Heat Conduction

Example implementations of two dimensional heat equation with various parallel programming approaches.

Heat (or diffusion) equation is

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

where **u(x, y, t)** is the temperature field that varies in space and time, and α is thermal diffusivity constant. The two dimensional Laplacian can be discretized with finite differences as

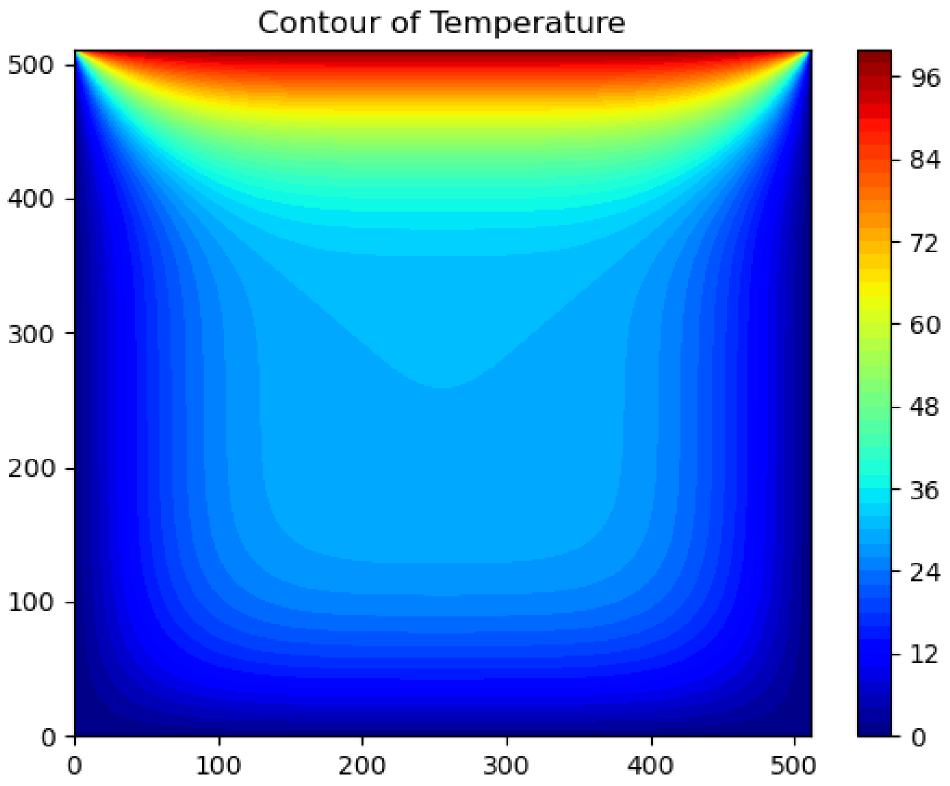
$$\nabla^2 u = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2}$$

Given an initial condition (u(t=0) = u0) one can follow the time dependence of the temperature field with explicit time evolution method:

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t \alpha \nabla^2 u^m(i,j)$$

Note: Algorithm is stable only when

$$\Delta t < \frac{1}{2\alpha} \frac{(\Delta x \Delta y)^2}{(\Delta x)^2 (\Delta y)^2}$$



开发环境

硬件

CPU: Intel Core i5-10490F 6C12T
GPU: RTX 2070 8G
RAM: 32G DDR4 3200MHz

软件

Python: Miniconda python3.8
Mpi4py: 3.1.4
pycuda: 2022.1+cuda116
Nvidia CUDA toolkit: 11.6
Nvidia SMI: 531.41
CUDA: 12.1

代码讲解

参数设置

```python
if __name__ == '__main__':
    opts, _ = getopt.getopt(sys.argv[1:], 'd:t:i:l:', ['device=', 'timesteps=',
'image_interval=', 'len='])
    for opt, arg in opts:
        if opt in ('-d', '--device'):
            device = arg
        elif opt in ('-t', '--timesteps'):
            timesteps = int(arg)
        elif opt in ('-i', '--image_interval'):
            image_interval = int(arg)
        elif opt in ('-l', '--len'):
            lenX = int(arg)
            lenY = int(arg)
            X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))
    main()
```

