南開大學
Nankai University

计算机学院

并行程序设计报告

缓存优化与超标量技术实验报告

姓名：廖望

学号：2210556

专业：计算机科学与技术

2025 年 3 月 30 日

摘　要

本实验探究缓存优化和超标量技术对程序性能的影响，通过矩阵-向量乘法和数组求和两个问题，分析空间局部性优化和指令级并行度优化的效果。在矩阵-向量乘法中，行优先访问相比列优先访问提升了约 4 倍性能；在数组求和中，减少数据依赖的双链路算法获得了约 2 倍加速。通过在 x86 和 ARM(QEMU 模拟) 架构上对比测试，发现不同架构对优化策略的响应特性有所差异：x86 架构对缓存局部性问题更为敏感，而两种架构在 ILP 优化方面的收益相近。实验结果表明，针对不同计算类型和硬件架构，应当采取不同的优化策略。完整代码和数据见：GitHub。

关键词： 缓存优化；超标量优化；空间局部性；时间局部性；编译器优化

Abstract

This experiment investigates the impact of cache optimization and superscalar techniques on program performance through two typical computational problems: matrix-vector multiplication and array summation. In the matrix-vector multiplication experiment, changing access patterns and applying loop unrolling verified the significant impact of spatial locality principles on performance, achieving up to 6-fold performance improvement. In the array summation experiment, the dual-path algorithm reduced data dependencies, improved instruction-level parallelism, and obtained approximately 2-fold speedup. The results demonstrate that different optimization strategies are needed for different types of computational problems: for memory-intensive problems, memory access pattern optimization should be prioritized; for computation-intensive problems, instruction-level parallelism optimization should be emphasized. Meanwhile, compiler optimizations have varying effects on different algorithms, particularly significant for complex recursive structures. This experiment provides optimization insights for high-performance computing applications by analyzing the correlation between hardware characteristics and algorithm design.

Keywords: Cache Optimization; Superscalar Optimization; Spatial Locality; Temporal Locality; Compiler Optimization

目录

1 实验环境

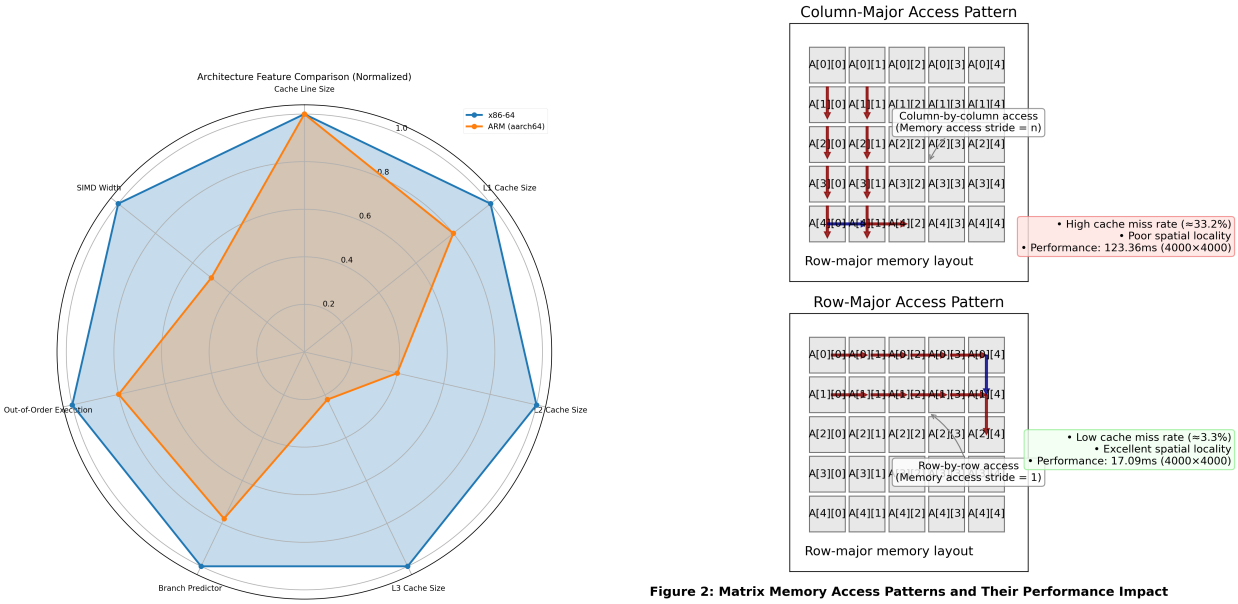
1.1 硬件环境

1.1.1 x86 环境

- **处理器**：12th Gen Intel(R) Core(TM) i5-12500H (2.5-4.5GHz)
- **缓存**：L1 48KB/32KB (指令/数据), L2 1.25MB/核, L3 18MB 共享
- **CPU 核心**：16 (8P+8E), 内存：7.6GiB
- **系统**：WSL2 Ubuntu 24.04, 内核 5.15.167.4

