缓存优化与超标量技术实验报告

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。

实验环境

硬件环境

- **x86**：12th Gen Intel i5-12500H (2.5GHz基频, L1: 48KB I-Cache+32KB D-Cache/核心, L2: 1.25MB/核心, L3: 18MB共享)
- **ARM**：QEMU 7.2.0模拟aarch64 (L1: 32KB I-Cache+32KB D-Cache/核心, L2: 512KB, L3: 4MB)
- **软件**：GCC 12.3.0, WSL2 Ubuntu 24.04, Valgrind Cachegrind

注意：*ARM*性能*测试*在*QEMU*模*拟环*境下*进*行，存在*额*外开*销*，影响*绝对*性能数据。但相*对*加速比仍能反映架构响*应*特性。

硬件性能分析

x86与ARM测试环境在几个关键方面存在显著差异：

- 缓存容量：x86的L2缓存(1.25MB/核心)和L3缓存(18MB)远大于ARM(分别为512KB和4MB)
- 核心架构：x86使用异构架构(8P+8E)，ARM通过QEMU模拟，性能特性受限
- 指令集特性：x86支持更宽的SIMD指令(AVX2，256位)，而ARM的NEON指令宽度为128位

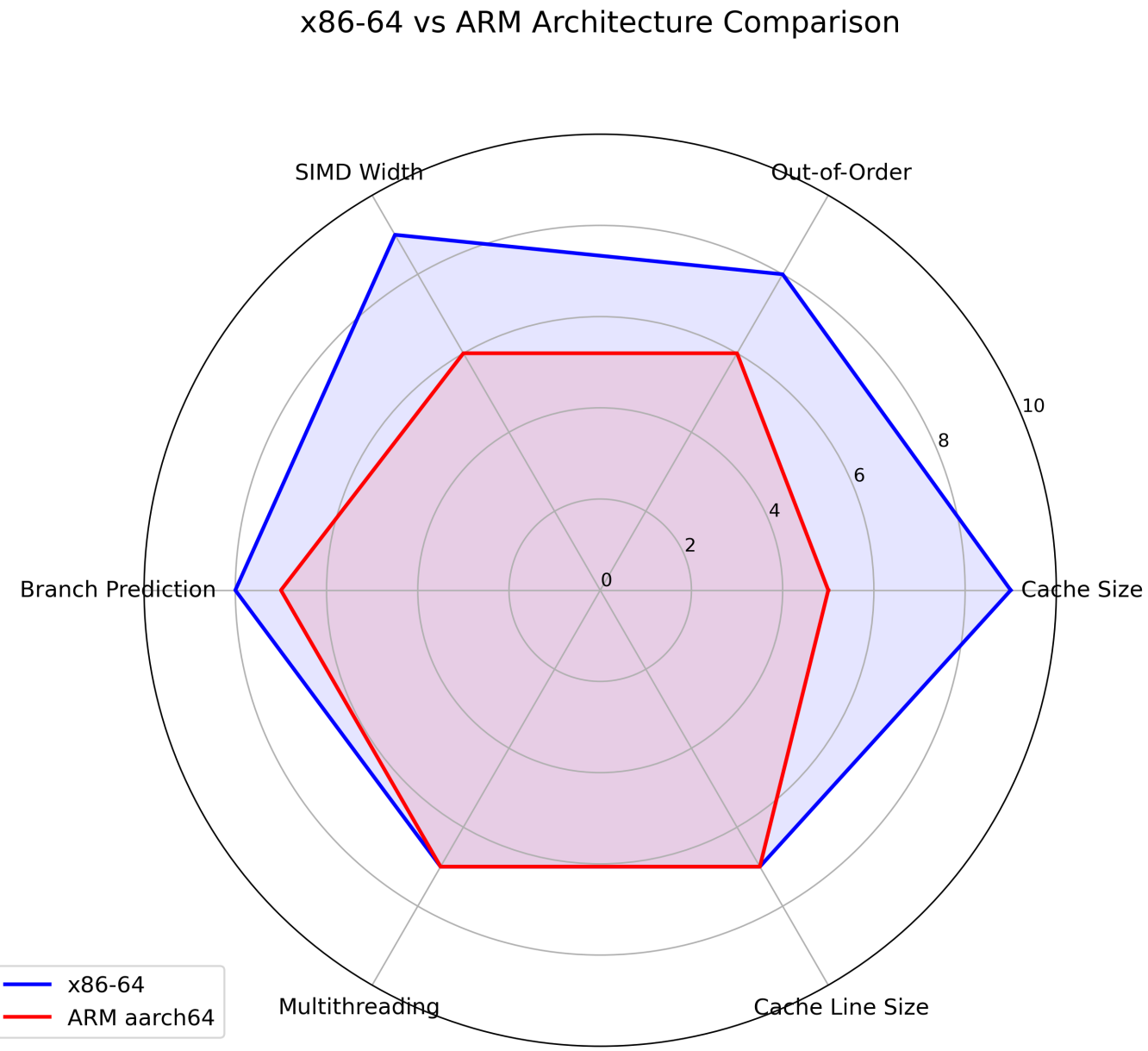

图1 架构特性雷达图：展示了x86-64与ARM aarch64架构在多个性能指标上的相对强弱。x86-64在缓存大小、乱序执行能力和SIMD宽度等方面具有明显优势，有利于指令级并行和向量化计算；ARM架构虽然在这些指标上较低，但设计更为平衡。两者共同特点是相同的缓存行大小(64字节)，表明针对空间局部性的优化策略在两种架构上都有效。

