EECS E6895 Advanced Big Data Analytics

Homework 3 (using grace day)
Name: Jingyi Yuan
UNI: jy2736

Algorithm 1: Matrix multiplication

I use CUDA to calculate the product of two matrices. The first matrix is an A*B matrix and the second one is a B*C one.

I write both the CPU and the GPU way to compute the product and record the start time and end time of these two ways.

CPU:

```
clock_t startc, finishc;

//start using CPU to multiply matrix
startc = clock();
for (int i = 0; i < A; i++){
    for (int j = 0; j < C; j++)
    {
        float sum = 0;
        for (int k = 0; k < B; k++)
        {
            sum += a[i * B + k] * b[k * C + j];
        }
        c[i * C + j] = sum;
    }
}
finishc = clock();
```

GPU:

```
#define A 2
#define B 3
#define C 4

__global__ void product(float *a, float *b, float *c, int aa, int bb, int cc)
{
    int ix = threadIdx.x + blockIdx.x * blockDim.x;
    int iy = threadIdx.y + blockIdx.y * blockDim.y;
    if (ix < aa && iy < cc)
    {
        float sum = 0;
        for (int index = 0; index < bb; index++)
        {
            sum += a[ix * bb + index] * b[index * cc + iy];
        }
        c[ix * cc + iy] = sum;
    }
}
```

```
//********** GPU, matrx initialization **********
clock_t start, finish;
float *d_a, *d_b, *d_c;

//start using GPU to multiply matrix
start = clock();

cudaMalloc(&d_a, A * B*sizeof(float));
cudaMalloc(&d_b, B * C*sizeof(float));
cudaMalloc(&d_c, A * C*sizeof(float));

cudaMemcpy(d_a, a, A * B*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, B * C*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_c, c, A * C*sizeof(float), cudaMemcpyHostToDevice);

int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((A + block.x - 1) / block.x, (C + block.y - 1) / block.y);

product << <grid, block >> >(d_a, d_b, d_c, A, B, C);

cudaMemcpy(c, d_c, A * C*sizeof(float), cudaMemcpyDeviceToHost);
```

I first define A = 2, B = 3, C = 4 to verify the correctness of the code and the output is:

```
input matrix 1:
2.00    2.00    2.00
2.00    2.00    2.00

input matrix 2:
2.00    2.00    2.00    2.00
2.00    2.00    2.00    2.00
2.00    2.00    2.00    2.00

output matrix (CPU):
12.00   12.00   12.00   12.00
12.00   12.00   12.00   12.00
**********************************************
The total time using CPU: 0.000000 seconds
**********************************************

output matrix (using GPU):
12.00   12.00   12.00   12.00
12.00   12.00   12.00   12.00
**********************************************
The total time using GPU: 0.171000 seconds
**********************************************
```

We can see that the result is correct, and when matrix is small, CPU is faster than GPU since we need to initialize when using GPU.
Then I used large matrix to do the calculation and change the value of A, B and C.

```
#define A 1000
#define B 1500
#define C 2000
```

This time, I only outputted one value in the product since I initialized every number in input1 and input 2 to be 2, thus every number in the output is the same and I only need to ensure that the number is within the range of float and does not overflow.

```
number in matrix: 6000.00
**********************************************
The total time using CPU: 23.146000 seconds
**********************************************
**********************************************
The total time using GPU: 1.612000 seconds
**********************************************
```

We can see that this time GPU is much faster than CPU. When dealing with large dataset, GPU parallel computing largely improves the speed of the program.

Algorithm 2: Linear Regression

In the hw2, I used linear regression to predict stock price using the returns of 7 days. In this homework, I applied linear regression to a simple dataset x and y: x = [0 1 2 3 4 5] and y=[0 20 60 68 77 110].

