CS2610 - Assignment 5

CS19B028 Maddula Jaya Kamal

March 2021

Analysing Matrix Multiplication :
A detailed analysis of Matrix Multiplication, time taken and number of memory fetches and instructions implemented

About the Code and Algorithm: This is a C program to multiply 2 square matrices of order 1024 and store the result in the third square matrix. In this code we first allocate heap memory to 3 integer double pointers in order to represent 3 matrices. Now we used **rand()** to get random integer and them we divide them with 11 to find remainder that lies between 0 to 10 and then we assign that as an elements of matrices A and B. After assigning all elements of A , B and C we write a 3 nested for loops to carry out all n^3 multiplications where n is the order of the square matrix. Then we assign all results into C.

Row major: Adjacent elements in a given row are stored in adjacent addresses
Column major: Adjacent elements in a given column are stored in adjacent addresses

Algorithm(Both Row Major):

```
CS19B028_1.c:
for i=0 to 1024 do:
   for j=0 to 1024 do:
     C[i][j] = A[i][0]*B[0][j];
     for k=1 to 1024 do:
             C[i][j] = C[i][j]+A[i][k]*B[k][j];
```

Algorithm(A Row Major, B Column Major):

```
CS19B028_2.c:
for i=0 to 1024 do:
   for j=0 to 1024 do:
     C[i][j] = A[i][0]*B[j][0];
     for k=1 to 1024 do:
             C[i][j] = C[i][j]+A[i][k]*B[j][k];
```

We also have a **gettimeofday()** at the beginning and end of the code.
The function **gettimeofday()** returns the time elapsed from epoch to the current point. When we use **gettimeofday()** twice, once at the beginning of the code and at the and of the code we

can calculate the time elapsed for the execution of code by taking difference between 2nd time stamp and first time stamp. we have sec and usec for calculating seconds and micro-seconds with a great precision.

Processor Info: Intel core i5, 2.3 GHz clock frequency, dual core, L1 cache size 32KB(per core), L2 cache size 256KB(per core), L3 cache Size 4MB(per core), 8GB main memory.

Performance Analysis(Both Row Major)		
S.No	Parameter	Value
1	Average RunTime	1.3860±0.0026*sec*
2	CPU Cycles	4,95,31,08,280
3	Instructions	7,66,94,50,114
4	Branches	1,11,39,98,399
5	Cache References	20,61,26,452
6	Cache Misses	3,12,99,882
7	L1 D Cache Loads	3,26,95,05,332
8	L1 D-Cache Loads Misses	1,28,02,29,035
9	dTLB-Loads	3,25,73,29,900
10	dTLB-Misses	19,38,306
11	LLC Loads	7,85,02,628
12	LLC Load-Misses	84,64,787

Performance Analysis(A Row Major, B Column Major)		
S.No	Parameter	Value
1	Average RunTime	0.4574±0.00781*sec*
2	CPU Cycles	1,81,69,22,613
3	Instructions	4,21,73,10,567
4	Branches	30,66,45,153
5	Cache References	15,80,12,180
6	Cache Misses	6,06,74,668
7	L1 D Cache Loads	1,11,80,62,821
8	L1 D-Cache Loads Misses	6,98,69,329
9	dTLB-Loads	1,11,40,17,476
10	dTLB-Misses	21,24,131
11	LLC Loads	45,97,373
12	LLC Load-Misses	2,05,633

These stats can be obatained by using perf tool. Perf is a performance analysis tool to analyse performance of a give code.

```
gcc CS19B028_1.c -o CS19B028_1.out -O3
gcc CS19B028_2.c -o CS19B028_2.out -O3
```

The above two commands are used to get the output files of C program files. -O3 is used for optimisations and .out files are the output files. These output files are used to obtain performance statistics of the code using the **perf** tool.

Observations:

1. The time taken and CPU cycles for executing second program(A row major, B column major) is far less than the time taken to compute the first program(both row major).
2. The total number of instructions computed in the second program is far less than the number of instructions computed in the first program
3. Cache loads and Cache misses is almost same for both programs for level 1 caches. But LLC cache loads and cache misses of program 2 are very less when compared to program 1.
4. Number of Branches for program 2 is far less than those of program 1.
5. The dTLB loads of program 2 is almost half the number of dTLB loads of program 1, where as the number of dTLB misses are almost same.

Inference:

The program has become really efficient when the 2^{nd} matrix was changed from a row major orientation to a column major orientation.

Reason:

The reason for a very significant change in the run-time is that in the Matrix Multiplication Algorithm the first Matrix is traversed row-vise and each element is multiplied with its corresponding column-wise elements of matrix 2, so most of the traversal in matrix 2 are column-wise. So when the orientation of matrix 2 is changed to column major orientation then most of the column elements of matrix 2 will be loaded into cache due to spatial locality, unlike in row major where row elements are loaded into cache due to spatial locality. So this increase hit rate of the cache which in turn reduces the average memory access time which contributes the maximum for the run-time of the program. So the run time of the code decreases drastically