南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

CPU架构相关编程

李政霖

年级：2023级

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 27 日

目录

一、 引言

（一） 实验代码

本实验用到的所有代码可以通过此网址进入github进行查看

（二） 实验环境

（1）CPU:13th Gen Intel(R) Core(TM) i9-13900HX 2.20 GHz

（2）内存容量：16GB

（3）指令集架构：x86架构和arm 。

（4）操作系统和版本：Windows 11 家庭中文版26100.3476

（5）编译器：g++10.3.0 (tdm64-1)

二、 实验一：n*n矩阵与向量内积

（一） 算法设计

平凡算法为按列遍历，逐行计算内积。而Cache优化算法为按行遍历，逐列累加，利用空间局部性。

（二） 编程实现

逐列访问平凡算法

```
void naive_matrix_vector_product(const double* matrix, const double* vector
    , double* result, int n) {
        for (int j = 0; j < n; ++j) {
                double sum = 0.0;
                for (int i = 0; i < n; ++i) {
                        sum += matrix[i * n + j] * vector[i];
                }
                result[j] = sum;
        }
}
```

Cache优化算法

```
        void optimized_matrix_vector_product(const double* matrix, const
            double* vector, double* result, int n) {
                memset(result, 0, n * sizeof(double));
                for (int i = 0; i < n; ++i) {
                        double v = vector[i];
```

```
            for (int j = 0; j < n; ++j) {
                result[j] += matrix[i * n + j] * v; // 按行遍历
            }
        }
    }
```

（三） 性能测试

测试实验设计

1.测试数据生成：使用固定值全**1**矩阵和向量，便于验证正确性并且同时可以消除数据随机性对性能测试的干扰。

2.测试数据规模选择：我选取了**n=256,512,1024,2048**四个数据规模，刚好可以覆盖不同缓存层级（**L1**、**L2**、**L3**）的容量边界，保证了测试的全面性。

