhead got amortized.

Figure 2 illustrates the speedup efficiency results for second optimization of OpenMP-based parallel code. The baseline to calculate speedup is the runtime of single thread. As you can see the speedup is pretty linear for different data set input. For large and extreme case, the maximum speedup can achieve ~15x and ~18x, for small test case, the maximum speedup is

~11x, which is much better for opt1 implementation. This is due to more coarse granularity of parallelism reduce the thread launch and scheduling overhead. From Figure 1&2, we can come up with the conclusion that for smaller data set, OPT2 is better and for larger data set the two version of implementation is about the same.

Figure 3 illustrates the speedup efficiency results for GPU parallel code. The baseline to calculate speedup is the runtime of **16 threads per block**. For large and extreme case, the maximum speedup can achieve ~8x, for small test case, the maximum speedup is ~6.5x. Since GPU hardware is different from CPU cores, so the absolute value of efficiency is weird, but the trend should be correct.

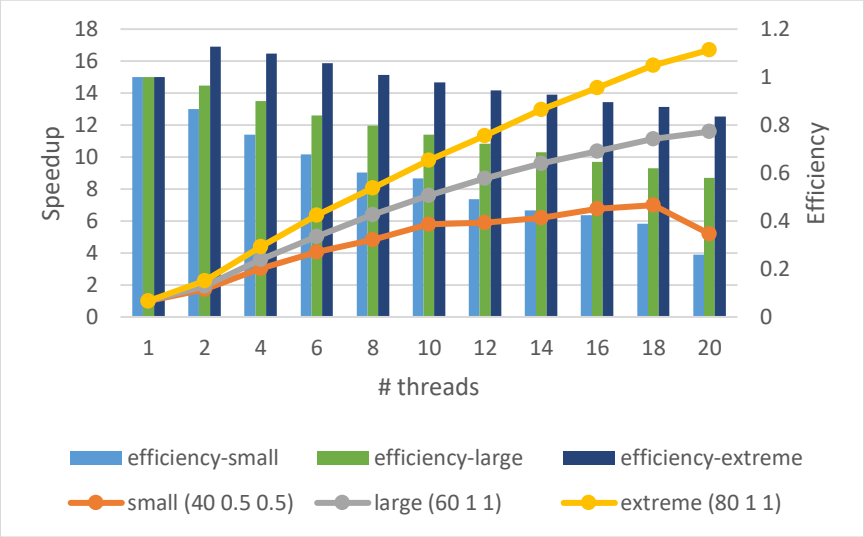

Figure 1 Speedup & efficiency results for OpenMP-opt1

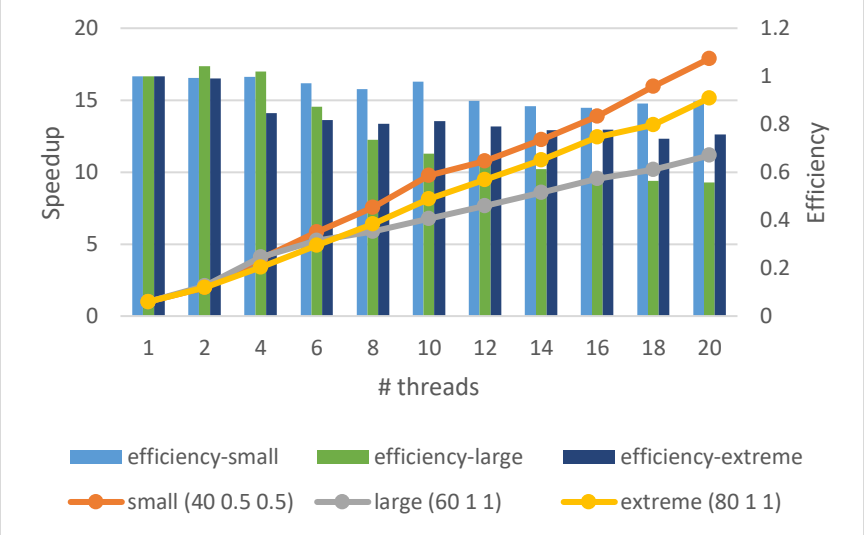

