## 3. 矩阵优化运算

针对矩阵的运算，本实验提出5个大方向的改进，其中会有参数的测试。比如第3个分块法，本实验测试了多种分块的大小，得到了不一样的运行速度，并进行了比较。具体的改进方法如下表所示。

表3-1 优化思路方案

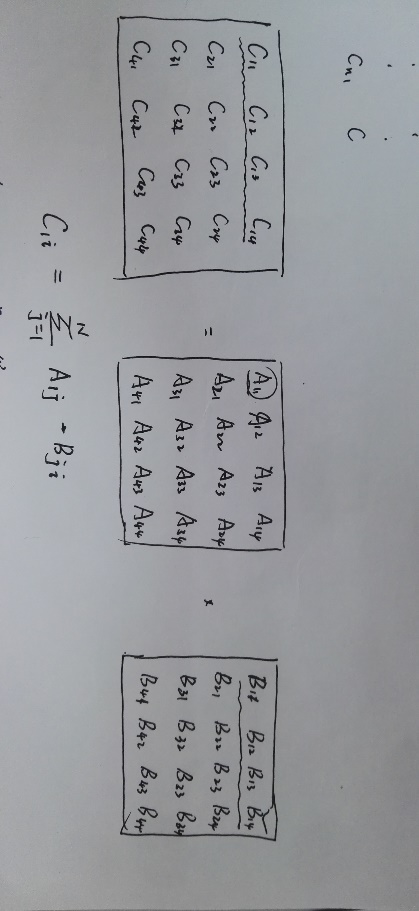
|  |  |
| --- | --- |
| 方案代码 | 方案思路 |
| M0 | 原始计算方法，直接将两个矩阵相乘 |
| M1 | 将矩阵乘法的最外侧循环和第二侧循环交换，这样我们就是用A矩阵的每一行乘B矩阵的同一列，得到C矩阵的一列。 |
| M2 | 对右矩阵B进行转置操作，因为cache中按行存储，而我们普通的矩阵乘法是按列读取B矩阵，转置后能提高cache命中率。 |
| M3 | 采用分块法，将局部相乘的结果进行累加，分块数为8。同样的原理，当数组非常大的时候cache无法一次性装载整个数组，如果采用分块的方法将增加cache命中率。 |
| M3\_1 | 采用分块法，分块数为16. |
| M3\_2 | 采用分块法，分块数为32. |
| M4 | 采用向量化进行计算。简单的原理就是每次取A矩阵一行一个数，乘以B矩阵一行。然后累加起来就是C矩阵的一行。 |
| M5 | 采用多核编程的思想，在M4的基础上改进，同时加载K个数据进行乘法运算。令K=4 |
| M5\_1 | 在M5的基础上令K=8 |
| M5\_2 | 在M5的基础上令K=14 |

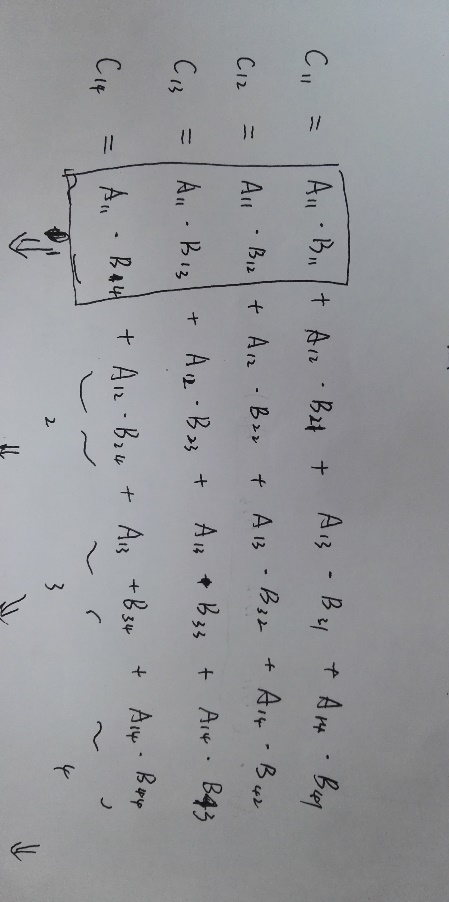
由于在课程中已经降到了上述前3中改进方案，所以实验中不再赘述。这里主要阐述一下M4和M5的改进思路。

**M4的改进思路**

在矩阵计算中，比如A矩阵乘以B矩阵。其实A矩阵中的一个数据和B中的每个数据都相乘了，但是在原始计算方法和前3次的改进中，其实每次从内存中取回一个数据我们只进行了一次的运算，然后又取出了下一个数据放入缓存相乘。也就是说我们只是增加cache的命中率，提高了数据获取的速度。但是我们并没有将缓存（寄存器）充分发挥作用。

所以在M4中，我们希望每次读取A矩阵的数据后能乘以一行B矩阵的数据，其实这就是向量化的思想，能够有效的减少A矩阵数据的访问。按照该思路，A矩阵每个元素其实只需要从cache或者内存中访问N次，而上述3中方法需要访问N的平方次。





由上图所示，我们每次取一个A元素，然后乘以B矩阵中的一行，依次保存到C矩阵中。依次计算，就可以实现该策略。

**M5的改进方法**

本实验在实验室的服务器上运行，CPU共14核。所以M5改进策略是基于多核编程的思想，在M4的基础上，每次取K个A矩阵元素同时和K行B矩阵元素相乘。基本思想还是向量化+多核编程的思想。下图是本实验机器CPU核数展示，所以在M5改进中，本实验测试了4核、8核、14核运行结果。

