附录 A 应用: 矩阵乘积的快速算法

A.1 矩阵乘积的普通方法

设 $A=[a_{ij}], B=[b_{ij}]\in\mathbb{R}^{n\times n}$, 则 $C=[c_{ij}]=AB$, 其中

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

易知, 总运算量为: n^3（乘法）+ n^3（加法）$=2n^3$.

算法 A.1. 矩阵乘积: IJK 顺序

```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c_ij = c_ij + a_ik b_kj
        end for
    end for
end for
```

以上是计算矩阵乘积的常规实现方式, 但并不是速度最快的实现方式. 基于矩阵乘积在实际应用中的重要性, 人们在不断寻求计算矩阵乘积的快速方法. 对于矩阵乘法的加速, 有如下两种策略:

- 基于算法的优化, 如著名的 Strassen 算法.
- 基于硬件的优化

A.2 基于硬件的加速方法

对于硬件优化, 有循环展开、基于内存布局等的优化技巧. 此外还可以多线程 (比如 C++11 有 `<thread>` 库) 并发执行. 这里探讨一个最简单技巧: 调换循环顺序. 比如将原来的 IJK 顺序改成 IKJ 顺序:

算法 A.2. 矩阵乘积: IKJ 顺序

```
for i = 1 to n do
    for k = 1 to n do
        for j = 1 to n do
            c_ij = c_ij + a_ik b_kj
        end for
```

```
    end for
end for
```

例 A.1 在个人电脑 (CPU i9-10900K, 内存 128G) 上用 C 语言测试. (`test_dgemm.c`)

- `n=1024`, `gcc` 编译加 `-O2` 选项, IKJ 顺序大概比 IJK 顺序快 2.5 倍. 如果加 `-O3` 选项, 则快 3.5 倍.
- `n=2048`, `gcc` 编译加 `-O2` 选项, IKJ 顺序大概比 IJK 顺序快 6 倍. 如果加 `-O3` 选项, 则快 8 倍.

以下分析摘自网络 (https://zhuanlan.zhihu.com/p/146250334), 仅供参考.

下面分析为什么这两种顺序的计算效果差这么多.

首先从矩阵的存储方式说起. 一般而言, 矩阵有两种存储方式: 一维数组和二维数组. 对于矩阵乘法而言, 一维数组显然比二维数组好得多 (下文的分析也能看出这一点). 但是如果用矩阵类实现其他功能的话, 可能其他功能用二维数组更方便一些. 所以下面对于这两种实现方式都加以分析。

造成矩阵乘法慢的原因, 除了算法本身的复杂度以外, 还有内存访问的不连续, 这会导致 cache 命中率不高. 所以为了加速, 就要尽可能使内存访问连续, 即不要跳来跳去. 我们定义一个概念: **跳跃数**, 来衡量访问的不连续程度.

对于最普通的实现方式 (IJK 顺序), 它是依次计算 C 中的每个元素, 对应的计算公式是

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}.$$

因此计算 c_{ij}, 需要将 A 的第 i 行与 B 的第 j 列依次相乘相加. 按假设, 矩阵是按行存储的, 所以 A 在第 i 行中不断向右移动时, 内存访问是连续的. 但 B 在第 j 列不断向下移动时, 内存访问是不连续的. 计算完 c_{ij} 时, B 已经间断地访问了 n 次, 而 A 只间断 1 次 (这一次就是算完后跳转到本行的开头), 故总共是 $n+1$ 次. 这样, 计算完 C 中所有 n^2 个元素, 跳转了 n^3+n^2 次. 但刚才没有统计 C 的跳转次数, 加上以后是 n^3+n^2+n. (注意, 在计算完 C 中每行的最后一个元素时, A 是从相应行末尾转到下一行开头. 如果使用一维数组实现的话, 这是连续地访问, 要减掉这 n 次. 同时, C 也没有跳转次数了, 还要减掉 n 次. 因此对于一维数组, 跳转数是 n^3+n^2-n 次)

