南開大學
Nankai University

计算机学院

并行程序设计实验报告

Lab1:
体系结构相关实验及性能测试

姓名：陈语童

学号：2311887

专业：计算机科学与技术

2025 年 3 月 27 日

目录

1 实验环境

1.1 Windows x86 CPU 配置

Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz.
L1 Cache = 64KB/Core, L2 Cache = 256KB/Core, L3 Cache = 8MB Shared.

1.2 使用软件

Visual Studio 2022（编程）；Code::Blocks（编译测试）；VTune 2025（Profiling）.

2 基础实验

2.1 算法设计与代码实现

- **Cache 优化** 旨在提高数据局部性，让缓存（Cache）命中率提高，减少主存（DRAM）访问频次，达到 CPU 性能提高的目的。其原理是现代 CPU 访问 L1/L2/L3 三级缓存的速度比直接访问主存快十几甚至百倍；而缓存大小有限，访问可能引起大量缓存未命中（Cache Miss），降低性能。Cache 优化所做的是根据不同数据结构的空间存储策略（数据的空间局部性），以及优先考虑需要频繁访问的数据（时间局部性），合理地调整访问次序，让相邻访问局部性提高，从而降低未命中率，提高 CPU 效能。
- **超标量优化** 旨在通过指令重排、循环展开等策略，减少 CPU 指令之间的依赖并提高其并行度，从而让提高 CPU 指令吞吐量，让更多指令在一个 CPU 时钟周期内执行，从而相对地，减少了整个任务的执行周期，也能达到提高 CPU 性能的目的。其背后原理和逻辑更加复杂多样，但大多数应用包含重要技术如超标量流水线（Superscalar）和 SIMD（Single Instruct Mulitple Data，单指令多数据流）。本实验使用的方法较为浅显，本质上只是对高级程序逻辑上的改进。

2.1.1 Cache 优化：矩阵向量内积

题目重现

给定一个 $n \times n$ 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路：**a)** 逐列访问元素的平凡算法；**b)** Cache 优化算法。

平凡算法

一个朴素的想法是，依次计算每一列与向量的内积，将逐列访问矩阵元素、对应元素相乘和整个求和过程视为一个原子操作。

逐列访问元素的平凡算法的伪代码如下：

```
1: FOR each column i in b DO
2:     sum[i] ← 0
3:     FOR each row j in b DO
4:         sum[i] ← sum[i] + (b[j][i] * a[j])
5:     END FOR
6: END FOR
```

Cache 优化算法

进一步分析，C++ 对于矩阵的存储是有特定空间顺序的：按行紧密排列。因此，如果使用平凡算法，当矩阵的列数足够大时，每次存在缓存中的数据可能只有一行或不到一行，那么平凡算法一个原子操作每次都需要访问每一行的一个元素的带来的代价是矩阵行数规模的 Cache Miss，即几乎每次都需要访问内存，这使得 CPU 的运行效率大打折扣。

为了充分利用 C++ 存储矩阵方式的空间优势，我们考虑一种每次原子操作以行为单位的算法。很容易想到的一种策略是，每次原子操作完成一行每个元素与向量一个元素的乘积并存入相应的和数组元素中。这样的策略虽然存在跳转（找到相应的和数组元素）带来的额外开销，但对于 Cache Miss 来说是小许多的。

因此，我们设计一种逐行访问元素的 Cache 优化算法，伪代码如下：

```
1: FOR each column i in b DO
2:     sum[i] ← 0
3: END FOR
4: FOR each row j in b DO
5:     FOR each column i in b DO
6:         sum[i] ← sum[i] + (b[j][i] * a[j])
7:     END FOR
8: END FOR
```

2.1.2 超标量优化：简单求和

题目重现

计算 n 个数的和，考虑两种算法设计思路：**a)** 逐个累加的平凡算法（链式）；**b)** 适合超标量架构的指令级并行算法（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。