从结果发现，基本上在M4（可以认为是单核运算）的基础上成倍数提高，8核运行速度比4核高一倍。具体提高倍数请看图3-2。

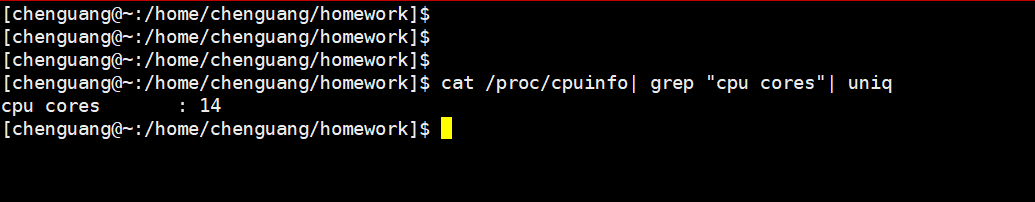


图3-1 CPU核数

接下来实验测试了不同算法在不同维度的运行时间，采用多次计算取平均值的方法，并在实验中采用了运行时间比率来绘制了一个折线图。运行时间比率是以M0方法为基准，用M0方法不同维度的运行时间除以其它方法在同等维度的运行时间，得到了一个改进的速度倍率关系。具体计算时间与维度的关系请看表3-2，绘制出的图像请看图3-1。

表3-2 计算时间与维度关系表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 计算数组维数N | 128 | 256 | 512 | 1024 | 2048 |
| M0(计算时间S) | 0.069577 | 0.383324 | 3.458911 | 27.965429 | 226.938403 |
| M1(计算时间S) | 0.045831 | 0.223914 | 1.762538 | 13.987237 | 111.535090 |
| M2(计算时间S) | 0.038977 | 0.201698 | 1.596884 | 12.702632 | 101.366805 |
| M3(计算时间S) | 0.036309 | 0.199731 | 1.718328 | 14.321457 | 114.679613 |
| M3\_1(计算时间S) | 0.034079 | 0.185665 | 1.564253 | 13.962758 | 111.820679 |
| M3\_2(计算时间S) | 0.032614 | 0.180920 | 1.588385 | 12.709746 | 133.493778 |
| M4(计算时间S) | 0.028246 | 0.148197 | 1.175409 | 9.370743 | 74.825910 |
| M5(计算时间S) | 0.012400 | 0.065342 | 0.521527 | 4.139669 | 33.119315 |
| M5\_2(计算时间S) | 0.006207 | 0.032777 | 0.260916 | 2.075956 | 16.568249 |
| M5\_3(计算时间S) | 0.003970 | 0.020740 | 0.165629 | 1.205998 | 9.624428 |

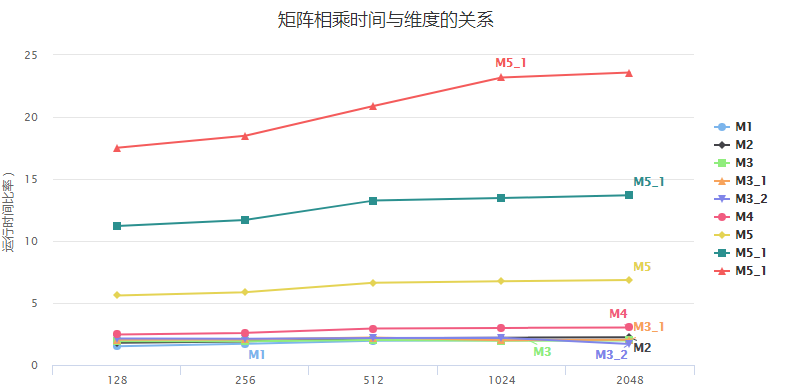


图3-2 矩阵相乘运行时间比率图

**总结与分析**

本实验主要改进了5个版本M1~M5，其中分块法M3和多核编程M5测试了多种参数。由上述运行时间比率的折线图可以看出。

* 采用M1方法大概能提高1.5倍左右的计算速度，提高原因是先计算列能够提高B矩阵元素cache的命中率。
* 采用M2方法大概能提高1.7倍左右的计算速度，提高原因与M1类似，cache调入B矩阵元素是按行调入，而我们是按列取，所以转置后能提高cache的命中率。
* 采用M3方法大概能提高2倍左右的计算速度，提高是原本无法一次性调入cache的矩阵，经过分块计算后，基本能一次调入cache，由此计算矩阵乘法访问元素能增加cache的命中率，从而提高运算速度。但是分块大小也将对运算速度产生一定影响。
* 采用M4方法大概能提高3倍左右的计算速度，提高原因是向量化操作后，A矩阵中每个元素不再必要读取N次，而只需要读取一次，这样大大的降低了访问A矩阵元素的时间。从而降低运算速度。
* 采用M5方法4核大概能提高6倍左右的计算速度，而且随着核数的增加，运算速度线性增加，当到14核时提高约为24倍的计算速度。这种方法是基于多核编程的思想，因为服务器是14核的，但是多核与多个CPU不同，多核共用数据通路，所以在M4中其实每次只计算了一次乘法，也就是只用了单核CPU在运行。在14核中，我们计算了14次乘法，大大的提高了运行速度。