矩阵乘法加速技术的调研与实验

曹弈轩（2300271012）

摘 要：加速矩阵乘法计算在人工智能时代具重要意义．本文定义了矩阵乘法的基准算法，介绍了降低算法理论复杂度、线程级和数据级并行计算等多种加速方式，并实现利用程序访问内存的局部性、Strassen 算法、线程级并行和数据级并行的多种加速计算的程序．实验证明行优先访问矩阵对提高高速缓存命中率的显著作用，多线程和向量指令能有效加速矩阵乘法计算．然后，本文将 4 线程并行、向量指令、行优先访问几种优化方式结合，实验结果证明这几种方法协同作用在大规模矩阵运算上，加速比可达 50 以上．最后，本文实现并评估 GPU 加速矩阵乘法，并拆解分析其主要的性能开销．实验结果表明 GPU 程序能更显著地加速矩阵乘法计算，其开销主要来源于乘法计算．

关键词：矩阵乘法；并行计算；POSIX 线程；单指令多数据（SIMD）；Strassen 算法；高速缓存

矩阵乘法（matrix multiplication）是根据 2 个矩阵得到第 3 个矩阵的二元运算．第 3 个矩阵即前 2 者的矩阵积（matrix product）．给定矩阵$\boldsymbol{A}_{n\times m}$和$\boldsymbol{B}_{m\times p}$，其矩阵积为$\boldsymbol{C}_{n\times p} = \boldsymbol{A}_{n\times m}\cdot\boldsymbol{B}_{m\times p}$，其中

$$c_{ij} = \sum_{r=1}^{n} a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

矩阵乘法是线性代数的基础工具，在应用数学、物理学、工程学等领域十分常见．与此同时，矩阵乘法是深度学习任务中最基本的运算之一．根据矩阵乘法定义，3 层循环的时间复杂度为$O(nmp)$．为简化算法复杂度分析，我们有时将其视为方阵乘法，即复杂度为$O(n^3)$，在大规模矩阵乘法中性能较低．因此，加速矩阵乘法计算对提高人工智能的模型训练和应用部署的效率有重要意义．

加速矩阵乘法主要有以下 2 种思路：1）设计更低复杂度的串行算法．1969 年，Strassen[1]利用矩阵分块的思想将时间复杂度降到$O(n^{2.81})$，从 1990 年代至今，已有多项工作[2-8]将矩阵乘法算法复杂度降低到$O(n^{2.37})$附近．2）利用并行技术加速，按照粒度不同分为指令级并行、数据级并行和线程级并行．指令级并行主要指现代流水化处理器上指令间的并行，在软件层面，编译器使用循环展开、分支预测、硬件推测等技术，减少流水化处理器中的停顿．数据级并行依赖单指令多数据（single instruction multiple data，SIMD）体系结构，其中有向量体系结构、图形处理单元（graphics processing unit，GPU）等多种变体．线程级并行指将矩阵乘法问题划分成若干个独立的子任务，然后利用多个线程求解这些子任务．实际上，降低串行算法的复杂度和并行加速并不矛盾，例如，Strassen 算法[1]也可以使用并行加速．此外，高速缓存的局部性也是重要的考虑，实验证明，在大规模的矩阵乘法中，矩阵的行优先访问比列优先访问的高速缓存命中率更高，速度更快．

除通用的矩阵乘法计算外，近年来也有面向稀疏矩阵乘法计算[9,10]和大规模矩阵乘法近似计算[11]的工作，但是本文仅讨论通用的矩阵乘法计算方法，因此不详细展开研究面向特定场景的矩阵乘法加速技术．

本文面向《高级计算机体系结构》实验要求，探索利用 CPU 的特性设计程序访存的局部性原理优化、利用 Strassen 算法降低复杂度，并设计数据级并行、线程级并行的矩阵乘法程序，通过实验评估其效率．然后，通过实验验证多线程、数据级并行和利用程序访存的局部性原理等多种加速方式的协同作用．最后，使用 CUDA 框架设计 GPU 程序并在实验中拆解分析 GPU 加速矩阵乘法计算中的主要性能开销的来源．

1 基准算法

我们依照矩阵乘法的定义，确定基准算法如算法 1 所示，时间复杂度为$O(n^3)$．矩阵$\boldsymbol{A}_{n\times m}$和$\boldsymbol{C}_{n\times p}$按行优先顺序访问，矩阵$\boldsymbol{B}_{m\times p}$按列优先顺序访问．

算法 1：基准算法

输入：矩阵$\boldsymbol{A}_{n\times m}$和$\boldsymbol{B}_{m\times p}$

输出：矩阵$\boldsymbol{C}_{n\times p}$

```
for i = 0 to n − 1 do
    for j = 0 to p − 1 do
        c_ij = 0
        for k = 0 to m − 1 do
            c_ij += a_ik · b_kj
        end
    end
end
```

2 串行算法优化

加速矩阵乘法的串行算法，主要有 2 种思路：1）利用高速缓存的局部性原理优化；2）降低矩阵乘法算法的理论复杂度.

