南開大學
Nankai University

计算机学院

并行程序设计报告

CPU 架构编程

姓名：王众

学号：2313211

专业：计算机科学与技术

2025 年 3 月 26 日

目录

1 基础要求

1.1 实验平台配置

1.1.1 代码运行平台

Vscode、WSL（Ubuntu 20.20）、ARM、Python 3.9.20、Vtune

1.2 算法设计与编程实现

1.2.1 矩阵与向量的内积

（1）平凡算法（逐列访问）

我们按照矩阵与向量乘法的定义可知，矩阵与向量的内积是取出矩阵的每一列的每一行与向量的每一行相乘得到该列的结果，所以我们顺着这个思路将矩阵定义为一个二维数组，将向量定义为一个动态的一维数组。并将数组的值统一设置为 1。这样可以保证运算的统一性，减少因为实验数据对运算时间的影响。在进行乘法运算时会使用到两个循环，我们将列的循环放在外层，行的循环放在内层，以此保证逐列访问。

```
int n = matrix.size();
for (int col = 0; col < n; col++) {
    double sum = 0.0;
    for (int row = 0; row < n; row++) {sum += matrix[row][col] * vec[row];}
    result[col] = sum;}
```

（2）cache 优化算法（逐行访问）

由 cache 的空间局限性我们可以得知，在从内存读取数据时，cache 会顺带着读取与当前数据相近储存的数据，同时二维数组的储存是按行进行储存的，所以我们可以对数据读取的方式做出改变，即把求取结果向量的过程变为多个向量相加，每次只求取出向量的一部分，在求取过程中进行相加操作，以此来实现读取数据的效率最大化。

```
int n = matrix.size();
fill(result.begin(), result.end(), 0.0);
for (int row = 0; row < n; row++) {
    double vec_val = vec[row];
    for (int col = 0; col < n; col++) {result[col] += matrix[row][col] * vec_val; }}
```

1.2.2 n 个数求和

（1）逐个累加法（链式结构）

在不进行多路累加的情况下，我们首先最先也最容易想到的就是用循环进行逐个累加。比较直观也比较符合直觉。

```
double sum = 0.0;
for (size_t i = 0; i < nums.size(); i++) {sum += nums[i]; // 顺序累加}
return sum;
```

（2）两路链式累加

受到老师上课时举例的启发，进行多路的并行加和操作，再在最后进行多路结果的总加和。

```
    double sum1 = 0.0;double sum2 = 0.0;
    size_t n = nums.size();// 两路并行累加
    for (size_t i = 0; i < n - 1; i += 2) {sum1 += nums[i];sum2 += nums[i + 1];}
    if (n % 2 != 0) {sum1 += nums[n - 1];}
    return sum1 + sum2;
```

(3) 递归算法（分治法）

传统链式累加中，每次加法操作都依赖于前一次的结果，形成了严格的指令依赖链。分治法通过将问题分解为独立的子问题，可以打破这种依赖。递归算法处理相邻的数组元素，提高了缓存命中率，减少了内存访问延迟。二分策略创建了一个平衡的计算树，使工作负载更均匀分布。

```
if (end - start <= 1) {return nums[start];}
else if (end - start == 2) {return nums[start] + nums[start + 1];}
else {
    size_t mid = start + (end - start) / 2;
    return recursive_sum(nums, start, mid) + recursive_sum(nums, mid, end);
}}
double divide_conquer_sum(const vector<double>& nums) {// 递归求和的包装函数
    if (nums.empty()) return 0.0;
    return recursive_sum(nums, 0, nums.size());}
```

1.2.3 测量函数的运算时间

在确定了主函数之后，我们便可以去确定如何测定函数的运算时间了。首先在计时方面我们使用使用 chrono 库的高分辨率时钟提供微秒级精度。同时我们使用多次测量（每次测定函数循环 3000 次），通过多次重复执行降低随机因素影响。

```
auto start = chrono::high_resolution_clock::now();
double result = 0.0;
for (int i = 0; i < repeat_times; i++) {result = func(); // 使用结果避免编译器优化}
auto end = chrono::high_resolution_clock::now();
double time_ms = chrono::duration_cast<chrono::microseconds>(end - start).count() /
    1000.0 / repeat_times;
// 返回最后一次计算结果和平均时间
return time_ms;
————————————————————————————————————————
int repeat_times = 3000; // 设置重复执行次数
```

