ThreadGo: A multithreaded hybrid acceleration of the matrix multiplication

by Shrohan Mohapatra
Email ID: shrohanapple@gmail.com

July 5, 2019

1 Introduction

There have been several formal algorithmic attempts towards the optimisation of the matrix multiplication, some focussing on exploration of bilinear and trilinear algorithms [1, 7, 5, 3, 2], and others being averse to calculate the matrix multiplication [4]. The fastest known algorithm for matrix multiplication is the one given by Le Gall [6] which is of complexity $O(n^{2.3728639})$. But what lies in the hindsight of such theoretical algorithm designs is that

1. Post-Strassen sub-cubic matrix multiplication algorithms are *mere approximations*. These show an asymptotic possibility of the tensor power of a certain algebraic identity to work out that way.

2. Also, the complexity of these algorithms often do not imply their pragmatic applicability because the associated constant of proportionality is *estimably large*, which may render these slower than the existing *asymptotically slower* algorithms.

The current fastest *serial algorithm* for matrix multiplication popular in the commercial domain is the Strassen's algorithm, which is known to run fairly faster than the standard school-book algorithm. Many cache-aware and cache-oblivious algorithm designs have been proposed [8, 9, 10] in order to accelerate the process of matrix multiplication in a single core. To improve further upon the computational aspect of the same, many parallel and distributed versions [11, 12, 13] of the school book algorithm have been suggested.

In this Python 3.7 package "ThreadGo", I have tried the implementation of a hybrid matrix multiplication algorithm that switches between the multithreaded versions of the Strassen's algorithm and the naive definition as per the availability of the virtual memory to the program. The package assumes significant number of cores for the platform and innocently launches scalable number of threads into the environment in an attempt to parallelise the *sequential independent* sections of the individual algorithms. One could easily explain the

design formally using the *cellular automata based approach* [14] but that would unnecessarily complicate the overall mathematical foundations. So in the subsequent sections of this documentation, I would sketch the pen portrait of the strategy of parallelization of the respective sections of the algorithm.

2 Parallelization of the school book algorithm

The mathematical definition of the multiplication of two $n \times n$ matrices $A_{i,j}$ and $B_{i,j}$ is as follows

$$(AB)_{(i,j),1\leq i\leq n,1\leq j\leq n} = \sum_{k=1}^{n} A_{i,k}B_{k,j} \tag{1}$$

Algorithm 1 Naive matrix multiplication

Require: Input $n \times n$ matrices A and B
Ensure: Output matrix $C = A \times B$
  $C \leftarrow$[[0,0,0,0, $\cdots (n \text{ times}) \cdots 0]$, $\cdots (n \text{ times}) \cdots$, $[0, \cdots (n \text{ times}) \cdots 0]$]
  for $i = 1; i \leq n; i++$ **do**
    for $j = 1; j \leq n; j++$ **do**
      for $k = 1; k \leq n; k++$ **do**
        $C[i][j] \leftarrow C[i][j] + A[i][k] * B[k][j]$
      end for
    end for
  end for

A simple pseudocode of the direct definition-based implementation has been shown in algorithm 1. But one can notice that line 4 in algorithm 1, i.e. the operation $C[i][j] \leftarrow C[i][j] + A[i][k] * B[k][j]$ is trivially the only *sequential independent* part of the code. What the ThreadGo package does is very simple; it launches n^3 threads that perform the updation $C[i][j] \leftarrow C[i][j] + A[i][k] * B[k][j]$ for all 3-tuples $(i, j, k), 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq n$. So even though the number of threads grows in $O(n^3)$, the time complexity of the parallel algorithm is $O(1)$.

3 Parallelization of the Strassen's algorithm

In the Strassen's algorithm, the input $n \times n$ matrices A and B and the output $n \times n$ matrix C, is broken into 12 $\frac{n}{2} \times \frac{n}{2}$ sub-matrices as follows.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{2}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{4}$$

Then the algorithm calls itself seven times recursively with the following multiplicands.

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

From the products $M_1, M_2, \cdots M_7$, the components of the matrix product are calculated.

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

If one were to follow the Strassen's algorithm in its serial mode, the time complexity $T_{serial}(n)$ follows the recurrence relation

$$T_{serial}(n) = 7 \cdot T_{serial}\left(\frac{n}{2}\right) + k_1 n^2 \tag{5}$$

k_1 being a constant. From Master's theorem, $T_{serial}(n) = O(n^{log_2 7})$. In ThreadGo, two aspects of the code are scalably multi-threaded.

1. The seven recursive matrix multiplication procedure calls to calculate the product $M_1, M_2, \cdots M_7$.
2. Basic completely parallelizable matrix operations such as initialisation, addition, subtraction etc.

