中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab0-环境设置与串行矩阵乘法

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 3 月 24 日

1 实验目的

- 理解并行程序设计的基本概念与理论。
- 掌握使用并行编程模型实现常见算法的能力。
- 学习评估并行程序性能的指标及其优化方法。

2 实验内容

- 设计并实现以下矩阵乘法版本：
  - 使用 C/C++ 语言实现一个串行矩阵乘法。
  - 比较不同编译选项、实现方式、算法或库对性能的影响：
    * 使用 Python 实现的矩阵乘法。
    * 使用 C/C++ 实现的基本矩阵乘法。
    * 调整循环顺序优化矩阵乘法。
    * 应用编译优化提高性能。
    * 使用循环展开技术优化矩阵乘法。
    * 使用 Intel MKL 库进行矩阵乘法运算。
- 生成随机矩阵 A 和 B，进行矩阵乘法运算得到矩阵 C。
- 衡量各版本的运行时间、加速比、浮点性能等。
- 分析不同实现版本对性能的影响。

3 实验结果

版本	实现描述	运行时间 (s)	相对加速比	绝对加速比	浮点性能	峰值性能百分比
1	Python	186.933	1.0	1.0	0.0014	0.00036%
2	C/C++	37.323	5.01	5.01	0.0573	0.015%
3	调整循环顺序	24.652	1.51	7.59	0.087	0.023%
4	编译优化	7.863	3.16	23.77	0.273	0.071%
5	循环展开	1.852	4.25	101.07	1.159	0.30%
6	Intel MKL	0.088	21.16	2135.23	24.532	6.39%

表 1: 不同版本矩阵乘法的性能对比

4 实验分析

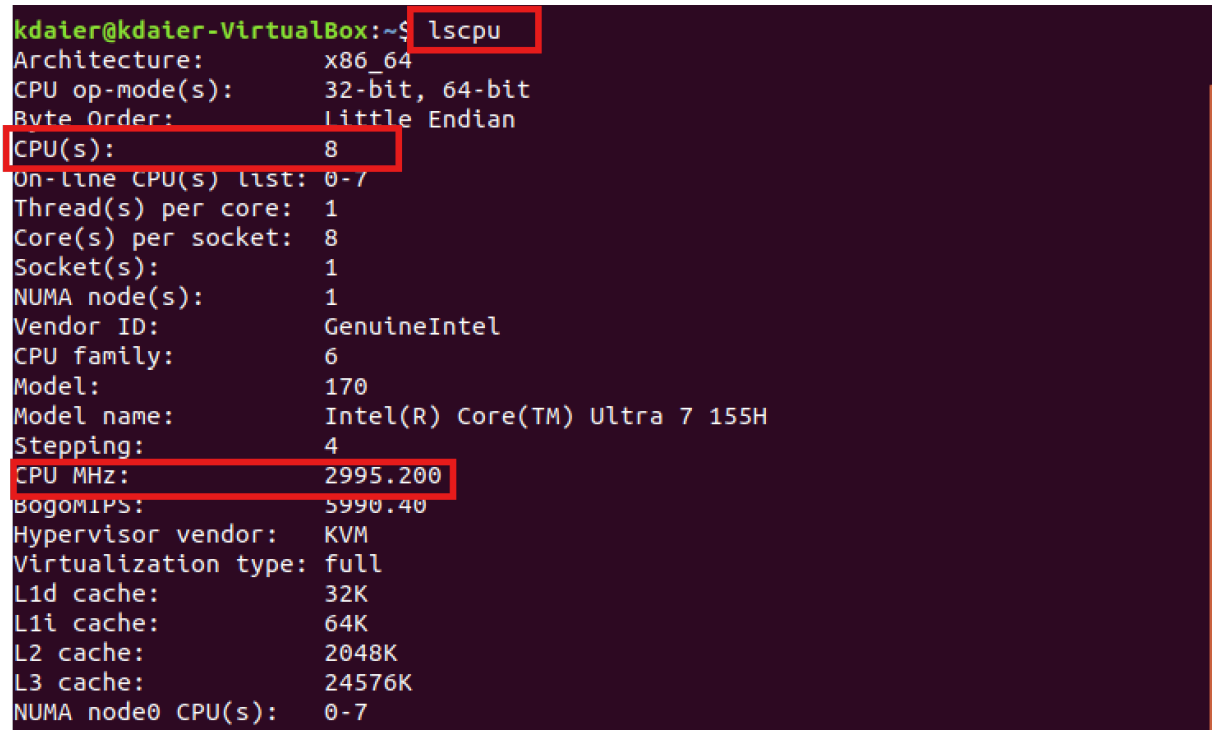

图 1: lscpu

- 我的 CPU 是 8 核，主频 3.0 GHz，每周期每核能执行 16 次浮点运算，因此理论峰值性能可以按下式计算：

  $$\text{理论峰值} = \text{核心数} \times \text{主频 (GHz)} \times \text{FLOP/核心} = 8 \times 3.0 \times 16 = 384 \text{ GFLOPS}$$

