



南開大學  
Nankai University

计算机学院  
体系结构调研报告

CPU 架构相关编程

姓名：郭允轩

学号：2311833

专业：计算机科学与技术

2025 年 3 月 30 日

# 目录

|   |          |
|---|----------|
| <b>1 实验配置</b>                           | <b>2</b> |
| 1.1 硬件配置 . . . . .                      | 2        |
| 1.2 软件配置 . . . . .                      | 2        |
| <b>2 基础实验</b>                           | <b>2</b> |
| 2.1 实验一: $n \times n$ 矩阵与向量内积 . . . . . | 2        |
| 2.1.1 算法设计 . . . . .                    | 2        |
| 2.1.2 编程实现 . . . . .                    | 2        |
| 2.1.3 性能测试 . . . . .                    | 4        |
| 2.1.4 profiling . . . . .               | 5        |
| 2.1.5 结果分析 . . . . .                    | 6        |
| 2.2 实验二: $n$ 个数求和 . . . . .             | 6        |
| 2.2.1 算法设计 . . . . .                    | 6        |
| 2.2.2 编程实现 . . . . .                    | 6        |
| 2.2.3 性能测试 . . . . .                    | 7        |
| 2.2.4 profiling . . . . .               | 8        |
| 2.2.5 结果分析 . . . . .                    | 8        |
| <b>3 进阶实验</b>                           | <b>8</b> |
| <b>4 总结</b>                             | <b>9</b> |

### Abstract

本报告主要介绍了两个实验：矩阵与向量内积和  $n$  个数求和，分别实现了平凡算法和 cache 优化算法，适合超标量架构的指令级并行算法，并且绘制表格和图来展示了高精度运行时间的不同，运用 VTune 进行了 profiling 部分，最后进行了算法上的拓展实验。（附 github 仓库链接）。

## 1 实验配置

### 1.1 硬件配置

在 x86 处理器上的实验，使用的处理器为 AMD Ryzen 9 7945HX，运行 VTune 的处理器为 13th Gen Intel(R) Core(TM) i9-13900HX。

### 1.2 软件配置

在本机上安装 Code::Blocks 20.03 作为集成开发环境，安装 TDM-GCC-64-10.3.0 编译器，在 Code::Blocks 中配置编译器的页面选择安装 TDM-GCC-64-10.3.0 的路径，于是配置好了一套完整的 GCC 编译环境。在 Intel 机器上安装 VTune 对算法程序进行 profiling 工作。

## 2 基础实验

### 2.1 实验一： $n \times n$ 矩阵与向量内积

实验一的实验内容是给定一个  $n \times n$  大小的矩阵，计算与一个长度为  $n$  的向量作内积的结果。通过高精度测量两种不同的算法执行计算的时间和其他性能指标，来评价不同的方法。

#### 2.1.1 算法设计

平凡算法：逐列访问矩阵的元素，逐列计算内积，最后得到全部的列的结果，是一种平凡的符合平时计算思维的直观想法

cache 优化算法：旨在体现出 cache 的作用，方法改为逐行访问矩阵元素，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，所有列的内积只有在最后一行全部访问完之后，才会一次性得出所有最后内积结果。这样访存模式具有很好空间局部性，便于发挥出 cache 的能力。

#### 2.1.2 编程实现

首先，两种方法使用的给定  $n \times n$  矩阵应保持一致，长度为  $n$  的向量也应当相等。两种编程只有算法设计上的不同，其他变量应该保持一致。在数值的初始化方面保持一致即可。测时间参数  $N$  选择 150。

```
1 .....
2 //同样的初始化方法
3 #define N 150
4 int main() {
5     float** b = new float* [N];
6     for (int i = 0; i < N; i++) {
7         b[i] = new float[N];
8     }
```

```

9     float* a = new float[N];
10    float* sum = new float[N];
11    //统一的初始化 b[N][N]
12    for (int i = 0; i < N; i++)
13    {
14        for (int j = 0; j < N; j++)
15        {
16            float value = i * j + i + j;
17            b[i][j] = value;
18        }
19    }
20    //统一的初始化 a[N]
21    for (int i = 0; i < N; i++)
22    {
23        a[i] = (i + 1) * 1.0;
24    }
25    .....
26    for (int i = 0; i < N; i++) delete[] b[i];
27    delete[] b;
28    delete[] a;
29    delete[] sum;
30 }

