Strassen 算法的并行化实现*

廖钰蕾

(LSEC, 中国科学院, 数学与系统科学研究院, 计算数学与科学工程计算研究所, 北京 100190)

摘 要

矩阵乘法是科学计算中必不可少的部分, 经典的三层循环算法时间复杂度为 $O(N^3)$. 事实上, 还有一些时间复杂度低于 $O(N^3)$ 的工作, 例如 Strassen 算法, 以及 Winograd 变体等, 渐进复杂度为 $O(N^{2.81})$. 本文对 Strassen 算法进行了并行化实现与测试. 在 CPU 环境下的 openMP 实现中, 通过测试选取合适的参数后, 在大规模矩阵下的计算速度相比经典的 $O(N^3)$ 时间复杂度矩阵乘法有明显提高, 在大规模矩阵下的效率能保持在 60% 以上, 扩展性较好. 在 GPU 环境下的 CUDA 实现中, 经典的三层循环方法已经能达到很好的计算效率, 并且随着矩阵规模增大, GPU 的浮点运算性能显著提高.

关键词: Strassen 算法 并行计算 openMP CUDA

1. 问题描述

矩阵乘法是科学计算中必不可少的部分. 常见的加速方法是基于经典的三层循环算法, 通过行划分, 列划分, 快划分等方式并行实现, 其串行算法时间复杂度为 $O(N^3)$, 其中 N 是矩阵行列规模.

事实上, 还有一些时间复杂度低于 $O(N^3)$ 的工作, 例如 Strassen 算法 [3], 以及 Winograd 变体 [4] 等, 这两个算法的渐进复杂度为 $O(N^{2.81})$. 我们关注的即是 Strassen 算法的在 CPU 上的 openMP 实现, 以及在 GPU 上的 CUDA 实现 [2].

Strassen 算法涉及递归调用, 递归层数过深或过浅都会影响算法效率. 我们通过测试选取不同并行平台上的合适参数, 并测试并行算法的加速比, 并行效率, 强可扩展性, 弱可扩展性, 浮点运算性能等并行性能.

2. 数学模型

计算矩阵乘法 $C = AB$, 其中 $A, B, C \in \mathbb{R}^{N\times N}$. 为了简单起见, 我们假设 $N = 2^k$, 其中 k 是非负整数. 其它情形可以通过填充 0 实现.

* 收到.

Strassen 算法的基本思想是按图 1的方式划分矩阵, 再按下述公式计算 [3]:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
M_2 &= (A_{21} + A_{22})B_{11}, \qquad & C_{11} &= M_1 + M_4 - M_5 + M_7, \\
M_3 &= A_{11}(B_{12} - B_{22}), & C_{12} &= M_3 + M_5, \\
M_4 &= A_{22}(B_{21} - B_{11}), & C_{21} &= M_2 + M_4, \\
M_5 &= (A_{11} + A_{12})B_{22}, & C_{22} &= M_1 - M_2 + M_3 + M_6. \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned}
$$

等式中的矩阵乘均为递归调用. 注意到递归调用该方法时, 会产生很多临时的矩阵 M_i, 可以通过在每一层递归调用中使用 2 个临时矩阵解决这一问题 [1].

$$
\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}
$$

图 1 A, B, C 的矩阵划分

3. 数值方法

3.1. 串行算法

Strassen 算法的渐进时间复杂度为 $O(N^{2.81})$, 然而在实际计算中, 临时矩阵的生成与释放, 以及递归过程会耗费大量的时间. 因此我们实际考虑的是带参数的 Strassen 算法 [2]. 具体算法由零层实现, 一层实现, 多层实现三部分组成.

3.1.1. 零层实现

零层实现采用时间复杂度为 $O(N^3)$ 的经典三层嵌套矩阵乘法. 定义核函数 mul 计算 $A \times B$, 见表 1.

表 1 零层实现

步骤	操作	核函数
1	$C = A \times B$	$\mathrm{mul}(A, B, C)$

3.1.2. 一层实现

一层实现采用非递归调用的 Strassen 算法 [3], 其复杂度仍为 $O(N^3)$, 但复杂度隐藏的常数项不同. 具体实现见表 2.