3.计时方法：首先进行缓存预热，先运行多次消除冷启动误差，保证准确性，然后执行**100**次并取一个平均时间的结果，这样可以降低系统调度和噪声的干扰。

4.验证逻辑：验证每行内积结果是否为**n**。

5.性能指标：平均执行时间；不同规模下的加速比；缓存命中率

实验结果

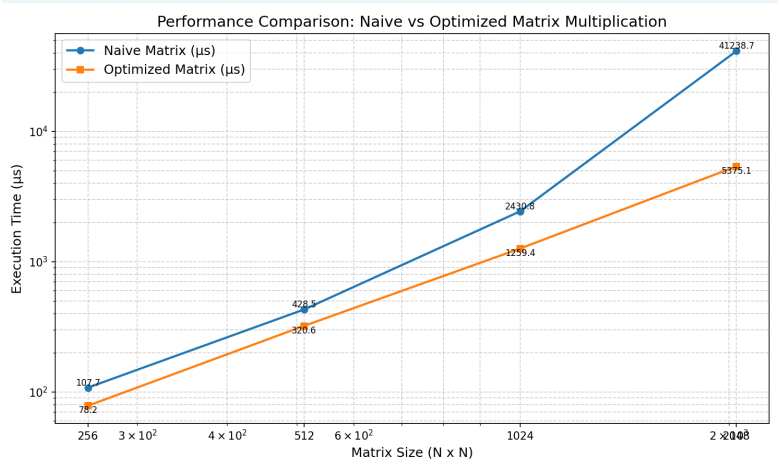


图 1: 性能随数据规模变化图表

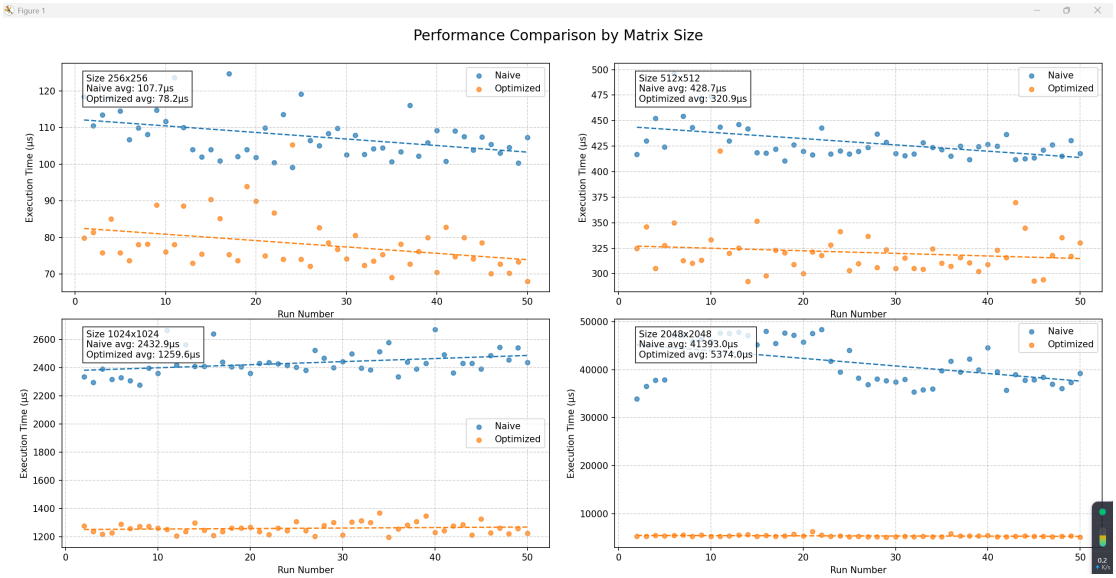


图 2: 各数据规模性能对比图表

实验结果见图1（体现平均执行时间）、图2（体现不同规模的加速比）以及profiling中的表格数据。其中蓝色代表平凡算法，橙色代表cache算法。

（四） profiling

	1500					2500				
Bound	Memory	DRAM	L1	L2	L3	Memory	DRAM	L1	L2	L3
2000	29.5%	43.4%	2.4%	0.1%	0.1%	42.4%	39.4%	0.8%	5.8%	0.0%
3000	30.8%	22.6%	13.6%	7.9%	0.0%	29.6%	25.2%	0.7%	0.0%	0.9%
4000	39.8%	51.4%	0.0%	0.0%	0.2%	42.4%	40.8%	0.0%	0.2%	4.7%
5000	27.5%	13.8%	0.0%	5.8%	2.4%	32.3%	25.7%	5.6%	8.6%	0.0%

图 3: 平凡算法

	1500					2500				
Bound	Memory	DRAM	L1	L2	L3	Memory	DRAM	L1	L2	L3
2000	17.5%	1.1%	0.0%	6.1%	6.1%	7.9%	5.8%	0.0%	3.0%	3.2%
3000	13.2%	5.6%	0.0%	1.3%	3.8%	6.5%	6.9%	6.6%	0.0%	3.3%
4000	1.9%	0.1%	0.2%	2.8%	2.9%	8.9%	6.8%	3.1%	0.0%	1.3%
5000	10.8%	8.2%	1.7%	0.0%	4.0%	7.3%	0.0%	0.1%	1.5%	4.6%

图 4: cache算法

此外，还通过使用vtune来获得了一些取值情况下的Memory bound的相关数值来反映缓存命中率，缓存命中率与数值大小成反比。具体结果见图3,4

（五） 结果分析

1.通过图一可以看出，随着数据规模的增大，平凡算法和cache算法运行的平均时间都近似为线性增长，但平凡算法的增长速率明显较快且呈现越来越快的趋势。

2.由图二还可以看出，**cache**优化后的程序性能提升十分明显，加速比很大，且随着数据规模的增大，**cache**算法和平凡算法的加速比差距逐渐增大。

3.对于上面的实验结果，可以结合缓存命中率进行分析，从**profiling**中的数据可以看出，**cache**优化后的缓存命中率大大提升，且随着数据规模的增大缓存命中率的提升幅度也在增加，可以得知优化算法通过优化缓存命中率来和平凡算法拉开性能差距。

4.另外，可以看出**cache**算法缓存未命中的情况多在**L1/L2/L3**几个缓存层级，平凡算法多在**DRAM**区域，这一结果也验证了连续的内存访问有助于**CPU**进行缓存分配，这可以帮助实现性能优化。

