ME 571 Project 2: Matrix Multiplication Methods

Gurpal Singh

October 15, 2019

Abstract

The purpose of this project was to compare different methods of implementing matrix multiplication on the CPU as well as the GPU. Specifically, six different methods were used which involved using the "cblas" and "cublas" libraries. The results show that the GPU kernel was the most efficient method to perform matrix multiplication.

1 Introduction

In this assignment, parallel computing was explored. CUDA, a parallel computing platform and application programing interface, was used to explore parallel computing. Matrix multiplication was done using different methods. Overall six methods were used which consisted of performing matrix multiplication using for-loops on the CPU, the level-1 CBLAS function cblas_ddot(), the level-1 CBLAS function cblas_daxpy(), matrix multiplication on the GPU, the CUBLAS function cublasDdot(), and lastly the CUBLAS function cublasDaxpy(). The results are presented in a later section of this report and are also saved to the file "Matrix_Multiplication_Results.txt." To compile the code make sure to load gcc version 4.8.1, cuda version 7.5, and gsl version 2.3. From the source code directory you can compile by typing "make", zip using "make tar", and clean using "make clean." The executable will be located in the bin directory and will ask the user for the matrix dimension n.

2 Matrix Multiplication Methods

This section describes the methods used in detail. The code was tested for small matrix sizes to see if it was producing the right results which it was.

2.1 Method 1: CPU Matrix Multiplication

The CPU matrix multiplication was straightforward. This method uses nested for-loops to compute the C_{ij} component by performing component of component multiplication of matrices A and B. The for-loop essentially computes the dot product of the row of matrix A with the column of Matrix B.

2.2 Method 2: Level-1 CBLAS function cblas_ddot()

This method implemented the use of the CBLAS library. The cblas_ddot() function requires the following arguments in this order.

int N: The number of elements in a vector

double *X: A vector X of length N

int incX: Stride within X

double *Y: A vector Y of length N

int incY: Stride within Y

For the matrix multiplication problem this function was implemented by passing the rows of A and the columns of B to the function within a for-loop to compute each entry of C. The value for incX and incY were 1 since we are computing the dot product which uses every element of a vector. To optimize the multiplication the row of A and columns of B were simply passed as addresses to the function.

2.3 Method 3: Level-1 CBLAS function cblas_daxpy()

This method also makes use of the CBLAS library. The cblas_daxpy() function computes the result of a constant alpha times a vector plus a vector. With this method each column of matrix C can be formed as a linear combination of the columns of A, where the value of alpha takes on the value from the columns of matrix B for each column of matrix A.

$$Y = \alpha X + Y \tag{1}$$

To apply the equation above for matrix multiplication, we must set our input vector Y equal to zero initially since it will also store the result. the columns of A were input as the X vector and the value for alpha changed for each column of A. The example below shows how to compute the first column of C for 3x3 matrices.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$\begin{bmatrix} C_{11} \\ C_{21} \\ C_{31} \end{bmatrix} = B_{11} \begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} + B_{21} \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix} + B_{31} \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

As you can see above, α will take on the value from the B column vector. The implementation of this function was a little trickier than the cblas_ddot() function. Nested for-loops were used to cycle through the columns of matrix A and the values of α The cblas_daxpy() function takes the following arguments in this order.

int N: The number of elements in the vectors

double α: Scaling factor for the values in X (For our case alpha will take its values from B)

double *X: Input vector X

int incX: Stride within X

double *Y: Input vector Y (NOTE: The result is saved to this vector)
int incY: Stride within Y

2.4 Method 4: GPU Kernel Matrix Multiplication

A GPU kernel was created to perform the matrix multiplication. The block size was initialized to be 16 x 16 and the grid size to be (n+15)/16 x (n+15)/16. One for-loop was used to compute the sum of each row of A with each column of B. On the GPU we are able to avoid the nested for-loops and compute the sums separately. The B matrix was first transposed and then passed to the kernel since all matrices were stored as row-major flat arrays. The results show that this was the most efficient method for the square matrix multiplication.

2.5 Method 5: Level-1 CUBLAS function cublasDdot()

The cublasDdot() function is similar to cblasddot() except that it is the CUDA version meaning it makes use of the GPU. The function was called in the same way as cblasddot() and takes the same arguments(see section 2.3).

2.6 Method 6: Level-1 CUBLAS function cublasDaxpy()

This function was similar to cblasdaxpy() and also worked the same way as the cpu version except it made use of the GPU. It was called in the same manner as the cpu version. It turned out, this was the most inefficient way to compute the matrix multiplication.

3 Results

The results were saved to the file named "Matrix_Multiplication_Results.txt". The results are also presented in the graph below. As you can see the GPU kernel is the most efficient method and took about 3.8 s to compute the result for square matrices of size 4096 by 4096. The cblas functions cblas_ddot() and cblas_daxpy() were also fairly efficient and took about 83.8 s and 110.2 s respectively to do a matrix multiplication of size 4096 by 4096. The cublasDdot() and cublasDaxpy() function were not optimal for matrices larger than 1000 by 1000 since they begin to take much longer at larger matrices sizes. For example for matrices of size 4096 by 4096 the cublasDdot() function took about 4064.3 s and the cublasDaxpy() function took about 10092.1 s. For matrices below the size of about 600 x 600 all of the methods take about the same time. However for larger matrices the GPU kernel would be the best method to use.

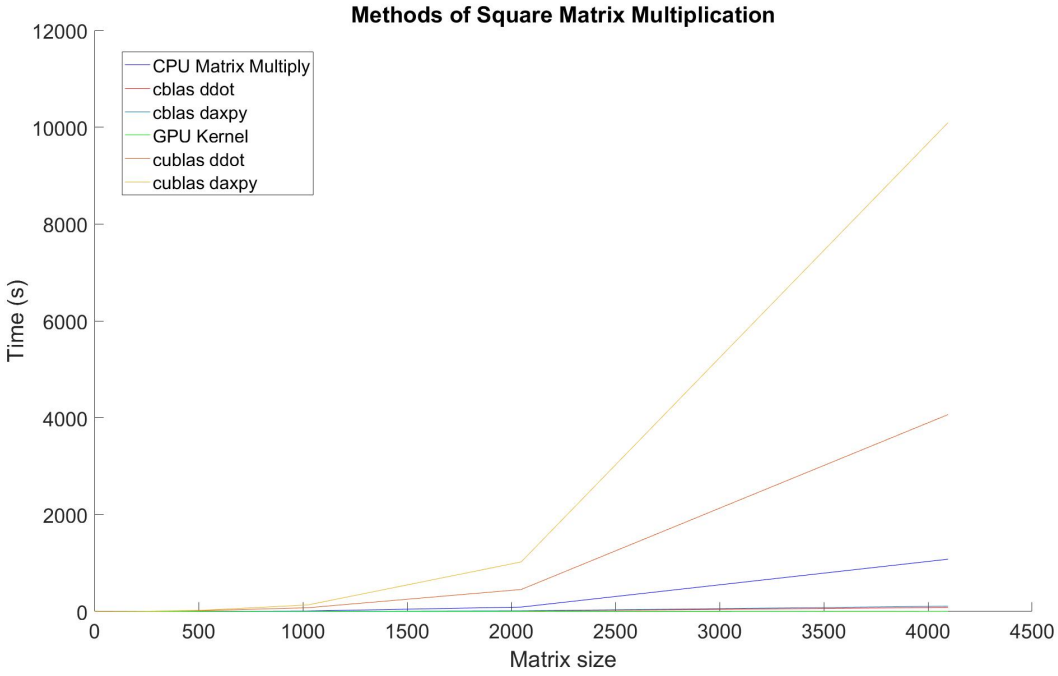

Figure 1: Plot of the methods used for matrix multiplication