2.1 利用高速缓存的局部性原理优化

访问局部性指在程序访问内存时，倾向与访问位置靠近的值．时间局部性指被访问过 1 次的存储器位置在未来可能会被多次访问；空间局部性指 1 个内存位置被访问，那么将来他附近的位置也可能会被访问．因为局部性原理，硬件设计中加入了高速缓存，从而提高访存速度．然而，在基准算法中，矩阵$\boldsymbol{B}_{m\times p}$按照列优先顺序访问，不能很好地利用程序的访问局部性．优化算法如算法 2 所示，我们先将矩阵$\boldsymbol{B}_{m\times p}$转置，然后执行与基准算法相似的过程.

算法 2：利用高速缓存的局部性原理优化基准算法

输入：矩阵$\boldsymbol{A}_{n\times m}$和$\boldsymbol{B}_{m\times p}$

输出：矩阵$\boldsymbol{C}_{n\times p}$

```
T = B^T
for i = 0 to n − 1 do
    for j = 0 to p − 1 do
        c_ij = 0
        for k = 0 to m − 1 do
            c_ij += a_ik · t_jk
        end
    end
end
```

2.2 降低矩阵乘法算法的理论复杂度

我们假定参与乘法的矩阵为规模为n的方阵，其复杂度为$O(n^{\omega})$．显然在基准算法中，$\omega = 3$．实际上，从 1969 年至今，陆续有更优的矩阵乘法算法被提出用以减小ω的取值，如表 1 所示.

需要注意的是，有的算法尽管降低了理论复杂度，但是未能很好利用计算机体系结构的特性，甚至由于其过于复杂的设计引起的巨大常数，导致在一般的矩阵乘法计算中表现得并不好，因此在真实世界应用中落地难度极大，被称作“银河式算法”[1]．我们重点介绍并评估 Strassen 算法．Strassen 算法是 1 种分治算法，它基于矩阵分块思想，减少乘法次数，将矩阵乘法复杂度降低到$O(n^{\log_2 7})$，其中$\log_2 7 \approx 2.81$，小于基准算法的$O(n^3)$．Strassen 算法能计算形如$\boldsymbol{C}_{n\times n} = \boldsymbol{A}_{n\times n} \cdot \boldsymbol{B}_{n\times n}$且$n$为 2 的整数次幂的矩阵乘法．如果矩阵不为方阵或方阵的边长不为 2 的整数次幂，我们可以填充 0 转化成 Strassen 算法所需要的方阵再求解，因此 Strassen 算法是通用的．Strassen 算法的过程如下.

表 1 矩阵乘法优化算法的年份、复杂度和提出者

年份	ω	提出者
1969	2.8074	Strassen V[1]
1978	2.796	Pan V Y[12]
1979	2.780	Bini D、Capovani M、Romani F[13]
1981	2.522	Schönhage A[14]
1981	2.496	Coppersmith D、Winograd S[15]
1982	2.517	Romani F[16]
1986	2.479	Strassen V[17]
1990	2.3755	Coppersmith D、Winograd S[2]
2010	2.3737	Stothers A J[3]
2012	2.3729	Williams V V[4]
2014	2.3728639	Le Gall F[5]
2020	2.3728596	Alman J、Williams V V[6]
2022	2.371866	Duan R、Wu H、Zhou R[7]
2023	2.371552	Williams V V、Xu Y、Xu Z 等[8]

[1] 银河式算法是一类对于极大规模数据表现优异的复杂算法，在常规问题中往往无法展现出优势，甚至效率低于一般的解决方案，而当数据规模足够大时，效率提升才能体现．这里的“足够大”脱离了现实需求，以至于这类算法从未在实践中发挥作用．Strassen 算法在实践中被使用过，严格说不属于银河式算法．Coppersmith-Winograd 算法在其基础上利用复杂的群论进一步改进，使ω降低到 2.373，而这是“银河式”的．摘自维基百科词条“银河式算法（Galactic algorithm）”，链接：https://en.wikipedia.org/wiki/Galactic_algorithm，最后访问时间：2023-11-12.