Hence the time complexity of the scalably parallel version of Strassen's algorithm $T_{parallel}(n)$ follows the recurrence relation

$$T_{parallel}(n) = T_{parallel}\left(\frac{n}{2}\right) + k_2 \tag{6}$$

k_2 being another constant. From Master's theorem, $T_{parallel}(n) = O(log(n))$. Also the number of threads $F(n)$ follows a recurrence relation similar to that in equation 5,

$$F(n) = 7 \cdot F\left(\frac{n}{2}\right) + k_3 n^2 \tag{7}$$

k_3 being yet another constant. So the complexity of ThreadGo based implementation of the Strassen's algorithm in terms of the number of threads is $O(n^{log_2 7})$. Even though the number of threads are asymptotically less, in a pragmatic setup, the number of recursive calls and subsequent thread joins take more time than the parallel version of the naive implementation, which actually accosts to a lot of thread switching time, thus having an adverse effect of the total time. Thus, ThreadGo makes a "switch" of the algorithms at certain input size from the parallel version of the Strassen's algorithm to that of the naive implementation.

4 A deeper insight to ThreadGo

4.1 How to install

Installation is completely hassle-free; just download the ThreadGo package, root it to your favourite folder, and import it in your program from the path properly whenever you want to use it. Since it's a Python 3.7.3 in which this package has been written, it's preferable to import it in a program written in Python 3.7.3 or higher.

4.2 What else it contains

For beginners, hobbyists, end-use programmers and other enthusiasts, the root ThreadGo folder contains a script 'example.py' that serves as a simple demonstration as to how one can use ThreadGo package. For those who want to know what and how the script has been written, there is a 'project' sub-directory that consists of several files.

1. For contributors, analysts and others, the source code is there in the program 'threadGo.py'. Since it is very simply written and designed, it will be very easy for the ones well-versed with Python, multithreading and object-oriented design to understand the model.

2. For accessors and analysts who are curious to know whether the algorithm discussed is practically correct or not, there is a program 'correctnessTest.py'. This program performs randomised testing by means of partitioning the input space into four of my wishful categories: Non-negative integers, integers, non-negative floating point numbers and the negative ones; and picking random test cases from each of the categories. The tests are performed quickly so the program can rerun for as many as times as the accessor wants.

3. Also, to exhibit the robustness of algorithm, there is a program 'performanceTest.py' that essentially pops up a plot of the time and space complexities, one by one (the first plot reveals the time complexity, so if you close that figure the other plot pops up), against the growing size of the matrix. The 'spike' noticeable in the plots show the 'switch' of the algorithm.

4. For enthusiasts questioning how and when to use matrix multiplication algorithm, the program 'transitiveClosure.py' serves as an inspiration for them. This program takes a random directed unweighted graph in the form of an adjacency matrix and computes its transitive closure using a reduction to the matrix multiplication algorithm presented in [8].

References

[1] V. Strassen, "Gaussian Elimination is not optimal", *Numer. Math.*, 13:354-356, 1969

[2] V. Williams, "Multiplying matrices in $O(n^{2.373})$ time", *ACM Press*, pp. 887-8988, 2014

[3] H. Cohn, R. Kleinberg, B. Szegedy, C. Umans, "Group-theoretic Algorithms for Matrix Multiplication", 46^{th} *Annual IEEE Symposium in Foundations of Computer Sciences*, 2005

[4] R. Raz, "On the complexity of matrix multiplication product", *ACM Press*, 2002

[5] D. Coppersmith, S. Winograd, "Matrix multiplication via arithmetic progression", *Journal of Symbolic Computation*, pp. 251-280, 1990

[6] F. Le Gall, "Powers of tensors and fast matrix multiplication", *Proceedings of the* 39^{th} *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2014

[7] A. Schönhange, "Partial and Total Matrix Multiplication", *SIAM Journal of Computation*, Volume 10, pp.434-455, 1981

[8] Skiena, Steven, "Sorting and Searching", *The Algorithm Design Manual, Springer*, pp. 4546, 401403, 2008

[9] Prokop, Harald, "Cache-Oblivious Algorithms", 1999, *Master's Thesis (Massachusetts Institute of Technology)*

[10] Lam, Monica S., Rothberg, Edward E., Wolf, Michael E., "The Cache Performance and Optimizations of Blocked Algorithms", *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991

[11] Randall, Keith H., "Cilk: Efficient Multithreaded Computing", *Ph.D. Thesis (Massachusetts Institute of Technology)*, pp. 54-57, 1998

[12] Irony, D., Toledo, S., Tiskin, A., "Communication lower bounds for distributed-memory matrix multiplication", *J. Parallel Distrib. Comput. 64*, 9:1017-1026, 2004

[13] Kak, S., "Efficiency of matrix multiplication on the cross-wired mesh array", `arXiv:1411.3273`, 2014

[14] Mohapatra, S., "Novel applications of cellular automata in computing and computational astrophysics", *B.Tech. Thesis(Indian Institute of Technology Bhubaneswar)*, 2019