表 2 一层实现

步骤	操作	核函数
1	$[A_{11}, A_{12}; A_{21}, A_{22}] = A$	split$(A_{11}, A_{12}, A_{21}, A_{22}, A)$
2	$[B_{11}, B_{12}; B_{21}, B_{22}] = B$	split$(B_{11}, B_{12}, B_{21}, B_{22}, B)$
3	$T_1 = A_{11} - A_{21}$	sub(A_{11}, A_{21}, T_1)
4	$T_2 = B_{22} - B_{12}$	sub(B_{22}, B_{12}, T_2)
5	$C_{21} = T_1 \times T_2$	mul(T_1, T_2, C_{21})
6	$T_1 = A_{21} + A_{22}$	add(A_{21}, A_{22}, T_1)
7	$T_2 = B_{12} - B_{11}$	sub(B_{12}, B_{11}, T_2)
8	$C_{22} = T_1 \times T_2$	mul(T_1, T_2, C_{22})
9	$T_1 = T_1 - A_{11}$	sub(T_1, A_{11}, T_1)
10	$T_2 = B_{22} - T_2$	sub(B_{22}, T_2, C_{22})
11	$C_{11} = T_1 \times T_2$	mul(T_1, T_2, C_{11})
12	$T_1 = A_{12} - T_1$	sub(A_{12}, T_1, T_1)
13	$C_{12} = T_1 \times B_{22}$	
14	$C_{12} = C_{22} + C_{12}$	mul_add$(T_1, B_{22}, C_{22}, C_{12})$
15	$T_1 = A_{11} \times B_{11}$	
16	$C_{11} = C_{11} + T_1$	
17	$C_{12} = C_{11} + C_{12}$	
18	$C_{11} = C_{11} + C_{21}$	mul_inc_inc_inc$(A_{11}, B_{11}, T_1, C_{21}, C_{11}, C_{12})$
19	$T_2 = T_2 - B_{21}$	sub(T_2, B_{21}, T_2)
20	$C_{21} = A_{22} \times T_2$	
21	$C_{21} = C_{11} - C_{21}$	
22	$C_{22} = C_{11} + C_{22}$	mul_sub_inc$(A_{22}, T_2, C_{11}, C_{21}, C_{22})$
23	$C_{11} = A_{12} \times B_{21}$	
24	$C_{11} = T_1 + C_{11}$	mul_add$(A_{12}, B_{21}, T_1, C_{11})$
25	$C = [C_{11}, C_{12}; C_{21}, C_{22}]$	merge$(C_{11}, C_{12}, C_{21}, C_{22}, C)$

3.1.3. 多层实现

多层实现采用递归调用的 Strassen 算法 [2], 具体过程见算法 1, 算法中调用了零层实现以及一层实现中定义的核函数. 算法中的参数值 τ_1, τ_2 待确定.

算法 1 Strassen 算法

输入: 矩阵 $A, B, C \in \mathbb{R}^{N\times N}$, 矩阵规模 N.

输出: 矩阵 $C \in \mathbb{R}^{N\times N}$.

