CacheLab 报告

姓名：卢虹宇

学号：2023202269

csim 分数	case1 speedup	case2 speedup	case3 speedup	weighted speedup
100	8.68	5.92	5.69	6.67

Autograder 截图（网络有问题push不上去，所以用了本地截图）：

```
case  | miss_cache | miss_reg | latency | speedup
------+------------+----------+---------+------------------
case0 | 2          | 20       | 50      | 1.0
case1 | 250        | 5632     | 9382    | 8.677041142613515
case2 | 3416       | 53248    | 104488  | 5.919914248526147
case3 | 4597       | 53940    | 122895  | 5.694007079213963
Weighted Speedup: 6.6567
```

```
status | trace_file         | (s, E, b) | ref: (hits, misses, evictions) | handin: (hits, misses, evictions)
-------+--------------------+-----------+--------------------------------+---------------------------------
OK     | traces/yi2.trace   | (5, 1, 5) | (15, 1, 0)                     | (15, 1, 0)
OK     | traces/yi.trace    | (5, 1, 5) | (3, 4, 0)                      | (3, 4, 0)
OK     | traces/dave.trace  | (5, 1, 5) | (2, 3, 0)                      | (2, 3, 0)
OK     | traces/trans.trace | (5, 1, 5) | (211, 7, 0)                    | (211, 7, 0)
OK     | traces/long.trace  | (5, 1, 5) | (246213, 21775, 21743)         | (246213, 21775, 21743)
OK     | traces/yi2.trace   | (2, 4, 3) | (14, 2, 0)                     | (14, 2, 0)
OK     | traces/yi.trace    | (2, 4, 3) | (2, 5, 0)                      | (2, 5, 0)
OK     | traces/dave.trace  | (2, 4, 3) | (0, 5, 0)                      | (0, 5, 0)
OK     | traces/trans.trace | (2, 4, 3) | (192, 26, 10)                  | (192, 26, 10)
OK     | traces/long.trace  | (2, 4, 3) | (243398, 24590, 24574)         | (243398, 24590, 24574)
OK     | traces/yi2.trace   | (4, 2, 4) | (15, 1, 0)                     | (15, 1, 0)
OK     | traces/yi.trace    | (4, 2, 4) | (2, 5, 2)                      | (2, 5, 2)
OK     | traces/dave.trace  | (4, 2, 4) | (2, 3, 0)                      | (2, 3, 0)
OK     | traces/trans.trace | (4, 2, 4) | (206, 12, 0)                   | (206, 12, 0)
OK     | traces/long.trace  | (4, 2, 4) | (247163, 20825, 20793)         | (247163, 20825, 20793)
OK     | traces/yi2.trace   | (1, 1, 1) | (8, 8, 6)                      | (8, 8, 6)
OK     | traces/yi.trace    | (1, 1, 1) | (0, 7, 5)                      | (0, 7, 5)
OK     | traces/dave.trace  | (1, 1, 1) | (0, 5, 4)                      | (0, 5, 4)
OK     | traces/trans.trace | (1, 1, 1) | (25, 193, 192)                 | (25, 193, 192)
OK     | traces/long.trace  | (1, 1, 1) | (35393, 232595, 232594)        | (35393, 232595, 232594)
Score: 100.00
```

Part A: cache 模拟器

实现简述

由于我们只需要模拟hit还是miss,所以缓冲行（`line`）内的data部分可以去掉，保留有效位和tag位即可。缓冲区类中实现一个元素为缓冲行的链表，并维护变量`hits`，`misses`，`evictions`，每次load一个`address`时检查有无匹配的有效`line`,有则将该`line`头插到链表头,并且`hits++`;没有就检查`line`数是否达到上限，如果达到上限就删除末尾`line`,`evictions++`，然后再将`address`对应`line`插到链表头,`miss++`.