三、 实验二：n个数求和

（一） 算法设计

平凡算法：通过链式累加的方式来求和。超标量优化：一种方式是两路展开，另一种优化方式是四路循环展开，减少数据依赖。

（二） 编程实现

求和平凡算法

```
double naive_sum(const double* a, int n) {
        double sum = 0.0;
        for (int i = 0; i < n; ++i) {
                sum += a[i];
        }
        return sum;
}
```

超标量优化（两路展开）

```
        double optimized_sum(const double* a, int n) {
                double sum0 = 0.0, sum1 = 0.0;
                int i;
                for (i = 0; i < n - 1; i += 2) {
                        sum0 += a[i];
                        sum1 += a[i + 1];
                }
                for (; i < n; ++i) sum0 += a[i];
                return sum0 + sum1;
        }
```

循环展开（四路展开）

```
double unrolled_sum(const double* a, int n) {
        double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
        int i = 0;
        for (; i <= n - 4; i += 4) {
                sum0 += a[i];
```

```
                sum1 += a[i + 1];
                sum2 += a[i + 2];
                sum3 += a[i + 3];
        }
        for (; i < n; ++i) sum0 += a[i];
        return sum0 + sum1 + sum2 + sum3;
}
```

（三） 性能测试

测试实验设计

1.测试数据生成：数组值全为**1**，刚好结果为**n**，便于验证正确性并且可以避免浮点数精度误差干扰。

2.测试数据规模选择：选择**1000000**、**5000000**、**10000000**三组数据规模，可以进行对大规模数据优化效果的查看（经过预试验该实验数据规模太小时间太短无法捕捉，故从**1000000**作为起始）

