对于pro2，我所做的所有修改，都在git的modified_version_pro2分支下；对于pro3的修改都在modified_version_pro3分支下；原文件在git的master分支下。可以将这些分支对比，得到修改的部分。

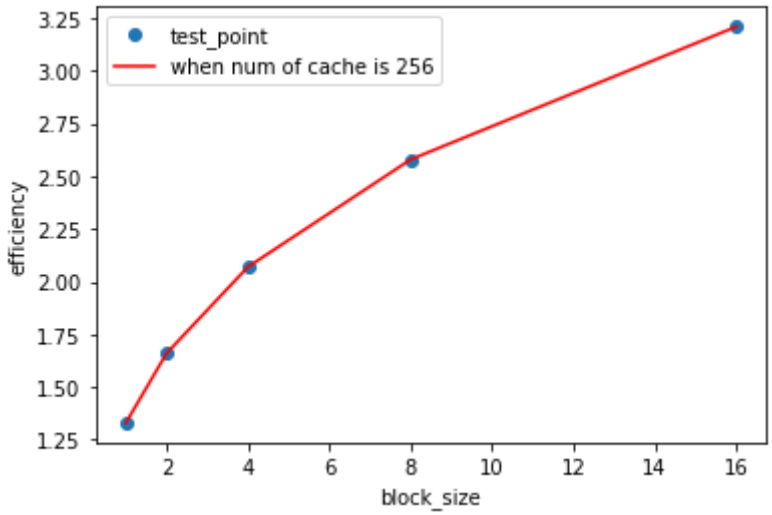
pro1

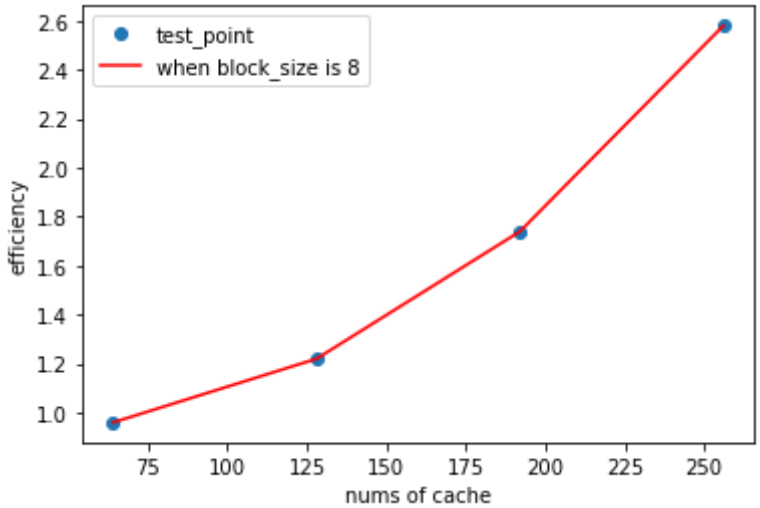
- 内置的cache项数为：**256**
- cache的组织形式为：（全相联？等等） **直接相连**
- cache的block_size为： **8B**
- cache的替换策略为： **随机替换**
- 内置的conv实现方案可以提高效率多少倍？**假设主存的访问时间为cache的10倍，则整体访存效率提高了2.57倍！**

pro2

- 修改cache的静态指标，如项数、block_size，并分析对应的结果

  我按照教科书上的讲解，尝试增大cache块的大小，以期降低cache访问缺失率。下图是我改变cache块大小，对应访问效率的曲线图：



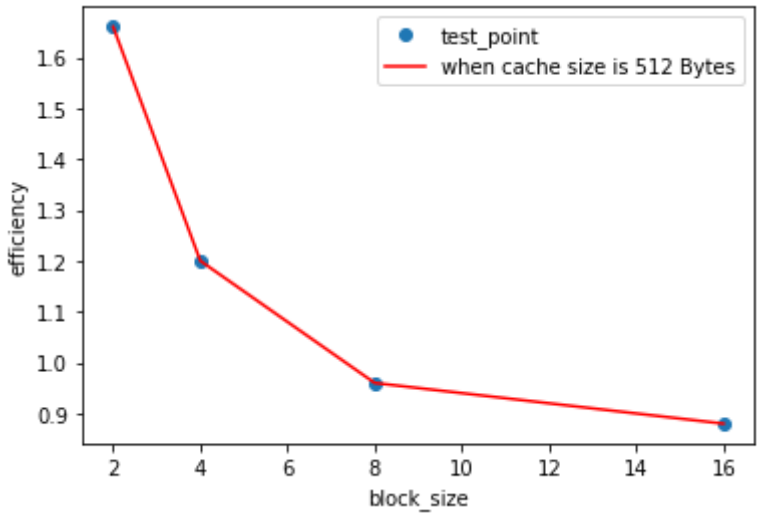


  目前比较明显能看出的影响，是当cache数量和cache大小增加时，明显能发现与内存交换减少，存储访问性能增加。

然而，现在暂时看不出过大的影响，可能是由于程序cache参数65536的限制，无法大到看出区别（大会收到硬件限制？）；也可能是因为程序计算效率的方式不一定正确，当cache数量和大小增加时，访问cache 和访问内存的时间比例未必是10:1了。