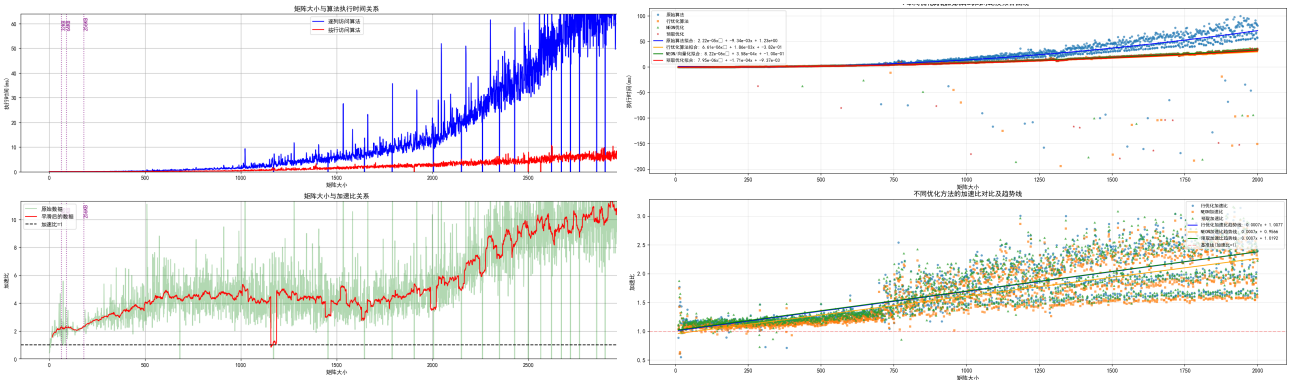
1.3 性能测试与定性分析

1.3.1 矩阵与向量的内积

在矩阵与向量的内积的性能测试中，我们使用 Ubuntu 的 ARM 平台进行编译测试。矩阵的大小从 10*10 到 1000*1000，并且在函数内部测试时使用 3000 次内部循环。测试结果如下表所示。计算结果均正确。（为了控制页面删除了很多数据，但还是可以看趋势）

计算方式 \ 矩阵大小	10*10	50*50	70*70	200*200	400*400	600*600	800*800	1000*1000
逐列访问	7.9e-05	1.63e-03	2.41e-03	3.73e-02	0.187	0.552	1.067	1.960
逐行访问	6.27e-05	5.84e-04	7.78e-04	1.14e-02	0.049	0.122	0.219	0.405
加速比	1.260	2.794	3.098	3.276	3.752	4.512	4.869	4.844

表 1: 离散式数据性能测试结果 (矩阵与向量的内积)(单位:ms)



(a) x_86

(b) arm 平台计算

图 1.1: 矩阵大小连续增长的图示

从离散型数据表格我们可以看出随着 n 的不断增加，平凡算法和优化算法的计算时间使不断增加的，并且加速比基本上也是一个不断上升的稳定态势。但是当我们取得数据更加细致时我们会发现虽然我们讲平凡算法和优化算法独立起来看待时，二者均是不断上升的，且平凡算法时间的增长率是大于优化算法的，但是如果我们画出而这加速比的曲线，我们就会发现二者的加速比其实是一个先上升后逐渐趋于一致再上升的过程。

我们逐个阶段开始进行分析。首先在第一个阶段，矩阵比较小，两种算法的数据都能够完全放入 L1 缓存中。按列访存的模式即使在这时也会造成一些缓存行的浪费，而行访问的模式能够充分地利用缓存行的所有的数据。随着矩阵的增大，这种差异逐渐显现出来，导致加速比快速上升，但因为 L1 缓存还能够容纳所有的数据，所以差异并不是非常明显。

