HW 5: GPU Programming

Mian Qin, UIN: 725006574

1. ***Develop GPU code to compute f(x,y) for a given point (x,y). The code should use a single processor of the device but should be parallelized to exploit all the cores within the processor.***

   The GPU implementation of **Gaussian Process Regression** is based on previous C++ implementation (Minor project). From the observations in Minor project, LU factorization takes up most of the runtime (~99%) in the whole program phases. Other parts such as initialize grid data, initialize K'=(tI+K) and forward/back substitution only takes up ~1% of the overall runtime. So, in the GPU code, I mainly implemented the LU factorization part (I used Cholesky Factorization for LU factorization). Besides, the K'=(tI+K) is embedded in to the Cholesky Factorization code. Other parts are implemented in the host code (same as Minor project implementation).

   A) Code description.

   The C++ implementation doesn't rely on any mathematical libraries. I implemented CUDA implementation of **CholeskyFactorization** which use **single SM** (streaming multiprocessor).

   The main GPR function prototype is

   ```
   double GPR(int m, double rstar_l, double rstar_r, int cuda_threads)
   ```

   which takes an integer m and two double rstar_l and rstar_r and another integer cuda_threads as the input and return the predicted value fstar as double type. cuda_threads specify the number of threads used in the GPU SM.

   I implemented two version of CUDA code for CholeskyFactorization.

   The LU factorization function prototype is

   ```
   __global__ void CholeskyFactorization_v1(double *K, double *U, int n, i
   nt row)
   ```

   This version only process 1 row (specify by the last argument) of the result upper triangular matrix. (Which means the host needs to invoke kernel multiple times to get the final results) The work is distributed evenly to multiple CUDA threads within a SM.

   ```
   __global__ void CholeskyFactorization_v2(double *K, double *U, int n)
   ```

   This version processes the whole LU factorizations within a single kernel launch. Each thread needs to maintain synchronization to finish process 1 row's results before getting to the next row.

B) Build & Run

Compile: The openmp and gpu implementation code is compiled as follow:

```
icpc -qopenmp -std=c++11 GPR.cc -O3 -o GPR_omp
nvcc -arch=compute_35 -code=sm_35 -ccbin=icc -Xcompiler="-fopenmp -std=c++11 -O3" -O3 -o GPR_gpu GPR.cu
nvcc -arch=compute_35 -code=sm_35 -ccbin=icc -Xcompiler="-fopenmp -std=c++11 -O3" -O3 -DOPT -o GPR_gpu_opt GPR.cu
```

Run: There is three program argument for setting the grid size and coordinates of rstar. Number of OpenMP threads is set through environment variable **OMP_NUM_THREADS**. OMP code example as follow:

```
OMP_NUM_THREADS=16 OMP_PLACE=sockets ./GPR_omp 16 0.5 0.5
```

GPU code example as follow:

```
./GPR_gpu 16 0.5 0.5 1024
```

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 GPU code. I compare the result generated by the previous OpenMP implementation and the GPU implementation.

C++ with OpenMP (20 threads) results (input argument 40 0.5 0.5) :

```
[celery1124@ada1 HW5-689]$ OMP_NUM_THREADS=20 ./GPR_omp 40 0.5 0.5
Initialize m x m grid 0.758 ms
Initialize observed data f 0.047 ms
Initialize k 0.025 ms
Initialize K 21.200 ms
LU factorization 517.221 ms
Compute predicted value fstar 10.878 ms
Free up 0.582 ms
Finished in 550.82 miliseconds [Wall Clock]
fstar 0.969372
```

CUDA (1SM-1024Threads) result with same input:

```
[celery1124@ada1 HW5-689]$ ./GPR_gpu 40 0.5 0.5 1024
Initialize m x m grid 1.147 ms
Initialize observed data f 0.063 ms
Initialize k 0.042 ms
Initialize K 55.407 ms
[CUDA] LU factorization setup device buffer 88.573 ms
[CUDA] LU factorization kernels 538.946 ms
[CUDA] LU factorization write-back results 9.612 ms
Compute predicted value fstar 11.423 ms
CUDA Compute predicted value fstar 0.031 ms
Free up 0.421 ms
Finished in 705.80 miliseconds [Wall Clock]
fstar 0.969372
```

