

October 2, 2026

Today: fast linear algebra

So far we've seen

- fast algos for polynomials
- parallel algos for polynomials
- parallel algos for linear algebra.

We'll complete our tour of algorithms with fast linear algebra.

The most important problem is matrix multiplication.

Given $A, B \in \mathbb{C}^{n \times n}$, how quickly can we compute $C := AB$?

$$C_{i,j} = \sum_{k=1}^n A_{i,k} B_{k,j}$$

Obvious algorithm: $O(n^3)$ arithmetic ops.

Theorem (Strassen 1969)

Can multiply $n \times n$ matrices using $O(n^w)$ ops for $w \leq \log_2 7 \approx 2.81$.

Proof Divide & conquer.

Assume n is a power of two (pad with zeroes otherwise).

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(-B_{11} + B_{21})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (-A_{11} + A_{21})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C = AB$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 \cdot M_2 + M_3 + M_6 \end{pmatrix}$$

$T_{\text{mult}}(n)$ = # ops to multiply $n \times n$ matrices

$T_{\text{add}}(n)$ = # ops to add $n \times n$ matrices.

$$T_{\text{add}}(n) \leq n^2$$

$$\begin{aligned} \text{Strassen: } T_{\text{mult}}(n) &\leq 7 \cdot T_{\text{mult}}(n/2) + 15 T_{\text{add}}(n) \\ &\leq 7 \cdot T_{\text{mult}}(n/2) + O(n^2) \end{aligned}$$

$$\Rightarrow T_{\text{mult}}(n) \leq O(n^{\log_2 7}). \quad \square$$

$$\begin{aligned} \text{Compare with obvious D\&C: } T_{\text{mult}}(n) &\leq 8 \cdot T_{\text{mult}}(n/2) + O(n^2) \\ &\Rightarrow T_{\text{mult}}(n) \leq O(n^3). \end{aligned}$$

There have been many improvements since Strassen. Currently $\omega \leq 2.371...$

$$\omega = \inf \left\{ \tau : \exists \text{ algorithm for matrix mult that uses } O(n^\tau) \text{ ops} \right\}$$

↑ Called the exponent of matrix mult

We'll see some of these improvements. But first, let's see how to use matrix mult for other problems.

Lemma. Triangular matrices can be inverted in $O(n^\omega)$ ops.

Proof. Divide & conquer. If M is $n \times n$ triangular, write

$$M = \begin{pmatrix} A & 0 \\ B & C \end{pmatrix}. \text{ Then } M^{-1} = \begin{pmatrix} A^{-1} & 0 \\ -BA^{-1} & C^{-1} \end{pmatrix}$$

$$\text{So } T_{\text{inv}}(n) \leq 2 \cdot T_{\text{inv}}(n/2) + O(n^\omega)$$

$$\leq \begin{cases} O(n^\omega) & \text{if } \omega > 2 \\ O(n^\omega \lg n) & \text{if } \omega = 2. \end{cases} \quad \square$$

For general matrices, we will use the Schur complement.

Lemma $A, B, C, D \in \mathbb{C}^{n \times n}$. D invertible

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & BD^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} A - CD^{-1}B & 0 \\ 0 & D \end{pmatrix} \cdot \begin{pmatrix} I & 0 \\ CD^{-1} & I \end{pmatrix}.$$

Schur complement $\swarrow$

Proof. Block Gaussian elimination.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix} = \begin{pmatrix} A - CD^{-1}B & B \\ 0 & D \end{pmatrix}$$

$$\begin{pmatrix} I & -BD^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} A - CD^{-1}B & B \\ 0 & D \end{pmatrix} = \begin{pmatrix} A - CD^{-1}B & 0 \\ 0 & D \end{pmatrix}$$

So

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & -BD^{-1} \\ 0 & I \end{pmatrix}^{-1} \begin{pmatrix} A - CD^{-1}B & 0 \\ 0 & D \end{pmatrix} \cdot \begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix}^{-1}.$$

$$\begin{pmatrix} I & -BD^{-1} \\ 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & BD^{-1} \\ 0 & I \end{pmatrix}$$

$$\begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix}^{-1} = \begin{pmatrix} I & 0 \\ CD^{-1} & I \end{pmatrix}. \quad \square$$

Lemma Can invert general matrices*
with $O(n^3)$ arithmetic operations.

