Assignment Two : Cache Performance

In this assignment, you are going to implement different techniques for multiplying matrices. The idea is to investigate the performance impacts of different cache architectures and different algorithm designs on matrix multiplication. The goals of this assignment are:

- Show how algorithms have different behaviors as the micro-architecture changes.
- Show how changing the algorithm can change performance on the *same* micro-architecture.
- Improve your understanding of cache architectures.

Base matrix multiplication algorithm (C stationary, or ijk)

For this assignment, assume that all A, B, and C matrices are stored in row major order. I.e., matrices are indexed like A[row number][column number]. The starter code first iterates over the rows of A (*i*), then iterates over the columns of B (*j*), and then iterates over the elements in the selected row and column (*k*).

```
        void multiply(double **A, double **B, double **C, int size)
{
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

A stationary matrix multiplication (ikj)

Our nested loops can be rearranged and we will still obtain the right response. In actuality, the C matrix will be produced by all permutations of our three nested loops. Furthermore, changing the sequence of the for loops has no effect on the algorithm's overall complexity. Rearranging the for loops, however, modifies the memory access pattern. Consequently, it can cause our cache hit rate to rise or fall. Below is an alternate implementation of the matrix multiplication program for this phase.

```
void multiply(double **A, double **B, double **C, int size)
{
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            for (int j = 0; j < size; j++) {
                C[i][j] += A[i][k] * B[k][j];
```

```
            }
        }
    }
}
```

Blocked matrix multiplication

Organising data structures in a software into big units called blocks is the basic principle behind blocking. (Here, "block" does not relate to a cache block; rather, it refers to an application-level data chunk.) The way the program is set up, it loads a chunk into the L1 cache, does all necessary reads and writes on that piece, discards it, loads the next chunk, and so on.

```
1  void bijk(array A, array B, array C, int n, int bsize)
2  {
3      int i, j, k, kk, jj;
4      double sum;
5      int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */
6
7      for (i = 0; i < n; i++)
8          for (j = 0; j < n; j++)
9              C[i][j] = 0.0;
10
11     for (kk = 0; kk < en; kk += bsize) {
12         for (jj = 0; jj < en; jj += bsize) {
13             for (i = 0; i < n; i++) {
14                 for (j = jj; j < jj + bsize; j++) {
15                     sum = C[i][j];
16                     for (k = kk; k < kk + bsize; k++) {
17                         sum += A[i][k]*B[k][j];
18                     }
19                     C[i][j] = sum;
20                 }
21             }
22         }
23     }
24 }
```

Use different board modules [different microarchitecture] and cache models [gem5] to demonstrate that writing good code helps in improving performance. Also run in real hardware along with gem5.

Submission:

Please submit a well formatted pdf document containing as many graphs and tables as you can generate.