Matrix Multiplication Performance Analysis

Name- Parth Singh

Entry Number- 2022ME11438

Experimental Setup

Three versions of matrix multiplication were tested:

1. Without O2 optimization and without vectorization — Case 1
2. With O2 optimization but without vectorization — Case 2
3. With O2 optimization and vectorization — Case 3

The performance was measured for matrix sizes ranging from 50x50 to 800x800.

Performance Results

Without O2 and Vectorization

```
Size: 50, IJK: 0.002039, IKJ: 0.002217, JIK: 0.001623, JKI: 0.002261, KIJ: 0.001826, KJI: 0.001863
Size: 100, IJK: 0.009227, IKJ: 0.010531, JIK: 0.006824, JKI: 0.009628, KIJ: 0.009438, KJI: 0.009640
Size: 150, IJK: 0.022882, IKJ: 0.031683, JIK: 0.022917, JKI: 0.032456, KIJ: 0.031782, KJI: 0.032640
Size: 200, IJK: 0.054140, IKJ: 0.075112, JIK: 0.054201, JKI: 0.077066, KIJ: 0.075232, KJI: 0.077040
Size: 250, IJK: 0.105477, IKJ: 0.146399, JIK: 0.105424, JKI: 0.152614, KIJ: 0.146836, KJI: 0.152606
Size: 300, IJK: 0.191819, IKJ: 0.252756, JIK: 0.193061, JKI: 0.291226, KIJ: 0.253381, KJI: 0.291924
Size: 350, IJK: 0.304412, IKJ: 0.400546, JIK: 0.304886, JKI: 0.463443, KIJ: 0.402665, KJI: 0.463981
Size: 400, IJK: 0.439108, IKJ: 0.597403, JIK: 0.442074, JKI: 0.713151, KIJ: 0.599582, KJI: 0.710151
Size: 450, IJK: 0.625304, IKJ: 0.851333, JIK: 0.626388, JKI: 1.018087, KIJ: 0.854693, KJI: 1.030821
Size: 500, IJK: 0.965530, IKJ: 1.172774, JIK: 0.868108, JKI: 1.457710, KIJ: 1.177078, KJI: 1.607286
Size: 550, IJK: 1.185850, IKJ: 1.587205, JIK: 1.245149, JKI: 1.852379, KIJ: 2.158298, KJI: 2.126235
Size: 600, IJK: 1.568832, IKJ: 2.929117, JIK: 1.542898, JKI: 2.430899, KIJ: 2.037245, KJI: 2.377549
Size: 650, IJK: 1.950212, IKJ: 2.571622, JIK: 1.965709, JKI: 3.024080, KIJ: 2.585790, KJI: 2.988110
Size: 700, IJK: 2.434365, IKJ: 3.205761, JIK: 2.439918, JKI: 3.699862, KIJ: 3.218447, KJI: 3.747071
Size: 750, IJK: 3.033044, IKJ: 3.928166, JIK: 2.991073, JKI: 4.568852, KIJ: 3.945712, KJI: 4.558689
Size: 800, IJK: 3.632123, IKJ: 4.766025, JIK: 3.644185, JKI: 5.587129, KIJ: 4.788356, KJI: 5.752818
```

- For small matrices (50x50), execution times range from 0.001623 to 0.002261 seconds.
- For large matrices (800x800), execution times range from 3.632123 to 5.752818 seconds.