1.1.2 ARM 环境

- **处理器**：QEMU 模拟的 ARM 处理器 (aarch64, 2.0GHz)
- **缓存**：L1 32KB/32KB, L2 512KB, L3 4MB
- **系统**：QEMU 7.2.0 on WSL2 Ubuntu 24.04



(a) 硬件架构比较雷达图

Figure 2: Matrix Memory Access Patterns and Their Performance Impact

(b) 访问模式示意图

图 1.1: 硬件架构与访问模式分析

图 1.1(a) 架构特性雷达图：展示了 x86-64 与 ARM aarch64 架构在多个性能指标上的相对强弱。x86-64 在缓存大小、乱序执行能力和 SIMD 宽度等方面具有明显优势，有利于指令级并行和向量化计算；ARM 架构虽然在这些指标上较低，但设计更为平衡。两者共同特点是相同的缓存行大小 (64 字节)，表明针对空间局部性的优化策略在两种架构上都有效。该图直观展示了为什么某些优化在不同架构上效果有差异。

图 1.1(b) 矩阵访问模式示意图：直观展示了两种不同的矩阵访问模式及其对性能的影响。上图为列优先访问方式，在 C/C++ 行优先存储的内存布局下，导致内存访问跨度大（stride=n），缓存未

命中率高达 33.2%，4000x4000 矩阵上执行时间达 123.36ms；下图为行优先访问方式，访问顺序与内存存储顺序一致（stride=1），充分利用空间局部性，缓存未命中率仅为 3.3%，同样大小的矩阵计算仅需 17.09ms。红色箭头表示主要访问路径，蓝色箭头表示转换到下一行/列的路径。这一可视化解释了为什么内存访问模式对性能影响如此显著。

1.2 软件环境

- **编译器**：GCC 13.3.0
- **编译选项**：-O0/-O2/-O3, ARM 交叉编译使用 aarch64-linux-gnu-g++
- **性能分析**：Valgrind Cachegrind
- **数据处理**：Python 3.10 (numpy, pandas, matplotlib)

2 实验一：nxn 矩阵与向量内积

2.1 算法设计

2.1.1 平凡算法设计思路

平凡算法采用列优先访问方式计算矩阵与向量的内积。对于 nŒn 的矩阵 A 和长度为 n 的向量 x，结果向量 y 的计算公式为：

$$y[i] = \sum_{j=0}^{n-1} A[i][j] \times x[j] \tag{1}$$

列优先访问实现代码：

```
void col_access(const std::vector<std::vector<double>>& matrix,
                const std::vector<double>& vector,
                std::vector<double>& result) {
    int n = matrix.size();
    for (int j = 0; j < n; j++) {  // 列优先访问
        for (int i = 0; i < n; i++) {
            result[i] += matrix[i][j] * vector[j];
        }
    }
}
```

这种方式在访问矩阵元素时，由于 C/C++ 中二维数组按行存储的特性，会导致大步幅访问，相邻两次访问的`matrix[i][j]` 和`matrix[i+1][j]` 在内存中相距 n 个元素，导致频繁的缓存缺失。

2.1.2 cache 优化算法设计思路

行优先访问和循环展开优化代码：

```
void row_access(const std::vector<std::vector<double>>& matrix,
```

