華中科技大學

研究生课程报告

带宽测试实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　姚杰

院（系、所）　机械科学与工程学院

2024 年 12 月 28 日

目　　录

1 （绪论）

（姚老师给的模板太完善导致有点不知道该怎么写这个报告了 QAQ，本质上主要工作就是把姚老师的模板函数复制一份稍加修改就写完了函数，把姚老师的代码复制一份稍加修改就完成了带宽测试）

2 纯 C 语言带宽测试

本章节主要关注的是通过利用 CPU 的位宽、循环展开、使用预取指令来提高拷贝效率。同时为了降低本次实验的难度，拷贝操作均发生在对齐后的内存地址上，但需要了解内存地址对齐对于内存拷贝效率的影响。

内存对齐核心公式是 $align\_pos = \lfloor pos + (block - 1)/block \rfloor * block$，其中 $alignpos$ 表示对齐到的位置，pos 表示原本的位置，$block$ 为对齐的块大小，该公式将 pos 对齐到 $block$ 的下一个整数倍。在 $block$ 为 2 的整数倍时，可用 C 语言运算符表示为 $align\_pos = pos + (block - 1)\&\sim(block - 1)$。需要完成的函数 *alloc_four_aligned_buffers* 只需要按照需要分配的内存累加指针并对齐到指定块大小即可。

CPU 位宽的充分使用在 C 语言中的体现时每次内存复制时，使用 int64_t* 类型的指针，从而使得 CPU 的 64 位充分使用。同时，我们在每次循环中，进行数次内存的拷贝，进行人为的循环展开，以尽可能地提高程序地效率。由于在地址递增取值的情况下，可能出现无效的硬件地址预取，所以我们还是用反向拷贝的方法，优化程序的效果。

本章节中需要进行一定探索的是通过 *__builting_prefetch* 预取指令优化，预取太晚，在硬件预取之后，那么只是徒增了预取指令本身的开销，而预取过早，则可能在预取数据使用前，又被从 cache 中换出，因此需要一定的尝试探索预取指令的发生时机。同时，预取指令取出的 cache 行大小也是需要确定的，分别写出假设预取 32 字节和预取 64 字节的代码进行对比。首先，通过实验发现预取当前存取的位置后（由于是反向，实际上按指针大小应该是前）448~512 字节的数据，较为理想。同时，对于假设预取 32 字节时，使用两条语句预取该片段 64 字

节的数据，而假设预取长度位 64 字节时，代码使用一条语句，如果测试结果中，64 字节性能更好，则表示预取长度为 64 字节。

表 2-1　CPU 预取优化实验数据 单位 MB/s

次数	假设长度预取长度 32			假设长度预取长度 64		
	优化	不优化	提升	优化	不优化	提升
1	11 864.7	11 635.7	0.0197	12 098.8	11 888.2	0.0177
2	11 820.1	11 593.1	0.0196	12 502.7	12 302.8	0.0162
3	11 728.8	11 581.2	0.0127	12 337.8	12 116.2	0.0183
4	11 862.8	11 612.9	0.0215	11 895.7	11 699.4	0.0168
5	12 509.2	12 218.2	0.0238	12 449.8	12 192.9	0.0211
6	11 842.2	11 643.2	0.0171	12 405.9	12 215.2	0.0156
7	12 415.6	12 135.3	0.0231	12 406.4	12 227.9	0.0146
8	11 868.3	11 637.8	0.0198	12 469.6	12 294.3	0.0143
9	12 311.4	12 096	0.0178	12 439.3	12 202.5	0.0194
10	12 487.4	12 213.1	0.0225	12 052.6	11 911.2	0.0119
平均			0.0198			0.0166

表 2-1 为进行 10 次实验记录的到的数据，可以推测预取指令预取 cache 行大小为 32 字节。

实验过程中还发现，正向拷贝与逆向拷贝差异不大，一般情况下正向内存拷贝稍快于逆向，推测是加法运算的速度快于减法，同时，由于在本实验中简化了场景，仅对大块已对齐内存进行拷贝，因此正向拷贝的无效预取效能损失极少，可以忽略。

3　汇编内联带宽测试

本章节主要考虑的问题也是充分利用 CPU 的位宽以及预取。

实验中发现，nontemporal copy 效率高于普通的 copy，查找资料得到的解释是，*moventdq* 所执行的是以非缓存形式写入，写入数据绕过缓存，直接写入内存，因此，能够提高效率，提高性能大约 40%。

而对于预取，则遇到了失败。*prefetchnta* 尝试过无数种添加该指令的时间点，均以无效告终，因此也无法测试其预取大小，但是从参考文档[1] 中可以了解到，其至少取 32 字节数据。

综合纯 C 代码和汇编内联代码，可以在测试框架下得到结果如表 3-1 ，从中可以发现，C 语言实现中，使用预取优化可以得到性能的微小提升，接近于汇编

使用 *movedqa* 语句的内存拷贝实现。而 C 库函数 memcpy 的性能，则接近于汇编中使用 *moventdq* 语句的内存拷贝，性能最好的是 nontemporal copy，可以达到 17263.8MB/s 的带宽。

