南開大學
Nankai University

计算机学院

并行程序设计实验报告

作业二：体系结构及性能相关测试

姓名：张明昆

学号：2211585

专业：计算机科学与技术

2024 年 3 月 22 日

目录

1 实验目标

1. 以矩阵每一列与向量的内积为例，通过编写代码实践 cache 优化算法。
2. 以求数组累加和为例，通过编写代码实践两路链式相加、循环展开和递归相加等超标量优化算法。
3. 利用 prof 和 uprof 等工具，通过运行计时和事件计数的方法，量化分析普通算法和优化算法之间的性能差异。

2 实验环境

2.1 X86 平台

X86 平台使用的是一台笔记本电脑，具体参数如下：

```
Architecture:                    x86_64
CPU(s):                          8
L1d cache:                       128 KiB
L1i cache:                       256 KiB
L2 cache:                        2 MiB
L3 cache:                        4 MiB
```

2.2 ARM 平台

ARM 平台使用课程提供的服务器，具体参数如下：

```
Architecture:          aarch64
CPU(s):                96
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              49152K
```

3 实验设计及分析

3.1 cache 优化

3.1.1 实验设计

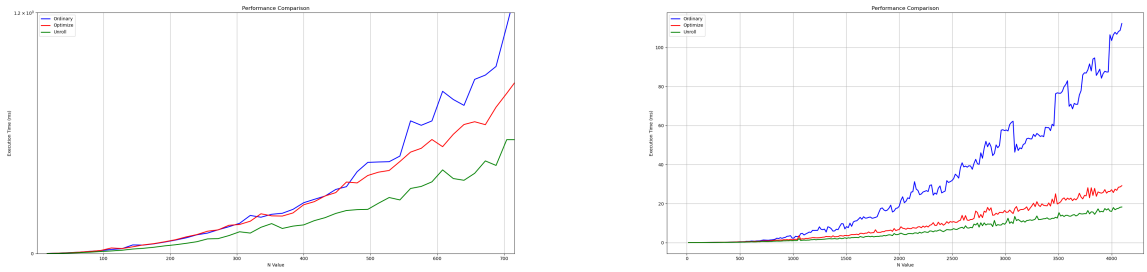
在针对给定问题的改进算法设计中，需考虑到矩阵在内存中是以行优先顺序存储的事实。这种存储方式意味着内存中矩阵的数据按行连续排列，对于逐列访问的算法，这种存储模式会导致较高的缓存未命中率。具体来说，CPU 从内存中预取连续的数据块至缓存时，这些数据块可能仅包含计算特定列元素所需的部分数据。因此，计算下一元素时，CPU 可能需要重新从较低层次的缓存或主内存中获取数据，由于访问内存的延迟远高于执行计算的时间，这种访问模式显著降低了程序的执行效率。

为了缓解这一问题，提出采用逐行访问并进行 cache 优化的算法，旨在最大化利用每次缓存中预取的行数据。算法通过对当前预取到 cache 中的一整行数据进行计算，并将计算结果累加到结果数组的相应位置，从而减少对主内存的访问频率，优化了数据的缓存利用率。虽然此策略并未直接减少计算量，但通过降低数据访问延迟，有效提高了程序整体的执行效率。

进一步地，为了降低算法执行中的条件判断和指令跳转等开销，对逐行访问算法进行循环展开优化。通过在单次迭代中计算多个元素的值，此方法不仅减少了循环次数，还能够充分利用 CPU 的多条执行流水线，提升超标量处理器的计算性能。循环展开策略通过减少循环控制逻辑的开销，进一步提高了算法的执行效率，尤其在处理大规模数据时表现显著。

3.1.2 实验分析

我分别设计了三种算法，分别是原始算法、cache 优化算法、和循环展开优化算法。并在 x86 架构的 PC 上测量了 n<=4096 三种算法所消耗的时间，实验测试数据如图3.1所示，其中，矩阵行列数在实验中以 16 为步进。在研究矩阵访问优化方法时，通过对不同数据规模（N）下算法性能的测试分



