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

体系结构相关及性能测试实验报告

姓名：卻铭恺

学号：2012411

专业：计算机科学与技术

指导教师：王刚

2022 年 3 月 15 日

摘 要

本文是南开大学计算机学院并行程序设计课程的第二次实验的实验报告。内容包含两部分，分别为矩阵与向量内积计算的平凡算法与 cache 优化算法的编程实现、性能测试与对比，以及对 n 个数求和的平凡算法与超标量优化算法的编程实现、性能测试与对比。分别在 x86 平台和 arm 两个平台上完成实验，使用本机上的两个操作系统（Windows 10 和 Arch Linux）以及华为鲲鹏服务器，且使用 Vtune 和 perf 作为性能分析工具。

关键词：性能测试 优化

目录

一、 引言

本次实验是并行程序设计课程的第一次实验。实验分为两部分，第一部分为 cache 优化部分，第二部分为超标量优化部分。下面简要介绍我使用的电脑概况、编程工具、测试软件，为后文的实验数据和分析做出真实保障。

(一) 电脑概况

我的电脑型号是微星(MSI)GS66，采用 Intel(R) Core(TM) i7-10750H 处理器，主频 2.60GHz，最高睿频 5.0GHz，具有超线程功能，具有六核心十二线程。其中，L1 代码缓存为 32KB 每核心，L1 数据缓存为 32KB 每核心，L2 缓存为 256KB 每核心，L3 缓存为 12MB。具有 16GB（准确数据为 16201MB）的内存大小，另外有 29513MB 的虚拟内存。

电脑具有双系统，分别为 Windows10 和 Arch Linux。后文中会分析在两个系统上运行的程序的各自性能。

(二) 编程工具

本次实验的编程工具采用了 gcc（g++）编译器，Windows 下采用 codeblocks IDE，Linux 下采用 VS code 编辑器配合 gcc 编译器，另外，使用 Linux 下的 qemu 模拟器检验代码正确性。

(三) 测试软件及其他工具

采用 mobaxterm 连接鲲鹏服务器进行 Arm 平台上的性能测试，在 Windows 下使用 VTune 进行性能分析，在 Linux 下采用 perf 进行性能分析。另外，使用了高精度计时工具，Windows 下采用了 windows.h 中的 QueryPerformance 进行精确计时，而 Linux 中采用 gettimeofday()。详见后文。

我在我个人的 Github 账号上上传了我的项目，包括源代码和一个包含优化版本的代码，您可以在https://github.com/TwinIsland-CCC/ParallelHomework02里找到它。

二、 体系结构相关实验分析——cache 优化

(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，考虑两种算法设计思路：逐列访问元素的平凡算法和 cache 优化算法，进行实验对比：

1. 对两种思路的算法编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

(二) 实验设计

为了证明本文中提出的 cache 优化算法具有更好的性能，我将本实验分为几个部分。

1. 准备工作

准备工作是指在算法正式编译运行之前，为算法编写完整的程序框架以及设计测试样例。

本实验中，采用 rand 函数进行数据的初始化，所有初值均为 100 以内的正整数。此外，人为设置了控制变量，通过修改代码可以修改问题规模和执行次数。代码的正确性已经检验过。

2. 程序分析

平凡算法的设计思路为纯模拟思路，让 a 中的每一个元素和 b 中的每一列元素相乘，其中逐列访问 b 中元素，再将结果加到 sum 中。

cache 优化算法则是让 a 中的每一个元素和 b 中的每一行元素相乘，逐行访问 b 中元素，并且把局部结果加到 sum 中，当循环全部结束后才得到所有结果。cache 优化算法与 C++ 使用的行主存储模式相匹配，具有很好的空间局部性，令 cache 作用能够很好的发挥。

3. 算法设计

两种思路分别为平凡算法(这里命名为 normalAlg)和 cache 优化算法(这里命名为 cacheOptAlg)，参数均为 a（给定向量），b（给定矩阵）以及 sum（结果），返回值为 void。

平凡算法的代码如下:

```
1 //平凡算法
2 void normalAlg(float* a, float** b, float* sum){
3     for(int i = 0; i < n; i++){
4         sum[i] = 0.0;
5         for(int j = 0; j < n; j++)
6             sum[i] += b[j][i] * a[j];
7     }
8 }
```

cache 优化算法的代码如下:

```
1 //cache优化算法
2 void cacheOptAlg(float* a, float** b, float* sum){
3     for(int i = 0; i < n; i++) sum[i] = 0.0;
4     for(int j = 0; j < n; j++)
5         for(int i = 0; i < n; i++)
6             sum[i] += b[j][i] * a[j];
7 }
```