而如果以 IKJ 顺序实现, 则对 C 一行一行计算, 即

$$\tilde{c}_i = a_{i1}\tilde{b}_1 + a_{i2}\tilde{b}_2 + \cdots + a_{in}\tilde{b}_n, \quad i = 1, 2, \ldots, n,$$

其中 $\tilde{c}_i$ 和 $\tilde{b}_i$ 分别表示 C 和 B 的第 i 行. 当计算 C 的第 i 行时, 先计算 $a_{i1}\tilde{b}_1$, 即访问 a_{i1} 和 B 的第 i 行 (不间断往右移). 然后计算 $a_{i2}\tilde{b}_2$, 此时 A 往右挪一个元素 (不间断), B 则跳转到下一行 (如果二维数组则间断一次, 一维数组不间断). 依次类推, 算完 C 的第 i 行后, 恰好按顺序将 B 遍历一遍, 间断了 n 次 (一维数组是 1 次), 且恰好从左往右遍历了 A 的第 i 行, 间断了 1 次 (一维数组没有间断), 加起来是 $n+1$ 次 (一维数组是 1 次). 故计算 C 的所有 n 行后, 跳转了 n^2+n 次 (一维数组是 n 次). 刚才没有算 C 的跳转, 算上后跳跃数是 $2n^2+n$ 次 (一维数组是 n^2 次).

由此可见:

- IKJ 顺序的跳转数渐进地少于 IJK 顺序的跳转数;
- 一维数组比二维数组好.

跳跃数总结

下面是各个循环顺序的跳跃数列表 (写文章时现算的, 可能会粗心犯错)

- IKJ 顺序: $2n^2 + n$ (二维数组), n^2 (一维数组)
- KIJ 顺序: $3n^2$ (二维数组), $2n^2$ (一维数组)
- IJK 顺序: $n^3 + n^2 + n$ (二维数组), $n^3 + n^2 - n$ (一维数组)
- JIK 顺序: $n^3 + 2n^2$ (二维数组), $n^3 + n^2 + n$ (一维数组)
- KJI 顺序: $2n^3 + n$ (二维数组), $2n^3$ (一维数组)
- JKI 顺序: $2n^3 + n^2$ (二维数组), $2n^3 + n^2$ (一维数组)

因此从跳跃数来看, 不同顺序的执行效率排序为:

$$\text{IKJ} \sim \text{KIJ} > \text{IJK} \sim \text{JIK} > \text{KJI} \sim \text{JKI}.$$

实测速度

测试环境: 双 CPU Xeon 4215R (8 核 16 线程), 32G 内存. 带 `-O2` 编译选项.

N	IKJ	KIJ	IJK	JIK	KJI	JKI
1024	0.550	0.570	3.430	3.610	7.920	7.770
2018	4.360	4.880	25.050	27.560	97.100	99.610

注记: 需要指出的是, 影响矩阵乘积的因素有许多, 除了前面介绍的跳跃数外, 还包括:

- 向量化指令运算, 如 X86 架构上的 AVX 指令集等.
- 循环展开 (loop unrolling), 编译器中加优化选项后, 一般会自动进行循环展开.
- cache blocking (也称 tiling).
- 多线程, 如 C/C++ 多线程, OpenMP.
- … …

关于内存优化可参考

[1] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd edn, Pearson, 2016. 《深入理解计算机系统》, 简称 CSAPP, 龚奕利, 贺莲译.