```

如代码所示，创建大小为  $150 \times 150$  的矩阵  $b[N][N]$ ，每个元素初始化为  $i*j+i+j$  (其中  $i$  为元素所处的行数， $j$  为元素所处的列数)，初始化大小为 150 的向量  $a[N]$ 。这一部分，两个算法是相同的。

下面是平凡算法的实现。

```

1  //矩阵与向量内积的平凡算法
2  void cal_mul(float **b, float *a, float *sum)
3  {
4      //逐列访问矩阵元素：一步外层循环（内存循环一次完整执行）计算出一个内积结果
5      for (int i = 0; i < N; i++)
6      {
7          sum[i] = 0.0;
8          for (int j = 0; j < N; j++)
9          {
10             sum[i] += b[j][i] * a[j];
11          }
12      }
13 }

```

定义 `cal_mul` 函数用于计算，参数传入  $b[N][N]$  矩阵和  $a[N]$ 、 $sum[N]$  向量。代码中先采用一个外层循环，遍历每一列，对于每一列  $i$ ，有一个  $sum[i]$  来记录内积结果，然后再采用一个内层循环，遍历矩阵中第  $i$  列的每一个元素。逐列访问矩阵元素，一步外层循环、内存循环一次完整执行后计算出一个内积结果，当所有列遍历完成后，最后一个内积结果计算出来， $sum[N]$  也就计算出来了。

下面是 `cache` 优化算法的实现。

```

1  //矩阵与向量内积的 cache 优化算法
2  void cal_mul_pro(float **b, float *a, float *sum)
3  {
4      // 改为逐行访问矩阵元素：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
5      for (int j = 0; j < N; j++) //特定的行
6      {
7          for (int i = 0; i < N; i++) //一行中，每列往后遍历

```

```

8         {
9             sum[i] += b[j][i] * a[j]; //sum 只有在最后一行运行完后才会出现全部的结果。
10        }
11    }
12 }

```

定义 `cal_mul_pro` 函数用于计算。这次不按列进行遍历，改为逐行访问矩阵元素（一步外层循环），然后对于特定的行，遍历这行的每一个元素，计算结果累加到所有列的内积上。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，只有在最后一行运行完后才会出现全部的结果。

### 2.1.3 性能测试

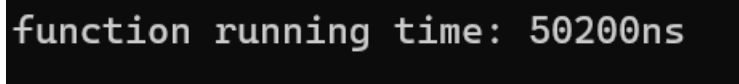
我们在这里完成高精度计时测试。调用 `c++` 的 `chrono` 库，在代码中添加如下的代码：

```

1  #include <chrono> // 高精度时间库
2  int main()
3  {
4      //开始计时
5      auto start = std::chrono::high_resolution_clock::now();
6      //需要测试的函数
7      function();
8      //结束计时
9      auto end = std::chrono::high_resolution_clock::now();
10     //以纳秒为单位进行转换
11     auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
12     //输出
13     cout << "function running time: " << duration.count() << "ns" << endl;
14 }

```

对于矩阵与向量内积的平凡算法运行一次，如下图 2.1 所示



```
function running time: 50200ns
```

图 2.1: 展示示例

接下来，对于平凡算法和优化算法都进行多次高精度时间测量，结果记录如下。

| 算法 \ 时间    | 1     | 2     | 3     | 4     | 5     | 平均    |
|------------|-------|-------|-------|-------|-------|-------|
| 平凡算法       | 50200 | 50400 | 50800 | 50300 | 50500 | 50440 |
| cache 优化算法 | 35500 | 29200 | 30300 | 35200 | 34400 | 32920 |

