Summary on Matrix Multiplication Algorithms

Chihang Wang

Revised 9 Nov 2015

1 Introduction

A study on cache oblivious algorithm and Strassen's algorithm has taken place. This document summarizes the key concepts of these two algorithms.

2 Naive approach to Matrix multiplication

... The naive approach is a 3-level nested for loop [Pseudo code required here], hence the complexity is cubic. For large matrix size, and assume we use a row-major layout, since one of the matrix is read in column order, each read will cause a cache-miss. This implies the cache miss in worst case can be cubic as well!

3 Cache Oblivious Algorithm

A cache oblivious algorithm is a divide and conquer recursive approach. It iteratively divides the matrices concerned into smaller matrices. For the extreme case, the base case for the recursion is when the length of the matrix reaches 1. This approach is cache oblivious because no manual tuning is required to set what is the size of matrix at which point to stop the recursion and apply a general algorithm is solve. The divide approach can be visualized as the follow:

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

The formula to compute the output matrix is given as:

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

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

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

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

The number of multiplication and addition performed is still the same. For cache of size Z and cacheline of length L, the cache miss is reduced to:

$$\Theta(m + n + p + \frac{mn + np + mp}{L} + \frac{mnp}{L\sqrt{Z}})$$

In practice, we may not want to use extra memory to hold temporary matrices, e.g. the product of $A_{1,1} \times B_{1,1}$, therefore, we may want to write the computed value directly to $C_{1,1}$ This will mean that an extreme version of this cache oblivious algorithm will result a lot of update to matrix C. To cope with this problem, it may be desirable to stop the recursion and compute the small-enough matrices using naive algorithm.

4 Strassen's Algorithm

Evidently, the bottleneck for a matrix multiplication routine is the number of multiplications required. In both the naive algorithm or the cache oblivious algorithm, the number of multiplication is $O(n^3)$ where n is the size of the matrix. Strassen's algorithm redeuces the complexity to $O(n^2.7)$ in the following way

4.1 Basic idea

Strassen's algorithm computes 7 sub-matrices and then use them to compute the output matrix. The outline is as the follow:

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2}) \times B_{1,1} \\
M_3 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})
\end{aligned}$$

Afterwards, the four sub-submatrices can be computed as:

$$\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

For a 2x2x2 matrix multiplication, it requires 8 multiplication in the naive algorithm. However, one can see that only 7 multiplication is required in this case, hence reducing the overall complexity to:

$$O(n^{\log_2 7})$$

4.2 Complexity Analysis

Suppose the number of multiplications for a routine of size $n \times n \times n$ is $A(n)$, then we have the following recursive relation:

$$A(n) = 7 \times A[\frac{n}{2}]A(1) = 1$$

solve this gives:

$$A(n) = 7^{\log_2 n}$$

hence

$$A(n) \in n^{\log_2 7}$$

4.3 The disadvantages

It can be seen from the equations given above that there are extra addition operations required to compute the output. If we denote the number of additions required for a $n \times n \times n$ routine by $B(n)$, then we observe the recursion means there are 7 matrix multiplications of size $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$, and 18 matrix addition of size $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$

We know that the matrix addition has complexity $O(n^2)$, hence we have the equality:

$$B(n) = 7 \times B(\frac{n}{2}) + 18 \times O((\frac{n}{2})^2)$$

Master's theorem:
For recurrence relation:

$$T(n) = aT(\frac{n}{b}) + f(n) where a > 1, b > 1$$

if

$$f(n) \in O(n^c) where c < \log_b a$$

then

$$T(n) \in \Theta(n^{\log_b a})$$

We have that $2 < \log_2 7$, hence:

$$B(b) \in O(n^{\log_2 7})$$

Note although the asymptotic complexity is the same, but the hidden constant is much larger

4.4 Other issues

All the discussion of strassen's algorithm has so far assumed that the input size is nice, i.e. the matrix dimension are powers of 2 (hence the recursion can always divide the matrix into 4 equal-sized sub-matrices)

The typical treatment for matrices of which the size is not 2^k is to pad it with zero entries to make it so. This will imply that there will be a lot of wasted computation in the actual work. For example, a $800 \times 800 \times 800$ routine, the matrices need to be padded to $1024 \times 1024 \times 1024$, this is not efficient at all!

there are 2 ways we can tackle this problem. Firstly, notice $800 = 32 \times 25$ the matrix can be divided recursively until we reach the size 25, at which point a general approach can be used (However, this means it will not be cache-oblivious anymore)
Secondly, one can use a lazy-padding scheme. For example, when we have input of size $199 \times 199 \times 199$, we divide them into four sub-matrices of size 100×100, 100×99, 99×100 99×99. We then pad the other three smaller matrices to the size 100×100. There is no need to increment the size to 256×256 at the first place.
the two tricks presented above can be hybridized together to recursively reduce the concerning matrix is smaller sizes. If we allow the size to drop to 1, then effectively it is a cache-oblivious method. Otherwise, we can choose a threshold, for example, 25, at which point we can apply a standard blocking algorithm to efficiently compute the product.