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

体系结构相关实验及性能测试

林晖鹏

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 28 日

摘要

本文在阿里云 ARM 架构 ECS 实例（4 核 16GB，Yitian 710 芯片），运行 Linux 系统，使用 g++ 11.3 编译器，固定 CPU 性能模式进行测试。通过设计矩阵向量内积和数组求和两类基础算法实验，系统探究了不同代码优化策略对程序性能的影响。实验采用高精度计时和 Linux perf 工具，定量分析了缓存优化、循环展开、递归改写等方法在 ARM 架构下的性能表现。在 O0 优化下测试，测试结果表明：通过合理的算法重构，构建更符合并行架构的代码，考虑数据的复用率、缓存的局部性，能大大提升程序的性能。研究同时揭示了编译器优化级别（O0-O3）对性能指标的显著影响，发现编译器优话能将性能增强百倍，而 O3 优化在保持指令数不变情况下仍能通过依赖链优化进一步降低 5.4% 的周期数。本实验为编写高性能计算代码提供了实证性的优化指导。

关键字：Parallel 性能优化、缓存局部性、指令级并行、编译器优化、ARM 架构

目录

一、 高精度计时测试时间

对于实验中的程序，我们在 linux 系统下使用 clock_gettime 函数来测试精确到纳秒级的执行时间，并且通过重复执行求平均值来获得平均执行时间。代码如下：

```
double get_elapsed_time(timespec start, timespec end){
  return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}

for (int n = 0; n <= 1000; n += step)
{
timespec start_time, end_time;
clock_gettime(CLOCK_MONOTONIC, &start_time);
int counter = 0;
while (true){
  counter++;
  ***算法执行***
  clock_gettime(CLOCK_MONOTONIC, &end_time);
  if (get_elapsed_time(start_time, end_time) >= 0.1)
    break; // 至少 0.1s
}
double total_time = get_elapsed_time(start_time, end_time);
cout << n << "␣" << total_time / counter << endl;
if (n == 100)step = 100;
}
```

二、 n*n 矩阵与向量内积

(一) 实验要求

给定一个 n × n 矩阵，计算每一列与给定向量的内积。用不同算法实现，并且用高精度计时测试程序执行时间，比较并分析不同算法的性能。

(二) 算法设计及编程实现

1. 逐列访问元素的平凡算法

算法设计：用一个双层循环计算结果。外层循环迭代矩阵的每一列；内层循环迭代矩阵中某列的每个元素与向量的每个元素的乘积，得到结果向量的某个元素。

```
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++)sum[i] += b[j][i] * a[j];
  }
```

2. cache 优化算法

算法设计：用一个双层循环计算结果。外层循环迭代矩阵的每一行；内层循环迭代矩阵中某行中的每个元素与向量的每个元素的乘积，得到目标向量每个元素的部分积，并加到目标向量中对应元素上面。

```
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++)sum[j] += b[i][j] * a[i];
  }
```

3. unroll 优化算法

算法设计：在前面 cache 优化算法的基础上，每次内层循环计算两个甚至四个数，来减少循环次数，从而减少了循环判断条件的执行次数。在每次内层循环之后，需要 if 判断有没有漏计算的元素。

```
for (int i = 0; i < n; i++){
    int j = 0;
    for (j = 0; j < n - 3; j += 4){
      sum[j] += b[i][j] * a[i];
      sum[j+1] += b[i][j+1] * a[i];
      sum[j+2] += b[i][j+2] * a[i];
      sum[j + 3] += b[i][j + 3] * a[i];
    }
    if (j == n - 5)sum[n-1] += b[i][n - 1] * a[i];
    else if (j == n - 6){
      sum[n-2] +=b[i][n - 2] * a[i];
      sum[n - 1] += b[i][n - 1] * a[i];}
    else if (j == n - 7){
      sum[n - 3] += b[i][n - 3] * a[i];
      sum[n - 2] += b[i][n - 2] * a[i];
      sum[n - 1] += b[i][n - 1] * a[i];}
    }
```