表 1: 性能测试结果 (单位:ns)

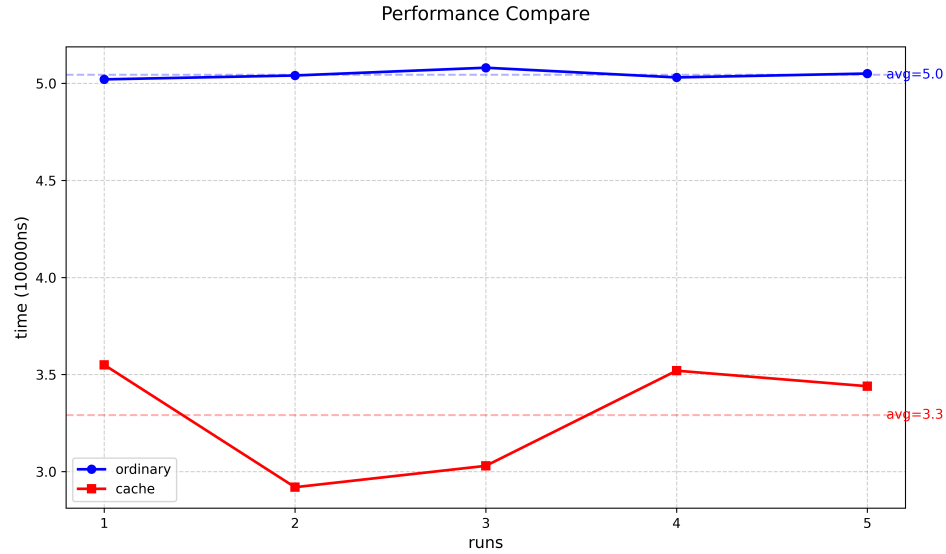


图 2.2: 折线图展示 (单位: 万 ns)