function STRASSEN(A, B, C, N)
  if $N \le \tau_1$ **then**
    调用零层实现
  else if $N \le \tau_2$ **then**
    调用一层实现
  else
    split($A_{11}, A_{12}, A_{21}, A_{22}, A$); split($B_{11}, B_{12}, B_{21}, B_{22}, B$);
    sub(A_{11}, A_{21}, T_1); sub(B_{22}, B_{12}, T_2); Strassen($T_1, T_2, C_{21}, N/2$);
    add(A_{21}, A_{22}, T_1); sub(B_{12}, B_{11}, T_2); Strassen($T_1, T_2, C_{22}, N/2$);
    sub(T_1, A_{11}, T_1); sub(B_{22}, T_2, C_{22}); Strassen($T_1, T_2, C_{11}, N/2$)
    sub(A_{12}, T_1, T_1);
    Strassen($T_1, B_{22}, C_{12}, N/2$); add(C_{22}, C_{12}, C_{12}); //mul_add
    Strassen($A_{11}, B_{11}, T_1, N/2$); add(C_{11}, T_1, C_{11});
    add(C_{11}, C_{12}, C_{12}); add(C_{11}, C_{21}, C_{11}); //mul_inc_inc_inc
    sub(T_2, B_{21}, T_2);
    Strassen($A_{22}, T_2, C_{21}, N/2$); sub(C_{11}, C_{21}, C_{21});
    add(C_{11}, C_{22}, C_{22}); //mul_sub_inc
    Strassen($A_{12}, B_{21}, C_{11}, N/2$); add($T_1, C_{11}, C_{11}$); //mul_add
    merge($C_{11}, C_{12}, C_{21}, C_{22}, C$);
  end if
  return C
end function

3.2. 并行化策略

并行化策略是将零层实现与一层实现中定义的核函数并行实现.

为了充分利用缓存提高速度, 在 openMP 实现中, 对于 split, merge, add, sub 等操作, 采用行划分分配到各个线程, 对于 mul, mul_add, mul_sub_inc, mul_inc_inc_inc 等含有矩阵乘法运算的操作, 采用列划分分配到各个线程.

在 CUDA 实现中, 我们选定每个线程块的大小为 $m \times m$, 线程块的个数为 $\lceil n/m \rceil \times \lceil n/m \rceil$, 其中 n 为核函数所操作的矩阵规模. 具体到每一个线程只负责计算矩阵中一个元素的值.

4. 数值算例: openMP

4.1. 实验介绍

实验平台科学与工程计算国家重点实验室的 LSSC-IV 四号集群系统, 其超算部分主体包含 408 台新一代 ThinkSystem SD530 模块化刀片, 每个刀片包括 2 颗主频为 2.3GHz 的 Intel Xeon Gold 6140 18 核 Purley 处理器和 192GB 内存, 总共拥有 14688 个处理器核, 理论峰值性能为 1081TFlops, 实测 LINPACK 性能 703TFlops. 操作系统为 Red Hat Enterprise Linux Server 7.3.

我们首先通过实验确定合适的参数值 τ_1, τ_2, 之后在考察并行性能指标有:

- 加速比: 串行执行时间 T_S 与并行执行时间 T_P 的比值 T_S/T_P.
- 效率: 加速比与线程数的比值, 即 $E_P = T_S/(P \times T_P)$. 其中 P 为并行程序执行进程数.
- 强可扩展性: 保持总体计算规模不变, 随着线程个数的增加, 观察加速比与效率的变化.
- 弱可扩展性：保持单个线程的计算规模不变, 随着线程个数的增加，观察加速比与效率的变化.

由于 openMP 是通过创建线程并行, 我们所有的实验都是在一个节点上进行, 不考虑跨节点通信. 为了减小测试误差, 我们连续进行 10 次迭代, 再取平均值作为结果. 矩阵 A, B 的数值随机生成.

4.2. 确定参数值

我们通过测试确定参数值 τ_1, τ_2, 考虑线程数为 16, 不同矩阵规模 N 下的运行时间见表 3.

表 3 线程数 16, 不同矩阵规模的运行时间

$\tau_1, \tau_2 \backslash N$	32	64	128	256	512	1024	2048	4096
∞, ∞	5.70e-5	1.67e-4	3.01e-3	8.84e-3	8.62e-2	9.63e-1	10.38	196.83
$1, \infty$	5.46e-4	1.78e-3	1.07e-3	7.83e-3	5.59e-2	6.24e-1	9.16	80.15
$64, \infty$	5.70e-5	1.67e-4	1.07e-3	7.83e-3	5.59e-2	6.24e-1	9.16	80.15
$64, 64$	5.70e-5	1.67e-4	1.86e-3	1.06e-2	7.63e-2	5.54e-1	3.88	28.77
$64, 512$	5.70e-5	1.67e-4	1.07e-3	7.83e-3	5.59e-2	4.14e-1	2.93	21.38

首先测试 $\tau_1 = \tau_2 = \infty$ 和 $\tau_1 = 1, \tau_2 = \infty$ 的运行时间, 前者在不同矩阵规模 N 均采用零层实现, 后者在不同矩阵规模 N 均采用一层实现, 可以发现 $N \leq 64$ 时零层实现有明显优势, 而 $N \geq 128$ 时一层实现开始显现优势. 因此我们选定 $\tau_1 = 64$.