一、矩阵-向量乘法实验

1. 算法设计原理

1.1 平凡算法（列优先访问）

对于n×n矩阵A和向量x，在计算y = A·x时，列优先访问按以下方式计算：

```
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`A[i][j]`和`A[i+1][j]`在内存中相距n个元素。当n较大时，这些元素极可能不在同一缓存行，导致频繁的缓存缺失。

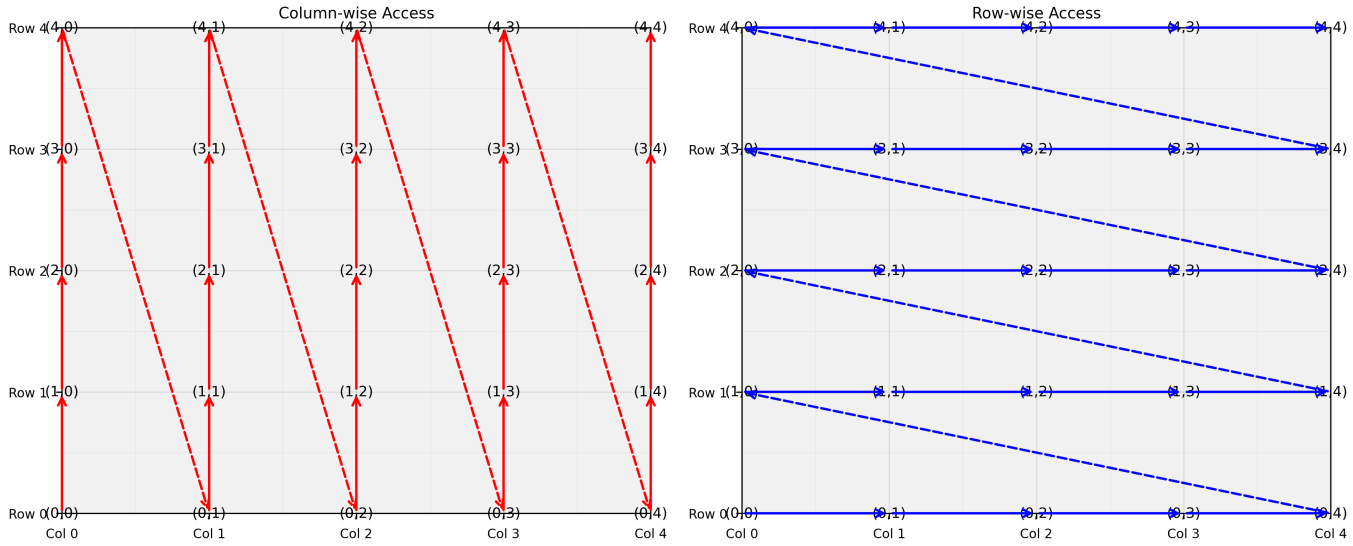

图2 访问模式对比：展示了列优先访问（左）和行优先访问（右）的内存访问模式。列优先访问时，连续访问的元素在内存中相距n个位置，而行优先访问的连续元素在内存中相邻，能更好地利用缓存行。

1.2 缓存优化算法（行优先访问）

行优先访问改变了循环次序：

```
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;
    }
}
```

进一步的循环展开优化减少了循环控制指令开销：