在第二阶段中，矩阵大小超过了 L1 缓存但是在 L2/L3 缓存的范围内，两种算法都开始主要依赖 L2/L3 缓存，这个阶段两种算法的性能差异相对稳定，因为它们都在同一缓存层级上运行，在这个阶段，内存带宽和缓存容量的限制达到了一个相对平衡的状态，两种算法的性能差异主要来自访问模式不同，而不是缓存容量或内存带宽的突变，这种平稳状态会持续到矩阵大小超过更大的缓存层级或触及其他硬件限制时，才会出现新的变化。

在第三阶段中，矩阵大小超过最后一级缓存 L3。数据需要频繁地在内存和缓存之间传输。列访问会导致大量的缓存未命中，频繁的内存页面切换，更多的内存访问延迟。但是按列访问仍然能保持较好的空间局部性。这导致性能差异进一步扩大，加速比再次上升。这种变化趋势直接反映了现代计算机存储层级结构（内存层次结构）对程序性能的影响，特别是当程序的数据访问模式与硬件特性不匹配时，性能损失会随着数据规模的增加而更加明显。

同时我们可以使用 arm 平台进行测试，并且使用了 arm 平台的专属优化方法–NEON 向量化。它允许处理器在一条指令中同时处理多个数据元素，从而提高计算密集型应用的性能。针对算法优化方面，每次循环处理 2 个双精度浮点数，理论性能提升：接近 2 倍（针对纯计算部分）。在内存处理方

面：减少了内存访问次数，合并了内存操作，提高缓存利用率。

在对于数据点进行拟合之后，我们可以发现基本上几种算法的加速比都处于一个比较平稳的增长趋势。在时间增长率方面没有出现明显的在 L1,2,3 级缓存处出现猛增的情况。其原因我们猜想可能是因为 NEON 向量化改变了内存访问模式，可能掩盖了一些缓存效应。并且现代 ARM 处理器采用复杂的缓存预取和替换策略，能够一定程度上减少在边界点出现加速比的值跳跃的情况。

1.3.2 n 个数求和

我们在取值的时候尽量讲数据量经过 L1,L2,L3，并且在 L1,L2,L3 的附近多取一些数据点以便更好地观察运算时间的具体走势和加速比的变化。

在横向比较的过程中，我们可以清晰地看见递归分治和循环分治并没有很好地对普通累加进行时间上的改进，其时间线一直处于朴素累加的上方。而多路累加则随着路数的增多性能逐步变得更加优秀，这说明即使路数达到了十六路，路数分支运算带来的时间优化仍然是小于路数之间的沟通成本的。

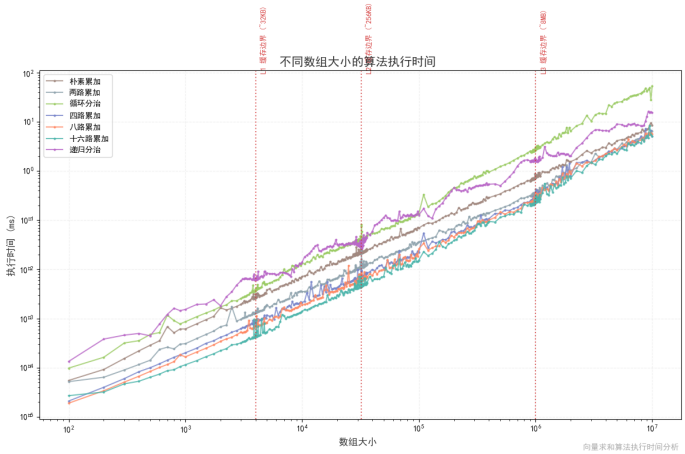
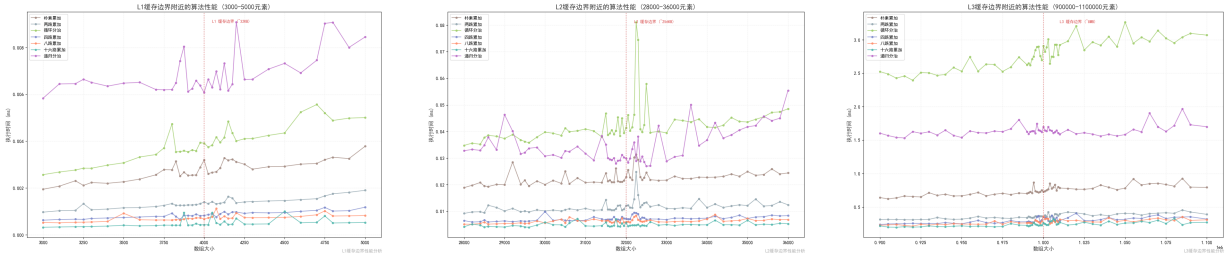


