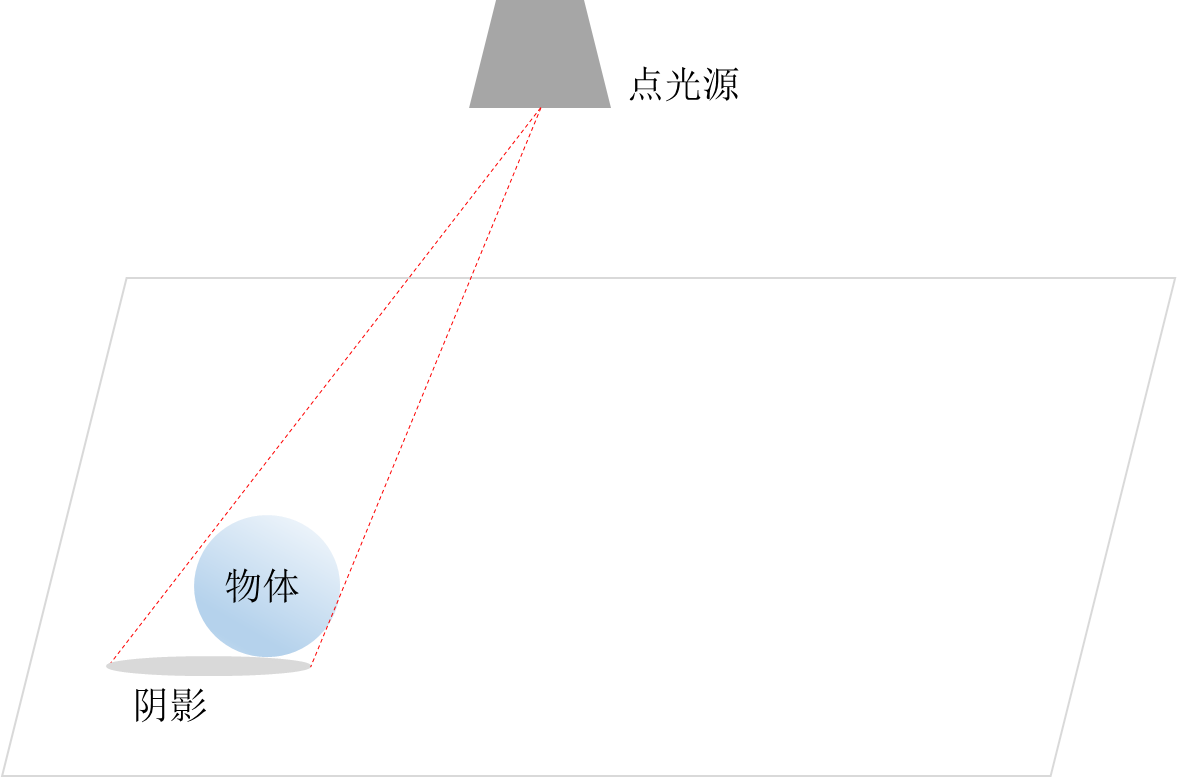
Python写Shadow算法

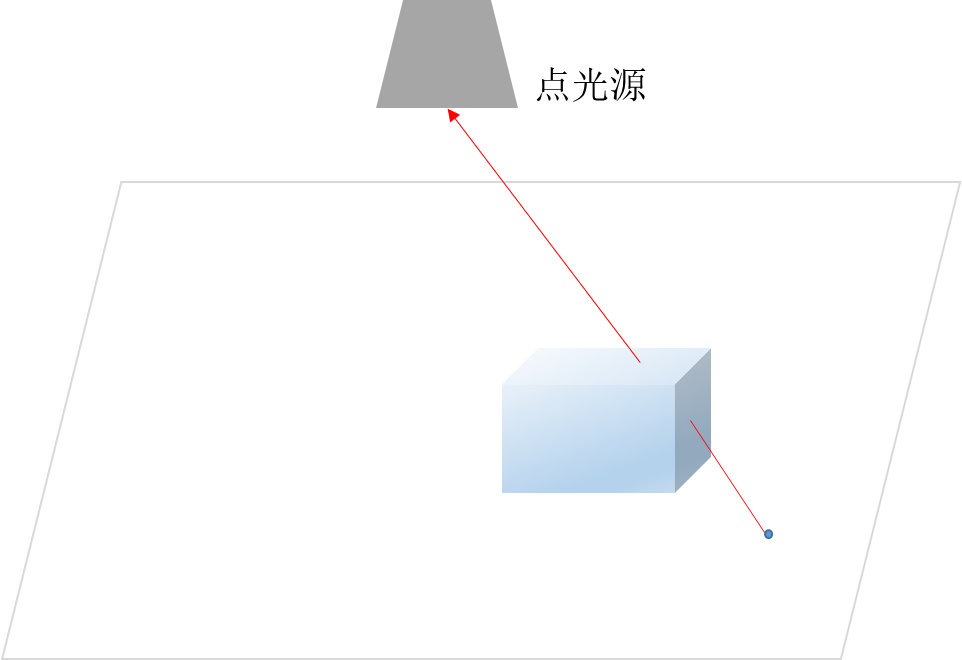
前一段时间由于项目需要，要写一个小算法，就是根据城市地形（包括建筑物高度）结合某一天太阳的真实位置计算城市中各个地方是否被阴影遮挡，即Shadow。