4. 性能测试设计

本程序中，由于是对 Cache 进行优化，而由 CPU 相关知识可知我们主要应当关注 Cache 命中率。所以在 VTune 和 perf 中进行性能测试时候的时候主要关注 L1、L2、L3 缓存的命中率和未命中率。详见后文“程序测试”模块。

5. 可选的优化策略设计

此处简单设想，由于两种算法可能在不同数据规模下各有益处，在实际使用中可以根据不同数据规模混合使用两种算法。

(三) 实验过程

上文的设计和准备工作结束后，可以开始进行实验了。

问题规模\时间	运行次数	总时间		平均时间	
		NormalAlg/ms	CacheOptAlg/ms	NormalAlg/ms	CacheOptAlg/ms
0	10000	0.0197	0.5261	0	0.00005
10	10000	2.5851	3.6504	0.00025	0.00037
20	10000	11.4514	11.761	0.00115	0.00118
50	5000	38.5141	36.8817	0.0077	0.00737
100	1000	28.0128	27.7306	0.02801	0.02773
150	1000	61.6834	58.6902	0.06168	0.05869
200	500	58.0061	52.9009	0.11601	0.1058
500	100	102.483	72.6537	1.02483	0.72654
1000	10	72.3217	34.0422	7.23217	3.40422
2000	10	351.316	98.5436	35.1316	9.85436
3000	1	95.986	23.5428	95.986	23.5428
5000	1	209.091	62.2788	209.091	62.2788
10000	1	949.733	248.95	949.733	248.95
20000	1	4875.64	992.48	4875.64	992.48

表 1: 实验一在 Windows10 下程序运行时间数据（采用 g++ 编译器，未开启优化）

1. 程序测试

Windows 平台下的实验结果（x86）如表 1 和图 1 所示:
Linux 平台下的实验结果（x86）如表 2 和图 2 所示:
使用鲲鹏服务器（ARM）的实验结果如表 3 和图 3 所示:

2. 性能分析

采用 Windows 下的 VTune 对实验一通过 g++ 编译生成的程序进行性能分析，其中问题规模为 1000，重复执行 10 次，结果如表 4 所示:

采用鲲鹏服务器下的 perf 再次对实验一通过 g++ 编译生成的程序进行性能分析，其中问题规模为 1000，重复执行 10 次，结果如表 5 所示:

(四) 实验结果与数据分析

下文中将对上文的数据进行分析。

1. 执行时间

容易知道，算法的时间复杂度均为 $O(n^2)$。由 Windows 下的测试数据可知，在问题规模较小的时候，Cache 优化算法似乎运行地更加缓慢，而当问题规模大概在 20 50 的时候两个算法性能接近持平，之后随着问题规模的增大，Cache 优化算法变得相比平凡算法而言更加迅速，在问题规模上万的时候有了接近 5 6 倍的性能提升，这足以证明 Cache 优化算法在性能优化上的有效性。

与 Windows 不同，在鲲鹏服务器以及 Arch Linux 下，Cache 优化算法都是从数据规模为 0 开始时就要比平凡算法快速的。这两个平台本质都是 linux 系统，区别在于指令集，我本机的 Arch Linux 是 X86 指令集，鲲鹏服务器为 Arm 指令集。在这两个数据的对比下可以直观看到（参见表 2 和表 3），同样代码所编译成的程序在鲲鹏服务器下的运算速度明显低于在我本机下的运行速度。这或许是多种原因所导致的，首先我们运算这个程序使用的计算机就不一样；也有