平凡算法

计算一个数组所有数据的和，最基础的方法就是遍历，依次加和到一个指定变量，求和。

这种逐个累加的平凡算法伪代码如下：

```
1: sum ← 0
2: FOR each element x in a DO
3:     sum ← sum + x
4: END FOR
```

超标量优化 1：多链路

超标量优化最朴素的一个策略，就是在单位时间内增加指令运行数量。那么，我们可以设法将多个加法在同一单位时间内运行。

最简单的是两个加法同时进行。保证两个操作同时进行的关键是，两个操作互不影响，不需要任何一方的先行完成另一方才能进行。那么我们采用简单的分治思想即可，设置两个副求和数 sum1 和 sum2，把数组每两个进行切割，前一个数加给 sum1，后一个数加给 sum2，最后进行 sum1 和 sum2 的一次简单加和即可求得总和 sum。这就形成了两条求和链路，起始分支，最后汇合。

以下是这种双链路超标量优化算法的伪代码：

```
1: sum1, sum2 ← 0
2: FOR i ← 0 TO N - 1 STEP 2 DO
3:     sum1 ← sum1 + a[i]
```

```
    sum2 ← sum2 + a[i + 1]
END FOR
sum ← sum1 + sum2
```

超标量优化 2：递归函数

另一个常见的想法是递归。我们对于给定数量的元素，两两相加，获得一次中间量；将获得的中间量两两相加，获得第二轮的中间量；将第二轮中间量两两相加再获得第三轮中间量..... 以此类推，每次中间量的个数都是上一轮的 1/2，易证，总的运算步数只需要 log(N)（每轮运算中两两相加同步进行的前提下）。

这个思路下的递归优化算法伪代码如下所示：

```
function RECURSIVE(n)
    IF n == 1 THEN
        return
    END IF
    FOR i FROM 0 TO (n / 2) - 1 DO
        a[i] ← a[i] + a[n - i - 1]
    END FOR
    Recursive(n / 2)
end function
```

然而，实际上这种“假”递归并不能实现计算效率的提高。原因在于，亦即此算法“假”在，每一轮递归中的加法并不是并行进行的，而是通过 for 循环依次运算的，这样计算下来，其运算步骤和平凡算法其实是一致的。再加上调用递归函数需要额外开销，其速度甚至比平凡算法还慢。稍后会在测试环节验证。

超标量优化 3：二重循环

递归的另一种实现，是通过循环。循环可以减省调用递归函数所需要的额外时间开销，理论上会比递归函数实现的递归算法更加高效，但相对于运算步骤数量对于效率的影响来说，这可能是微不足道的。

二重循环实现的超标量优化伪代码如下：

```
SET n = N
FOR n FROM N DOWNTO 2 DO
    SET half = n / 2
    FOR i FROM 0 TO half - 1 DO
        a[i] ← a[2 * i] + a[2 * i + 1]
    END FOR
    SET n ← n / 2
END FOR
```

同样，二重循环也没有解决加法依次而不是同时完成的问题——显而易见，这里的每一轮递归中的加法也是 for 循环实现的。

2.2 运行测试与结果分析

在个人电脑的 Windows x86 架构、Code::Block 配置好的 GCC 编译器环境下编译并运行以上各个程序，检测各个算法在不同参数下的运行时间，以运行时间为度量对比分析 CPU 性能提升效果。

2.2.1 运行 1：矩阵向量内积

取值 $n = [0, 100]$. 根据实际情况调整不同区间下的放缩度。测量结果如下：

N	Ordin.(ms)	Cache(ms)	N	Ordin.(ms)	Cache(ms)	N	Ordin.(ms)	Cache(ms)
10	0.0003	0.0003	250	0.1384	0.1328	2000	22.6717	8.8693
20	0.001	0.0011	300	0.2163	0.1885	2500	48.0231	13.8166
30	0.002	0.002	350	0.3265	0.2781	3000	58.6494	19.0348
40	0.0038	0.0041	400	0.4592	0.3661	3500	95.8848	26.5411
50	0.0061	0.006	450	0.6773	0.3862	4000	106.46	34.7025
60	0.0078	0.0085	500	1.0503	0.5538	4500	159.235	43.3376
70	0.0104	0.0101	600	1.2262	0.7643	5000	180.198	54.7634
80	0.0144	0.0157	700	2.1608	1.1747	6000	259.948	77.7282
90	0.0181	0.0176	800	2.4694	1.5118	7000	382.035	105.36
100	0.0241	0.0247	900	3.5184	2.2053	8000	576.57	137.873
150	0.049	0.0478	1000	3.9585	2.5619	9000	672.139	176.945
200	0.0964	0.0926	1500	12.1395	5.2834	10000	794.792	215.556