3.计时方法：先跑几组数据进行预热，然后进行分组处理，每组测试**50**组数据并计算平均值。

4.验证逻辑：检查求和结果和**n**的误差是否在**0.00001**以内。

5.性能测试指标：平均执行时间，加速比和指令并行度。

实验结果

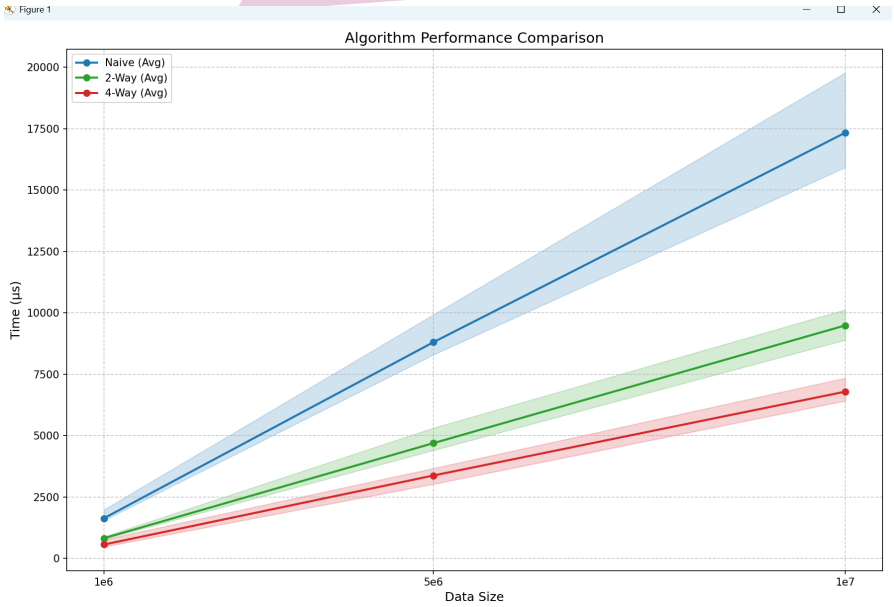


图 5: 性能随数据规模变化图表1.2

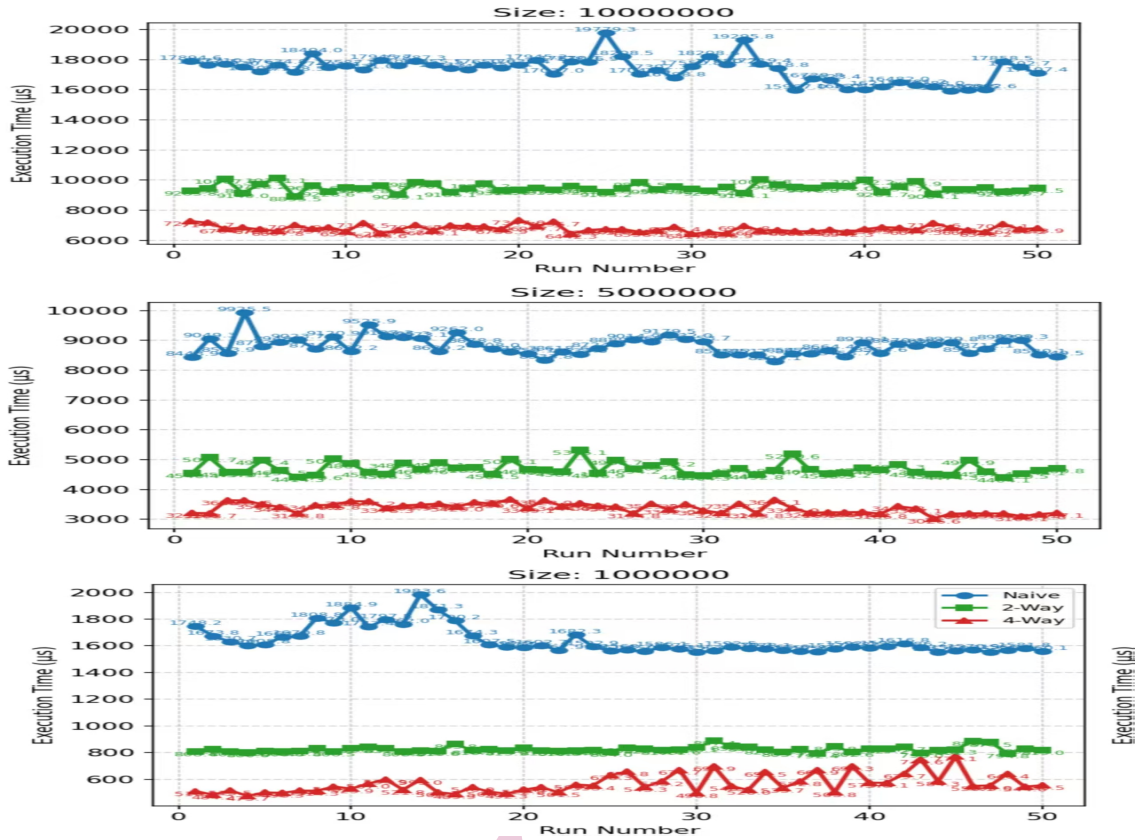


图 6: 各数据规模性能对比图表1.2（数据规模从上到下为由大到小）

实验结果见图3（体现平均执行时间）、图4（体现不同规模的加速比）以及profiling中的数据。

（四） profiling

	10000	50000	100000
平凡	0. 732	0. 725	0. 737
两路	0. 456	0. 429	0. 432
四路	0. 301	0. 315	0. 344