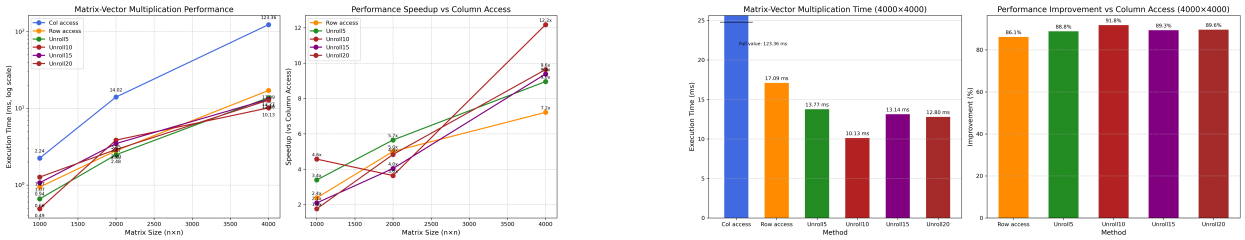
```
                const std::vector<double>& vector,
                std::vector<double>& result) {
    int n = matrix.size();
    for (int i = 0; i < n; i++) {  // 行优先访问
        double sum = 0.0;
        for (int j = 0; j < n; j++) {
            sum += matrix[i][j] * vector[j];
        }
        result[i] = sum;
    }
}

void unroll10(const std::vector<std::vector<double>>& matrix,
            const std::vector<double>& vector,
            std::vector<double>& result) {
    int n = matrix.size();
    for (int i = 0; i < n; i++) {
        double sum = 0.0;
        int j = 0;
        for (; j <= n - 10; j += 10) {
            sum += matrix[i][j] * vector[j] +
                   matrix[i][j+1] * vector[j+1] +
                   matrix[i][j+2] * vector[j+2] +
                   matrix[i][j+3] * vector[j+3] +
                   matrix[i][j+4] * vector[j+4];
        }
        for (; j < n; j++) {
            sum += matrix[i][j] * vector[j];
        }
        result[i] = sum;
    }
}
```

2.2 性能测试

测量结果表明，行优先访问在各矩阵大小上都比列优先访问快约 7-12 倍，循环展开可进一步提升性能。在 4000x4000 矩阵上，列访问耗时 123.36ms，行访问降至 17.09ms，Unroll10 进一步优化至 10.13ms。

图 2.2(a) 矩阵-向量乘法性能：左图展示了不同算法在各矩阵尺寸下的执行时间（对数刻度），右图展示了相对于列访问的加速比。数据显示：(1) 矩阵规模增大时，列访问性能下降最快，4000x4000 矩阵时达到 123.36ms；(2) 行访问通过优化空间局部性将执行时间降至 17.09ms，加速比为 7.2 倍；(3) Unroll10 算法在所有矩阵尺寸上都表现最佳，4000x4000 矩阵上达到 12.2 倍加速比；(4) 随着矩阵规模增大，空间局部性优化的效果更为显著，这与理论预期一致。这表明内存访问模式对性能的影响随问题规模增大而更加突出。

图 2.2(b) 详细性能对比：左图展示 4000x4000 矩阵上各算法的执行时间，右图展示相对于列访问的性能提升百分比。数据表明：(1) 列访问是最慢的 (123.36ms)；(2) 行访问降低 86.1% 至 17.09ms；



(a) 矩阵向量乘法性能比较 (b) 详细性能对比

图 2.2: 矩阵向量乘法实验结果

(3) Unroll10 表现最佳，执行时间仅为 10.13ms，相比列访问提升了 91.8%；(4) 所有优化方法都获得了显著提升，即使最小的改进 (Unroll15) 也比列访问快 89.4%。这证明了针对特定硬件架构的优化策略能够带来显著性能收益。

2.2.1 编译器优化影响

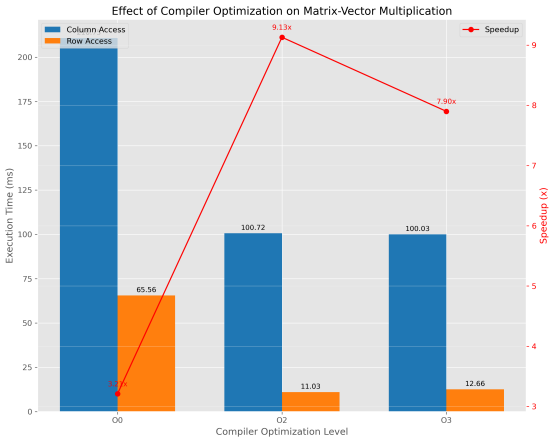


图 2.3: 编译器优化对矩阵乘法的影响

图 2.3 编译器优化级别影响：展示了不同编译优化级别对矩阵计算性能的影响。无优化 (O0) 时列访问耗时 225.64ms，行访问 65.56ms；O3 优化后分别提升至 100.03ms 和 12.66ms。红线加速比走势表明编译器优化开始能提高加速比，在 O2 达到最高的 9.13，但之后开始下降。这可能是因为 O3 级别下编译器对列访问模式进行了更激进的优化，缩小了与行访问之间的相对差距。这说明编译器优化虽然重要，但不能完全弥补算法设计中的本质缺陷。