可见 cache 改进后的算法性能有了显著提升, 时间耗费约为原始算法 65.27%。

### 2.1.4 profiling

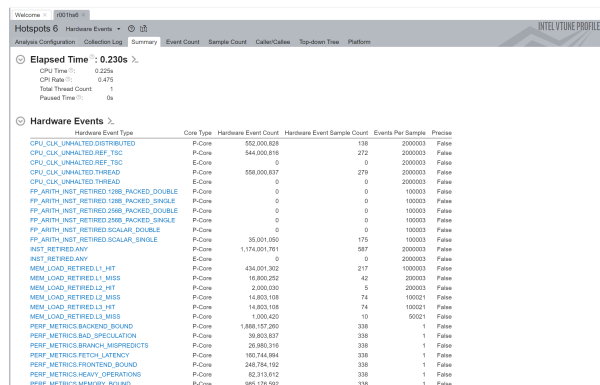


图 2.3: 平凡算法

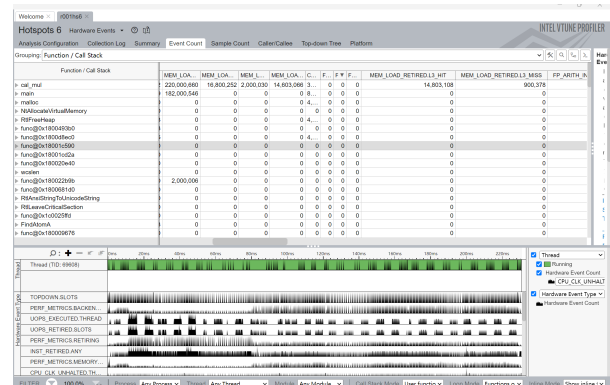


图 2.4: 平凡算法

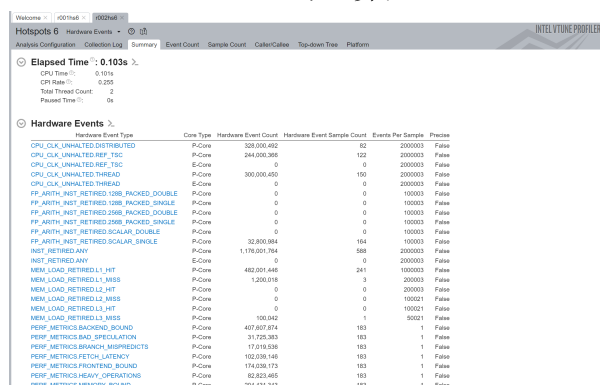


图 2.5: 改进算法

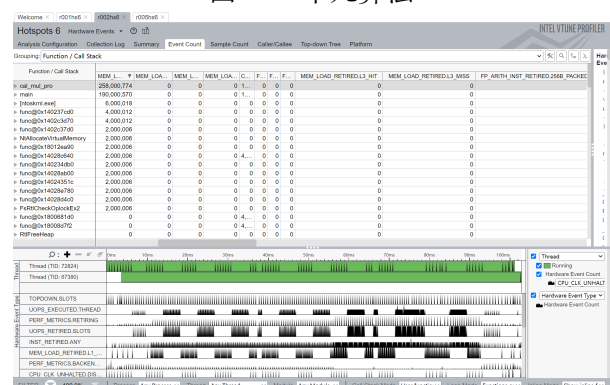


图 2.6: 改进算法

为了测试结果更加直观明显, 将数据规模扩大至 4096 进行比较。具体数据如下: 平凡算法的 CPU Time 是 0.225s, CPI Rate 是 0.475。改进后的算法 CPU Time 是 0.101s, CPI Rate 为 0.255。改进后

的算法程序运行占用的 CPU 时间是平凡算法的 44.89%。下面测试 cache 命中率, 在 VTune 中勾选相应的测试, 第一行能够看到相应的函数名称 cal\_mul 和 cal\_mul\_pro。如图, 平凡算法的 L1 缓存

| 算法 \ 缓存    | L1_HIT      | L1_MISS    | L2_HIT    | L2_MISS    | L3_HIT     | L3_MISS |
|------------|-------------|------------|-----------|------------|------------|---------|
| 平凡算法       | 220,000,660 | 16,800,252 | 2,000,030 | 14,603,066 | 14,803,108 | 900,378 |
| cache 优化算法 | 258,000,774 | 0          | 0         | 0          | 0          | 0       |

表 2: 缓存利用结果

Miss/Hit 比为 7.64%, L2 缓存 M/H 比为 730.14%, L3 缓存 M/H 比为 6.08%。而 cache 优化算法, 仅在 L1 缓存就达到了全部 HIT 的结果, 对缓存的利用率有极大的提升。

### 2.1.5 结果分析

cache 改进后的算法性能有了显著提升, 时间耗费约为原始算法 65.27%, 算法函数仅在 L1 缓存就达到了全部 HIT 的结果, 极大提升了缓存的利用率

## 2.2 实验二: n 个数求和

实验的内容是计算给定的 n 个数求和的结果。通过高精度测量两种不同的算法执行计算的时间和其他性能指标, 来评价不同的方法。

### 2.2.1 算法设计

平凡算法: 遍历 n 个数, 然后将结果累加到 sum 中。

优化算法: 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。

### 2.2.2 编程实现

数组 a[N] 和 sum 初始化都保持一致, 此处不再赘述。

下面是平凡算法的实现。

---

```

1 //n 个数求和的平凡算法
2 #define N 2048
3 void cal_adder(float *a, float &sum)
4 {
5     sum = 0.0;
6     for(int i=0; i<N; i++)
7     {
8         sum+=a[i];
9     }
10 }
```