With O2, Without Vectorization

```
Size: 50, IJK: 0.000187, IKJ: 0.000182, JIK: 0.000136, JKI: 0.000229, KIJ: 0.000141, KJI: 0.000229
Size: 100, IJK: 0.001643, IKJ: 0.001182, JIK: 0.001060, JKI: 0.001731, KIJ: 0.001130, KJI: 0.001415
Size: 150, IJK: 0.003655, IKJ: 0.003089, JIK: 0.003123, JKI: 0.004137, KIJ: 0.003469, KJI: 0.003627
Size: 200, IJK: 0.005439, IKJ: 0.005306, JIK: 0.005652, JKI: 0.008099, KIJ: 0.005758, KJI: 0.008110
Size: 250, IJK: 0.009902, IKJ: 0.009836, JIK: 0.010134, JKI: 0.016453, KIJ: 0.010670, KJI: 0.016492
Size: 300, IJK: 0.032085, IKJ: 0.017390, JIK: 0.033456, JKI: 0.083376, KIJ: 0.018085, KJI: 0.079451
Size: 350, IJK: 0.051270, IKJ: 0.026568, JIK: 0.052909, JKI: 0.136075, KIJ: 0.027655, KJI: 0.129131
Size: 400, IJK: 0.074370, IKJ: 0.040150, JIK: 0.078674, JKI: 0.526505, KIJ: 0.041943, KJI: 0.527811
Size: 450, IJK: 0.106846, IKJ: 0.056437, JIK: 0.111608, JKI: 0.721684, KIJ: 0.057747, KJI: 0.705833
Size: 500, IJK: 0.147539, IKJ: 0.075468, JIK: 0.151944, JKI: 1.190108, KIJ: 0.080602, KJI: 1.188377
Size: 550, IJK: 0.199615, IKJ: 0.101452, JIK: 0.203726, JKI: 0.517559, KIJ: 0.108139, KJI: 0.497054
Size: 600, IJK: 0.257684, IKJ: 0.129487, JIK: 0.265098, JKI: 0.662205, KIJ: 0.140701, KJI: 0.655016
Size: 650, IJK: 0.320499, IKJ: 0.165705, JIK: 0.331538, JKI: 0.891346, KIJ: 0.183385, KJI: 0.864350
Size: 700, IJK: 0.398500, IKJ: 0.205854, JIK: 0.414056, JKI: 1.121171, KIJ: 0.223816, KJI: 1.121334
Size: 750, IJK: 0.492981, IKJ: 0.251663, JIK: 0.506907, JKI: 1.370464, KIJ: 0.285263, KJI: 1.340818
Size: 800, IJK: 0.622136, IKJ: 0.306370, JIK: 0.634002, JKI: 1.651025, KIJ: 0.340773, KJI: 1.603404
```

- For small matrices (50x50), execution times range from 0.000136 to 0.000229 seconds.
- For large matrices (800x800), execution times range from 0.306370 to 1.651025 seconds.

With O2 and Vectorization

```
Size: 50, IJK: 0.000120, IKJ: 0.000042, JIK: 0.000119, JKI: 0.000217, KIJ: 0.000038, KJI: 0.000216
Size: 100, IJK: 0.000949, IKJ: 0.000249, JIK: 0.000950, JKI: 0.001427, KIJ: 0.000222, KJI: 0.001417
Size: 150, IJK: 0.002849, IKJ: 0.000633, JIK: 0.002535, JKI: 0.004302, KIJ: 0.000593, KJI: 0.003772
Size: 200, IJK: 0.004153, IKJ: 0.001260, JIK: 0.004215, JKI: 0.008079, KIJ: 0.001641, KJI: 0.008118
Size: 250, IJK: 0.008265, IKJ: 0.002387, JIK: 0.008673, JKI: 0.016312, KIJ: 0.003040, KJI: 0.016388
Size: 300, IJK: 0.029751, IKJ: 0.004760, JIK: 0.031288, JKI: 0.083401, KIJ: 0.005367, KJI: 0.078914
Size: 350, IJK: 0.048611, IKJ: 0.007449, JIK: 0.051026, JKI: 0.139767, KIJ: 0.008199, KJI: 0.133834
Size: 400, IJK: 0.071358, IKJ: 0.009872, JIK: 0.075965, JKI: 0.536414, KIJ: 0.010999, KJI: 0.530861
Size: 450, IJK: 0.102467, IKJ: 0.013889, JIK: 0.107557, JKI: 0.716989, KIJ: 0.015361, KJI: 0.703962
Size: 500, IJK: 0.141213, IKJ: 0.016342, JIK: 0.147066, JKI: 1.184715, KIJ: 0.020813, KJI: 1.173837
Size: 550, IJK: 0.189903, IKJ: 0.025185, JIK: 0.197164, JKI: 0.511875, KIJ: 0.028942, KJI: 0.498665
Size: 600, IJK: 0.245460, IKJ: 0.028838, JIK: 0.255231, JKI: 0.666257, KIJ: 0.037153, KJI: 0.648894
Size: 650, IJK: 0.305496, IKJ: 0.041804, JIK: 0.320035, JKI: 0.888381, KIJ: 0.048079, KJI: 0.866209
Size: 700, IJK: 0.379792, IKJ: 0.050730, JIK: 0.397982, JKI: 1.099489, KIJ: 0.095058, KJI: 1.156038
Size: 750, IJK: 0.467562, IKJ: 0.056707, JIK: 0.487655, JKI: 1.369863, KIJ: 0.074492, KJI: 1.343657
Size: 800, IJK: 0.587537, IKJ: 0.070063, JIK: 0.608328, JKI: 1.668611, KIJ: 0.085342, KJI: 1.645664
```