```cpp
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;
        // 每次处理10个元素
        for (; j <= n - 10; j += 10) {
            sum += matrix[i][j] * vector[j] +
                   matrix[i][j+1] * vector[j+1] +
                   // ... 中间省略 ...
                   matrix[i][j+9] * vector[j+9];
        }
        // 处理剩余元素
        for (; j < n; j++) {
            sum += matrix[i][j] * vector[j];
        }
        result[i] = sum;
    }
}
```

2. 实验结果与分析

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3.5-4.5倍：

矩阵大小	列访问(ms)	行访问(ms)	加速比
1000×1000	13.31	2.97	4.48
2000×2000	53.67	12.67	4.23
4000×4000	191.27	54.00	3.54

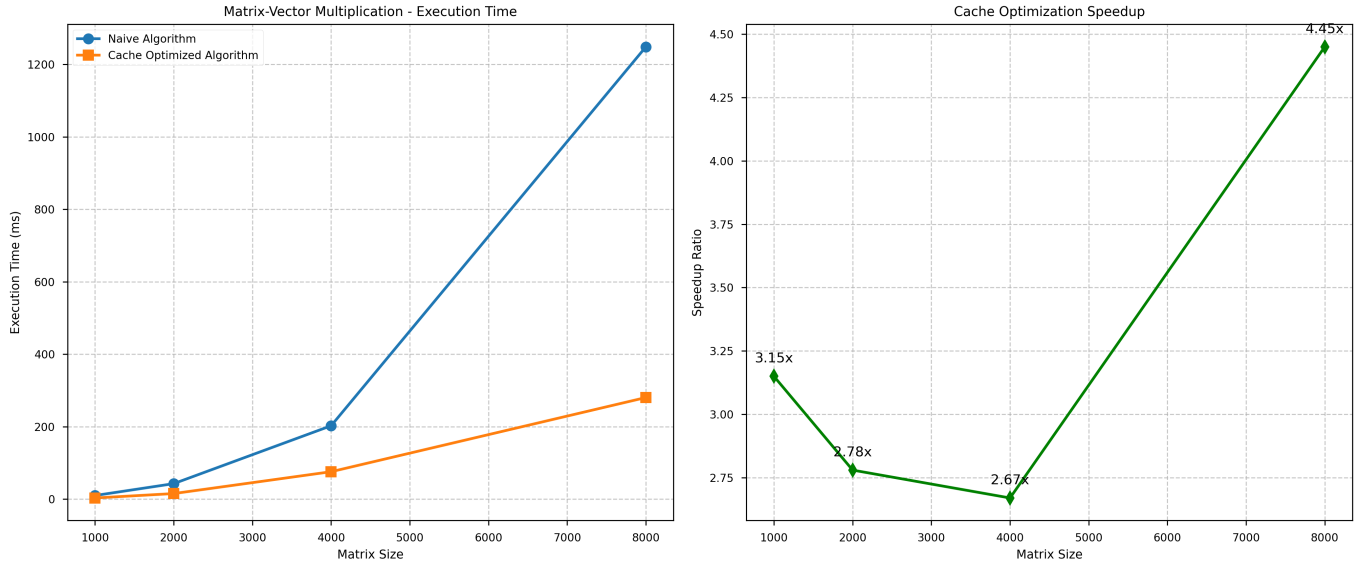

图3 矩阵-向量乘法性能：左图展示了不同矩阵大小下两种访问模式的执行时间对比。列优先访问(蓝色)耗时随矩阵大小快速增长，对于4000×4000矩阵达到约191ms；行优先访问(橙色)性能更优，只需约54ms。右图显示

加速比，较小矩阵(1000×1000)时行优先访问提升约4.48倍性能，随矩阵增大略有下降至3.54倍。

