Matrix Algorithms

Matrix Multiplication Algorithms – A Survey

Ashwath Venkataraman | University of Florida | UFID: 5198-9461 | COT5405

Abstract/Introduction

Matrix Algorithms have so many important applications in various fields of computer science and often amount to large amounts of computing space. They are found in many applications including scientific computing, pattern recognition, medical, image analysis etc. and in basic applications such as graphs. When computational work is spread over multiple systems, say parallel or distributed systems, matrix algorithms play a major role in computation. Matrix multiplication as we know multiplies two matrices of order n corresponding to rows in a matrix A and columns in matrix B to give product AB. Because it is predominant and central to so many mathematical problems, there is lot of work done to make matrix algorithms efficient. Matrix multiplication is a core building block for numerous scientific computing and, more recently, machine learning applications

The generic algorithm to multiply two matrices takes time on the order of n^3 [$O(n^3)$ in the BigO notation for nxn matrices]. Better asymptotic bounds have been provided by means of various algorithms, but it is still unknown what the most efficient or optimum time is. Proposed solutions are disagreed and hence this is an interesting subtopic of matrix algorithms that makes its way into the list of unsolved computer science problems. Theoretically the perfect time for a nxn order matrix multiplication should be $O(n^2)$.

In this survey we review through the various algorithms, their backdrops, results, problems and their subsequent improved versions [9].

A Review of the algorithms with their improved versions

Simple Iterative Algorithm

The simple iterative algorithm is given by the definition of matrix multiplication that is Matrix C = AB where A is n x q matrix and B is q x p matrix, then C is a n x p matrix given by:

$C_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$

```
for i from 1 to n
  for j from 1 to p
  Cij is the product and sum = 0
    for k from 1 to q
       sum = sum +  Aik × Bkj
       sum = Cij
return sum
```

The above algorithm takes a time complexity of nxqxp, that is, O(nqp) , if matrix order is assumed to be nxn then the time complexity is $O(n^3)$ which is **cubic**.

Divide and Conquer Approach 1

In this algorithm approach we assume that n is always an exact power of 2. We partition the matrices A,B,C into four n/2 x n/2 matrices. This relies on block partitioning.

$$C=\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Multiplying A and B gives us 8 multiplications plus a base case where we keep C_{11} as $A_{11}.B_{11}$ computing the smaller multiplications recursively. The algorithm recursively computes the products and sums, is pretty easy and found in the CLRS book [1]. The complexity of this algorithm can be given easily as T(1)=O(1) and $T(n) = 8T(n/2) + O(n^2)$, for 8 multiplications, 4 additions and a partitioning. On applying the Masters theorem for divide and conquer recurrences, the time complexity is got as $O(n^3)$, which is **no better than the naive algorithm** in the previous section !

Strassens Matrix Multiplication algorithm, DQ Approach 2, improved exponent

As an improvement over the simple divide and conquer approach in the previous section, **Volker Strassen proved that the cubic time complexity is not optimal** and developed a faster than standard matrix multiplication algorithm that is useful for large matrices. The goal was to reduce the number of multiplications in the previous divide and conquer approach (from 8 to 7). In this algorithm we partition each of the matrices into four n/2 x n/2 sub-matrices. We then create 10 matrices S1,S2…S10, where each is a n/2 x n/2 matrix and is got from the sum or difference of the two matrices in the previous step. After this we recursively compute products P1,P2…P7 each n/2 x n/2. We get the final product by computing various combinations of P1..P7. Algorithm is pretty straightforward and can be found in the CLRS book [1].

This is similar to the previous DQ algorithm but has only 7 multiplications and hence the recurrence is given $7T(n/2) + O(n^2)$ whose complexity is given by $O(n^{\log 7}) = \mathbf{O(n^{2.807})}$. It is unclear how Strassen derived the individual sub matrix computation, but one can quote it with the symmetry aspect of the products.

The inherent considerations and problems lie in the assumptions of this algorithm proposed by Strassen. Matrices are considered to be square, and the size if a power of two and it is padded if needed. Even though this restriction simplifies explanation it is not necessary and padding will only increase the computation time. The question arises. Can we extend it to a 3x3 or higher

order matrices? We can do this trivially if we figure out the right transformations. Also Strassens can be extended to nxn matrices where n is not an exact power of 2 by padding of zeroes to rows and columns. However Strassens as a result is mainly used in favor of computational efficiency required, rather than its simplicity (Strassen's is not simple!)

The divide and conquer approaches comes with various bottlenecks like communication costs, in distributed systems, when Strassens keeps breaking up matrices into smaller chunks, there is a heavy burden placed. Also disk and ram differences are bottlenecks for recursive algorithms.

Other Algorithms superior to Strassen's method – Improving the exponent further

Strassens method was an amazing one and for over a decade no one could devise an algorithm better than that. However it spawned a long line of research to reduce the exponent (2.807) over time.