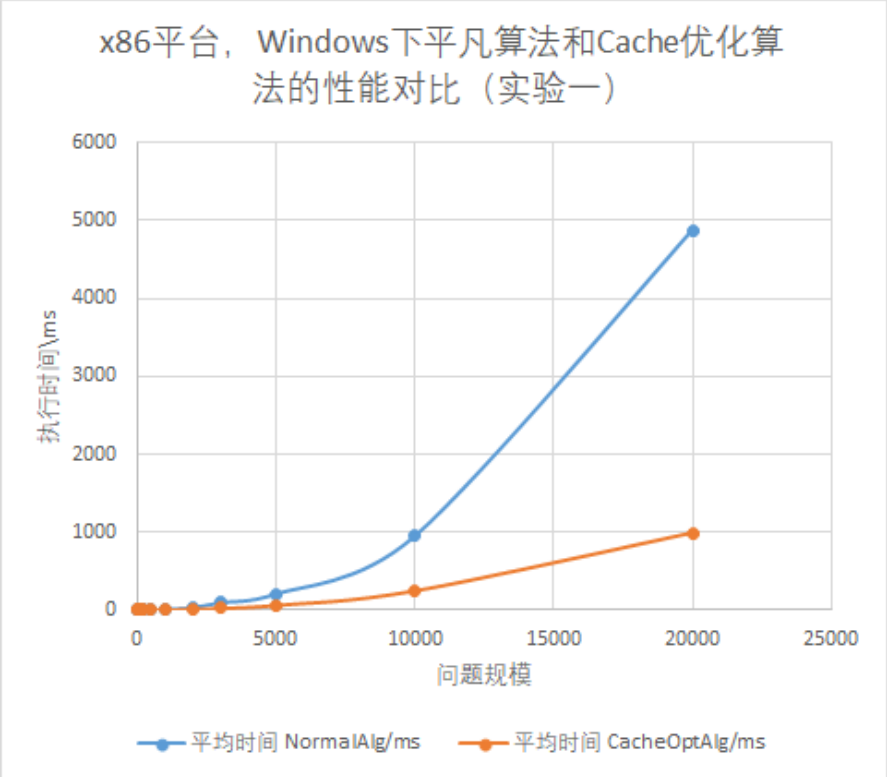

图 1: x86 平台，Windows 下平凡算法和 Cache 优化算法的性能对比（实验一）

问题规模\时间	运行次数	总时间		平均时间	
		NormalAlg/ms	CacheOptAlg/ms	NormalAlg/ms	CacheOptAlg/ms
0	10000	0.102	0.108	0.00001	0.00001
10	10000	3.502	2.389	0.00035	0.00024
20	10000	11.902	10.128	0.00119	0.00101
50	5000	30.152	27.779	0.00603	0.00556
100	1000	24.078	22.245	0.02408	0.02225
150	1000	51.111	48.36	0.05111	0.04836
200	500	44.415	42.742	0.08883	0.08548
500	100	72.536	54.301	0.72536	0.54301
1000	10	31.804	21.51	3.1804	2.151
2000	10	218.314	85.135	21.8314	8.5135
3000	1	56.161	19.407	56.161	19.407
5000	1	155.662	53.207	155.662	53.207
10000	1	661.211	221.231	661.211	221.231
20000	1	3360.88	850.158	3360.88	850.158

表 2: 实验一在 Arch Linux 下程序运行时间数据（采用 g++ 编译器）

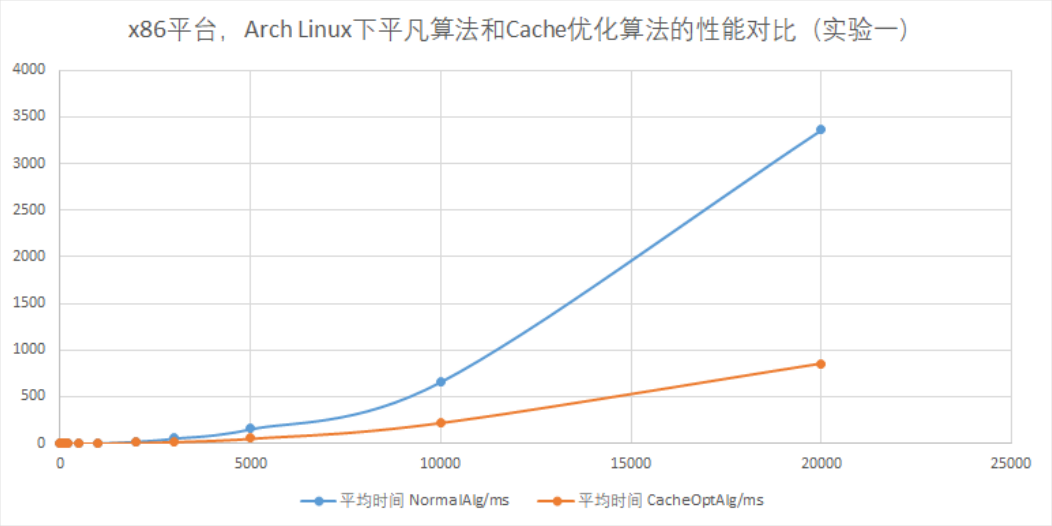

图 2: x86 平台，Arch Linux 下平凡算法和 Cache 优化算法的性能对比（实验一）

问题规模\时间	运行次数	总时间		平均时间	
		NormalAlg/ms	CacheOptAlg/ms	NormalAlg/ms	CacheOptAlg/ms
0	10000	0.027	0.027	0	0
10	10000	6.31	5.812	0.00063	0.00058
20	10000	24.629	22.147	0.00246	0.00221
50	5000	78.511	68.519	0.0157	0.,01370
100	1000	61.91	53.689	0.06191	0.05369
150	1000	140.237	120.949	0.14024	0.12095
200	500	123.354	107.036	0.24671	0.21407
500	100	157.386	133.315	1.57386	1.33315
1000	10	73.309	54.253	7.3309	5.4253
2000	10	304.098	213.651	30.4098	21.3651
3000	1	83.483	49.703	83.483	49.703
5000	1	373.699	140.483	373.699	140.483
10000	1	1775.06	573.465	1775.06	573.465
20000	1	8451	2282.06	8451	2282.06