(三) 性能测试

我们对 n=10～10000 的部分情况进行了算法性能测试，由于计算结果不稳定，我们对于每个 n 的循环总时长设置为 1 秒。结果如表2

表 1: 算法执行时间对比（单位：s）

输入规模 (n)	原始算法	缓存优化	循环展开 + 缓存
0	$5.087\,81\times10^{-8}$	$5.095\,33\times10^{-8}$	$5.130\,57\times10^{-8}$
50	$5.988\,57\times10^{-6}$	$5.488\,54\times10^{-6}$	$4.926\,44\times10^{-6}$
100	$2.388\,96\times10^{-5}$	$2.174\,00\times10^{-5}$	$2.005\,67\times10^{-5}$
500	$6.013\,24\times10^{-4}$	$5.631\,61\times10^{-4}$	$4.999\,00\times10^{-4}$
1000	$2.353\,26\times10^{-3}$	$2.254\,59\times10^{-3}$	$2.016\,89\times10^{-3}$
5000	$5.874\,05\times10^{-2}$	$5.553\,66\times10^{-2}$	$5.055\,05\times10^{-2}$
10 000	$2.388\,89\times10^{-1}$	$2.211\,04\times10^{-1}$	$1.980\,19\times10^{-1}$

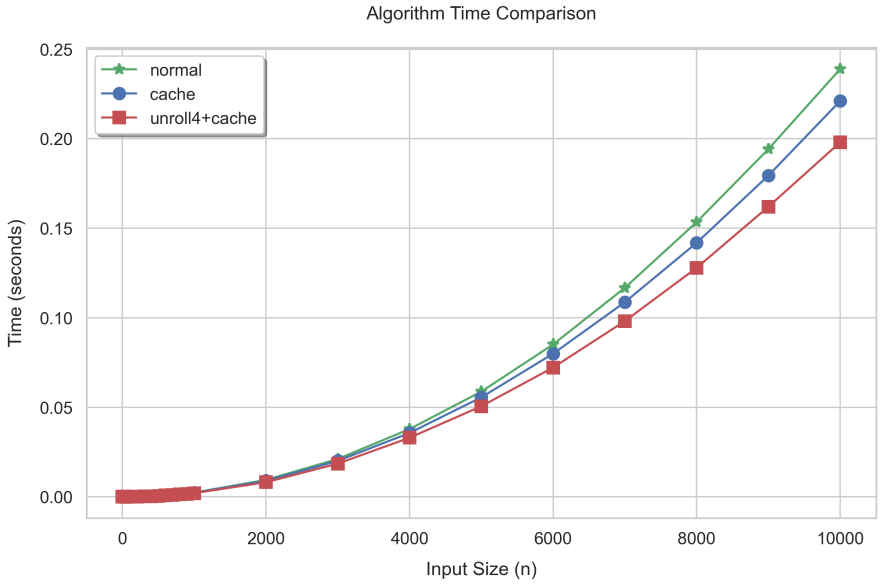

图 1: 性能对比

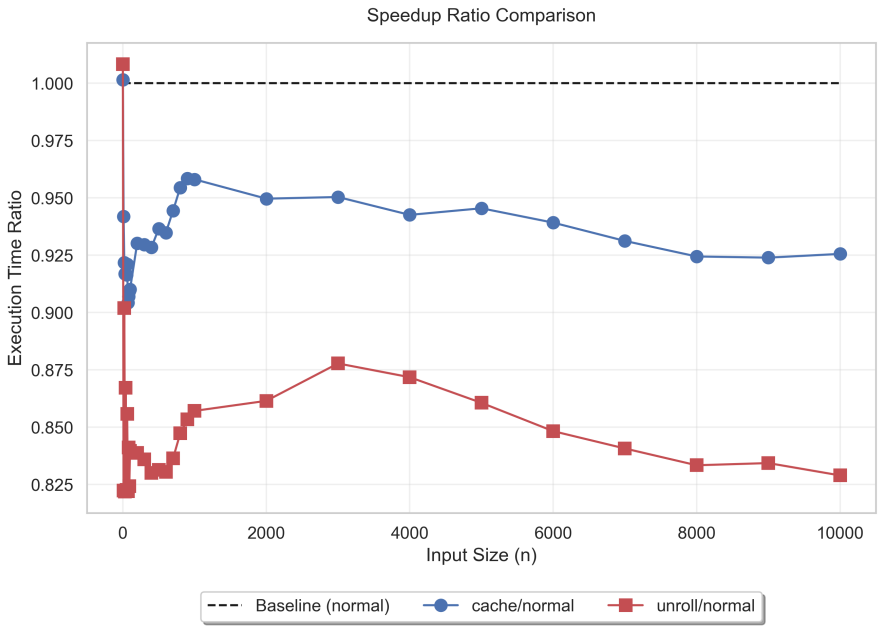

图 2: 执行时间比率

(四) 结果分析

在 n 规模比较小的时候，性能差别不大，此时主要是系统本身占用时间较多，并且时间不稳定。

cache 优化整体为平凡算法 92.5～95%，而 unroll4+cache 优化为平凡算法的 87.5～82.5% 左右，而且随着 n 的变大，这个比率进一步变小，可见这些优化有效缩短了执行时间。

