Matrix multiplication algorithm

Because matrix multiplication is such a central operation in many numerical algorithms, much work has been invested in making **matrix multiplication algorithms** efficient. Applications of matrix multiplication in computational problems are found in many fields including scientific computing and pattern recognition and in seemingly unrelated problems such as counting the paths through a graph.[1] Many different algorithms have been designed for multiplying matrices on different types of hardware, including parallel and distributed systems, where the computational work is spread over multiple processors (perhaps over a network).

> **Unsolved problem in computer science**:
>
> *What is the fastest algorithm for matrix multiplication?*
>
> (more unsolved problems in computer science)

Directly applying the mathematical definition of matrix multiplication gives an algorithm that takes time on the order of n^3 to multiply two $n \times n$ matrices ($\Theta(n^3)$ in big O notation). Better asymptotic bounds on the time required to multiply matrices have been known since the work of Strassen in the 1960s, but it is still unknown what the optimal time is (i.e., what the complexity of the problem is).

Contents

Iterative algorithm

The definition of matrix multiplication is that if $C = AB$ for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix with entries

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

From this, a simple algorithm can be constructed which loops over the indices i from 1 through n and j from 1 through p, computing the above using a nested loop:

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- For i from 1 to n:
    - For j from 1 to p:
        - Let sum = 0
        - For k from 1 to m:
            - Set sum $\leftarrow$ sum $+ A_{ik} \times B_{kj}$
        - Set $C_{ij} \leftarrow$ sum
- Return C

This algorithm takes time $\Theta(nmp)$ (in asymptotic notation).[1] A common simplification for the purpose of algorithms analysis is to assume that the inputs are all square matrices of size $n \times n$, in which case the running time is $\Theta(n^3)$, i.e., cubic.[2]
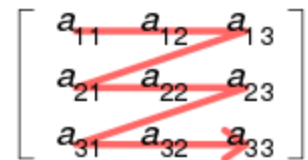
Cache behavior

The three loops in iterative matrix multiplication can be arbitrarily swapped with each other without an effect on correctness or asymptotic running time. However, the order can have a considerable impact on practical performance due to the memory access patterns and cache use of the algorithm;[1] which order is best also depends on whether the matrices are stored in row-major order, column-major order, or a mix of both.

In particular, in the idealized case of a fully associative cache consisting of M cache lines of b bytes each, the above algorithm is sub-optimal for A and B stored in row-major order. When $n > \frac{M}{b}$, every iteration of the inner loop (a simultaneous sweep through a row of A and a column of B) incurs a cache miss when accessing an element of B. This means that the algorithm incurs $\Theta(n^3)$ cache misses in the worst case. As of 2010, the speed of memories compared to that of processors is such that the cache misses, rather than the actual calculations, dominate the running time for sizable matrices.[3]

Illustration of row- and column-major order

The optimal variant of the iterative algorithm for A and B in row-major layout is a *tiled* version, where the matrix is implicitly divided into square tiles of size $\sqrt{M}$ by $\sqrt{M}$:[3][4]

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- Pick a tile size $T = \Theta(\sqrt{M})$
- For I from 1 to n in steps of T:
    - For J from 1 to p in steps of T:
        - For K from 1 to m in steps of T:
            - Multiply $A_{I:I+T,\ K:K+T}$ and $B_{K:K+T,\ J:J+T}$ into $C_{I:I+T,\ J:J+T}$, that is:
            - For i from I to $\min(I + T, n)$:
                - For j from J to $\min(J + T, p)$:
                    - Let sum = 0
                    - For k from K to $\min(K + T, m)$:
                        - Set sum $\leftarrow$ sum $+ A_{ik} \times B_{kj}$

- Set $C_{ij} \leftarrow C_{ij} + \text{sum}$

- Return C

In the idealized cache model, this algorithm incurs only $\Theta(\frac{n^3}{b\sqrt{M}})$ cache misses; the divisor $b\sqrt{M}$ amounts to several orders of magnitude on modern machines, so that the actual calculations dominate the running time, rather than the cache misses.[3]

Divide and conquer algorithm

An alternative to the iterative algorithm is the divide and conquer algorithm for matrix multiplication. This relies on the block partitioning

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

which works for all square matrices whose dimensions are powers of two, i.e., the shapes are $2^n \times 2^n$ for some n. The matrix product is now

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

which consists of eight multiplications of pairs of submatrices, followed by an addition step. The divide and conquer algorithm computes the smaller multiplications recursively, using the scalar multiplication $c_{11} = a_{11}b_{11}$ as its base case.