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.***

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

A) **Cholesky factorization**. This is a major portion of the code that worth optimize. For Cholesky factorization, it contains three loops. The outer loop cannot be parallelized since the inner two loops has dependency. Each outer loop iteration compute 1 row of the result upper triangular matrix **U** from top to bottom and the later iteration needs to use the results generated from previous iterations. Thus, to leverage parallelism in GPU, we can only distribute the second loop (no dependency) to multiple threads.
Besides, the second loop, calculating the diagonal for each row can be also paralleled (each GPU thread calculate partial sum and thread0 calculate the final sum and calculate the diagonal).
To get better memory performance, each GPU thread calculate continuous iteration (to avoid stride memory access). The first version of CUDA Cholesky factorization implementation is shown as follow (inside the kernel parallel calculate single row of the final results).

```
__global__ void CholeskyFactorization_v1(double *K, double *U, int n, int row)
{
    __shared__ double sum;
    __shared__ double partial_sum[MAX_THREADS];
    // caller make sure Matrix U is zeroed (through calloc)
    // Cholesky-Crout Algorithm on row_id row
    int partial_row = (int ) ceil( (double)row / (double)blockDim.x);
    int row_start = partial_row * threadIdx.x;
    int row_end = partial_row + row_start < row ? partial_row + row_start : row;

    // Calculate Diagnols
    partial_sum[threadIdx.x] = 0;
    for (int k = row_start; k < row_end; k++) {
        partial_sum[threadIdx.x] += (U[k*n + row] * U[k*n + row]);
    }
    __syncthreads();
    // Thread-0 calculate and write Diagnols for this row
```

```
    if (threadIdx.x == 0) {
        sum = 0;
        for (int i = 0; i < blockDim.x; i++) sum += partial_sum[i];
        U[row*n + row] = sqrt(K[row*n + row] + 0.01 - sum);
    }
    __syncthreads();

    partial_row = (int) ceil( (double)(n-1-row) / (double)blockDim.x );
    row_start = row+1 + partial_row * threadIdx.x;
    row_end = partial_row + row_start < n ? partial_row + row_start : n;
    for (int i = row_start; i < row_end; i++) {
        double sum2 = 0;
        for (int k = 0; k < row; k++) sum2 += (U[k*n + i] * U[k*n + row]);
        U[row*n + i] = (K[i*n + row] - sum2) / U[row*n + row];
    }
}
```

B) **Forward/Back substitution**. Since forward/back substitution are serial code (has dependence for processing each element in the result), there is no parallelism to exploit. Thus, this part is better implemented in the host. I also did a GPU implementation of forward/back substitution (using single thread). However, the performance is much worse than running on host (x86). So, the following evaluation didn't use this.

3. ***Compute the flop rate you achieve in the factorization routine and in the solver routine using all the cores. Compare this value with the peak flop rate achievable on a single core, and estimate the speedup obtained over one core and the corresponding efficiency/utilization of the cores on the node.***

For evaluation, I evaluated the three run cases (small: 40, 0.5 0.5, large: 60 1 1, extreme: 80 0.4 0.6) and compared with OpenMP implementation. I ran each experiment three times to rule out some performance jitter. Table 2 shows the flop rate for factorization routine. (Here the flop contains add/sub, multiply, divide and exp and sqrt operations). Since the LU factorization takes up 99% of the overall runtime (O(N^3) vs O(N^2)). Thus, the GPR slover flop rate should be similar to LU factorization flop rate.

Table 2 Flop rate for LU factorization

	omp (1 core)	omp (20 cores)	gpu (1SM-1threads)	gpu (1SM-1024threads)	K20 per SM peak
m=40 (GFLOPS)	0.5748	4.6859	0.0114	2.199	90.37
m=60 (GFLOPS)	0.5472	9.8508			90.37

Figure 1 illustrates the speedup efficiency results for first version gpu implementation (invoke multiple kernel calls to process factorization row by row). The baseline to calculate speedup is the runtime of single SM single thread. Since in this particular GPU in ada (k20m), the maximum

threads per SM is 1024, I run different thread counts from 1 to 1024. For different input size (small, large, extreme), the maximum speedup compared to single thread is ~180-190. Due to k20m has 192 cuda cores per SM, the speedup scaling is pretty good.

Figure 2 illustrates the speedup efficiency results for second version gpu implementation (invoke single kernel call to process the whole factorization). From the results, the two GPU implementations seems perform similarly, which indicates that host call for launching kernel isn't introduce much overhead. Or, the CUDA synchronization is too slow, at least comparable to the host kernel launch call.

Figure 3 illustrates the speedup compared to c++ openMP implementation of the GPR solver (the baseline, i.e. speedup 1 is 20 core openMP results). From the results, single SM in GPU can achieve around 30% of the total 20 core CPU performance. Given the fact that K20m has 13 SMs, ideally the speedup can be achieved by utilizing all SMs is **~4x.**

