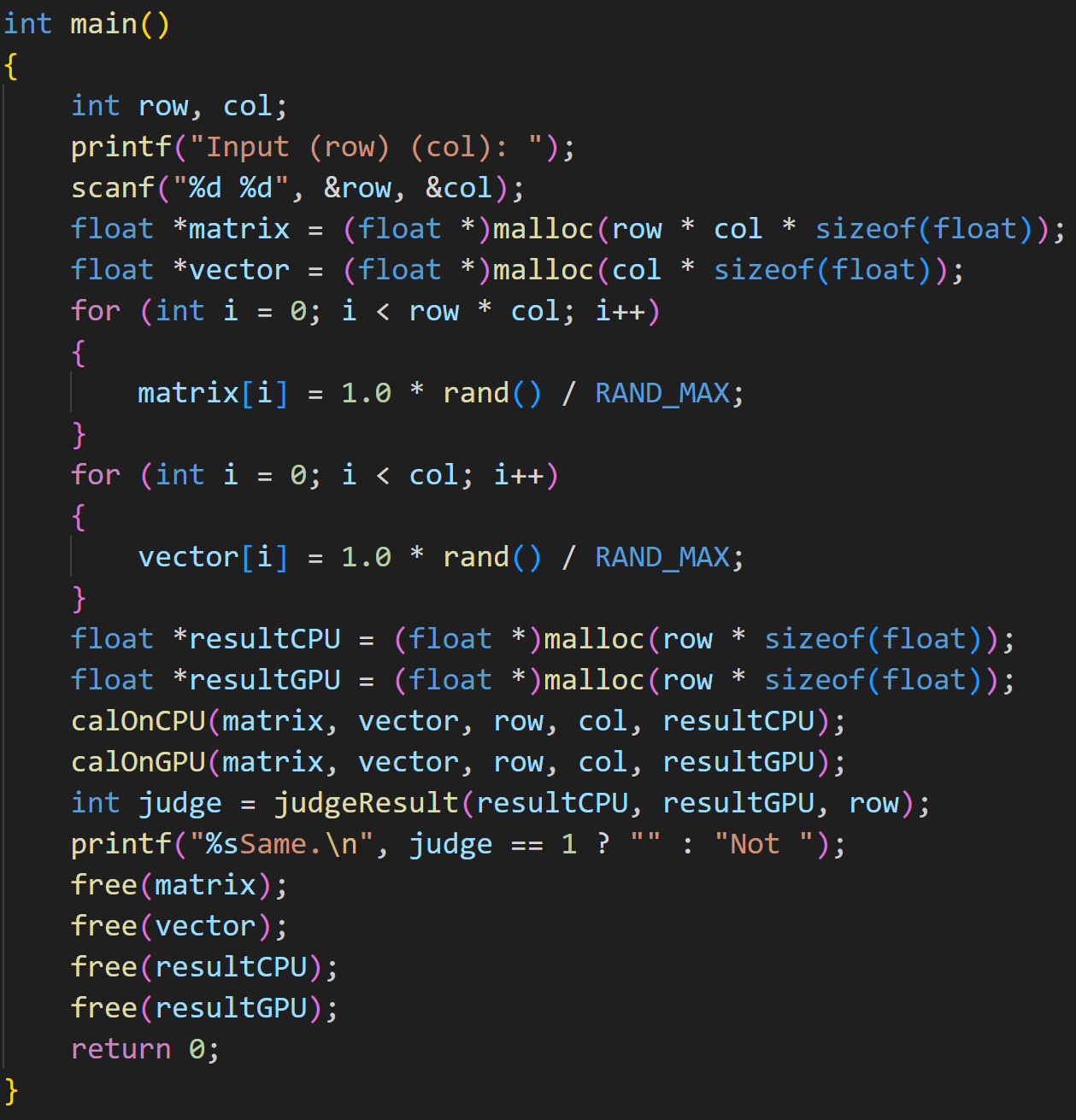
# 问题1

## 实验思路

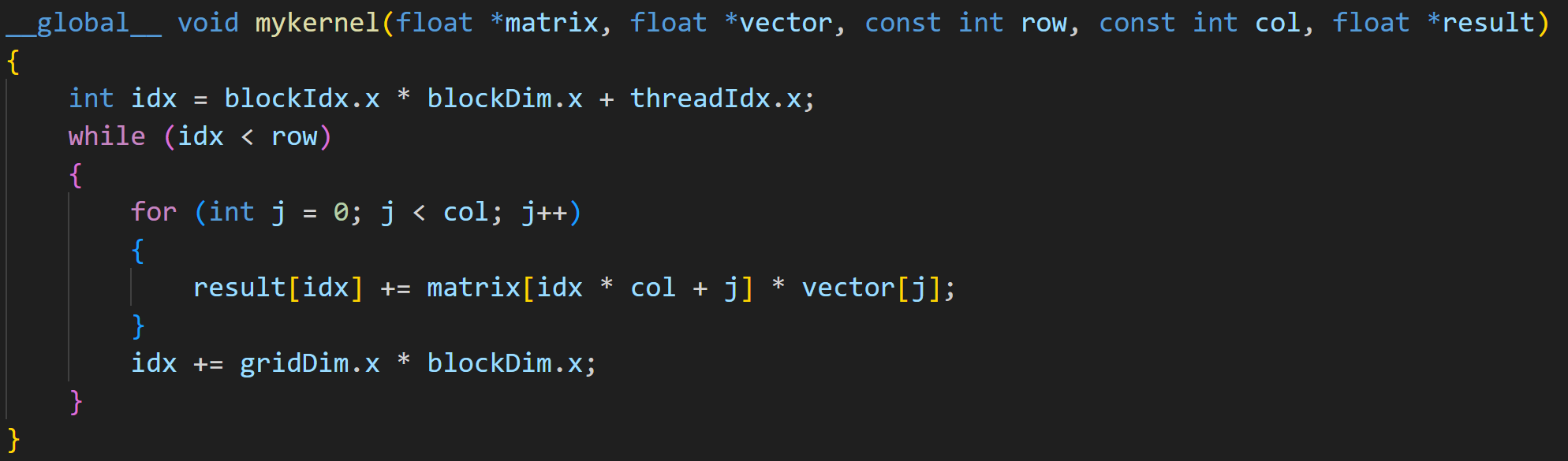
整体上，程序读入用户输入的矩阵行和列，然后生成随机的矩阵和向量。分别在CPU和GPU上执行矩阵与向量相乘，打印计算时间并对比计算结果。



对于GPU计算，每个Block的Thread固定为1024个。每个Grid的Block数目根据输入的数据来动态计算，最小为1，最大为128。另外，只计算了在GPU上运行的时间。