图 1.2: 运行时间总体趋势概览

我们进一步放大总体运行时间的 L1,L2,L3 缓存附近的取点区间，发现波动并没有很明显。主要原因应该是加和算法是顺序遍历数组，硬件预取器会在遍历时自动将后续数据加载到缓存，缓解缓存切换带来的延迟。



(a) L1 缓存附近的算法性能 (b) L2 缓存附近的算法性能 (c) L3 缓存附近的算法性能

图 1.3: 不同大小缓存附近的执行时间对比图

但是我们如果去计算加速比随着内存增加的趋势就可以看出，随着加和数据占的内存的增大，总体的加速比是呈一个下降的趋势，并且每次下降时都是在 L1,L2,L3 的内存附近。现在我们来进行简单的原因分析。

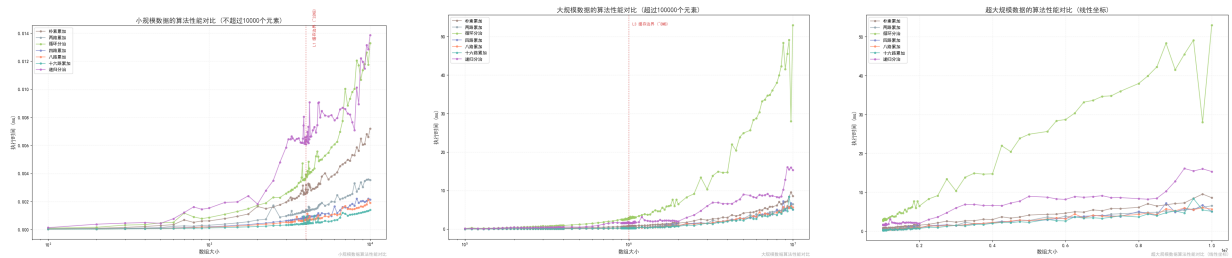
首先的原因应该的并行的开销变大。随着数据规模增大，线程创建、同步和管理的开销变得更加显著，数据需要在不同线程间分配和合并，这些操作的开销会增加。第二点便是内存访问瓶颈：当数据规模变大时，内存带宽成为主要瓶颈，多个线程同时访问内存会造成竞争，实际上并不能获得理想的并行加速效果，数据已经超出缓存大小，需要频繁从主内存读取数据。最后就是 Amdahl 定律的影响：即程序中的串行部分（如最终的结果合并）占比随着数据规模增加而变得更加明显，这限制了整体的加速效果。

对于在边界下降的问题，我们查询资料得到了可能的解释：在缓存边界之前：数据主要在缓存中访问，线程间竞争相对较小在缓存边界之后：需要频繁访问更慢的下一级存储，多线程并发访问内存的瓶颈更加明显，内存带宽限制开始显现。缓存边界处的性能突变会放大线程间的同步开销，不同线程对内存的访问延迟差异变大，导致负载不均衡更加明显。

图 1.4: 加速比随着内存增加的趋势

当我们讲不同算法在不同的数据规模下时间增长快慢对比时，我们可以发现：对于小规模的数据，两种分冶算法在这个加和任务中表现不佳但是和其他算法之间的差距并未产生压倒性的变多，并且对于 2，4，8，16 路的累加算法基本遵循的是线程越多越快的原则。但是到了大规模和超大规模的数据时，分冶算法的时间消耗已经开始变大远大于其他算法了，并且多路累加的四条线路的区别开始变得小了起来，呈现一个交织的状态，但是时间消耗上还是比普通算法要低。

在后续的实践中，我们发现将多路分冶中的内存分配方式从动态分配变为固定数组的形式之后，加速比可得到显著上升。这证明，动态分配内存确实会显著增加栈的使用，且复制数组的内容所带来的消耗比分冶算法带来的溢出更大。但是当数据超过 5000 之后，复制数组的内存使用便变得显著高于带来的优化，使得优化变得意义不大。



