# 图像卷积

## 实验内容

本实验要求使用卷积操作对图像进行处理，并用并行化编程的思想对卷积算法进行优化。

卷积一词最开始出现在信号与线性系统中，信号与线性系统中讨论的就是信号经过一个线性系统以后发生的变化。数字图像是一个二维的离散信号，对数字图像做卷积操作其实就是利用卷积核（卷积模板）在图像上滑动，将图像点上的像素灰度值与对应的卷积核上的数值相乘，然后将所有相乘后的值相加作为卷积核中间像素对应的图像上像素的灰度值，并最终滑动完所有图像的过程。

实验要求：

1. 输入图像得到该图像的卷积后的结果
2. 在卷积过程中可以自定义参数对卷积参数进行调整

## 2. 实验目的

1）通过使用Pthread熟悉并行化编程

2）通过本实验熟悉图像卷积的过程以及背景知识

## 3. 实验环境

本实验在macOS系统下使用python编程语言进行，具体参数如下图：



## 实验步骤

在本实验当中，我们构建了两种卷积的方式。第一种是正常的卷积（算法1），我们通过卷积操作的定义使用python写出了图像卷积操作，同时我们进行了数据并行，建立了多个线程来对数据集进行分配并进行卷积操作。第二种是在第一种卷积的基础上加入了并行的思想（算法2）。pthread的核型思想在于线程上，我们不同channel处创建线程以达到加速的作用，最后实验结果发现我们的方法与第一种方法的加速比能够达到108%。

### 4.1搭建Pthread运行环境

1）安装threading库

2）导入threading库

3）导入其他所需要的库



### 4.2 创建源代码

4.2.1创建图像输入操作

import os

from mpi4py import MPI

import numpy as np

import cv2

IMAGE\_H = 200

IMAGE\_W = 300

def div\_list(ls, n):

if not isinstance(ls, list) or not isinstance(n, int):

return []

ls\_len = len(ls)

if n <= 0 or 0 == ls\_len:

return []

if n > ls\_len:

return []

elif n == ls\_len:

return [[i] for i in ls]

else:

j = ls\_len//n

k = ls\_len % n

# j,j,j,...(前面有n-1个j),j+k

# 步长j,次数n-1

ls\_return = []

for i in range(0, (n-1)\*j, j):

ls\_return.append(ls[i:i+j])

# 算上末尾的j+k

ls\_return.append(ls[(n-1)\*j:])

return ls\_return

def load\_Img(filedir):

file\_ = []

for fname in os.listdir(filedir):

file\_path = filedir + '/' + fname

file\_.append(file\_path)

return file\_

def load\_img\_batch(img\_name\_list):

num = len(img\_name\_list)

img\_list = np.zeros((num, 3, IMAGE\_H, IMAGE\_W))

for i in range(num):

img = cv2.imread(img\_name\_list[i])

if img is None:

continue

img\_ = np.resize(img, (IMAGE\_H, IMAGE\_W, 3))/255

img\_ = img\_.transpose((2, 0, 1))

img\_list[i] = (img\_)

return img\_list

4.2.2 创建图像卷积操作

import numpy as np

class Conv2d:

def \_\_init\_\_(self, in\_plains, out\_plains, kernel\_size=(3, 3), stride=1, padding=0):

kernel\_param\_num = kernel\_size[0] \* kernel\_size[1]

# self.kernel = np.ones(kernel\_size) / kernel\_param\_num

self.kernel = np.array([

[-1, -1, -1],

[-1, 9, -1],

[-1, -1, -1]

])

self.kernel\_size = kernel\_size

self.in\_plains = in\_plains

self.out\_plains = out\_plains

self.stride = stride

self.padding = padding

self.H = 0

self.W = 0

self.C = out\_plains

def convolution(self, image\_batch):

n, c, h, w = image\_batch.shape

self.H = (h - self.kernel\_size[0] + 2\*self.padding)//self.stride + 1

self.W = (w - self.kernel\_size[1] + 2\*self.padding)//self.stride + 1

tensor = np.zeros([n, self.C, self.H, self.W])

for num in range(n):

img = image\_batch[num]

h\_ = h+2\*self.padding

w\_ = w+2\*self.padding

img\_padded = np.zeros([c, h\_, w\_])

img\_padded[:, self.padding:h\_-self.padding,

self.padding:w\_-self.padding] = img

img\_new = np.zeros([self.C, self.H, self.W])

for channel in range(c):

for i in range(self.H):

for j in range(self.W):

i\_ = i\*self.stride

j\_ = j\*self.stride

a = img\_padded[channel, i\_:i\_ +

self.kernel\_size[0], j\_:j\_+self.kernel\_size[1]]

img\_new[channel, i, j] = np.sum(a\*self.kernel)

tensor[num] = img\_new

return tensor

4.2.3 创建并行图像卷积操作

在本小节当中，我们使用了并行操作对图像卷积进行了优化。

import numpy as np

import threading

from time import time

class MyThread(threading.Thread):

def \_\_init\_\_(self, func, args=()):

super(MyThread, self).\_\_init\_\_()

self.func = func

self.args = args

def run(self):