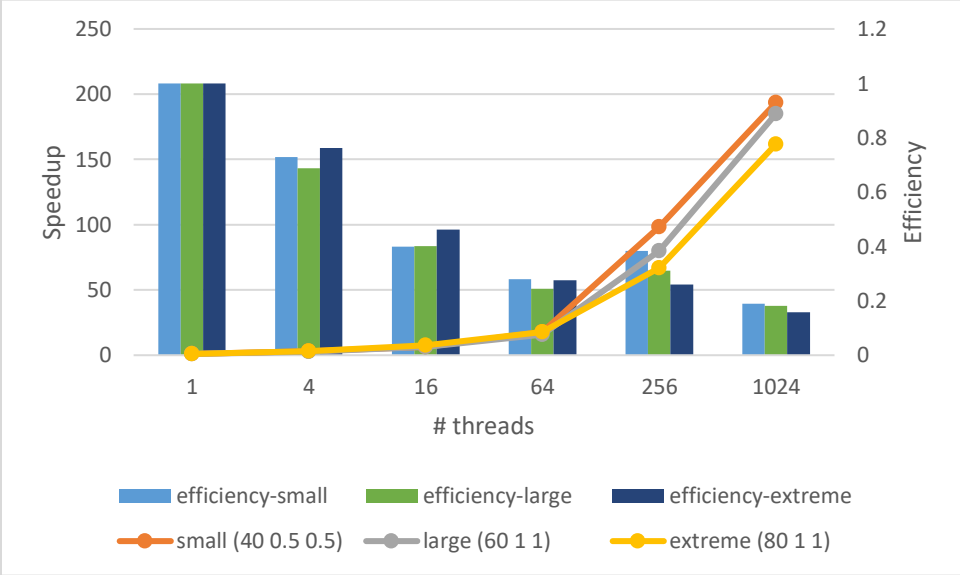

Figure 1 Speedup & efficiency results gpu factorization

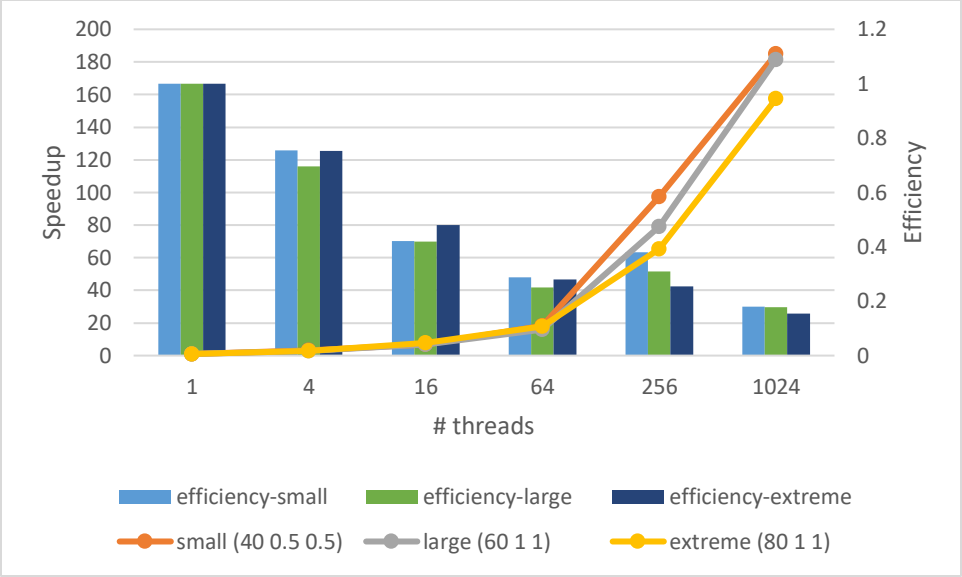

Figure 2 Speedup & efficiency results gpu-opt factorization

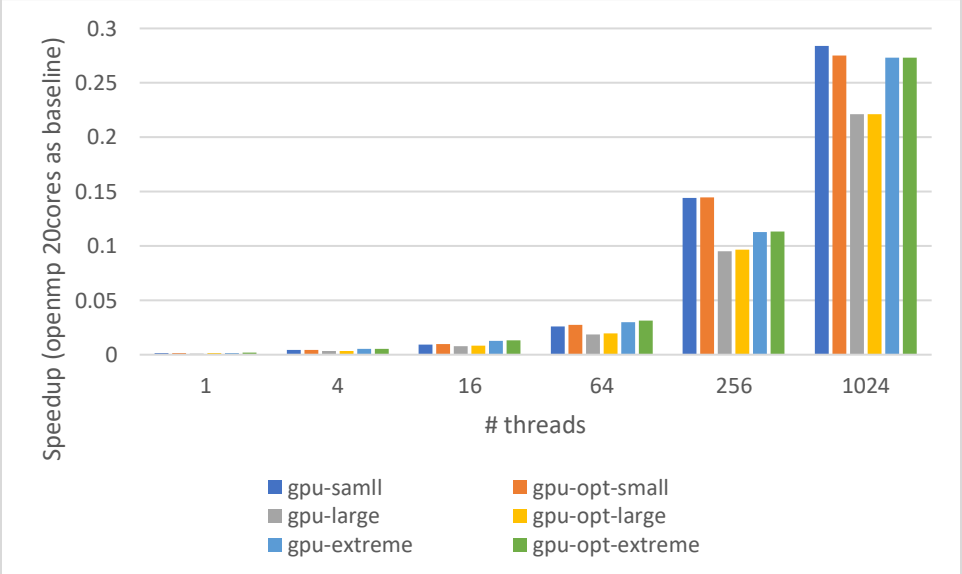

Figure 3 Speedup comparison for 2 gpu implementations