* Almost all matrices.

Proof Divide & Conquer again.

Write $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix} \begin{pmatrix} (A - CD^{-1}B)^{-1} & \\ & D^{-1} \end{pmatrix} \begin{pmatrix} I & -B^{-1} \\ 0 & I \end{pmatrix}$$

$$\text{So } T_{\text{inv}}(n) \leq 2 \cdot T_{\text{inv}}\left(\frac{n}{2}\right) + O(n^w)$$

$$\Rightarrow T_{\text{inv}}(n) \leq \begin{cases} O(n^w) & \text{if } w > 2 \\ O(n^w \lg n) & \text{if } w = 2. \quad \square \end{cases}$$

This algorithm assumes D^{-1} and $(A - CD^{-1}B)^{-1}$ exist. This is true for almost all invertible matrices.

We can get an algorithm that works for all matrices if we allow randomness.

Fact Let $M \in \mathbb{C}^{n \times n}$ invertible. Sample $X, Y \in [10n]^{n \times n}$. Then

$$\Pr_{X, Y} \left[\begin{array}{c} \text{Bottom-right block of } XMY \\ \text{is invertible} \end{array} \right] \leq 1/10$$

Proof uses Cauchy-Binet identity + Schwartz-Zippel Lemma. Beyond scope of this lecture.

Lemma Can compute determinant of general $n \times n$ matrix in $O(n^w)$ arith ops.

Proof Divide & conquer + Schur complement

Write $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$.

$$M = \begin{pmatrix} I & BD^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} A - CD^{-1}B & 0 \\ 0 & D \end{pmatrix} \begin{pmatrix} I & 0 \\ CD^{-1} & I \end{pmatrix}$$

Use $\det(XY) = \det(X) \det(Y)$
 $\det \begin{pmatrix} X & 0 \\ 0 & Y \end{pmatrix} = \det(X) \det(Y)$.

$$\Rightarrow \det(M) = \det(A - CD^{-1}B) \det(D).$$

$$T_{\det}(n) \leq 2 T_{\det}(n/2) + O(n^w).$$

$$\Rightarrow T_{\det}(n) \leq \begin{cases} O(n^w) & \text{if } w > 2 \\ O(n^w \lg n) & \text{if } w = 2. \end{cases} \quad \square$$

To design faster algorithms for matrix mult, it will be useful to build a little theory around bilinear algorithms. Bilinear algorithms suffice to capture all algebraic algorithms for matrix multiplication, but we won't prove that today.

We'll warm up with linear algorithms.

Definition Let $T \in \mathbb{C}^{m \times n}$. A linear algorithm to compute $\vec{x} \mapsto T\vec{x}$ is a set of linear forms

$$a_1(\vec{x}) = \sum_i a_{1,i} x_i$$

$$\vdots$$

$$a_r(\vec{x}) = \sum_i a_{r,i} x_i$$

so that

$$(T\vec{x})_1, \dots, (T\vec{x})_m \in \text{span}\{a_1(\vec{x}), \dots, a_r(\vec{x})\}$$

The rank of the linear algorithm is r .

The linear complexity of T is the smallest r such that there is a rank- r linear algorithm computing $T\vec{x}$.

