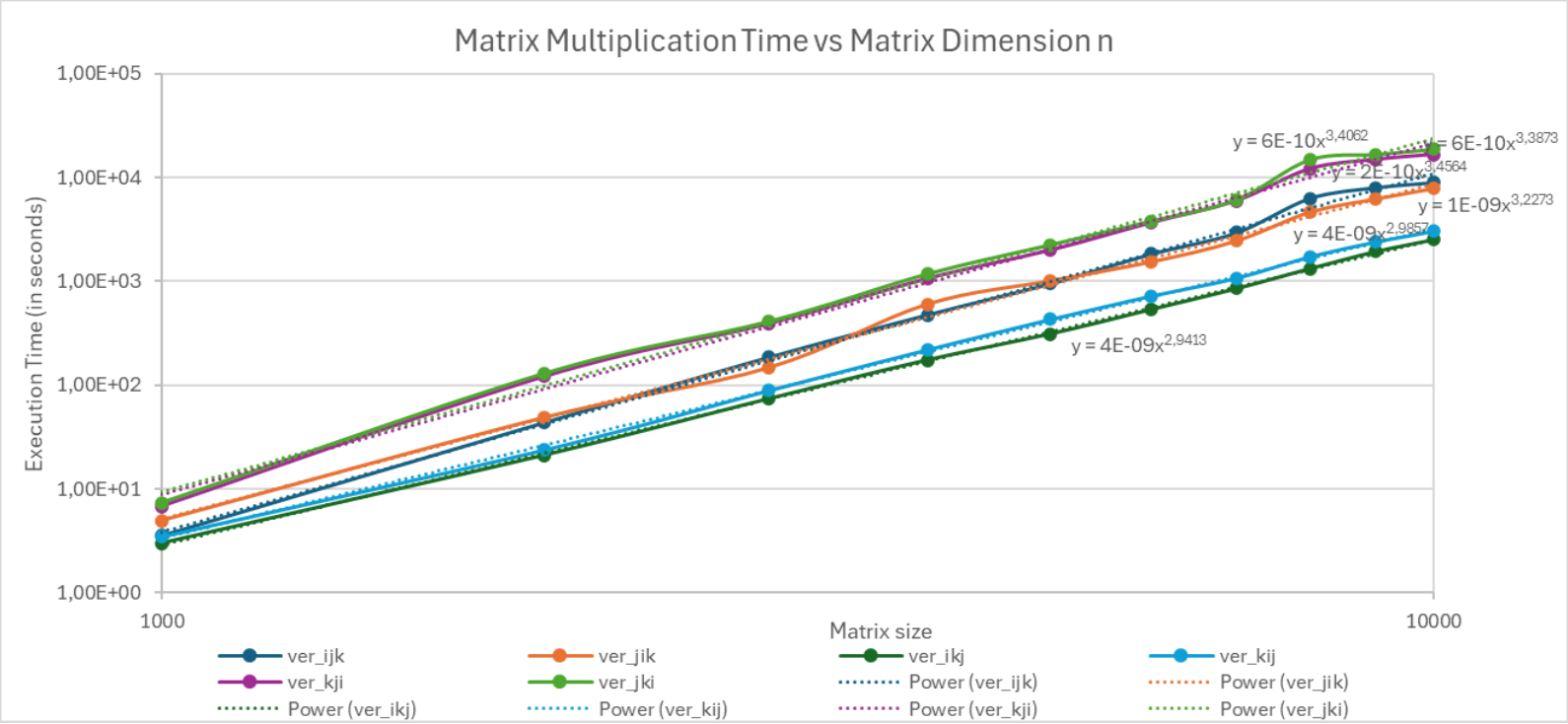
Name: Khang Huy Dinh
ID: 300373471

Matrix Multiplication Time Analysis



First, we calculate the qualitative miss rates of 6 versions, assuming that cache has a size of 32-bits which is able to hold 4 doubles. To calculate the miss rate, we only consider the most inner loop. And based on the miss rate, we will find the most efficient version.

Version 1:
```
void ver_ijk(double* A, double* B, double* C, int n)
{
    . . .
                sum += A[i * n + k] * B[k * n + j];
    . . .
}
```
→ Matrix A access by rows and Matrix B access by columns → miss rate = 0.25 + 1 = 1.25

Version 2:
```
void ver_jik(double* A, double* B, double* C, int n)
{
    . . .
                sum += A[i * n + k] * B[k * n + j];
    . . .
}
```
→ A access by rows, B access by columns → miss rate is also 1.25

Version 3:
```
void ver_ikj(double* A, double* B, double* C, int n)
{
    . . .
        C[i * n + j] += temp * B[k * n + j];
}
```
→ Access to A is first stored in a local variable in a register, C access by rows, B access by rows → miss rate = 0.25 + 0.25 = 0.5

Version 4:
```
void ver_kij(double* A, double* B, double* C, int n)
{
    . . .
                C[i * n + j] += temp * B[k * n + j];
    . . .
}
```
→ Same with version 3 → miss rate: 0.5

Version 5:
```
void ver_kji(double* A, double* B, double* C, int n)
{
    . . .
                C[i * n + j] += A[i * n + k] * temp;
    . . .
}
```
→ C access by columns, A also access by columns → miss rate = 1 + 1 = 2.0

Version 6:
```
void ver_jki(double* A, double* B, double* C, int n)
{
```

Name: Khang Huy Dinh
ID: 300373471

```
    . . .
                C[i * n + j] += A[i * n + k]
* temp;
    . . .
```

```
} → C and A both access by columns → miss
rate = 2.0
```

From the qualitative miss rate calculation, we can estimate the execution time of 6 versions by this ascending order: version 5,6 < version 1,2 < version 3,4 (miss rate: 2.00 > 1.25 > 0.5, respectively).

This estimation is proved to be correct based on the actual running time of the program for each version, showing on the plot. **The lowest curve is what we are looking for, the fastest version are ver_ikj and ver_kij.** Then above version 3,4 are version 1,2 then 5,6 with qualitative miss rate of 1.25 and 2.00.

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

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

The reason for version 3 (ikj) and 4 (kij) to be faster is the better usage of cache. Having a closer look into the 2 implementations above, you can see that in both functions, matrix C and B are accessed by row-major which is cache-friendly because cache is loaded by rows, resulting in much lower miss rate compared to the other 4 versions that have at least 1 matrix being accessed using column-major.

So again, the worst versions are version 5 and 6 (kji and jki) since they access Matrices all by column-major (2.00 miss rate).

The average efficient among 6 of them are the first 2 versions (ijk and jik) since they access 1 matrix by row and 1 by column (1.25 miss rate).

The best versions are version 3 and 4 (ikj and kij) since both of them access matrices all by row-major (0.5 miss rate).

➔CONCLUSION: The fastest version between 6 versions are version 3 (ver_ikj) and version 4 (ver_kij)