表 1: 平凡算法和 Cache 优化算法运行时间数据统计（多次实验求均值结果）

根据运行测试获得的数据，绘制折线图观察趋势：

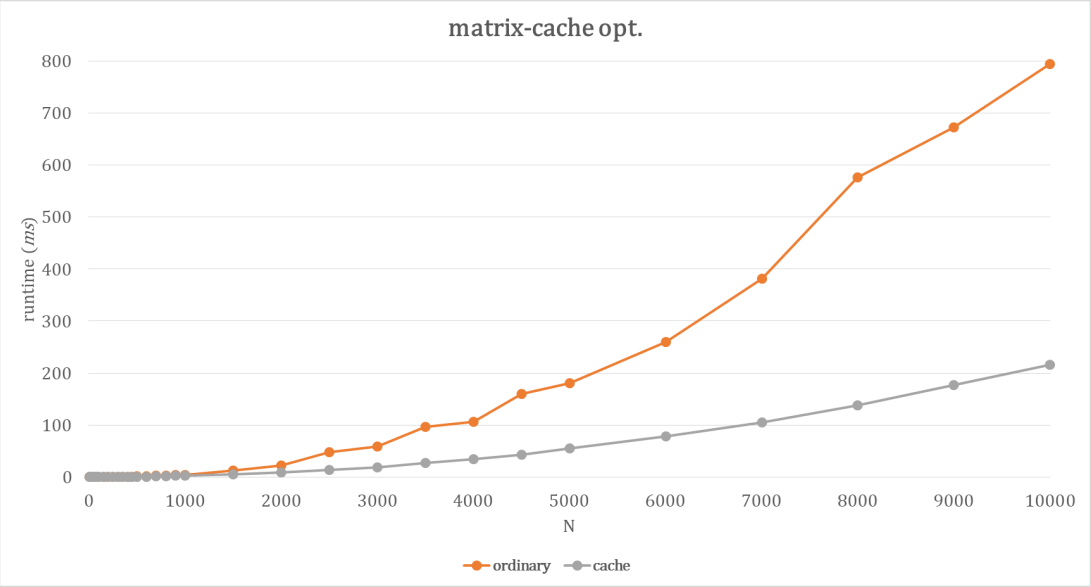

图 2.1: 平凡算法和 Cache 优化算法的运行时间变化趋势图

可以看到，Cache 优化算法的运行时间明显小于平凡算法，且优势随着 N 的增大而变得更加显著。说明优化有效。

2.2.2 运行 2：简单求和

取值 $N = [2^{10}, 2^{25}]$. 结果如下：

$N = 2^?$	Ordinary(ms)	Double Chain(ms)	Recursion(ms)	Cycle(ms)	Double Cycle(ms)
10	0.0016	0.001	0.0026	0.0025	0.0029
11	0.0034	0.0021	0.0038	0.0037	0.004
12	0.0071	0.0044	0.0073	0.0072	0.0085
13	0.0139	0.0086	0.015	0.0155	0.0187
14	0.0276	0.0197	0.0356	0.0343	0.0389
15	0.0557	0.0336	0.0574	0.056	0.0645
16	0.1106	0.0653	0.1264	0.122	0.1312
17	0.2268	0.138	0.2531	0.266	0.2805
18	0.4589	0.26	0.4904	0.4822	0.659
19	0.9657	0.5203	1.0279	1.207	1.2369
20	1.8588	1.0046	2.846	3.0225	1.954
21	3.6583	1.892	3.7203	3.7027	4.0328
22	7.862	3.9523	8.368	8.4103	10.007
23	15.5327	8.0891	16.6407	17.34	21.2744
24	30.6247	18.0065	40.971	38.1239	43.2223
25	63.0402	38.874	69.9312	74.4536	91.3058