Example $T = \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix}$. $\left. \begin{array}{l} a_1(\vec{x}) = x_1 \\ a_2(\vec{x}) = x_2 \\ a_3(\vec{x}) = x_3 \end{array} \right\} \text{rank } 3$

Example $T = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$

$$\left. \begin{aligned} a_1(\vec{x}) &= x_2 + x_3 \\ a_2(\vec{x}) &= x_1 + x_3 \\ a_3(\vec{x}) &= x_1 + x_2. \end{aligned} \right\} \text{rank } 3.$$

Example $T = \begin{pmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix}$

$$\left. \begin{aligned} a_1(\vec{x}) &= x_2 + x_3 \\ a_2(\vec{x}) &= -x_1 + x_3 \\ a_3(\vec{x}) &= -x_1 - x_2. \end{aligned} \right\} \text{rank } 3.$$

$$\left. \begin{aligned} a_1(\vec{x}) &= x_1 + x_2 \\ a_2(\vec{x}) &= x_2 + x_3. \end{aligned} \right\} \text{rank } 2$$

$$\begin{aligned} (T\vec{x})_1 &= x_2 + x_3 = a_2(\vec{x}) \\ (T\vec{x})_2 &= -x_1 + x_3 = -a_1(\vec{x}) + a_2(\vec{x}) \\ (T\vec{x})_3 &= -x_1 - x_2 = -a_1(\vec{x}). \end{aligned}$$

A linear algorithm of rank r corresponds to a factorization

$$T\vec{x} = B A \vec{x}$$

$$T - m \times n$$

$$B - m \times r$$

$$A - r \times n.$$

or if we write $B = \begin{bmatrix} | \\ \vdots \\ \vec{b}_i \\ \vdots \\ | \end{bmatrix}$ $A = \begin{bmatrix} \hline \vec{a}_i \hline \end{bmatrix}$

$$T\vec{x} = \sum_{i=1}^r \vec{b}_i (\vec{a}_i \vec{x}).$$

Either decomposition proves $\text{rank}(T) \leq r$ as a matrix. In fact, $\text{rank}(T)$ is the smallest possible rank of a bilinear algorithm, so

$$\text{linear complexity of } T = \text{matrix rank of } T.$$

Now we move to bilinear algorithms.
Start with a tensor $T \in \mathbb{C}^{m \times n \times p}$.

$$T(\vec{x}, \vec{y})_k = \sum_{i=1}^m \sum_{j=1}^n T_{i,j,k} x_i y_j, \quad k \leq p.$$

T defines a set of p bilinear forms.

Example: Matrix mult

$$(AB)_{i,j} = \sum_k A_{i,k} B_{k,j}.$$

Definition A bilinear algorithm for $T(\vec{x}, \vec{y})$ is a set of linear forms

$$a_1(\vec{x}) = \sum_{i=1}^m a_{1,i} x_i$$

$\vdots$

$$a_r(\vec{x}) = \sum_i a_{r,i} x_i$$

$$b_1(\vec{y}) = \sum_{j=1}^n b_{1,j} y_j$$

$\vdots$

$$b_r(\vec{y}) = \sum_j b_{r,j} y_j$$

so that each $T(\vec{x}, \vec{y})_k$ is in $\text{span}(a_1(\vec{x})b_1(\vec{y}), \dots, a_r(\vec{x})b_r(\vec{y}))$

The rank of the algorithm is r .

Example 2×2 matrix mult

Textbook: rank 8

Strassen: rank 7.

Why bilinear algorithms?

They never use commutativity,
so we can do divide &
conquer like Strassen.

Def $T \in \mathbb{C}^{m \times n \times p}$ a tensor.

$\text{rank}(T)$ = smallest r st Frank- r
bilinear algo for T .

Lemma Let $\langle n \rangle \in \mathbb{C}^{n^2 \times n^2 \times n^2}$ be the
 $n \times n$ matrix multiplication tensor.

If $\text{rank}(\langle n \rangle) \leq r$, then

$$\omega \leq \log_n(r).$$

Sketch Divide & conquer using $n \times n$
block matrices.

Next time : more tools to bound
ranks of tensors.