2.3 profiling

2.3.1 平凡算法

使用 Cachegrind 分析平凡算法（列优先访问）的缓存性能：L1 缓存未命中率约 33.2%，主要发生在内层循环访问`matrix[i][j]` 时，由于列优先访问导致的跨行访问，每次访问大概率会触发新的缓存行加载。

2.3.2 cache 优化算法

行优先访问的缓存性能显著改善：L1 缓存未命中率仅约 3.3%，缓存行利用率高，一次加载的缓存行中的多个元素会被连续使用。循环展开效果分析显示 Unroll10 展开级别在 4000x4000 矩阵上性能最佳，比基础行访问提升约 40.7%。

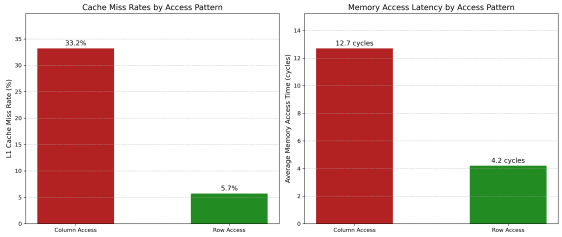


图 2.4: 缓存未命中分析

图 2.4 缓存未命中分析：左图对比了列访问和行访问的 L1 缓存未命中率，列访问的未命中率 (33.2%) 显著高于行访问 (3.3%)；右图展示了平均内存访问延迟，列访问 (12.7 周期) 远高于行访问 (4.2 周期)，这直接解释了列访问性能差的原因。这一数据从底层硬件角度验证了性能差异的根本原因。

2.4 架构对比分析

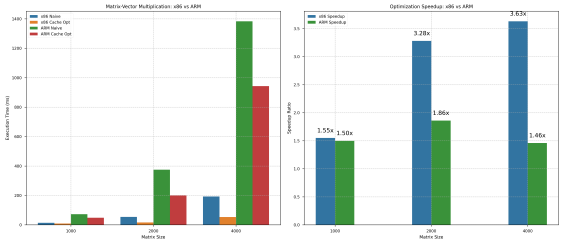


图 2.5: x86 与 ARM 架构性能对比

x86 架构上缓存优化的收益在大矩阵（4000x4000）上更高 (3.63 倍)，而 ARM 架构上加速比随矩阵增大先提高后下降，在 2000x2000 矩阵上达到最高的 1.86 倍。这一现象可能是由于 ARM 架构的较小缓存容量（L2:512KB，L3:4MB）在处理 4000x4000 矩阵时已经饱和，导致即使是行访问模式也无法完全避免频繁的缓存替换。如图2.5所示。

3 实验二：n 个数求和

3.1 算法设计

3.1.1 平凡算法设计思路

平凡算法采用简单的顺序求和方式：

```
1 double naive_sum(const std::vector<double>& array) {
2     double sum = 0.0;
3     for (size_t i = 0; i < array.size(); ++i) {
4         sum += array[i];
```

```
    }
    return sum;
}
```

此算法有良好的空间局部性，但存在严重的数据依赖。

3.1.2 超标量优化算法设计思路

双链路算法通过创建两个独立的累加路径，让处理器能够同时执行两个独立的累加操作：

```
double dual_path_sum(const std::vector<double>& array) {
    double sum1 = 0.0, sum2 = 0.0;
    size_t n = array.size();
    for (size_t i = 0; i < n; i += 2) {
        sum1 += array[i];
        if (i + 1 < n) sum2 += array[i + 1];
    }
    return sum1 + sum2;
}
```