表 3: 实验一在鲲鹏服务器（ARM）下程序运行时间数据（采用 g++ 编译器，未开启优化）

Vtune 数据		Normal	CacheOpt
实验一（问题规模 1000，重复执行 10 次，指标省略共同前缀 MEM_LOAD_RETIRED.）	L1_HIT	122,907,267	175,674,156
	L1_MISS	8,199,429	176,679
	L2_HIT	2,462,616	156,276
	L2_MISS	5,736,861	20,439
	L3_HIT	6,093,099	6,438
	L3_MISS	249,879	4,053

表 4: 实验一 VTune 性能分析结果（其中问题规模为 1000，重复执行 10 次）

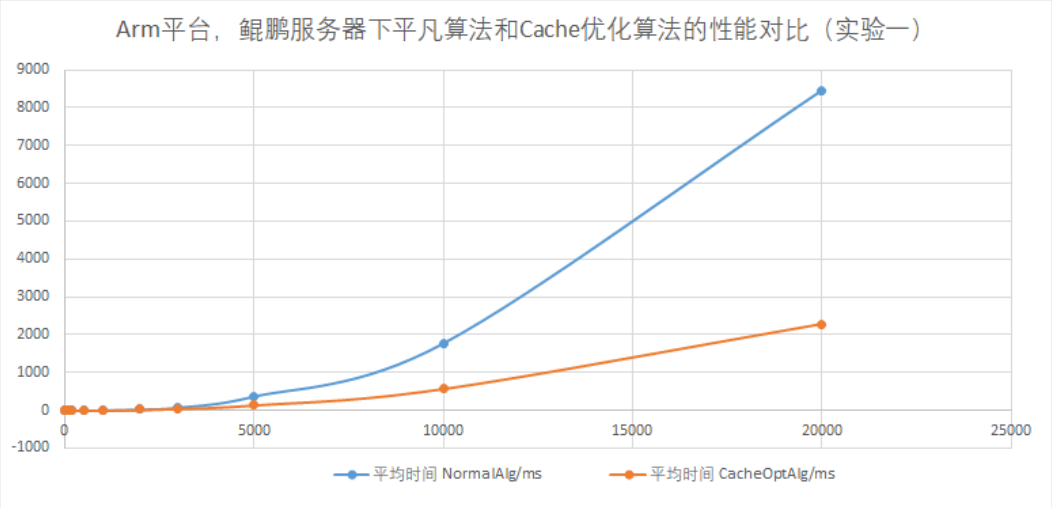

图 3: ARM 平台，鲲鹏服务器下平凡算法和 Cache 优化算法的性能对比（实验一）

perf 数据		Normal	CacheOpt
实验一（问题规模 1000，重复执行 10 次）	L1-dcache-load-misses	86.43%	11.53%
	L1-dcache-loads	45.53%	45.50%
	L1-dcache-stores	45.47%	45.64%

表 5: 实验一 perf 性能分析结果（鲲鹏服务器，其中问题规模为 1000，重复执行 10 次）

可能是指令集本身的原因。说回 Cache 优化算法的性能提升，我们可以看到 Cache 优化算法不论是在 ARM 还是在 X86 指令集下都有了不错的性能提升。另外，考虑到算法的时间复杂度为 $O(n^2)$，我通过数据处理软件计算出了 Arch Linux 下两条时间-问题规模曲线的拟合函数。

Arch Linux 下的函数计算结果如下：

平凡算法：

$$y = 10^{-5}x^2 - 0.0261x + 10.644$$

Cache 优化算法：

$$y = 2\times10^{-6}x^2 + 0.001x - 0.573$$

鲲鹏服务器下的函数计算结果如下：

平凡算法：

$$y = 2\times10^{-5}x^2 - 0.057x + 16.01$$

Cache 优化算法：

$$y = 6\times10^{-6}x^2 + 7\times10^{-5}x - 0.3521$$

可以观察到，Cache 优化算法在 x86 指令集下的优化力度略微大于 ARM 指令集。而要知道他为什么能够优化性能，就要观察下文中的 Cache 命中率指标。后文中分别使用 VTune（本机的 Windows 下）和 perf（鲲鹏服务器下）分析了程序性能，并且做了简要的对比。

2. Cache 命中率

我们知道，CPU 的寄存器保存着最为常用的数据，靠近 CPU 的小的、快速的高速缓存存储器（cache memory）作为一部分存储在相对慢速的主存储器（main memory）中数据和指令的缓冲区域。如果程序需要的数据是存在寄存器中的，那么在指令的执行期间，在 0 个周期内就能访