亮点

无

Part B: 矩阵乘法优化

亮点

1.分块 对于case1,一级分块就以达到我的算法的极限（使用二级分块也可以但感觉没必要）；对于case2,二级分块才能达到性能极限；对于case3,由于参数为质数，参考了pjd师兄使用rust批量修改参数的项目，得出当前算法下最优的二级分块参数。

2.使用寄存器来预存被大量使用的矩阵数据，避免反复取用，减少cache_miss和reg_miss（实际上寄存器总是不够用的，寄存器越多，两种miss都能持续下降，一直到理论最优）

3.调整循环顺序，通过调整循环顺序来使内层循环尽量按照行访问矩阵，可以有效减少miss

4.调整case_3中内层循环中未用寄存器储存的矩阵快的`offset`,最初是想把B矩阵和C矩阵整体都搬迁对齐，后来意识到不仅仅是要开头对齐而是要每一行都对齐才有意义，而buffer显然不够用，所以退而求其次选择只将内层循环的矩阵块搬迁对齐。

未被采纳的想法

1.B矩阵转置：由于B矩阵转置后使用naive的算法也可以直接按行访问，可以避免大量miss，所以最初想尝试转置B矩阵，但是原地转置也会带来不可弥补的miss，而buffer也不够用，所以作罢。

2.多级分块，理论上讲多次分块可以降低方寸次数，但是经过实践发现miss不降反增，我认为分块级数应当与矩阵大小和缓冲块大小的比值有关，此处矩阵规模使用二级分块已经足够。

我认为的最优秀的实现排序

1. case2
2. case3
3. case1

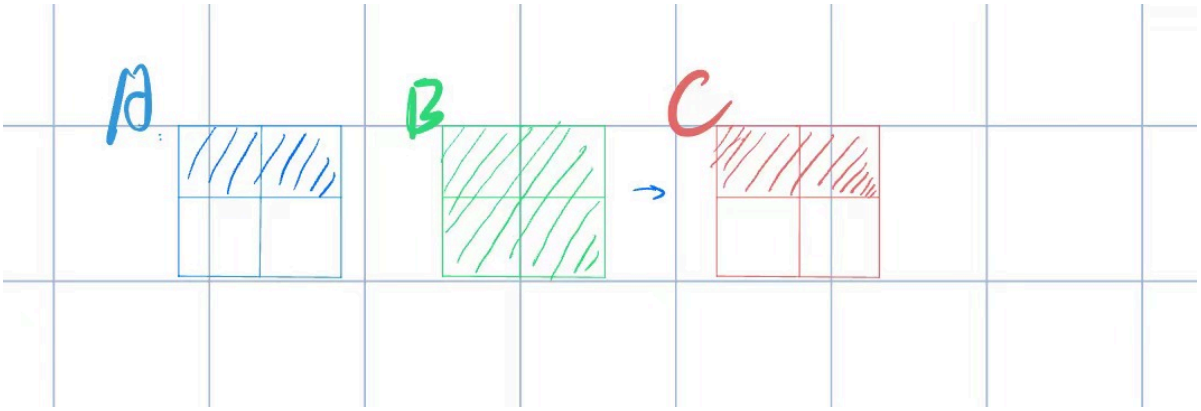
case2

miss_cache	miss_reg	latency	speedup
3416	53248	104488	5.9199

思考历程

循环调整2.7->一级分块3.3->在内层循环预储存矩阵B块4.2->变为预储存矩阵A和C块5.0->多级分块5.9（数字代表加速比）

循环调整和一级分块思路比较基础不再赘述，对于预储存，我在内层循环是按照`ikj`的顺序访问A、B、C矩阵（A块：`i×k`，B块：`k×j` C块：`i×j`），可以发现在一次最内部循环中B对应的访问数据总是在发生变化，当走完`k`、`j`的一次循环B的整个矩阵已经被访问完但是A和C都只访问了一行（如图），



所以最初我考虑是否能把内层循环的B块先存进寄存器，再把A和C的当前访问行存进寄存器，这样可以最大程度提高访存效率，但是经过计算发现，刨除循环所用的`reg`和充当临时变量的`reg`,留给我用于预存的`reg`不能多于19个（通过某些极端手段可以扩展预留个数），而分块的尺寸又最好能与cache大小匹配，这意味着分块行尺寸尽量往8靠（缓冲行一行能存8个数据），所以同时预存B块和A和C的当前访问行是行不通的。