The complexity of this algorithm as a function of n is given by the recurrence[2]

$$T(1) = \Theta(1);$$
$$T(n) = 8T(n/2) + \Theta(n^2),$$

accounting for the eight recursive calls on matrices of size $n/2$ and $\Theta(n^2)$ to sum the four pairs of resulting matrices element-wise. Application of the master theorem for divide-and-conquer recurrences shows this recursion to have the solution $\Theta(n^3)$, the same as the iterative algorithm.[2]

Non-square matrices

A variant of this algorithm that works for matrices of arbitrary shapes and is faster in practice[3] splits matrices in two instead of four submatrices, as follows.[5] Splitting a matrix now means dividing it into two parts of equal size, or as close to equal sizes as possible in the case of odd dimensions.

- Inputs: matrices A of size $n \times m$, B of size $m \times p$.
- Base case: if $\max(n, m, p)$ is below some threshold, use an unrolled version of the iterative algorithm.
- Recursive cases:

  - If $\max(n, m, p) = n$, split A horizontally:

    $$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

  - Else, if $\max(n, m, p) = p$, split B vertically:

$$C = A(\, B_1 \quad B_2 \,) = (\, AB_1 \quad AB_2 \,)$$

- Otherwise, $\max(n, m, p) = m$. Split A vertically and B horizontally:

$$C = (\, A_1 \quad A_2 \,) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

Cache behavior

The cache miss rate of recursive matrix multiplication is the same as that of a tiled iterative version, but unlike that algorithm, the recursive algorithm is cache-oblivious:[5] there is no tuning parameter required to get optimal cache performance, and it behaves well in a multiprogramming environment where cache sizes are effectively dynamic due to other processes taking up cache space.[3] (The simple iterative algorithm is cache-oblivious as well, but much slower in practice if the matrix layout is not adapted to the algorithm.)
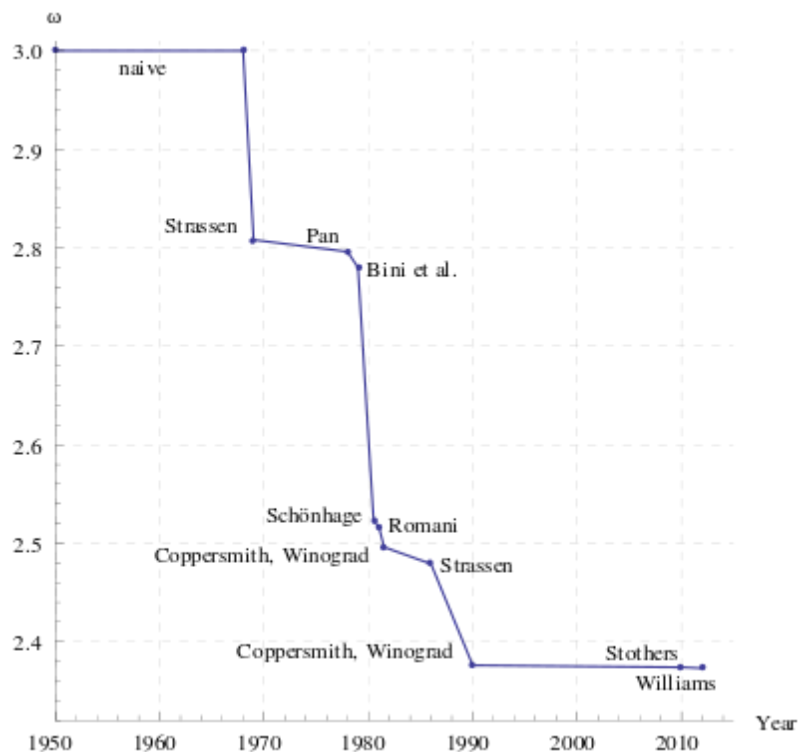
The number of cache misses incurred by this algorithm, on a machine with M lines of ideal cache, each of size b bytes, is bounded by[5]:13

$$\Theta\left(m + n + p + \frac{mn + np + mp}{b} + \frac{mnp}{b\sqrt{M}} \right)$$