在调用本python文件时可以加入参数，如`mpiexec -np 4 python main.py -d cpu -t 10000 -i 1000 -l 64`就可以直接设置本次运行的设备、时间步、保存图片的频次和field的形状大小

```python
# Basic parameters
a = 0.5  # Diffusion constant
timesteps = 10000  # Number of time-steps to evolve system
image_interval = 1000  # Write frequency for png files

# Set Dimension and delta
lenX = lenY = 64  # we set it rectangular
delta = 1

# Boundary condition
Ttop = 100
Tbottom = 0
Tleft = 0
Tright = 0
```

```
# Initial guess of interior grid
Tguess = 30

# Grid spacings
dx = 0.01
dy = 0.01
dx2 = dx ** 2
dy2 = dy ** 2

# For stability, this is the largest interval possible
# for the size of the time-step:
dt = dx2 * dy2 / (2 * a * (dx2 + dy2))

# Set colour interpolation and colour map.
# You can try set it to 10, or 100 to see the difference
# You can also try: colourMap = plt.cm.coolwarm
colorinterpolation = 50
colourMap = plt.cm.jet

# Set meshgrid
X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))

# device
device = 'gpu'

# benchmark
isBenchmark = True

# MPI globals
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Up/down neighbouring MPI ranks
up = rank - 1
if up < 0:
    up = MPI.PROC_NULL
down = rank + 1
if down > size - 1:
    down = MPI.PROC_NULL

# get CUDA kernel
evolve_kernel = get_cuda_function()
```

这是一些基本的参数初始化，基本都用注释来讲解了

主函数

```
def main():
    # Read and scatter the initial temperature field
    if rank == 0:
        field, field0 = init_fields()
        shape = field.shape
        dtype = field.dtype
        comm.bcast(shape, 0)  # broadcast dimensions
        comm.bcast(dtype, 0)  # broadcast data type
    else:
        field = None
        shape = comm.bcast(None, 0)
        dtype = comm.bcast(None, 0)
    if shape[0] % size:
        raise ValueError('Number of rows in the temperature field (' \
                         + str(shape[0]) + ') needs to be divisible by the
number ' \
                         + 'of MPI tasks (' + str(size) + ').')

    n = int(shape[0] / size)  # number of rows for each MPI task
    m = shape[1]  # number of columns in the field
    buff = np.zeros((n, m), dtype)
    comm.Scatter(field, buff, 0)  # scatter the data
    local_field = np.zeros((n + 2, m), dtype)  # need two ghost rows!
    local_field[1:-1, :] = buff  # copy data to non-ghost rows
    local_field0 = np.zeros_like(local_field)  # array for previous time step

    # Fix outer boundary ghost layers to account for aperiodicity?
    if True:
        if rank == 0:
            local_field[0, :] = local_field[1, :]
        if rank == size - 1:
            local_field[-1, :] = local_field[-2, :]
    local_field0[:] = local_field[:]

    # Plot/save initial field
    if rank == 0:
        write_field(field, 0)

    # Iterate
    t0 = time.time()
    iterate(field, local_field, local_field0, timesteps, image_interval)
    t1 = time.time()

    # Plot/save final field
    comm.Gather(local_field[1:-1, :], field, root=0)
    if rank == 0:
        write_field(field, timesteps)
        if (isBenchmark):
            import pandas as pd
            import os
            # 若不存在csv文件，则创建，若存在，则追加
            if not os.path.exists('data.csv'):
                df = pd.DataFrame(columns=['MPI_thread', 'device', 'len',
'timesteps', 'time'])
                df.to_csv('data.csv', index=False)
            df = pd.read_csv('data.csv')
            df = df.append(
                {'MPI_thread': size, 'device': device, 'shape': lenX,
'timesteps': timesteps, 'time': t1 - t0},
                ignore_index=True)
            df.to_csv('data.csv', index=False)
        print("Running time: {0}".format(t1 - t0))
```

主函数为初始化field并进行`数据分片`，然后进入`迭代`，得到最后的benchmark时间`保存为csv`

初始化field

```python
# init numpy matrix fields
def init_fields():
    # init
    field = np.empty((lenX, lenY), dtype=np.float64)
    field.fill(Tguess)
    field[(lenY - 1):, :] = Ttop
    field[:1, :] = Tbottom
    field[:, (lenX - 1):] = Tright
    field[:, :1] = Tleft
    # field = np.loadtxt(filename)
    field0 = field.copy()  # Array for field of previous time step
    return field, field0
```

迭代

```python
# iteration
def iterate(field, local_field, local_field0, timesteps, image_interval):
    for m in tqdm(range(1, timesteps + 1)):
        exchange(local_field0)
        comm.Barrier()
        evolve(local_field, local_field0, a, dt, dx2, dy2, device)
        if m % image_interval == 0:
            comm.Gather(local_field[1:-1, :], field, root=0)
            comm.Barrier()
            if rank == 0:
                write_field(field, m)
```

每次迭代时，要先调用exchange来进行MPI线程间的通信，`交换数据`

```python
# MPI thread communication between up and down
def exchange(field):
    # send down, receive from up
    sbuf = field[-2, :]
    rbuf = field[0, :]
    comm.Sendrecv(sbuf, dest=down, recvbuf=rbuf, source=up)
    # send up, receive from down
    sbuf = field[1, :]
    rbuf = field[-1, :]
    comm.Sendrecv(sbuf, dest=up, recvbuf=rbuf, source=down)
```

