Homework 5

Dustin (Ting-Hsuan) Ma

I. Introduction

In this homework assignment, I implemented Basic Linear Algebra Subprograms (BLAS) function DGEMM into my matrix multiplication code. DGEMM is a level 3 function that takes in matrices of type double and multiplies them. I took my existing code from project 2, and added on DGEMM in order to compare the speed up between DDOT, DAXPY, and serial/parallel matrix multiplication.

II. Method

Continue from my project 2 code, I was able to add in my implementation of DGEMM to compare wall time using different mesh values with other methods. From [1] and [2], I learned how to implement DGEMM for the the CPU and GPU, respectively. For the GPU portion, a block size of 16 was used to run the timing comparison portion of the assignment.

III. Result

A. CPU

Using the intel documentation [1], I was able to implement DGEMM into my exisiting code with ease. In Table I, results are printed in milliseconds (ms), and significant speed up can be seen in the level 3 DGEMM method compared to the others.

TABLE I
CPU Comparison (ms)

N	CPU Mult	Ddot	Daxpy	Dgemm
100	1.126	1.271	1.154	0.871
500	157.549	170.527	432.903	97.270
1000	1261.86	1404.857	3424.589	756.206
2000	22347.985	22131.350	42428.332	7839.65
5000	425979.35	428617.164	743113.217	127956.53

B. GPU

Using the CUDA toolkit documentation [2], I found out that CUDA reads matrices in column majors instead of the row major code I use in my code. In Table II, results are printed in milliseconds (ms), and significant speed up can be seen in the Kernel and the level 3 DGEMM method compared to the times of DDOT and DAXPY.

TABLE II
GPU Comparison (ms)

N	GPU Mult	Ddot	Daxpy	Dgemm
100	0.340	327.620	37.070	0.380
500	2.270	7847.130	882.180	1.950
1000	12.320	31656.340	3597.480	11.590
2000	78.630	123974.760	14219.760	67.080
5000	961.700	784770.120	88685.240	965.000

C. Matrix Output

Matrix output in Section V and VI are printed from CRC into an "output.txt" file. The CPU results matches exactly using a 5 by 5 matrix. When looking at the GPU results, the DGEMM matrix is transposed. This change in orientation is due to CUDA using column major instead of of the row major I used to write the rest of my code.

IV. Conclusion

In this assignment, I implemented the level 3 BLAS function into my CPU and GPU code. A 5 by 5 matrix multiplication result was printed out int section V and section VI to show that the different methods returned the same results. In the GPU portion, DGEMM returned the correct output, but in a transposed format. I believe this is caused by CUDA reading matrices in column major instead of row major that I used in the remainder of my code. From the CPU and GPU timing comparison, DGEMM and Kernel operation is more efficient to utilize for matrix multiplication.

References

[1] "Multiplying matrices using dgemm." https://software.intel.com/en-us/mkl-tutorial-c-multiplying-matrices-using-dgemm, Nov 2018.
[2] "cublas." https://docs.nvidia.com/cuda/cublas/index.html.