在计算机视觉领域，已经有很多成熟的方法计算阴影，但是这些计算大多是模拟视角机制进行渲染，或者在某一场景下给定光源位置模拟光照形成的阴影。其计算核心是检测遮挡，即从某一条光线出发，追踪光线在路径上是否被其它物体遮挡,即光线追踪算法，阴影可以看为光源看不到的地方，从光源往各个方向发送光线，则光线无法照射到的区域则为阴影。下图显示了在点光源下光线追踪算法的原理。

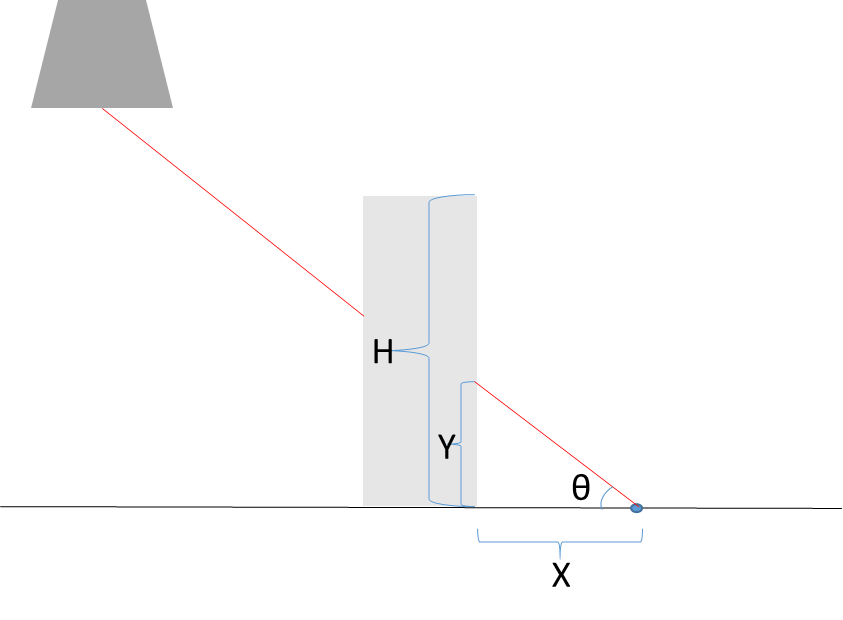


上面这张图是模拟点光源的场景，模拟视角机制其实也是一回事，因为我们人眼之所以能看到物体表面，是因为物体表面反射或折射的光线进入人眼，即光线只不过是从物体表面出发到达人眼而已，原理还是上面那张图，把点光源换成人眼即可。

我们对某一点进行检测是否落在阴影里时具体做法是，先从这点向光源引一条光线，光线方向指向光源，若该光线在到达光源之前碰撞到某些物体，则该点位于阴影中，而碰撞则为相交判断。