使用了 `Sendrecv` 方法，即发送和接收数据的组合操作。首先，当前进程将最后一行的数据发送给下一个进程，并接收上一个进程发送的数据到第一行；接着，当前进程将第一行的数据发送给上一个进程，并接收下一个进程发送的数据到最后一行。这样就完成了进程间的数据交换，实现了相邻进程间数据的共享。

其中，`sbuf` 和 `rbuf` 分别表示发送缓存和接收缓存。`dest` 和 `source` 分别表示目标进程和源进程的进程号，这里的 `up` 和 `down` 分别表示当前进程的上一个进程和下一个进程的进程号。在这段代码中，我们可以看到当前进程和上一个进程、下一个进程分别进行了一次数据交换，所以这段代码适用于三个或更多进程之间的数据交换

```
# main calculate function
def evolve(u, u_previous, a, dt, dx2, dy2, device):
    """Explicit time evolution.
       u:            new temperature field
       u_previous:   previous field
       a:            diffusion constant
       dt:           time step
       dx2:          grid spacing squared, i.e. dx^2
       dy2:            -- "" --          , i.e. dy^2"""
    if device == 'cpu':
        u[1:-1, 1:-1] = u_previous[1:-1, 1:-1] + a * dt * (
                (u_previous[2:, 1:-1] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[:-2, 1:-1]) / dx2 +
                (u_previous[1:-1, 2:] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[1:-1, :-2]) / dy2)
        u_previous[:] = u[:]
    elif device == 'gpu':
        block_size = (16, 16, 1)
        grid_size = (int(np.ceil(u.shape[0] / block_size[0])),
                     int(np.ceil(u.shape[1] / block_size[1])),
                     1)

        # Call the evolve_kernel with the prepared grid and block sizes
        # import pandas as pd
        # df = pd.DataFrame(u)
        # df.to_csv('u.csv')
        evolve_kernel(driver.InOut(u), driver.InOut(u_previous), np.float64(a),
np.float64(dt),
                                    np.float64(dx2), np.float64(dy2),
np.int32(u.shape[0]),
                                    np.int32(u.shape[1]), block=block_size,
grid=grid_size)
        u_previous[:] = u[:]
    else:
        raise ValueError('device should be cpu or gpu')
```

分别用numpy的方法和pycuda的核函数来实现evolve

```
__global__ void evolve_kernel(double* u, double* u_previous,
                              double a, double dt, double dx2, double dy2,
                              int nx, int ny) {
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
    if (i >= nx - 1 || j >= ny - 1) return;
    u[i * ny + j] = u_previous[i * ny + j] + a * dt * (
                    (u_previous[(i + 1) * ny + j] - 2 * u_previous[i * ny + j] +
                     u_previous[(i - 1) * ny + j]) / dx2 +
                    (u_previous[i * ny + j + 1] - 2 * u_previous[i * ny + j] +
                     u_previous[i * ny + j - 1]) / dy2);
    u_previous[i * ny + j] = u[i * ny + j];
}
```

这里是cuda核函数的实现

```
# save image
def write_field(field, step):
    plt.gca().clear()
    # Configure the contour
    plt.title("Contour of Temperature")
    plt.contourf(X, Y, field, colorinterpolation, cmap=colourMap)
    if step == 0:
        plt.colorbar()
    plt.savefig(

 'img/heat_{thread}_{device}_{shape}_{timesteps}_{step}.png'.format(thread=size,
device=device, shape=lenX,

timesteps=timesteps, step=step))
```