V. CPU Matrix Output

```
=====Matrix A=====
    4.00 ;     7.00 ;     8.00 ;     6.00 ;     4.00 ;
    6.00 ;     7.00 ;     3.00 ;    10.00 ;     2.00 ;
    3.00 ;     8.00 ;     1.00 ;    10.00 ;     4.00 ;
    7.00 ;     1.00 ;     7.00 ;     3.00 ;     7.00 ;
    2.00 ;     9.00 ;     8.00 ;    10.00 ;     3.00 ;

=====Matrix B=====
    1.00 ;     3.00 ;     4.00 ;     8.00 ;     6.00 ;
   10.00 ;     3.00 ;     3.00 ;     9.00 ;    10.00 ;
    8.00 ;     4.00 ;     7.00 ;     2.00 ;     3.00 ;
   10.00 ;     4.00 ;     2.00 ;    10.00 ;     5.00 ;
    8.00 ;     9.00 ;     5.00 ;     6.00 ;     1.00 ;

=====CPU=====
  230.00 ;   125.00 ;   125.00 ;   195.00 ;   152.00 ;
  216.00 ;   109.00 ;    96.00 ;   229.00 ;   167.00 ;
  223.00 ;   113.00 ;    83.00 ;   222.00 ;   155.00 ;
  159.00 ;   127.00 ;   121.00 ;   151.00 ;    95.00 ;
  280.00 ;   132.00 ;   126.00 ;   231.00 ;   179.00 ;

elapsed wall time (CPU) = 1.192093 microseconds

=====DDOT()=====
  230.00 ;   125.00 ;   125.00 ;   195.00 ;   152.00 ;
  216.00 ;   109.00 ;    96.00 ;   229.00 ;   167.00 ;
  223.00 ;   113.00 ;    83.00 ;   222.00 ;   155.00 ;
  159.00 ;   127.00 ;   121.00 ;   151.00 ;    95.00 ;
  280.00 ;   132.00 ;   126.00 ;   231.00 ;   179.00 ;

elapsed wall time (DDOT) = 4.053116 microseconds

=====DAXPY()=====
  230.00 ;   125.00 ;   125.00 ;   195.00 ;   152.00 ;
  216.00 ;   109.00 ;    96.00 ;   229.00 ;   167.00 ;
  223.00 ;   113.00 ;    83.00 ;   222.00 ;   155.00 ;
  159.00 ;   127.00 ;   121.00 ;   151.00 ;    95.00 ;
  280.00 ;   132.00 ;   126.00 ;   231.00 ;   179.00 ;

elapsed wall time (DAXPY) = 2.861023 microseconds

=====DGEMM()=====
  230.00 ;   125.00 ;   125.00 ;   195.00 ;   152.00 ;
  216.00 ;   109.00 ;    96.00 ;   229.00 ;   167.00 ;
  223.00 ;   113.00 ;    83.00 ;   222.00 ;   155.00 ;
  159.00 ;   127.00 ;   121.00 ;   151.00 ;    95.00 ;
  280.00 ;   132.00 ;   126.00 ;   231.00 ;   179.00 ;

elapsed wall time (DGEMM) = 5.960464 microseconds
```