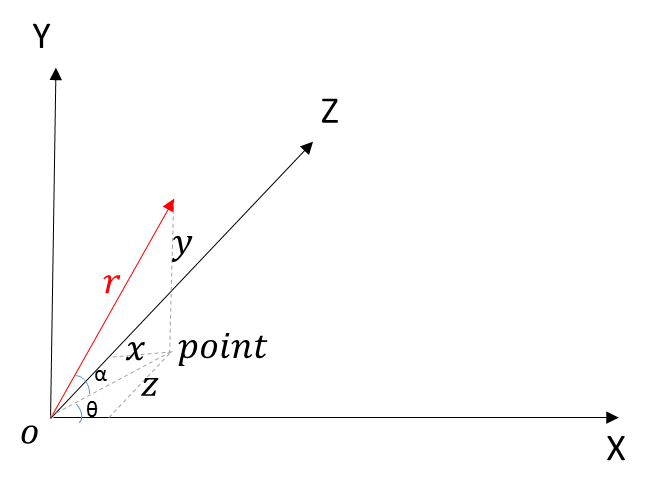
相交判断在二维直角坐标系下面想必大部分人都会做。那我们就先把上面这个三维世界的相交判断简化成二维世界的相交问题，简单的来说就是下面这个问题：



故可判断从该点沿光源方向出发的光线被物体遮挡，即可知该点处于阴影范围内。说到这里光线追踪算法其实已经很简单了，相信有一点编程基础的同学都能够实现了。那么接下来我将抛出我的问题，就是我们在现实3D地理空间里，尤其是在大场景里模拟太阳光照阴影的话，与上面有何不同？首先我们要考虑的是地球为球体，所以我们在计算某一区域的太阳光照阴影时，地球表面不能当作平面对待，而应该是曲面，同时太阳光线照射到地球表面基本上可以看成是平行光线，不同于上面所说的点光源，下面我将具体介绍基于DSM数据计算太阳光照阴影的问题。

DSM数据，即数字地表模型，包含了地表建筑物、桥梁和树木等要素的地面高程模型。在地理空间中，对于每一个点，我们同样从这一点出发，沿着光源方向进行检测遮挡。我们的计算是基于球面坐标进行迭代计算的。然后将球面坐标转换成经纬度和光线高度，将光线高度与地表高度进行比较，判断该点是否落在阴影内。

首先我们在计算光线高度的时候，要确定逐增半径来计算光线高度，逐增半径可根据经纬度分辨率计算。

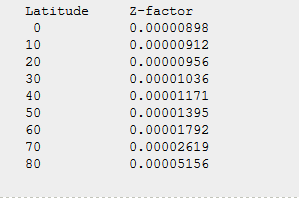


初始状态，注意这里计算得到的是根据经纬度单位计算得到的，需要将其单位转换为m,这就涉及到经纬度坐标下直线距离的计算，事实上在我们地理空间里，经纬度坐标下是没有直线距离的，我举个很简单的例子吧，平时我们进行坐标转换都是进行线性转换的，比如当我们把米转换成厘米时，放大100倍即可，而且对于1m,2m,3m都只需要乘以100都可以转换成厘米。但是在地理空间中，由于不同地区的相同的经纬度所代表的距离实际上是不一样的，即非等距投影，可能在A地，1度代表\*\*米，但在B地1度代表##米，所以我们不能直接转换，实际情况是我们针对不同地方的两个点都只能通过比例推算，而且这种推算都是不完全正确的，具体推算方法是

这地方的h就是我们要推算的对应转化成米为单位的高度

这个meter是根据勾股定理计算而来，即为带判断点的纬度，相信大家已经发现问题了，你刚才不是说经纬度不能直接转换为米吗，怎么在这里将经度和纬度分别乘以一个经纬度以米为单位的分辨率就转换成了米呢，是的这么做是不正确的，但是这种做法针对小范围估算还是比较适用的。

事实上就是做不到转换，但是有时候我们还需要这个线性转换系数将度转换到米，世界上最大的GIS公司Esri,在他们Arcgis产品中是这么做的，他们在每个纬度段都估算（注意也是估算，后面我做了对比，对于小范围这种估算和上面我说的估算结果都是差不多的）了一个转换系数，以此来将度转换成米，这个转换系数就是我们在工具里经常看到的Z factor。



/Z factor即将我们计算得到的转换成以米为单位的高度，再据此来计算光线高度rayelevation

/Z factor，这里的为我们待判断点的高程值，即上图中点高程值。

将光线高度与该位置上的地表高度pointelevation做比较。



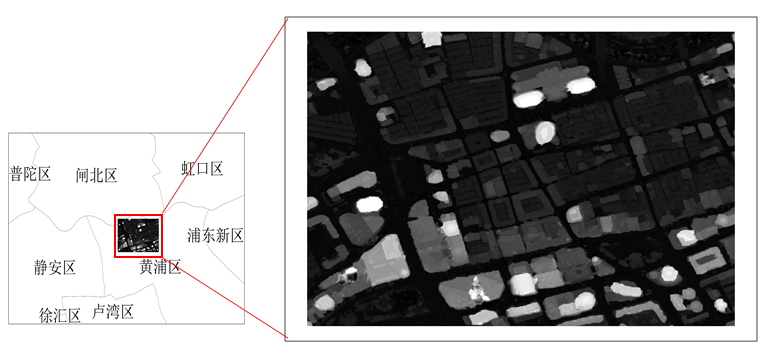
若rayelevation<pointelevation，则出现相交，待判断点即落在阴影区域，否则则增加进行下一个循环计算：