图 7: CPI值

此外，用vtune测量了三种方法的CPI,结果如图7

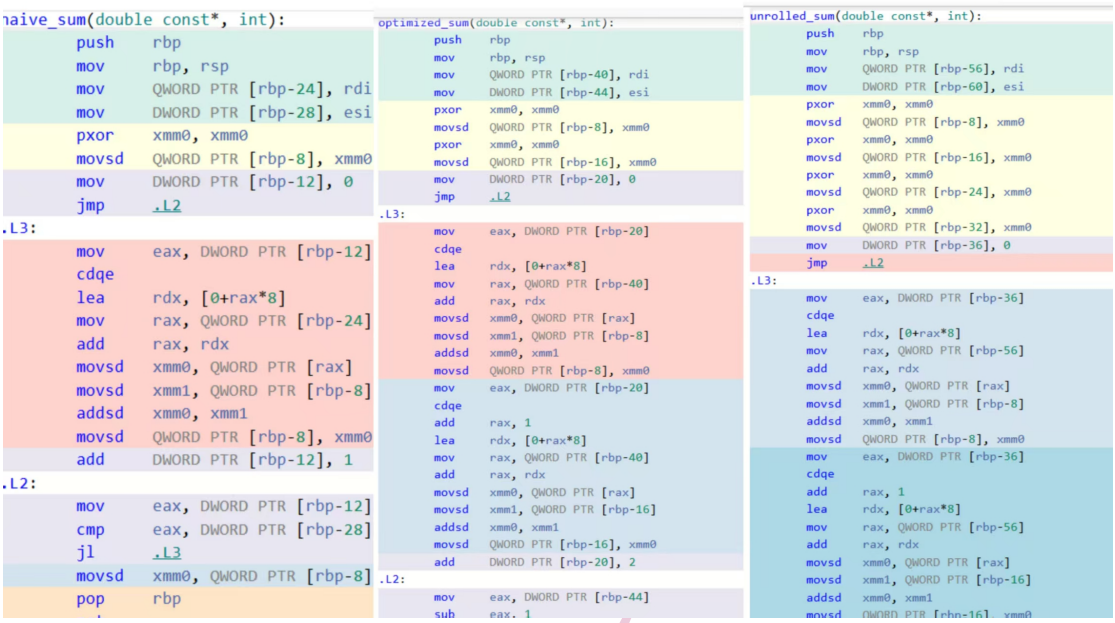


图 8: CPI值

另外，我还使用了godbolt获取了汇编代码进行分析，结果如图8。

（五） 结果分析

1.和平凡算法相比，两种超标量优化算法在性能方面均有显著提升。

2.随着测试数据规模的增大，三种算法的执行时间呈线性增长。

3.平凡算法到二路展开的性能优化的提升程度明显多于二路展开到四路展开优化的性能提升，这是因为四路展开虽然进一步减少了循环开销，但在超标量CPU上可能受限于寄存器数量和指令调度效率。

4.通过对获取的汇编代码的分析，二路优化两个独立的 addsd 指令分别操作 xmm0 和 xmm1，无数据依赖，可被CPU乱序执行引擎并行处理，理论IPC提升到2.0，每次迭代处理2个元素，循环次数减半，分支判断（cmp/jne）开销减少百分之五十。四路优化4个独立的 addsd 指令使用4个不同的寄存器（xmm0–xmm3），理论上可被4个执行单元并行处理。理论IPC提升到4.0。每次迭代处理4个元素，分支判断（cmp/jne）开销减少。

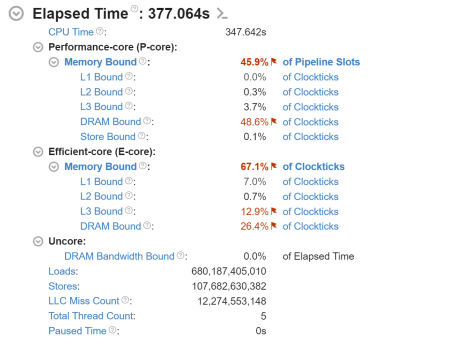
四、 进阶要求体现说明

1.实验2中采用了二路优化和四路优化两种优化力度，并分析了不同优化力度对性能的影响

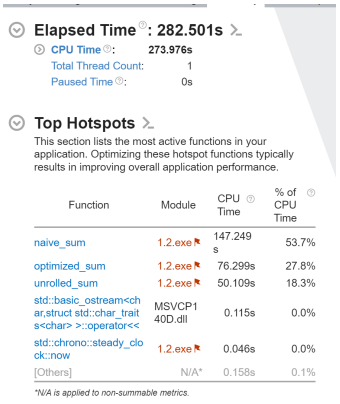
2.实验2中的优化使用了unroll技术，成功降低了循环操作对性能的影响

3.在两个实验中，我还使用vtune工具对性能更细致的进一步分析，特别是在实验1中对于缓存命中率的探究和在实验2CPI的研究中，vtune的使用截图见下方

4.在实验中，使用了https://godbolt.org/网站获取了算法的汇编代码并进行分析。



beginfigure[H]



beginfigure[H]

五、 实验总结与反思

1.通过本次实验，对于算法的优化问题有了很多了解，特别是学习到了cache和unroll两种性能优化方式。

2.通过本次实验，我对算法的性能测试标准以及测试方案的设计有了很深的理解，在此方面有了很大的提升。

3.通过本次实验，我对g++等编译环境，x86平台等的特性有了很深的理解，也vtune等性能分析工具有了一定的学习和了解，学会了vtune和godbolt等工具和平台的使用。