VI. GPU Matrix Output

```
=====Matrix A=====
     4.0 ;      7.0 ;      8.0 ;      6.0 ;      4.0 ;
     6.0 ;      7.0 ;      3.0 ;     10.0 ;      2.0 ;
     3.0 ;      8.0 ;      1.0 ;     10.0 ;      4.0 ;
     7.0 ;      1.0 ;      7.0 ;      3.0 ;      7.0 ;
     2.0 ;      9.0 ;      8.0 ;     10.0 ;      3.0 ;

=====Matrix B=====
     1.0 ;      3.0 ;      4.0 ;      8.0 ;      6.0 ;
    10.0 ;      3.0 ;      3.0 ;      9.0 ;     10.0 ;
     8.0 ;      4.0 ;      7.0 ;      2.0 ;      3.0 ;
    10.0 ;      4.0 ;      2.0 ;     10.0 ;      5.0 ;
     8.0 ;      9.0 ;      5.0 ;      6.0 ;      1.0 ;

=====MultKernel=====
   230.0 ;    125.0 ;    125.0 ;    195.0 ;    152.0 ;
   216.0 ;    109.0 ;     96.0 ;    229.0 ;    167.0 ;
   223.0 ;    113.0 ;     83.0 ;    222.0 ;    155.0 ;
   159.0 ;    127.0 ;    121.0 ;    151.0 ;     95.0 ;
   280.0 ;    132.0 ;    126.0 ;    231.0 ;    179.0 ;

elapsed wall time (GPU) =  0.24 ms
=====cublasDDOT=====
   230.0 ;    125.0 ;    125.0 ;    195.0 ;    152.0 ;
   216.0 ;    109.0 ;     96.0 ;    229.0 ;    167.0 ;
   223.0 ;    113.0 ;     83.0 ;    222.0 ;    155.0 ;
   159.0 ;    127.0 ;    121.0 ;    151.0 ;     95.0 ;
   280.0 ;    132.0 ;    126.0 ;    231.0 ;    179.0 ;

elapsed wall time (cublasDDOT) =  7.84 ms
=====cublasDAXPY=====
   230.0 ;    125.0 ;    125.0 ;    195.0 ;    152.0 ;
   216.0 ;    109.0 ;     96.0 ;    229.0 ;    167.0 ;
   223.0 ;    113.0 ;     83.0 ;    222.0 ;    155.0 ;
   159.0 ;    127.0 ;    121.0 ;    151.0 ;     95.0 ;
   280.0 ;    132.0 ;    126.0 ;    231.0 ;    179.0 ;

elapsed wall time (cublasDAXPY) =  0.66 ms
=====cublasDGEMM=====
   230.0 ;    216.0 ;    223.0 ;    159.0 ;    280.0 ;
   125.0 ;    109.0 ;    113.0 ;    127.0 ;    132.0 ;
   125.0 ;     96.0 ;     83.0 ;    121.0 ;    126.0 ;
   195.0 ;    229.0 ;    222.0 ;    151.0 ;    231.0 ;
   152.0 ;    167.0 ;    155.0 ;     95.0 ;    179.0 ;

elapsed wall time (cublasDGEMM) =  0.07 ms
```

```c
/*
 * Purpose: Demonstrate and time matrix multiplication on the CPU
 *
 * Date and time: 04/09/2014
 * Last modified: 03/16/2016
 * Author: Inanc Senocak
 *
 * to compile: gcc -O3 -lcblas -o CPU.exe cpu_matrixMultiply.c
 * to execute: ./CPU <m> <n> <k>
 *
 */

#include "timer.h"
#include <cblas.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <time.h>

typedef double REAL;

void printMatrix(REAL *matrix, const int nrow, const int ncol)
{
        int i, j, idx;

        for (j = 0; j < nrow; j++) {
                for (i = 0; i < ncol; i++) {
                        idx = i + j * ncol;
                        printf("%8.2f ; ", matrix[idx]);
                }
                printf("\n");
        }
        printf("\n");
}

void InitializeMatrices(REAL *a, REAL *b, const int M, const int N, const int K)
{
        int i, j, idx;

        // initialize matrices a & b
        for (j = 0; j < M; j++) {
                for (i = 0; i < K; i++) {
                        idx    = i + j * K;
                        a[idx] = (REAL) idx;
                }
        }

        for (j = 0; j < K; j++) {
                for (i = 0; i < N; i++) {
                        idx    = i + j * N;
                        b[idx] = (REAL) idx;
                }
        }
}

void RandomInitialization(REAL *a, REAL *b, const int M, const int N, const int K)
{
        int i, j, idx;
        for (j = 0; j < M; j++) {
                for (i = 0; i < K; i++) {
                        idx    = i + j * K;
                        a[idx] = (REAL)(rand() % 10) + 1.0;
                }
        }
        for (j = 0; j < K; j++) {
                for (i = 0; i < N; i++) {
                        idx    = i + j * N;
                        b[idx] = (REAL)(rand() % 10) + 1.0;
                }
        }
}

