Project 4 - Fast Matrix Multiplication in C

Name 颜云翔 (Yunxiang Yan)

SID 1912525

CS205 C/C++ PROGRAMMING

2022 FALL

PROJECT4 - FAST MATRIX MULTIPLICATION IN C

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPT. OF COMPUTER SCIENCE AND ENGINEERING

Contents

1 Introduction

The link to the github repo for this project is: (https://github.com/RaccoonOnion/fast-matrix-mul.git) The fourth project is a specified extension of project 3 and its main focus lies on performance improving. As usual, I will first summarize the main challenges and the performance improving techniques that I adopt in this project.

1.1 Challenges

This project is intriguing and challenging because it not only requires a good grasp of high-performance coding techniques but also demands us to have a clear idea of computer architecture. Specifically, some of the main challenges I have during this project are:

1. Configuration of library OpenMP on MacOS (M1 chip). Unlike Windows OS user and MacOS with Intel chip user, the installation and linking process of OpenMP library are reasonably hard and lack enough online public information about it. This is the reason that I decided to write a separate section to explain the process of using OpenMP on Mac with M1 chip.

2. The correct combination of performance boosting techniques. While we learn a variety of performance boosting methods in class, only the right combination of them will give us the best result and simply stacking methods on one another will sometimes deteriorate the performance.

3. Smart usage of insights from computer architecture. At the end of the day, when we want to move forward after we reach a certain limit, we have to modify the algorithm itself leveraging the physical constraints of computer architecture. Among them, the caching mechanism is of great importance.

1.2 Performance Boosting Tricks

Below is a list of performance boosting tricks that I adopt:

1. An algorithm that creates continuous memory storage and manipulation. This is most important trick in this project and with the help of this trick my project beats openblas

library in cases when sizes of the matrices are smaller than 2000 * 2000. The idea of the algorithm is very simple: when we perform the matrix multiplication A * B, if we store both A and B as **1D array of each rows**, then we have to access the memory that stores B in a incontinuous way because we need to perform dot product of A's row and B's column. Thus, in this project when I do the matrix multiplication, I first calculate the **transpose** of matrix B so that multiplication will happen between the row of A and B-transpose. This will also facilitate the future process of SIMD.

2. Flattened for loops. For both the process of getting transpose and the process of calculating matrix multiplication, I flatten the nested for loops to create a single, flattened for loop. This action not only reduces the cost of using more stacks but also makes the compiler easier to optimize the code and facilitates parallel computing with OpenMP.

3. Smart usage of pointers to avoid memory copy. When computing the dot-product of each rows in A and B-transpose, I send correctly computed pointers and step size to the dot-product function instead of copies of the rows. This saves a lot of time for memory copying.

4. Avoid unnecessary computations. I minimize the access of struct members by creating local variables to store them when we need them the first time.

5. SIMD and parallel computing using OpenMP. With the help of OpenMP library and intrinsics set of NEON, I implemented a SIMD version of dotproduct calculation and make use of all CPU cores during the entire matrix multiplication process.

6. O3. Last but not least, I make use of the compiler to further optimize my code by adding an "-O3" compilation flag.

2 Experimental Performance

The comprehensive result table **Figure1** shows that the Improved method and Openblas method both have significant improvement in running time compared to the plain method, which uses nested loops to do the calculation. The red number indicates the best result among the three methods. We can see that except for the 16 * 16 case, Improved method beats Openblas. When the size of the matrices surpass 2000 * 2000, the Openblas method is

the best method.

Experimental Data (Unit: second)							
Sizes	16	128	1000	2000	4000	8000	16000
Openblas	0.000003	0.014291	0.079082	0.558474	2.666825	21.001253	204.905449
Improved	0.000424	0.000378	0.058576	0.521788	3.769493	34.305933	293.394000
Plain	0.000056	0.016708	3.515406	29.708994	409.913834	4999.973905	NA

Figure 1: Experimental Result Table

The reason behind this phenomenon is the difference in computation complexity. Because Openblas implements Strassen algorithm (https://en.wikipedia.org/wiki/Strassen_algorithm). Its complexity is $O(n^{2.807})$ while the improved method is around $O(n^3)$. Thus, when the size of the matrix gets larger and larger, the increase in time for Openblas is slower than our improved method. This can also be seen from **Figure1,2**, where the fitted equations are shown.

