Performance of Non-Optimized Matrix Multiplication Algorithms

Chihang Wang

18 November 2015

1 Introduction

This document is a brief report on the work done in the second fortnight period of my Part II project.
The following has been achieved:

- A primitive implementation of matrix multiplication, i.e. 3 level nested for loop
- Cache Oblivious Algorithm on matrix multiplication, according to this paper [1]
- Strassen's Algorithm, according to the Wikipedia entry [2]
- All the code is properly documented, and can be found in this Github entry
- a brief study on their relative performance is taken, (see next section)

2 Performance Comparison

2.1 Performance without compiler optimization

Fig.1 shows the performance of the 3 algorithms without any compiler optimization flags turned on. The datatype used is 32 bit integer

The result confirms that within these 3 implementations, Strassen algorithm has the optimal performance for larger values. It it worth noticing that for smaller matrices, the simple approach is faster.

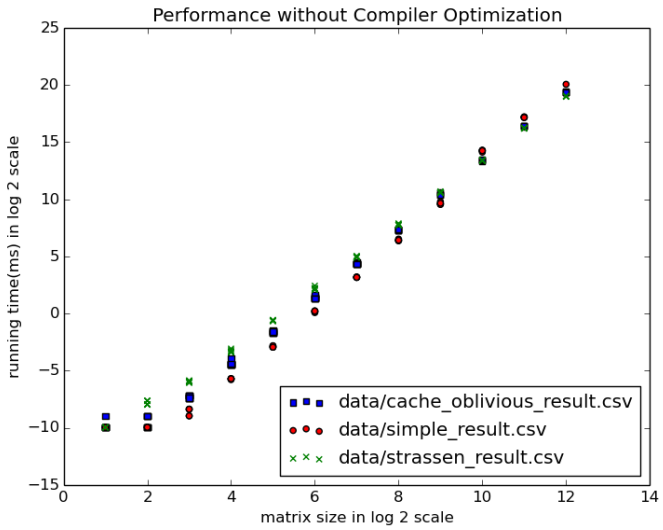

Figure 1: Running Time without compiler optimization

2.2 Performance with -O3 flag

Fig.2 shows the performance of the 3 algorithm with the -O3 optimization flag turned on. The datatype used is 32 bit integer. the series labelled "GEMM" is measured from the performance of ATLAS BLAS library. Note the datatype for GEMM is float. In limited trials of float on other algorithms, it is observed that there is no performance difference between float type and integer type.

There are two observations worth noticing

- BLAS GEMM is much much faster than the other algorithms. This shows the importance of cache performance
- Now with -O3 flag turned on, cache oblivious is actually faster than Strassen's algorithm.

2.3 Hybridize Strassen with BLAS GEMM

The fact that Strassen has worse performance after -O3 optimization suggest that cache performance may not be as good as we can expect. In order to explore this further, I increased the base case of Strassen from 2x2x2 to 256x256x256. In addition, GEMM is used for matrices of size lower than 256x256x256. This offers a hybridized approach. The performance summary is displayed in figure 3:

Not surprisingly, this time the hybrid approach has performance better than pure ATLAS GEMM approach for larger matrices. The data shown that doubling the dimension increases the running time by a factor of 8 in ATLAS