对于 cache 算法，虽然代码执行数和循环周期数不变，但是单次加载的数据的复用率提高了，减少了内存中寻找数据的次数，从而提高效率。

而对于 unroll 算法，一方面每个周期执行数变大导致周期数减少，每个周期的判断条件也

减少次数；另一方面同一周期内的指令可以并行计算，从而加快效率。

三、 n 个数求和

(一) 实验要求

给定 n 个元素的数组，计算这个数组的求和。用不同算法实现，并且用高精度计时测试程序执行时间，比较并分析不同算法的性能

(二) 算法设计及编程实现

1. 单链路式算法

算法设计：循环迭代每个元素，进行求和。

```
for (int i = 0; i < n; i++)
  sum = sum+a[i];
```

2. 多链路式算法

算法设计：类似于上一个实验的 unroll 做法，这里分别实现 **unroll2** 和 **unroll4**，多个链路能并行计算，从而减少时间。

```
//unroll2
  int i = 0, sum1 = 0, sum2 = 0;
  for (i = 0; i < n - 1; i += 2)
  {
    sum1 = sum1 + a[i];
    sum2 = sum2 + a[i + 1];
  }
  if (i == n - 3) sum = sum + a[n - 1];
  sum = sum1 + sum2;
```

```
//unroll4
  int sum1 = 0, sum2 = 0,sum3=0,sum4=0, i=0;
  for (i = 0; i < n-3; i += 4){
    sum1 = sum1 + a[i];
    sum2 = sum2 + a[i+1];
    sum3 = sum3 + a[i+2];
    sum4 = sum4 + a[i+3];
  }
  if(i == n-5) sum+=a[n-1];
  else if(i == n-6) sum+= a[n-1]+a[n-2];
  else if (i == n - 7) sum += a[n - 1] + a[n - 2]+a[n-3];
  sum = sum1 + sum2 +sum3+sum4;
```

3. 归并优化算法

算法设计：采用归并的思想，两两相加，并行执行，一方面复杂度变成了 $logn$, 并且同时能有效利用并行来加快效率。这里我们实现**二重循环**和**递归**两种做法：

```
//二重循环
  for (int m = n; m > 1; m = (m + 1) / 2){
    for (int i = 0; i < m / 2; i++) a[i] += a[m - i - 1];
  }
  sum = a[0];
```

```
//递归优化
void recursion(int n, int a[]){
      if (n == 1 || n == 0) return;
      else{
        for (int i = 0; i < n / 2; i++) a[i] += a[n - i - 1];
        n = n / 2;
        recursion(n, a);
      }
    }
  recursion(n, a);
  sum = a[0];
```

(三) 性能测试

我们对 n=10～5000 的部分情况进行了算法性能测试，由于计算结果不稳定，我们对于每个 n 的循环总时长设置为 1 秒。部分结果如表所示：

表 2: 算法执行时间对比（单位：s）

输入规模 (n)	单链路式	unroll2	unroll4	二重循环	递归优化
0	5.13×10^{-8}	5.36×10^{-8}	5.27×10^{-8}	5.12×10^{-8}	5.27×10^{-8}
50	1.56×10^{-7}	1.09×10^{-7}	8.98×10^{-8}	1.75×10^{-7}	1.75×10^{-7}
100	2.73×10^{-7}	1.71×10^{-7}	1.36×10^{-7}	3.15×10^{-7}	3.10×10^{-7}
500	1.21×10^{-6}	6.73×10^{-7}	4.85×10^{-7}	1.46×10^{-6}	1.40×10^{-6}
1000	2.37×10^{-6}	1.29×10^{-6}	9.24×10^{-7}	2.87×10^{-6}	2.79×10^{-6}
3000	7.00×10^{-6}	3.77×10^{-6}	2.65×10^{-6}	8.58×10^{-6}	8.38×10^{-6}
5000	1.16×10^{-5}	6.24×10^{-6}	4.36×10^{-6}	1.42×10^{-5}	1.40×10^{-5}