1. 若光线超出计算范围，无法获取光线位置的地表高度，则停止循环
2. 若光线所对应位置无地表高度值，则增加进行下一个循环
3. 若光线高度大于所计算区域内地表最高高程，则待判断点不落在阴影里
4. 若光线高度小于地表高程值，则待判断点落在阴影范围内。

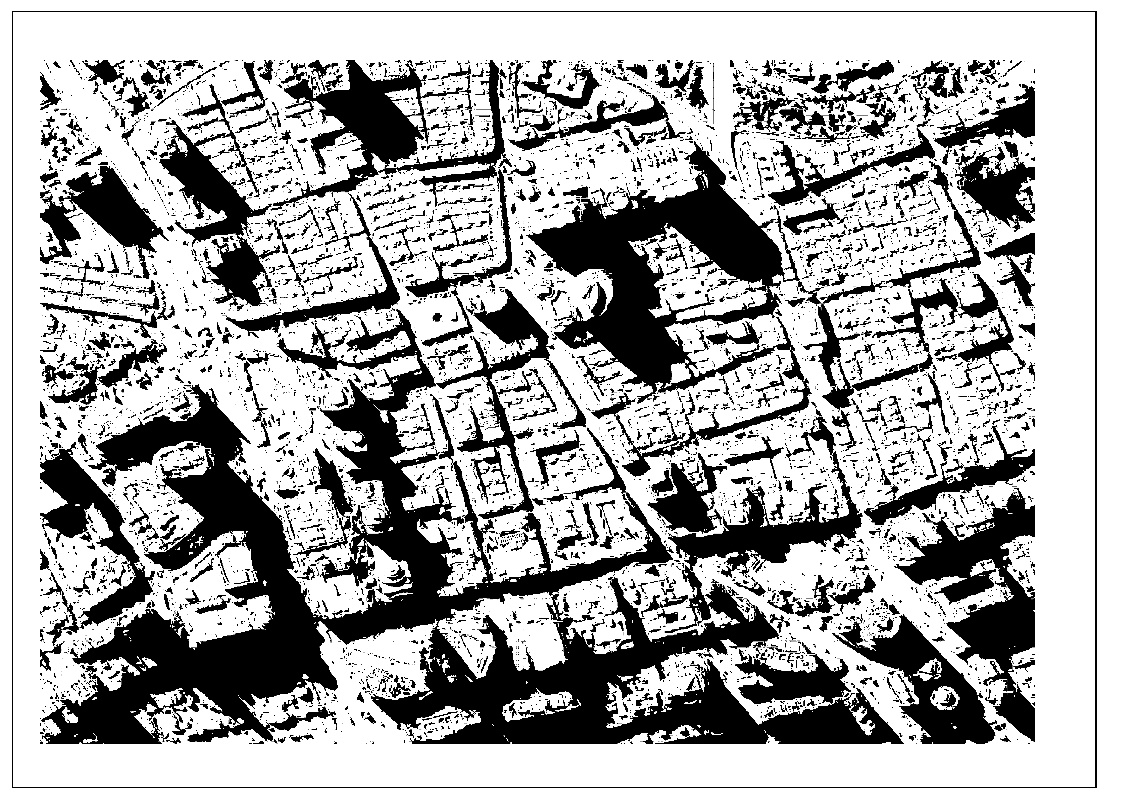
按上述方法对DSM数据上的每个点做遍历，即可计算出来该区域的太阳光照阴影，即shadow。这是一个非常大的计算，因为要遍历整个DSM数据，而且DSM上每一个点要做那么多次循环来check光线是否被遮挡，耗时长。