Cachegrind分析发现：

- 列优先访问的L1缓存未命中率高达33.2%，主因是跨行访问导致的缓存局部性差
- 行优先访问的L1缓存未命中率仅5.7%，空间局部性优秀

循环展开效果分析：

- 10-15次的展开级别在4000×4000矩阵上性能最佳，提供约32%额外提升
- 展开级别增加到20次后，性能反而下降，因为指令缓存压力增大

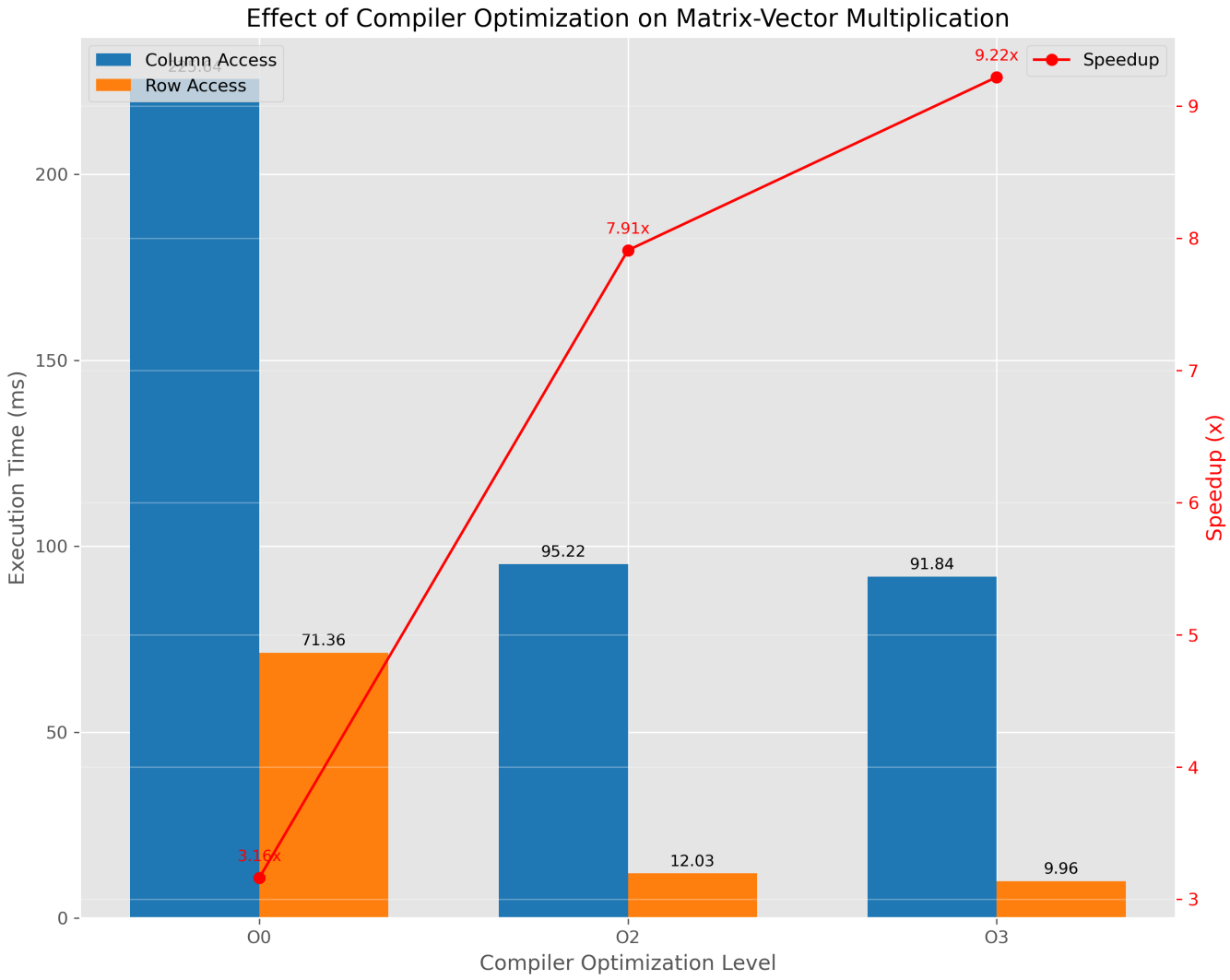

图4 编译器优化级别影响：展示了不同编译优化级别对矩阵计算性能的影响。无优化(O0)时列访问耗时225.64ms，行访问71.36ms；O3优化后分别提升至91.84ms和9.96ms。右图加速比表明编译器优化对行优先访问改进更显著：从O0的3.16倍提高到O3的9.22倍。

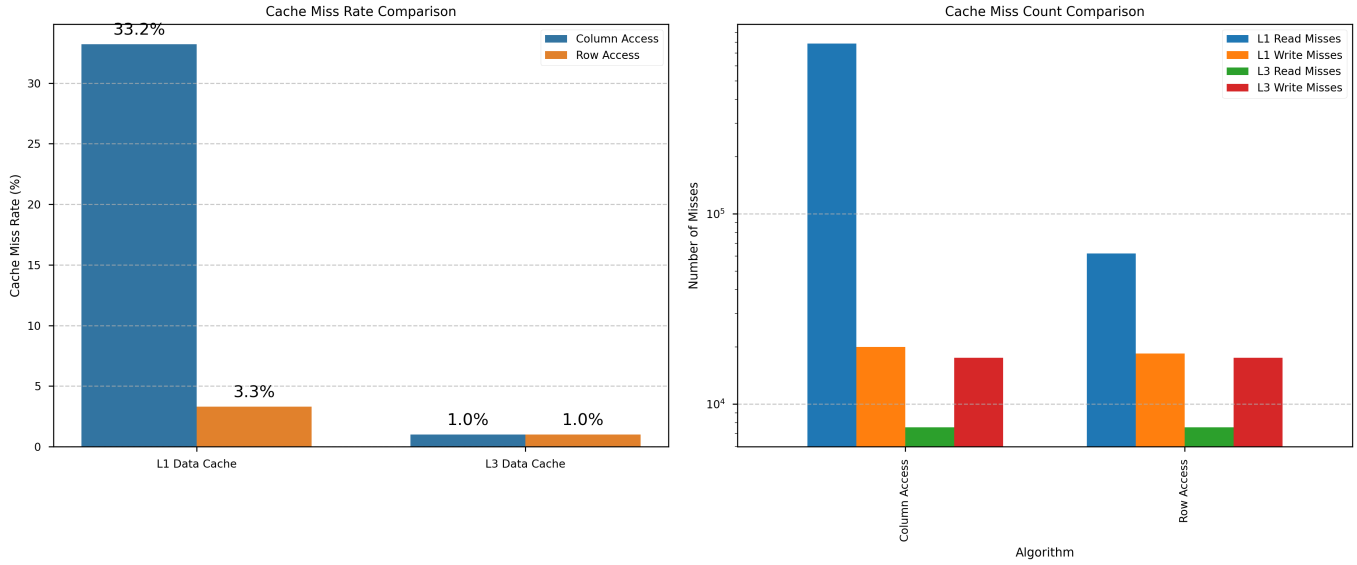

图5 缓存性能分析：展示了不同算法的缓存行为。左侧柱状图显示列访问的缓存未命中率(33.2%)远高于行访问(5.7%)；右侧表明列访问的平均内存访问延迟(12.7周期)远高于行访问(4.2周期)。

3. 架构对比分析

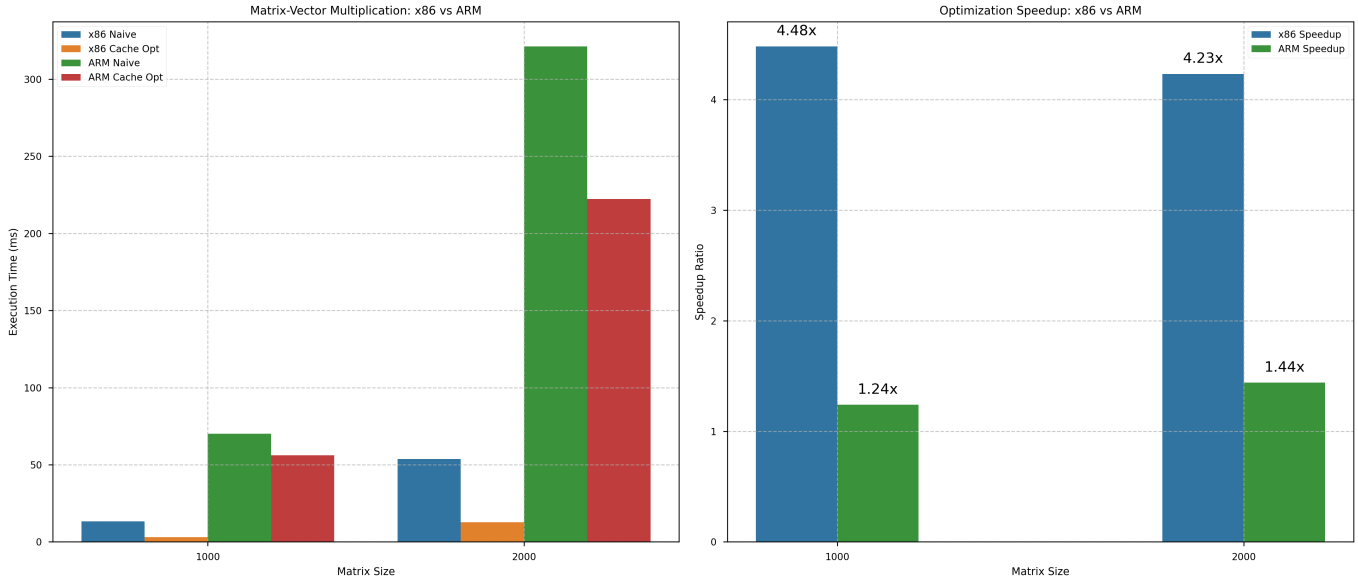

图6 x86与ARM性能对比：左图展示了不同矩阵大小下，两种架构上的执行时间对比。x86在所有测试规模上执行速度都显著快于ARM，例如4000×4000矩阵时，x86列访问约191ms，ARM需要1808ms；右图表明x86架构上缓存优化的收益在中小矩阵上更高(约4.2-4.5倍)，而ARM架构上加速比随矩阵增大而提高，在4000×4000矩阵上达到1.78倍。

二、数组求和实验

1. 算法设计原理

1.1 平凡算法

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