void matrixMultiply(REAL *a, REAL *b, REAL *c, const int M, const int N, const int K)
{
        // this function does the following matrix multiplication c = a * b
        // a(m x k); b(k x n); c(m x n)

        int  i, j, idk, idx;
        REAL sum = 0.f;
        // multiply the matrices C=A*B
        for (i = 0; i < N; i++) {
                for (j = 0; j < M; j++) {
                        for (idk = 0; idk < K; idk++) {
                                sum += a[idk + j * K] * b[i + idk * N];
                        }
```



```c
                        c[i + j * N] = sum;
                        sum          = 0.f;
                }
        }
}

void my_ddot(REAL *A, REAL *B, REAL *C, const int M, const int N, const int K)
{
        int i, j;
        for (j = 0; j < M; j++) {
                for (i = 0; i < N; i++) {
                        C[i + j * N] = cblas_ddot(K, A + j * K, 1, B + i, N);
                }
        }
}

double my_daxpy(REAL *A, REAL *B, REAL *C, const int M, const int N, const int K)
{
        int i, idk;
        for (i = 0; i < N; i++) {
                for (idk = 0; idk < K; idk++) {
                        cblas_daxpy(M, B[i + idk * N], A + idk, K, C + i, N);
                }
        }
}

double my_dgemm(REAL *A, REAL *B, REAL *C, const int M, const int N, const int K)
{
        REAL alpha = 1.0;
        REAL beta  = 0.0;
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha, A, K, B, N, beta, C,
N);
}

int main(int argc, char *argv[])
{
        if (argc < 3) {
                perror("Command-line usage: executableName <m> <k> <n>");
                exit(1);
        }

        int M = atof(argv[1]);
        int K = atof(argv[2]);
        int N = atof(argv[3]);

        REAL *a = (REAL *) calloc(M * K, sizeof(*a));
        REAL *b = (REAL *) calloc(K * N, sizeof(*b));

        REAL *c = (REAL *) calloc(M * N, sizeof(*c)); // Used for CPU
        REAL *d = (REAL *) calloc(M * N, sizeof(*d)); // Used for DDOT
        REAL *e = (REAL *) calloc(M * N, sizeof(*e)); // Used for DAXPY
        REAL *f = (REAL *) calloc(M * N, sizeof(*f)); // Used for DGEMM

        //      InitializeMatrices(a, b, M, N, K);
        RandomInitialization(a, b, M, N, K);
        /*
            printf("=====Matrix A=====\n");
            printMatrix(a,M,K);
            printf("=====Matrix B=====\n");
            printMatrix(b,K,N);
        */
        double startCPU, finishCPU, elapsedTimeCPU;
        GET_TIME(startCPU);
        matrixMultiply(a, b, c, M, N, K);
        GET_TIME(finishCPU);
        elapsedTimeCPU = finishCPU - startCPU;

        printf("=====CPU=====\n");
        printf("CPU C[2] = %3.1f\n", c[2]);
        //      printMatrix(c, M, N);
        printf("elapsed wall time (CPU) = %.6f ms\n", elapsedTimeCPU * 1.0e3);
        printf("\n");

        double startDDOT, finishDDOT, elapsedTimeDDOT;
        GET_TIME(startDDOT);
        my_ddot(a, b, d, M, N, K);
        GET_TIME(finishDDOT);
        elapsedTimeDDOT = finishDDOT - startDDOT;

        printf("=====DDOT()=====\n");
        printf("DDOT d[2] = %3.1f\n", d[2]);
        //      printMatrix(d, M, N);
        printf("elapsed wall time (DDOT) = %.6f ms\n", elapsedTimeDDOT * 1.0e3);
        printf("\n");

        double startDAXPY, finishDAXPY, elapsedTimeDAXPY;
```

```
        GET_TIME(startDAXPY);
        my_daxpy(a, b, e, M, N, K);
        GET_TIME(finishDAXPY);
        elapsedTimeDAXPY = finishDAXPY - startDAXPY;

        printf("=====DAXPY()=====\n");
        printf("DAXPY e[2] = %3.1f\n", e[2]);
        //      printMatrix(e, M, N);
        printf("elapsed wall time (DAXPY) = %.6f ms\n", elapsedTimeDAXPY * 1.0e3);
        printf("\n");

        double startDGEMM, finishDGEMM, elapsedTimeDGEMM;
        GET_TIME(startDGEMM);
        my_dgemm(a, b, f, M, N, K);
        GET_TIME(finishDGEMM);