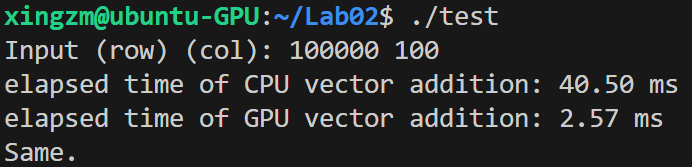


内核的代码如下所示。对row使用了1维的编号，使用while循环来防止计算不足或者越界。

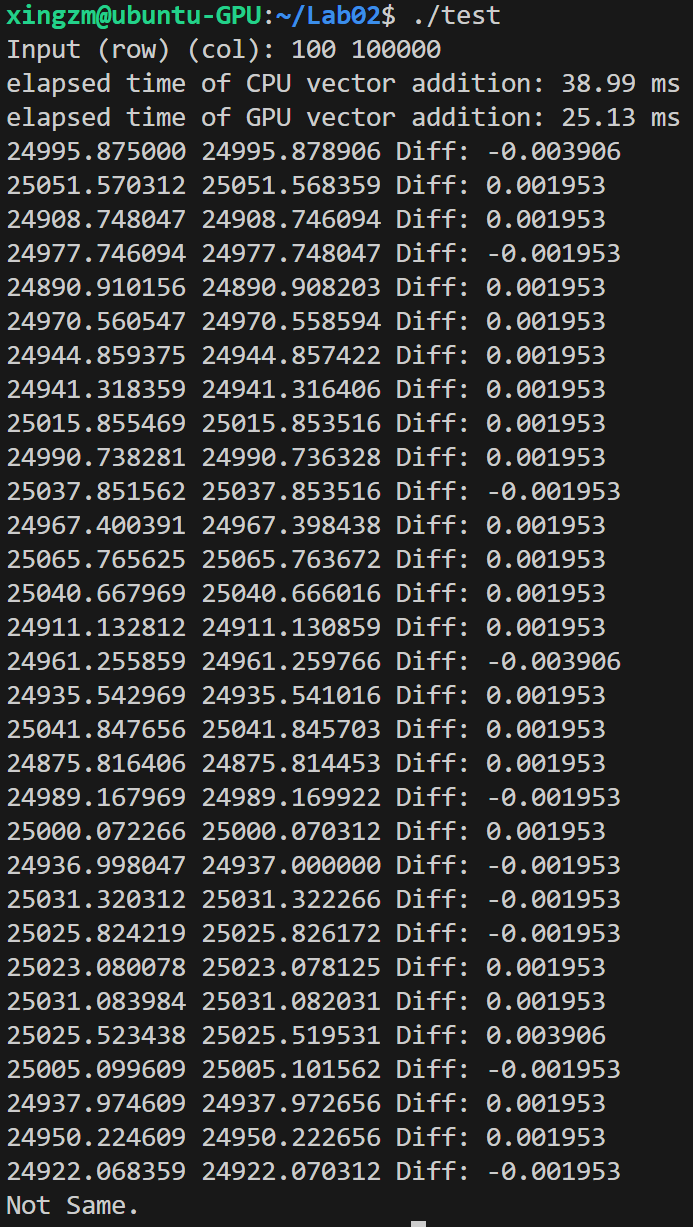


## 实验结果

在矩阵行数较大时，GPU计算速度明显快于CPU，提升了约15倍。



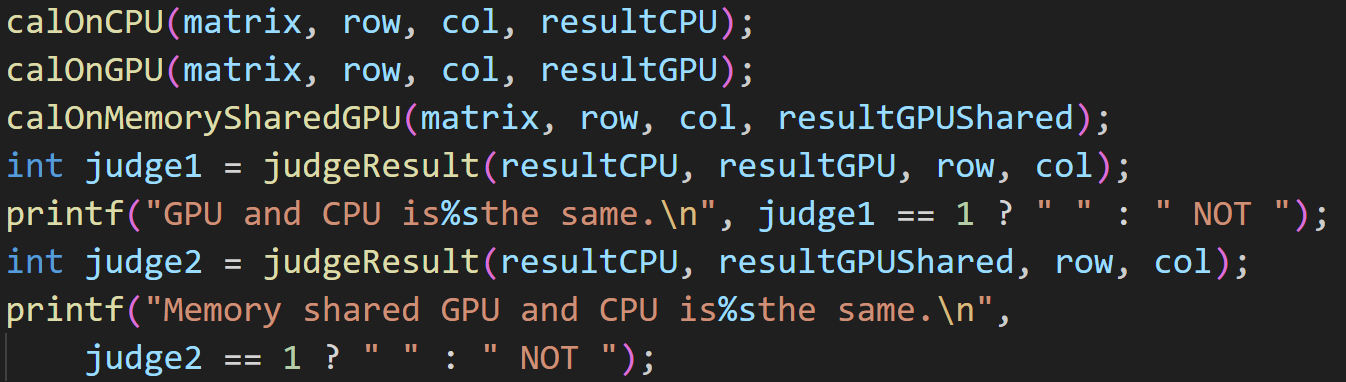
对比精度为。当向量列数较多时，GPU上运行的结果与CPU出现了一些不同。打印异常位置的元素，可以看到误差为量级。