如果暴力地将cache nums增大，也可以得到相当可观的改观，可以将访问性能提升至**9.75倍**！如果增长到cache nums=1048576，cache block_size=16，甚至可以达到9.95倍！这样的方法其实相当于把所有主存内容全部load进了cache中，而抛弃主存于不顾。但道理类似上面所说，可能在暴力增大了cache nums后，访问cache的开销也会同步增大，cache和main memory的性能比未必还是10:1了。因此，在接下来的优化中，我**假设cache nums最大到256，而不允许其无限制的暴力增长。**

如果按照内存总量不变，cache块大小变化的角度来分析，那么得到的曲线是这样的：



看来还是cache块大小的减少，对cache性能的影响更明显。

本实验的具体测试数据可见附录，下同。

- 修改卷积计算的6层循环的顺序，找到一种更优的循环方案

  首先，根据课内知识，我认为应当在循环时将矩阵的高维放在循环的外侧，低维放在循环的内侧，以期获得最大的空间局部性和时间局部性。因此，按照此原则，我首先采取将第一个维度放置in_channel变量和out_channel变量，然后下一个维度放置h和H维度，最后放置w和W变量，即遍历顺序为$i \to j \to h \to H \to w \to W$。这样的排布收获了非常好的效果，cache的性能从原来的2.58倍提升到了**9.62倍**！为了验证到底是两个channel变量顺序改变的效果明显，还是h和w变量改变的效果明显，我交换了h和w变量的顺序，调整成了$i \to j \to w \to W \to h \to H$，发现提升效果依旧很显著，达到了**9.57倍**

  。看来是最外层的channel变量影响最为严重。为了验证这一点，我将j变量移到了最后，将顺序变成了$i \to w \to W \to h \to H \to j$，效果直接下降到了**2.1倍**。

  以我看，这样的效果之所以出现，还是因为访问局部性的问题。如果访问局部性较好，可以充分利用cache中已经从主存拷贝好的数据，防止cache频繁调度数据，从而提升性能。如果遍历顺序不佳，比如将channel变量放在最后遍历，那么访问的局部性会受到很大程度的破坏，离散的访问导致cache反复从主存拷贝数据，性能得到损失。

- 修改cache的替换策略为FIFO（可能需要额外设置变量来记录相关信息）

  我使用了python自带的队列包来实现FIFO策略。每当cache中存入数据时，我让队列记录下此时cache存入的index；在选择index时，使用队列中最先出现的index。然而令人费解的是，采用了这种策略的kickoff，使得效率从2.57倍下降到了**1.29**倍！不清楚具体是什么原因，猜测可能与卷积运算独特的访问顺序有关。

  不过，如果把循环顺序改成最优顺序（即矩阵外层变量在外循环），那么替换策略的改变对效率没有显著影响，都是原来的**9.6**倍左右。代码的修改详见modified_version_pro2分支。

- 最终，我将pro2的所有优化方法汇集到了一起，循环顺序采用$i \to j \to h \to H \to w \to W$，替换策略采用FIFO，nums=256，blocksize=16，得到的优化效果为**9.84倍。**

pro3

- 我选择了将替换算法修改为LRU算法。我使用了python的字典方法，统计cache每个块的最近访问时间。当出现cache需要替换的情景时，选择最近访问时间最远的cache块删除，并替换成新块。这样的方法取得了1.66倍的效率提升，相比原来随机方法的2.57倍效率提升，反而出现了下降，这可能与卷积运算的访问顺序有关。

  本节具体代码实现详见modified_version_pro3分支，下同。

- 我选择了新建一个字典的方式来记录dirty位信息，并没有改原有数据结构。当新创建一个cache块时，dirty位置0；当删除cache块时，删除该index的dirty字典；当新数据写入cache块时，dirty位置1；最后，当dirty位为0时，不写回该cache块数据；为1才写回cache数据。这样收获的效果并不明显，效果仍然是前面的1.66倍，因为最后所有cache的dirty位均置为了1.

- 在与其他同学交流后，发现如果采用最初始的遍历顺序，其访问数据的空间局部性较差，难以发挥各种替换策略、dirty回写策略的优势。通过交流，我学习到了应该用最优遍历顺序，观察主存访问次数，来测试优化效果。得到的结果如下表所示：

方法:	**基准方法**	**使用dirty方法**	**使用FIFO方法**	**使用LRU方法**	**使用LRU和dirty方法的结合**
主存访问次数	44020	43941	44134	29436	29328
提升倍数	9.62	9.62	9.64	9.74	9.74