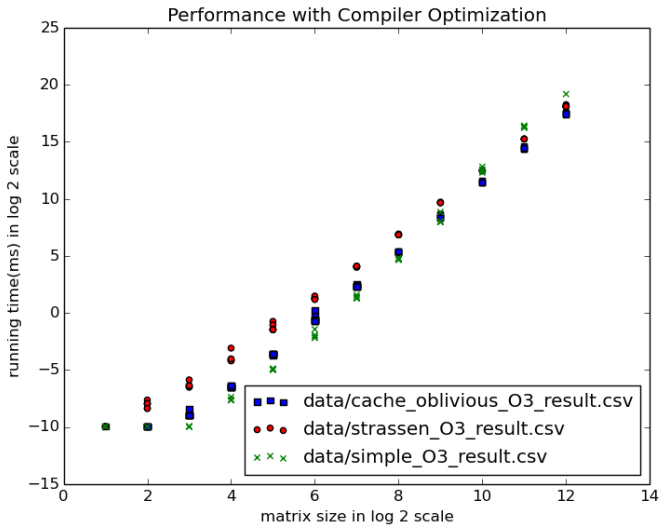

Figure 2: Running Time with compiler optimization in Log2 scale

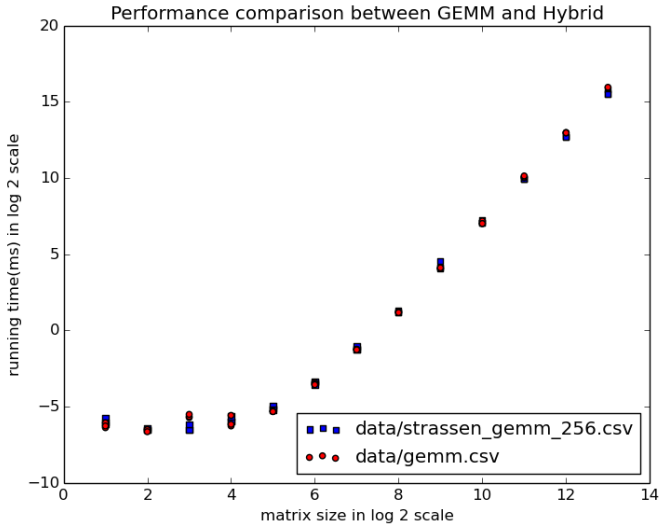

Figure 3: Running Time comparison between GEMM and Hybrid

GEMM approach, but a factor of 7 in Hybrid approach, which agrees with the theoretical complexity.

Dimension	GEMM	Hybrid
256	2.32	2.30
512	16.68	18
1024	129	132
2048	1079	997
4096	8053	6779
8192	63703	47536

Table 1: Running time (ms) comparison between GEMM and Hybrid

3 Notes on additional memory requirement

In the last report, I analyzed the time complexity of each algorithms. However, the space complexity was not being examined. In the implementation, I noticed that Strassen's algorithm requires additional workspace for temporary matrices.

3.1 Additional workspace for Cache Oblivious Algorithm

For the matrix multiplication:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Recall that the COA works as the follow:

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$
$$C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$$
$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$
$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}$$

If we are to perform the 8 sub matrix multiplication in sequential order, we will need temporary memory space to store the temporaries such as the result of $A_{1,1} \times B_{1,1}$

Forunately, this can be avoided by breaking the equation into:

$$C_{1,1} = A_{1,1} \times B_{1,1}$$
$$C_{1,1} += A_{1,2} \times B_{2,1}$$

This can be done if we have routine supporting $C += A \times B$
My implementation takes this approach and hence does not require additional memory
The problem of this approach is that the 8 multiplication may not be able to parallelize all together. Unless, the write can be done in sequential order, i.e. there is no over-write

3.2 Additional Workspace for Strassen's Algorithm

Recall that Strassen's Algorithm requires the construction of 7 temporary matices, as follow:

$$M_1 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2}) \times B_{1,1}$$

$$.....$$

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Also notice that for the construction of M_1, M_6, M_7, we need 2 more temporary matrices to hold the intermediate values from matrix addition/subtraction. These 9 matrices are simultaneously alive.
Suppose the space requirement for dimension n is $S(n)$, then we can derive the following recurrence relation:

$$S(n) = 7 \times S(\frac{n}{2}) + 9 \times (\frac{n}{2})^3$$

$$S(2) = 1$$

We can see that S(n) is equal to:

$$n^3 \times \frac{9}{8} \times \sum_{n=0}^{\log_2 n} (\frac{7}{8})^n$$

which equals to

$$n^3 \times 9 * (1 - (\frac{7}{8})^{\log_2 n}$$

for large n, this means the extra workspace can be as large as 9 times the usual requirement

3.3 Reducing Workspace by rearrange construction order of temporary matrices

It is observed that not all 9 temporaries are required in the entire course of the multiplication routine. For instance, M_7 is only required in the calculation of $C_{1,1}$, the memory can be freed as soon as possible.
According to my calculation, only 6 sub matrices are required to be alive at peak time. Hence, the factor is reduced from 9 to 6.
Also notice that, by assigning to the resultant matrices as soon as possible. i.e. perform $C_{1,2} = M_3 + M_5$ as soon as M_3 and M_5 are available, instead of

Dimension	GEMM	Hybrid	Order Rearrange
512	16.68	18	17
1024	129	132	128
2048	1079	997	977
4096	8053	6779	6340
8192	63703	47536	44310

Table 2: Running time (ms) comparison

waiting for all sub-matrices being computed, the performance improved by an observable amount, as shown in table 2.

This is because that the computed values can be used right away, instead of being flushed back to lower level cache, and brought up at a later time, which wastes memory bandwidth

4 Next Step

One of the problem encountered is that I cannot test performance for matrices which has dimension larger than 8192. Ubuntu crashes on dimension 16384. It will be valuable to see if the observed pattern continues for larger matrices
The implementation of Strassen only works for square matrices right now, by the next week, it should be able to work in any given dimension with reasonable performance.

References

[1] Harald Prokop Cache Oblivious Algorithm, 1999

[2] Strassen's Algorithm Wikipedia https://en.wikipedia.org/wiki/Strassen$_algorithm$