问到它们，如果在高速缓存中，需要 4 到 75 个周期，如果存储在主存储器中则需要上百个周期。但是，如果存储在磁盘中，则需要几千万个周期。[1] 所以，让程序的设计符合高速缓存的使用，是一种提升程序性能的良好方法。

本实验中所用处理器的缓存分为三级，分别为 L1、L2、L3 缓存，各级缓存的大小已在引言——电脑概况中介绍。在 VTune 中，我们监测 MEM_LOAD_RETIRED.L1_HIT 指标和 MEM_LOAD_RETIRED.L1_MISS 指标（L2 和 L3 同理），而在 perf 中，我们主要观察 L1 数据缓存的命中率和 miss 率，来观察程序对 Cache 的利用率，从而得知算法的性能提升在何处，并对 Cache 优化对性能提升的效果进行证明。

观察表 4 可以了解到，Cache 优化算法的 L1、L2、L2 的 MISS 都远远小于 HIT，尤其大部分数据都在 L1 缓存中被存储，只有一小部分到了 L2 和 L3 中。相比之下，Normal 算法的 L1_MISS 则远远高于 Cache 优化算法的 L1_MISS，甚至相当大的一部分存到了 L3 缓存中，因为 L1、L2、L3 缓存的读取速度是逐级递减的，为什么速度会更慢也就不奇怪了。

观察表 5，发现 Normal 算法的 L1CacheMiss 率远远高于 Cache 优化算法。这表明不管是在 x86 指令集下还是在 arm 指令集下，使用符合编程语言的存储方式的 Cache 优化算法都能起到良好的优化作用。

3. 进一步的思考

上文中，我在 Windows 下做完实验后去电脑上的 Linux 系统中又做了一次实验，代码中只更改了精确计时的方式。观察到，同一台电脑上，在问题规模足够大时，Linux 下的程序执行速度要比 Windows 下程序的执行速度要稍快一些，而且数据规模越大，快得越明显。这可能与 linux 和 windows 的内部有关。我个人的感觉是，linux 是轻量级系统，windows 相对比较臃肿，或许在 linux 上运行同样的源码（确保都能编译）会比在 windows 上要快一些。或许在 Windows 系统的设计上还有优化程序性能的空间。

此外，如上文所说，我在基于 ARM 指令集下的鲲鹏服务器上执行出的程序普遍比 x86 下要慢一些，基于 ARM 指令集的处理器在设计时也有优化的空间。考虑世界登顶的超级计算机——富岳就使用了 ARM 架构 [2]，说明 ARM 正在朝着好的方向发展，最终性能将不亚于 x86。

三、 体系结构相关实验分析——超标量优化

(一) 实验要求

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；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。完成如下作业：

1. 对两种算法思路编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

(二) 实验设计

为了合理评估本文中提出的超标量优化算法的性能，我将本实验分为以下几个部分，与上一个实验的框架类似。

1. 准备工作

准备工作是指在算法正式编译运行之前，为算法编写完整的程序框架以及设计测试样例。

与上文大体相同，本实验也采用 rand 函数将所有初值均初始化为 100 以内的正整数。也设置了与上文类似的控制变量。代码的正确性已经检验过。需要注意的是，为了更好地保证递归算法的效率，我在递归算法中没有采取将值存储起来的策略，而是每一次程序执行完毕以后数组都会改变，这样会造成程序重复运行不能得到同样的结果，但是对性能来说影响相对较小。另外，这里的测试数据（数组大小）最好设置为 2 的整数次方（大于等于 1），便于递归算法计算。

2. 程序分析

有几种思路，分别为平凡算法（这里命名为 normalAlg）和超标量优化算法（这里命名为 superscalraOptAlg）。

平凡算法很简单，即链式累加。

考虑到本实验所用的 CPU 支持超标量优化，使得不相关的指令可以同时执行，于是尝试将上下两条指令分为独立的指令，即将原本的程序转换为两路链式累加，就得到了简单的超标量优化算法。

指导书中还提到了两种递归算法。第一种递归算法是每次循环将前后的元素叠加起来以让操作的数组减半，只需要 $log(n)$ 次操作就能结束递归；第二种是用循环实现的递归，让相邻元素相加存到数组最前面。

3. 算法设计

平凡算法的设计思路为纯模拟思路，即链式累加，代码如下：

```
1  //平凡算法
2  //a为输入数组，sum为和
3  void normalAlg(int* a, int& sum) {
4      for (int i = 0; i < n; ++i) {
5          sum += a[i];
6      }
7  }
```

两路链式累加算法代码如下：

```
1  //超标量优化算法——两路链式累加
2  //a为输入数组，sum为和
3  void superscalraOptAlg(int* a, int& sum) {
4      int sum1 = 0, sum2 = 0;
5      for (int i = 0; i < n; i += 2) {
6          sum1 += a[i];
7          sum2 += a[i + 1];
8      }
9      sum = sum1 + sum2;
10 }
```

