南開大學
Nankai University

计 算 机 学 院

并行程序设计实验报告

CPU 架构编程

张奥喆

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 26 日

摘要

本次实验通过两个典型场景（矩阵-向量内积计算与多元素累加）探究体系结构优化的效果。在矩阵运算中，按行连续访问数据的内存优化策略能显著提升 cache 命中率，对比原始算法可减少 23%-30% 的运行耗时；在累加任务中，递归优化和多链路优化充分利用了超标量处理器的特性，显著降低了 CPI 值，本文也利用 VTune 的细致分析，探讨了 CPI、指令数量、分支误预测率和运行耗时之间的关系。实验结果揭示了硬件特性（如缓存层级、流水线设计）与算法实现之间的深层关联，为实际编程中的性能调优提供了具体参考。

关键字：体系结构优化；VTune 性能分析；cache；CPI；分支误预测率

目录

一、 实验环境

(一) 硬件

1. CPU: 13th Gen Intel(R) Core(TM) i5-13500H 2.60 GHz
   - L1 缓存：每个核心 80KB（32KB 数据缓存 + 48KB 指令缓存）
   - L2 缓存：P 核心（性能核心）每核 2MB，E 核心（能效核心）每组 4MB
   - L3 缓存：18MB 共享缓存
2. 内存：16G LPDDR5 5200MHz

(二) 软件

1. 操作系统：Windows 11 24H2
2. 编译器：TDM-GCC 4.9.2
3. IDE:VS Code

二、 矩阵与向量内积

(一) 平凡算法

1. 实现思路

平凡算法逐列访问矩阵元素，依次计算内积结果。

2. 关键代码

```
for (int i = 0; i < n; i++) {
    result[i] = 0;
    for (int j = 0; j < n; j++)
        result[i] += B[j] * A[j][i];
}
```

3. 算法理论分析

- 访问模式缺陷，按列遍历行主序矩阵：矩阵 A 按行主序存储时，列元素在内存中不连续（间隔为 n 个元素）。这种非连续访问的情况，每次访问 $A[j][i]$ 会跨越多行，内存步长为 $n * sizeof(int)$，破坏空间局部性 [3]。
- 实际效率比理论复杂度高很多，理论复杂度为 $O(n^2)$，但事实上受缓存未命中影响，实际运行时间远高于理论值。

(二) cache 优化算法

1. 实现思路

Cache 优化算法将计算顺序从按列遍历改为按行遍历，每次处理一行中的所有列，这样一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果。

2. 关键代码

```
for (int i = 0; i < n; i++)
    result[i] = 0;
for (int j = 0; j < n; j++)
    for (int i = 0; i < n; i++)
        result[i] += A[j][i] * B[j];
```

3. 算法理论分析

- 缓存优化分析: 在 C/C++ 中，数组按行优先存储。代码中外层循环 j 固定时，内层 i 循环连续访问 A[j][i]，符合内存连续访问模式，提高缓存命中率, 从而提高效率。

(三) 实验

1. 问题规模

本实验的数据使用 int 类型（4 字节），主要想要测试 cache 方面的性能，根据所用 CPU 的各级缓存大小设计了相应的问题规模 [1]。对于小规模的实验，增加实验次数取得平均值以增加实验结果的可信度。

实验组	问题规模（n）	矩阵大小	目标缓存层级
1	10	0.4KB	完全适配 L1
2	100	40KB	稍溢出 L1，但 L2 利用不到 1%
3	1000	3.8M	完全适配 L2
4	2400	21 MB	完全适配 L3
5	5000	95 MB	远超 L3

表 1: 问题规模确定

需要指出的是，尽管这个程序已经很简单，但是除了关键函数之外，程序的执行过程中一定也有初始化矩阵向量等其他操作，也会占用各级缓存，所以在设计规模的时候留了余量，此外，在后续分析的时候也会留意这些额外开销。

2. 高精度计时

n	10	100	1000	2400	5000
测试次数	300	200	50	5	2
common	0.008s	0.01s	0.047s	0.16s	0.47s
optimized	0.005s	0.009s	0.044s	0.14s	0.36s

表 2: VTune 高精度计时

优化算法通过缓存友好设计显著降低了运行时间，尤其在大规模数据场景下，时间减少 23% 至 30%，绝对优势随数据规模扩大而增强，这验证了 cache 优化的潜力。