        elapsedTimeDGEMM = finishDGEMM - startDGEMM;

        printf("=====DGEMM()=====\n");
        printf("DAXPY e[2] = %3.1f\n", e[2]);
        //      printMatrix(f, M, N);
        printf("elapsed wall time (DGEMM) = %.6f ms\n", elapsedTimeDGEMM * 1.0e3);
        printf("\n");

        // Deallocating Memory
        free(a);
        a = NULL;
        free(b);
        b = NULL;
        free(c);
        c = NULL;
        free(d);
        d = NULL;
        free(e);
        e = NULL;
        free(f);
        f = NULL;
        return (EXIT_SUCCESS);
}
```

```
/*
 * Purpose: Demonstrate matrix multiplication in
 * CPU and GPU with global memory and shared memory usage
 * Date and time: 04/09/2014
 *
 * Last modified: Dustin (Ting-Hsuan) Ma
 * Date : November 20, 2018
 * Author: Inanc Senocak
 *
 * to compile blas: nvcc -lcublas -O3 gpu_matrixMultiply.cu -o GPU.exe
 * to execute: ./matrixMult.exe <m> <n> <k>
 */

#include "cublas_v2.h"
#include "timer.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <time.h>

#define BLOCKSIZE 16

typedef double REAL;
typedef int    INT;

void printMatrix(REAL *matrix, const int nrow, const int ncol)
{
        int i, j, idx;

        for (j = 0; j < nrow; j++) {
                for (i = 0; i < ncol; i++) {
                        idx = i + j * ncol;
                        printf("%8.1f ;", matrix[idx]);
                }
                printf("\n");
        }
        printf("\n");
}

void InitializeMatrices(REAL *a, REAL *b, const int M, const int N, const int K)
{
        int i, j, idx;

        // initialize matrices a & b
        for (j = 0; j < M; j++) {
                for (i = 0; i < N; i++) {
                        idx    = i + j * N;
                        a[idx] = (REAL) idx;
                }
        }

        for (j = 0; j < N; j++) {
                for (i = 0; i < K; i++) {
                        idx    = i + j * K;
                        b[idx] = (REAL) idx;
                }
        }
}

void RandomInitilization(REAL *a, REAL *b, const int M, const int N, const int K)
{
        int i, j, idx;

        // initialize matrices a & b
        for (j = 0; j < M; j++) {
                for (i = 0; i < N; i++) {
                        idx    = i + j * N;
                        a[idx] = (REAL)(rand() % 10) + 1.0;
                }
        }

        for (j = 0; j < N; j++) {
                for (i = 0; i < K; i++) {
                        idx    = i + j * K;
                        b[idx] = (REAL)(rand() % 10) + 1.0;
                }
        }
}

__global__ void matrixMultiplyGPU_gl(const REAL *a, const REAL *b, REAL *c, const int M,
                                     const int N, const int K)
{
        // Block index

        int bx = blockIdx.x;
```



```
        int by = blockIdx.y;

        // Thread index

        int tx = threadIdx.x;
        int ty = threadIdx.y;

        // Row index of matrices a and c

        int row = by * BLOCKSIZE + ty;

        // Column index of matrices a and b
        int col = bx * BLOCKSIZE + tx;

        REAL C_temp = 0.;

        if (row < M && col < K) {
                for (int idk = 0; idk < N; idk++)
                        C_temp += a[idk + row * N] * b[col + idk * K];

                c[col + row * K] = C_temp;
        }
}

int main(INT argc, char *argv[])
{
        if (argc < 3) {
                perror("Command-line usage: executableName <M> <N> <K>");
                exit(1);
        }

        int M = atof(argv[1]);
        int N = atof(argv[2]);
        int K = atof(argv[3]);

        REAL *a_d, *b_d, *c_d, *d_d, *e_d, *f_d;

        cudaMallocManaged(&a_d, M * N * sizeof(*a_d));
        cudaMallocManaged(&b_d, N * K * sizeof(*b_d));
        cudaMallocManaged(&c_d, M * K * sizeof(*c_d)); // Used for GPU
