Aerial Robotics Kharagpur Documentation

Ankit Ranjan

Abstract: This documentation presents in detail the learning and research done while completing the ARK cpp code optimisation task.

The task was to optimize the program to allow it to run on matrices of sizes up to 1000*1000. The time complexity of the functions FindMinCostA and FindMaxCostB was of the order 2^n.

Some techniques like memorization for recursion was used and I tried to optimise the matrix multiplication in filling ProductMat using cache optimisation.

These two methods gave a significant improvement in runtime and it was highly efficient for matrices of order up to 1000 * 1000.

I. INTRODUCTION

I had to optimize the given cpp code. The code involved two functions to be optimized, FindMinCostA() and FindMaxCostB(). The initial time complexity of the functions was of the order 2^n, where n*n was the order of the matrix to be used in the function.

I first implemented the dynamic programming technique with memorization, which is an algorithm to solve a problem recursively, breaking it into simple subproblems and following the bottom-up approach. I couldn't continue with it, even though it reduced the time complexity to $O(n^2)$. As to prevent space complexity, I used one memorization matrix for the entire program and calling the functions once filled the matrix with values that couldn't be utilised for the next calling, and I needed to initialize the matrix all over again.

To omit that I implemented memorization in a top-down fashion and that allowed me to call the function once and fill the memorization matrix with corresponding cost values.

To optimize multiplication further, I used a technique called thrashing, which is a cache-friendly method for matrix multiplication. Instead of multiplying matrices in a row-major fashion, in thrashing, we use a column-major fashion, which increases computation speed as accessing data from cache is much faster than accessing data from RAM.

II. PROBLEM STATEMENT

As part of this task, I was allotted a cpp code file, which I had to optimise. The program defined two functions FindMinCostA() and FindMaxCostB().

The function FindMinCostA() gives the minimum possible cost of the path to reach the index (n-1,n-1) from any given index (i,j) in the matrix costMatrixA of size n*n, while the function

FindMaxCostB gives the maximum cost for the same in the costMatrixB of the size n*n.

The initial time complexity of both the functions was given by:

$$T = O(2^n)$$

where n is the size of the matrix on which I run the functions.

For a matrix of size 1000, T is of the order $O(2^{1000})$ which is tremendously large. So the program can't work for such a large n.

After that, I had to fill the matrix ProductMat, using matrix multiplication of the values of the costs returned by the functions FindMinCostA() and FindMaxCostB().

In the last part of the task, I had to optimize a filter application on the matrix ProductMat.The dimension of the filter given to me was `4` x `n`.

III. RELATED WORK

I tried to learn about different methods of optimization using online resources like StackOverflow, youtube, and some other platforms.

In the process of completing this task, I learnt and tried dynamic programming, memorization algorithm for recursion.

I also tried to learn parallel programming.

IV. INITIAL ATTEMPTS

Initially, I tried to use dynamic programming to optimize the program using a bottom-up recursion approach. I implemented dynamic programming using memorization.

But for already filled values of the arrays, the approach didn't give the output as required which I found by dry running the code. Hence I changed my approach.

V. FINAL APPROACH

In my initial attempt, there was one implementation issue. Whenever the functions, FindMinCostA() and/or FinaMaxCostB() was called, some indices of the memorization matrix were filled with values which were visited while travelling from (n-1,n-1) to that index, where n*n represents the size of the matrix.

I tried out dry run to find out the actual problem. The solution to that problem which I implemented was to follow a top-down approach, i.e. instead of dynamically programming the recursive method, I traversed tha array *from* the index of which I had to calculate the path, say (i,j) { 0 ≤ i , j < n }, *to* the final index, i.e. (n-1,n-1).

This allowed me to fill the entire memorisation table in a single function call. Therefore, I called the function once, after initialisation of the cost matrices CostMatrixA and CostMatrixB, in the *main()* function for the (0,0) index.

The calling of the functions FindMinCostA() and FindMaxCostB() filled their memorization table with values of the minimum and maximum cost of traversals, respectively, corresponding to every index.

Thus, now I had simply two matrices, i.e. memorization tables with the corresponding cost values in each cell of the matrix.

To return the value of cost required of any index, I could simply return the value of that index in the corresponding memorization matrix.

I figured out that ProductMat matrix can now be easily constructed by matrix multiplication of the two memorisation matrix, say memA and memB respectively.

I optimised the codes in the above stated manner, and now I didn't need to call both the functions for every iteration in ProductMat filling, rather I can directly use the values in memA and memB.

I tried a cache-friendly method for multiplication of memA and memB, which gave me significant improvement in runtime.

In Spite of these, since matrix multiplication has a time complexity of order $O(n^3)$, when the size of the matrix is large, it increases the overall runtime of the program.

The overall runtime is governed by the matrix multiplication to fill the matrix productMat, and the filter application on the matrix isn't consuming much time and resources, I preferred not to modify that section of code as the end result of modification won't be significantly observable.

VI. RESULTS AND OBSERVATIONS

Algorithm	Time taken(n= 500)	Time taken(n= 1000)
Without optimising	NA	NA
After optimising, without using cache-friendly method	2.1 s	16s
Using cache-friendly method	3.5s	11.43s

Using a cache-friendly method gives some improvement in runtime for large array size, but for smaller size of array, due to more calculation involved, it increases runtime.

VII. FUTURE WORK

I am currently exploring the domain of parallel programming and computing which will help me to further optimise the matrix multiplication and thus help reduce the runtime of the entire program.

The cache-friendly multiplication isn't a good technique for matrices of sizes around 500. So, I will also try to plug that loophole.

CONCLUSION

The task was to optimize a cpp program that contained functions and matrices. Initially the program couldn't run for a higher - sized array, but after I optimised it, there was a massive improvement in runtime. I tried and implemented some programming techniques and algorithms new to me, like dynamic programming, memorization etc.

Since to obtain efficiency in program and reduction in runtime is vital for the program to be deployed or installed on any autonomous robot system, hence the above problem and its solution would be helpful to ARK.

REFERENCES

[1] Geeks for Geeks
[2] Stack Overflow
[3] Springer