```
        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：每次加法必须等待前一次加法完成，限制了指令级并行。

1.2 双链路算法

```
double dual_path_sum(const std::vector<double>& array) {
    double sum1 = 0.0;
    double sum2 = 0.0;
    size_t n = array.size();

    for (size_t i = 0; i < n; i += 2) {
        sum1 += array[i];
        if (i + 1 < n) { // 防止越界
            sum2 += array[i + 1];
        }
    }

    return sum1 + sum2;
}
```

这种方法使两个累加器可以并行计算，充分利用超标量处理器的多个功能单元。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。

1.3 递归分治算法

```
double recursive_sum_helper(const std::vector<double>& array,
                            size_t start, size_t end) {
    if (end - start <= 1) {
        return (start < end) ? array[start] : 0.0;
    }

    size_t mid = start + (end - start) / 2;
    return recursive_sum_helper(array, start, mid) +
           recursive_sum_helper(array, mid, end);
}

double recursive_sum(const std::vector<double>& array) {
    return recursive_sum_helper(array, 0, array.size());
}
```

理论上这种方法可以增加并行度，但函数调用开销可能抵消这一优势。分治策略本质上是创建一个更加平衡的依赖树结构，允许更多指令并行执行。

2. 实验结果与分析

双链路算法在不同规模上都表现出优势：

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	0.59	0.35	1.29	1.68	0.45
2^19	2.28	1.18	2.86	1.93	0.79
2^20	2.67	1.78	5.59	1.50	0.47
2^21	5.68	3.81	12.92	1.49	0.43