  可以看到，各种优化方法还是有一定效果的，尤其LRU方法效果显著。

附录

原始代码输出结果：nums = 256, block_size=8

```
进度：4.52%
进度：9.04%
进度：13.56%
进度：18.08%
进度：22.61%
进度：27.13%
进度：31.65%
进度：36.17%
进度：40.69%
进度：45.21%
进度：49.73%
进度：54.25%
进度：58.77%
进度：63.3%
进度：67.82%
进度：72.34%
进度：76.86%
进度：81.38%
进度：85.9%
进度：90.42%
进度：94.94%
进度：99.46%
Pass Correctness Check!
```

```
总共访存量为337.5MiB，在这过程中与主存交互字节数778.09MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存3187056次，假设主存的访问时间为cache的10倍，则整体访存效率提高了2.58倍！
进程已结束，退出代码 0
```

nums = 256, block_size = 16

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1142.761MiB，如果不使用cache，共需与主存交互5.273GiB字节数据！
总共访问cache 11059200次，总共访问主存2340374次，假设主存的访问时间为cache的10倍，则整体访存效率提高了3.21倍！
进程已结束，退出代码 0
```

nums = 256, block_size = 24

```
block_size>时16时，会出现list out of range
```

nums = 256, block_size = 4

```
总共访存量为337.5MiB，在这过程中与主存交互字节数516.807MiB，如果不使用cache，共需与主存交互1.318GiB字节数据！
总共访问cache 11059200次，总共访问主存4233682次，假设主存的访问时间为cache的10倍，则整体访存效率提高了2.07倍！
进程已结束，退出代码 0
```

nums = 256, block_size = 12

```
Cannot Pass Correctness Check!
```

nums = 256, block_size = 2

```
总共访存量为337.5MiB，在这过程中与主存交互字节数339.638MiB，如果不使用cache，共需与主存交互0.659GiB字节数据！
总共访问cache 11059200次，总共访问主存5564630次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.66倍！
```

nums = 256, block_size = 1

```
总共访存量为337.5MiB，在这过程中与主存交互字节数220.788MiB，如果不使用cache，共需与主存交互0.33GiB字节数据！
总共访问cache 11059200次，总共访问主存7234786次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.33倍！
```

nums = 128, block_size = 8

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1937.646MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存7936598次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.22倍！
```

nums = 192, block_size = 8

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1277.43MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存5232354次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.74倍！
```

nums = 64, block_size = 8

```
总共访存量为337.5MiB，在这过程中与主存交互字节数2552.347MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存10454412次，假设主存的访问时间为cache的10倍，则整体访存效率提高了0.96倍！
```

nums = 1024, block_size = 8

```
总共访存量为337.5MiB，在这过程中与主存交互字节数6.884MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存28198次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.75倍！
```

nums = 1048576, block_size = 8

```
总共访存量为337.5MiB，在这过程中与主存交互字节数2.687MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存11004次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.9倍！
```

nums = 1048576, block_size = 16

```
总共访存量为337.5MiB，在这过程中与主存交互字节数2.687MiB，如果不使用cache，共需与主存交互5.273GiB字节数据！
总共访问cache 11059200次，总共访问主存5502次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.95倍！
```

调调整循环顺序为$i \to j \to h \to H \to w \to W$:

```
总共访存量为337.5MiB，在这过程中与主存交互字节数10.792MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存44202次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.62倍！
```

调整循环顺序为$i \to j \to w \to W \to h \to H$:

```
总共访存量为337.5MiB，在这过程中与主存交互字节数12.013MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存49204次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.57倍！
```

调整循环顺序为$i \to w \to W \to h \to H \to j$:

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1016.108MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存4161980次，假设主存的访问时间为cache的10倍，则整体访存效率提高了2.1倍！
```

FIFO的kickoff方法：

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1821.094MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存7459200次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.29倍！
```

循环顺序为$i \to j \to h \to H \to w \to W$，并且替换策略为FIFO：

```
总共访存量为337.5MiB，在这过程中与主存交互字节数10.759MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存44134次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.64倍！
```

pro2的所有优化一起上：

```
总共访存量为337.5MiB，在这过程中与主存交互字节数8.622MiB，如果不使用cache，共需与主存交互5.273GiB字节数据！
总共访问cache 11059200次，总共访问主存17658次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.84倍！
```

pro3的LRU+dirty位：

```
总共访存量为337.5MiB，在这过程中与主存交互字节数1358.531MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存5564544次，假设主存的访问时间为cache的10倍，则整体访存效率提高了1.66倍！
```

改变循环顺序之后：

base：

总共访存量为337.5MiB，在这过程中与主存交互字节数10.747MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存44020次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.62倍！

单dirty：

总共访存量为337.5MiB，在这过程中与主存交互字节数10.728MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存43941次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.62倍！

单fifo：

总共访存量为337.5MiB，在这过程中与主存交互字节数10.759MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存44134次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.64倍！

单lru：

总共访存量为337.5MiB，在这过程中与主存交互字节数7.187MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存29436次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.74倍！
cache替换数：[14462]

lru+dirty：

总共访存量为337.5MiB，在这过程中与主存交互字节数7.16MiB，如果不使用cache，共需与主存交互2.637GiB字节数据！
总共访问cache 11059200次，总共访问主存29328次，假设主存的访问时间为cache的10倍，则整体访存效率提高了9.74倍！
cache替换数：[14462]