实例介绍：

研究区域选取上海市黄浦区某一区域的DSM数据，数据分辨率为0.5米2001\*1601个像素，纬度位于31.2353982575898~31.2426231088964，经度位于31.2353982575898~31.2353982575898，原始数据是上海地方坐标，将其投影成经纬度坐标后，数据大小为（2159, 1485）行列，经纬度分辨率为4.8652197e-006，如下所示：



选择python语言实现上述算法并对数据进行shadow计算，耗时：6947.950 seconds，计算结果如下所示：



# -\*- coding: utf-8 -\*-

import gdal

import numpy

from numpy import \*

import math

import time

start = time.clock()

dsm\_file = r"F:\output/test.tif"#stringIndex = int(dsm\_file.rfind("/"))

workSpacePath = dsm\_file[0:int(stringIndex)]

inputRaster = dsm\_file[stringIndex + 1:]

fn=workSpacePath+ '\\'+ inputRaster

ds = gdal.Open(fn)

a=gdal.Dataset.ReadAsArray(ds)

geoTransform=ds.GetGeoTransform()

if ds is None:

print 'open failed'

cols = ds.RasterXSize

rows = ds.RasterYSize

x\_cellsize=math.fabs(geoTransform[5])

y\_cellsize=geoTransform[1]

slope=numpy.zeros\_like(a)

hillshadeaspect=numpy.zeros\_like(a)

hillshade=numpy.zeros\_like(a)

shadow=numpy.zeros\_like(a)

shadow[:,:]=255

latitudeResolution =geoTransform[5]#纬度分辨率，注意纬度分辨率是负值

longitudeResolution = geoTransform[1]#经度分辨率

LatMax = geoTransform[3]#最大纬度

LatMin = LatMax+rows\*latitudeResolution#最小纬度

LonMin = geoTransform[0]#最小经度

LonMax = LonMin+cols\*longitudeResolution#最大经度

radiusInterval=math.sqrt(math.pow(latitudeResolution,2)+math.pow(longitudeResolution,2))#逐增半径

maxElevationValue=a.max()#研究范围地表最大高程值

Altitude = 45#太阳高度角α

Zen\_deg = 90 - Altitude

Zen\_rad = Zen\_deg \* math.pi / 180#高度角转弧度

Azimuth = 50#太阳方位角θ

Azimuth\_math = 360 - Azimuth + 90

if Azimuth\_math > 360:

Azimuth\_math = Azimuth\_math - 360

Azimuth\_rad = Azimuth\_math \* math.pi / 180#方位角转弧度

Zfactor=0.00001036#参考Arcgis

RAD\_90 = math.radians(90.0)

RAD\_180 = math.radians(180.0)

RAD\_270 = math.radians(270.0)

def SpheresgetPoint3D(Azi,Alt,radiu):#迭代半径与太阳高度角方位角计算光线对应球面坐标xyz

theta=math.radians(Azi)

phi = math.radians(Alt)

\_y=math.sqrt(math.pow(radiu, 2) - math.pow(radiu\* math.cos(phi), 2))

r0=math.sqrt(math.pow(radiu, 2) - math.pow(\_y, 2))

\_b = r0 \* math.cos(theta)

\_z = math.sqrt(math.pow(r0, 2) - math.pow(\_b, 2))

\_x = math.sqrt(math.pow(r0, 2) - math.pow(\_z, 2))

if theta<= RAD\_90:

\_z \*= -1.0

elif theta <= RAD\_180:

\_x \*= -1.0

\_z \*= -1.0

elif theta <= RAD\_270:

\_x \*= -1.0

if (phi >= 0) :

\_y = math.fabs(\_y)

else :

\_y = math.fabs(\_y) \* -1

points=[]

points.append(\_x)

points.append(\_y)

points.append(\_z)

return points

for i in xrange(1, rows - 1):

for j in xrange(1, cols - 1):

centerlongitude = geoTransform[0] + j \* geoTransform[1] + i \* geoTransform[2]#待判断点经度

centerlatitude = geoTransform[3] + j \* geoTransform[4] + i \* geoTransform[5]#待判断点纬度

centerelevation = a[i, j]#待判断点高度

# 初始化迭代半径

r = radiusInterval

while True:

points = SpheresgetPoint3D(Azimuth, Altitude, r)

pointlatitude = centerlatitude + points[0]#光线点纬度

pointlongitude = centerlongitude - points[2]#光线点经度

# 进行遮挡检测

if pointlatitude > LatMax or pointlatitude < LatMin or pointlongitude > LonMax or pointlongitude < LonMin:

shadow[i, j] = 255

break

try:

pointelevation = a[(LatMax - pointlatitude) / (-latitudeResolution), (pointlongitude - LonMin) / longitudeResolution]

except Exception, e:

print e.message

r += radiusInterval

continue

rayelevation = centerelevation + points[1] / 0.00001036 if rayelevation > maxElevationValue:

shadow[i, j] = 255

break

if rayelevation < pointelevation:

shadow[i, j] = 0

break

r += radiusInterval

#下面是计算hillshade部分，没有需要的可以不看下面这部分

if shadow[i,j]==0:

hillshade[i,j]=shadow[i,j]

else:

xrate=((a[i-1,j+1]+a[i+1,j+1]+2\*a[i,j+1])-(a[i-1,j-1]+a[i+1,j-1]+2\*a[i,j-1]))/(8\*x\_cellsize)

yrate=((a[i+1,j-1]+a[i+1,j+1]+2\*a[i+1,j])-(a[i-1,j-1]+a[i-1,j+1]+2\*a[i-1,j]))/(8\*y\_cellsize)

rise\_run=sqrt(((xrate\*xrate)+(yrate\*yrate)))

slope[i,j]=math.atan(Zfactor\*rise\_run)

if xrate!=0:

hillshadeaspect[i,j]=math.atan2(yrate, -xrate)

if hillshadeaspect[i,j]<0:

hillshadeaspect[i, j] = 2\*math.pi + hillshadeaspect[i, j]

if xrate==0:

if yrate>0:

hillshadeaspect[i,j]=math.pi/2

elif yrate<0:

hillshadeaspect[i,j] = 2\*math.pi- math.pi/2

elif yrate == 0:

hillshadeaspect[i, j]=hillshadeaspect[i,j]

hillshade[i,j] =255.0 \* ((cos(Zen\_rad) \* cos(slope[i,j])) +(sin(Zen\_rad) \* sin(slope[i,j]) \* cos(Azimuth\_rad - hillshadeaspect[i,j])))

if hillshade[i,j]<0:

hillshade[i, j]=0

hillshade[i,j]=int(hillshade[i,j])

print"!"

print"!!"

end=time.clock()

print ("运行时间%.03f seconds" %(end-start))

#计算结果导出为栅格文件

driver = ds.GetDriver()

filename=r'F:\Lidar\output\hillshadeshadow.tif'

outDataset=driver.Create(filename, ds.RasterXSize, ds.RasterYSize, 1,gdal.GDT\_Float32)

outBand = outDataset.GetRasterBand(1)

outBand.WriteArray(hillshade, 0, 0)

outDataset.SetGeoTransform(geoTransform)

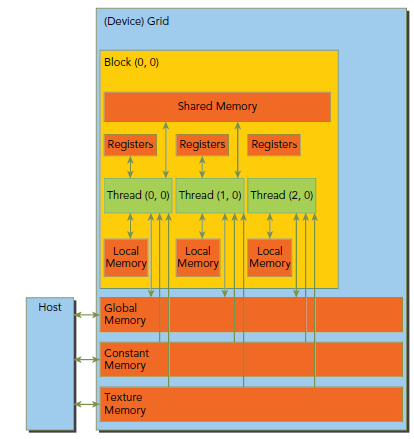
proj = ds.GetProjection()

outDataset.SetProjection(proj)

GPU运算加速是利用GPU的多线程特点，可以大幅提高程序运算速度。要想编程实现GPU加速，我们就得了解一下CUDA，Compute Unified Device Architecture的简称，是由NVIDIA公司创立的基于他们公司生产的图形处理器GPUs（Graphics Processing Units,可以通俗的理解为显卡）的一个并行计算平台和编程模型。通过CUDA，GPUs可以很方便地被用来进行通用计算（有点像在CPU中进行的数值计算等等）。在没有CUDA之前，GPUs一般只用来进行图形渲染（如通过OpenGL，DirectX）。开发人员可以通过调用CUDA的API，来进行并行编程，达到高性能计算目的。

这里我不对硬件做过多介绍，想玩CUDA的可以去<https://developer.nvidia.com/cuda-gpus>官网查看哪些显卡支持CUDA。

CUDA中内存类型分为几类，具体介绍可以参考一篇博客，或者翻阅高性能CUDA应用设计与开发



寄存器是GPU最快的memory，kernel中没有什么特殊声明的自动变量都是放在寄存器中的。寄存器变量是每个线程私有的，一旦thread执行结束，寄存器变量就会失效。