表 2: 平凡算法和各种优化版本算法运行时间数据统计（多次实验求均值结果）

同样绘制折线图：

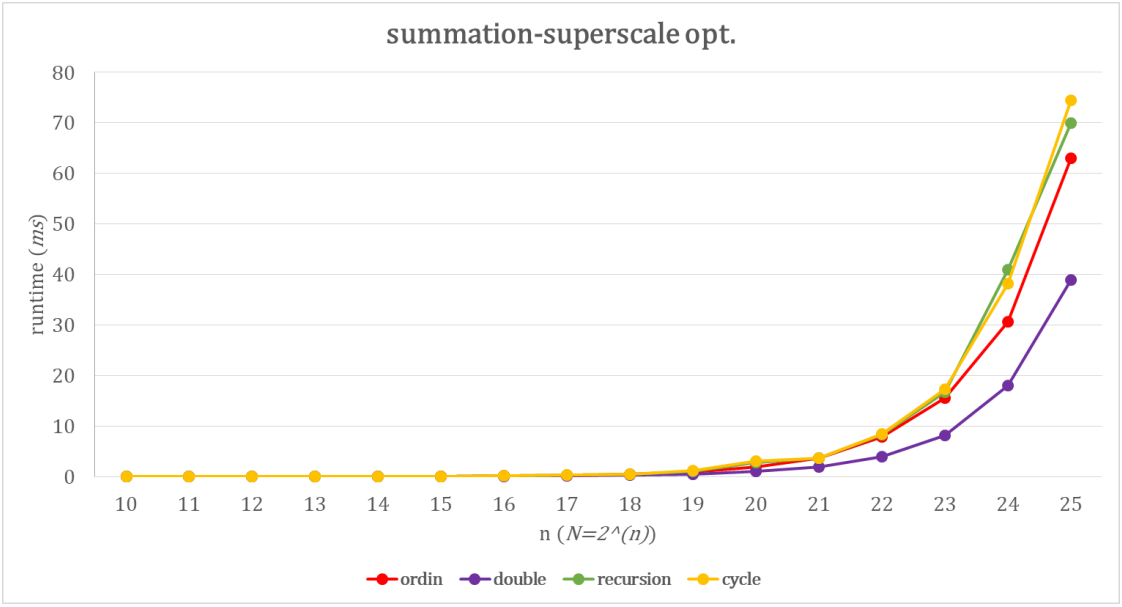

图 2.2: 平凡算法和各种优化版本算法的运行时间变化趋势图

观察到双链路超标量优化有较为显著的性能提升效果——N 足够大以后其运行时间可以缩减到平凡算法的近一半，这与其同一单位时间压缩运行两条指令的改进策略在逻辑上契合。

而除此以外，其他“优化”并没有性能提升的效果，反而在 N 提升后运行时间可见且稳定地超过了平凡算法的运行时间。这证明了前面设计算法中的猜想——递归的策略在代码编写后并没有实现循

环内的并行，从而无法提升性能。而递归函数和循环方法实现递归策略超出平凡算法的运行时间，分别是调用函数所需的额外开销和外层循环的额外开销。

2.3 VTune profiling

下面通过 Intel VTune 工具获得程序运行指标进一步分析性能改变的原理。

2.3.1 分析 1：矩阵向量内积的 Cache 命中分析

选择 $N = 100, 1000, 10000$ 进行 VTune Hotspot 测试，对比分析 Hardware Events 中的三级 Cache Hit 和 Cache Miss. 结果如表所示：