之后测试 $\tau_2 = \infty$ 和 $\tau_2 = 64$ 的运行时间, 前者在 $N \geq 128$ 时采用一层实现, 后者在 $N \geq 128$ 时采用多层实现, 直至递归调用到 $N = 64$ 时采用零层实现, 可以发现 $N \leq 512$ 时一层实现更有优势, 而 $N \geq 1024$ 时多层实现更有优势. 因此我们选定 $\tau_2 = 512$.

最终选定的 $\tau_1 = 64, \tau_2 = 512$ 的 Strassen 算法, 相比经典的三层循环算法 (即 $\tau_1 = \tau_2 = \infty$) 有明显优势, 在 $N = 2048$ 时, 运行时间相差 3 倍, $N = 4096$ 时, 运行时间相差 9 倍.

4.3. 并行测试结果

4.3.1. 加速比与效率测试

表 4测试了线程数为 16, 参数 $\tau_1 = 64, \tau_2 = 512$ 时, 不同矩阵规模 N 下的并行时间与串行时间, 并计算加速比与效率. 当 $N \geq 128$ 时, 并行效率稳定在 60% 以上, $N = 512$ 时, 并行效率最高.

表 4 线程数 16, 不同矩阵规模的加速比与效率

N	64	128	256	512	1024	2048	4096
并行时间	1.67e-4	1.07e-3	7.83e-3	5.59e-2	4.14e-1	2.93	21.38
串行时间	1.18e-3	8.44e-3	7.55e-2	6.09e-1	4.33	31.72	215.58
加速比	7.07	7.89	9.64	10.89	10.46	10.83	10.08
效率	44.16%	49.30%	60.27%	68.09%	65.37%	67.66%	63.02%

4.3.2. 强可扩展性测试

表 5测试了 $\tau_1 = 64, \tau_2 = 512$ 时, 保持计算规模不变, 矩阵规模 $N = 2048$ 时, 不同线程数下的加速比和效率. 随着线程数成倍增大, 并行效率略有下降, 但速率较缓, 强可扩展性较好, 主要因为是程序中的串行部分占比很小.

表 5 矩阵规模 2048, 强可扩展性

线程数	2	4	8	16	32
并行时间	21.69	10.78	5.62	2.93	1.50
串行时间	31.72	31.72	31.72	31.72	31.72
加速比	1.46	2.94	5.64	10.83	21.15
效率	73.12%	73.56%	70.55%	67.66%	66.08%

4.3.3. 弱可扩展性测试

表 6测试了 $\tau_1 = 64, \tau_2 = 512$ 时, 保持计算规模随线程数成倍增大, 不同线程数下的加速比和效率. 随着线程数成倍增大, 并行效率略有下降, 原因是随着 N 增大, 每个线程的任务量实际也会增大. 由于递归算法的复杂性, 很难保证每个线程的并行任务量完全相等.

表 6 弱可扩展性

N	256	512	1024	2048	4096
线程数	2	4	8	16	32
并行时间	5.45e-2	2.18e-1	7.45e-1	2.93	10.62
串行时间	7.55e-2	6.09e-1	4.33	31.72	215.58
加速比	1.39	2.79	5.81	10.83	20.30
效率	69.27%	69.84%	72.65%	67.66%	63.44%

5. 数值算例: CUDA

5.1. 实验介绍

实验平台是腾讯云的 GN7.2XLARGE32, 其 GPU 为 1 颗 Tesla T4, GPU 显存为 16GB, CPU 为 8 核, 内存为 32GB, 操作系统为 Ubuntu Server 18.04.1 LTS 64 位, GPU 驱动版本为 418.126.02, CUDA 版本为 10.1.105.

