ECS 230 - Homework 03

- **Student:** Qirun Dai (qrdai@ucdavis.edu)

1. Problem 1 (`matmul1.c`)

1.1 Implementation Details:

a. For storage of matrix, I use "**column major ordering**" which groups matrix columns in contiguous memory space. The code is as follows:

```c
double *a = malloc(n*n*sizeof(double));
double *b = malloc(n*n*sizeof(double));
double * restrict c = calloc(n*n, sizeof(double));
```

b. For loop ordering in matrix calculation, I use "**dot product version**" which uses the formula $C_{ij} = \sum_k A_{ik} B_{kj}$:

```c
// dot product loop ordering for matrix multiplication
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
          c[i + n*j] += a[i + n*k] * b[k + n*j];
```

c. To calculate floating point performance **in GFlops**, I use the following formula:

$$\text{fl\_performance} = \frac{2 \times n^3}{10^9 \times \text{t\_elapsed}}$$

d. To measure elapsed time accurately, I repeat runs **3 times for each different n**.

1.2 Experiment Results

a. The average elapsed time and floating-point performance over 3 runs are as follows (using CSIF pc25 machine, which uses Core i7-10700 2.9 GHz):

n	t_elapsed (s)	Floating-point performance (GFlops)
100	0.002942	0.679737
200	0.024482	0.653573
500	0.411765	0.607146
1000	3.755919	0.532494
2000	47.435919	0.337298

1.3 Comparison and Analysis

a. **Phenomenon 1:** Compared with the peak performance of Core i7-10700 CPU I experiment with: $\frac{371.2}{8} = 46.4\ GFlops/s$, we can find the floating-point performance of my self-implemented matrix multiplication program extremley poor.

b. **Phenominon 2:** Also, as the matrix order `n` increases, the averaged floating-point performance consistently degrades.

c. **Analysis:**

   i. For phenomenon 1, it's probably because my self-implemented program only uses simple loops for matrix calculation **without any parallelization or compiler optimization**. Thus, the program is far from utilizing the full floating-point calculation capability of the CPU.

   ii. For phenomenon 2, the problem lies in **memory access**. Since I use the dot product version of loop ordering, as `k` changes in the inner loop `c[i + n*j] += a[i + n*k] * b[k + n*j]`, the program has to access `a[i + n*k]` which is not contiguous in memory. As `n` scales, the distance between `a[i + n*k]` and `a[i + (n+1)*k]` becomes greater, thus degrading the memory spatial locality of this program and leading to an increased amount of cost in memory access. This accounts for the monotonous decrease in average floating-point performance as `n` scales.

2. Problem 2 (`matmul2.c`)

2.1 Implementation Details:

a. The naming and definition of the 6 loop orderings are demonstrated as follows:

b. **Dot product 1**:

```
// loop ordering: dot product 1
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

c. **Dot product 2**:

```
// loop ordering: dot product 2
for (int j = 0; j < n; j++)
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

d. **Outer product 1**:

```
// loop ordering: outer product 1
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

e. **Outer product 2**:

```
// loop ordering: outer product 2
for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
        for (int i = 0; i < n; i++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

f. **Gaxpy column**:

   i. Column-based gaxpy, using the formula: $A\mathbf{x}$

```
// loop ordering: gaxpy column
for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

g. **Gaxpy row**:

i. Row-based gaxpy, using the formula: $\mathbf{x}^\top A$

```
// loop ordering: gaxpy row
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            c[i + n*j] += a[i + n*k] * b[k + n*j];
```

2.2 Experiment Results

1. The average floating-point performance in **GFlops/s** over 3 repeated runs, spanning across **6 loop orderings** and **3 optimization options** (using CSIF pc25 machine, which uses Core i7-10700 2.9 GHz). The best performance in each row is highlighted.

n	dot_prod_1	dot_prod_2	out_prod_1	out_prod_2	gax_column	gax_row
			-O0			
100	0.685502	0.687047	0.697172	0.706707	**0.707483**	0.692189
200	0.677387	0.647352	0.569849	0.707768	**0.712875**	0.572542
500	0.607276	0.555314	0.487962	**0.717742**	0.717218	0.488902
1000	0.533269	0.505180	0.310281	**0.716512**	0.714937	0.314474
2000	0.347239	0.315122	0.234458	0.708851	**0.709255**	0.234044
			-O2			
100	2.874969	2.905224	3.378868	4.010341	**4.076216**	3.775820
200	2.294186	2.488471	3.148722	4.204007	**4.315316**	3.161085
500	2.338932	2.361099	2.059369	4.366213	**4.368189**	2.072893
1000	1.771101	1.734385	0.341457	4.290187	**4.315883**	0.340937
2000	0.637537	0.612553	0.247246	3.447856	**3.620745**	0.246579
			-O3			
100	2.197500	2.796445	3.592353	7.944411	**11.430256**	5.191866

200	2.449481	2.497799	3.144306	7.564979	**11.412451**	4.521857
500	2.330633	2.361042	2.054783	7.670754	**10.886994**	3.339613
1000	2.002434	1.997303	0.348384	7.700024	**11.281760**	0.672144
2000	0.642511	0.617008	0.249252	4.713545	**6.805452**	0.485487

2.3 Comparison and Analysis

1. **`dot product 1` and `dot product 2`**
    - As can be seen in the results, for all the three optimization options, the performances of `dot product 1` and `dot product 2` are close in value. This is because their inner-most loops are exactly the same:

    ```
    for(int k = 0; k < n; k++)
        c[i + n*j] += a[i + n*k] * b[k + n*j];
    ```

    - Therefore, they have similar memory spatial locality and share similar memory access cost.
2. **`outer product 1` and `outer product 2`**
    - For all the three optimization options, it can be seen that `outer product 2` consistently outperforms `outer product 1` by a great margin. Let's analyze their memory access in the inner-most loop. For `outer product 1`, the inner-most loop is:

    ```
    for (int j = 0; j < n; j++)
        c[i + n*j] += a[i + n*k] * b[k + n*j];
    ```

    - We can find that both variables `c[i + n*j]` and `b[k + n*j]` are indexed and accessed in a noncontiguous way, because when j increases by 1, both indices have to increse by n. The remaining term `a[i + n*k]` remains unchanged throughout this loop. On the contrary, the inner-most loop for `outer product 2` is:

    ```
    for (int i = 0; i < n; i++)
        c[i + n*j] += a[i + n*k] * b[k + n*j];
    ```

- We can find that the term `b[k + n*j]` is unchanged. But the other two terms `c[i + n*j]` and `a[i + n*k]` are both indexed and accessed in a contiguous way, with i increasing by 1 and both indices also increasing by 1. This property greatly improves the memory spatial and temporal locality of `outer product 2`, making it possible to load one memory chunk into cache and then only need to access that chunk to complete all the computation in the inner-most loop. Therefore, `outer product 2` possesses a significantly better memory locality than `outer product 1`, and naturally has significantly better floating-point performance.

3. **`gaxpy column` and `gaxpy row`**
    - It can also be found that for all optimization options, `gaxpy column` has consistently and significantly better performance than `gaxpy row`. The reason behind this is also similar to the above. Let's review the inner-most loop of these two orderings again:

```
// loop ordering: gaxpy column
for (int i = 0; i < n; i++)
    c[i + n*j] += a[i + n*k] * b[k + n*j];