两路链式累加算法循环中的上下两条指令相互独立，便于超标量执行。

递归算法 1：

```
1  //递归优化算法1
2  void recursionOptAlg1(int* a, int& sum, int n) {
```

```
    if (n == 1) {
        sum = a[0];
        return;
    }
    else {
        for (int i = 0; i < n / 2; ++i) {
            a[i] += a[n - i - 1];
        }
        n /= 2;
        recursionOptAlg1(a, sum, n);
    }
}
```

递归算法 2:

```
//递归优化算法2
void recursionOptAlg2(int* a, int& sum, int n) {
    for (int m = n; m > 1; m /= 2) // log(n)个步骤
        for (int i = 0; i < m / 2; i++)
            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
    // a[0]为最终结果
    sum = a[0];
}
```

4. 性能测试设计

本程序中对超标量进行了优化，有一个评价指标 IPC（Instruction Per Clock），即每个时钟周期执行的指令数，可以直观地比较这两种算法的区别。直观感觉上，两种算法执行了差不多的指令，但是由于超标量的优化，可以使两路链式累加同时执行独立的指令，所以两路链式累加算法的 IPC 应该显著高于平凡算法的 IPC。所以，在使用 VTune（或 perf）进行性能测试的时候，我们将主要关注指标 IPC（CPI）。

5. 可选的优化策略设计

此处简单设想，可以把两路链式累加根据问题规模优化为多路链式累加，这里简单设想为根号下问题规模或者由 CPU 核心数所决定的多路链式累加，可以辅以上文中的 Cache 优化策略。

另外，有一个循环展开策略（unroll）可以在此处被应用，即每个循环步进行多次加法运算，相当于将多个循环步的工作展开到一个循环步，从而大幅度降低簿记操作的比例。

(三) 实验过程

上文的设计和准备工作结束后，可以开始进行实验啦。

1. 程序测试

Windows 平台下的实验结果（x86）如表 5 和图 3 所示:

问题规模/时间	运行次数	总时间/ms				平均时间			
		Normal	SuperscalarOpt	Recursion1	Recursion2	Normal	SuperscalarOpt	Recursion1	Recursion2
1	10000	0.029	0.3857	0.3114	0.319	0	0.00004	0.00003	0.00003
128	5000	1.1786	1.4302	1.8825	1.6779	0.00024	0.00029	0.00038	0.00033
1024	1000	1.963	1.4311	2.5811	3.136	0.00196	0.00143	0.00258	0.00313
4096	500	3.9692	2.8113	4.7175	4.5754	0.00794	0.00562	0.00943	0.00915
8192	500	8.0503	4.566	9.3974	9.0149	0.0161	0.00913	0.01879	0.01802
16384	500	15.2731	9.2904	18.0614	17.4523	0.03047	0.01858	0.03612	0.0349
32768	100	6.3089	3.7237	7.4076	7.8643	0.06309	0.03724	0.07408	0.07864
65536	100	12.2266	7.6421	14.4074	14.5876	0.12227	0.07642	0.14407	0.14588
131072	100	27.4138	15.5518	29.4346	28.9301	0.27413	0.15552	0.29435	0.2893
262144	10	5.6739	3.6546	5.6372	6.0795	0.56739	0.36546	0.56372	0.60795
524288	10	12.3572	7.3752	12.9098	14.7812	1.23572	0.73752	1.29098	1.47812
1048576	10	24.1281	15.1198	28.2944	32.3781	2.41281	1.51198	2.82944	3.23781