---

这部分代码就是遍历 a[N] 的每一个元素, 然后累加到 sum 中。

下面是优化算法: 两路链式和递归方法的实现。

---

```

1 #define N 2048
2 //下面是两路链式的算法实现
```

---

```

3 void cal_adder_pro1(float *a,float &sum)
4 {
5     float sum1=0.0;
6     float sum2=0.0;
7     sum = 0.0;
8     for (int i = 0;i < N; i += 2)
9     {
10         sum1 += a[i];
11         sum2 += a[i + 1];
12     }
13     sum = sum1 + sum2;
14 }
15
16 //下面是递归算法的实现
17 void cal_adder_pro2(float *a,float &sum)
18 {
19     int n=N;
20     for (int m = n; m > 1; m /= 2) // log(n) 个步骤
21     {
22         for (int i = 0; i < m / 2; i++)
23         {
24             a[i] = a[i * 2] + a[i * 2 + 1]; // 相邻元素相加连续存储到数组最前面
25         }
26     }
27     sum = a[0]; //最终结果为 a[0]
28 }

```

如代码所示，两路链式算法仍然是遍历数组，但是步长变成了 2，然后在循环内每次进行 sum1 和 sum2 分别的累加：一个进行偶数链的累加，一个进行奇数链的累加；递归算法利用了递归的思想，每次处理一半的数组，外层循环每次除以二，只有  $\log(n)$  的步骤，内存循环访问这一半的元素，然后将后半部分相邻的元素加到前面。由于 N 值取的是 2048，是 2 的幂次，所以函数的索引在递归中是符合预期的。当递归结束，所有的结果就被存储在了 a[0] 中。

### 2.2.3 性能测试

在这里对平凡算法和两路链式、递归算法分别进行时间测量，结果记录如下。

| 算法 \ 时间 | 1    | 2    | 3    | 4    | 5    | 平均   |
|---------|------|------|------|------|------|------|
| 平凡算法    | 4700 | 4900 | 4800 | 5100 | 4900 | 4880 |
| 两路链式算法  | 2900 | 2800 | 2800 | 2900 | 3100 | 2900 |
| 递归算法    | 2500 | 2200 | 2600 | 2400 | 2300 | 2400 |

表 3: 性能测试结果 (单位:ns)



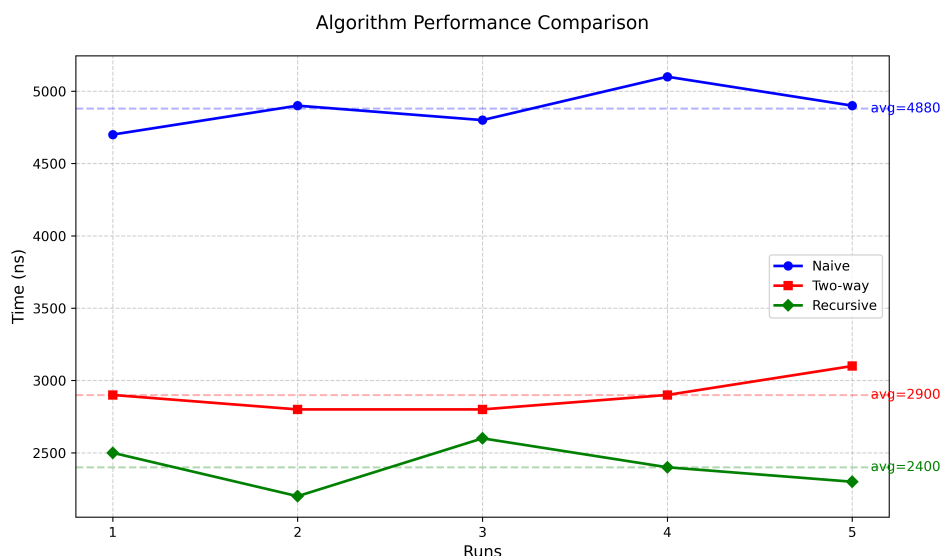


图 2.7: 折线图展示 (单位:ns)