有时候如果register不够用了，那么就会使用local memory来代替这部分寄存器空间。shared memory是threads之间通信的基本方式。同一个block中的thread通过shared memory中的数据来相互合作。获取shared memory的数据前必须先用\_\_syncthreads()同步。Global Memory是空间最大，latency最高，GPU最基础的memory。“global”指明了其生命周期。任意SM（流处理器）都可以在整个程序的生命期中获取其状态。global中的变量既可以是静态也可以是动态声明。可以使用\_\_device\_\_修饰符来限定其属性。global memory的分配就是之前频繁使用的cudaMalloc，释放使用cudaFree。在其生命周期内，global memory可以被所有kernel的所有线程访问。通过多线程访问global memory时必须谨慎，因为blocks之间的线程执行不能被同步。当不同线程blocks中的多个线程同时并发的修改global memory中相同位置的数据时，有一种潜在的危险将产生未定义行为。

流处理器是GPU。GPU线程以网格（GRID）的方式组织，一个GRID里有多个BLOCK，BLOCK里有Thread，至于线程数量和线程块的大小这个需要根据不同计算能力的显卡来确定，同一网格（GRID）内的不同块之间存在不需要通信的粗粒度并行，而一个块内的线程之间又形成了允许通信的细粒度并行。程序员在进行程序设计时，需要封装一个内核函数，启动CUDA 内核时，需要将网格信息从CPU传输到GPU。任务分配单元根据这些信息将块分配到SM上，任务分配单元使用的是轮询策略：轮询查看SM是否还有足够的资源来执行新的块，如果有则给SM分配一个新的块，如果没有则查看下一个SM，同一线程块中的线程需要SM中的共享存储器共享数据，因此它们必须在同一个SM中发射。线程块中的每一个线程被发射到一个SP上。任务分配单元可以为每个SM分配最多8个块。而SM中的线程调度单元又将分配到的块进行细分，将其中的线程组织成更小的结构，称为线程束（warp）,每一个块内线程数应该首先是32的倍数，因为这样的话可以适应每一个warp包含32个线程的要求，每一个warp中串行执行。简单的说cuda表面上设计时并行，但是由于gpu芯片的限制，实际上底层有串行流程，这也就为什么很多时候我们在设计线程块和线程数的时候发现达到一定值后计算性能无法提升了。

好了，接下来讲我们这个算法吧，按并行思想进行设计，我们希望所有的像元都能够并行进行阴影遮挡检测，但是通过前面的算法介绍，大家也知道遮挡检测需要迭代进行全局的检索，因此我在进行程序设计时，就讲整个DSM数据从主机拷贝至设备端存在全局内存里了，这样每个线程都可以访问了。

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

import gdal

import numpy

from numpy import \*

import math

from numbapro import cuda

import CUDA\_hillshade

import time

start = time.clock()

dsm\_file = r"F:\\output/test.tif"

stringIndex = int(dsm\_file.rfind("/"))

workSpacePath = dsm\_file[0:int(stringIndex)]

inputRaster = dsm\_file[stringIndex + 1:]

fn=workSpacePath+ '\\'+ inputRaster

ds = gdal.Open(fn)

a=gdal.Dataset.ReadAsArray(ds)

b=a.reshape(-1)

geoTransform=ds.GetGeoTransform()

geoTransform=numpy.array(geoTransform)

geoT=geoTransform.reshape(-1)

# geoTransform2\_dev = cuda.to\_device(geoT) #地理坐标数据读入GPU，直接放在核函数参数里也可以

if ds is None:

print 'open failed'

cols = ds.RasterXSize

rows = ds.RasterYSize

x\_cellsize=math.fabs(geoTransform[5])

y\_cellsize=geoTransform[1]

slope=numpy.zeros\_like(a,dtype=numpy.float64)

Hillshade=numpy.zeros\_like(a,dtype=numpy.int16)#dtype=numpy.int16

Hillshade=Hillshade.reshape(-1)

xrate=numpy.zeros\_like(a,dtype=numpy.float64)

yrate=numpy.zeros\_like(a,dtype=numpy.float64)

latMax = geoTransform[3]

latMin = latMax+rows\*geoTransform[5]

lonMin = geoTransform[0]

lonMax = lonMin+cols\*geoTransform[1]

radius=math.sqrt(math.pow(geoTransform[5],2)+math.pow(geoTransform[1],2))