最后在特定的时间节点要进行图片的保存

遇到的问题和解决

Pycuda和cuda toolkit的版本不兼容

问题

本身cuda toolkit版本为11.8，而pycuda最高支持的cuda版本为11.6

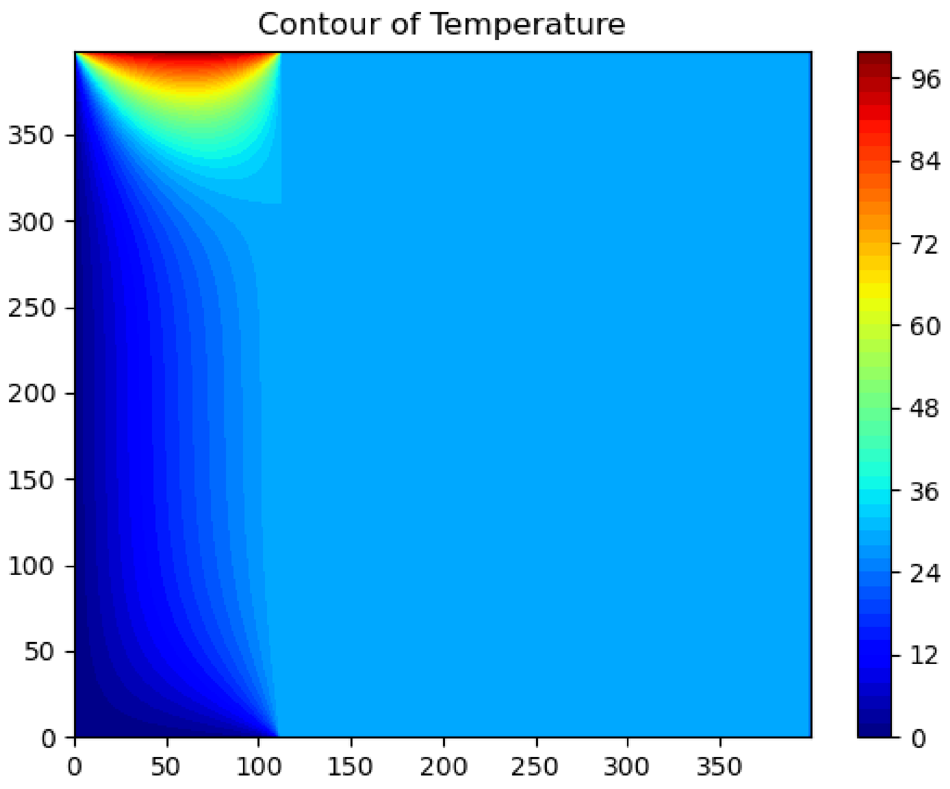
解决

将cuda toolkit降级为11.6，然后根据Installing Pycuda On Windows 10 · Issue #237 · inducer/pycuda (github.com)还要在环境变量中添加新的环境变量 `CUDA_PATH` 为cuda toolkit的根目录(注意不要设置为cuda toolkit的bin目录)

调用cuda方法多线程bug

问题

在调用多线程的mpi分片然后调用cuda核函数来计算时，仅会有第一个线程的区域来计算，如下图



解决

原先的cuda是直接copy别人的代码[heat-equation/core_cuda.cu at main · cschpc/heat-equation · GitHub]

```
/* Update the temperature values using five-point stencil */
__global__ void evolve_kernel(double *currdata, double *prevdata, double a,
double dt, int nx, int ny,
                       double dx2, double dy2)
{

    /* Determine the temperature field at next time step
     * As we have fixed boundary conditions, the outermost gridpoints
     * are not updated. */
    int ind, ip, im, jp, jm;

    // CUDA threads are arranged in column major order; thus j index from x, i
from y
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    if (i > 0 && j > 0 && i < nx+1 && j < ny+1) {
        ind = i * (ny + 2) + j;
        ip = (i + 1) * (ny + 2) + j;
        im = (i - 1) * (ny + 2) + j;
    jp = i * (ny + 2) + j + 1;
    jm = i * (ny + 2) + j - 1;
```

```
        currdata[ind] = prevdata[ind] + a * dt *
          ((prevdata[ip] -2.0 * prevdata[ind] + prevdata[im]) / dx2 +
          (prevdata[jp] - 2.0 * prevdata[ind] + prevdata[jm]) / dy2);

    }

}
```

和这个mpi配合起来还是有些问题，于是直接把本身的cpu的numpy代码

```
u[1:-1, 1:-1] = u_previous[1:-1, 1:-1] + a * dt * (
                (u_previous[2:, 1:-1] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[:-2, 1:-1]) / dx2 +
                (u_previous[1:-1, 2:] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[1:-1, :-2]) / dy2)
        u_previous[:] = u[:]
```

转换成了cuda的形式

```
__global__ void evolve_kernel(double* u, double* u_previous,
                                  double a, double dt, double dx2, double dy2,
                                  int nx, int ny) {
        int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
        int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
        if (i >= nx - 1 || j >= ny - 1) return;
        u[i * ny + j] = u_previous[i * ny + j] + a * dt * (
                    (u_previous[(i + 1) * ny + j] - 2 * u_previous[i * ny +
j] +
                     u_previous[(i - 1) * ny + j]) / dx2 +
                    (u_previous[i * ny + j + 1] - 2 * u_previous[i * ny + j]
+
                     u_previous[i * ny + j - 1]) / dy2);
        u_previous[i * ny + j] = u[i * ny + j];
    }
```

然后python端做好处理后就可以成功跑通了

```
elif device == 'gpu':
    block_size = (16, 16, 1)
    grid_size = (int(np.ceil(u.shape[0] / block_size[0])),
                 int(np.ceil(u.shape[1] / block_size[1])),
                 1)

    # Call the evolve_kernel with the prepared grid and block sizes
    evolve_kernel(driver.InOut(u), driver.InOut(u_previous), np.float64(a),
np.float64(dt),
                                np.float64(dx2), np.float64(dy2),
np.int32(u.shape[0]),
                                np.int32(u.shape[1]), block=block_size,
grid=grid_size)
    u_previous[:] = u[:]
```