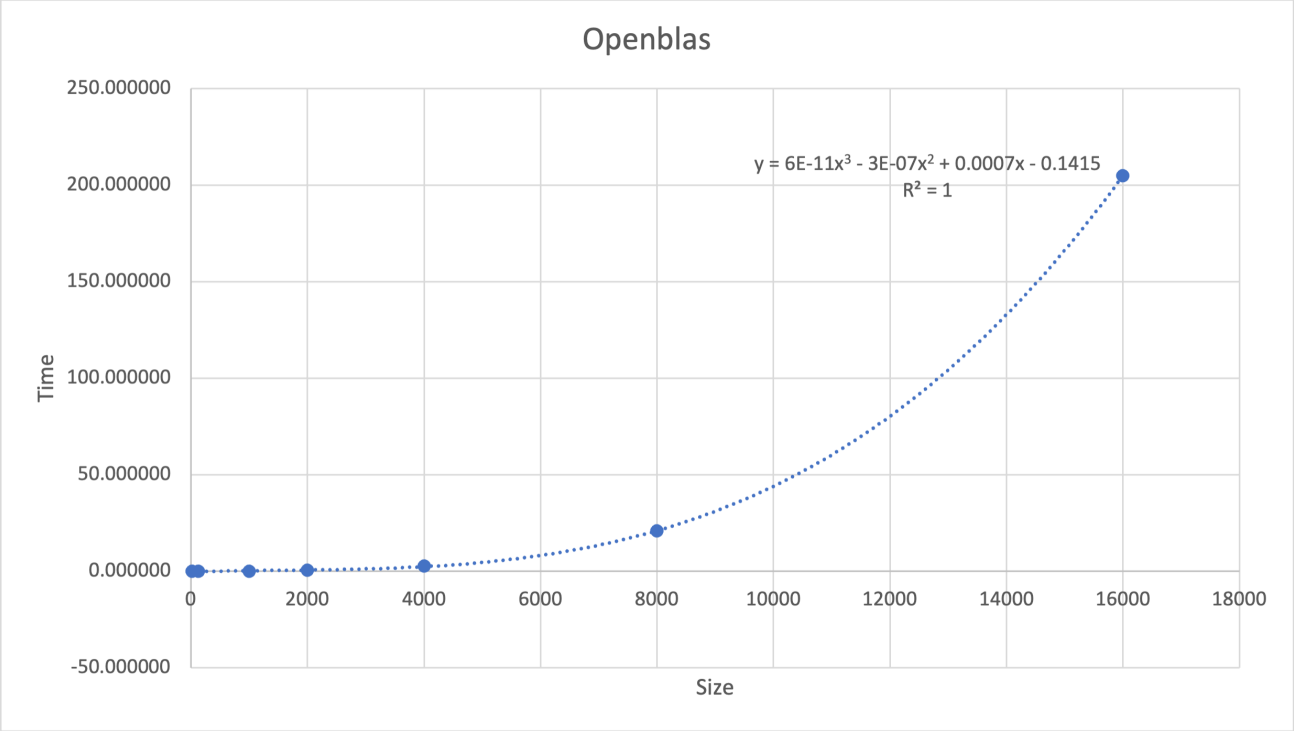

Figure 2: Openblas Result Scatter Plot

3 OpenMP for M1 Mac

Here I will briefly explain the process for M1 Mac user to use OpenMP.