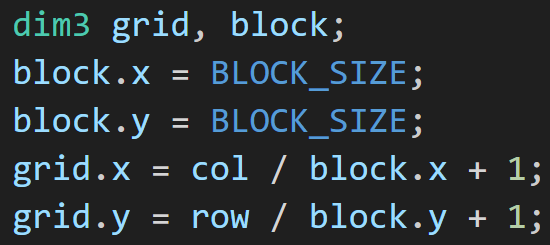
# 问题2

## 实验思路

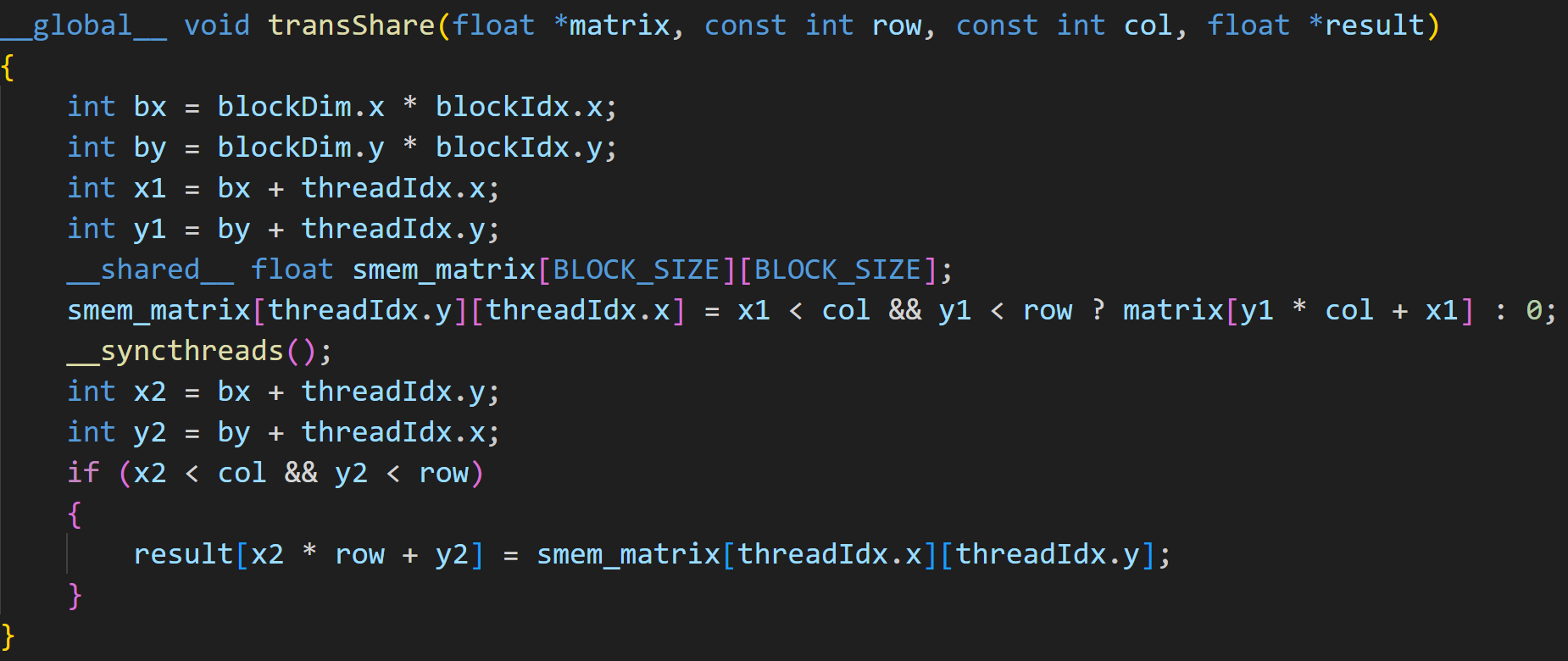
使用模块化编程，分别运行CPU、GPU和使用了共享内存的GPU的计算函数，然后判断两种GPU方式与CPU执行的结果是否一致。



两个维度的block大小都设置为了32，grid大小由矩阵的行列数目决定。保证了线程的充足性，每个线程对一个元素进行转置。在使用了共享内存的GPU计算中，x设置为列变量，y设置为了行变量。



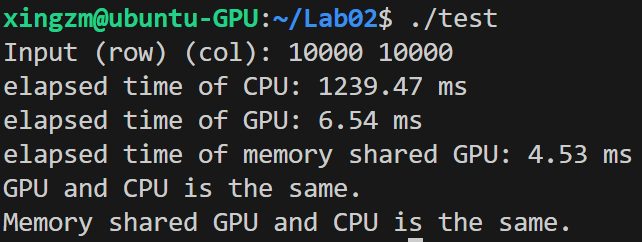
使用了共享内存的GPU计算内核代码如下图所示。在每个块中，声明块大小的共享内存区域。对于一个处理线程，将其值存放到共享内存变量的位置，其中、为线程号。然后等待线程同步，当这个块中所有线程都将值写入共享内存后，每个线程处理在块中对称的线程，因此是从共享内存中读位置的值。最后将这个值存放到result的转置位置，也就是，但此时行列发生变化，y是列。



