南開大學
Nankai University

南　　开　　大　　学

计 算 机 学 院

并行实验报告

CPU 架构相关编程

赵熙

学号：2210917

专业：计算机科学与技术

2025 年 3 月 26 日

目录

1 实验环境

1.1 硬件环境

CPU 型号	12th Gen Intel(R) Core(TM) i7-12700H
CPU 核数	20
CPU 主频	2.3GHz
CPU 缓存 (L1)	1.2MB
CPU 缓存 (L2)	11.5MB
CPU 缓存 (L3)	24.0MB
内存容量	16384MB RAM

1.2 系统环境

系统版本	Windows 11 家庭中文版 (23H2)
操作系统版本	10.0.26100.3476
编译器版本	g++ (tdm64-1) 10.3.0

2 基础要求

2.1 实验内容

2.1.1 n*n 矩阵与向量内积

计算给定 n*n 矩阵的每一列与给定向量的内积，设计两种算法:

1. 平凡算法。
2. cache 优化算法。

2.1.2 n 个数求和

计算 n 个数的和，设计两种算法:

1. 逐个累加的平凡算法。

2. 超标量优化算法 (相邻指令无依赖)，如最简单的两路链式累加；再如递归算法–两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。

2.2 算法设计

此处只展示算法主体部分

2.2.1 n*n 矩阵与向量内积

平凡算法

```
    for (int col = 0; col < N; ++col)
    {
        for (int row = 0; row < N; ++row)
        {
            result[col] += matrix[row][col] * vec[col];
        }
    }
```

Cache 优化算法

```
    for (int row = 0; row < N; ++row)
    {
        for (int col = 0; col < N; ++col) {
            result[col] += matrix[row][col] * vec[row];
        }
    }
```

2.2.2 n 个数相加

平凡算法:

```
long long sumArray() {
    long long sum = 0;
    for (int i = 0; i < N; i++) {
        sum += arr[i];
    }
    return sum;
}
```

双链路算法：在每次循环中设置两条并行“链路”，同时对两个数组元素进行加法累积，分别累加到 sum1 和 sum2 中，最后将两部分结果相加得到最终和。这种方法利用了处理器对指令级并行性（ILP）的支持，可以减少数据相关性，提高流水线效率，进而提升性能。

```
long long sumArray() {
    long long sum1 = 0;
    long long sum2 = 0;
    for (int i = 0; i < N; i+=2) {
        sum1 += arr[i];
        sum2 += arr[i + 1];
    }
    return sum1+sum2;
}
```

递归算法：类似于二叉树的递归，使用循环模拟的方式实现了一个自底向上的“递归”过程：每次将相邻两项相加，将结果写回数组前半部分。每一轮操作后，数组有效长度减半，最终在 arr[0] 中得到整体和。

```
long long sumArray() {
    for (int m = N; m > 1; m /= 2)
    {
        for (int i = 0; i < m / 2; i++)
        {
            arr[i] = arr[i * 2] + arr[i * 2 + 1];
        }
    }
    return arr[0];
}
```

2.3 性能测试

2.3.1 n*n 矩阵与向量内积

在代码中定义计算运行次数 times：

```
        const int times = 100000000000;//总计算次数
        const int t = times / (N * N);//矩阵计算循环次数
            for (int i = 0;i < t;i++) {
        vector<ull> result = columnVectorInnerProduct();
    }
```

其中 columnVectorInnerProduct() 为实验的算法主体部分。最终输出每次进行矩阵乘法的时间：

```
cout << "Execution time: " << duration.count() / t << " ms" << endl;
```

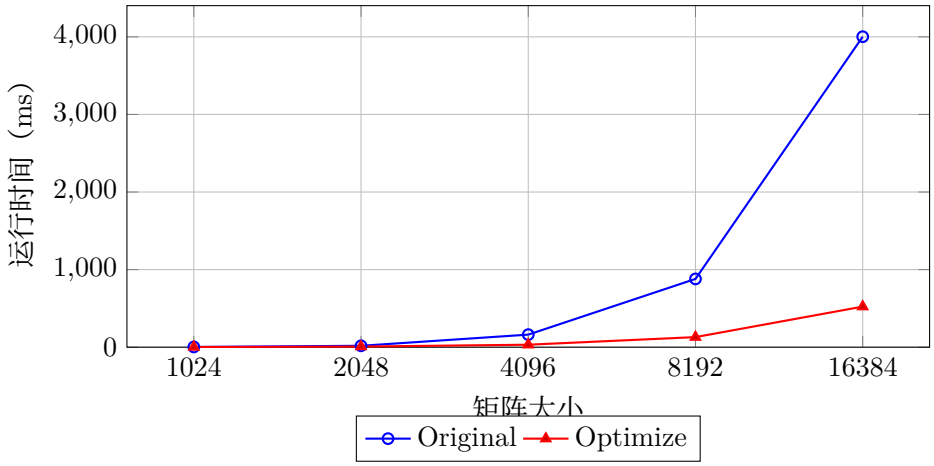
目的为多次实验获取平均数据。

最终实验结果如图：

矩阵大小	Original（ms）	Optimize（ms）
1024	3	2
2048	18	8
4096	162	32
8192	880	130
16384	4002	523