        cudaMallocManaged(&d_d, M * K * sizeof(*d_d)); // Used for cublasDDOT
        cudaMallocManaged(&e_d, M * K * sizeof(*e_d)); // Used for cublasDAXPY
        cudaMallocManaged(&f_d, M * K * sizeof(*f_d)); // Used for cublasDGEMM

        //      InitializeMatrices(a_d, b_d, M, N, K);

        RandomInitilization(a_d, b_d, M, N, K);
        /*
                printf("=====Matrix A=====\n");
                printMatrix(a_d,M,N);
                printf("=====Matrix B=====\n");
                printMatrix(b_d,N,K);
        */

        // Setting up GPU enviorment
        dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
        dim3 dimGrid((K + (BLOCKSIZE - 1)) / BLOCKSIZE, (M + (BLOCKSIZE - 1)) / BLOCKSIZE);
        //      dim3 dimGrid( K/BLOCKSIZE+1, M/BLOCKSIZE+1);

        float elapsedTime_gpu, elapsedTime_DDOT, elapsedTime_DAXPY, elapsedTime_DGEMM;

        printf("=====MultKernel=====\n");
        cudaEvent_t timeStart, timeStop; // WARNING!!! use events only to time the device
        cudaEventCreate(&timeStart);
        cudaEventCreate(&timeStop);
        cudaEventRecord(timeStart, 0);

        matrixMultiplyGPU_gl<<<dimGrid, dimBlock>>>(a_d, b_d, c_d, M, N, K);

        cudaDeviceSynchronize();
        cudaEventRecord(timeStop, 0);
        cudaEventSynchronize(timeStop);
        cudaEventElapsedTime(&elapsedTime_gpu, timeStart, timeStop);
        //      printMatrix( c_d, M, K );
        printf("C[2] = %3.1f\n", c_d[2]);
        printf("elapsed wall time (GPU) = %5.2f ms\n", elapsedTime_gpu);

        printf("=====cublasDDOT=====\n");
        cublasHandle_t handle;
        cublasCreate(&handle);

        cudaEventRecord(timeStart, 0);

        for (int i = 0; i < M; i++) {
                for (int j = 0; j < K; j++) {
                        cublasDdot(handle, N, a_d + j * N, 1, b_d + i, K, d_d + i + j * K);
```

```
                }
        }

        cudaEventRecord(timeStop, 0);
        cudaEventSynchronize(timeStop);
        cudaEventElapsedTime(&elapsedTime_DDOT, timeStart, timeStop);
        //      printMatrix( d_d, M, K );
        printf("D[2] = %3.1f\n", d_d[2]);
        printf("elapsed wall time (cublasDDOT) = %5.2f ms\n", elapsedTime_DDOT);

        printf("=====cublasDAXPY=====\n");
        cudaEventRecord(timeStart, 0);

        for (int j = 0; j < M; j++) {
                for (int i = 0; i < K; i++) {
                        cublasDaxpy(handle, M, b_d + j + i * K, a_d + i, N, e_d + j, K);
                }
        }

        cudaEventRecord(timeStop, 0);
        cudaEventSynchronize(timeStop);
        cudaEventElapsedTime(&elapsedTime_DAXPY, timeStart, timeStop);
        //      printMatrix( e_d, M, K );
        printf("E[2] = %3.1f\n", e_d[2]);
        printf("elapsed wall time (cublasDAXPY) = %5.2f ms\n", elapsedTime_DAXPY);

        printf("=====cublasDGEMM=====\n");
        const REAL alpha = 1.0, beta = 0.0;
        cudaEventRecord(timeStart, 0);

        //      cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, M, N, K, &alpha, a_d, M, b_d, K, &beta
, f_d, M
        //);
        cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, M, N, K, &alpha, a_d, M, b_d, N, &beta, f_d, K
);

        cudaEventRecord(timeStop, 0);
        cudaEventSynchronize(timeStop);
        cudaEventElapsedTime(&elapsedTime_DGEMM, timeStart, timeStop);
        //        printMatrix( f_d, M, K );
        printf("E[2] = %3.1f\n", e_d[2]);
        printf("elapsed wall time (cublasDGEMM) = %5.2f ms\n", elapsedTime_DGEMM);
        printf("\n");
        cublasDestroy(handle);

        // Deallocating Memory
        cudaFree(a_d);
        cudaFree(b_d);
        cudaFree(c_d);
        cudaFree(d_d);
        cudaFree(e_d);
        cudaEventDestroy(timeStart);
        cudaEventDestroy(timeStop);

        return (EXIT_SUCCESS);
}
```