$N = 100$

Ordinary	L1 Hit	4.2×10^7	0.50%	0	6.6×10^7	L1 Hit	Cache opt.
	L1 Miss	2.1×10^5			2.0×10^3	L1 Miss	
	L2 Hit	2.0×10^5	2.70%	5.30%	1.8×10^3	L2 Hit	
	L2 Miss	5.6×10^3			1.0×10^2	L2 Miss	
	L3 Hit	1.0×10^2	0	0	9.5×10^1	L3 Hit	
	L3 Miss	0			0	L3 Miss	

$N = 1000$

Ordinary	L1 Hit	1.4×10^8	6.7%	0.03%	2.0×10^8	L1 Hit	Cache opt.
	L1 Miss	1.0×10^7			6.3×10^4	L1 Miss	
	L2 Hit	8.2×10^6	2.7%	9.8%	6.0×10^4	L2 Hit	
	L2 Miss	2.3×10^5			6.5×10^3	L2 Miss	
	L3 Hit	1.5×10^5	1.2%	0.70%	2.4×10^3	L3 Hit	
	L3 Miss	1.8×10^3			1.8×10^1	L3 Miss	

$N = 10000$

Ordinary	L1 Hit	1.5×10^8	15.3%	0.06%	1.8×10^8	L1 Hit	Cache opt.
	L1 Miss	2.7×10^7			1.0×10^5	L1 Miss	
	L2 Hit	1.4×10^7	24.3%	7.6%	9.3×10^4	L2 Hit	
	L2 Miss	4.5×10^6			7.6×10^3	L2 Miss	
	L3 Hit	2.5×10^6	28.6%	17.6%	1.4×10^3	L3 Hit	
	L3 Miss	1.0×10^6			3.0×10^2	L3 Miss	

表 3: 不同 N 取值下平凡算法和 Cache 优化算法的 Cache 命中和未命中

表中间部分的数据是每组（Hit+Miss）数据对应的该级缓存未命中率。

观察发现随着 N 数量级的提升，相比于平凡算法，Cache 优化算法的缓存未命中率明显较低。当 N 在 10^3 级别以下时，两者三级缓存未命中率都较低，影响与差异不是特别大；而当 N 达到 10^4 数量级的时候，平凡算法的各级缓存未命中率明显高于优化算法，L2 Cache Miss 率高达 24.3%、L3 Cache Miss 率高达 28.6%，这使得其需要更多地直接访问内存来完成任务；相反地，此时优化算法的优势就显现出来。

2.3.2 分析 2：简单求和的超标量分析

对简单求和的链式加和平凡算法和各种优化算法进行超标量测试，对比指令退役量（Instructions Retired）和指令周期（CPI Rate）两个指标并分析算法效率效益。取一个较大的 $N = 2^n$. 结果如下表所示：

Algo.	Instructions Retired	CPI Rate	Inst.Ret. $\times$ $CPIRate$
Ordinary	4.356×10^8	0.583	2.54×10^8
DoubleCycle	7.722×10^8	0.38	2.93×10^8
Recursive	8.065×10^8	0.335	2.70×10^8
Cycle	8.045×10^8	0.336	2.70×10^8
DoubleChain	3.69×10^8	0.376	1.39×10^8

表 4: 不同

通过指令数与 CPI Rate 形成可以获得整个程序运行所需 CPU 周期，可以发现双链路优化有着明显更少的周期，大约只是平凡算法的一半，而其他未成功的优化所需周期都多于平凡算法。这验证了我们之前对于直接运行程序的分析。

相较于平凡算法，双链路优化的优势体现在 CPI Rate 的大幅减低，这是因为它同时进行两个加法操作的策略提升了程序并行性；而相较于其他未成功的优化，双链路优化的优势则体现在总指令数量，这是因为它不需要额外的循环或者函数调用开销。