不同数据规模的测试

批处理测试

写了一个bat来分别对不同`线程数`，`设备`，`field形状大小`来进行时间的测试，由于`2048×2048`和`4096×4096`的数据量太大，所以`timestamps`变为了其他的`1/10`，最后设计的标准是`it/s`所以还是没有太大问题

```
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 2 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 2 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 2 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 2 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 4 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 4 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 4 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 4 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 8 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 8 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 8 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 8 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 16 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 16 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 16 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 16 python ../main.py -d gpu -t 1000 -i 100 -l 4096
```

测试结果

MPI_thread	device	timesteps	time(s)	shape	it/s
2	cpu	10000	1.48	64	6772.58
2	cpu	10000	3.41	256	2934.66
2	cpu	10000	35.64	512	280.57
2	cpu	10000	177.49	1024	56.34
2	cpu	1000	88.70	2048	11.27
2	cpu	1000	350.17	4096	2.86
2	gpu	10000	9.46	64	1056.79
2	gpu	10000	11.76	256	850.33
2	gpu	10000	21.98	512	455.02
2	gpu	10000	65.98	1024	151.56
2	gpu	1000	37.68	2048	26.54
2	gpu	1000	145.31	4096	6.88
4	cpu	10000	1.41	64	7082.81
4	cpu	10000	2.56	256	3906.46
4	cpu	10000	9.20	512	1087.11
4	cpu	10000	133.29	1024	75.02
4	cpu	1000	69.63	2048	14.36
4	cpu	1000	278.52	4096	3.59
4	gpu	10000	19.31	64	517.87
4	gpu	10000	17.31	256	577.56
4	gpu	10000	24.97	512	400.43
4	gpu	10000	71.42	1024	140.01
4	gpu	1000	37.61	2048	26.59
4	gpu	1000	141.01	4096	7.09
8	cpu	10000	1.48	64	6774.35
8	cpu	10000	2.38	256	4200.50
8	cpu	10000	6.54	512	1529.36
8	cpu	10000	103.56	1024	96.56
8	cpu	1000	59.64	2048	16.77
8	cpu	1000	230.30	4096	4.34
8	gpu	10000	27.77	64	360.12
8	gpu	10000	29.98	256	333.54
8	gpu	10000	38.87	512	257.26
8	gpu	10000	76.59	1024	130.56
8	gpu	1000	38.86	2048	25.74
8	gpu	1000	141.22	4096	7.08
16	cpu	10000	141.58	64	70.63
16	cpu	10000	144.80	256	69.06
16	cpu	10000	155.64	512	64.25
16	cpu	10000	172.54	1024	57.96
16	cpu	1000	64.80	2048	15.43
16	cpu	1000	229.28	4096	4.36
16	gpu	10000	184.78	64	54.12
16	gpu	10000	202.33	256	49.42
16	gpu	10000	212.20	512	47.13
16	gpu	10000	229.22	1024	43.63
16	gpu	1000	55.17	2048	18.13

16	gpu	1000	55.17	2048	18.13
16	gpu	1000	158.05	4096	6.33

可以发现随着MPI线程数的增多shape越小的field运算速率越慢，而shape越大的运算速率越快，同样的gpu在shape较小的field中运算效率较低，而shape逐渐增大也与cpu的计算效率拉开差距，可以说明数据量更大的运算，就需要更多的MPI线程和gpu的参与。

若数据量较小的话，MPI线程越多可能会造成线程间通讯的浪费更多，GPU和host的通信也同样会有损失。

Heat Conduction

Example implementations of two dimensional heat equation with various parallel programming approaches.

Example implementations of two dimensional heat equation with various parallel programming approaches.

Heat (or diffusion) equation is

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

where **u(x, y, t)** is the temperature field that varies in space and time, and α is thermal diffusivity constant. The two dimensional Laplacian can be discretized with finite differences as

$$\begin{aligned} \nabla^2 u &= \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} \\ &+ \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2} \end{aligned}$$

Given an initial condition (u(t=0) = u0) one can follow the time dependence of the temperature field with explicit time evolution method:

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t \alpha \nabla^2 u^m(i,j)$$

Note: Algorithm is stable only when

$$\Delta t < \frac{1}{2\alpha} \frac{(\Delta x \Delta y)^2}{(\Delta x)^2 (\Delta y)^2}$$

开发环境

硬件

CPU: Intel Core i5-10490F 6C12T

GPU: RTX 2070 8G

RAM: 32G DDR4 3200MHz

软件

Python: Miniconda python3.8

Mpi4py: 3.1.4

pycuda: 2022.1+cuda116

Nvidia CUDA toolkit: 11.6

Nvidia SMI: 531.41

CUDA: 12.1

代码讲解

参数设置

```python
if __name__ == '__main__':
    opts, _ = getopt.getopt(sys.argv[1:], 'd:t:i:l:', ['device=', 'timesteps=',
'image_interval=', 'len='])
    for opt, arg in opts:
        if opt in ('-d', '--device'):
            device = arg
        elif opt in ('-t', '--timesteps'):
            timesteps = int(arg)
        elif opt in ('-i', '--image_interval'):
            image_interval = int(arg)
        elif opt in ('-l', '--len'):
            lenX = int(arg)
            lenY = int(arg)
            X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))
    main()
```