表 3-1　内存带宽测试结果

测试项目	内存带宽 (MB/s)	样本标准偏差 (%)
C copy	11737.3	1.9
C copy backwards	11735.3	0.4
C copy prefetched	12064.6	0.3
C fill	11896.8	0.2
C *memcpy*()	17055.4	2.8
SSE2 copy	12081.1	0.4
SSE2 nontemporal copy	17263.8	1.7
SSE2 nontemporal copy prefetched	17153.1	0.7
SSE2 nontemporal fill	17253.2	2.9

4　多线程带宽测试

本章使用多线程测试带宽，并使用 CPU 亲和性 API 探索达到峰值性能的最小线程数。细节上对于 *pthread* 及 *sched_setaffinity* 的使用不做具体叙述。主要说明测试方案的选择及论述其合理性。

代码框架基本上未作太大修改，多线程版本相当于多个单线程版本同时运行。但是，单线程版本中，以统计到的最大带宽作为实现方法的带宽结果表示，在该情况下是合理的，但是在多线程需要聚合时可能并非合理，因为很可能存在线程调度中，线程分别占用较多资源而不同时达到最大带宽。因此此处稍做的修改是，使多个线程在几乎相同的时间开始进行内存拷贝，并统计最终结束时间，如果结束时间接近，则认为是合理的，同时带宽聚合使用平均带宽而非最大带宽。在线程同步方面，使用了 Barrier 的策略，在每个线程绑定到对应 CPU 之后设置该同步点。

由于记录了每个线程的情况，输出数据较多，因此以 9 个线程时为例分析。表 4-1 中数据表明，线程基本同步开始进行，符合预期，同时可以由记过可以发现，最大速度之和远大于平均速度之和，高出 33% 显然该结果是不合适的。同时平均速度的方差较大，说明了线程调度中存在一定的负载不平衡问题，而 id 为 8 的线程平均速度远远快于其他线程，可能与 CPU 底层调度逻辑相关。

表 4-1 线程性能数据

线程 id	平均速度 (MB/s)	开始时间 (ms)	结束时间 (ms)	最大速度 (MB/s)
0	2347.2	347900.4	347905.6	5265.4
1	2635.8	347900.4	347904.5	5305.7
2	2340.9	347900.4	347905.6	5258.8
3	2635.8	347900.4	347904.5	5306.3
4	4198.0	347900.4	347905.6	5314.7
5	5267.6	347900.4	347904.5	5429.4
6	4198.5	347900.4	347905.6	5295.0
7	5267.4	347900.4	347904.5	5437.5
8	7540.7	347900.4	347903.3	7573.1
均值	4048.0	347900.4	347904.8	
方差	1768.3	1.0	1.3	
总和	36431.9			50185.9

为探索使带宽达到最大的最小线程数，基于 CPU 编号的规律，可以猜测将线程分配到不同 CPU 能够使用最少线程达到最大的带宽占用，主观猜测将线程分别分配到 0~13，14~27，28~41，42~55 CPU 节点各一个即可达到最大带宽，但是测试中数据表现上并非如此，分配一个线程 0~13，一个 14~27，两个线程后，即可达到 33781.3 MB/s 带宽，然而分配 0~13，14~27，28~41，分别一个后，带宽下降到 22256.1 MB/s 并出现了较高的负载不平衡问题，直到分配 9 个线程，按均匀分配的方法（多出线程仍然按照该顺序分配），达到了 36431.9MB/s 带宽，最高带宽出现在 14 线程时，为 42013.0 MB/s。当然，上述结果只有一定的参考性，因为实验时间过长，服务器可能有其他程序的占用变化。

5 调试记录

不知道为什么不能使用 vscode 的 GUI 进行代码调试，因此接触了 gdb 命令行调试方法，感受到了其在多线程调试中的巨大作用和在各种情况均能使用的泛用性。

调试过程中遇到的最大的问题就是因为疏忽将多线程内存拷贝中，源地址和目的地址写为一个地址，从而导致出现了性能上急剧下降的情况。由于缺乏基本常识，我以为单独开一个线程的运行本来就会有所下降，因此前期的调试并未走在正确的道路上。直到联系了姚老师，他指出了这个问题，还耐心帮我看了代码，并建议首先使用 *memcpy* 测试，使用 *memcpy* 的测试中立马发现了问题，带宽变成了接近 inf，因此我是用调试功能查看了进入 *memcpy* 函数时的状态，然后发现传入的两个参数相同，定位到了问题。这次 debug 的经历让我明白了，debug 过程中需要尽可能使用稳定的函数一步一步测试，同时需要有一定的经验常识，否则容易误入歧途。

致谢

感谢姚老师在我调试出问题的时候的帮助！没有姚老师我将无法完成本次任务，再次感谢。

参考文献

[1] UNKOWN. Prefetch Data Into Caches[EB/OL]. .
https://www.felixcloutier.com/x86/prefetchh.