Sub-cubic algorithms

Algorithms exist that provide better running times than the straightforward ones. The first to be discovered was Strassen's algorithm, devised by Volker Strassen in 1969 and often referred to as "fast matrix multiplication". It is based on a way of multiplying two 2×2-matrices which requires only 7 multiplications (instead of the usual 8), at the expense of several additional addition and subtraction operations. Applying this recursively gives an algorithm with a multiplicative cost of $O(n^{\log_2 7}) \approx O(n^{2.807})$. Strassen's algorithm is more complex, and the numerical stability is reduced compared to the naïve algorithm,[6] but it is faster in cases where $n > 100$ or so[1] and appears in several libraries, such as BLAS.[7] It is very useful for large matrices over exact domains such as finite fields, where numerical stability is not an issue.

The lowest ω such that matrix multiplication is known to be in $O(n^\omega)$, plotted against time.

The current $O(n^k)$ algorithm with the lowest known exponent k is a generalization of the Coppersmith–Winograd algorithm that has an asymptotic complexity of $O(n^{2.3728639})$, by François Le Gall.[8] The Le Gall algorithm, and the Coppersmith–Winograd

algorithm on which it is based, are similar to Strassen's algorithm: a way is devised for multiplying two $k \times k$-matrices with fewer than k^3 multiplications, and this technique is applied recursively. However, the constant coefficient hidden by the Big O notation is so large that these algorithms are only worthwhile for matrices that are too large to handle on present-day computers.[9][10]

Since any algorithm for multiplying two $n \times n$-matrices has to process all $2n^2$ entries, there is an asymptotic lower bound of $\Omega(n^2)$ operations. Raz proved a lower bound of $\Omega(n^2 \log(n))$ for bounded coefficient arithmetic circuits over the real or complex numbers.[11]

Cohn *et al.* put methods such as the Strassen and Coppersmith–Winograd algorithms in an entirely different group-theoretic context, by utilising triples of subsets of finite groups which satisfy a disjointness property called the triple product property (TPP). They show that if families of wreath products of Abelian groups with symmetric groups realise families of subset triples with a simultaneous version of the TPP, then there are matrix multiplication algorithms with essentially quadratic complexity.[12][13] Most researchers believe that this is indeed the case.[10] However, Alon, Shpilka and Chris Umans have recently shown that some of these conjectures implying fast matrix multiplication are incompatible with another plausible conjecture, the sunflower conjecture.[14]

Freivalds' algorithm is a simple Monte Carlo algorithm that, given matrices A, B and C, verifies in $\Theta(n^2)$ time if $AB = C$.

Parallel and distributed algorithms

Shared-memory parallelism

The divide and conquer algorithm sketched earlier can be parallelized in two ways for shared-memory multiprocessors. These are based on the fact that the eight recursive matrix multiplications in

$$
\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}
$$

can be performed independently of each other, as can the four summations (although the algorithm needs to "join" the multiplications before doing the summations). Exploiting the full parallelism of the problem, one obtains an algorithm that can be expressed in fork–join style pseudocode:[15]

Procedure multiply(C, A, B):

- Base case: if $n = 1$, set $c_{11} \leftarrow a_{11} \times b_{11}$ (or multiply a small block matrix).
- Otherwise, allocate space for a new matrix T of shape $n \times n$, then:

    - Partition A into $A_{11}, A_{12}, A_{21}, A_{22}$.
    - Partition B into $B_{11}, B_{12}, B_{21}, B_{22}$.
    - Partition C into $C_{11}, C_{12}, C_{21}, C_{22}$.
    - Partition T into $T_{11}, T_{12}, T_{21}, T_{22}$.
    - Parallel execution:

        - *Fork* multiply(C_{11}, A_{11}, B_{11}).
        - *Fork* multiply(C_{12}, A_{11}, B_{12}).
        - *Fork* multiply(C_{21}, A_{21}, B_{11}).
        - *Fork* multiply(C_{22}, A_{21}, B_{12}).
        - *Fork* multiply(T_{11}, A_{12}, B_{21}).

- - - *Fork* multiply(T_{12}, A_{12}, B_{22}).
    - *Fork* multiply(T_{21}, A_{22}, B_{21}).
    - *Fork* multiply(T_{22}, A_{22}, B_{22}).
  - *Join* (wait for parallel forks to complete).
  - add(C, T).
  - Deallocate T.