(a) n<400

(b) n<4096

图 3.1: 算法消耗时间随问题规模的变化

析，我发现在小规模数据（N<300）情况下，如图1(a)所示，逐行与逐列访问方式的性能差异不显著，我猜测这主要因为 CPU 缓存（L1 和 L2）的较快访问速度能够缓解访存延迟的影响。即使 L1 cache 的命中率已经很低了，但是由于 L2 cache 的访问速度相对较快，所以访存时间差距不大。然而，引入循环展开技术的优化算法在此情境下即显示出显著的性能提升。

随着数据规模的增加（300<N<3000），逐行访问相比于逐列访问的性能增长速度更慢，时间的差距迅速拉开。表明在这一数据规模下，缓存优化的效果开始显现。猜测这一现象与 CPU 缓存的层级结构和容量密切相关：L1 cache 为 64KB，L2 为 512KB，L3 为 48MB。由于每个 unsigned long long int 类型元素占据 10 字节，可估算出各级 cache 能容纳的元素数量分别约为 80，200，2000。数据规模进一步扩大到 N>3000 时，逐列访问的效率明显低于逐行访问，因为此时 L3 cache 的命中率也降低，导致频繁的内存访问，增加了显著的访存开销。

为了验证上述理论，本研究设计了两组实验，分别在数据规模为 64、256 和 4096 的情境下进行，通过利用 VTune 工具分析其各级缓存的访问频次及命中率，详细数据见表1。

在数据规模为 64 的实验中，观察到逐列访问相比逐行访问在 L1 缓存的未命中率显著提高，这导致了对 L2 缓存的访问次数相应增加。尽管如此，由于 L2 缓存在大多数情况下都能成功命中，两种访问方式在效率上的差异并不明显。

随着数据规模增至 256，逐列访问方式在 L2 缓存的未命中率明显高于逐行访问，进而导致对 L3 缓存的访问次数大幅增加。虽然 L3 缓存的命中率依旧很高，但考虑到访问 L3 缓存的相对时间成本较

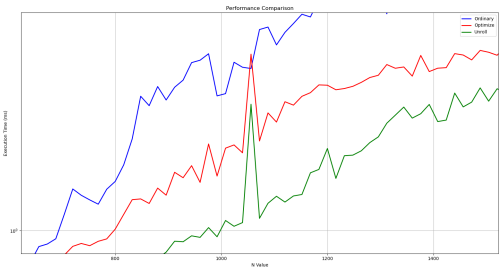
表 1: 不同规模下普通算法和 cache 优化算法各级 cache 访问次数和命中情况

Ordinary						
n	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss
64	7.81E+07	3.19E+05	3.11E+05	7.91E+03	2.64E+02	1.20E+01
256	1.45E+08	1.31E+07	1.28E+07	2.13E+05	1.63E+05	1.50E+02
4096	1.49E+08	1.58E+07	1.19E+07	3.93E+06	2.39E+06	8.89E+05
Optimized						
n	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss
64	9.14E+07	4.08E+03	3.86E+03	1.37E+02	1.20E+02	0.00E+00
256	1.83E+08	5.62E+04	5.11E+04	5.12E+03	9.36E+02	0.00E+00
4096	1.45E+08	8.02E+04	7.65E+04	3.76E+03	3.57E+02	8.70E+01

高，这一阶段两种方法的性能差距开始显著。

数据规模增至 4096 时，情况更加明显。逐列访问方式的 L3 缓存访问次数远超逐行访问，且 L3 缓存的未命中率高达 27.1%，这意味着需要进行大量的内存访问，因而时间成本极高，这成为了两种方法性能差异巨大的关键所在。

进一步的测试在 X86 平台上揭示了算法性能在特定数据规模时出现了尖峰现象。在 n=1024 处，算法执行时间出现了一个明显的尖峰，如图2(a)所示除该尖峰外，整个曲线也呈现出有规律的上下波动，经过几次重复实验后，发现曲线上相近的位置仍然存在尖峰。在 perf 事件计数测量时发现，当矩