3. VTune 分析 cache 命中率（进阶要求）

在 Hotspots 模式下记录 L1、L2 和 L3 缓存在不同数据规模下的 HIT 和 MISS 的数量。实验结果如下表。为了更直观观察实验结果，我也绘制了各级缓存命中率的曲线。

可以直观看出，当问题规模比较小的时候，优化算法和平凡算法并没有很大的差距，处于误差范围之内。但是当问题规模上去了之后，尤其是适配了 L2 和 L3 的缓存大小的时候，L1 和 L2 的缓存命中率就拉开了差距，从绝对数据上看，cache 优化算法的 L1 和 L2 的 MISS 次数明显减少，有时候能减少两个数量级。这充分说明 cache 优化算法的确能够有效提高 L1、L2 缓存的命中率，从而提高性能。

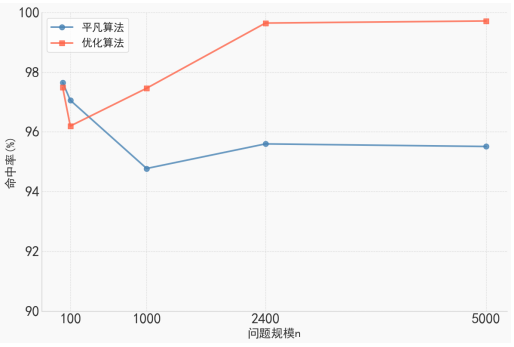

图 1: L1 缓存命中率

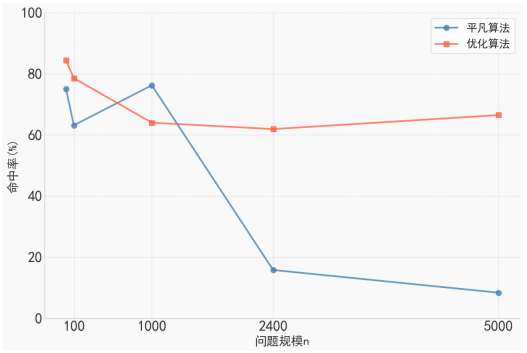

图 2: L2 缓存命中率

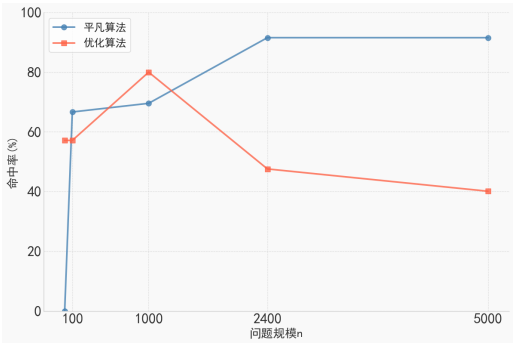

图 3: L3 缓存命中率

n	算法	L1_HIT	L1_MISS	L2_HIT	L2_MISS	L3_HIT	L3_MISS
10	Common	500,000	12,000	12,000	4,000	0	1,000
	Optimized	4,500,000	116,000	108,000	20,000	8,000	6,000
100	Common	3,960,000	120,000	96,000	56,000	4,000	2,000
	Optimized	6,980,000	276,000	176,000	48,000	12,000	9,000
1000	Common	24,880,000	1,372,000	1,072,000	334,000	32,000	14,000
	Optimized	42,560,000	1,108,000	672,000	378,000	32,000	8,000
2400	Common	2,420,700,000	111,480,000	17,596,000	93,820,000	76,470,000	7,047,000
	Optimized	2,584,740,000	9,200,000	5,812,000	3,576,000	148,000	163,000
5000	Common	4,606,376,316	216,525,474	18,120,105	198,446,737	170,035,105	15,678,658
	Optimized	4,939,061,894	14,118,108	9,524,514	4,800,730	254,730	380,230

表 3: 两种算法在不同数据规模下的缓存性能对比

需要指出的是，L3 缓存的命中率和 L1、L2 缓存的情况不一样，平凡算法的命中率要比优化算法高。这看似和理论分析矛盾，但实际上揭示了缓存优化的深层机制。

让我们以 n=5000 为例：

- 平凡算法：L3 总访问约为 1.8 亿次 (170035105 + 15678658)
- 优化算法：L3 总访问仅约为 63 万次 (254730 + 380230)