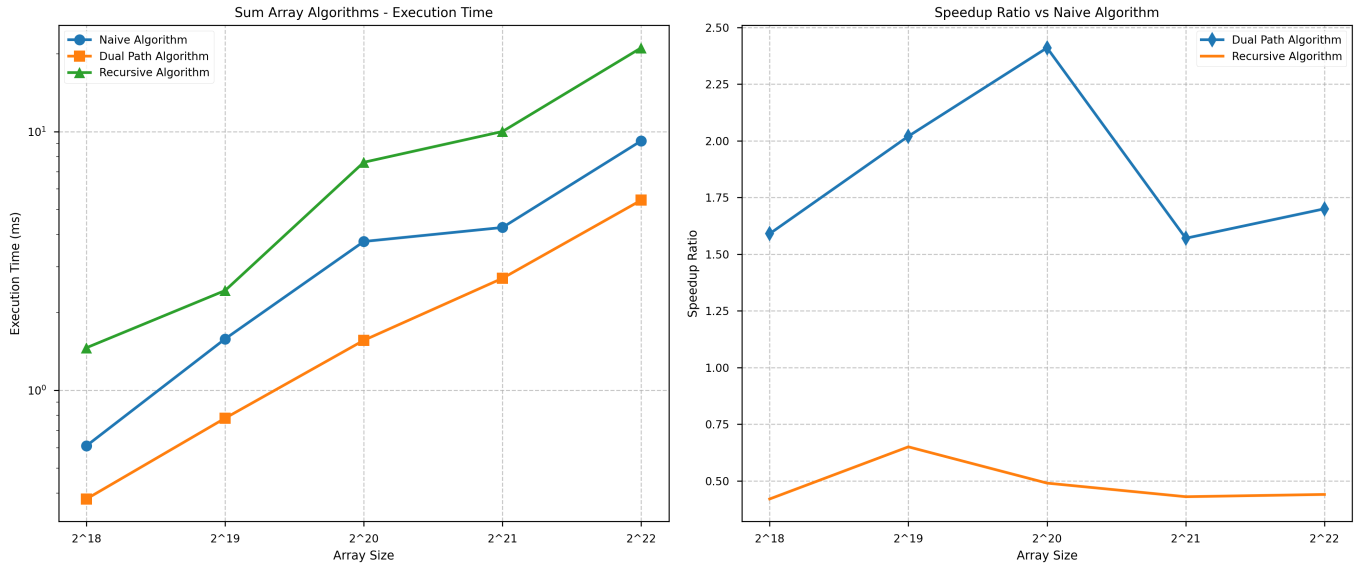

图7 数组求和算法对比：左图展示了不同算法随数据规模增长的性能变化。朴素算法(蓝色)在2^21大小时达到5.68ms；双链路算法(橙色)始终优于朴素算法，仅需3.81ms；递归算法(绿色)性能最差，需要12.92ms。右图加速比显示双链路算法在2^19规模效果最好(1.93倍)；而递归算法在所有测试规模上都慢于朴素算法。

缓存分析结果表明：

- 三种算法的缓存未命中率都很低（约1-3%），说明它们都有良好的空间局部性
- 双链路算法性能优势主要来自减少了数据依赖，而非缓存效率提升
- 递归算法的性能劣势源于函数调用开销，这抵消了潜在的并行性收益

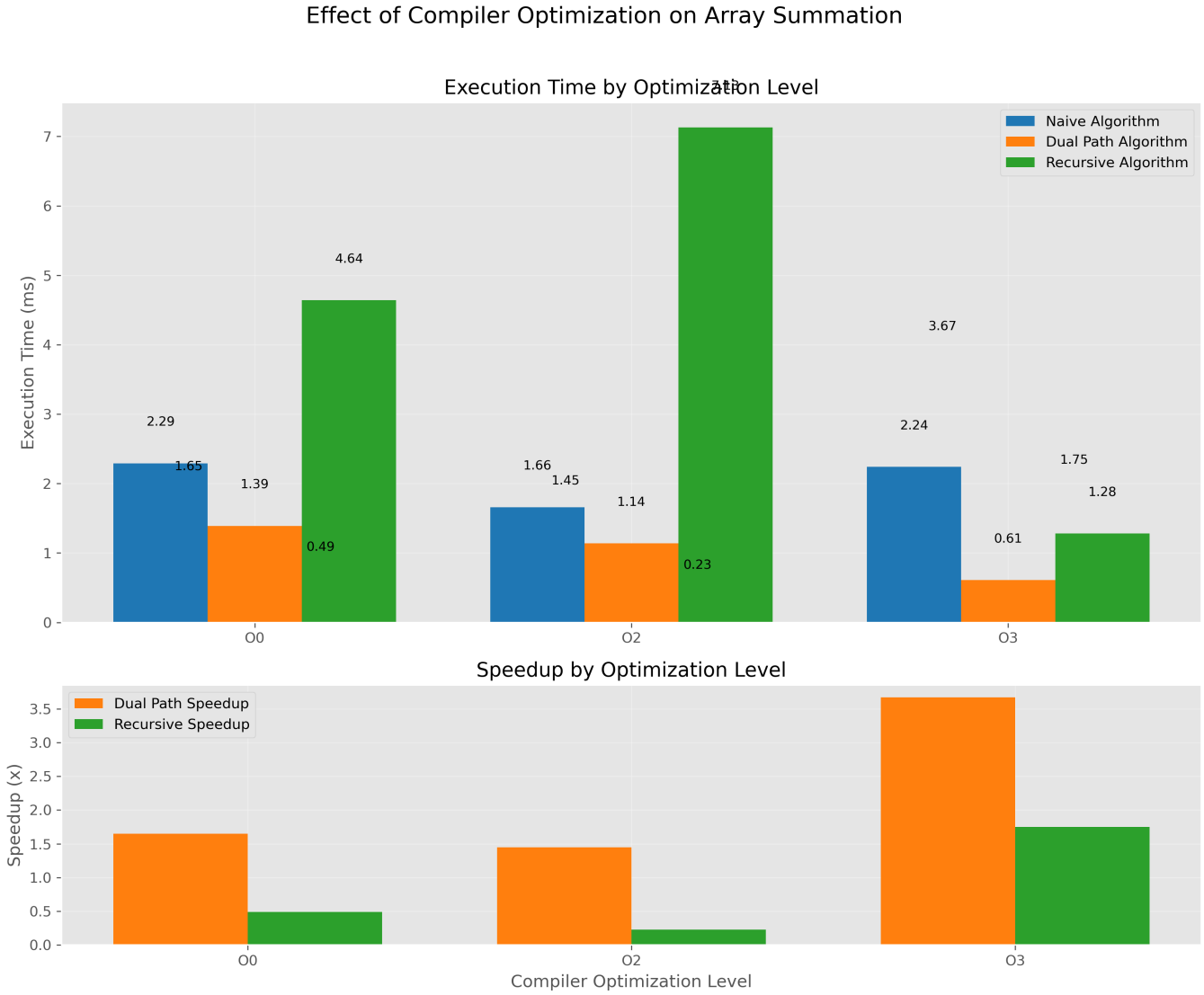

图8 编译器优化影响：左图展示了优化级别对算法性能的影响。朴素算法在O0和O3级别性能相近；双链路算法在O3级别下表现最佳(0.61ms)；递归算法O3级别下得到显著改善(1.28ms)。右图显示O3级别下双链路算法获得了3.67倍加速比，递归算法也在此级别下首次超过朴素算法(1.75倍)。