2.3.2 n 个数相加

与矩阵乘法部分类似，以下是计算次数相关定义：

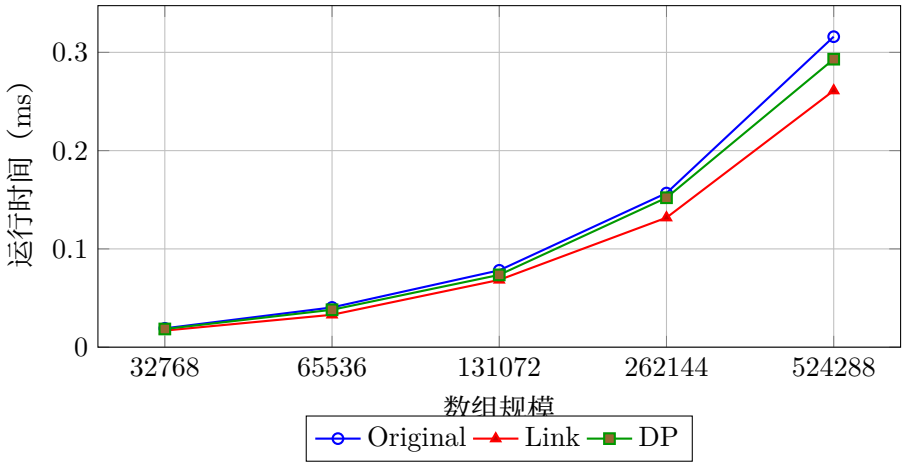



```
// 设置数组大小（n）
const int N = 1 << 15;          // 这里相当于 2^15 = 32768
// 设置总操作数
const long long times = 100000000000LL;
// 每次 sumArray() 做了 N 次加法，所以循环次数 t = times / N
const long long t = times / N;

// 全局数组，用来存放随机数据
static long long arr[N];
```

最终实验结果如图:

数组规模	Original（ms）	Link（ms）	DP（ms）
32768	0.019164	0.0169099	0.0185256
65536	0.0404495	0.0329758	0.0380678
131072	0.078212	0.0684681	0.0735698
262144	0.156812	0.131775	0.1520751
524288	0.315932	0.26107	0.2930687



3 进阶要求

3.1 循环展开优化

对 n*n 矩阵与向量内积进行循环展开优化，算法如下：

```
for (i = 0; i < n; i++) {
    sum[i] = 0.0;
}
for (j = 0; j < n; j++) {
    for (i = 0; i < n; i += 3) {
        sum[i]   += b[j][i]   * a[j];
        sum[i+1] += b[j][i+1] * a[j];
        sum[i+2] += b[j][i+2] * a[j];
    }
}
return sum;
```

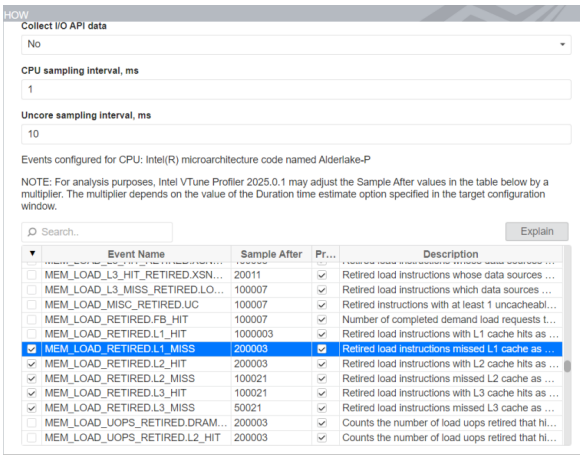
该例子中展开度为 3，展开度为几，每次在循环中就计算几个 sum。

实验结果如下：经过反复实验，发现几乎没有优化效果。

展开度 \ 数据规模	2^8	2^9	2^{10}	2^{11}	2^{12}
$k=2$	0.1043	0.4171	1.6840	6.7867	27.0294
$k=3$	0.1046	0.4195	1.6823	6.7741	27.1842
$k=4$	0.1042	0.4276	1.6797	6.7600	27.1293