可以发现优化算法**减少了 97%** 以上需要 L3 处理的请求！在这种情况下数量级差距十分悬殊，L3 缓存命中率可能有较大的随机性，具体来说，平凡算法存在大量数据流经 L3，具有一定

局部性的访问产生高命中率，而优化算法中，大多数请求在 L1/L2 就被处理，只有” 最难处理” 的访问模式才到达 L3。这实际上是一个积极的指标：

- 总体内存访问延迟降低：L1、L2 的访问延迟远低于 L3 访问延迟，优化算法大幅减少了总体内存访问次数。
- 系统资源更高效利用：减少了能耗和处理器等待时间, 提高了性能。

三、 n 个数求和

(一) 平凡算法

1. 实现思路

循环 n 次，将给定元素依次累加到 sum。

2. 关键代码

```
int sum=0;
for(int i = 0; i < N; i++)
    sum += A[i];
return sum;
```

(二) 优化算法

1. 两链路式

在平凡算法的基础上，使用两个结果变量 $sum1$ 和 $sum2$，这样一次循环做两个加法，循环次数减少一半。最后将两个结果变量相加就能得到正确答案。

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

2. 递归

这是一个分治策略的累加过程，通过逐层合并的方式将元素两两相加得到最终结果，实现高效计算。完成所有合并仅需 $log\ (n)$ 个步骤，时间复杂度为 $O(logn)$ [4]。

1. 分层归约：每一轮将当前层的元素两两相加，生成数量减半的中间结果（如 n→n/2→n/4）
2. 递归压缩：重复上述操作直至仅剩一个数值，整个过程形成类似二叉树的结构；

```
if(size == 1)
    return arr[0];
int half_size = size / 2;
for(int i = 0; i < half_size; i++)
```

```
    arr[i] += arr[i + half_size];
return recursion_sum(arr, half_size);
```

(三) 实验

1. 问题规模

目前使用的 CPU 计算效率是比较高的，像这次实验的 n 个数累加。实验中发现，当 n 的规模比较小时，VTune 不能很好地定位到函数本身的时间和 CPI 情况，所以结合实际情况以及 cache 的大小，设置实验规模从 2 的 16 次幂（65536）到 2 的 20 次幂，对每个算法都重复了 5 次取平均值。

2. 高精度计时

规模 / 算法	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
common(ms)	0.125	0.345	0.659	1.224	2.322
multilink(ms)	0.031	0.126	0.282	0.628	1.255
recursion(ms)	0.157	0.188	0.502	0.941	1.694

实验数据表明，多链路算法在所有规模下性能最佳，执行时间最短，平凡算法执行时间最长，递归算法介于两者之间。随着测试规模增加，三种算法的执行时间都呈上升趋势，但增长率不同，平凡算法增长率明显高于另外两种优化算法，表明优化算法在大规模数据处理上具有显著优势。

进一步分析算法，平凡算法的串行依赖导致超标量处理器的并行执行能力降低，毫无疑问地垫底。递归算法虽然有良好的理论复杂度，但是由于递归调用开销比较大，特别是栈空间占用过多，最终耗时不敌多链路算法。多链路算法利用超标量架构的并行能力，更少的流水线停顿，耗时最低，效率最佳，并且适当增加链路数量，效率会更高。