表 6: 实验二在 Windows10 下程序运行时间数据（采用 g++ 编译器，未开启优化）

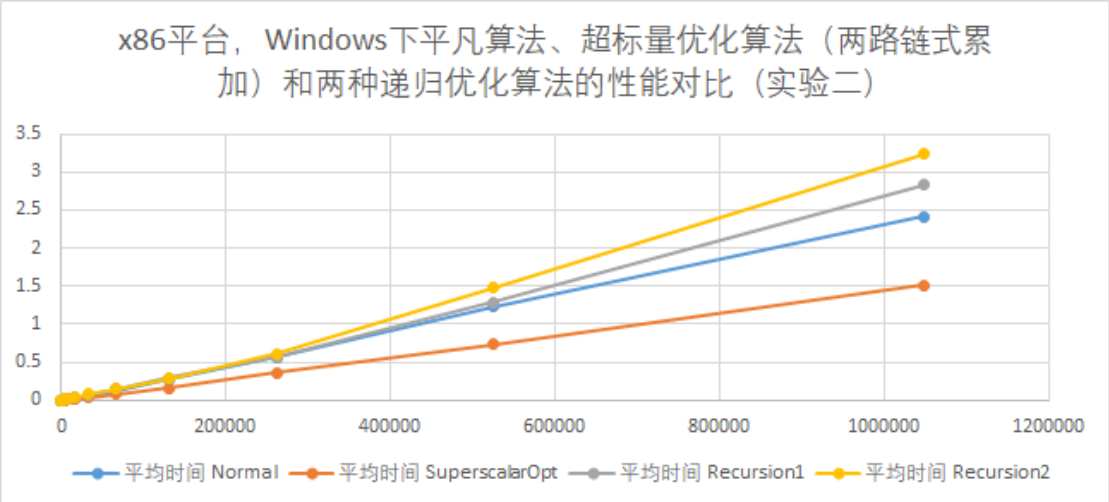

图 4: x86 平台，Windows 下四种算法的性能对比（实验二）

问题规模\时间	运行次数	总时间/ms				平均时间			
		Normal	SuperscalarOpt	Recursion1	Recursion2	Normal	SuperscalarOpt	Recursion1	Recursion2
1	10000	0.042	0.071	0.033	0.035	0	0	0	0
128	5000	2.169	1.244	3.026	2.845	0.00043	0.00025	0.0006	0.00057
1024	1000	3.408	1.884	4.138	4.262	0.0034	0.00188	0.00414	0.00426
4096	500	6.796	3.753	8.197	8.459	0.01359	0.00751	0.01639	0.01692
8192	500	13.644	7.525	16.557	16.824	0.02729	0.01505	0.03311	0.03365
16384	500	27.166	15.067	32.812	34.528	0.05433	0.03013	0.06562	0.06906
32768	100	10.895	6.086	13.357	13.602	0.10895	0.06086	0.13357	0.13602
65536	100	21.863	12.205	26.55	27.588	0.21863	0.12205	0.2655	0.27588
131072	100	43.643	24.38	53.364	54.95	0.43643	0.2438	0.53364	0.5495
262144	10	8.802	4.982	11.191	11.661	0.8802	0.4982	1.1191	1.1661
524288	10	17.621	10.002	23.06	23.95	1.7621	1.0002	2.306	2.395
1048576	10	35.293	20.038	46.586	48.355	3.5293	2.0038	4.6586	4.8355

表 7: 实验二在鲲鹏服务器（ARM）下程序运行时间数据（采用 g++ 编译器，未开启优化）

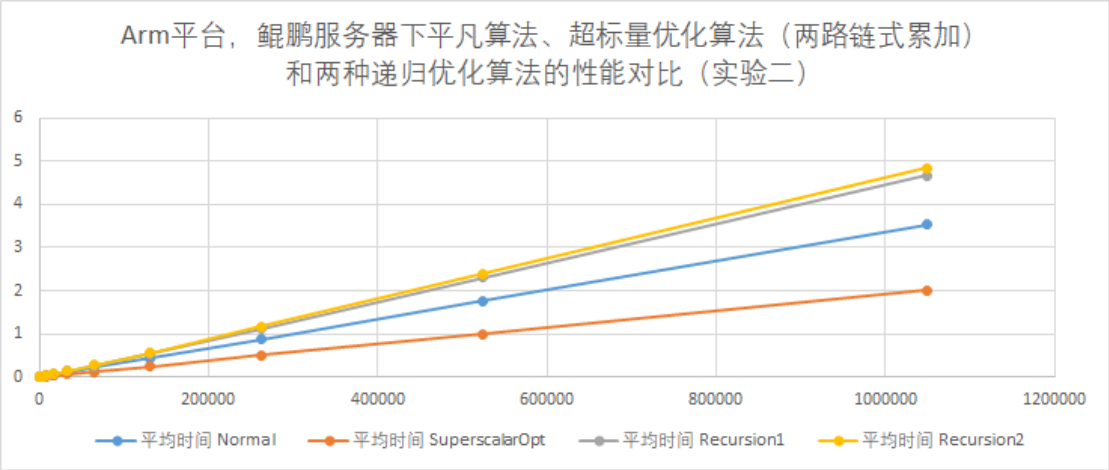

图 5: ARM 平台，鲲鹏服务器下四种算法的性能对比（实验二）

使用鲲鹏服务器（ARM）的实验结果如表 7 和图 5 所示:

2. 性能分析

在 VTune 中，我们可以看到总体执行的周期数（Clockticks），执行指令数（Instructions Retired）以及 CPI（IPC 的倒数，每条指令执行的周期数）。采用 Windows 下的 VTune 对实验二通过 g++ 编译生成的程序进行性能分析，其中问题规模为 1048576，重复执行 10 次，结果如表 8 所示:

Vtune 数据		normalAlg	superscalraOptAlg
实验二（问题规模 1048576，重复执行 10 次）	Clockticks	88,400,000	49,400,000
	Instructions Retired	153,400,000	98,800,000
	CPI Rate	0.576	0.5

表 8: 实验二 VTune 性能分析结果（其中问题规模为 1048576，重复执行 10 次）

（四） 实验结果与数据分析

下文中将对上文的数据进行分析。

1. 执行时间

容易知道，算法的时间复杂度均为 $O(n)$。在问题规模较小的时候，从数据来看，仿佛平凡算法更加快速；但是数据规模没增大多少时，超标量优化算法的运行速度就超过了平凡算法。数据表明，将链式累加改为两路链式累加的策略对性能提升很有帮助。

对比两个递归算法，性能提升并不明显甚至略有下降；究其原因，可能是因为递归函数调用开销较大，并且两个递归算法中间也并没有见到可以使用超标量优化的部分。为了解决这一问题，我尝试使用 Unroll 策略对递归算法进行优化，详见下文。

2. CPI（IPC）

超标量就是可以同时执行独立的指令。在 VTune 中我们注意到了 IPC（CPI），这个指标表示着每个时钟周期执行的指令数（CPI 即 IPC 的倒数）。同上文中的设想一致，我们在 VTune 的测试结果中观察到两路链式累加算法的 CPI 明显低于平凡算法，即 IPC 变高，可以充分说明两路链式累加算法在超标量优化的作用下，性能有所提升。

3. 进一步的思考

通过将链式累加算法改写为两路链式累加，在数据规模足够大时，有了明显的性能提升。因此可以考虑将两路链式累加根据问题规模改写为多路链式累加，或者使用循环展开（loop unrolling）策略，以牺牲代码长度的方式直接将循环去掉，可以继续提高 IPC。

递归算法中，我们也可以试着采用循环展开策略，将每一次循环中的执行步数提升为原来的两倍。采用循环展开策略对递归算法 1 进行优化，代码如下：

```
1  //Unroll优化的递归算法1
2  void optedRecursionAlg1(int* a, int& sum, int n) {
3      if (n == 1) {
4          sum = a[0];
5          return;
6      }
7      else {
8          for (int i = 0; i < n / 2; i += 2) {
9              a[i] += a[n - i - 1];
10             a[i + 1] += a[n - i - 2];
11         }
12         n /= 2;
13         optedRecursionAlg1(a, sum, n);
14     }
15 }
```

用 VTune 分析编译生成的程序的数据如表 8 所示：

在表 8 中可以看到，优化后的算法的 CPI 有所下降，这代表着 IPC 有所提升，也就是说在同一时间内执行了更多的指令，对性能有所提升。表 9 中列举出了一部分三种算法性能测试的数据。经过再次测量实验数据，证明了此种方法的确对性能有所提升。

Vtune 数据		recursionOptAlg1	recursionOptAlg2	optedRecursionAlg1
实验二（问题规模 1048576，重复执行 10 次）	Clockticks	10,400,000	10,400,000	7,800,000
	Instructions Retired	31,200,000	33,800,000	28,600,000
	CPI Rate	0.333	0.308	0.273

表 9: 实验二的 Unroll 优化算法的 VTune 性能分析结果（其中问题规模为 1048576，重复执行 10 次，optedRecursionAlg1 为优化后的算法）

问题规模\时间	运行次数	总时间/ms			平均时间/ms		
		Recursion1	Recursion2	Opted	Recursion1	Recursion2	Opted
1024	1000	2.817	2.7216	2.2032	0.00282	0.00272	0.0022
16384	500	17.604	17.7751	15.5716	0.0352	0.03555	0.03114
1048576	10	27.847	28.216	24.2405	2.7847	2.8216	2.424

表 10: 实验二在 Windows10 下两种递归算法和递归优化算法的程序运行时间数据（采用 g++ 编译器，未开启优化（其中问题规模为 1048576，重复执行 10 次，Opted 为优化后的算法）

四、 实验感想及收获

通过本次实验，我了解了 Cache 优化和超标量优化两种对单核串行程序进行优化的策略，通过不同平台性能对比我也感受到了系统与系统、指令集与指令集之间的差异。我在本次实验中还学会了性能测试工具、远程连接工具、gcc 编译器以及鲲鹏服务器系统的基本使用，为我以后的实验的顺利进行打下基础。并且，我在文章中尝试发散思维，在实验二的递归算法中尝试用 Unroll 策略优化程序，最终也取得了理想的结果。

参考文献

[1] Randal E.Bryant, David R.O'Hallaron Computers Systems A Programmer's Perspective. Third Edition (深入理解计算机系统第三版中文版龚奕利，贺莲译).

[2] Dongarra J. Report on the Fujitsu Fugaku system[J]. University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06, 2020.