矩阵乘法优化算法

第一部分：引言

1.1 问题提出

矩阵乘法是科学计算、机器学习、图形处理等领域的核心运算

传统三重循环复杂度O(n³)难以满足大规模计算需求（例：1000×1000矩阵需10^9次运算）

核心矛盾：算法理论复杂度 vs 实际硬件性能瓶颈

1.2 研究意义

提升计算效率可直接影响：

深度学习训练速度（矩阵乘占比>70%）

天气预报模拟精度

金融风险评估实时性

（这个地方可以找几张图片）

第二部分：理论与文献综述

2.1 经典算法发展

1969年Strassen算法：首次突破O(n³)（O(n^2.81)）

1990年CoppersmithWinograd算法（O(n^2.376) 理论突破但工程实现困难

2000年后：并行计算、GPU加速、分布式计算成为主流方向

2.2 关键优化维度

算法层面：降低计算复杂度（如Strassen）

硬件层面：利用缓存局部性（分块优化）、多核并行（多线程）

编程层面：内存访问模式优化、SIMD指令集

2.3 文献支撑

[Strassen algorithm - Wikipedia](https://en.wikipedia.org/wiki/Strassen_algorithm) 首次提出低于 O(n³) 的矩阵乘法算法

[GPU computing performance analysis on matrix multiplication - Huang - 2019 - The Journal of Engineering - Wiley Online Library](https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/joe.2018.9178)分析了 GPU 在不同规模矩阵乘法中的性能表现

[Separating concurrent languages with categories of language embeddings | Proceedings of the twenty-third annual ACM symposium on Theory of Computing](https://dl.acm.org/doi/abs/10.1145/103418.103423)提出 O(n^2.376) 的矩阵乘法算法

（要求中说明要体现出对文献的收集与整理，我大致找了几个，可以酌情看一下）

第三部分：算法实现与代码解析（10分钟）

3.1 基准算法：三重循环

原理：直接利用三重循环按照公式 C[i][j] = ∑ A[i][k] \* B[k][j] 计算，每个元素计算时都逐个遍历所有乘法项。

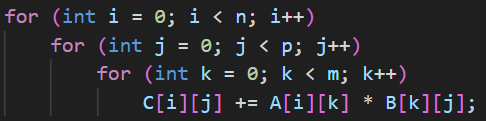
// 代码片段：naiveMultiply

for (int i=0; i<n; i++)

for (int j=0; j<p; j++)

for (int k=0; k<m; k++)

C[i][j] += A[i][k] B[k][j];



优点：

实现简单，易于理解

缺点：

内存访问不连续导致缓存命中率低

无并行化使得单线程性能瓶颈

3.2 Strassen分治算法

原理：利用分治法将矩阵分成 4 个子矩阵，构造 7 个乘积来替代传统的 8 个乘法，从而降低运算复杂度。

// 代码片段

Matrix M1 = strassenRecursive(A11+A22, B11+B22);

Matrix C11 = M1 + M4 M5 + M7;

优点：降低乘法次数

缺点：额外加减操作、内存消耗增加。当矩阵规模较小时，递归深度引入的函数调用、内存分配等开销反而不利于性能。

3.3 循环分块优化

原理：将矩阵划分为多个小块进行处理，利用块数据能够完全或大部分装入 CPU 缓存的特性，降低因频繁访问主内存而产生的延迟，提高缓存局部性。

// 代码片段：optimizedMultiply

for (int i0=0; i0<n; i0+=blockSize)

for (int k0=0; k0<n; k0+=blockSize)

for (int j0=0; j0<n; j0+=blockSize)

// 处理块内计算

优点：通过分块（如32×32）确保数据块完整驻留在L1/L2缓存，减少主存访问延迟。较适合中等规模矩阵，此时缓存效应显著而并行开销尚未主导。

缺点：纯单线程设计，无法发挥现代CPU多核优势。1000×1000测试中落后多线程版本达4.7倍。

3.4 多线程并行

原理：利用 C++11 std::thread，按矩阵行数均分工作，每个线程负责计算部分行的结果。多线程可充分利用多核处理器，分摊计算负担，从而显著加速大规模矩阵乘法。

// 代码片段：parallelMultiply

threads.emplace\_back([&, startRow, endRow]() {

for (int i=startRow; i<endRow; i++)

// 计算指定行范围

});

优点：对于大尺寸矩阵，多线程能将普通三重循环的计算时间大幅降低，且加速比随矩阵规模增大而提升；

缺点：线程启动与同步有一定开销，在小矩阵上可能不划算。

第四部分：实验分析与结果（3分钟）

4.1 测试环境

硬件：12th Gen Intel(R) Core(TM) i5-12500H (十二核 / 十六逻辑处理器)

软件：C++17, GCC (Deepin 12.3.0-17deepin12) 12.3.0, Deepin 25 Community

4.2 性能对比

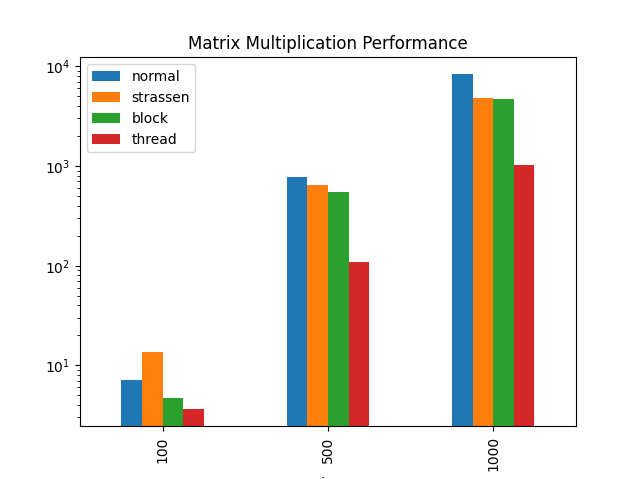
size,normal,strassen,block,thread,same

100,7.18895,13.6618,4.74128,3.6003,1

500,782.974,651.597,551.345,109.786,1

1000,8382.02,4784.79,4739.32,1015.76,1

（这个地方时间充裕的话可以做一个表格）



4.3 关键发现

Strassen算法规模敏感性：Strassen算法在矩阵小于阈值时性能反降，验证了递归算法存在"临界规模"特性

优化维度：多线程在各类规模下均保持显著优势，说明并行化是当前硬件条件下的首选优化方向

scalability收益差异：从500到1000规模时，Strassen的加速比从1.2倍提升到1.75倍，但多线程从7.1倍增至8.25倍，表明不同优化策略的 scalability 存在本质差异

第五部分：综合讨论与扩展

（扩展部分不一定用我给的例子，有更合适的可以替换，同样要配几张图）

5.1 优化组合可能性

分块+多线程：提升缓存利用率与并行度

Strassen+SIMD：向量化加速递归步骤

5.2 工业级优化案例

Intel MKL库：自适应算法选择（AVX512指令集）

NVIDIA cuBLAS：GPU并行+混合精度计算

5.3 未来研究方向

异构计算（CPU+GPU协同）

量子矩阵乘法算法（HHL算法）

第六部分：Q&A