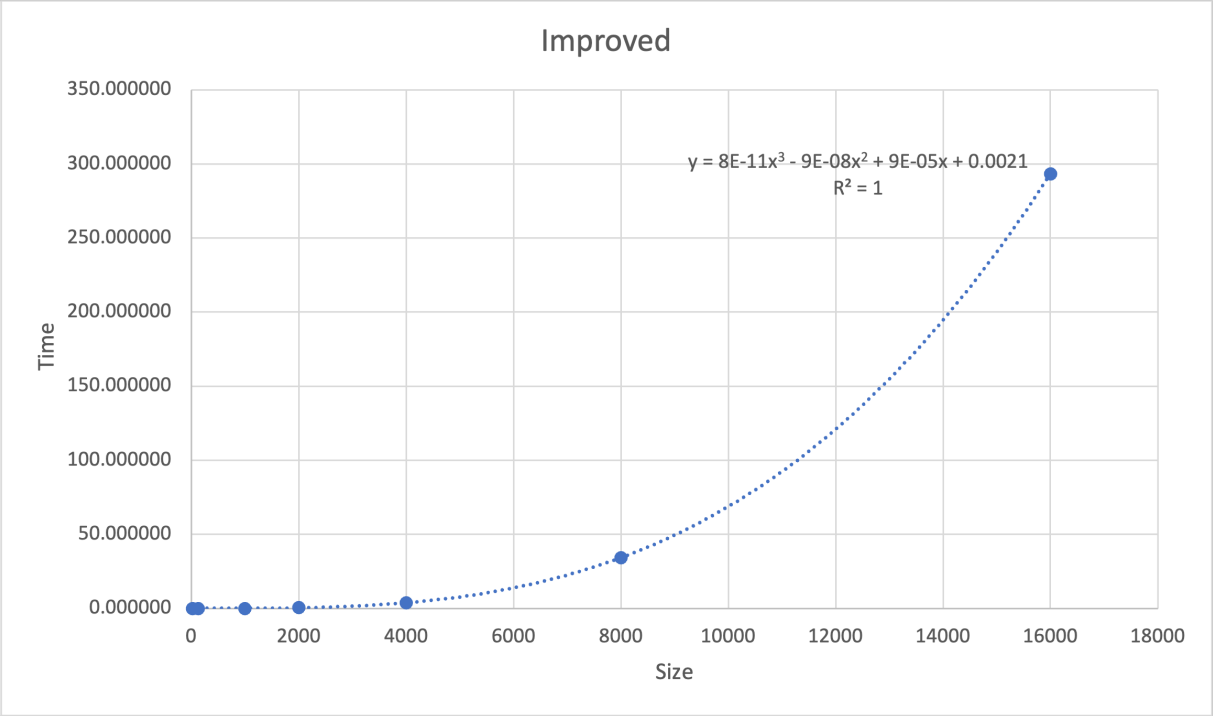

Figure 3: Improved Method Result Scatter Plot

There are two ways to install OpenMP. The first way to download OpenMP is through homebrew. Just type in terminal **brew install libomp**. Note that the OpenMP package brew installs in this way is keg-only, so we have to add two compilation flags when we want to use it. **Figure6**

Another much easier way is to just use GNU gcc compiler with **-fopenmp** flag during compilation. Notice that in order to invoke the GNU gcc compiler, we need to use **gcc-x ...** where x is the version of your installed GNU gcc compiler. This process is detailed in (https://stackoverflow.com/questions/58344183/how-can-i-install-openmp-on-my-new-macbook-pro-with-mac-os-catalina)

On my system, which has GNU gcc 12.2.0 the compilation command is: **gcc-12 -o test-improved -fopenmp -O3 test-improved.c matrix.c** .