- For small matrices (50x50), execution times range from 0.000038 to 0.000217 seconds.
- For large matrices (800x800), execution times range from 0.070063 to 1.668611 seconds.

Performance Improvement

1. **O2 Optimization Impact**:
    - Comparing Case 1 and Case 2, we see a significant performance improvement with O2 optimization.
    - For 800x800 matrices, the execution time is reduced from ~3.6-5.7 seconds to ~0.3-1.6 seconds.
2. **Vectorization Impact**:
    - Comparing Case 2 and Case 3, we observe further improvement for some loop orderings with vectorization.
    - The most notable improvement is for the KIJ ordering at 800x800, reducing from 0.340773 to 0.085342 seconds.

Loop Ordering Analysis

The performance varies significantly based on the loop ordering (IJK, IKJ, JIK, JKI, KIJ, KJI):

1. **Without Optimizations (Case 1)**:
    - IJK, IKJ, and JIK perform similarly.
    - JKI and KJI are consistently slower.
    - KIJ shows moderate performance.
2. **With O2 Optimization (Case 2)**:
    - KIJ becomes the fastest ordering.
    - JKI and KJI remain the slowest.
    - Other orderings show similar performance to each other.
3. **With O2 and Vectorization (Case 3)**:
    - KIJ maintains its position as the fastest ordering, with even more pronounced improvement.
    - IKJ shows significant improvement, becoming the second-fastest ordering.
    - JKI and KJI remain the slowest.

Cache Efficiency Inference

Although we don't have direct cache measurements, we can infer some information about cache efficiency:

1. The KIJ ordering performs best with optimizations, likely due to better cache utilization. This ordering accesses matrix elements in a more cache-friendly manner.
2. JKI and KJI consistently perform poorly, suggesting they have poor cache access patterns, leading to more cache misses.

3. The significant performance boost from O2 optimization suggests that it's improving memory access patterns and possibly prefetching, leading to better cache utilization.

Vectorization Effect

The impact of vectorization is most noticeable for certain loop orderings, particularly KIJ:

- For KIJ at 800x800: Time reduced from 0.340773 (O2 only) to 0.085342 seconds (O2 with vectorization).
- This suggests that the KIJ ordering is particularly amenable to SIMD (Single Instruction, Multiple Data) operations, allowing for efficient parallel processing of matrix elements.

Conclusion

1. O2 optimization provides substantial performance improvements across all matrix sizes and loop orderings.
2. Vectorization offers additional speedup, especially for cache-friendly loop orderings like KIJ.
3. The choice of loop ordering significantly impacts performance, with KIJ being the most efficient in optimized scenarios.
4. Cache efficiency plays a crucial role in matrix multiplication performance, as evidenced by the varying performance of different loop orderings.