首先，矩阵$\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$分成相等大小的方块矩阵

$$\boldsymbol{A}=\begin{bmatrix}\boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2}\\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2}\end{bmatrix}$$

$$\boldsymbol{B}=\begin{bmatrix}\boldsymbol{B}_{1,1} & \boldsymbol{B}_{1,2}\\ \boldsymbol{B}_{2,1} & \boldsymbol{B}_{2,2}\end{bmatrix}$$

$$\boldsymbol{C}=\begin{bmatrix}\boldsymbol{C}_{1,1} & \boldsymbol{C}_{1,2}\\ \boldsymbol{C}_{2,1} & \boldsymbol{C}_{2,2}\end{bmatrix}$$

显然有

$$\boldsymbol{C}_{1,1}=\boldsymbol{A}_{1,1}\boldsymbol{B}_{1,1}+\boldsymbol{A}_{1,2}\boldsymbol{B}_{2,1}$$
$$\boldsymbol{C}_{1,2}=\boldsymbol{A}_{1,1}\boldsymbol{B}_{1,2}+\boldsymbol{A}_{1,2}\boldsymbol{B}_{2,2}$$
$$\boldsymbol{C}_{2,1}=\boldsymbol{A}_{2,1}\boldsymbol{B}_{1,1}+\boldsymbol{A}_{2,2}\boldsymbol{B}_{2,1}$$
$$\boldsymbol{C}_{2,2}=\boldsymbol{A}_{2,1}\boldsymbol{B}_{1,2}+\boldsymbol{A}_{2,2}\boldsymbol{B}_{2,2}$$

我们引入新矩阵$\boldsymbol{M}_i(i=1,2,\cdots,7)$，其中

$$\boldsymbol{M}_1=(\boldsymbol{A}_{1,1}+\boldsymbol{A}_{2,2})(\boldsymbol{B}_{1,1}+\boldsymbol{B}_{2,2})$$
$$\boldsymbol{M}_2=(\boldsymbol{A}_{2,1}+\boldsymbol{A}_{2,2})\boldsymbol{B}_{1,1}$$
$$\boldsymbol{M}_3=\boldsymbol{A}_{1,1}(\boldsymbol{B}_{1,2}-\boldsymbol{B}_{2,2})$$
$$\boldsymbol{M}_4=\boldsymbol{A}_{2,2}(\boldsymbol{B}_{2,1}-\boldsymbol{B}_{1,1})$$
$$\boldsymbol{M}_5=(\boldsymbol{A}_{1,1}+\boldsymbol{A}_{1,2})\boldsymbol{B}_{2,2}$$
$$\boldsymbol{M}_6=(\boldsymbol{A}_{2,1}-\boldsymbol{A}_{1,1})(\boldsymbol{B}_{1,1}+\boldsymbol{B}_{1,2})$$
$$\boldsymbol{M}_7=(\boldsymbol{A}_{1,2}-\boldsymbol{A}_{2,2})(\boldsymbol{B}_{2,1}+\boldsymbol{B}_{2,2})$$

利用引入的 7 个新矩阵可得$\boldsymbol{C}$的 4 个子矩阵

$$\boldsymbol{C}_{1,1}=\boldsymbol{M}_1+\boldsymbol{M}_4-\boldsymbol{M}_5+\boldsymbol{M}_7$$
$$\boldsymbol{C}_{1,2}=\boldsymbol{M}_3+\boldsymbol{M}_5$$
$$\boldsymbol{C}_{2,1}=\boldsymbol{M}_2+\boldsymbol{M}_4$$
$$\boldsymbol{C}_{2,2}=\boldsymbol{M}_1-\boldsymbol{M}_2+\boldsymbol{M}_3+\boldsymbol{M}_6$$

子问题的计算也使用 Strassen 算法. 设 Strassen 算法在数据规模为n的耗时为$T(n)$，有

$$T(n)=\begin{cases}\Theta(1), & n=1\\ 7T(\frac{n}{2})+\Theta(n^2), & n\geq 2\end{cases}$$

由主定理（master theorem）得$T(n)=\Theta(n^{\log_2 7})$. 需要注意的是，该算法虽然降低了矩阵乘法的复杂度，但是频繁的递归调用、频繁地动态分配和释放临时内存显然引入了不小的开销. 在我们设计的实验中，该算法表现并不出色.

3 并行加速设计

3.1 线程级并行程序设计

3.1.1 基于基准算法和行优先优化的线程级并行

基准算法和利用高速缓存的局部性原理优化算法的本质是用矩阵乘法定义，计算矩阵$\boldsymbol{C}_{n\times p}$中的各个元素$c_{ij}$，各个元素的计算彼此独立. 因此可以并行化改造. 假设我们使用k个线程加速基准算法，则可以将矩阵均等划分为k个区域，每个区域由 1 个线程负责，每个线程都使用基准算法或利用高速缓存的局部性原理优化算法.

我们使用 POSIX 线程中 pthread_create()创建k个线程，分别处理k个子任务. 我们必须等待所有的子任务完成后才能认为矩阵乘法运算完成，因此要使用 pthread_join()实现线程间的同步.

3.1.2 基于 Strassen 算法的线程级并行

基于矩阵分块的 Strassen 算法同样是可并行的. 以 4 线程并行为例，Strassen 算法将矩阵$\boldsymbol{C}$分块为$\boldsymbol{C_{1,1}}$、$\boldsymbol{C_{1,2}}$、$\boldsymbol{C_{2,1}}$和$\boldsymbol{C_{2,2}}$四个部分，每个部分作为 1 个独立子问题，分别由 4 个线程单独使用 Strassen 算法求解. 4 个线程求解完成后，再将$\boldsymbol{C_{1,1}}$、$\boldsymbol{C_{1,2}}$、$\boldsymbol{C_{2,1}}$和$\boldsymbol{C_{2,2}}$拼接回矩阵$\boldsymbol{C}$.

