Max Williams

Assignment 4

CSE 611
October 1, 2020

Loop Performance

Introduction

Choosing the right loop order (or, equivalently, indexing strategy) affects performance greatly. The cache is able to read and store contiguous chunks of memory, and follows strategies that maximize performance for contiguous reads. In this paper, I evaluate the 6 different iteration orders for naive matrix multiplication. In Table 1, the 6 different orders and their respective runtime for multiplying two 1000×1000 matrices is listed for two compiler optimization levels.

order	time (seconds) -O0	time (seconds) -O3
i-j-k	4.977	1.623
i-k-j	21.557	21.919
k-i-j	21.501	21.821
j-i-k	7.025	2.108
k-j-i	3.948	0.563
j-k-i	3.907	0.368

Table 1: Timings for all six orderings, for two optimization settings: -O0 and -O3.

Runtime Difficulty

The original problem statement called for 5000×5000 matrices, but the program wouldn't terminate in a reasonable amount of time. This may have been due to the large size overflowing the largest cache. The program has $3 \times 5000 \times 5000 = 75 \times 10^6$ floats or 300 million bytes of memory, which is well below my 8 GB of RAM but still quite large compare to the cache size. Regardless of the cause, the array size needed to be reduced to 1000×1000.

Best Performance

The j−k−i ordering performed best, and k−j−i performed almost identically. The thing to notice here is that both have i as the innermost loop variable. Looking at the array access:

C[i+j∗N] += A[i+k∗N] ∗ B[k+j∗N];

I see that i only appears as a base index (not multiplied by anything). This means that within the innermost loop, j and k are constant and i is used to access contiguous memory.

Worst Performance

The orders i−k−j and k−i−j both have the worst performance. This is because they both have j in their innermost loop, which is multiplied by N in the index for both B and C. This causes the largest number of cache misses, because single floats are being accessed from memory locations spaced out by 1000 floats. By the time it gets to the next iteration and j is back at 0, those locations have already been taken out of cache and cause a cache miss.

Middlest Performance

The order i−j−k and j−i−k both have middling performance. This is mostly due to k being in the innermost loop, which only causes one spaced-out read in the innermost loop.

Optimization Level

For the best performing loops, compiler optimization makes a big improvement, but for the worst performing loops, it actually makes them slightly worse.

Conclusion

Iteration order optimization can make a huge difference in performance - nearly sixty times in the most severe case. This means that when developing performance critical sections of code, careful consideration of iteration order should be made or experimental testing should be done. Based on these results, I conclude that the following rule is useful when deciding iteration order: "The innermost loop is most important, and having the innermost loop variable multiplied by anything will harm performance."

Of course, premature optimization should be avoided - profile before optimizing, and only optimize if it makes a difference for your product. Doing this kind of optimization can lead to code which is harder to read then putting your loops in their logical, intuitive order.

Code Listing

```
#define N 1000
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

// `matmul_macro` allows for the loop variables i, j, k to be
// put in any order with a single macro call
#define matmul_macro(name, var1, var2, var3)\
void (name)(float* A, float* B, float* C) {\
    for (int var1 = 0; var1 < N; var1++) {\
        for (int var2 = 0; var2 < N; var2++) {\
            for (int var3 = 0; var3 < N; var3++) {\
                C[i+j*N] += A[i+k*N] * B[k+j*N]; \
            }\
        }\
    }\
}

// This defines all six functions using the above macro
matmul_macro(matmul1, i, j, k);
matmul_macro(matmul2, i, k, j);
matmul_macro(matmul3, k, i, j);
matmul_macro(matmul4, j, i, k);
matmul_macro(matmul5, k, j, i);
matmul_macro(matmul6, j, k, i);

// TIME takes a function name and calls it on A, B, C and times how long it t
// printing the result
#define TIME(name)\
t = clock();\
(name)(A, B, C);\
printf("The output is %f\n", C[0]);\
t = clock() - t;\
printf (#name " took me %lu clicks (%.3f seconds).\n", t, ((float)t)/(CLOCKS_


// Sets the elements of an array to random numbers 0.00 to .99 by increments
float* initialize(float* arr) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            arr[i+j*N] = (rand() % 100) / 100.0;
        }
    }
    return arr;
}

int main() {
    float* A = initialize(malloc(sizeof(float) * N * N));
    float* B = initialize(malloc(sizeof(float) * N * N));
    float* C = malloc(sizeof(float) * N * N);
    printf ("Done allocating\n");
    srand(time(NULL));
    clock_t t;
    TIME(matmul1);
    TIME(matmul2);
    TIME(matmul3);
    TIME(matmul4);
    TIME(matmul5);
    TIME(matmul6);
    return 0;
}
```