In 1978, **V.Ya Pan [6,5]** found algorithms to reduce the exponent value further using a technique known as **trilinear aggregation**. Pan showed that two 70x70 matrices can be accelerated in 143640 operations. We can understand trilinear coordinates of a matrix as ; given a reference triangle ABC we have another trilinear matrix with coordinates A'B'C'. The vertex matrix has trilinear coordinates $\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$. Given a reference triangle the trilinear coordinates of a point p with respect to triangle ABC are an ordered triple of numbers, each of which is proportional to the directed distance from P to one of the sides and denoted by α,β,γ and can also be multiplied with a non-zero constant(constants are huge in this case). For simplicity, the three vertices A,B,C are commonly written as 1:0:0, 0:1:0, 0:0:1 as represented by the matrix. This is done for n vector ordered matrix. Matrices in this method are represented as linear forms and decomposed using constants that are not large according to Pan. He managed to bring the exponent value to **2.79** (An improvement to the 2.807 of Strassens method)

In 1979**, Bini [4]** further reduced the number of operations by using **bilinear approximate algorithms** and algorithm similar to that of trilinear aggregation and brought down the exponent further to **2.78**. For matrix multiplication we can prove a normal form for all algorithms by using the following specifications to form a bilinear form – certain linear combinations α_i,β_i of input and output matrices a_{jk} and b_{jk} are calculated respectively, we calculate γ_i as $\alpha_i x \beta_i$, each entry in the output matrix is linear combination of γ_is . Using a tensor based approach (recursively applying same algorithm), minimum number of products to compute matrix multiplication can be put as finding the rank or the dimension of a tensor. A tensor is a generalized form of a matrix based on transformation rules.

In 1981, **Schonage [6]** developed a complex theory to show that these bilinear approximation algorithms are very powerful and involving bilinear complexity of rectangular matrix

multiplication. Using his devised **sum inequality theorem** he proved the exponent to be less than **2.55**. This is not the winner however!

The fastest matrix multiplication algorithm atleast asymptotically as of now is given by **Coppersmith–Winograd in 1982 [2,8]**. It was an improvement to Strassens algorithm (reloaded version) that used a trilinear form. In Strassens trilinear form, the variables are collected into blocks. The block structure and the fine structure(arrangement of blocks and variables) are matrix products, however the overall structure is not. After taking a tensor power of the blocks he reduces it to several scalar disjoint multiplications that is further approximated by using Schonage`s theorem giving the exponent to be less than 2.49. Coppersmith-Winograd reduced this even further to **2.376** by using Strassens improvement and hashing the trilinear form of products to form an arithmetic progression.

A thesis by the **University of Cambridge [3]** combined **dynamic multithreading** and the Strassens method to produce a parallelized algorithm, that takes $O(n^3)$ work and O(logn) span. Total time to execute everything on a single processor is the work and span refers to the longest time to execute threads along any path. Work law and span law of multithreading, i.e work on p processors can't be less than time on infinite processors, along with the usage of parallel for loops and a series of spawns led to a critical length O(log n) span time, a faster version of the algorithm, not practical however, since there are communication costs.

Implications of all the algorithms

To summarize the various algorithms presented for matrix multiplication in this survey towards improving the exponent value; the complexities (exponent value of n) are given by: **Naive approach – 3, Strassen – 2.808, Pan – 2.796, Schonage – 2.522, Strassen(Reloaded) – 2.496, Coppersmith and Winograd – 2.376 [3]**. Even though various algorithms have been devised to reduce the exponent, practical applications of the said improvement is still currently not prevalent, besides the Strassens algorithm. Coppersmith-Winograd's is the clear winner with the lowest exponent, however it is very complex and due to the large constant factors involved, it is impractical in nature and there is no proper tensor embedding in the bilinear problem. Having said this there is nothing preventing us from practically implementing them. It would be very complicated, and crossover point on where it would improve over the naive cubic algorithm is massive (there is not really an improvement to observe). Strassens algorithm is our best bet so far as it performs better than cubic algorithm when the dimension of n gets larger than 100. The crossover point may depend on the system. On summation the naive $O(n^3)$ algorithm is likely our best bet unless the matrices are very large. We can simply say that for a sparse matrix, the naive algorithm works well, for a dense matrix Strassen's is better in terms of real world performance.

In a pragmatic approach, the best way to perform fast matrix multiplication algorithm would be to take a slow algorithm, the naive approach and run it in parallel. Modern processors may be

able to do 8 additions in one instruction as fast as a single one. We can use multiple processors (multiple cores, multiple memory architectures) and if implemented correctly the total runtime complexity may be one cycle per eight instructions per processor. However there is also a communication complexity that gets introduced here, moving data from and to a RAM and cache or in distributed systems, between nodes.

Future Scope/Conclusion

Future scope for topic such as matrix multiplication is left to interpretation as such, with so many algorithms being devised and still not being optimal or practical in terms of implementations. However with the multitude of computing resource available it is safe to say that we can parallel process any amount of computations – multiplications and additions (even if it uses the naive algorithm) by using combinations of forks and spawns to easily process numerous computations using the platform's vector instructions.

Matrix multiplication is still not clear in terms of what the optimal complexity is even though better asymptotic bounds are being calculated. It hence makes it way into the list of unsolved computer science problems [7]. Even after so much work being invested into this, the question arises – **"Is it possible to achieve the perfect $O(n^2)$?"**

References

1) https://mitpress.mit.edu/books/introduction-algorithms-third-edition - **CLRS Book 3rd Edition**
2) http://www.cs.umd.edu/~gasarch/TOPICS/ramsey/matrixmult.pdf - **Don Coppersmith and Shmuel Winograd – Matrix Multiplication via Arithmetic Progression**
3) https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/lec4_ann.pdf - **University of Cambridge Thesis**
4) https://www.maths.ed.ac.uk/sites/default/files/atoms/files/stothers.pdf - **On the Complexity of Matrix Multiplication, Andrew James Stothers**
5) https://pdfs.semanticscholar.org/4ff0/0f3aa791d40841f09675bdc153f3cf50cb52.pdf- **Fast Matrix Multiplication and Symbolic Computation, Victor Y. Pan**
6) https://epubs.siam.org/action/showAbstract?page=434&volume=10&issue=3&journalCode=smjcat - **Partial and Total Matrix Multiplication – A. Schonage**
7) https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_computer_science
8) https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm