OOP Homework Assignment № 3

Frank (Shuxin) Chen, Tsinghua University 08/24/2016

Problem Statement

This problem requires us to use different ways to loop through a 3-dimension array. For example, whatever order we use i, j and k to loop through the 1st, 2nd and 3rd dimension, the equation should always be

$$c[i][j][k] = a[i][j][k] + b[i][j][k]$$

a and b are given arrays while c stores the results. Since the ways to loop through the array is different, the execution time for the operations my be also different. We should write a program to calculate the execution time for different sizes of the array.

When obtaining the execution time results, the next step we should approach is to analyse the reasons for the difference between the results.

Implementation Method

We define N as the size of the array, and we will test the program with $N = 2^k(32, 64, 128, \cdots)$. Since the N may be very large, we should allocate the memory for the array dynamically. `int a[N][N][N]` is **NOT** preferred.

Listing 1: Allocation of array memory.

```
void initializeArray(int***& a, int N)
{
    a = new int**[N];
    for (int i = 0; i < N; ++i)
    {
        a[i] = new int*[N];
        for (int j = 0; j < N; ++j)
        {
            a[i][j] = new int[N];
        }
    }
}
```

After the allocation, we initialize the arrays with random values from 0 to 999, and then we can loop through the arrays, calculate the results and store them in array c.

Inevitably, when N is small, the execution time is close to 0.001 second, which will not tell the difference between different ways of implementing. So we define M as the repetition times(we run the piece of code for M times). It equals to enlarging the execution time M times, making it easier to tell the difference. We can also get average execution time by division.

Since we need to adjust N and M frequently for best results, it is preferred to use the input N and M but not adjust them in the prgram. We can use the arguments `argc` and `argv[]` in the `main()` class and give the arguments in the command line. The `main()` class codes are shown below.

Listing 2: Codes for main class.

```
int main(int argc, const char* argv[])
{
    srand(static_cast<unsigned int>(time(0)));
    if (argc != 3)
    {
        std::cout << "Arguments Error!" << std::endl;
        return 0;
    }
    int N = atoi(argv[1]);
    int M = atoi(argv[2]);
    checkTime(N, M);
    return 0;
}
```

```
$ g++ -o main main.cpp -std=c++11
$ ./main 128 2000
```

So we can just type the g++ commands in the Terminal(for MACOSX) and the program will output the results.

Implementation Results

The results for different N, M and ways are shown in the two tabulations below. Way 0, 1, 2, 3, 4, 5 stands for ijk, ikj, jik, jki, kij, kji.

N	M	$Way0$	$Way1$	$Way2$	$Way3$	$Way4$	$Way5$
32	50000	$9.028s$	$9.739s$	$9.211s$	$11.359s$	$13.812s$	$18.408s$
64	10000	$13.652s$	$14.356s$	$14.377s$	$20.769s$	$31.539s$	$30.420s$
128	2000	$23.058s$	$25.055s$	$30.135s$	$71.847s$	$66.705s$	$93.411s$
256	200	$18.472s$	$44.282s$	$22.747s$	$324.953s$	$94.322s$	$702.644s$
512	25	$18.013s$	$47.307s$	$18.565s$	$549.359s$	$152.060s$	$1062.936s$
1024	3	$16.938s$	$77.604s$	$17.598s$	$1243.117s$	$208.806s$	$1625.269s$

Table 1: Execution time.

We can also get the average execution time.

N	M	$Way0$	$Way1$	$Way2$	$Way3$	$Way4$	$Way5$
32	50000	$0.000s$	$0.000s$	$0.000s$	$0.000s$	$0.000s$	$0.000s$
64	10000	$0.001s$	$0.001s$	$0.001s$	$0.002s$	$0.003s$	$0.003s$
128	2000	$0.012s$	$0.013s$	$0.015s$	$0.036s$	$0.033s$	$0.047s$
256	200	$0.092s$	$0.221s$	$0.114s$	$1.625s$	$0.472s$	$3.513s$
512	25	$0.721s$	$1.892s$	$0.743s$	$21.974s$	$6.082s$	$42.517s$
1024	3	$5.646s$	$25.868s$	$5.866s$	$414.372s$	$69.602s$	$541.756s$

Table 2: Average execition time.

By comparing the values in a tabulation, we can find that as the N increases 2 times, the execution time of $Way0$ and $Way2$ increases 8 times(it is trivial since the time complexity is $O(n^3)$), but the other ways increase more than 8 times, it denotes that we cannot ignore the time complexity of memory addressing. The reasons will be demonstrated in the next section.

Data Analysis

The first fact we should realize is that, if we allocate memory for a simple array `int* a` of size N, the memory addresses of the N integers are consecutive. If we loop through the whole array, we can just start with the head address of the array and move to the memory unit next to it and so on.

However, there is no definition of 2-dimension array in C++. The 2-dimension array is actually an array with N pointers where each pointer points to an array with N integers. So the `a[0][N - 1]` may or may not next to the `a[1][0]`. It depends on the situation of the memory before the array is allocated.

The second important fact is the mechanism of **CACHE**. When some data is required, the computer will find the data in cache before in memory, for the reading speed of cache is faster than memory. After the data is found, the computer will put the data **block** into cache because the data in the data block is more likely to be required.

Now we can come back to the problem. For the cases with large N, if the loop order is ijk or jik, we need to read $a[][][k+1]$ after $a[][][k]$. Since they are next to each other, they are more likely to be in the same memory block, which means we can find the data in quick cache. For the loop ord jki or kji, there is quite a large distance between two i's, so the are less likely to be in the same memory block, which means we should scan the slow memory.

For the cases with small N, the data are stored very close to each other, so the execution time is very similar. All in all, we can have the in-equation that

$$ijk \approx jik \ll ikj \approx kij \ll jki \approx kji$$

In the actual data, ikj is less about one-third of kij, because the second loop(address for $\Theta(n^2)$ times) should not be ignored.

Implementation Parameters

```
Operation System: OS X El Capitan 10.11.6
CPU: 2.5 GHz Intel Core i7
Memory: 16 GB 1600 MHz DDR3
```