Procedure add(C, T) adds T into C, element-wise:

- Base case: if $n = 1$, set $c_{11} \leftarrow c_{11} + t_{11}$ (or do a short loop, perhaps unrolled).
- Otherwise:
  - Partition C into $C_{11}, C_{12}, C_{21}, C_{22}$.
  - Partition T into $T_{11}, T_{12}, T_{21}, T_{22}$.
  - In parallel:
    - *Fork* add(C_{11}, T_{11}).
    - *Fork* add(C_{12}, T_{12}).
    - *Fork* add(C_{21}, T_{21}).
    - *Fork* add(C_{22}, T_{22}).
  - *Join*.

Here, *fork* is a keyword that signal a computation may be run in parallel with the rest of the function call, while *join* waits for all previously "forked" computations to complete. partition achieves its goal by pointer manipulation only.

This algorithm has a critical path length of $\Theta(\log^2 n)$ steps, meaning it takes that much time on an ideal machine with an infinite number of processors; therefore, it has a maximum possible speedup of $\Theta(n^3/\log^2 n)$ on any real computer. The algorithm isn't practical due to the communication cost inherent in moving data to and from the temporary matrix T, but a more practical variant achieves $\Theta(n^2)$ speedup, without using a temporary matrix.[15]

Communication-avoiding and distributed algorithms

On modern architectures with hierarchical memory, the cost of loading and storing input matrix elements tends to dominate the cost of arithmetic. On a single machine this is the amount of data transferred between RAM and cache, while on a distributed memory multi-node machine it is the amount transferred between nodes; in either case it is called the *communication bandwidth*. The naïve algorithm using three nested loops uses $\Omega(n^3)$ communication bandwidth.

Cannon's algorithm, also known as the *2D algorithm*, is a communication-avoiding algorithm that partitions each input matrix into a block matrix whose elements are submatrices of size $\sqrt{M/3}$ by $\sqrt{M/3}$, where M is the size of fast memory.[16] The naïve algorithm is then used over the block matrices, computing products of submatrices entirely in fast memory. This reduces communication bandwidth to $O(n^3/\sqrt{M})$, which is asymptotically optimal (for algorithms performing $\Omega(n^3)$ computation).[17][18]

In a distributed setting with p processors arranged in a $\sqrt{p}$ by $\sqrt{p}$ 2D mesh, one submatrix of the result can be assigned to each processor, and the product can be computed with each processor transmitting $O(n^2/\sqrt{p})$ words, which is asymptotically optimal assuming that each node stores the minimum $O(n^2/p)$ elements.[18] This can be improved by the *3D algorithm*, which arranges the processors in a 3D cube mesh, assigning every product of two input submatrices to a single processor. The result submatrices are then generated by performing a reduction over each row.[19] This algorithm transmits $O(n^2/p^{2/3})$ words per processor, which is asymptotically optimal.[18] However, this requires replicating each input matrix element $p^{1/3}$ times, and so requires a factor of $p^{1/3}$ more memory than is needed to store the inputs. This algorithm can be combined with Strassen to further reduce runtime.[19]

"2.5D" algorithms provide a continuous tradeoff between memory usage and communication bandwidth.[20] On modern distributed computing environments such as MapReduce, specialized multiplication algorithms have been developed.[21]

Algorithms for meshes

There are a variety of algorithms for multiplication on meshes. For multiplication of two $n{\times}n$ on a standard two-dimensional mesh using the 2D Cannon's algorithm, one can complete the multiplication in $3n\text{-}2$ steps although this is reduced to half this number for repeated computations.[22] The standard array is inefficient because the data from the two matrices does not arrive simultaneously and it must be padded with zeroes.

The result is even faster on a two-layered cross-wired mesh, where only $2n\text{-}1$ steps are needed.[23] The performance improves further for repeated computations leading to 100% efficiency.[24] The cross-wired mesh array may be seen as a special case of a non-planar (i.e. multilayered) processing structure.[25]

See also

- Computational complexity of mathematical operations
- CYK algorithm, §Valiant's algorithm
- Matrix chain multiplication
- Sparse matrix-vector multiplication

References

1. Skiena, Steven (2008). "Sorting and Searching". *The Algorithm Design Manual*. Springer. pp. 45–46, 401–403. doi:10.1007/978-1-84800-070-4_4 (https://doi.org/10.1007%2F978-1-84800-070-4_4). ISBN 978-1-84800-069-8.
2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. pp. 75–79. ISBN 0-262-03384-4.
3. Amarasinghe, Saman; Leiserson, Charles (2010). "6.172 Performance Engineering of Software Systems, Lecture 8" (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-8-cache-efficient-algorithms/). *MIT OpenCourseWare*. Massachusetts Institute of Technology. Retrieved 27 January 2015.
4. Lam, Monica S.; Rothberg, Edward E.; Wolf, Michael E. (1991). *The Cache Performance and Optimizations of Blocked Algorithms*. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).
5. Prokop, Harald (1999). *Cache-Oblivious Algorithms* (http://supertech.csail.mit.edu/papers/Prokop99.pdf) (PDF) (Master's). MIT.