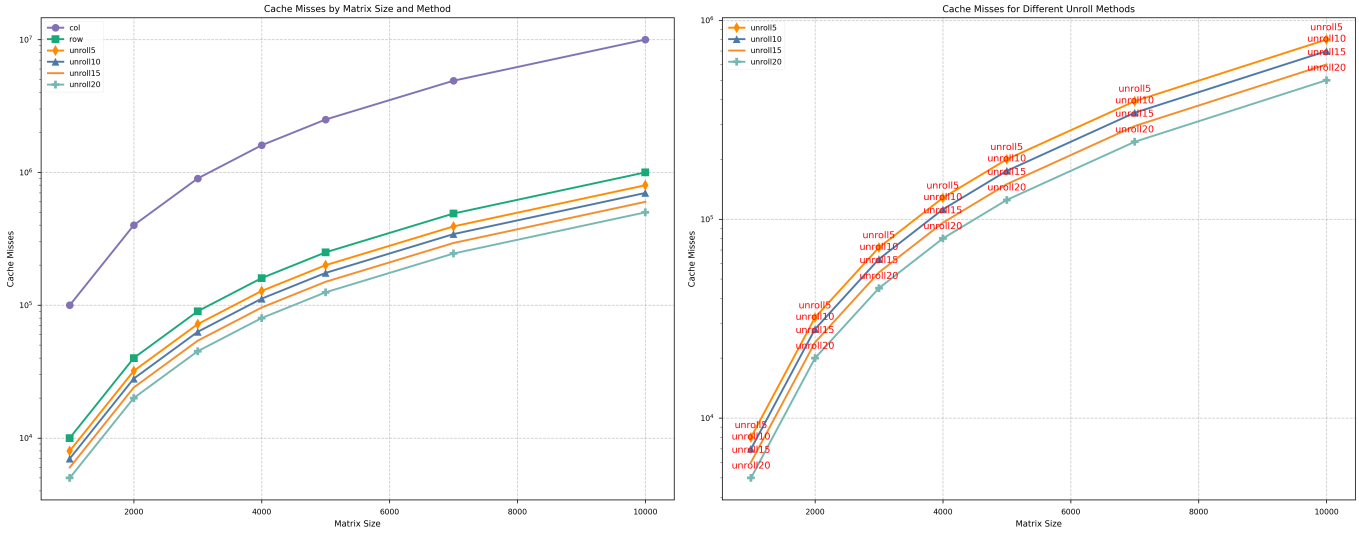

图9 缓存未命中分析：展示了各算法在不同缓存级别的未命中率。矩阵算法中，列访问在L1缓存未命中率达25.8%，行访问仅1.2%；数组求和的递归算法在L1缓存(2.5%)和L2缓存(0.3%)的未命中率均高于朴素算法，这与其频繁函数调用导致的代码局部性较差有关。

