**Assignment1 Report**

**CSE436, Summer 2016**

**Kazumi Malhan**

1. Function Implementation Description

* **matrix\_addition function**

The function first checks the storage type of input matrices by checking A\_rowMajor and B\_rowMajor parameters. Based on these parameters, there are four different types of implementation available inside the function. Generally, matrix addition is performed by adding each element at same location of two matrices, and placing the result at same location of result matrix. Following Figure 1 shows simple 2 by 2 matrix addition.

**Figure 1:** Example of matrix addition

Essentially, this can be performed by going through each element of matrix using two for loops, and adding each item. In my implementation, first for loop is used to go through all rows. For each row, second for loop is used to access each column of matrix. All of four implementations follow the same method to calculate matrix addition except how offsets are calculated. Assuming matrix A[N][M] has N rows and M columns, the offset is calculated using Equation 1 for row major matrix whereas column major matrix uses Equation 2 to calculate its offset.

[Row Major] --- **Equation 1**

[Column Major] --- **Equation 2**

The method to calculate offset for both row major matrix and column major matrix are used throughout all functions.

* **matrix\_multiplication**

Similar to matrix\_addition function, this function also checks the storage type of input matrices using input parameters. Depending on these values, different offset calculation is used. Details of method to calculate offset is explained in matrix\_addition function description. Figure 2 shows example matrix multiplication.

**Figure 2:** Example of matrix multiplication

It is important to note that matrix multiplication requires single row of matrix A to be calculated with all columns of matrix B. In code, the first for loop is setup to go through each rows of matrix A, and second for loop is placed to go through every column of matrix B. Since K represents columns of matrix A and rows of matrix B, adding third for loop with K enables function to go through each element of rows in matrix A and corresponding element of columns in matrix B. Two elements are multiplied, and accumulated into “sum” variable. Finally, the result will be placed in matrix C(row number of matrix A, column number of matrix B). By going through three for loops, the function achieves matrix multiplication operation.

* **mv\_multiplication**

mv\_multiplication function also starts with checking the storage type of first matrix using input parameter. This function only has two implementations as only first matrix has option to be row major or column major. The implementation of the function is similar to matrix\_multiplication function except second input matrix is a vector (i.e. only has one column). The Figure 3 shows example of mv multiplication.

**Figure 3:** Example of mv multiplication

The first for loop goes through every row of matrix A, and second for loop iterates through every column of matrix A and every row of vector B (matrix B). The two elements are multiplied, and accumulated into “sum” variable, and stored into result vector C(row number of matrix A). By repeating this process, the function achieves mv multiplication.

2. Performance Report

All execution of assignment 1 code is performed on **lennon.secs.oakland.edu** linux server via VPN connection. Bash script was prepared to automatically perform six variants of test runs, and save the result into a text file. Following three figures shows the performance of each function.

* **Performance Different due to Storage Type**

The cache system plays the key role in performance difference due to storage type. The cache uses locality of reference as key concept. The one of concept is “spatial locality”, which stores surrounding of accessed data into cache. If next data is found in cache, it is significantly efficient than accessing the RAM (Random Access Memory). In matrix\_addition, all matrices access each element in row first, then next column. Therefore, both input matrices being row major gets best performance. Having column major matrix as input makes the function perform slower due to increasing number of RAM access.

For matrix\_multilication, the first input matrix is accessed by row whereas second input matrix is access by column by nature of how matrix multiplication is defined. Therefore, first matrix being row major and second matrix being column major has highest cache hit rate, resulting is best performance time among four possible storage types. (about ½ faster). The opposite combination of storage type gets the worst performance. For mv\_multiplication, the difference is minimum because cache miss caused by second input matrix (vector) is larger than

* **Performance Difference due to Optimization Level**

In this assignment, same code was compiled using different optimization flag level, level 0 and level 3. At flag level 0, there is no optimization, and documentation states that this level is only good for debugging purpose. GNU GCC complier uses optimization flag level 2 as default if this option is not specified. At flag level 2, most of optimization flags are enabled that will not increase the size of code. At optimization flag level 3, more aggressive optimization is performed. The following list shows some of noticeable optimization performed by compiler.

* Replace function call with inline function. This increases the speed as program has fewer branch instructions.
* Perform loop distribution. By distributing the loop, it reducing the number of time the program needs to check the end loop condition.
* Perform “function cloning”. If the function is called several times with similar parameters, the compiler creates similar function with less parameter (making unchanged parameters as constant), and call newly created function.
* Replace standard C library function with faster alternative.
* Remove unused parameters.
* Re-order the instruction to avoid idling.
* Eliminate save and load of registers that are not used by called function.
* Perform “partial redundancy elimination”, which means removing the duplicated code.
* Dead logic removal.

With these optimization techniques, the program ran faster (about 2.5 times) than the one without any optimization. When the size of executable file is compared, one with flag level 3 was slightly larger.

* **Number of Flops Calculation**

Inside main function, MFLOPS was calculated with execution time. In order to calculate MFLOPS, it is necessary to understand number of flops performed by each function. Table 1 below summarizes the equations used for calculation.

|  |  |
| --- | --- |
| Function | # of flop |
| matrix\_addition | M \* N |
| matrix\_multilication | M \* N \* (2 \* K - 1) |
| mv\_multiplication | (2 \* M - 1) \* N |

**Table1:** Number of flops performed by each function

Note: When calculating MFLOPS for matrix\_multiplication, there is a possibility of an overflow with a large number (ex. 2048 \* 2048) at (M \* N \* (2 \* K – 1)) as all variables are declared as int. M variable is type casted with long to avoid an overflow.

3. Conclusion