但是，在小样本上使用 Strassen 算法并不能获得显著收益，而在大样本下测试的实验中，Strassen 算法的表现显著慢于基准算法. 因此，我们认为对 Strassen 算法做多线程并行加速并不是应该被优先考虑的方向，在本文中，我们没有测试 Strassen 算法和多线程加速的协同优化.

3.2 数据级并行程序设计

3.2.1 基于英特尔 AVX 向量指令集的加速

英特尔公司研发的全新的指令扩展集（Intel Advanced Vector Extensions，AVX）已经广泛应用于服务器和个人台式机硬件中. 在本文使用的机器上有 16 个 ymm 寄存器，每个寄存器 256 位，在 64 位操作系统上可以同时装载 $16\times 256\div 32=128$ 个单精度浮点数（C 语言中的 float 类型）.

以矩阵计算 $\boldsymbol{C}=\boldsymbol{AB}$为例，在向量计算中，我们应该充分利用向量乘法和加法的特性. 为了充分利用 ymm 寄存器，我们可以将 ymm 寄存器分为 2 组，ymm0~ymm7 装载矩阵$\boldsymbol{A}$的数据，ymm8~ymm15 装载矩阵$\boldsymbol{B}$的数据，1 次迭代可以计算 8 个向量乘法. 然后，对 8 个向量乘法的结果可以用向量加法指令并行计算加法. 具体细节见算法 3.

AVX256 支持 1 次计算 8 个单精度浮点数的向量计算，因此其理想加速比为 8. 但是，数据在内存和向量寄存器之间的相互搬移引入开销，不可能达到理想加速比. 使用 AVX 指令依然需要先访问矩阵的内存，将元素依次移到 ymm 寄存器，因此，行优先还是列优先的问题依然存在，行优先访问矩阵

内存依然能提高高速缓存的命中率．由于编程接口__builtin_ia32_loadups256()用于将 256 位连续内存加载到寄存器上，列优先访问显然不适合做这种操作，因此我们直接设计向量指令和行优先访问协同优化的算法，即先转置矩阵$\boldsymbol{B}$后计算．我们约定计算矩阵的元素大小为 32 字节，loadups()表示加载 8 个 32 位数据，mulps()表示将 256 位向量相乘，addps()表示将 256 位向量相加，storeups()表示将向量中的 8 个 32 位数据重新存到数组中．

算法 3 首先将矩阵$\boldsymbol{B}$转置，得到矩阵$\boldsymbol{T}$，然后进行和算法 2 无本质区别的第 4、5 和 7 行的 3 重循环．在第 7 行开始的循环中，我们尽可能充分利用有限的 ymm 寄存器，在同一轮循环中尽可能多地进行向量乘法和加法指令，来完成$m \times m$的向量点乘，这些过程可以拆分为如下 3 阶段：1）第 8~23 行，每轮从矩阵$\boldsymbol{A}$和$\boldsymbol{T}$中各载入 64 个数据用向量指令相乘再相加．该阶段能充分利用所有的 ymm 寄存器．2）第 24~37 行，每轮从矩阵$\boldsymbol{A}$和$\boldsymbol{T}$中各载入 32 个数据计算，该阶段有一半寄存器未能用上．3）第 38~40 行，将向量点乘中最后不到 32 个数据的“漏网之鱼”，用普通的方法依次相乘再相加．