在调用本python文件时可以加入参数，如`mpiexec -np 4 python main.py -d cpu -t 10000 -i 1000 -l 64`就可以直接设置本次运行的设备、时间步、保存图片的频次和field的形状大小

基本的参数初始化

```python
# Basic parameters
a = 0.5  # Diffusion constant
timesteps = 10000  # Number of time-steps to evolve system
image_interval = 1000  # Write frequency for png files

# Set Dimension and delta
lenX = lenY = 64  # we set it rectangular
delta = 1

# Boundary condition
Ttop = 100
Tbottom = 0
Tleft = 0
Tright = 0

# Initial guess of interior grid
Tguess = 30

# Grid spacings
dx = 0.01
dy = 0.01
dx2 = dx ** 2
dy2 = dy ** 2

# For stability, this is the largest interval possible
# for the size of the time-step:
dt = dx2 * dy2 / (2 * a * (dx2 + dy2))

# Set colour interpolation and colour map.
# You can try set it to 10, or 100 to see the difference
# You can also try: colourMap = plt.cm.coolwarm
```

```
colorinterpolation = 50
colourMap = plt.cm.jet

# Set meshgrid
X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))

# device
device = 'gpu'

# benchmark
isBenchmark = True

# MPI globals
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Up/down neighbouring MPI ranks
up = rank - 1
if up < 0:
    up = MPI.PROC_NULL
down = rank + 1
if down > size - 1:
    down = MPI.PROC_NULL

# get CUDA kernel
evolve_kernel = get_cuda_function()
```

这是一些基本的参数初始化，基本都用注释来讲解了

主函数

```
def main():
    # Read and scatter the initial temperature field
    if rank == 0:
        field, field0 = init_fields()
        shape = field.shape
        dtype = field.dtype
        comm.bcast(shape, 0)  # broadcast dimensions
        comm.bcast(dtype, 0)  # broadcast data type
    else:
        field = None
        shape = comm.bcast(None, 0)
        dtype = comm.bcast(None, 0)
    if shape[0] % size:
        raise ValueError('Number of rows in the temperature field (' \
                         + str(shape[0]) + ') needs to be divisible by the
number ' \
                         + 'of MPI tasks (' + str(size) + ').')

    n = int(shape[0] / size)  # number of rows for each MPI task
    m = shape[1]  # number of columns in the field
    buff = np.zeros((n, m), dtype)
    comm.Scatter(field, buff, 0)  # scatter the data
    local_field = np.zeros((n + 2, m), dtype)  # need two ghost rows!
    local_field[1:-1, :] = buff  # copy data to non-ghost rows
    local_field0 = np.zeros_like(local_field)  # array for previous time step

    # Fix outer boundary ghost layers to account for aperiodicity?
    if True:
        if rank == 0:
            local_field[0, :] = local_field[1, :]
        if rank == size - 1:
            local_field[-1, :] = local_field[-2, :]
    local_field0[:] = local_field[:]

    # Plot/save initial field
    if rank == 0:
        write_field(field, 0)

    # Iterate
    t0 = time.time()
    iterate(field, local_field, local_field0, timesteps, image_interval)
    t1 = time.time()

    # Plot/save final field
    comm.Gather(local_field[1:-1, :], field, root=0)
    if rank == 0:
        write_field(field, timesteps)
        if (isBenchmark):
            import pandas as pd
            import os
            # 若不存在csv文件，则创建，若存在，则追加
            if not os.path.exists('data.csv'):
                df = pd.DataFrame(columns=['MPI_thread', 'device', 'len', 
'timesteps', 'time'])
                df.to_csv('data.csv', index=False)
            df = pd.read_csv('data.csv')
            df = df.append(
                {'MPI_thread': size, 'device': device, 'shape': lenX, 
'timesteps': timesteps, 'time': t1 - t0},
                ignore_index=True)
            df.to_csv('data.csv', index=False)
        print("Running time: {0}".format(t1 - t0))
```

主函数为初始化field并进行`数据分片`，然后进入`迭代`，得到最后的benchmark时间`保存为csv`

初始化field

```
# init numpy matrix fields
def init_fields():
    # init
    field = np.empty((lenX, lenY), dtype=np.float64)
    field.fill(Tguess)
    field[(lenY - 1):, :] = Ttop
    field[:1, :] = Tbottom
    field[:, (lenX - 1):] = Tright
    field[:, :1] = Tleft
    # field = np.loadtxt(filename)
    field0 = field.copy()  # Array for field of previous time step
    return field, field0
```