于是我最初选择存B块，经过尝试选择了2×8这个尺寸，而A分块则是4×2，C为4×8,但效果一直不好，加速比一直在4左右，于是我将策略改为缓存A和C对应行，加速比来到了5.07。

最后通过查看blog尝试考虑二级分块，经过二级分块处理，并调整参数（尽量是32的因子），加速比来到了5.91。

miss分析

最后选择的参数为：一级分块：32×4×32，二级分块：2×4×8

一级分块后，相当于是1×8的矩阵A乘8×1的矩阵B，有8对块相乘，而对于每一个块内则有64对二级块相乘，每一对二级块为：2×4 4×8，按照我的预取策略，cache_miss:应该有A2次，B4次，C1次，共7次；miss,reg_miss:A8次，B64次（遍历完A时B会被访问两次），C16次，共76次，最后结果为：cache_miss:8×64×7=3584次，reg_miss:8×64×88=45056，

而实际cache_miss=3416,reg_miss=53248。我认为cache_miss的误差可能是B的二级分块行尺寸为4,再被加载进缓冲区后剩下的4个数据得到了复用。而reg_miss的误差是由于我只计算了取矩阵数据的reg_miss,还不包括用于循环和临时变量的reg_miss。

case1

miss_cache	miss_reg	latency	speedup
250	5632	9382	8.677

#####

思考历程

循环调整2.2->在内层循环预储存矩阵B块3.7->一级分块4.2->多级分块3.0(舍弃)->变为预储存矩阵A和C块6.2->调整参数8.7（数字代表加速比）

思路历程与2大致相同，只不过没有采用二级分块，经过尝试后采用分块参数1×8×8。

miss分析

分块后，相当于是16×2的矩阵A乘2×2的矩阵B，有32对块相乘，每一对二级块为：1×8 8×8，按照我的预取策略，cache_miss:应该有A1次，B8次，C1次，共10次；miss,reg_miss:A 8次，B 64次，C8次，共80次，最后结果为：cache_miss:32×10=320次，reg_miss:32×80=2560次。

而实际cache_miss=250次,reg_miss=5632次。我认为cache_miss的误差可能由于分块会被多次利用当再次访问时未被驱逐的数据就会被重复利用。而reg_miss的误差我也想不到一个很好的解释。

case3

miss_cache	miss_reg	latency	speedup
4597	53940	122895	5.69

#####

思考历程

循环调整+在内层循环预储存矩阵B块3.3->多级分块+变为预储存矩阵A和C块4.3->调参+对齐内层B块offset5.69

思路历程与2大致相同，但是在case2的基础上增加offset的优化和批量修改超参数来调参，最后分块参数为：31×15×4

miss分析

由于只有内层B块对齐所以不太好考虑，将矩阵乘法尺寸按大约32×32×32考虑，所以下面分析不太严谨，分块后，相当于是1×2的矩阵A乘2×8的矩阵B，有16对块相乘，每一对块为：31×15 15×4，按照我的预取策略，cache_miss:应该有A 62次，B 30次，C31次，共123次；miss,reg_miss:A465次，B1860次（遍历完A时B会被访问31次），C124次，共2449次，最后结果为：cache_miss:16×123=1968次，reg_miss:16×2449=39184，这中间的误差可能是由未对齐导致。

反馈/收获/感悟/总结

这个lab大概花费了我30个小时时间，大量时间都花在了纠结于哪种思路和查找资料上，说实话感觉像在玩一个脑筋急转弯，不太建议把这个lab做成打榜的形式，因为到最后可能会演变成大家都在扣超参数和寄存器，边际效应比较低，设置一个较高的base_line我觉得是更可行的方式。

参考的重要资料

[深入浅出GPU优化系列：GEMM优化（一） - 知乎]

[深入浅出GPU优化系列：GEMM优化（二） - 知乎]

https://github.com/panjd123/parabuild-rust