算法 3：向量指令和行优先访问的协同优化

输入：矩阵$\boldsymbol{A}_{n\times m}$和$\boldsymbol{B}_{m\times p}$

输出：矩阵$\boldsymbol{C}_{n\times p}$

1: $\boldsymbol{T} = \boldsymbol{B}^T$
2: $col\_reduced = m - m\%64$
3: $col\_reduced\_32 = m - m\%32$
4: **for** $i = 0$ **to** $n - 1$ **do**
5: **for** $j = 0$ **to** $p - 1$ **do**
6: $c_{ij} = 0$
7: **for** $k = 0,64,\cdots, col\_reduced - 64$ **do**
8: **for** $rid = 0$ **to 7 do**
9: $\text{ymm}_{rid} = loadups(\&a_{i,k})$
10: $\text{ymm}_{rid} = loadups(\&t_{j,k})$
11: **end**
12: **for** $rid = 0$ **to 7 do**
13: $\text{ymm}_{rid} = mulps(\text{ymm}_{rid}, \text{ymm}_{8+rid})$
14: **end**
15: **for** $rid = 0,2,4,6$ **do**
16: $\text{ymm}_{rid} = addps(\text{ymm}_{rid}, \text{ymm}_{8+rid})$
17: **end**
18: $\text{ymm}_0 = addps(\text{ymm}_0, \text{ymm}_2)$
19: $\text{ymm}_4 = addps(\text{ymm}_4, \text{ymm}_6)$
20: $\text{ymm}_0 = addps(\text{ymm}_0, \text{ymm}_4)$
21: $storeups(scratchpad, ymm0)$
22: c_{ij} += $scratchpad.sum()$
23: **end**
24: **for** $l = col\_reduced, col\_reduced + 32, \cdots, col\_reduced\_32 - 32$ **do**
25: **for** $rid = 0$ **to** 3 **do**
26: $\text{ymm}_{rid} = loadups(\&a_{i,l+8\cdot rid})$
27: $\text{ymm}_{rid+8} = loadups(\&t_{j,l+8\cdot rid})$
28: **end**
29: **for** $rid = 0$ **to** 3 **do**
30: $\text{ymm}_{rid} = mulps(\text{ymm}_{rid}, \text{ymm}_{rid+8})$
31: **end**
32: $\text{ymm}_0 = addps(\text{ymm}_0, \text{ymm}_1)$
33: $\text{ymm}_2 = addps(\text{ymm}_2, \text{ymm}_3)$
34: $\text{ymm}_0 = addps(\text{ymm}_0, \text{ymm}_2)$
35: $storeups(scratchpad, \text{ymm}_0)$
36: c_{ij} += $scratchpad.sum()$
37: **end**
38: **for** $l = col\_reduced\_32$ **to** $m - 1$ **do**
39: c_{ij} += $a_{i,l} \cdot t_{j,l}$
40: **end**
41: **end**
42:**end**

由于每个处理器核心都提供了向量指令的支撑硬件，因此 AVX 向量指令技术可以与多核多线程技术协同加速矩阵乘法计算．

3.2.2 基于 GPU 的加速

GPU 最早由英伟达公司于 1999 年提出．传统的 CPU 上每个核心通常只能处理 1 个任务，而 GPU 拥有强大的数据级细粒度并行能力，适合处理大规模的矩阵运算和向量计算．一般而言，使用英伟达显卡加速时，需要用到通用并行计算框架 CUDA．

GPU 能加速执行矩阵乘法计算，是因为由于矩阵$\boldsymbol{C}$中每个元素的计算均相互独立，因此在并行度映射中，我们能够让每个 CUDA 线程对应矩阵$\boldsymbol{C}$中 1 个元素的计算．每个 CUDA 线程的执行流为：从矩阵$\boldsymbol{A}$中读取 1 个行向量，从矩阵$\boldsymbol{B}$中读取 1 个列向量，对这两个向量做点积运算，最后将结果写回矩阵$\boldsymbol{C}$．在我们的测试中，简单起见，仅考虑方阵的计算，即$n = m = p = N$，核心代码如图 1 所示．

```
#define N 10000
#define BLOCK 64
float a[N * N], b[N * N], c[N * N];
__device__ float sum(float *cache, int id)
{
    int i = blockDim.x / 2;
    while (i != 0)
    {
        if (id < i)
        {
            cache[id] += cache[id + i];
        }
        __syncthreads();
        i /= 2;
    }
    return cache[0];
}
__global__ void matrix_dot(float *a, float *b, float *c)
{
    int i = blockIdx.x;
    int j = blockIdx.y;
    __shared__ float cache[BLOCK];
    int t = threadIdx.x;
    if (t < N)
        cache[t] = a[i * N + t] * b[t * N + j];
    else
        cache[t] = 0;
    __syncthreads();
    sum(cache, t);
    __syncthreads();
    c[i * N + j] = cache[0];
}
__host__ void dot(float *a, float *b, float *c)
{
    float *a_cuda, *b_cuda, *c_cuda;
    cudaMalloc((void **)&a_cuda, N * N * sizeof(float));
    cudaMalloc((void **)&b_cuda, N * N * sizeof(float));
    cudaMalloc((void **)&c_cuda, N * N * sizeof(float));
    cudaMemcpy(a_cuda, a, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(b_cuda, b, N * N * sizeof(float), cudaMemcpyHostToDevice);
    dim3 matrix(N, N);
    matrix_dot<<<matrix, BLOCK>>>(a_cuda, b_cuda, c_cuda);
    cudaMemcpy(c, c_cuda, N * N * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(a_cuda);
    cudaFree(b_cuda);
    cudaFree(c_cuda);
}
```

图 1 利用 GPU 加速矩阵乘法的核心代码

4 实验与评估

4.1 实验环境和实现方法

本文使用 Windows 10 台式工作站上的 Ubuntu 22.04 子系统（WSL2），有 8 核英特尔 i7-9700 CPU、64GB 物理内存和显存为 4GB 的英伟达 T400 显卡. 尽管一些编程语言提供了大量第三方开源库（如 Python 的 numpy、pytorch 等）用于矩阵乘法计算，但是为了贴近体系结构底层，减少无关因素的干扰，我们使用 C 语言实现基准算法和 Strassen 算法、行优先访存优化、4 线程优化、向量指令优化 4 种基本的优化方法，然后结合其中独立的优化方向，探索不同因素的协同作用，找出最佳的优化组合.