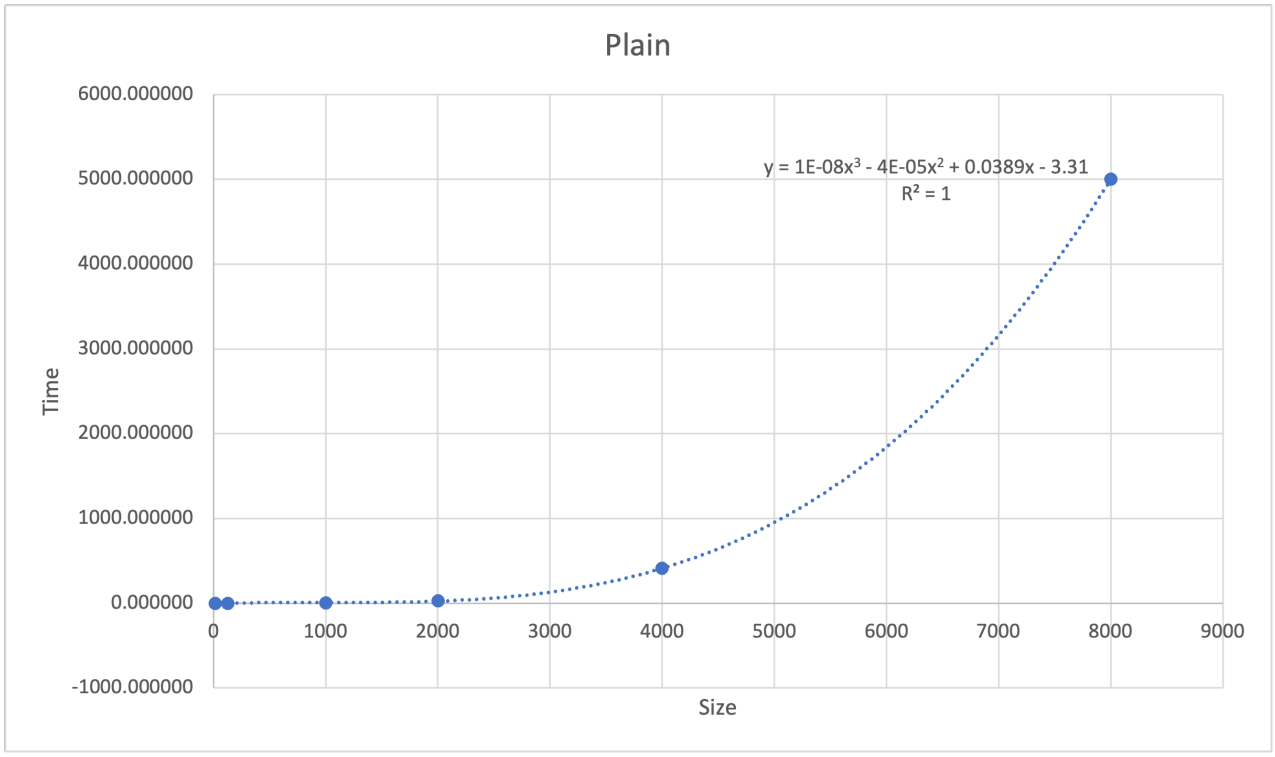

Figure 4: Plain Method Result Scatter Plot

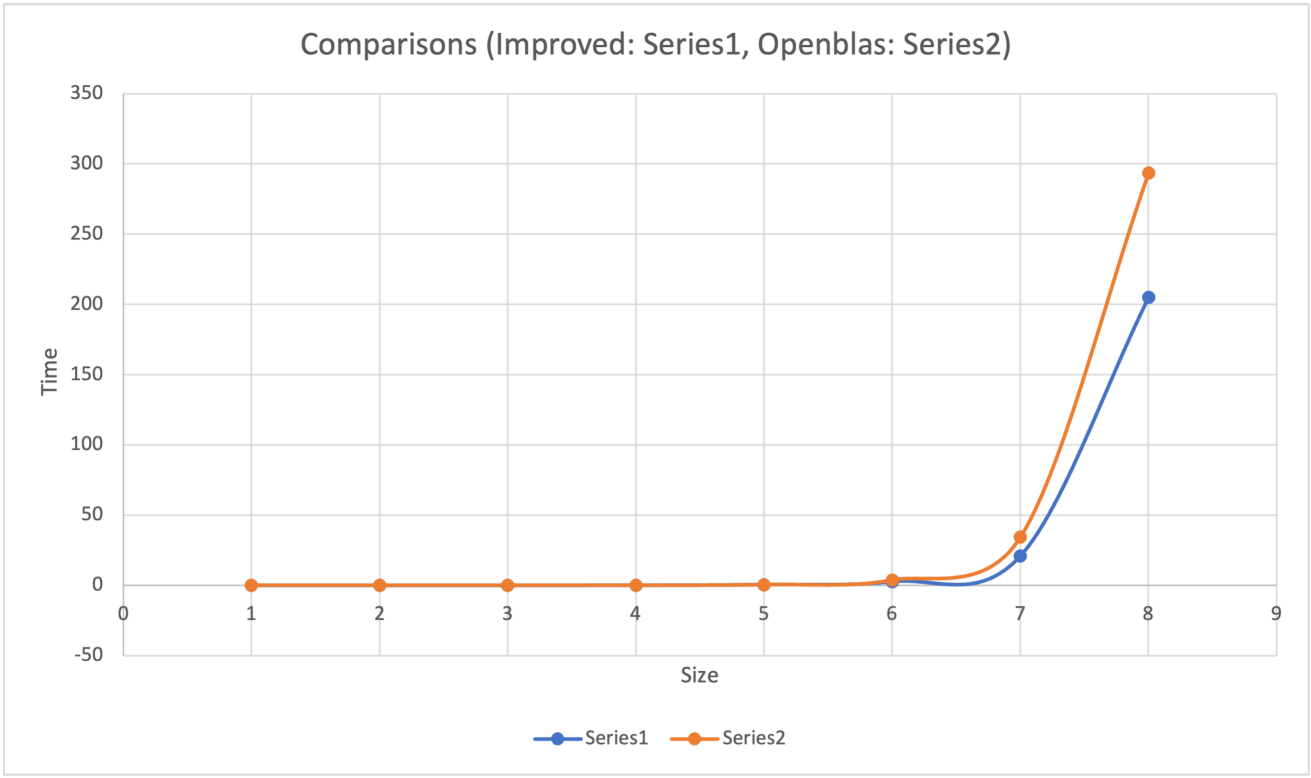

Figure 5: Comparison Between Improved Method and Openblas

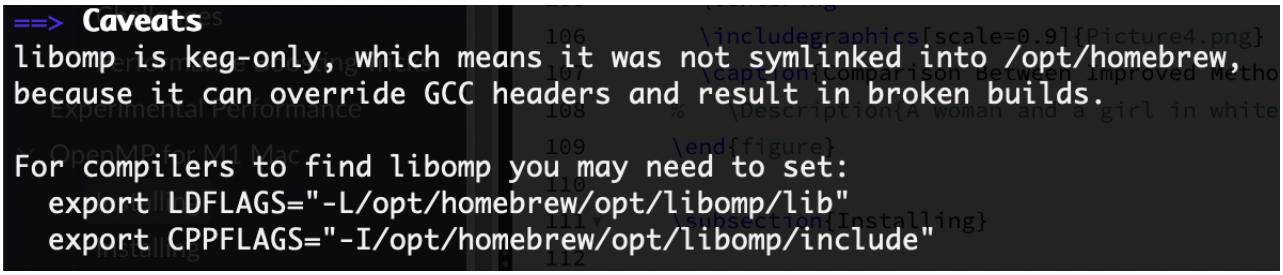

Figure 6: Prompts when Installing OpenMP through homebrew