// loop ordering: gaxpy row
for (int j = 0; j < n; j++)
    c[i + n*j] += a[i + n*k] * b[k + n*j];
```

    - Similarly, the memory access of `gaxpy column` is contiguous, while that of `gaxpy row` is noncontiguous. This also accounts for the significant performance margin of `gaxpy column` over `gaxpy row`.

4. **`dot product` compared with `outer product` and `gaxpy`**
    - Another observation is that the two orderings of `dot product` are consistently outperformed by the orderings of `outer product 2` and `gaxpy column`. The reason behind this also lies in the efficiency of memory access. For the two orderings of `dot product`, their inner-most loops both have a term with contiguous index and another term with noncontiguous index. But for `outer product 2` and `gaxpy column`, their inner-most loops only involve either unchanged term or contiguously-accessed terms. This makes their floating-point performance consistently higher than that of `dot product`.

5. **The effect of optimization options**
    - Generally, as optimization level increases, the performances of all the 6 orderings show a tendency of corresponding increase. However, there are two things worth noting:

i. For `dot product 1`, `dot product 2` and `outer product 1`, the performance increase from `-O2` to `-O3` is obviously smaller than from `-O0` to `-O2`.

ii. For `outer product 2`, `gaxpy column` and `gaxpy row`, there remains consistent and significant performance growth from `-O0` to `-O2` and `-O2` to `-O3`. Notably, `gaxpy column` gains an exceptionally significant performance gain from `-O2` to `-O3`, compare with all other orderings.

- The above phenomenon might be explained as follows: `dot product 1`, `dot product 2` and `outer product 1` cannot gain further performance gain from compiler optimization, due to their inherently poor memory locality that is hard to optimize further. But `outer product 2`, `gaxpy column` have inherently much better memory locality, leaving more space for further efficiency optimization in memory access. Especially, `gaxpy column` and `gaxpy row` are optimized further from `-O2` to `-O3`, probably due to the good calculation property of the `gaxpy` operation and relevant optimization algorithms that are additionally implemented in the `gcc` compiler for these 2 special operations.

3. Problem 3 (`matmul3.c`)

3.1 Implementation Details

1. The function signature of `dgemm_` in C is as follows:

```c
void dgemm_(const char *TRANSA, const char *TRANSB, const int *M,
            const int *N,const int *K, const double *ALPHA,
            const double *A, const int *LDA, const double *B,
            const int *LDB, const double *BETA, double *C,
            const int *LDC);
```

Each of these parameters has a specific purpose:

- `TRANSA` and `TRANSB` determine whether each matrix needs to be transposed.
- `M`, `N`, and `K` describe the dimensions of the matrices involved.
- `ALPHA` and `BETA` are scalars used in the multiplication.
- `A`, `B`, and `C` are the matrices involved.

- `LDA`, `LDB`, and `LDC` are the leading dimensions of these matrices, which are important for memory layout, especially when the matrices are not square or when they are part of larger matrices.

3.2 Experiment Results

1. The average elapsed time and floating-point performance over 3 runs are as follows (using CSIF pc25 machine, which uses Core i7-10700 2.9 GHz):

n	t_elapsed (s)	Floating-point performance (GFlops)
100	0.000054	41.416623
200	0.000313	52.048430
500	0.004078	61.323871
1000	0.030976	64.577923
2000	0.242477	65.983201

3.3 Comparison and Analysis

1. **Extremely higher floating-point performance than self-implemented routines**
   - This can be explained by the numerous optimization practices that are implemented in the `BLAS` library [**referenced from here**]. First, `BLAS` library uses **parallel algorithms** and **low-level optimizations** that exploit the specific architecture of the CPU hardware. Second, `dgemm` and similar library functions are designed to make efficient use of the CPU cache, which is crucial for floating-point performance in matrix operations. What's more, BLAS libraries use vectorized operations to perform multiple calculations simultaneously using vector processors within the CPU. This is far more efficient than scalar operations performed in a standard loop-based approach.
2. **How can it exceed the expected peak performance?**

- As was calculated above, the expected peak floating-point performance of Core i7-10700 CPU is 46.4 GFlops/s. But now using `dgemm_`, the average floating-point performance can reach 65 GFlops/s, exceeding the expected peak value by nearly 50%. Such a significant improvement is actually attributed to the `Turbo Boost` feature of Intel processors [**referenced from here**]. Intel's Turbo Boost is a technology that allows a processor to dynamically increase its clock rate above the base frequency when it's operating below specified power limits. This feature is particularly useful when not all cores of the CPU are fully utilized. In our case, only a single core is heavily utilized because `OMP_NUM_THREADS=1` is exported. Therefore the rest of the CPU cores are not heavily loaded, potentially allowing the active core to run at a higher clock rate under the mechanism of Turbo Boost.