对于所有的 CPU 程序，我们使用 gcc 11.4.0 编译，为排除编译优化的干扰，所有的程序的编译优化等级为“-O0”；为排除自动向量化的干扰，编译时使用“-fno-tree-vectorize”选项关闭编译器的自动向量化. 我们使用 avx 指令实现和评估 SIMD 的加速，使用 gcc 的“-mavx”选项完成编译. 此外，我们使用 POSIX 线程库实现 C 语言的多线程加速.

对于 GPU 程序，我们使用 CUDA 框架编写矩阵乘法算法，并使用 nvcc 编译. 实验证明 GPU 程序的性能远高于 CPU 程序，我们尝试拆解分析 GPU 程序的主要性能瓶颈.

实验中矩阵元素数据类型一律为单精度浮点数，每组实验在随机生成的$n \times n$的方阵上，测试$n = 500,1000,2000,5000$ 时的算法程序运行时间. 对于串行程序，我们使用函数 clock()采集算法执行前后的时间戳，获取算法执行前后的时间差. 对于并行程序，clock()采集到时间为各线程的时间之和，由于系统调度和子任务划分不完全均匀，简单将 clock()采集到的时间差除以线程数量并不能准确测算出用户感知的时间. 因此，我们改用 POSIX 库提供的接口 clock_gettime()获得时间戳来更准确地评估并发程序计算矩阵乘法的用时.

4.2 CPU 程序的实验结果与分析

CPU 程序包括所有串行程序和多线程程序的结果，如表 2 所示. 基准算法为$O(n^3)$算法，我们尝试用$T(n) = an^3$作为理论用时，以$n = 500$ 的实测值为基准，求得$a = 4.125 \times 10^{-9}$. 如图 2，基准算法在$n = 2000$ 时，算法运行的理论值和实际值相差不大，但是到了$n = 5000$ 时，理论值和实际值偏差数倍之多. 这是因为当矩阵规模增大到一定程度后，列优先访问内存时，高速缓存的命中率急剧降低，引入了额外的开销.

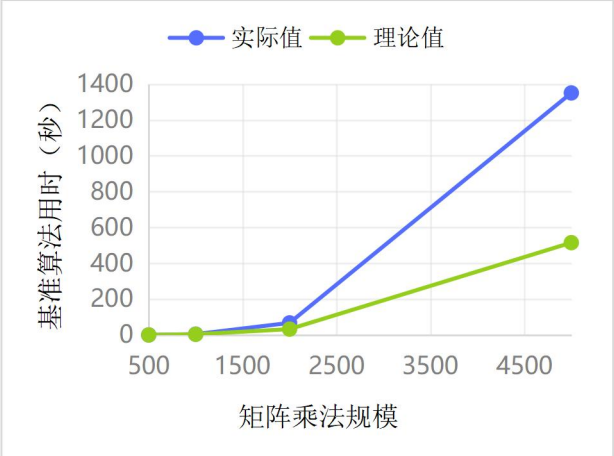

图 2 基准算法的理论用时和实际用时对比

Strassen 算法的理论复杂度为 $O(n^{\log_2 7})$，优于基准算法. 然而从实验上看，在$n < 1000$ 的矩阵运算中，Strassen 算法的表现并不优于基准算法；尽管在$n = 2000$ 时 Strassen 算法的运行速度略优于基准算法，然而随着矩阵规模继续增大，Strassen 算法高速缓存命中率低、递归调用开销大的缺点暴露出来，其性能开始显著低于基准算法，仅为基准算法的 37%. 尽管 Strassen 算法也可以做多线程改造，但是它和行优先访问优化不能协同使用，性能也不如行优先访问的矩阵乘法算法，因此我们没有尝试用多线程并行计算加速 Strassen 算法.

我们定义加速比为优化前程序运行时间和优化后的比值. 矩阵乘法算法及其协同优化策略组合的加速比如图 3 所示. 行优先优化利用程序访问内存的局部性原理，先转置第 2 个矩阵，避免了第 2 个矩阵必须按列访问导致高速缓存命中率降低的问题，在矩阵规模较大时显著提升了高速缓存的命中率，在$n = 5000$ 时加速到基准算法的 3.42 倍. 尽管矩阵转置引入了 $O(n^2)$复杂度的开销，但是实验证明影响很小. 4 线程优化使用 4 个线程并行计算，运行时间减少到基准算法的大约四分之一. 向量指令优化利用英特尔 AVX 指令的硬件特性，可以有显著的加速效果. 这些手段都有明显的效果，并且可以协同使用. 我们结合行优化、向量指令优化、多线程优化 3 种方式，将矩阵计算加速了 20.09 到 54.33 倍不等. 随着矩阵规模增大，加速效果更加显著.