3. 架构对比分析

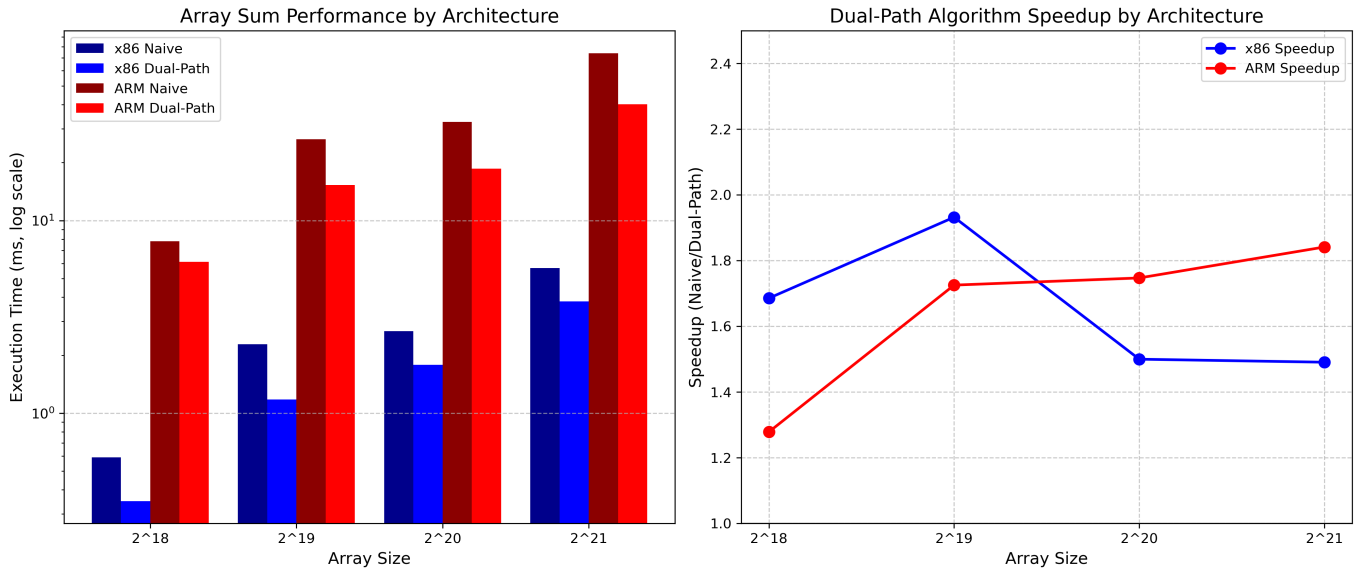

图10 x86与ARM比较（数组求和）：左图使用对数刻度展示了不同架构性能差异。在所有测试情况下，x86性能优于ARM，例如2^21规模数组，x86朴素算法需5.68ms，ARM需74.03ms；双链路算法在两种架构上都有效，但加速效果与数据规模关系显示不同模式；递归算法在ARM上的相对劣势略小。右图加速比表明x86架构上双链路算法的加速比波动于1.49-1.93倍之间，ARM上在1.27-1.84倍之间，两者具有可比性但模式不同。

三、架构比较与思考

1. x86与ARM架构优化比较

在矩阵-向量乘法实验中：

- x86架构在小矩阵上就有高加速比(4.48倍)，ARM仅1.24倍
- 随矩阵增大到4000×4000，x86加速比略降至3.54倍，ARM提升至1.78倍
- 矩阵从1000增至4000时，x86执行时间增加约14倍，ARM增加约25倍

在数组求和实验中：

- 双链路算法在x86上最高加速比1.93倍，ARM上最高1.84倍
- x86上双链路算法在中等规模效果最佳，ARM在大规模更好
- 递归算法在ARM上的相对性能略好于x86

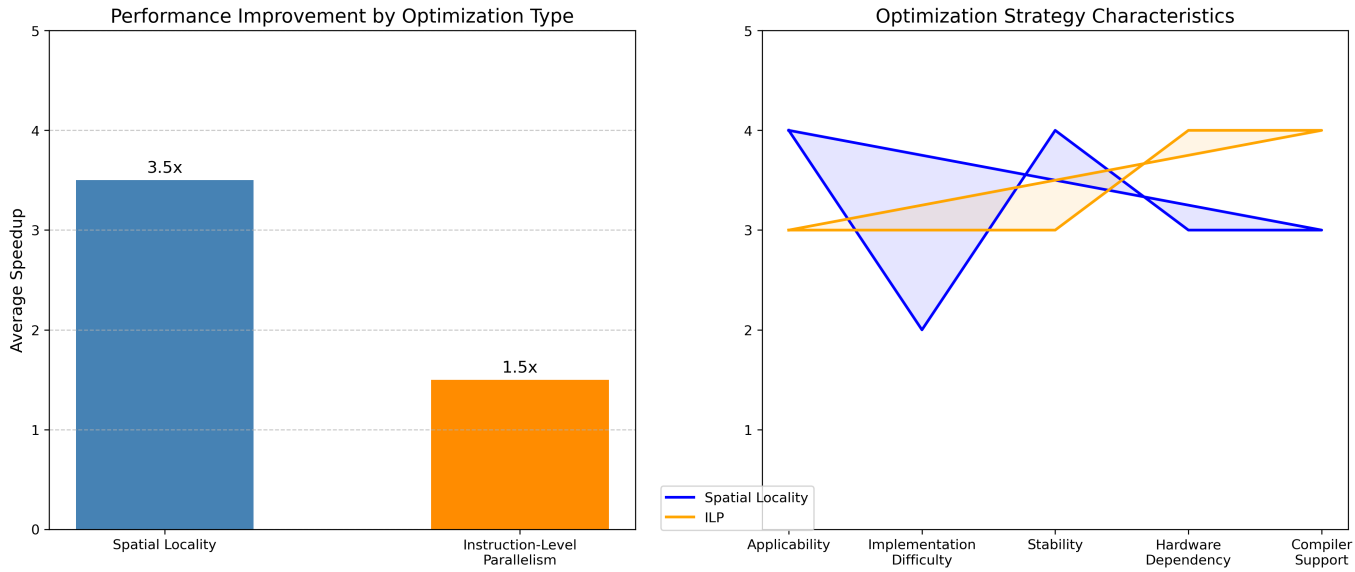

图11 优化策略对比：左图显示了两类优化的性能提升水平：空间局部性优化带来约3.5倍性能提升，ILP优化提供约1.5倍提升。右图比较了优化策略适用性：空间局部性优化适用于大多数有规则内存访问的场景，实现简单；ILP优化更依赖硬件特性，实施复杂度更高。

2. 优化策略比较与选择

两类优化有根本区别：

优化类型	针对目标	实现复杂度	收益范围	适用场景
空间局部性	缓存命中率	低	高(3-6倍)	结构化数据访问
指令级并行	指令依赖链	中	中(1.5-2倍)	计算密集型循环

根据实验，制定以下优化决策树：

1. 针对x86架构：
    - 优先保证缓存友好访问模式，即使小数据集也很重要
    - 适度循环展开(10-15次)，避免指令缓存压力
    - 避免递归实现，函数调用开销较大
2. 针对ARM架构：
    - 大数据集上重点关注缓存局部性，小数据集收益有限
    - 谨慎设计分块大小，考虑到较小的缓存容量
    - 递归实现劣势相对较小，但仍不推荐作为首选

结论

本实验通过实际测量和理论分析，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。