# time.sleep(2)

self.result = self.func(\*self.args)

def get\_result(self):

threading.Thread.join(self) # 等待线程执行完毕

try:

return self.result

except Exception:

return None

class Conv2d\_plus:

def \_\_init\_\_(self, in\_plains, out\_plains, kernel\_size=(3, 3), stride=1, padding=0):

kernel\_param\_num = kernel\_size[0] \* kernel\_size[1]

# self.kernel = np.ones(kernel\_size) / kernel\_param\_num

self.kernel = np.array([

[-1, -1, -1],

[-1, 9, -1],

[-1, -1, -1]

])

self.kernel\_size = kernel\_size

self.in\_plains = in\_plains

self.out\_plains = out\_plains

self.stride = stride

self.padding = padding

self.H = 0

self.W = 0

self.C = out\_plains

def sub\_conv(self,img\_padded,img\_new,channel):

for i in range(self.H):

for j in range(self.W):

i\_ = i\*self.stride

j\_ = j\*self.stride

a = img\_padded[channel, i\_:i\_ +

self.kernel\_size[0], j\_:j\_+self.kernel\_size[1]]

img\_new[channel, i, j] = np.sum(a\*self.kernel)

def convolution(self, image\_batch):

n, c, h, w = image\_batch.shape

self.H = (h - self.kernel\_size[0] + 2\*self.padding)//self.stride + 1

self.W = (w - self.kernel\_size[1] + 2\*self.padding)//self.stride + 1

tensor = np.zeros([n, self.C, self.H, self.W])

for num in range(n):

img = image\_batch[num]

h\_ = h+2\*self.padding

w\_ = w+2\*self.padding

img\_padded = np.zeros([c, h\_, w\_])

img\_padded[:, self.padding:h\_-self.padding,

self.padding:w\_-self.padding] = img

global img\_new

img\_new = np.zeros([self.C, self.H, self.W])

threads=[]

for channel in range(c):

t=MyThread(func=self.sub\_conv,args=(img\_padded,img\_new,channel))

threads.append(t)

for thread in threads:

thread.start()

for thread in threads:

thread.join()

tensor[num] = img\_new

return tensor

4.2.4 创建main函数

from get\_img import \*

import numpy as np

from conv import Conv2d

from mysum import \*

import time

from time import ctime

import threading

import cv2

import matplotlib

import tqdm

from conv\_withPARALLEL import Conv2d\_plus

IMAGE\_H = 200

IMAGE\_W = 300

class MyThread(threading.Thread):

def \_\_init\_\_(self, func, args=()):

super(MyThread, self).\_\_init\_\_()

self.func = func

self.args = args

def run(self):

# time.sleep(2)

self.result = self.func(\*self.args)

def get\_result(self):

threading.Thread.join(self) # 等待线程执行完毕

try:

return self.result

except Exception:

return None

if \_\_name\_\_ == '\_\_main\_\_':

file\_ = load\_Img("/Users/puyuandong613/Downloads/Parallel-convolution/Parallel-convolution-Thread/images")

start\_imread1=time.time()

#非并行化读文件

img\_batches1= np.zeros((len(file\_), 3, IMAGE\_H, IMAGE\_W))

for i in range(len(file\_)):

img = cv2.imread(file\_[i])

if img is None:

continue

img\_ = np.resize(img, (IMAGE\_H, IMAGE\_W, 3))/255

img\_ = img\_.transpose((2, 0, 1))

img\_batches1[i]=(img\_)

end\_imread1=time.time()

print("without parallel: imread time",end\_imread1-start\_imread1)

# # 非并行卷积

conv2d=Conv2d(in\_plains=3,out\_plains=3)

start\_imgConv1=time.time()

rec\_imgs1=[]

# for index in range(len(img\_batches1)):

rec\_img=conv2d.convolution(img\_batches1)

rec\_imgs1.append(rec\_img)

end\_imgConv1=time.time()

print("without parallel: conv time",end\_imgConv1-start\_imgConv1)

#并行化读文件

img\_save\_dir="/Users/puyuandong613/Downloads/Parallel-convolution/Parallel-convolution-Thread/img\_time"

cov\_save\_dir="/Users/puyuandong613/Downloads/Parallel-convolution/Parallel-convolution-Thread/cov\_time"

img\_time=[]

cov\_time=[]

process\_nums=[1,2,4,8,16,32,64,128]

conv2d\_plus=Conv2d\_plus(in\_plains=3,out\_plains=3)

for process\_num in tqdm.tqdm(process\_nums):

threads=[]

img\_batches2=[]

print("process\_num:", process\_num)

file\_list = div\_list(file\_, process\_num)

start\_imread2=time.time()

for index in range(len(file\_list)):

t = MyThread(

func=load\_img\_batch,args=(file\_list[index],)