可见通过超标量优化算法,程序的性能得到了很好的提升,两路链式算法耗时是平凡算法的 59.43%,递归算法耗时是平凡算法的 49.18%。

## 2.2.4 profiling

| Hardware Event Type                      | Core Type | Hardware Event Count | Hardware Event Sample Count | Events Per Sample | Precise |
|--|-----------|----------------------|-----------------------------|-------------------|---------|
| CPU_CLK_UNHALTED.DISTRIBUTED             | P-Core    | 24,000,028           | 8                           | 2000003           | False   |
| CPU_CLK_UNHALTED.REF_TSC                 | P-Core    | 14,000,021           | 7                           | 2000003           | False   |
| CPU_CLK_UNHALTED.REF_TSC                 | E-Core    | 0                    | 0                           | 2000003           | False   |
| CPU_CLK_UNHALTED.THREAD                  | P-Core    | 16,000,024           | 8                           | 2000003           | False   |
| CPU_CLK_UNHALTED.THREAD                  | E-Core    | 0                    | 0                           | 2000003           | False   |
| FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.SCALAR_DOUBLE      | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.SCALAR_SINGLE      | P-Core    | 0                    | 0                           | 1000003           | False   |
| INST_RETIRED.ANY                         | P-Core    | 16,000,024           | 8                           | 2000003           | False   |
| INST_RETIRED.ANY                         | E-Core    | 0                    | 0                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L1_HIT                  | P-Core    | 2,000,008            | 1                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L1_MISS                 | P-Core    | 400,006              | 1                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L2_HIT                  | P-Core    | 0                    | 0                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L2_MISS                 | P-Core    | 0                    | 0                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L3_HIT                  | P-Core    | 0                    | 0                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L3_MISS                 | P-Core    | 0                    | 0                           | 1000003           | False   |
| PERF_METRICS.BACKWARD_BOUND              | P-Core    | 38,059,831           | 9                           | 1                 | False   |
| PERF_METRICS.BAD_SPECULATION             | P-Core    | 6,274,507            | 9                           | 1                 | False   |
| PERF_METRICS.BRANCH_MISPREDICTS          | P-Core    | 5,990,781            | 9                           | 1                 | False   |
| PERF_METRICS.FETCH_LATENCY               | P-Core    | 26,509,809           | 9                           | 1                 | False   |
| PERF_METRICS.FPINTEND_BOUND              | P-Core    | 34,117,892           | 9                           | 1                 | False   |
| PERF_METRICS.HEAVY_OPERATIONS            | P-Core    | 1,927,885            | 9                           | 1                 | False   |
| PERF_METRICS.MEMORY_BOUND                | P-Core    | 26,549,024           | 9                           | 1                 | False   |

图 2.8: 平凡算法

| Hardware Event Type                      | Core Type | Hardware Event Count | Hardware Event Sample Count | Events Per Sample | Precise |
|--|-----------|----------------------|-----------------------------|-------------------|---------|
| CPU_CLK_UNHALTED.DISTRIBUTED             | P-Core    | 20,000,030           | 5                           | 2000003           | False   |
| CPU_CLK_UNHALTED.REF_TSC                 | P-Core    | 14,000,021           | 7                           | 2000003           | False   |
| CPU_CLK_UNHALTED.REF_TSC                 | E-Core    | 0                    | 0                           | 2000003           | False   |
| CPU_CLK_UNHALTED.THREAD                  | P-Core    | 18,000,027           | 9                           | 2000003           | False   |
| CPU_CLK_UNHALTED.THREAD                  | E-Core    | 0                    | 0                           | 2000003           | False   |
| FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.SCALAR_DOUBLE      | P-Core    | 0                    | 0                           | 1000003           | False   |
| FP_ARITH_INST_RETIRED.SCALAR_SINGLE      | P-Core    | 0                    | 0                           | 1000003           | False   |
| INST_RETIRED.ANY                         | P-Core    | 12,000,018           | 6                           | 2000003           | False   |
| INST_RETIRED.ANY                         | E-Core    | 0                    | 0                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L1_HIT                  | P-Core    | 2,000,008            | 1                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L1_MISS                 | P-Core    | 0                    | 0                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L2_HIT                  | P-Core    | 0                    | 0                           | 2000003           | False   |
| MEM_LOAD_RETIRED.L2_MISS                 | P-Core    | 0                    | 0                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L3_HIT                  | P-Core    | 0                    | 0                           | 1000003           | False   |
| MEM_LOAD_RETIRED.L3_MISS                 | P-Core    | 0                    | 0                           | 1000003           | False   |
| PERF_METRICS.BACKWARD_BOUND              | P-Core    | 96,431,383           | 14                          | 1                 | False   |
| PERF_METRICS.BAD_SPECULATION             | P-Core    | 9,696,270            | 14                          | 1                 | False   |
| PERF_METRICS.BRANCH_MISPREDICTS          | P-Core    | 9,372,545            | 14                          | 1                 | False   |
| PERF_METRICS.FETCH_LATENCY               | P-Core    | 44,079,435           | 14                          | 1                 | False   |
| PERF_METRICS.FPINTEND_BOUND              | P-Core    | 96,549,030           | 14                          | 1                 | False   |
| PERF_METRICS.HEAVY_OPERATIONS            | P-Core    | 3,088,033            | 14                          | 1                 | False   |
| PERF_METRICS.MEMORY_BOUND                | P-Core    | 38,362,160           | 14                          | 1                 | False   |