3.2 练习使用 vtune

在配置 VTune Profiler 采样事件时，选中与缓存相关的硬件事件，包括 L1、L2 和 L3 缓存的命中（hit）与未命中（miss）事件，在程序执行过程中采集不同级别缓存的访问情况，从



而用于分析整体的 cache 命中率与内存访问效率。

3.2.1 n*n 矩阵与向量内积

测试结果记录如下：

策略/规模	2^{10}	2^{11}	2^{12}
L1: 平凡算法	66.07%	66.10%	66.42%
L2: 平凡算法	88.36%	88.29%	87.66%
L1: 逐行访问优化	98.78%	99.11%	98.53%
L2: 逐行访问优化	91.03%	85.71%	89.04%

3.2.2 循环展开优化没有效果的解答

使用 VTune 对不同展开度的程序进行性能分析后发现，随着展开度的提高，虽然循环次数减少导致变量 i 相关的 hit 数有所下降，但 miss 数基本保持不变，整体命中率出现了明显下降。这一现象可能源于展开过程中并行度增加所带来的频繁数据访问，打乱了缓存中的访问顺序，从而降低了缓存的有效利用率。

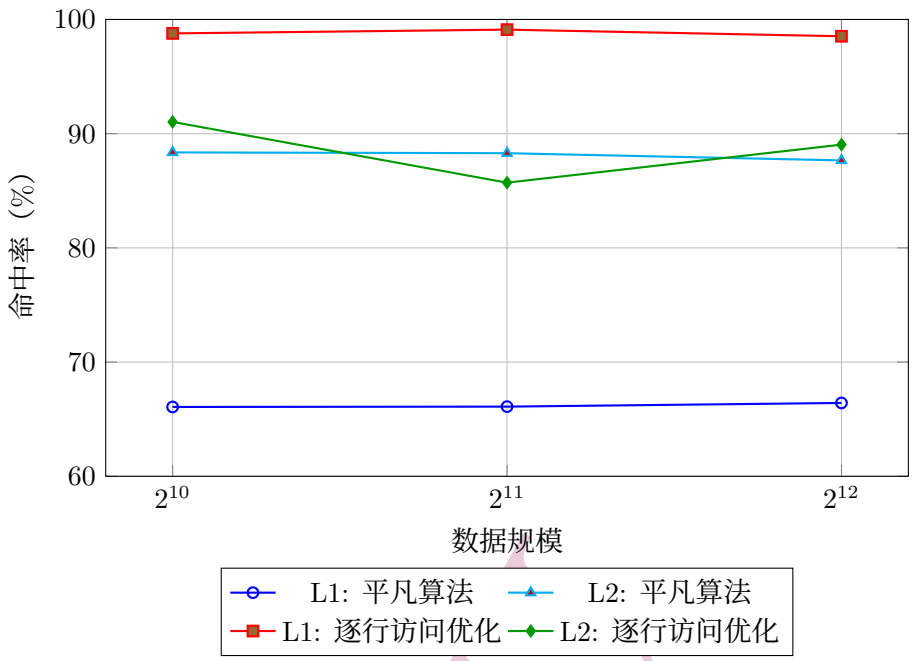


图 1: 不同优化策略在各级 Cache 下的命中率对比

4 实验总结

在矩阵乘法实验中，为提升矩阵与向量相乘运算的效率，在本次实验中对原始的计算方式进行了多方面优化。最初实现采用标准的两层嵌套循环，按列访问矩阵并逐元素与向量进行乘加操作。可以明显对比出逐行访问能够在较大问题规模下具有很好的性能表现，其原因是能够充分利用 cache 的缓存，提高数据在缓存中的命中率，进而降低了访存导致的额外开销。之后采用了循环展开（Loop Unrolling）技术，通过增大每次循环处理的向量元素数量（如 k=2, 4, 8），减少循环控制开销，并提升指令级并行性（ILP），但由于整体命中率出现了明显下降，没有得到优化效果。在数组求和实验中，实现了双链路加速算法，在一次循环中引入两个累加器并行累加数据，进一步提高流水线利用率。此外，还设计了一种类似递归归约的优化方法，通过模拟树形结构将相邻数据依次合并，从而实现更高效的累加过程。通过对这些优化方式的实验对比，分析其在不同数据规模下的运行性能与缓存利用效率。并结合 VTune Profiler 工具对各优化版本进行了性能分析，重点考察了运行时间、缓存命中率等指标。

GitHub 项目链接