中国人民大学

并行与分布式计算 2025 年秋季学期

实验 1: OpenMP 版本矩阵矩阵乘

2025 年 10 月 1 日

姓名：常皓飞　　学号：2023202311

代码与数据均已开源：https://github.com/HiFiChang/GEMM-Optimization-with-OpenMP

GFLOPS/s 从<0.5 提升到>500，不同规模下加速普遍达到 1000 倍以上

1 问题描述

本项目旨在通过实现一个并行的矩阵-矩阵乘法（GEMM）来熟悉 OpenMP 编程。实验提供的基础代码所执行的计算是一种累加形式，在每次迭代中，其计算公式为：

$$C_{new} = (1+\beta)C_{old} + \alpha AB$$

实验要求基于串行版本，使用 OpenMP 并行优化，并探索不同的优化策略以提升程序的性能。

2 方法与实现

对于 N x N 的矩阵，单次迭代的计算量（FLOPs）为：

- C += b*C: 包含 N*N 次乘法和 N*N 次加法，共 2 * N^2 FLOPs。
- a * A * B: 包含 N*N*N 次乘法和 N*N*N 次加法，共 2 * N^3 FLOPs。
- C += aAB: 包含 N*N 次乘法和 N*N 次加法，共 2 * N^2 FLOPs。
- **单次迭代总计 FLOPs** = 2 * N^3 + 4 * N^2

GFLOPS/秒的计算公式为：

$$\text{GFLOPS/s} = \frac{2N^3 + 4N^2}{\text{time} \times 10^9}$$

为了优化性能，我采用了多种递进的策略，从串行版本开始，逐步引入并行化和内存访问优化。

版本 0: 基础串行实现 (v0.cc)

这是未经任何优化的基准版本。它使用三层嵌套循环 (j, i, k) 来计算，这种循环顺序对缓存不友好，因为对矩阵 B 的访问是按列进行的，导致缓存行频繁换入换出。

版本 1: OpenMP 并行化与循环顺序优化 (v1.cc)

1. **OpenMP 并行化**: 使用 **#pragma omp parallel for** 指令对最外层循环进行并行化，将计算任务分配到多个 CPU 核心。
2. **循环顺序交换**: 将循环顺序从 (j, i, k) 调整为 **(i, k, j)**。这样，对 A 的访问是行优先，对 B 的访问也是行优先（在内层循环中），极大地提高了缓存命中率。

版本 2: 分块优化 (v2.cc)

为了进一步减少缓存未命中，我加入了**分块**技术。它将大矩阵划分为 **32x32** 的子矩阵块。通过 (bi, bj, bk) 三层循环遍历这些块，使得参与计算的数据子集能够完全加载到 cache 中，从而最大限度地重用数据并减少对主内存的访问。**并行化在块级别 (bi, bj) 上进行，内部设计对同一块的累加，不能并行，以此保证最终结果的正确性。**

版本 3: 内存对齐 (v3.cc)

SIMD 指令能够一次处理多个数据。为了使 SIMD 发挥最大效能，数据在内存中的起始地址应该是缓存行大小（64 字节）的倍数。此版本使用 **posix_memalign** 函数来分配**内存对齐**的矩阵，确保每一行的起始地址都对齐，帮助编译器生成更高效的 SIMD 代码。

3 实验与结果

3.1 实验环境

- **Platform:** Google Cloud

- **CPU**: Intel(R) Xeon(R) CPU @ 2.80GHz *(GCP 屏蔽了具体的型号)*
- **OS**: Ubuntu 20.04.6 LTS
- **Compiler**: g++ 9.4.40

3.2 理论峰值计算

这里的计算考虑了指令集支持等更复杂的因素。浮点运算的理论峰值可以通过以下公式计算:

$$Peak\ GFLOPS/s = F\_clock \times N\_cores \times IPC\_FMA \times FLOPs\_FMA$$

该 CPU 支持 **AVX-512** 指令集。对于单精度浮点运算，一个 AVX-512 FMA 指令可以处理 16 个浮点数，并完成一次乘法和一次加法，即 **32 FLOPs**。

该 CPU 架构每个核心拥有两个 FMA 单元，因此每个时钟周期每个核心可以执行 **2 个 FMA 指令**。理论峰值

$$2.80\ GHz \times 32\ Cores \times 2\frac{Instructions}{Cycle} \times 32\frac{FLOPs}{Instruction} = \mathbf{5734.4\ GFLOPS/s}$$

