Dustin (Ting-Hsuan) Ma
ME 2054
November 20, 2018

Project 2 – Function Utilization

In this project, I tested out the different functions built into BLAS on C and CUDA. I used Ddot, and Daxpy with the CPU and cudablasDdot and cudablasDaxpy with the GPU. The goal of this project is testing the efficiency of different functions. First, we wrote the base function to do matrix multiplication. Then, I implemented the Ddot and Daxpy. I did the exact same procedure for the GPU. Finally, I created a time vs. Mesh size to compare the efficiency.

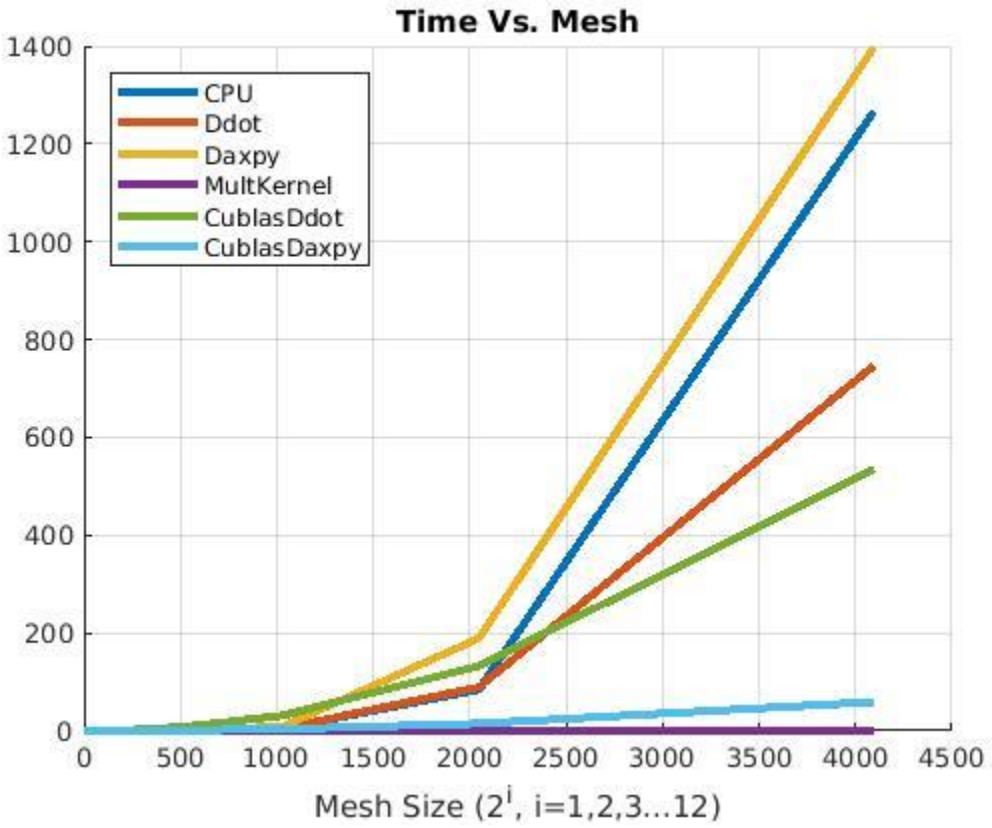

Figure 1 Time Vs. Mesh for 6 different methods of matrix multiplication

In figure 1. You can see that multKernel is the most efficient form to compute matrix multiplication. On the other hand, level cblas Daxby does not improve in efficiency. This graph does not look exactly like the one provided in the project 2 descriptions.

```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 -O2 -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 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[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)
```



```c
{
        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);
                }
        }
}

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

        InitializeMatrices(a, b, M, N, K);

        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 microseconds\n", elapsedTimeCPU * 1.0e6);
        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 microseconds\n", elapsedTimeDDOT * 1.0e6);
        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 microseconds\n", elapsedTimeDAXPY * 1.0e6);
        printf("\n");

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

```c
/*
 * 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 -O2 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("%8.2f ; ", 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 < 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;
                }
        }
}

__global__ void matrixMultiplyGPU_gl(REAL *a, 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.;

        for (int k = 0; k < N; k++)
                C_temp += a[k + row * N] * b[col + k * K];

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



```c
}

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;

        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

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

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

        float elapsedTime_gpu, elapsedTime_DDOT, elapsedTime_DAXPY;

        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("\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);
}
```