MaxElevationValue=a.max()

altitude = 45

Zen\_deg = 90 - altitude

zen\_rad = Zen\_deg \* math.pi / 180

Azimuth = 50

Theta = math.radians(Azimuth)

Phi = math.radians(altitude)

Azimuth\_math = 360 - Azimuth + 90

if Azimuth\_math > 360:

Azimuth\_math = Azimuth\_math - 360

azimuth\_rad = Azimuth\_math \* math.pi /180

Zfactor=0.00001036

rAD\_90 = math.radians(90.0)

rAD\_180 = math.radians(180.0)

rAD\_270 = math.radians(270.0)

for i in xrange(1, rows - 2):

for j in xrange(1, cols - 2):

xrate[i,j]= ((a[i-1, j+1] + a[i+1, j+1] + 2 \* a[i,j +1]) - (a[i-1, j-1] + a[i+1, j-1] + 2 \* a[i, j-1])) / (8\*x\_cellsize)

yrate[i,j]= ((a[i+1, j-1] + a[i+1, j+1] + 2 \* a[i+1,j]) - (a[i-1,j-1] + a[i-1, j+1] + 2 \* a[i-1,j])) / (8\*y\_cellsize)

rise\_run = math.sqrt(((math.pow(xrate[i,j],2)+ math.pow(yrate[i,j],2))))

slope[i, j] = math.atan(Zfactor \* rise\_run)

xrate=xrate.reshape(-1)

yrate=yrate.reshape(-1)

slope=slope.reshape(-1)

end1=time.clock()

#定义GPU核函数

griddim=40#网格纬度

blockdim=1024#线程块维度

stream = cuda.stream()

with stream.auto\_synchronize():

aNumpy2\_dev = cuda.to\_device(b,stream=stream) # 将DSM数据读入GPU

xrate2\_dev = cuda.to\_device(xrate,stream=stream)

yrate2\_dev = cuda.to\_device(yrate,stream=stream)

slope2\_dev = cuda.to\_device(slope,stream=stream) # slope数据读入gpu

Hillshade2\_dev =cuda.to\_device(Hillshade,stream=stream)

CUDA\_hillshade.hillshadeshadow[griddim,blockdim,stream](geoT[0],geoT[1],geoT[2],geoT[3],geoT[4],geoT[5],aNumpy2\_dev,slope2\_dev,cols,rows,radius,Theta,Phi,rAD\_90,rAD\_180,rAD\_270,latMax,latMin,lonMax,lonMin,MaxElevationValue,zen\_rad,azimuth\_rad,xrate2\_dev,yrate2\_dev,Hillshade2\_dev)

hillshade=hillshade.reshape(rows,cols)

hillshade=numpy.array(hillshade)

end2=time.clock()

print("CPU运行时间%.03f seconds"%(end1-start))

print("GPU运行时间%.03f seconds"%(end2-end1))

driver = ds.GetDriver()

filename=r'F:\Lidar\outputShanghai\hillshade1.tif'

outDataset=driver.Create(filename, ds.RasterXSize, ds.RasterYSize,1,gdal.GDT\_Float32)

outBand = outDataset.GetRasterBand(1)

outBand.WriteArray(hillshade, 0, 0)

outDataset.SetGeoTransform(geoTransform)

proj = ds.GetProjection()

outDataset.SetProjection(proj)

整个计算耗时：75.215seconds.性能提升很大

注意虽然运行速度有所提升，但是程序耗时还是很长的，在将数据从设备端拷贝至主机端时，时间较长，windows下有个显卡驱动响应机制，在数据拷贝的时候被检测认为显卡没有进行运算停止驱动响应，导致执行完核函数后数据拷贝失败，这里可以通过修改注册表的方法来关闭这个超时检测。具体操作方法：

打开注册表（运行中输入regedit）,依次打开HKEY\_LOCAL\_MACHINE---SYSTEM- -CurrentControlSet--Control,找到GraphicsDrivers,然后右键点击GraphicsDrivers,新建DWORD（32-位）值，名字为TdrLevel,再新建DWORD（32-位）值，名字为TdrDelay。对于TdrLevel，右键修改，将值改为0（十六进制下），为了防止修改之后仍然不生效，将TdrDelay的值改为你想延迟的时间，比如15或者20，默认是2.