我们首先通过实验确定合适的参数值 τ_1, τ_2, 之后在考察并行性能. 上一节讨论的并行性能指标在 GPU 并行中并没有很大的实际意义, 因此我们主要考虑浮点运算性能 GFlops. 定义

$$\text{GFlops} := \frac{N^3}{\text{times} \times 10^9},$$

其中 times 为运行时间.

同样地, 为了减小测试误差, 我们连续进行 10 次迭代, 再取平均值作为结果. 矩阵 A, B 的数值随机生成.

5.2. 确定参数值

我们通过测试确定参数值 τ_1, τ_2, 考虑线程块大小为 64×64, 网格大小为 $\lceil N/64 \rceil \times \lceil N/64 \rceil$, 不同规模 N 下的运行时间见表 7.

首先测试 $\tau_1 = \tau_2 = \infty$ 和 $\tau_1 = 1, \tau_2 = \infty$ 的运行时间, 前者在不同矩阵规模 N 均采用零层实现, 后者在不同矩阵规模 N 均采用一层实现, 可以发现两者的运行时间非常接近, 零层实现略有优势. 上一节中矩阵规模为 4096 时, 采用 16 线程的多层实现将运行

表 7 线程块 64*64, 不同矩阵规模的运行时间

$\tau_1, \tau_2 \backslash N$	1024	2048	4096	8192
∞, ∞	1.48e-2	2.42e-2	6.11e-2	1.99e-1
$1, \infty$	1.59e-2	2.78e-2	6.88e-2	2.09e-1

时间降到了 21.38s, 而 16 线程的零层实现运行时间为 196.83s, 可以看出 GPU 并行的显著优势.

受机器显存空间限制, 我们并没有尝试计算更大规模的矩阵. 由于零层实现的速度已经很快, 之后的实验都是直接在零层实现上进行.

5.3. 并行测试结果

5.3.1. 浮点运算性能

表 8计算了线程块大小为 64×64, 零层实现的浮点运算性能. 随着矩阵规模的增大, 浮点运算性能显著增大, 说明 GPU 的算力并不会对大规模矩阵运算造成瓶颈.

表 8 线程块 64*64, 浮点运算性能

N	1024	2048	4096	8192
times	1.48e-2	2.42e-2	6.11e-2	1.99e-1
GFlops	72.55	354.96	1124.71	2762.59

5.3.2. 线程块规模

表 9测试了矩阵规模为 4096, 线程块规模为 $m \times m$ 时的运行时间, 当 $m \geq 64$ 时, 线程块规模对运行时间几乎没有影响, 因为 GPU 并行时同一网格中的不同线程块是同步并行的.

表 9 矩阵规模 4096, 不同线程块规模的运行时间

m	64	128	256	512	1024
times	6.05e-2	6.02e-2	6.08e-2	6.10e-2	6.00e-2

6. 总结

本文对 Strassen 算法进行了并行化实现与测试.

在 CPU 环境下的 openMP 实现中, 通过测试选取合适的参数后, 在大规模矩阵下的计算速度相比经典的 $O(N^3)$ 时间复杂度矩阵乘法有明显提高, 在大规模矩阵下的效率能保持在 60% 以上, 扩展性较好.

在 GPU 环境下的 CUDA 实现中, 经典的三层循环方法已经能达到很好的计算效率, 并且随着矩阵规模增大, GPU 的浮点运算性能显著提高.

参 考 文 献

[1] Douglas C C, Heroux M, Slishman G, et al. GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm[J]. Journal of Computational Physics, 1994, 110(1): 1-10.

[2] Li J, Ranka S, Sahni S. Strassen's Matrix Multiplication on GPUs[J]. 2011.

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

[4] Winograd S. On multiplication of 2×2 matrices[J]. Linear Algebra and its Applications, 1971, 4(4): 381-388.