图 2.9: 改进算法

400, 006/2000, 006, 改进后的算法缓存效率提升了 20.00%。

## 2.2.5 结果分析

两路链式算法耗时是平凡算法的 59.43%，递归算法耗时是平凡算法的 49.18%，算法缓存效率提升了约 20.00%。

# 3 进阶实验

在  $n$  个数加法实验中,两路链式算法耗时达到了平凡算法的 59.43% 的效果。考虑假如将两路链式算法拓展到四路链式算法,看看能否有更好的效果。代码改进如下:

---

```

1 void cal_adder_pro1(float *a, float& sum)
2 {
3     float sum1=0.0,sum2=0.0,sum3=0.0,sum4 = 0.0;
4     sum = 0.0;
5     for (int i = 0; i < N; i += 4)
6     {
7         sum1 += a[i];
8         sum2 += a[i + 1];
9         sum3 += a[i + 2];
10        sum4 += a[i + 3];
11    }
12    sum = sum1 + sum2 + sum3 + sum4;
13 }

```

---

发现，四路链式算法效果还是有所提升。

| 算法 \ 时间 | 1    | 2    | 3    | 4    | 5    | 平均   |
|---------|------|------|------|------|------|------|
| 两路链式算法  | 2900 | 2800 | 2800 | 2900 | 3100 | 2900 |
| 四路链式算法  | 1400 | 1500 | 1400 | 1400 | 1600 | 1460 |

表 4: 性能测试结果 (单位:ns)

可见，两路扩展到四路链式算法，性能依然有所提升，达到了平凡算法的 29.92%，达到了两路链式算法 50.34%。考虑继续扩大路数：

| 算法     | 两路   | 四路   | 八路   | 十六路  | 三十二路 |
|--------|------|------|------|------|------|
| 平均运行时间 | 2900 | 1460 | 1160 | 1020 | 1240 |

表 5: 性能测试结果 (单位:ns)

路数扩大到八路链式算法时，算法平均耗时达到了 1160ns，继续扩大，十六路链式算法平均时间继续下降，达到了 1020ns，此时已经是平凡算法的 20.90% 了。再扩大路数，三十二路的耗时是 1240ns，稍微有所上升了。我认为的解释是，一定路数（在本例中，结果为十六路）的链式算法，更能充分利用缓存，减少缓存 miss 的次数；而继续扩展，每次循环处理的元素过多，反而削弱了多路处理的优势，在单个循环里数据又变得臃肿，结果也就更趋于平凡了。不妨大胆设想，如果路数达到 1024 甚至 2048，只需要循环一两次，而循环内计算上千个数据，这显然是个差劲的算法。

## 4 总结

实验最终结果是 Cache 优化算法和适合超标量架构的指令级并行算法都取得了较大的性能改进。最后附上源码的 Github 仓库链接：<https://github.com/GYunnnnnX/NanKai-Parallel-Works>