Code

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
#include <iomanip>
using namespace std;
void multiply_matrices1(const vector<vector<int> >& A,
                        const vector<vector<int> >& B,
                        vector<vector<int> >& C, int n) {
   for(int i = 0; i < n; i++) {
       for(int j = 0; j < n; j++) {
            int s = 0;
           for(int k = 0; k < n; k++) {
               s += A[i][k] * B[k][j];
```

```
            }
            C[i][j] = s;
        }
    }
}
void multiply_matrices2(const vector<vector<int> >& A,
                        const vector<vector<int> >& B,
                        vector<vector<int> >& C, int n) {
    for(int i = 0; i < n; i++) {
        for(int k = 0; k < n; k++) {
             int s = 0;
            for(int j= 0; j < n; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
void multiply_matrices3(const vector<vector<int> >& A,
                        const vector<vector<int> >& B,
                        vector<vector<int> >& C, int n) {
    for(int j = 0; j < n; j++) {
        for(int i = 0; i < n; i++) {
             int s = 0;
            for(int k = 0; k < n; k++) {
                s += A[i][k] * B[k][j];
            }
            C[i][j] = s;
        }
    }
}
void multiply_matrices4(const vector<vector<int> >& A,
                        const vector<vector<int> >& B,
                        vector<vector<int> >& C, int n) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < n; k++) {
            for(int i = 0; i < n; i++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
void multiply_matrices5(const vector<vector<int> >& A,
                        const vector<vector<int> >& B,
                        vector<vector<int> >& C, int n) {
    for(int k = 0; k < n; k++) {
```

```
       for(int i = 0; i < n; i++) {
           for(int j = 0; j < n; j++) {
               C[i][j] += A[i][k] * B[k][j];
           }
       }
   }
}
void multiply_matrices6(const vector<vector<int> >& A,
                     const vector<vector<int> >& B,
                     vector<vector<int> >& C, int n) {
    for(int k = 0; k < n; k++) {
       for(int j = 0; j < n; j++) {
           for(int i = 0; i < n; i++) {
               C[i][j] += A[i][k] * B[k][j];
           }
       }
   }
}
vector<vector<int> > generate_matrix(int n){
   vector<vector<int > > matrix(n, vector<int>(n));
   random_device rd;
   mt19937 gen(rd());
   uniform_int_distribution<int> dis(1, 10);
   for(int i = 0; i < n; i++) {
       for(int j = 0; j < n; j++) {
           matrix[i][j] = dis(gen);
       }
   }
   return matrix;
}
double measureTime(int n, void (*matrixMultFunc)(const vector<vector<int> >&, const
vector<vector<int> >&, vector<vector<int> >&, int)){

   auto A = generate_matrix(n);
   auto B = generate_matrix(n);
   auto C = vector<vector<int> >(n, vector<int>(n, 0));

   auto start = chrono::high_resolution_clock::now();
   matrixMultFunc(A, B, C, n);
   auto end = chrono::high_resolution_clock::now();
   return chrono::duration<double>(end - start).count();
}

int main() {
   int nn =  800;
```

```
    for (int n = 50; n <= nn; n += 50) {
        double timeIJK = measureTime(n, multiply_matrices1);
        double timeIKJ = measureTime(n, multiply_matrices2);
        double timeJIK = measureTime(n, multiply_matrices3);
        double timeJKI = measureTime(n, multiply_matrices4);
        double timeKIJ = measureTime(n, multiply_matrices5);
        double timeKJI = measureTime(n, multiply_matrices6);

        cout << "Size: " << n
             << ", IJK: " << fixed << setprecision(6) << timeIJK
             << ", IKJ: " << timeIKJ
             << ", JIK: " << timeJIK
             << ", JKI: " << timeJKI
             << ", KIJ: " << timeKIJ
             << ", KJI: " << timeKJI << endl;
    }
    return 0;
}
```