Figure 2 Speedup & efficiency results for OpenMP-opt2

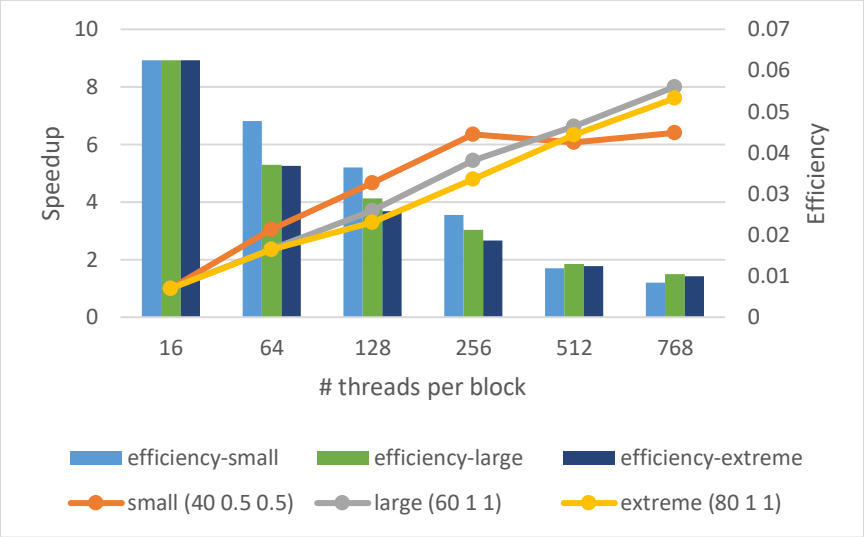

Figure 3 Speedup & efficiency results for GPU implementation

3. ***<u>Apply your code to another data set to see how it performs. You may choose any appropriate data set that you can find.</u>***

I also demonstrated the speedup of opt1 & opt2 of OpenMP-based parallel code and compared to GPU parallel code in Figure 4. As the results shows, OpenMP-based (CPU code) parallel code works better on smaller data set. However, on large and extreme data set, the GPU code can achieve about the same speedup.

Ideally, GPU have more hardware (especially floating point units), and should achieve better speedup. However, there are two reasons that GPU speedup got limited. First, there are a lot of serial code in the GPR call (calculate sum, forward/back substitution, etc.) which is execute more efficiently on CPU than GPU. Second, due to the device memory limitation on GPU (4GB on K20), for large matrix input, GPU may not launch as much as parallel GPR calls as CPU. If we use a GPU with more device memory or limited the CPU main memory, GPU implementation may get better performance.

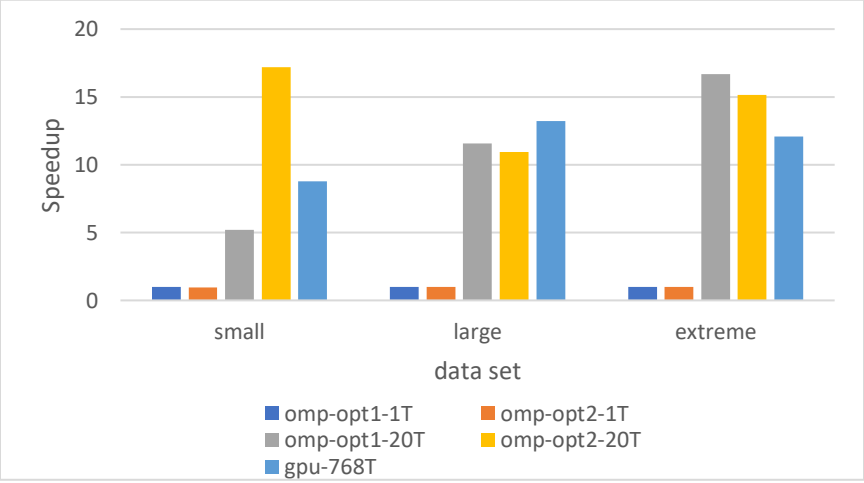

Figure 4 Speedup comparison for OpenMP-based and GPU parallel code