表 2 计算 2 个$n \times n$单精度浮点数方阵乘法用时（单位：秒）

矩阵乘法算法及其协同优化策略组合	方阵规模n			
	500	1000	2000	5000
基准算法	0.515625	4.984375	67.906250	1351.109375
Strassen 算法	**0.968750**	**6.859375**	56.046875	**3588.828125**
行优先优化	0.406250	3.171875	25.718750	394.734375
4 线程优化	0.138613	1.255158	18.288986	343.199325
行优先+4 线程优化	0.125602	1.015911	8.190333	119.686579
行优先+向量指令优化	0.093750	0.671875	5.859375	91.515625
行优先+向量指令+4 线程优化	0.025664	0.185946	1.669275	24.866638

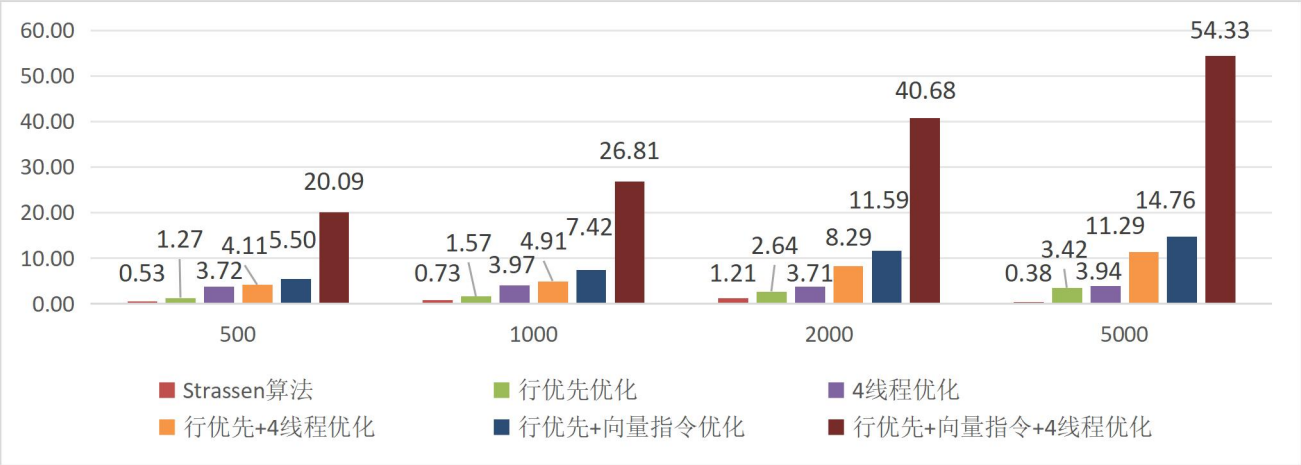

图 3 矩阵乘法算法及其协同优化策略组合的加速比（基准算法为 1）

4.3 GPU 程序的实验结果与分析

GPU 程序的加速比如表 3 所示，随着n逐渐增大，GPU 加速矩阵乘法计算的效果会更加显著；当n达到一定规模的时候，GPU 加速的矩阵乘法速度将高于 CPU 程序 2 个数量级. 因此，GPU 程序在矩阵乘法计算任务中显然比 CPU 程序表现更好，无需再对两者间的性能差异作过多讨论.

表 3 GPU 加速矩阵乘法的加速比

n	500	1000	2000	5000
用时/秒	0.135595	0.221536	0.401435	0.927146
加速比	3.80	22.50	**169.16**	**1457.28**

一般而言，使用 GPU 加速矩阵乘法需要 3 个步骤：首先，分配显存，并将参与计算的向量从内存中加载到显存中. 然后，在 CUDA 线程中计算向量乘法. 最后，将计算结果存储到内存中，并释放显存，避免内存泄漏. 为了分析各个环节的用时，我们将 CUDA 线程的工作拆解为如下 3 部分：1）乘法计算；2）数据搬移；3）显存分配和释放. 为统计$n = 1000,5000,10000$ 时每个部分的耗时，我们首先运行完整的 CUDA 程序，统计完整用时；然后依此去除 1）、2）和 3）的代码，观察用时的变化情况，就可以看出被去除掉的环节的用时，结果如图 4 所示.

通过图 4 的数据，我们得出如下结论：第一，乘法计算是运行时的主要开销. 第二，数据搬移的开销随矩阵规模增大而增大，总体而言在整个 GPU 程序运行时长中占比不大. 第三，显存分配和释放的管理代价虽小，但不容忽视. 我们通过 CUDA 接口 cudaMalloc()和 cudaFree()管理显存，3 组实验中显存管理的代价差别不大，说明显存管理是常数级复杂度的操作. 尽管在$n = 1000$ 时显存管理的代价与乘法计算相当，但是随着n的增大，该部分的代价在整个 GPU 的运行时开销中占比极小. 此外，去除显存的分配和释放后，剩下的开销可以忽略不计.