A.3 Strassen 方法

德国数学家 Strassen 在 1969 年提出了计算矩阵乘积的快速算法, 将运算量降为约 $O(n^{2.81})$. 下面以二阶矩阵为例, 描述 Strassen 方法的实现过程. 设

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

则 $C = AB$ 的每个分量为

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}, \quad c_{12} = a_{11}b_{12} + a_{12}b_{22},$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}, \quad c_{22} = a_{21}b_{12} + a_{22}b_{22}.$$

在 Strassen 算法中, 我们并不直接通过上面的公式来计算 C, 而是先计算下面 7 个量:

$$\begin{aligned}
x_1 &= (a_{11} + a_{22})(b_{11} + b_{22}),\\
x_2 &= (a_{21} + a_{22})b_{11},\\
x_3 &= a_{11}(b_{12} - b_{22}),\\
x_4 &= a_{22}(b_{21} - b_{11}),\\
x_5 &= (a_{11} + a_{12})b_{22},\\
x_6 &= (a_{21} - a_{11})(b_{11} + b_{12}),\\
x_7 &= (a_{12} - a_{22})(b_{21} + b_{22}).
\end{aligned}$$

于是, C 的各元素可以表示为:

$$\begin{aligned}
c_{11} &= x_1 + x_4 - x_5 + x_7,\\
c_{12} &= x_3 + x_5,\\
c_{21} &= x_2 + x_4,\\
c_{22} &= x_1 + x_3 - x_2 + x_6.
\end{aligned}$$

易知, 总共需要做 7 次乘法和 18 次加法.

下面考虑一般情形. 我们采用分而治之的思想, 先将矩阵 A, B 进行 2×2 分块, 即

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

则 $C = AB$ 也可以写成 2×2 分块形式, 即

$$\begin{aligned}
C_{11} &= X_1 + X_4 - X_5 + X_7,\\
C_{12} &= X_3 + X_5,\\
C_{21} &= X_2 + X_4,\\
C_{22} &= X_1 + X_3 - X_2 + X_6,
\end{aligned}$$

其中

$$
\begin{aligned}
X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}),\\
X_2 &= (A_{21} + A_{22})B_{11},\\
X_3 &= A_{11}(B_{12} - B_{22}),\\
X_4 &= A_{22}(B_{21} - B_{11}),\\
X_5 &= (A_{11} + A_{12})B_{22},\\
X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}),\\
X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).
\end{aligned}
$$

需要 7 次子矩阵的乘积和 18 次子矩阵加法. 假定采用普通方法计算子矩阵的乘积, 即需要 $(n/2)^3$ 乘法和 $(n/2)^3$ 次加法, 则采用 Strassen 方法计算 A 和 B 乘积的运算量为

$$7 \times \left((n/2)^3 + (n/2)^3\right) + 18 \times (n/2)^2 = \frac{7}{4}n^3 + \frac{9}{2}n^2.$$

大约是普通矩阵乘积运算量的 $\frac{7}{8}$. 在计算子矩阵的乘积时, 我们仍然可以采用 Strassen 算法. 依此类推, 于是, 由递归思想可知, 则总运算量大约为（只考虑最高次项, 并假定 n 可以不断对分下去）

$$7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807\ldots}$$

课外阅读

记两个 n 阶矩阵乘积的运算量为 $O(n^\alpha)$. 在 Strassen 算法出现之前, 大家一直认为 $\alpha = 3$. 但随着 Strassen 算法的出现, 大家意识到这个量级是可以小于 3 的. 但 α 能不能进一步减少? 如果能的话, 它的极限又是多少? 这些问题引起了众多学者的极大兴趣. 1978 年, Pan 证明了 $\alpha < 2.796$. 一年后, Bini 等又将这个上界改进到 2.78. 在 Pan 和 Bini 等的工作的基础上, Schönhage 于 1981 年证明了 $\alpha < 2.522$. 而上界首次突破 2.5 是由 Coppersmith 和 Winograd 提出来的, 他们证明了 $\alpha < 2.496$. 1990 年, 他们又在 Strassen 算法的基础上提出了一种新的矩阵乘积计算方法, 将运算量级降至著名的 2.376. 这个记录一直持续了近二十年, 直到 2010 年前后, Stothers, Vassilevska-Williams 和 Le Gall 分别将这个上界降到 2.37293, 2.3728642 和 2.3728639. 2021 年, Alman 和 Williams 将上界进一步降到 2.3728596. 虽然有许多学者相信, α 的极限应该是 2, 但至今无法证实.