迭代

```
# iteration
def iterate(field, local_field, local_field0, timesteps, image_interval):
    for m in tqdm(range(1, timesteps + 1)):
        exchange(local_field0)
        comm.Barrier()
        evolve(local_field, local_field0, a, dt, dx2, dy2, device)
        if m % image_interval == 0:
            comm.Gather(local_field[1:-1, :], field, root=0)
            comm.Barrier()
            if rank == 0:
                write_field(field, m)
```

每次迭代时

要先调用exchange来进行MPI线程间的通信，`交换数据`

```
# MPI thread communication between up and down
def exchange(field):
    # send down, receive from up
    sbuf = field[-2, :]
    rbuf = field[0, :]
    comm.Sendrecv(sbuf, dest=down, recvbuf=rbuf, source=up)
    # send up, receive from down
    sbuf = field[1, :]
    rbuf = field[-1, :]
    comm.Sendrecv(sbuf, dest=up, recvbuf=rbuf, source=down)
```

使用了`Sendrecv`方法，即发送和接收数据的组合操作。

其中，`sbuf` 和 `rbuf` 分别表示发送缓存和接收缓存。

分别用numpy的方法和pycuda的核函数来实现evolve

```
# main calculate function
def evolve(u, u_previous, a, dt, dx2, dy2, device):
    """Explicit time evolution.
       u:            new temperature field
       u_previous:   previous field
```

```
        a:              diffusion constant
        dt:             time step
        dx2:            grid spacing squared, i.e. dx^2
        dy2:              -- "" --          , i.e. dy^2"""
    if device == 'cpu':
        u[1:-1, 1:-1] = u_previous[1:-1, 1:-1] + a * dt * (
                (u_previous[2:, 1:-1] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[:-2, 1:-1]) / dx2 +
                (u_previous[1:-1, 2:] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[1:-1, :-2]) / dy2)
        u_previous[:] = u[:]
    elif device == 'gpu':
        block_size = (16, 16, 1)
        grid_size = (int(np.ceil(u.shape[0] / block_size[0])),
                     int(np.ceil(u.shape[1] / block_size[1])),
                     1)

        # Call the evolve_kernel with the prepared grid and block sizes
        # import pandas as pd
        # df = pd.DataFrame(u)
        # df.to_csv('u.csv')
        evolve_kernel(driver.InOut(u), driver.InOut(u_previous), np.float64(a),
np.float64(dt),
                                    np.float64(dx2), np.float64(dy2),
np.int32(u.shape[0]),
                                    np.int32(u.shape[1]), block=block_size,
grid=grid_size)
        u_previous[:] = u[:]
    else:
        raise ValueError('device should be cpu or gpu')
```

cuda核函数的实现

```
__global__ void evolve_kernel(double* u, double* u_previous,
                              double a, double dt, double dx2, double dy2,
                              int nx, int ny) {
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
    if (i >= nx - 1 || j >= ny - 1) return;
    u[i * ny + j] = u_previous[i * ny + j] + a * dt * (
                (u_previous[(i + 1) * ny + j] - 2 * u_previous[i * ny + j] +
                 u_previous[(i - 1) * ny + j]) / dx2 +
                (u_previous[i * ny + j + 1] - 2 * u_previous[i * ny + j] +
                 u_previous[i * ny + j - 1]) / dy2);
    u_previous[i * ny + j] = u[i * ny + j];
}
```

在特定的时间节点要进行图片的保存

```
# save image
def write_field(field, step):
    plt.gca().clear()
    # Configure the contour
    plt.title("Contour of Temperature")
    plt.contourf(X, Y, field, colorinterpolation, cmap=colourMap)
    if step == 0:
        plt.colorbar()
    plt.savefig(

 'img/heat_{thread}_{device}_{shape}_{timesteps}_{step}.png'.format(thread=size,
device=device, shape=lenX,

timesteps=timesteps, step=step))
```