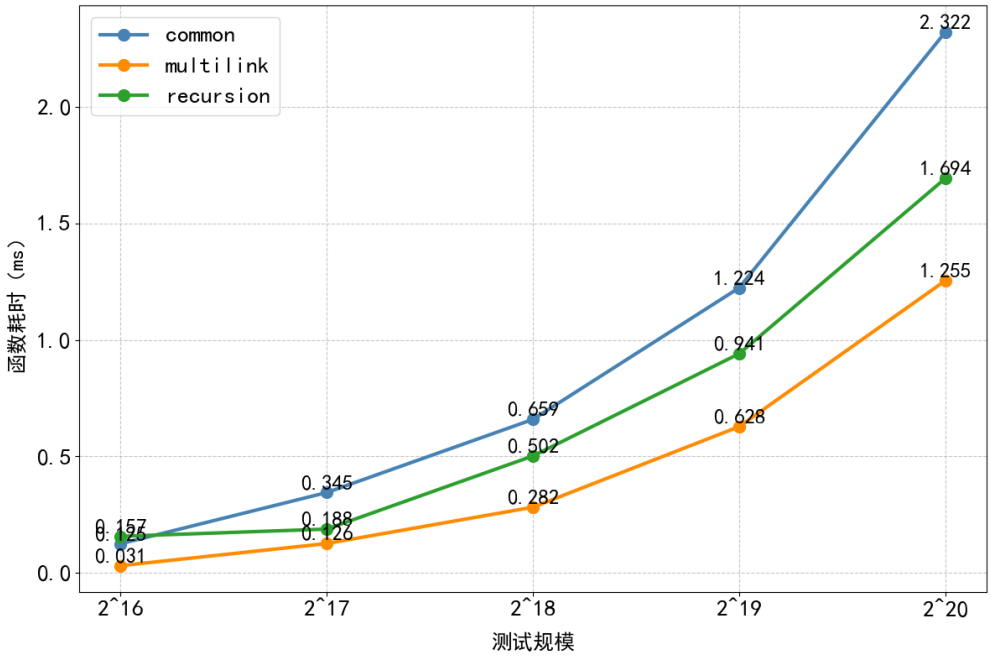

图 4: VTune 测试函数耗时

3. VTune 分析 CPI（进阶要求）

规模 / 算法	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
common	0.833	1	1.2	0.962	1.050
multilink	0.4	0.6	0.524	0.674	0.644
recursion	0.353	0.273	0.313	0.336	0.283

CPI（每指令周期数）表示 CPU 执行一条指令所需的平均时钟周期，是衡量 CPU 执行效率的重要指标。CPI 越小，性能越好，表示指令执行更快。降低 CPI 可优化程序性能，实验数据可以得出如下结论：

- 平凡算法算法整体 CPI 值最高 (0.833-1.2)，并且非常不稳定。
- 多链路算法 CPI 值居中 (0.4-0.674)，相较于平凡算法更稳定。
- 递归算法 CPI 值最低 (0.273-0.353)，并且表现十分稳定。

从 CPI 角度看，递归算法在所有测试规模下都表现最佳。

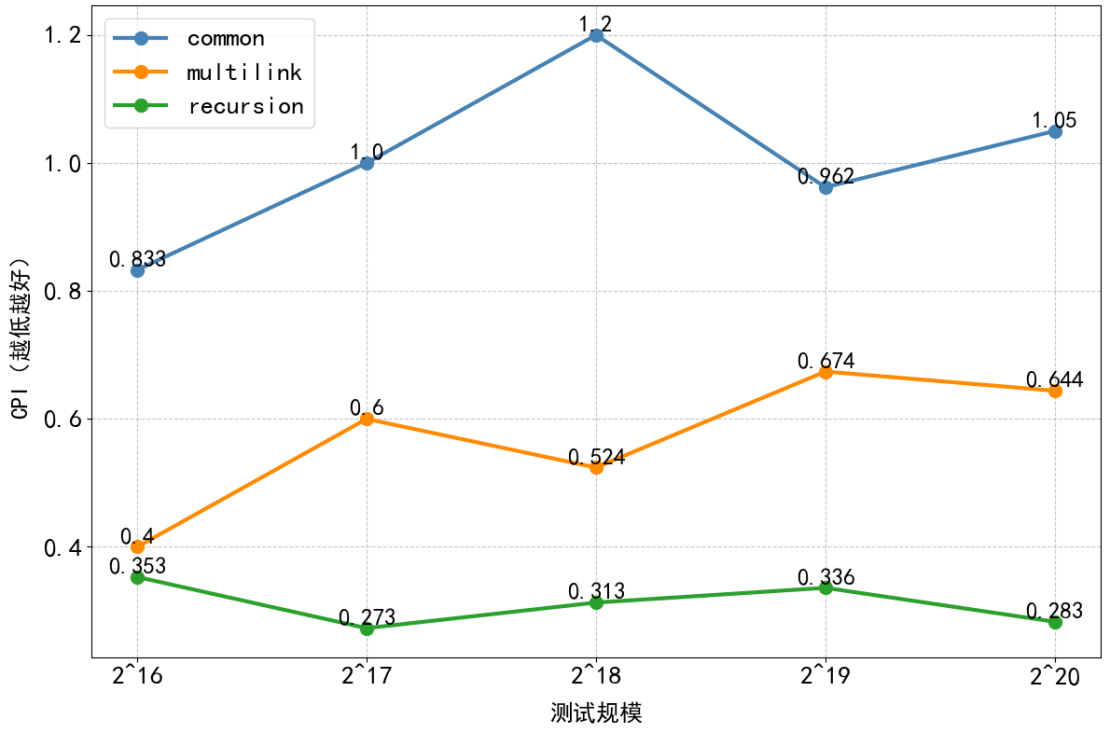

图 5: VTune 测试 CPI

(四) 关于耗时与 CPI 思考

在我的实验之中，相同规模下，递归的 CPI 比多链路优化算法低，但是递归的实际耗时比多链路高。这个现象比较反直觉，但是利用 VTune 进行更细致的分析，还是可以找到一些可能的解释 [2]。