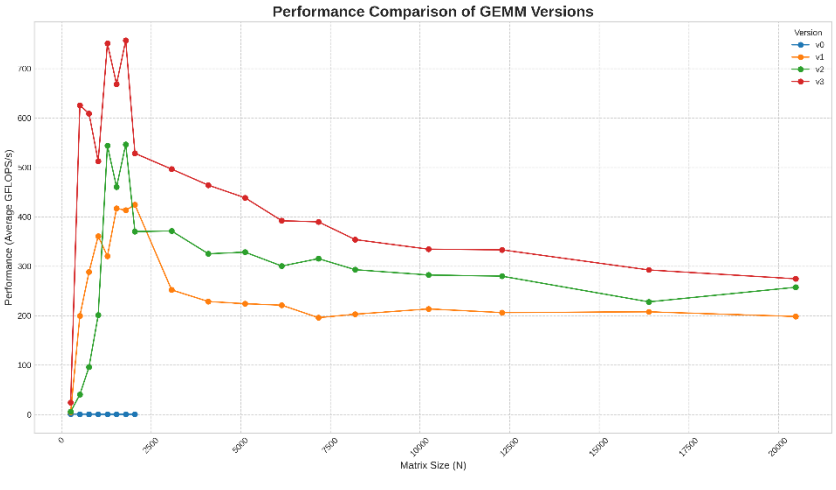
3.3 实验结果

我测试了从 v0 到 v3 四个版本在**从 256 到 20480 的不同矩阵规模下的性能**，结果如下:

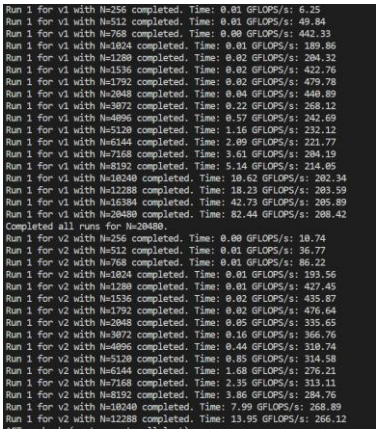
	256	512	768	1024	1280	1536	1792	2048	3072	4096	5120	6144	7168	8192	10240	12288	16384	20480
v0	0.84	0.51	0.61	0.42	0.43	0.37	0.36	0.35										
v1	4.5	199.5	288.4	360.9	320.3	416.9	413.2	424.3	252.0	228.6	224.1	221.1	195.8	202.9	213.5	206.2	207.7	198.2
v2	5.6	40.4	96.1	201.0	544.0	460.1	546.6	370.1	371.2	325.1	328.4	300.4	315.3	292.9	282.2	279.8	227.8	257.4
v3	**24.2**	**625.7**	**608.8**	**512.5**	**750.8**	**668.5**	**757.1**	**528.4**	**496.4**	**464.0**	**438.5**	**392.0**	**389.7**	**353.8**	**334.4**	**333.1**	**292.4**	**274.5**

- 列：矩阵规模，行：v0 为基准，v3 为最终版本。**部分数据耗时太长未测得。**数据单位为GFLOPS/s

我们可以观察到巨大的提升，同规模下**加速比达到 1000-2000 倍。**1792 规模下超过 2000 倍。值得注意的是，几个方案都**在 1k~2k 规模下取得最高性能，此后到 20480 规模过程中逐步下降。**尽管 v3 版本性能最高，**但与理论峰值仍有差距**。我认为原因可能包括:



- **计算理论峰值的时候考虑了 AVX512 等先进的指令集和机制，但实际并未充分利用上。**
- **该 CPU 分 2 个 NUMA 节点，OpenMP 默认线程绑定可能跨 NUMA 不均衡，传输带来损失。**
- **内存带宽限制**: 当计算核心等待数据从主内存加载时，会产生延迟。
- **线程同步开销**: OpenMP 线程的创建、同步和调度会带来一定的开销。
- **其他系统活动**: 尽管使用了专门的服务器，但依旧存在后台任务抢占 CPU 资源。



4 结论与复现

本次实验成功地通过 OpenMP 和多种内存优化策略（循环交换、分块、内存对齐等）显著提升了矩阵乘法的性能。实验结果表明，**并行化和数据局部性优化是实现高性能计算的关键。**最终的优化版本相比于基础串行版本取得了数量级的性能飞跃，验证了这些优化方法的有效性。

- **源代码路径**: code/src/ 目录下包含了 v0.cc 到 v3.cc 的所有实现。
- **编译与运行**:执行 python run.py 脚本，自动编译运行测试。
- **程序运行截图**: 篇幅有限，其中一次测试的部分运行输出如右图。