Block matrix multiplication. In the 2D algorithm, each processor is responsible for one submatrix of C. In the 3D algorithm, every pair of submatrices from A and B that is multiplied is assigned to one processor.
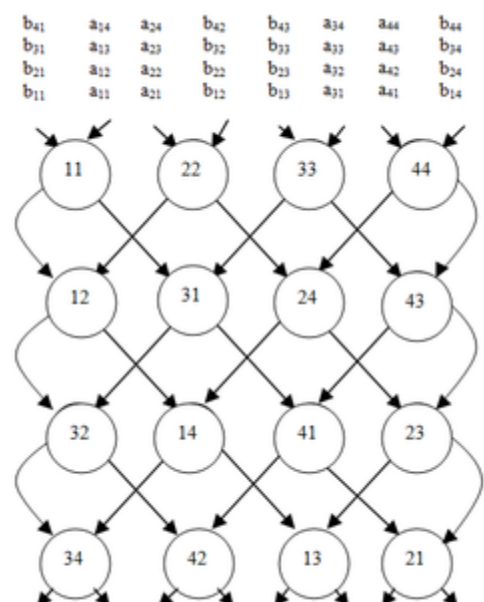
Matrix multiplication completed in 2n-1 steps for two n×n matrices on a cross-wired mesh.

6. Miller, Webb (1975), "Computational complexity and numerical stability", *SIAM News*, **4**: 97–107, CiteSeerX 10.1.1.148.9947 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.148.9947)

7. Press, William H.; Flannery, Brian P.; Teukolsky, Saul A.; Vetterling, William T. (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press. p. 108. ISBN 978-0-521-88068-8.

8. Le Gall, François (2014), "Powers of tensors and fast matrix multiplication", *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC 2014)*, arXiv:1401.7714 (https://arxiv.org/abs/1401.7714), Bibcode:2014arXiv1401.7714L (http://adsabs.harvard.edu/abs/2014arXiv1401.7714L). The original algorithm was presented by Don Coppersmith and Shmuel Winograd in 1990, has an asymptotic complexity of $O(n^{2.376})$. It was improved in 2013 to $O(n^{2.3729})$ by Virginia Vassilevska Williams, giving a time only slightly worse than Le Gall's improvement: Williams, Virginia Vassilevska. "Multiplying matrices faster than Coppersmith-Winograd" (http://www.cs.stanford.edu/~virgi/matrixmult-f.pdf) (PDF).

9. Iliopoulos, Costas S. (1989), "Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix" (http://www.williamstein.org/home/wstein/www/home/pernet/Papers/Hermite/Iliopoulos88.pdf) (PDF), *SIAM Journal on Computing*, **18** (4): 658–669, CiteSeerX 10.1.1.531.9309 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.531.9309), doi:10.1137/0218045 (https://doi.org/10.1137%2F0218045), MR 1004789 (https://www.ams.org/mathscinet-getitem?mr=1004789), "The Coppersmith–Winograd algorithm is not practical, due to the very large hidden constant in the upper bound on the number of multiplications required."

10. Robinson, Sara (2005). "Toward an Optimal Algorithm for Matrix Multiplication" (http://www.siam.org/pdf/news/174.pdf) (PDF). *SIAM News*. **38** (9).

11. Ran Raz. On the complexity of matrix product. In Proceedings of the thirty-fourth annual ACM symposium on Theory of computing. ACM Press, 2002. doi:10.1145/509907.509932 (https://doi.org/10.1145%2F509907.509932).

12. Henry Cohn, Robert Kleinberg, Balázs Szegedy, and Chris Umans. Group-theoretic Algorithms for Matrix Multiplication. arXiv:math.GR/0511460. *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, 23–25 October 2005, Pittsburgh, PA, IEEE Computer Society, pp. 379–388.