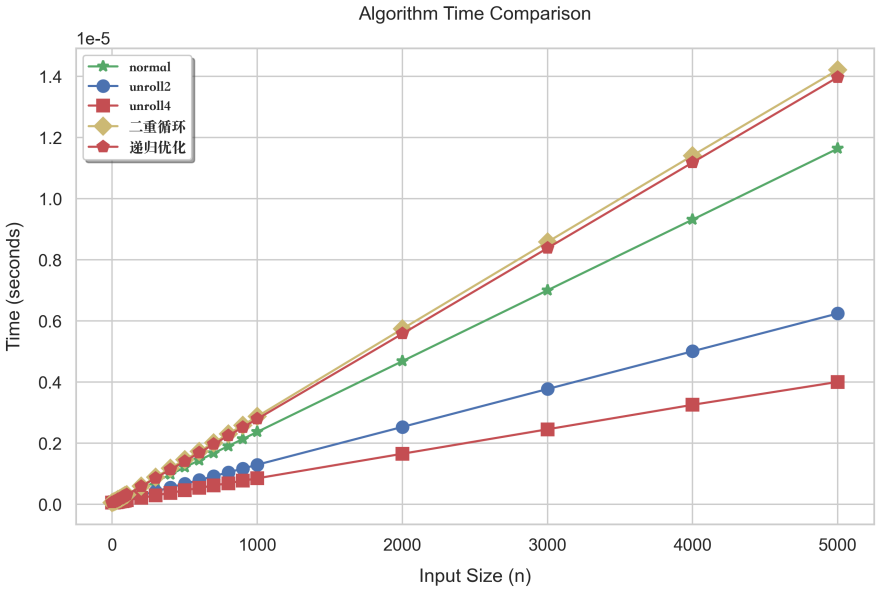

图 3: 性能对比

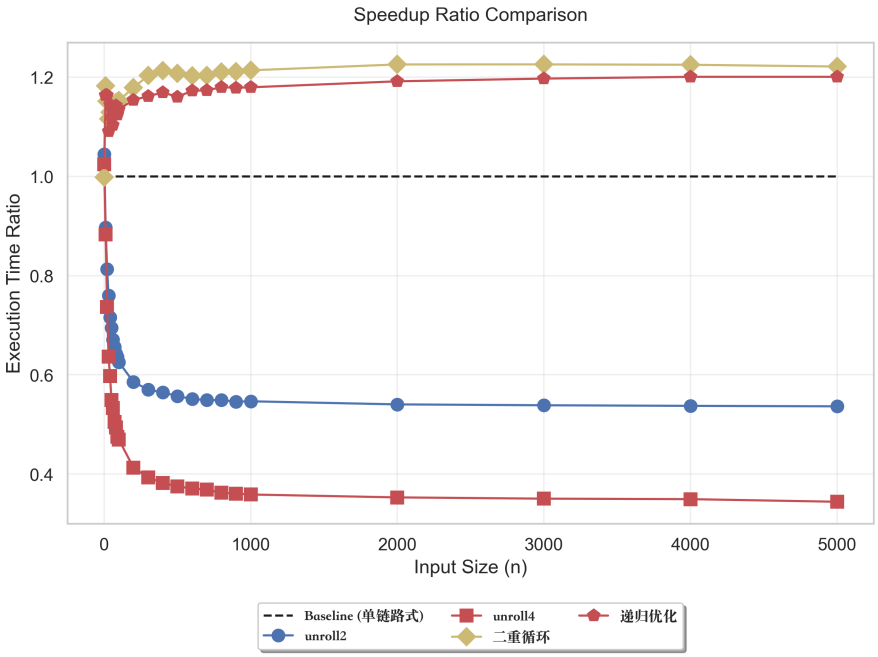

图 4: 执行时间比率

(四) 结果分析

对于 unroll 算法，unroll2 为单链路式的 55% 左右，而 unroll4 更是达到了 0.4%。unroll 算法一方面减少了循环周期数，另一方面，每个周期内的指令可以并行计算，从而大大加快了效率。

对于归并优化算法，虽然可以并行计算来提高效率，但是计算次数会导致翻倍；其次，每次求和的时候选取元素的位置较远，比顺序求和访问数据效率更慢，缓存命中率应该更低；再者每次都会重新写入数组，导致内存读写开销变大。多种原因下导致效率更差。