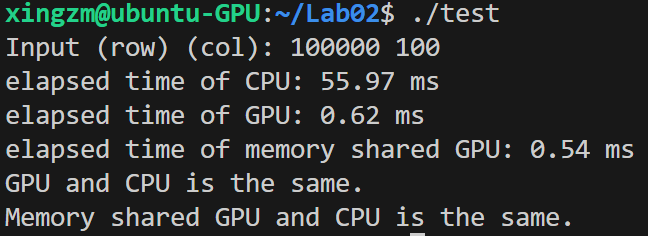
由于会申请到多余矩阵大小的线程，因此需要在读matrix数组和写result数组之前判断下标是否越界。在写共享内存时，越界了也要继续向后执行，因为对称的位置可能没有越界，需要在同步后读取并写回。

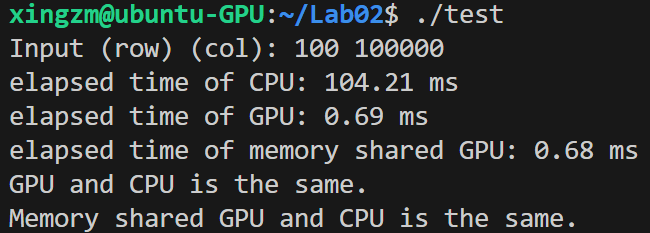
## 实验结果

在1000\*1000的方阵上分别运行3种代码，可以看到单纯使用GPU比CPU提升了189倍，用了共享内存的GPU版本又进一步地提升了1.4倍。



对于行和列相差较大的矩阵，共享内存的GPU计算依然是最快的且正确的，如下面两张图所示。