13. Henry Cohn, Chris Umans. A Group-theoretic Approach to Fast Matrix Multiplication. arXiv:math.GR/0307321. *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 11–14 October 2003, Cambridge, MA, IEEE Computer Society, pp. 438–449.

14. Alon, Shpilka, Umans, On Sunflowers and Matrix Multiplication (http://eccc.hpi-web.de/report/2011/067/)

15. Randall, Keith H. (1998). *Cilk: Efficient Multithreaded Computing* (http://supertech.csail.mit.edu/papers/randall-phdthesis.pdf) (PDF) (Ph.D.). Massachusetts Institute of Technology. pp. 54–57.

16. Lynn Elliot Cannon, *A cellular computer to implement the Kalman Filter Algorithm (http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=905686)*, Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.

17. Hong, J. W.; Kung, H. T. (1981). "I/O complexity: The red-blue pebble game" (https://apps.dtic.mil/dtic/tr/fulltext/u2/a104739.pdf) (PDF). *STOC '81: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*: 326–333.

18. Irony, Dror; Toledo, Sivan; Tiskin, Alexander (September 2004). "Communication lower bounds for distributed-memory matrix multiplication". *J. Parallel Distrib. Comput.* **64** (9): 1017–1026. CiteSeerX 10.1.1.20.7034 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.7034). doi:10.1016/j.jpdc.2004.03.021 (https://doi.org/10.1016%2Fj.jpdc.2004.03.021).

19. Agarwal, R.C.; Balle, S. M.; Gustavson, F. G.; Joshi, M.; Palkar, P. (September 1995). "A three-dimensional approach to parallel matrix multiplication". *IBM J. Res. Dev*. **39** (5): 575–582. CiteSeerX 10.1.1.44.3404 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.3404). doi:10.1147/rd.395.0575 (https://doi.org/10.1147%2Frd.395.0575).

20. Solomonik, Edgar; Demmel, James (2011). "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms". *Proceedings of the 17th International Conference on Parallel Processing*. Part II: 90–109.

21. Bosagh Zadeh, Reza; Carlsson, Gunnar. "Dimension Independent Matrix Square Using MapReduce" (http://stanford.edu/~rezab/papers/dimsum.pdf) (PDF). Retrieved 12 July 2014.

22. Bae, S.E., T.-W. Shinn, T. Takaoka (2014) A faster parallel algorithm for matrix multiplication on a mesh array (https://www.sciencedirect.com/science/article/pii/S1877050914003858/pdf?md5=4d641ad288db14733fb2f82ee445c258&pid=1-s2.0-S1877050914003858-main.pdf&_valck=1). Procedia Computer Science 29: 2230-2240

23. Kak, S. (1988) A two-layered mesh array for matrix multiplication (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.8527&rep=rep1&type=pdf). Parallel Computing 6: 383-385

24. Kak, S. (2014) Efficiency of matrix multiplication on the cross-wired mesh array. https://arxiv.org/abs/1411.3273

25. Kak, S. (1988) Multilayered array computing (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.4753&rep=rep1&type=pdf). Information Sciences 45: 347-365

Further reading

- Buttari, Alfredo; Langou, Julien; Kurzak, Jakub; Dongarra, Jack (2009). "A class of parallel tiled linear algebra algorithms for multicore architectures". *Parallel Computing*. **35**: 38–53. arXiv:0709.1272 (https://arxiv.org/abs/0709.1272). doi:10.1016/j.parco.2008.10.002 (https://doi.org/10.1016%2Fj.parco.2008.10.002).
- Goto, Kazushige; van de Geijn, Robert A. (2008). "Anatomy of high-performance matrix multiplication". *ACM Transactions on Mathematical Software*. **34** (3): 1–25. CiteSeerX 10.1.1.140.3583 (https://citeseerx.ist.psu.edu/vi

ewdoc/summary?doi=10.1.1.140.3583). doi:10.1145/1356052.1356053 (https://doi.org/10.1145%2F1356052.135 6053).

- Van Zee, Field G.; van de Geijn, Robert A. (2015). "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". *ACM Transactions on Mathematical Software*. **41** (3): 1–33. doi:10.1145/2764454 (https://doi.org/10.1145%2F2764454).
- How To Optimize GEMM (https://github.com/flame/how-to-optimize-gemm)

This page was last edited on 12 June 2019, at 23:49 (UTC).