To do linear regression, I use ordinary least square to calculate β. To do this, I set the first column of input X to be 1 and the second column to be x. That is:

$$\mathbf{Y} = \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} 1 & X_{11} & \dots & X_{1p} \\ \vdots & & & \\ 1 & X_{n1} & \dots & X_{np} \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ \vdots \\ w_p \end{bmatrix}$$

$$\mathbf{Y} = \mathbf{X}\mathbf{w} + \epsilon$$
$$Y_i = w_0 + w_1 X_{i1} + \cdots + w_p X_{ip}$$

(reference: STAT W4240 Section 1 Data Mining Giovanni Motta Lecture9_Sec01)

And X and y look like:

```
X:
1.0000  0.0000
1.0000  1.0000
1.0000  2.0000
1.0000  3.0000
1.0000  4.0000
1.0000  5.0000

y:
0.0000  20.0000 60.0000 68.0000 77.0000 110.0000
```

I use the equation:

$$\hat{\mathbf{w}} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{Y}$$

to calculate w.

First, calculate X transpose:

```
#define A 6
#define B 2
#define C 1

__global__ void transpose(float *odata, float* idata, int ny, int nx)
{
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < nx && iy < ny)
    {
        odata[ix * ny + iy] = idata[iy * nx + ix];
    }
}
```

```
//-------------------- matrx initialization for transpose --------------------
float *d_a, *d_b;

cudaMalloc(&d_a, A * B*sizeof(float));
cudaMalloc(&d_b, B * A*sizeof(float));

cudaMemcpy(d_a, X, A * B*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, B * A*sizeof(float), cudaMemcpyHostToDevice);

int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((B + block.x - 1) / block.x, (A + block.y - 1) / block.y);

transpose << <grid, block >> >(d_b, d_a, A, B);

cudaMemcpy(b, d_b, B * A*sizeof(float), cudaMemcpyDeviceToHost);
```

Then calculate the product of X transpose and X:

```
__global__ void product(float *a, float *b, float *c, int aa, int bb, int cc)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    if (ix < aa && iy < cc)
    {
        float sum = 0;
        for (int index = 0; index < bb; index++)
        {
            sum += a[ix * bb + index] * b[index * cc + iy];
        }
        c[ix * cc + iy] = sum;
    }
}
```

```
//-------------------- matrx initialization for multiplication X'*X--------------------
float *c;
c = (float *)malloc(B * B*sizeof(float));
float *d_c;

//start using GPU to multiply matrix
cudaMalloc(&d_c, B * B*sizeof(float));
cudaMemcpy(d_c, c, B * B*sizeof(float), cudaMemcpyHostToDevice);

dim3 grid2((B + block.x - 1) / block.x, (B + block.y - 1) / block.y);

product << <grid2, block >> >(d_b, d_a, d_c, B, A, B);

cudaMemcpy(c, d_c, B * B*sizeof(float), cudaMemcpyDeviceToHost);
```

Then calculate the inverse of the product of X transpose an X:

```
#define PERR(call) \
if (call) {\
    fprintf(stderr, "%s:%d Error [%s] on "#call"\n", __FILE__, __LINE__, \
    cudaGetErrorString(cudaGetLastError())); \
    exit(1); \
}
#define ERRCHECK \
if (cudaPeekAtLastError()) {\
    fprintf(stderr, "%s:%d Error [%s]\n", __FILE__, __LINE__, \
    cudaGetErrorString(cudaGetLastError())); \
    exit(1); \
}
```

```
__global__ void inv_kernel(float *a_i, float *c_o, int n)
{
    int *p = (int *)malloc(3 * sizeof(int));
    int *info = (int *)malloc(sizeof(int));
    int batch;
    cublasHandle_t hdl;
    cublasStatus_t status = cublasCreate_v2(&hdl);

    info[0] = 0;
    batch = 1;
    float **a = (float **)malloc(sizeof(float *));
    *a = a_i;
    const float **aconst = (const float **)a;
    float **c = (float **)malloc(sizeof(float *));
    *c = c_o;
    status = cublasSgetrfBatched(hdl, n, a, n, p, info, batch);
    __syncthreads();
    status = cublasSgetriBatched(hdl, n, aconst, n, p,
        c, n, info, batch);
    __syncthreads();
    cublasDestroy_v2(hdl);
}
static void run_inv(float *in, float *out, int n)
{
    float *a_d, *c_d;

    PERR(cudaMalloc(&a_d, n*n*sizeof(float)));
    PERR(cudaMalloc(&c_d, n*n*sizeof(float)));
    PERR(cudaMemcpy(a_d, in, n*n*sizeof(float), cudaMemcpyHostToDevice));

    inv_kernel << <1, 1 >> >(a_d, c_d, n);

    cudaDeviceSynchronize();
    ERRCHECK;

    PERR(cudaMemcpy(out, c_d, n*n*sizeof(float), cudaMemcpyDeviceToHost));
    PERR(cudaFree(a_d));
    PERR(cudaFree(c_d));
}

//-------------------- pinv(X'*X) --------------------
float *invmatrix;
invmatrix = (float *)malloc(B * B*sizeof(float));
run_inv(c, invmatrix, B);
```