相关参考文献:

[1] V. Strassen, Gaussian elimination is not optimal, *Numer. Math.*, 13 (1969), 354–356.

[2] V. Y. Pan, Strassen's algorithm is not optimal, In *Proc. FOCS*, 19 (1978), 166–176.

[3] D. Bini, M. Capovani, F. Romani and G. Lotti, $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication, *Inf. Process. Lett.*, 8 (1979), 234–235.

[4] A. Schönhage, Partial and total matrix multiplication, *SIAM J. Comput.*, 10 (1981), 434–455.

[5] D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, In *Proc. SFCS*, 82–90, 1981.

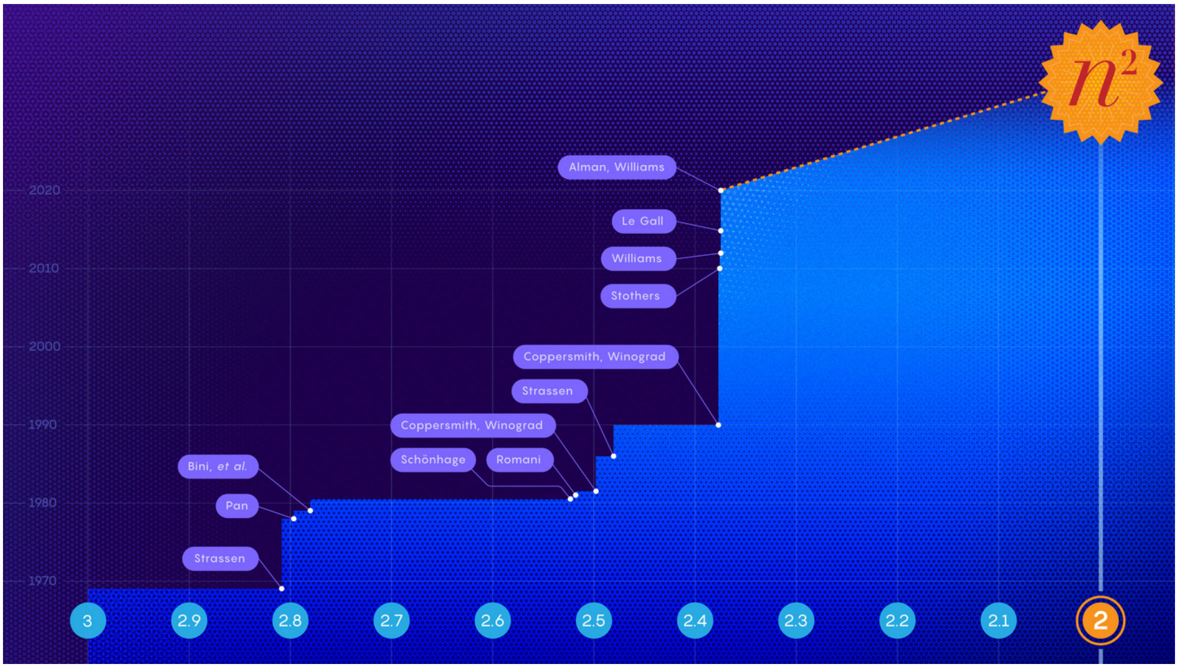
[6] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Computation*, 9 (1990), 251–280.

[7] A. J. Stothers, *On the complexity of matrix multiplication*, PhD thesis, 2010.

[8] V. Vassilevska-Williams, Breaking the Coppersmith–Winograd barrier, In *44th ACM Symposium on Theory of Computing (STOC 2012)*, 2012.

[9] F. Le Gall, Powers of tensors and fast matrix multiplication, *In 39th International Symposium on Symbolic and Algebraic Computation (ISSAC 2014)*, 2014, 296–303. arXiv:1401.7714.

[10] J. Alman and V. V. Williams, A refined laser method and faster matrix multiplication, Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), 2021.



(https://www.quantamagazine.org/mathematicians-inch-closer-to-matrix-multiplication-goal-20210323/)