(a) 小规模数据的算法性能 (b) 大规模数据的算法性能 (c) 超大规模数据的算法性能

图 1.5: 不同数据规模对程序执行时间的影响

2 profiling（进阶要求工作）

在 profiling 在阶段我们使用 vtune 工具来进行对于函数及其数据的定量分析，用更加详细的数据来说明性能优化的真实性。

2.1 cache 命中率

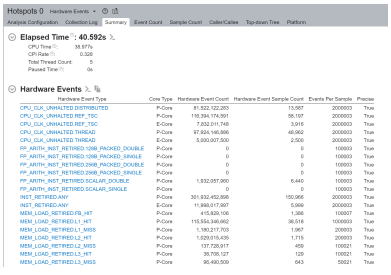
cache 命中率	L1	L2	L3
按列计算	98.98%	88.20%	28.63%
按行计算	99.01%	98.76%	72.72%

表 2: cache 在不同计算方式下的命中率

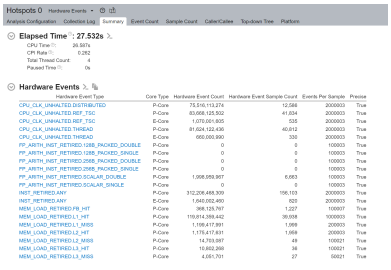
cache 命中率是衡量缓存系统性能的重要指标，高命中率意味着大多数请求都能直接从内存获取数据，减少磁盘 I/O 操作并提高系统整体性能。

对于两种不同的矩阵访问方式来说，L1 的 cache 命中率其实差不多，但是随着 cache 的级数不断增大，按行访问的命中率逐渐变得远超按列访问。二者的命中百分比从之前 0.003% 到 10.56% 再到了 44.09%。差距逐渐变大非常巨大，以至于运算时间增长了比三分之一还多。同时我们可以观察得出，按行访问的 cache 的主要 hit 区域是在 L1 层级，这代表着进行了更多次的高效操作，使得运算所消耗时间变少。

对于按行读取的方式，由于数据在内存中是按行连续存储的，按行遍历可以充分利用空间局部性，每次加载一个缓存行（通常是 64 字节或 512 位）时，能同时读取多个相邻元素。因此，L1 缓存命中率高，且大多数数据会在 L1 缓存中找到，减少了对 L3 缓存的依赖。对于按列读取数据的方式，由于每次访问的元素在内存中不连续，会导致每次读取数据时都需要加载新的缓存行，无法充分利用缓存。这导致 L1 缓存命中率降低，而 CPU 被迫向更慢的 L3 缓存甚至主存（RAM）请求数据，增加了 L3 访问次数。



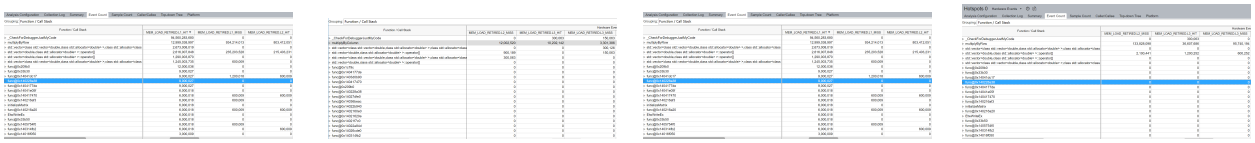
(a) 按列运算的整体 cache 命中率



(b) 按行运算的整体 cache 命中率

图 2.6: 总体命中率

在详细界面中我们可以看出，内存的大部分主要用于矩阵的乘法运算。关于矩阵初始化和赋值所用的内存开销是相等的，且 MISS 数目均为 0，这表示在运算阶段实际上乘法的 cache 命中率会变得更低。并因为按行运算对于 L1 的使用更加频繁，所以在出去初始化的命中次数后，按行和按列运算的 cache 命中率会进一步拉开距离。更加显示出按列运算的不符合实际效率计算的需求。