3 进阶实验

3.1 更多优化

3.1.1 Cache 优化：循环展开（Unroll）

我们可以在逐行运算算法的基础上再进行优化，获得更佳的 Cache 优化算法。

这里使用循环展开的思路，采取两个策略：

1. 预加载一行数据，进一步提高缓存命中率
2. 每次循环进行两次运算，即循环展开，加强程序并行性

此外，还可以将当前向量元素存为临时变量，削减过程中反复访问所需要的额外开销。

综合以上，循环展开的伪代码如下所示：

```
1: FOR j FROM 0 TO N-1 DO
2:     tmp ← a[j]
3:     bj ← b[j]
4:     FOR i FROM 0 TO N-1 STEP 2 DO
5:         sum[i] ← sum[i] + (bj[i] * tmp)
6:         sum[i+1] ← sum[i+1] + (bj[i+1] * tmp)
7:     END FOR
8: END FOR
```

为了节省报告空间，不展示详细运行数据，直接展示加入循环展开后的运行时间折线图：

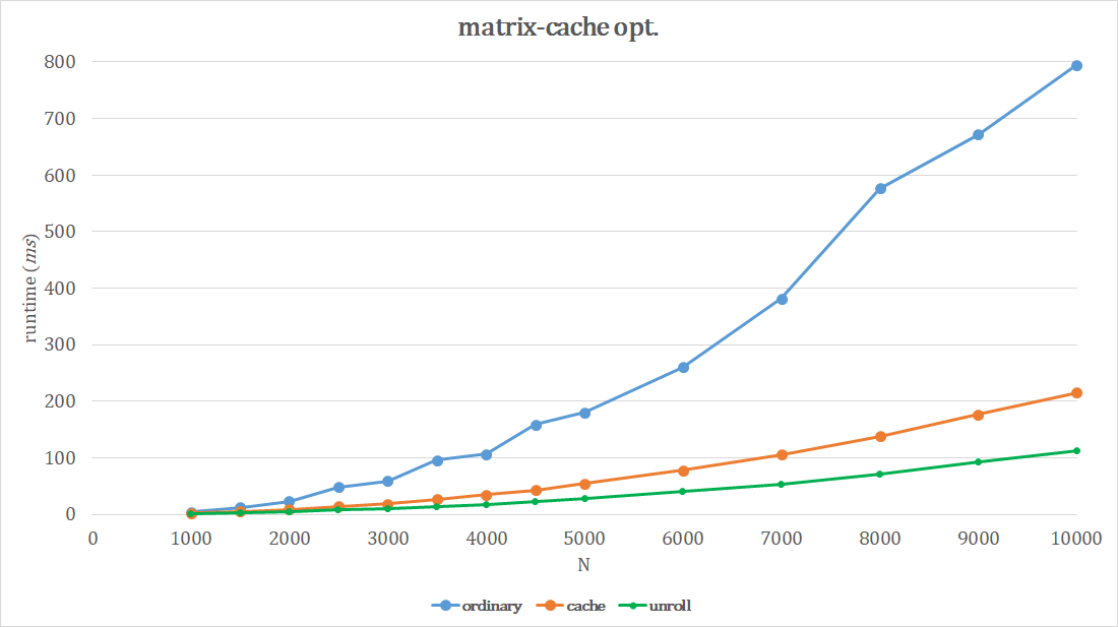

图 3.3: 加入 Unroll 之后的运行时间对比折线图

可以看到，相比于常规按行计算算法，循环展开算法几乎将运行时间又缩减了 1/2，这与其循环内压缩运行两次乘法和增量运算在逻辑上是契合的。

3.1.2 超标量优化：AVX 指令 SIMD 化递归

前面我们分析，原先采用的递归思想和编写的相关程序并没有起到效率提高的目的，其根本原因在于未实现循环内部的并行运算。那么，可以尝试通过一些特定并行指令实现 SIMD，进而充分发挥递归作用提升性能。

（注：*AVX 指令的使用方法为咨询 ChatGPT 所获取*）

AVX 指令（Advanced Vector Extensions），是一组由 Intel 和 AMD 引入的 SIMD（单指令多数据）指令集扩展，可提供 8 个数据的同时计算处理，提高浮点和整数运算的并行计算能力。