  对于浮点运算计数，对于矩阵乘法 $C = A \times B$ ，矩阵维度均为 1024，即 $m = k = n = 1024$，则每个 C 的元素计算需要 1024 次乘法和 1023 次加法，总共 $2 \times 1024 - 1 = 2047$ 次浮点运算。整个矩阵有 1024×1024 个元素，因此总浮点运算次数为

  $$1024^2 \times (2 \times 1024 - 1) = 1024^2 \times 2047 = 2 \times 1024^3 - 1024^2 \approx 2.146435072 \times 10^9 \text{ FLOPs}$$

  如果运行时间为 T 秒，则实际浮点性能为

  $$\text{实际 GFLOPS} = \frac{2.146435072 \times 10^9}{T \times 10^9} = \frac{2.146435072}{T} \text{ GFLOPS}$$

- 在本次实验中，我们实现了六种矩阵乘法版本，其中 Python 版本的运行时间为 186.933 秒，浮点性能仅为 0.0014 GFLOPS，峰值性能百分比仅有 0.00036%，而 C/C++ 列优先版本的运行时间为 24.652 秒，浮点性能达到 0.087 GFLOPS，峰

值性能百分比为 0.023%，调整循环顺序（行优先）的版本运行时间为 37.323 秒，浮点性能为 0.0573 GFLOPS，峰值性能百分比为 0.015%，编译优化版本的运行时间降低到 7.863 秒，浮点性能提升到 0.273 GFLOPS，峰值性能百分比为 0.071%，循环展开版本的运行时间进一步缩短到 1.852 秒，浮点性能达到 1.159 GFLOPS，峰值性能百分比约为 0.30%，而使用 Intel MKL 的版本仅需 0.088 秒即可完成运算，浮点性能高达 24.532 GFLOPS，峰值性能百分比达到 6.39%，这些数据充分说明了解释型语言的固有限制使得 Python 在大规模数值计算中远不能与编译型语言相提并论，而通过 C/C++ 编写的基本实现已经显著提高了性能，进一步采用编译器优化和手动循环展开可以大幅降低运行时间和提高浮点性能，而最终调用专业数值库 Intel MKL 则能够充分利用硬件底层的向量化、多线程以及缓存优化技术，从而实现接近理论峰值的性能水平。

- 从实验结果可以看出，不同版本的矩阵乘法计算性能差异极大，最高版本（Intel MKL）比最低版本（Python）快了 2135 倍，峰值性能利用率从 0.00036% 提升到了 6.39%。这些差异主要受编程语言、编译优化、缓存利用、指令并行、数据访问模式等多个因素影响。Python 是解释型语言，每一行代码都需要由 Python 解释器逐步解析执行，导致大量额外开销，运行速度远低于编译语言。例如 for i in range(n): 在 C 里可以直接执行，但 Python 需要逐步解释 range()，检查 i 的类型，导致性能下降。同时，缺乏优化的内存访问，Python 处理数组通常使用列表，这些列表存储的是对象引用，而不是直接的数值数据，因此访问一个数值需要额外的间接寻址，增加了计算延迟。C/C++ 是编译型语言，代码在运行前被编译成高效的机器指令，避免了解释执行的开销，计算速度提升显著。并且 C/C++ 直接访问数组内存，而不像 Python 需要对象引用，减少了寻址开销。调整循环顺序后，提高了缓存命中率，默认的 for i for j for k 访问顺序会导致缓存失效，因为 C[i][j] 可能会被反复访问，但不一定在缓存中。通过调整循环顺序 for i for k for j，让 C[i][j] 在缓存中存储更长时间，减少了缓存缺失，提升了性能。优化编译器指令，使用 -O3 编译选项后，编译器会进行指令级优化，如：消除冗余计算、更好地利用 CPU 寄存器、自动向量化。减少循环开销，每次循环迭代都涉及分支判断 (if 语句)、索引计算，这些操作都会占用 CPU 计算资源。循环展开将 for 循环展开成多个语句，从而减少了循环控制的额外开销。Intel MKL 使用了高度优化的 BLAS 库，手写 SIMD、并行计算和缓存优化，使其效率远超手写代码。MKL 能自动并行化，充分利用 8 核 CPU 资源，而普通代码需要手动实现 OpenMP 才能做到。MKL 采用了分块矩阵乘法，减少了内存访问延迟，提高了缓存利用率。这些优化减少了 CPU 计算的流水线停滞，提升计算效率。