(a) 曲线在 n=1024 处出现的尖峰

(b) Cache 的结构 - Cache Line

图 3.2

阵行数为 1024 时，L1-dcache-load-misses 的事件数量比矩阵行数为 1036 时多了 60%。进一步分析汇编代码，发现这些事件基本都集中在提取矩阵 mat[j][i] 这个操作上。查阅资料后发现，这个问题可能和 cache 的结构有关，在现代处理器上，Cache 划分成了 128 字节的小块，这种小块被称为 Cache Line，而同一个 Cache Line 是不可被同时访问的。由于出现尖峰的曲线对应的是列主访问的算法，行列数为整倍数的矩阵导致每次从 Cache 中提取元素都需要跨行访问，导致了额外的开销。也有可能由于超标量的特性，可能存在两条指令同时需要修改同一个 Cache Line 中的内容的情况，这就造成了伪共享（false sharing）现象，从而导致 Cache 命中率降低。

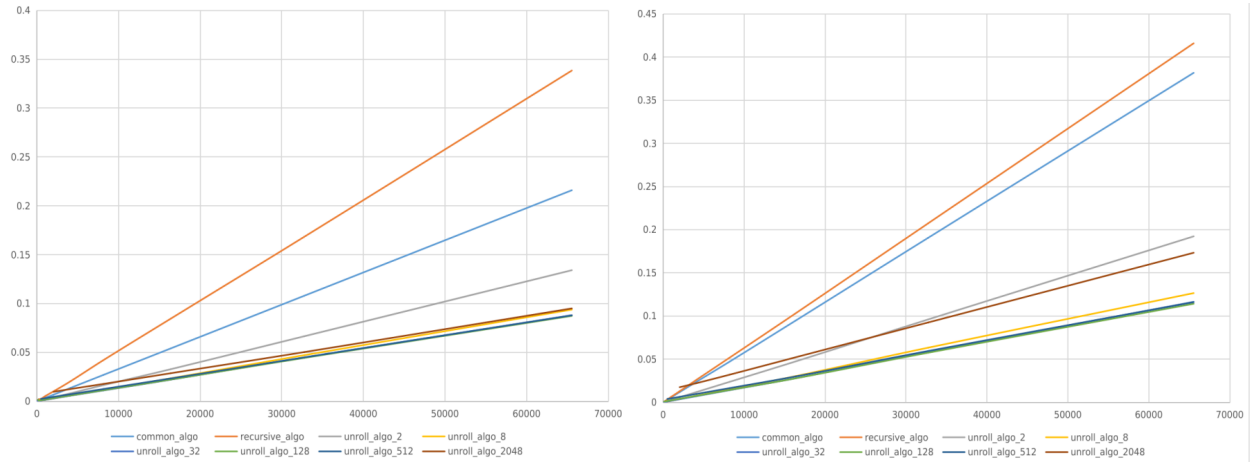
3.2 超标量优化

3.2.1 实验设计

对于给定的问题，要求计算 N 个数的和，对于常规的顺序算法而言，由于每次都是在同一个累加变量上进行累加，导致只能调用 CPU 的一条流水线进行处理，无法充分发挥 CPU 超标量优化的性能，因此考虑使用多链路的方法对传统的链式累加方法进行改进，即设置多个临时变量，在一个循环内同时用着多个临时变量对多个不同的位置进行累加，达到多个位置并行累加的效果，同时还能够减少循环遍历的步长，降低循环开销。

3.2.2 实验分析

累加实验运行计时的结果如下所示我测试了从 0 到 70000 不同问题规模的算法耗时。如图3.3所示。



(a) x86 (b) arm

图 3.3: 算法消耗时间随问题规模的变化

从表格数据可见，无论采用链式累加还是多链路展开策略，两者均为线性时间复杂度算法，其执行时间随问题规模增加而成比例增长。然而，采用双链路展开的超标量优化方法在时间效率上明显胜过传统的链式累加策略。原因在于，多链路策略通过解耦相互依赖的累加过程为两个独立的任务，允许 CPU 同时利用双重流水线进行处理，从而实现了超标量优化。

进一步的证据来自于对比链式累加与多链路累加方法的 CPI（每周期指令数）表现。如表2所示，多链路累加方法的 CPI 显著低于链式累加，意味着在相同的时钟周期内，多链路累加能够完成更多的指令处理。这不仅佐证了超标量优化的有效性，也表明我们成功地利用了 CPU 的多条流水线来提高处理效率