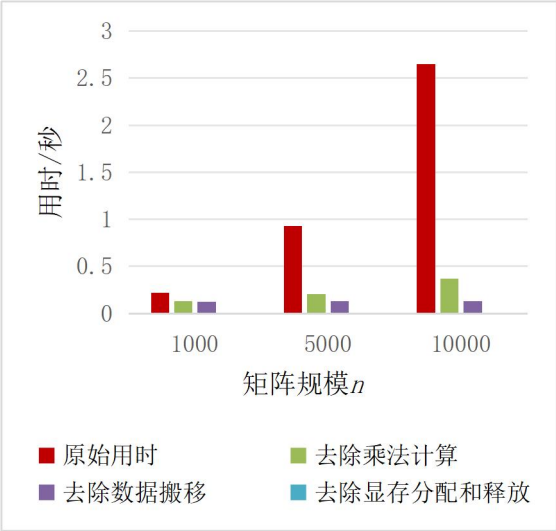

图 4 GPU 程序开销来源拆解：依次去除各个环节

结 语

本文介绍了矩阵乘法并定义基准算法，并梳理利用程序局部性、降低算法理论复杂度、利用线程级并行和数据级并行加速等多种优化方式．我们设计实验观察到如下事实：1）矩阵内存的行优先访问能有效提高高速缓存命中率；2）多线程并行能有效加速矩阵乘法计算，在线程数小于处理器核心的情况下，加速比约为线程数量；3）英特尔的向量指令能在只使用 CPU 的情况下加速矩阵乘法计算，并与多线程加速协同作用；4）尽管 Strassen 算法能够降低矩阵乘法计算的理论复杂度，但是在我们设计的样例场景中表现并不出色．然后，我们结合 4 线程并行、向量指令、行优先访问 3 种优化方式，在大规模矩阵乘法的性能测试中，取得高于基准算法 50 多倍的效果．且随着矩阵规模增大，加速效果随之更加明显．最后，我们在实验中证明 GPU 能更有效地加速矩阵乘法计算，向量乘法计算是 CUDA 程序的主要开销来源．总结而言，本文探索了在 CPU 程序中加速矩阵乘法的可行性，也证明 GPU 在人工智能时代作为基础设施的重要性．

参考文献

[1] Strassen V. Gaussian elimination is not optimal[J]. Numerische mathematik, 1969, 13(4): 354-356.

[2] Coppersmith D, Winograd S. Matrix multiplication via arithmetic progressions[C]//Proceedings of the nineteenth annual ACM symposium on Theory of computing. 1987: 1-6.

[3] Stothers A J. On the complexity of matrix multiplication[J]. 2010.

[4] Williams V V. Multiplying matrices faster than Coppersmith-Winograd[C]//Proceedings of the forty-fourth annual ACM symposium on Theory of computing. 2012: 887-898.

[5] Le Gall F. Algebraic complexity theory and matrix multiplication[C]//ISSAC. 2014: 23.

[6] Alman J, Williams V V. A refined laser method and faster matrix multiplication[C]//Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, 2021: 522-539.

[7] Duan R, Wu H, Zhou R. Faster matrix multiplication via asymmetric hashing[J]. arXiv preprint arXiv:2210.10173, 2022.

[8] Williams V V, Xu Y, Xu Z, et al. New bounds for matrix multiplication: from alpha to omega[J]. arXiv preprint arXiv:2307.07970, 2023.

[9] Srivastava N, Jin H, Liu J, et al. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product[C]//2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020: 766-780.

[10] Zhang Z, Wang H, Han S, et al. Sparch: Efficient architecture for sparse matrix multiplication[C]//2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020: 261-274.

[11] Jahani-Nezhad T, Maddah-Ali M A. Codedsketch: A coding scheme for distributed computation of approximated matrix multiplication[J]. IEEE Transactions on Information Theory, 2021, 67(6): 4185-4196.

[12] Pan V Y. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations[C]//19th Annual Symposium on Foundations of Computer Science (sfcs 1978). IEEE, 1978: 166-176.

[13] Bini D, Capovani M, Romani F, et al. O (n2. 7799) complexity for n× n approximate matrix multiplication[J]. Information processing letters, 1979, 8(5): 234-235.

[14] Schönhage A. Partial and total matrix multiplication[J]. SIAM Journal on Computing, 1981, 10(3): 434-455.

[15] Coppersmith D, Winograd S. On the asymptotic complexity of matrix multiplication[J]. SIAM Journal on Computing, 1982, 11(3): 472-492.

[16] Romani F. Some properties of disjoint sums of tensors related to matrix multiplication[J]. SIAM Journal on Computing, 1982, 11(2): 263-267.

[17] Strassen V. The asymptotic spectrum of tensors and the exponent of matrix multiplication[C]//27th Annual Symposium on Foundations of Computer Science (sfcs 1986). IEEE, 1986: 49-54.