为了简明地展示，此处直接展示采用 AVX 方法的改良版本循环策略实现的递归优化中的关键代码：

```
1     for (int n = N; n > 1; n /= 2) {
2         int half = n / 2;
3         int i = 0;
```

```
        for (; i <= half - 8; i += 8) {
            __m256i va = _mm256_loadu_si256((__m256i*)&a[i]);
            __m256i vb = _mm256_loadu_si256((__m256i*)&a[n - i - 8]);
            va = _mm256_add_epi32(va, vb);
            _mm256_storeu_si256((__m256i*)&a[i], va);
        }
        for (; i < half; i++) {
            a[i] += a[n - i - 1];
        }
    }
```

代码关键在于使用了 _mm256_loadu 和 _mm256_loadu 等一系列 AVX 核心代码，能够指导一次性载入多数据，而后同时进行运算。

此方法能够充分发挥递归策略的优势，达到与双链路基本持平甚至更高的效率：

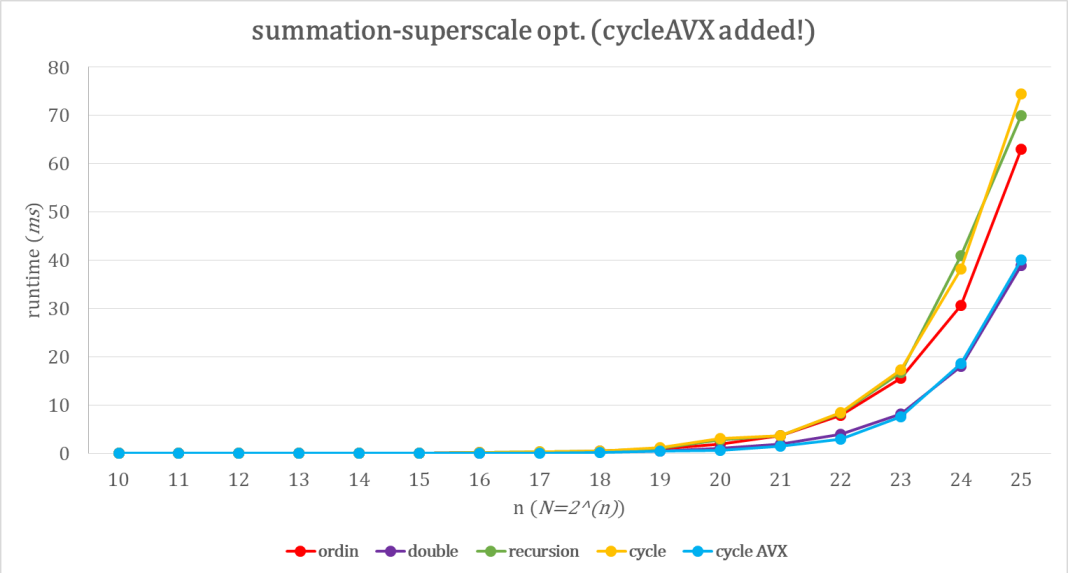

图 3.4: 加入 Cycle AVX 优化的运行时间对比折线图

对此优化也进行 VTune 的超标量测试，$InstructionsRetired = 1.548 \times 10^8$, $CPIRate = 0.64$. 计算得总指令用时只需 0.99×10^8 个 CPU 周期，优于之前任何一种优化。

4 总结思考

本实验通过两个非常简单的题目探究了体系结构性能提升上的方法与策略。最主要的思想有二：一是考虑数据存储的空间特点，并利用其特点制定访问策略提高 Cache 命中率，即 Cache 优化；一是循环展开，在一次循环中进行多个互不依赖的操作，达到并行目的，即超标量优化。在需要访问大量数据的任务中，Cache 优化作用压倒性地显著，需要优先考虑；而成功的超标量优化在一切可以设计成并行进程的任务中，均至少有整 1 倍的效率提升。

此实验还学习了基于 Intel VTune 的 Profiling 技巧，这对于日后自我编写的大体量工程进行效率测试和优化有着很大帮助。此次实验只进行了 x86 而未进行 ARM 架构上的性能测试，下次将尝试。

本次实验所有代码均已上传至[GitHub]