1. VTune 分析指令总数差异 (进阶要求)

CPI 低表示每条指令平均所需的周期少，但是递归算法在实现上需要执行较多的指令（例如，更多的函数调用、状态保存/恢复、递归展开的额外操作），那么整体执行的指令总数可能远大于多链路优化算法。最终实际耗时是由“CPI × 指令总数 × 时钟周期时间”决定的，即使 CPI 低，总指令数高也会导致总耗时增加。

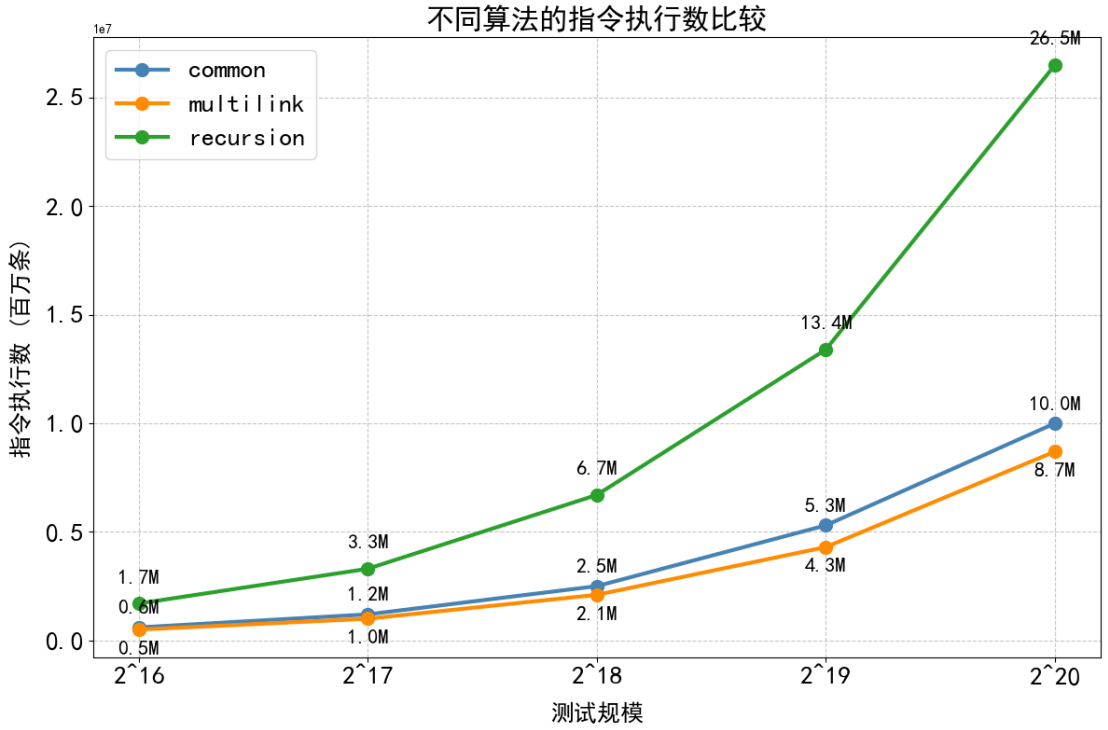

图 6: Instruction retired

在 VTune 中定位到了函数的指令执行数，实验结果表明，递归算法的指令执行数明显大于多链路算法和平凡算法（已经是倍数的关系），并且规模越大，增长率也变得不可控，因此，尽管递归算法的 CPI 比多链路算法低，但其总体执行时间仍然高于多链路算法。说明在大规模数据处理中，递归算法的性能瓶颈主要来自于过多的指令执行，这使得它不适合用于对效率有严格要求的大规模应用场景。