Finally, use the product function to calculate the final result:

```
//-------------------- pinv(X'*X)*X' --------------------
float *invma;
cudaMalloc(&invma, B * B*sizeof(float));
cudaMemcpy(invma, invmatrix, B * B*sizeof(float), cudaMemcpyHostToDevice);

float *e;
e = (float *)malloc(B * A*sizeof(float));
float *d_e;
cudaMalloc(&d_e, B * A*sizeof(float));
cudaMemcpy(d_e, e, B * A*sizeof(float), cudaMemcpyHostToDevice);

dim3 grid3((B + block.x - 1) / block.x, (A + block.y - 1) / block.y);
product << <grid3, block >> >(invma, d_b, d_e, B, B, A);
cudaMemcpy(e, d_e, B * A*sizeof(float), cudaMemcpyDeviceToHost);

//-------------------- pinv(X'*X)*X'*y' --------------------
float *res;
res = (float *)malloc(B * C*sizeof(float));
float *d_res;
cudaMalloc(&d_res, B * C*sizeof(float));
cudaMemcpy(d_res, res, B * C*sizeof(float), cudaMemcpyHostToDevice);

float *d_y;
cudaMalloc(&d_y, A * C*sizeof(float));
cudaMemcpy(d_y, y, A * C*sizeof(float), cudaMemcpyHostToDevice);

dim3 grid4((B + block.x - 1) / block.x, (C + block.y - 1) / block.y);
product << <grid4, block >> >(d_e, d_y, d_res, B, A, C);
cudaMemcpy(res, d_res, B * C*sizeof(float), cudaMemcpyDeviceToHost);
```

Then we have:

```
the result is:
3.7619
20.8286

-------------------------------------------
The regression model is:y = 20.8286 * x + 3.7619
-------------------------------------------
```

The output of this part is:

```
C:\Users\hpan4\Desktop\lr\lr>cuda-memcheck ./kernel
========= CUDA-MEMCHECK
X:
1.0000  0.0000
1.0000  1.0000
1.0000  2.0000
1.0000  3.0000
1.0000  4.0000
1.0000  5.0000

y:
0.0000  20.0000 60.0000 68.0000 77.0000 110.0000

X':
1.0000  1.0000  1.0000  1.0000  1.0000  1.0000
0.0000  1.0000  2.0000  3.0000  4.0000  5.0000

X'*X:
6.0000  15.0000
15.0000 55.0000

pinv(X'*X):
0.5238, -0.1429,
-0.1429, 0.0571,

pinv(X'*X)*X:
0.5238  0.3810  0.2381  0.0952  -0.0476 -0.1905
-0.1429 -0.0857 -0.0286 0.0286  0.0857  0.1429

the result is:
3.7619
20.8286

------------------------------------------
The regression model is:y = 20.8286 * x + 3.7619
------------------------------------------
========= ERROR SUMMARY: 0 errors
```

I went through this process in MATLAB and got the same result:

```
>> X'*X

ans =

     6    15
    15    55

>> pinv(X'*X)

ans =

    0.5238   -0.1429
   -0.1429    0.0571

>> pinv(X'*X)*X'

ans =

    0.5238    0.3810    0.2381    0.0952   -0.0476   -0.1905
   -0.1429   -0.0857   -0.0286    0.0286    0.0857    0.1429

>> pinv(X'*X)*X'*y

ans =

    3.7619
   20.8286
```

To verify this result, I use function polyfit in MATLAB and got exactly the same answer (polyfit change the position of β(0) and β(1)).

```
>> x=[0 1 2 3 4 5];
>> y=[0 20 60 68 77 110];
>> coef=polyfit(x,y,1);
>> coef

coef =

   20.8286    3.7619
```

Thus this function can be used to do linear regression. And we can also apply this algorithm to large dataset.