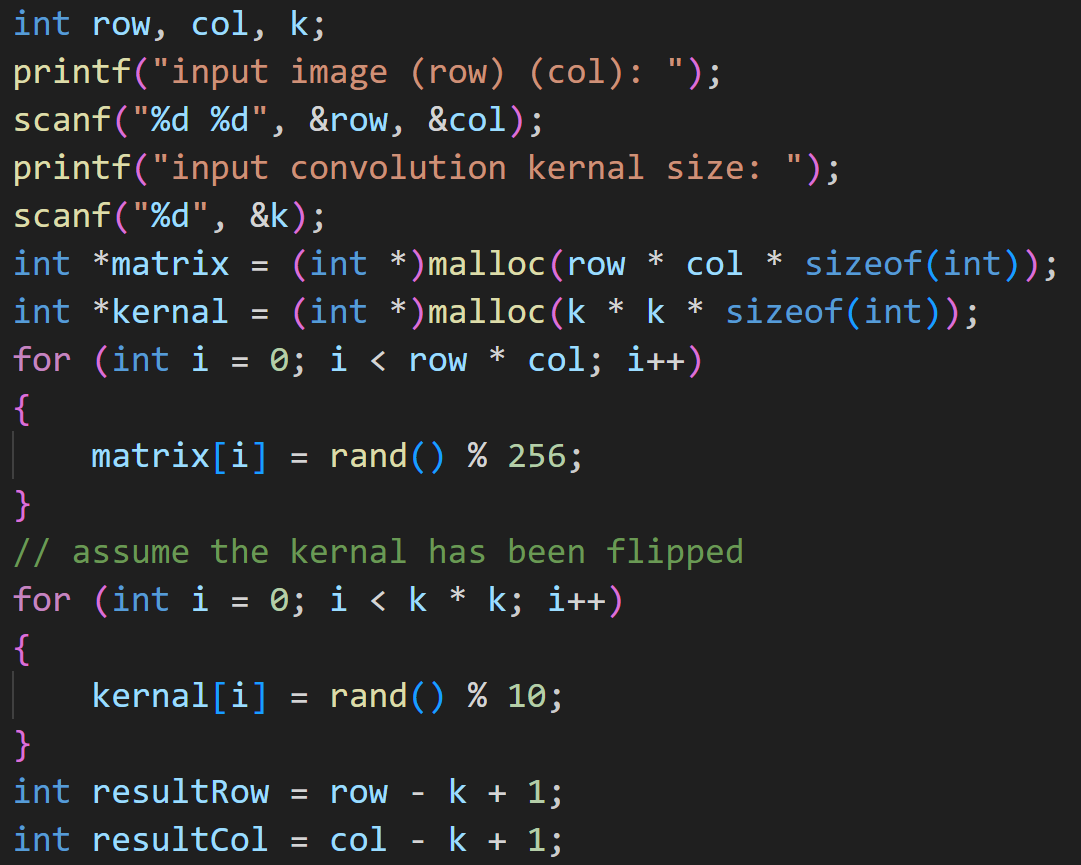




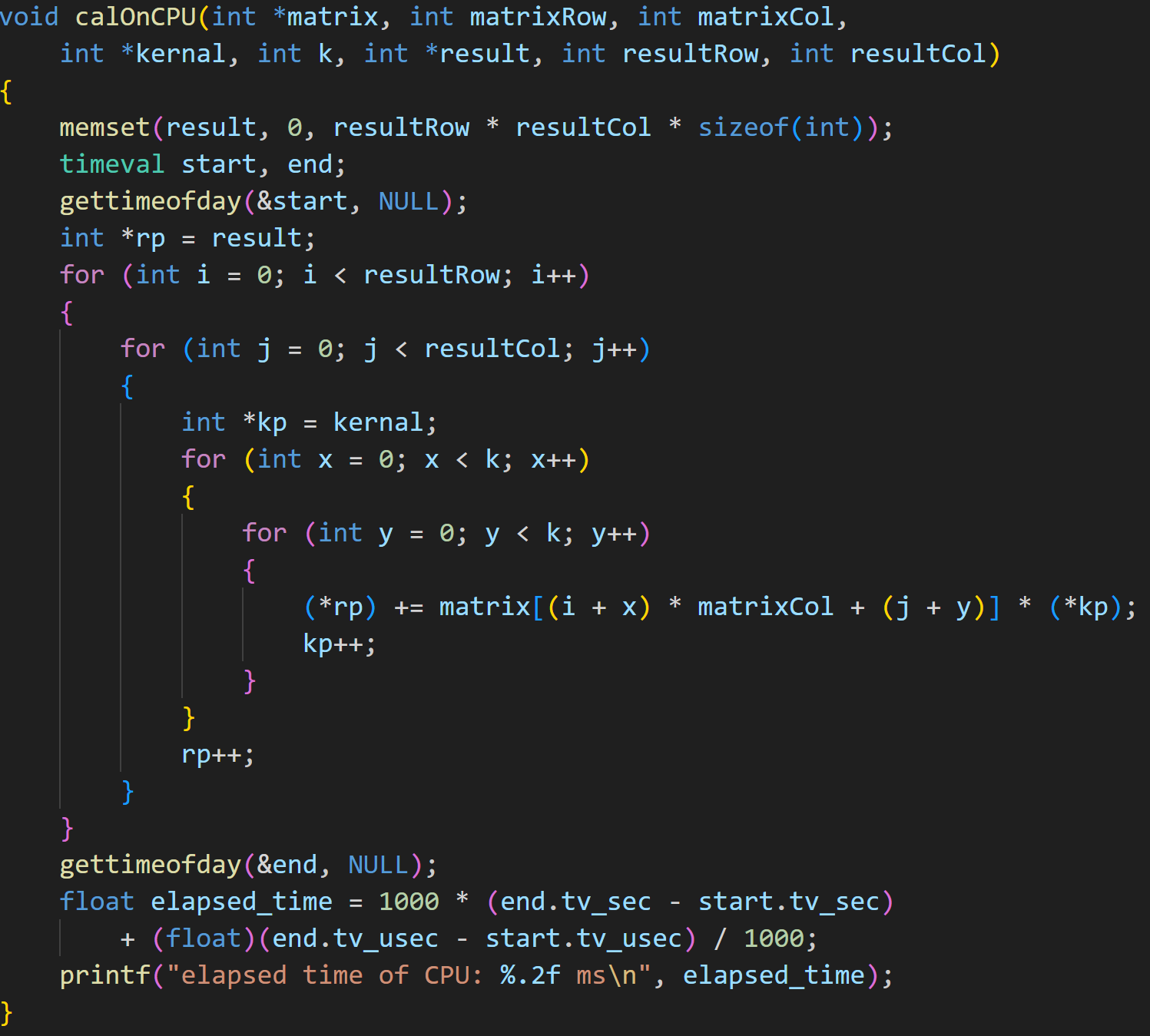
# 问题3

## 实验思路

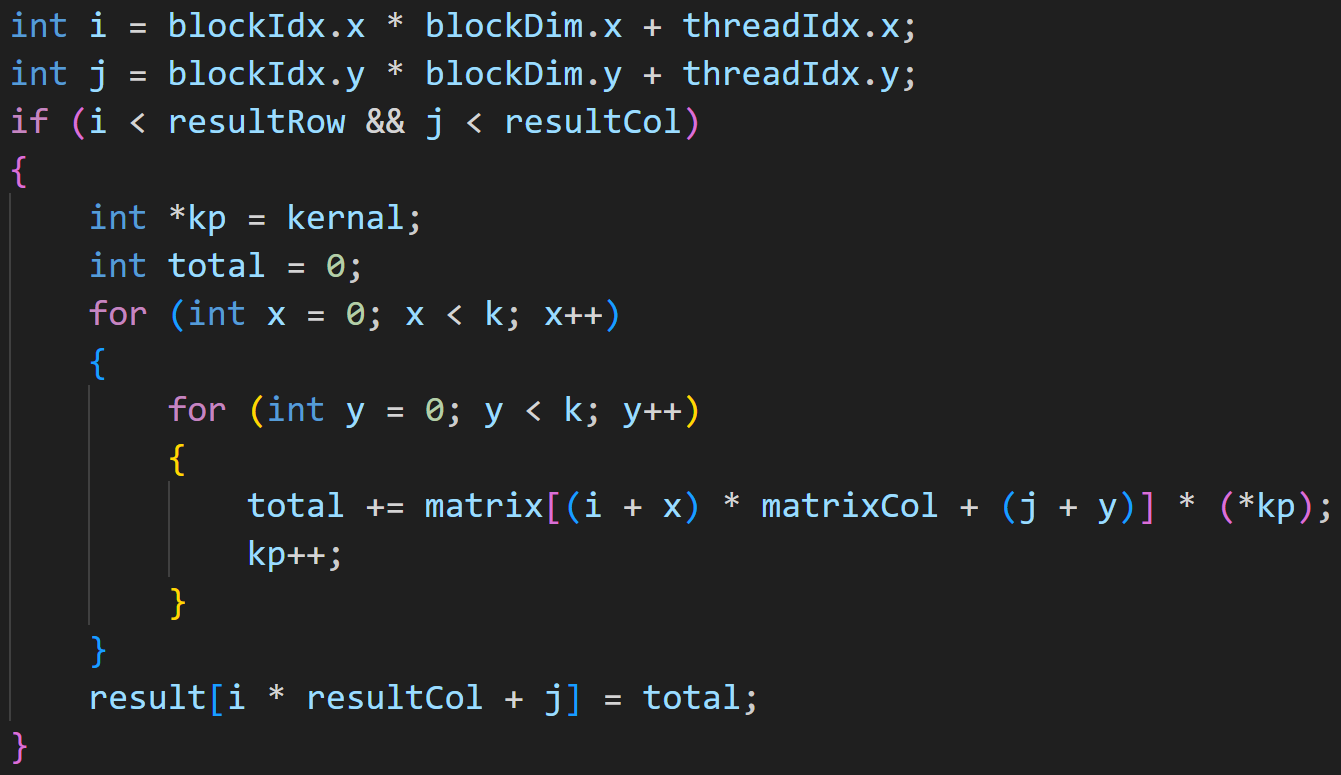
让用户输入矩阵行列大小以及卷积核大小，然后随机生成矩阵和卷积核，这里假设生成的卷积核就是翻转完的。本实验使用valid卷积，即不对边界补零，因此卷积后矩阵长度可计算，如下面的代码所示。



CPU版本的代码如下所示。对结果矩阵位置使用2重循环遍历，在对卷积核进行2重循环遍历。将卷积核元素与原矩阵对应元素相乘，然后加入到结果的对应位置。这里对CPU代码的遍历部分简单使用了指针优化，减少了计算下标的开销。

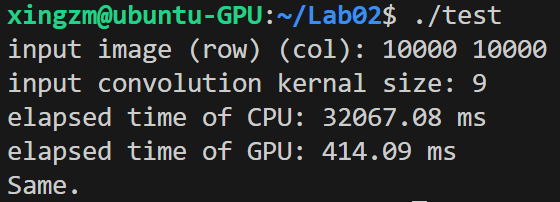


GPU版本部分的代码与实验中前2个问题几乎一致，内核代码使用了二维的block，其余部分与CPU版本类似。



## 实验结果

对大矩阵(10000\*10000)小卷积核(9\*9)进行实验，可以看到执行后的结果相同，GPU版本比CPU版本提升了约77倍。



对小矩阵(1000\*2000)大卷积核(998\*998)进行实验，GPU版本比CPU版本提升了约44倍。