)

# print(file\_list[index])

threads.append(t)

for thread in threads:

thread.start()

for thread in threads:

thread.join()

for thread in threads:

img\_batches2.append(thread.result)

end\_imread2=time.time()

print("with parallel: imread time",end\_imread2-start\_imread2)

img\_time.append(end\_imread2-start\_imread2)

#并行卷积

start\_imgConv2=time.time()

rec\_imgs2=[]

threads2=[]

for i in range(len(img\_batches2)):

# t=MyThread(func=conv2d.convolution,args=(img\_batches2[i],))

t=MyThread(func=conv2d\_plus.convolution,args=(img\_batches2[i],))

threads2.append(t)

for thread in threads2:

thread.start()

for thread in threads2:

thread.join()

for thread in threads2:

rec\_imgs2.append(thread.result)

end\_imgConv2=time.time()

print("with parallel: conv time", end\_imgConv2-start\_imgConv2)

cov\_time.append(end\_imgConv2-start\_imgConv2)

with open(img\_save\_dir,"a") as img\_f:

img\_f.write(str(img\_time))

img\_f.write("\n")

img\_f.close()

with open(cov\_save\_dir,'a') as cov\_f:

cov\_f.write(str(cov\_time))

cov\_f.write("\n")

cov\_f.close()

### 4.3 进行编译

cd Parallel-convolution-Thread

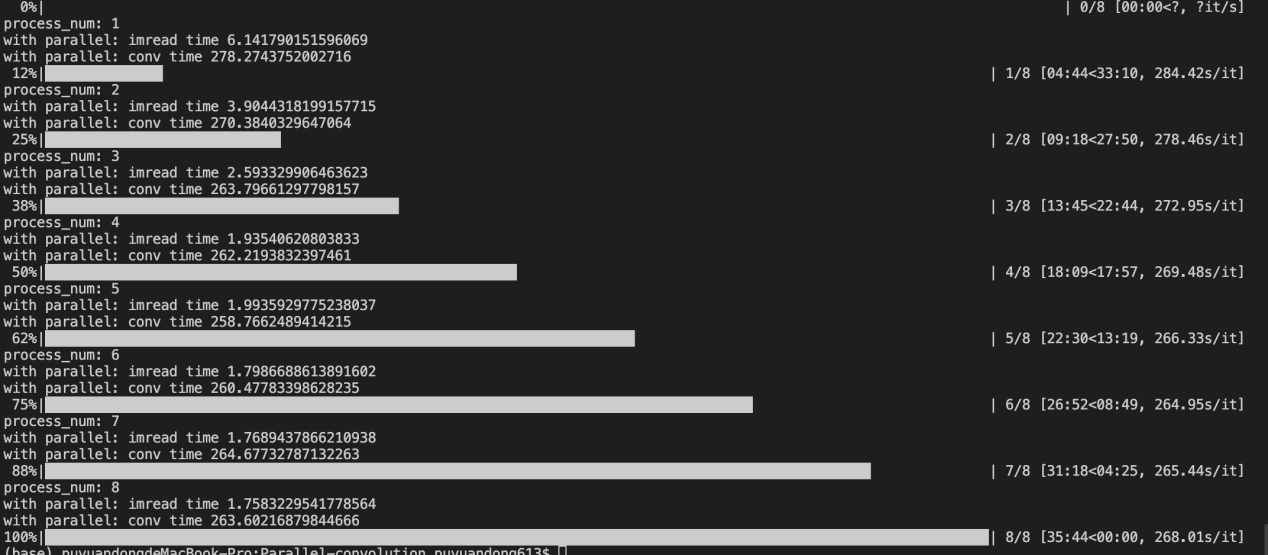
python main.py

### 4.4 运行程序

运行4.2.3代码结果如下

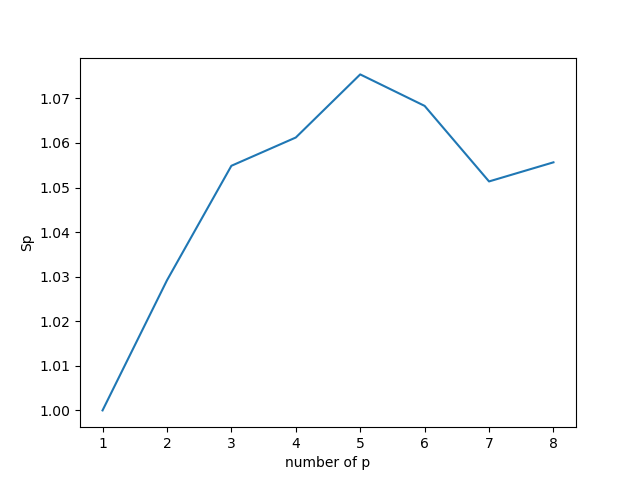
## 截屏2021-09-21 上午9.06.47

运行4.2.2代码结果如下

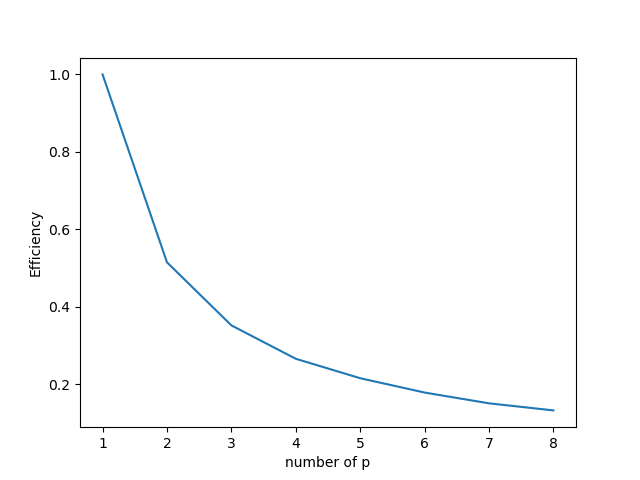


## 实验分析

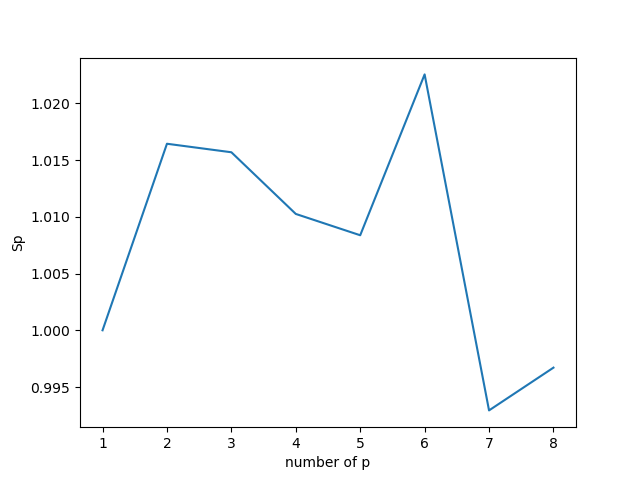
算法优化前的加速比：Figure1



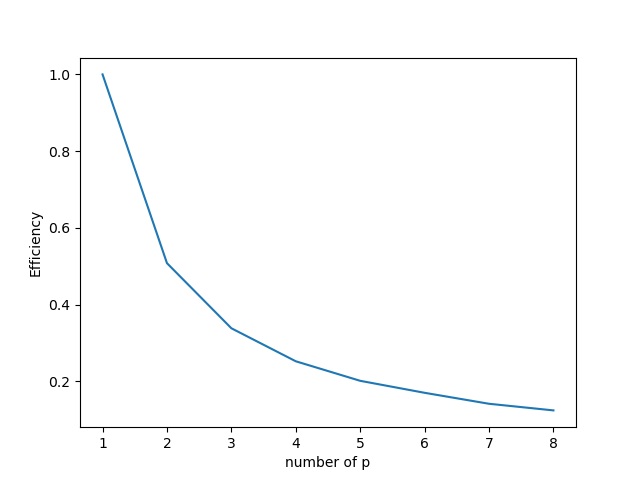
算法优化前的并行效率：Figure2



算法优化后的加速比：Figure3



算法优化后的并行效率：Figure4



**分析：**

从Figure1和Figure3中可以看出，算法1和算法2的加速比是不稳定的，随着线程的增加，算法的加速比开始震荡。同时，从Figure2和Figure4中可以看出，算法1和算法2的并行效率是逐步降低的。这说明我们的算法还需要进一步的改进。综上，我们认为这种情况来自于线程。随着线程的增加，我们认为创建并执行如此数量的线程也会花费时间，所以会导致加速比震荡和并行效率降低这种情况的发生。为了改善这种情况，可以从卷积算法本身进行改善，比如在卷积过程当中使用快速傅立叶变换（FFT）能够极大程度的增加卷积操作的效率。

## 6. 问题与讨论

如果有可以引发思考的问题，或者是实验中可能遇到的麻烦等，在这里列出来。

1）进行一次实验的时间可能过长，对代码的修改效率较低

2）一开始使用python中的threading库不是很熟练，导致了一些小错误。比如说threading库是不能直接调用线程函数的返回值的，所以不能直接得到卷积的结果。为了解决这个问题，我们定义了一个Mythread类，对threading库中的Thread对象进行继承从而解决问题。

## 参考资料：

所涉及的一些参考网站链接等