2. VTune 分析分支误预测率 (进阶要求)

递归算法的分支预测失误率较高，主要源于其执行路径的非线性叠加特性：每次递归调用产生独立的条件分支历史，不同深度的递归层在相同代码位置呈现差异化的分支行为，导致 CPU 基于局部历史或全局模式的预测机制失效。在 VTune 中测试分支误预测率时，发现平凡算法和双链路算法的分支误预测率时无法测出的，这说明它们的分支误预测率很小以至于 VTune 难以定位，但是递归算法的分支误预测率都在 20% 以上，并且极其不稳定，有时候甚至能到 80%，测试多次后发现，递归算法的分支误预测率平均在 20% 至 50%。如此高的失误率也是性能降低的原因之一。

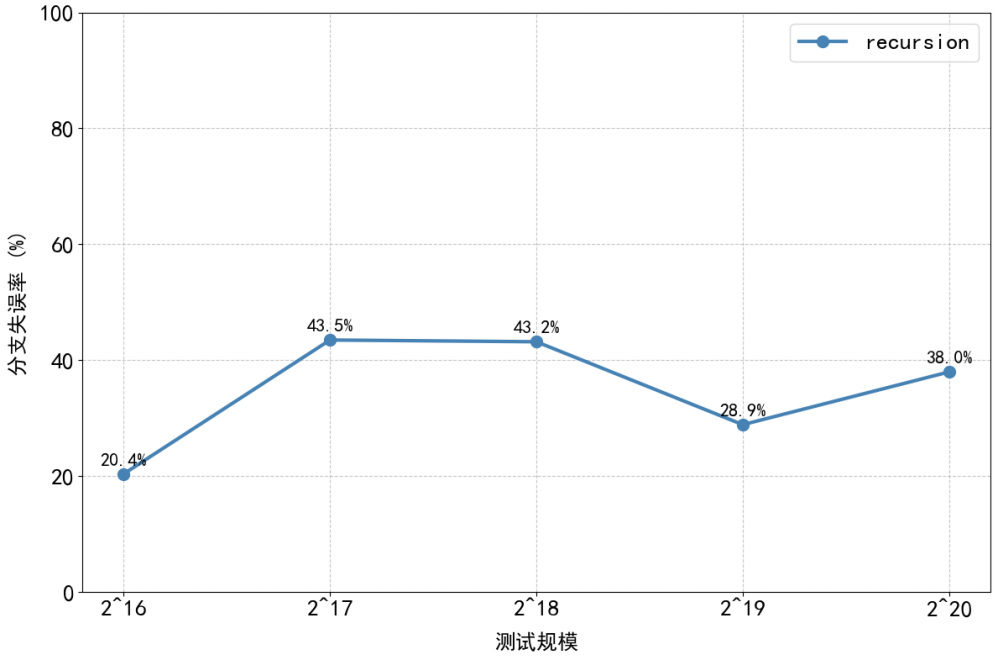

图 7: Bad Speculation

四、 总结

本次体系结构优化实验从矩阵向量内积和累加求和这两个典型场景出发，根据所用 CPU 的 cache、流水线等特性，编写了优化算法。除了对于性能及其相关指标的具体分析之外，研究还能得出一下具有启发意义的结论：

1. **算法复杂度 ≠ 执行效率 ≠ 实际耗时**：递归分治累加虽具备 O(logn) 的理论复杂度，但因频繁的函数调用开销，实际耗时反超 O(n) 的多链路算法。

2. **内存访问模式能决定性能瓶颈**：这是一个优化的新思路，不仅从理论的角度去优化算法，还可以根据实际硬件的特性去进行优化。本次实验得到的结论表明，内存访问模式的优化可能带来**数倍的性能提升**，收益远高于理论算法的优化。

3. **结合多种相关指标衡量算法的性能**：在累加实验中，VTune 分析的 CPI 值和实际耗时有矛盾。于是再次利用 VTune 细致分析其余相关指标，发现指令总数的差异和分支误预测率是解释这一现象的关键。因此在评价性能的时候，不能只局限单一评价指标，要综合多种指标进行分析。

跳转至：源代码地址，1.cpp 和 2.cpp 分别对应矩阵与向量内积和 n 个数求和的代码，平凡算法与优化算法以函数的形式编写在 cpp 文件中。

参考文献

[1] John L. Hennessy and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013.

[2] Intel Corporation. Intel vtune profiler, 2020. Available at https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[3] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2002.