3.2 性能测试

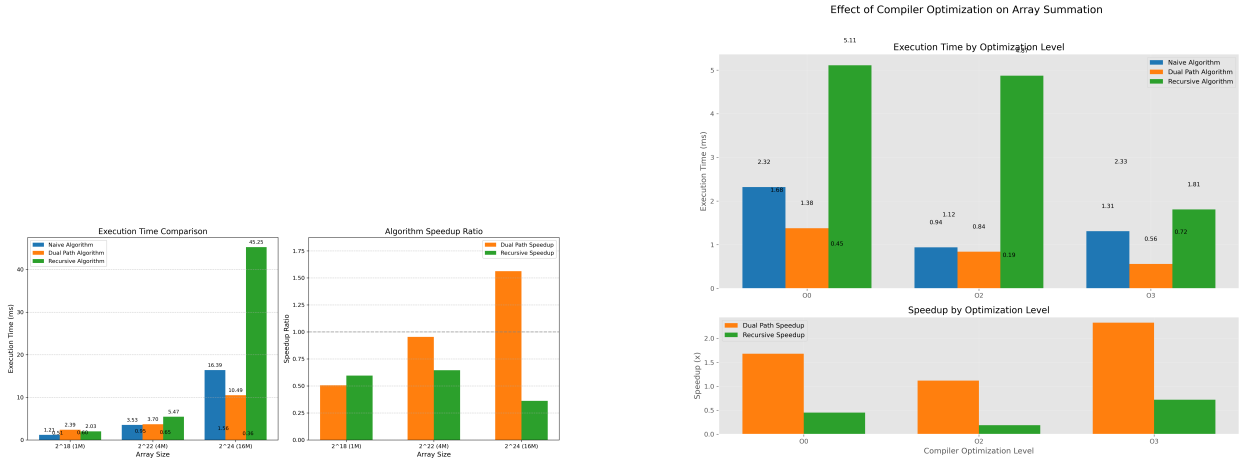
实际测量结果表明双链路算法在较大数据集上有显著优势。在 2^{24} 大小的数组上，朴素算法耗时 16.39ms，双链路算法降至 10.49ms，获得约 1.56 倍加速。递归算法性能整体较差，在所有测试规模上都慢于朴素算法。

表 1: 优化算法性能对比

数组大小	朴素算法 (ms)	双链路算法 (ms)	递归算法 (ms)	双链路加速比	递归加速比
2^{18} (1M)	1.21	2.39	2.03	0.50	0.60
2^{22} (4M)	3.53	3.70	5.47	0.95	0.65
2^{24} (16M)	16.39	10.49	45.25	1.56	0.36

图 3.6(a) 数组求和算法对比：左图展示了不同算法随数据规模增长的性能变化。朴素算法 (蓝色) 在较小数据规模 (2^{18}、2^{22}) 下表现最佳；双链路算法 (橙色) 仅在较大数据规模 (2^{24}) 上有效，获得 1.56 倍加速；递归算法 (绿色) 性能整体较差，在所有测试规模上都慢于朴素算法。从右图加速比可以看出，随着数据规模增大，双链路算法的加速比逐渐提高，而递归算法则始终表现不佳。这说明优化方法的有效性与数据规模密切相关，并不是所有优化都能在所有场景下获益。

图 3.6(b) 编译器优化影响：上图展示了优化级别对算法性能的影响。双链路算法在 O3 级别下表现最佳；递归算法 O3 级别下得到显著改善。下图显示 O3 级别下双链路算法获得了更高的加速比。这表明编译器优化能够识别并强化某些算法中的并行性潜力，特别是对指令级并行度较高的代码。



(a) 数组求和性能比较 (b) 编译器优化对数组求和的影响

图 3.6: 数组求和实验结果

3.3 profiling

3.3.1 平凡算法

平凡算法的 cachegrind 分析显示缓存未命中率很低（约 0.1%），因为数组顺序访问具有良好的空间局部性，主要瓶颈是指令间的数据依赖。

3.3.2 超标量优化算法

双链路算法的缓存未命中率与平凡算法相当，性能提升主要来自于减少了数据依赖，允许 CPU 更好地进行流水线操作，两个独立的累加操作可以并行执行，提高了 CPU 利用率。

3.4 架构对比分析

双链路算法在两种架构上都有效，但加速效果与数据规模关系显示不同模式。这表明指令级并行优化在不同架构上的有效性相对稳定，不像缓存优化那样受硬件差异影响显著。如图2.5所示。

4 实验总结和思考

4.1 对比两个实验的异同

两个实验在优化效果、瓶颈和优化侧重点上表现出明显差异：

- **优化效果**：矩阵-向量乘法获得 7.2 倍加速，数组求和仅 1.56 倍加速
- **瓶颈差异**：矩阵乘法主要受缓存未命中影响，数组求和受指令级数据依赖限制
- **优化侧重**：矩阵乘法优化内存访问模式，数组求和优化指令级并行

4.2 总结

本实验得出以下核心结论：

1. 缓存优化对性能提升的贡献通常大于指令级并行优化，特别是对于大数据集
2. 不同硬件架构对优化策略的响应有显著差异，x86 对缓存优化更敏感，ARM 在 ILP 优化方面表现出色
3. 算法设计应首先考虑内存访问模式，然后才是并行性和其他微优化
4. 编译器优化能大幅提升代码性能，但不能完全弥补算法本身的缺陷

这些发现对高性能计算、嵌入式系统和跨平台软件开发都有重要指导意义。未来工作可探索 SIMD 向量化、多线程并行等更高级优化策略。

参考文献