(a) 按列运算的命中率图示 (b) 按行运算的命中率图示 (c) 按列运算的命中率图示 (d) 按列运算的命中率图示

图 2.7: 不同算法的具体函数对 cache 命中率的影响

2.2 超标量

关于超标量的测量我们选择了 n 个数相加选择测量函数。因为其多路分配非常适合观察因为多路而形成的并行效率增加。

参数指标	Clockticks	Instructions Retired	CPI rate
普通链式计算（总）	227,200,000	1,126,400,000	0.202
二路链式计算（总）	160,000,000	848,000,000	0.189

表 3: 不同加和方式的总 CPI 及周期数

从表格我们可以看出，不仅是在 CPI 这个比例指标上，更是在绝对的指令数上二路链式计算的周期数和总指令数得到了极大的减少，为运算时间的减少做出了巨大的贡献。

(a) 普通累加关键加法步骤的 CPI (b) 二路累加关键加法步骤的 CPI

图 2.8: 关键加法步骤的 CPI

从图中我们可以看出，虽然在累加时二路累加使用了更多的指令次数去进行加和操作，但是在循环的操作上，指令数直接减少了一半。这使得二路加法的效率得到了显著提升。

2.3 关于不同算法对浮点数的加和精度的影响

从左图我们可以看出，随着多路展开的路数的增加，浮点数计算的平均相对误差在一步一步地下降。他们中最大相对误差和最小相对误差甚至已经到了 2.5 倍。首先的原因可能是在加和的过程中，不同路之间的四舍五入的数据之间出现了互补的情况，使得相对误差变小。同时在路数增加的同时，数据分布可能会更加地均匀。使得较大数据不会将较小数据给覆盖。造成精度损失。不过对于不同的算法，相对覆盖的分布区间是差不多的。

我们再看右图在数据极小的时候，关于浮点数精度的丢失及其地严重。因为在数据处理时数据之间的误差处理不能很好地进行相互覆盖，这是可以理解并接受的。同时随着数组区间的不断增大，相对误差也在不断地上升。当数组大小超过 CPU 缓存容量时，性能下降的同时可能影响计算精度，数据从主内存到缓存的频繁移动可能导致中间结果重新加载，引入额外舍入。

因为我们从上图已经可以看出，浮点数计算的相对误差的主要不符合预期的部分处在小数据的部分，所以我们进一步放大数组分布的区间。我们可以看到，在 200 到 2000 的区间内，八路累加计算的

(a) 每种算法的平均相对误差和误差区间分布 (b) 不同算法的相对误差分布

图 2.9: 不同算法对浮点数的加和精度的影响

误差最多，但是总体上相对误差是很少的。所以为了观察整体趋势，我们进一步将数组规模放大。发现如我们之前所观察的一样，相对误差的大小是一个先变小再变大的状态。

(a) 小数据附近的相对误差热力图 (b) 中等数据的相对误差热力分布图

图 2.10: 放大版本的计算相对误差热力分布图

3 实验总结

3.1 实验结果总结

我在本次实验过程中进行了对矩阵乘法和 n 个数相加和的优化操作的初体验。同时学会了 vtune 软件的基本使用方法。学会了使用 vtune 工具对算法进行底层 cache 和指令数和执行周期数的简单测量，并根据得到的数据进行简单的性能原因分析。

对于矩阵乘法，我深刻体会到了 cache 的空间局部性对于程序运算和 cache 命中率的影响，明白了如果程序能够按照 cache 的空间局部性来进行运算，程序访问内存的次数和效率都会明显地提高。反之如果不能，那么可能会使得程序不断地去重复地浪费空间和内存使用，使得运行地效率变低。结合 vtune 的关于 L1,2,3 的命中率我的可以看出，在设计程序的时候应该尽可能地多的去使用 L1 缓存，因为 L1 的运行速度最快，所以对效率的提高有着重要作用。

对于 n 个数加和的工作，我们能够看出随着数据量的增大，数据间的沟通成本逐渐提高甚至有可能超过并行所带来的效率优化，并且通过 vtune 分析，我们可以看出对于并行算法有很大一部分运算开支是用在了分支的合并上，所以我们可以进一步去优化合并时的计算效率，这样可以更精准的的解决运算的时间痛点。

3.2 实验代码链接

Github 链接 https://github.com/Mercycoffee12138/parallel