四、 profiling

(一) 评测结果

我们用 **perf** 来对程序进行缓存命中率和指标量的测量。这里测试的是实验 1 在 n=10000 的规模下的程序以及实验 2 在 n=5000 的规模下的程序，我们测试的指标量为：l1d_cache、l1d_cache_refill、l2d_cache、l2d_cache_refill、cycles、instructions 结果如表3所示。

表 3: 算法测试指标对比

算法	L1 缓存命中率	L2 缓存命中率	IPC
平凡算法	99.79 %	97.15 %	3.17
缓存优化	99.80 %	95.05 %	3.27
循环展开 + 缓存	99.76 %	96.15 %	3.38
单链路式	98.13 %	86.62 %	1.17
unroll	98.13 %	86.53 %	1.16
unroll	98.12 %	87.72 %	1.27
二重循环	98.16 %	87.27 %	1.19
递归优化	98.16 %	87.54 %	1.12

注：L1/L2 命中率通过公式 $\left(1-\frac{\text{refill}}{\text{access}}\right)\times 100\%$ 计算；IPC 为每周期指令数。

(二) 结果分析

对于第一个实验，L1 缓存命中率差别并不大。平凡算法的 L2 缓存命中率比较高可能是因为 L2 中存了 a 向量的值，每次外层循环都会每个用一次。循环展开 + 缓存算法的 L2 命中率优于缓存优化算法，说明展开操作改善了空间局部性。缓存优化算法的 L2 命中率反而最低，可能是因为” 写入分配” 策略导致缓存污染。

而对于第二个实验，归并优化的 L1L2 缓存命中率较另外三个稍微高点，可能是重复在使用前一半的 a 进行读写操作，但同时 IPC 较低，总体性能太低因为访问元素时候跨度大导致开销变大。而 unroll 优化无论在缓存命中率还是 IPC，都有所提高。

五、 自动优化对比

在前面的实验中，为了客观对比算法的优劣，我们都是采用 O0 优化，在本节中，我们在实验一的平凡算法在 n=10000 规模下对各种自动优化进行实验，并对比结果。结果如表4所示。

结果分析:

- **O0 异常高 IPC**：编译器未优化生成冗余指令，冗余指令导致虚假并行。
- **O1-O3 优化效果**：指令数减少 99.9%，耗时降低 400 倍
- **O3 较 O2 优势**：相同指令数下周期数减少 5.4%，O3 优化并预取了依赖链。

我们可以发现 O0 的 IPC 异常高并且缓存命中率也很高，但是是因为编译器未优化时生成冗余指令，导致并行了许多冗余指令，多次访问了不必要的数据，所以指令数和耗时也很大。

而随着逐步优化，我们发现 IPC 越来越少，编译器优化时候减少了很多执行指令数。O2 和 O3 相比，虽然指令数相差不大，但是 O3 优化进一步减少了周期数，耗时也更短。

表 4: 不同编译优化级别下的性能指标对比

优化级别	缓存命中率		IPC	周期数	指令数	耗时 (s)
	L1	L2				
O0	99.79%	97.15%	3.17	1.71e9	5.43e9	0.609
O1	98.08%	87.92%	1.98	2.04e8	4.04e8	0.069
O2	98.03%	86.43%	1.14	2.91e6	3.32e6	0.00172
O3	98.06%	87.62%	1.21	2.75e6	3.32e6	0.00152

注：L1 命中率计算：$(1 - \frac{\text{l1d\_cache\_refill}}{\text{l1d\_cache}}) \times 100\%$，IPC 是每周期指令数

六、 总结

本实验对不同算法下的性能指标进行研究，发现：

- 在矩阵-向量乘法中,循环展开结合缓存优化的算法表现最佳,相比朴素实现性能提升 18.4%，L2 缓存命中率达到 96.15%，IPC 提升至 3.38。
- 在数组求和中，unroll4 优化方案性能最优，执行效率达到单链路式的 2.7 倍，验证了指令级并行的重要性。
- 编译器优化效果显著，O3 优化相比 O0 实现 400 倍加速，即使在相同指令数下仍可通过指令重排和预取技术降低 5.4% 的周期数。
- 算法优化需结合硬件特性，在 ARM 架构下，内存访问连续性和缓存局部性是影响性能的关键因素。建议对计算密集型任务采用循环展开 4-8 次，并配合-O3 -march=native 编译选项。
- 需注意避免过度优化，如递归改写虽理论复杂度低，但因缓存访问模式不佳实际性能可能下降。