表 2: 普通算法和优化算法的 CPI 对比

	普通算法	2 路优化算法
CPI	0.4975	0.4761

在进行该实验时，我注意到 ARM 平台与 X86 平台在缓存效率上的表现差异显著，其中 ARM 平台的整体缓存命中率远高于 X86 平台（99.158% 相比于 96.659%）。这一现象可能源于 ARM 平台拥

有更为高效的缓存访问机制，并且 ARM 平台的 L1 缓存容量更大（64KB 对比 32KB），从而提高了数据处理的效率。

另一方面，比较两个平台在执行累加操作实验时的运行时间，ARM 平台相较于 X86 平台要长得多，这一差异很可能归因于两者在指令集架构上的根本差异。这表明，虽然 ARM 平台在缓存效率方面表现出色，但其在某些计算任务上的性能可能受到指令集架构差异的影响，从而影响总体的运行效率。

在图3.3中展示的结果出乎意料地显示，递归算法在所有比较的算法中效率最低，这一发现与我们的直觉相悖。深入分析表3与表4揭示了递归算法产生大量的 L1-dcache-load-misses 事件，原因可能在于递归算法频繁进行非连续内存访问，导致跨越 Cache Line，这是降低缓存命中率的主要原因。此外，递归算法本身相对于平凡算法和多路链式算法更为复杂，引入的额外变量可能也对性能产生不利影响。

同时，图3.3中的数据也揭示了一个有趣的现象：循环展开的效益并非随着展开路数的增加而线性提升。例如，在 ARM 平台上，2 路循环展开与 8192 路循环展开的性能相差无几。通过对表3与表4的分析发现，8192 路循环展开导致了大量的 L1-dcache-load-misses 和 L1-icache-load-misses 事件，从而影响了算法的性能。循环展开增加的每行代码都可能引发 misses 事件，这些小额的 misses 累积起来导致了缓存命中率的下降。此外，当输入数据规模增大到超出 L1 Cache 容量时，后续的循环展开将造成显著的性能损失。这些发现提示我们，在优化算法时，需要权衡循环展开的程度与算法性能之间的关系，避免过度优化导致的性能反而下降。

优化方法或 循环展开的层数	cycles	instructions	CPI	L1-dcache-load-misses	L1-icache-load-misses
不优化	13.41%	9.19%	0.6296	8.24%	0.00%
递归	15.55%	18.67%	0.3594	15.01%	0.00%
2	6.73%	7.07%	0.4107	6.50%	0.00%
32	4.14%	5.21%	0.3429	5.06%	0.00%
512	4.15%	5.10%	0.3511	6.01%	0.00%
8192	9.89%	6.51%	0.6555	8.04%	45.59%
事件总数	122855688833	284706374706	0.4315	633713953	2540277364

表 3: ARM 平台下累加优化算法的 perf 事件计数结果

优化方法或 循环展开的层数	cycles	instructions	CPI	L1-dcache-load-misses	L1-icache-load-misses
不优化	11.81%	6.92%	0.6762	5.70%	2.10%
递归	22.60%	17.53%	0.5108	16.13%	4.13%
2	7.24%	6.70%	0.4281	5.51%	1.46%
32	4.84%	5.76%	0.3329	5.36%	1.30%
512	4.64%	5.59%	0.3288	5.38%	2.10%
8192	5.05%	5.69%	0.3516	10.48%	61.37%
事件总数	52560137817	132651641398	0.3962	889797839	60817536

表 4: X86 平台下累加优化算法的 perf 事件计数结果

4 总结

在探究给定的矩阵乘法和数组累加问题时，本研究采取了缓存优化和超标量优化策略，旨在提升串行算法的执行效率。通过实验比较，发现对于矩阵乘法问题，逐行访问方式在处理大规模数据集时展现出显著的性能优势。这一优势源于其能够有效提高缓存命中率，由此减少了由内存访问延迟引起的性能损耗。此外，在数组累加实验中，通过引入超标量优化技术——即同时进行多个独立变量的累加操作，以此最终汇总得到总和——能够充分发挥 CPU 多流水线的并行处理能力，从而显著加速程序的执行。这些发现证实了针对特定计算模式采用相应优化策略的有效性。

该实验的全部代码在 https://github.com/ftygu/COSC0025-Parallel-Programming-Design 中给出。