遇到的问题和解决

问题

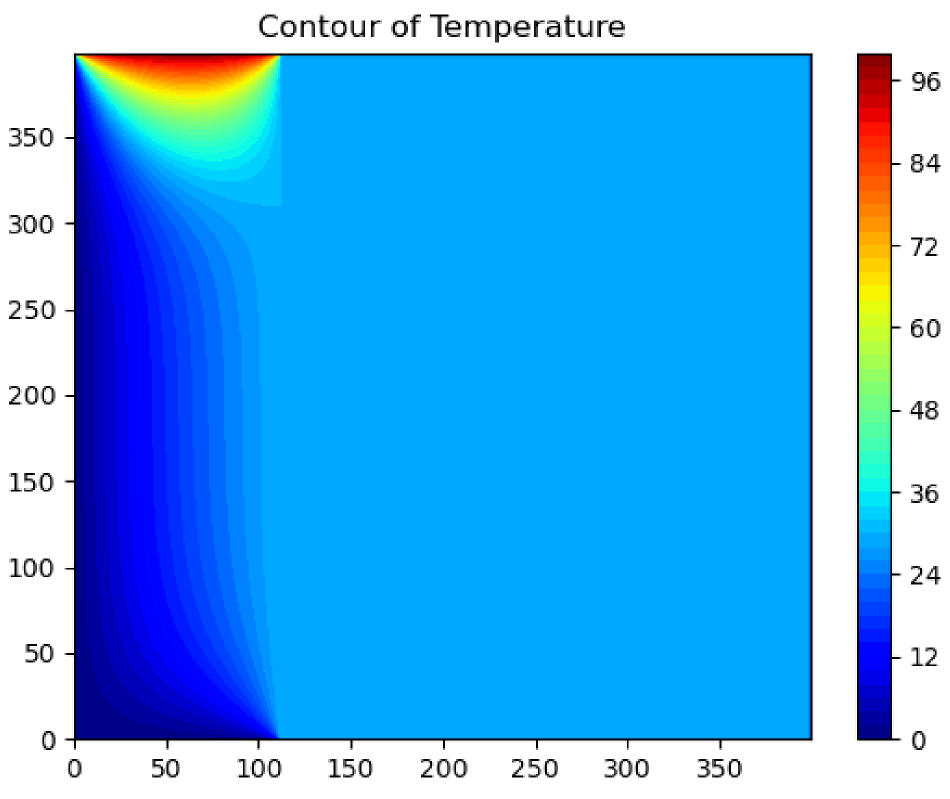
本身cuda toolkit版本为11.8，而pycuda最高支持的cuda版本为11.6

解决

将cuda toolkit降级为11.6，然后根据Installing Pycuda On Windows 10 · Issue #237 · inducer/pycuda (github.com)还要在环境变量中添加新的环境变量`CUDA_PATH`为cuda toolkit的根目录(注意不要设置为cuda toolkit的bin目录)

问题

在调用多线程的mpi分片然后调用cuda核函数来计算时，仅会有第一个线程的区域来计算，如下图



解决

原先的cuda是直接copy别人的代码heat-equation/core_cuda.cu, 和这个mpi配合起来还是有些问题

```
/* Update the temperature values using five-point stencil */
__global__ void evolve_kernel(double *currdata, double *prevdata, double a,
double dt, int nx, int ny,
                       double dx2, double dy2)
{

    /* Determine the temperature field at next time step
     * As we have fixed boundary conditions, the outermost gridpoints
     * are not updated. */
    int ind, ip, im, jp, jm;

    // CUDA threads are arranged in column major order; thus j index from x, i
from y
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    if (i > 0 && j > 0 && i < nx+1 && j < ny+1) {
        ind = i * (ny + 2) + j;
        ip = (i + 1) * (ny + 2) + j;
        im = (i - 1) * (ny + 2) + j;
    jp = i * (ny + 2) + j + 1;
    jm = i * (ny + 2) + j - 1;
        currdata[ind] = prevdata[ind] + a * dt *
          ((prevdata[ip] -2.0 * prevdata[ind] + prevdata[im]) / dx2 +
          (prevdata[jp] - 2.0 * prevdata[ind] + prevdata[jm]) / dy2);

    }

}
```

把本身的cpu的numpy代码

```
u[1:-1, 1:-1] = u_previous[1:-1, 1:-1] + a * dt * (
                (u_previous[2:, 1:-1] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[:-2, 1:-1]) / dx2 +
                (u_previous[1:-1, 2:] - 2 * u_previous[1:-1, 1:-1] +
                 u_previous[1:-1, :-2]) / dy2)
        u_previous[:] = u[:]
```

转换成了cuda的形式

```
__global__ void evolve_kernel(double* u, double* u_previous,
                                  double a, double dt, double dx2, double dy2,
                                  int nx, int ny) {
        int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
        int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
        if (i >= nx - 1 || j >= ny - 1) return;
        u[i * ny + j] = u_previous[i * ny + j] + a * dt * (
                        (u_previous[(i + 1) * ny + j] - 2 * u_previous[i * ny +
j] +
                         u_previous[(i - 1) * ny + j]) / dx2 +
                        (u_previous[i * ny + j + 1] - 2 * u_previous[i * ny + j]
+
                         u_previous[i * ny + j - 1]) / dy2);
        u_previous[i * ny + j] = u[i * ny + j];
    }
```

python端做好处理

```
elif device == 'gpu':
    block_size = (16, 16, 1)
    grid_size = (int(np.ceil(u.shape[0] / block_size[0])),
                 int(np.ceil(u.shape[1] / block_size[1])),
                 1)

    # Call the evolve_kernel with the prepared grid and block sizes
    evolve_kernel(driver.InOut(u), driver.InOut(u_previous), np.float64(a),
np.float64(dt),
                                np.float64(dx2), np.float64(dy2),
np.int32(u.shape[0]),
                                np.int32(u.shape[1]), block=block_size,
grid=grid_size)
    u_previous[:] = u[:]
```

成功跑通

不同数据规模的测试

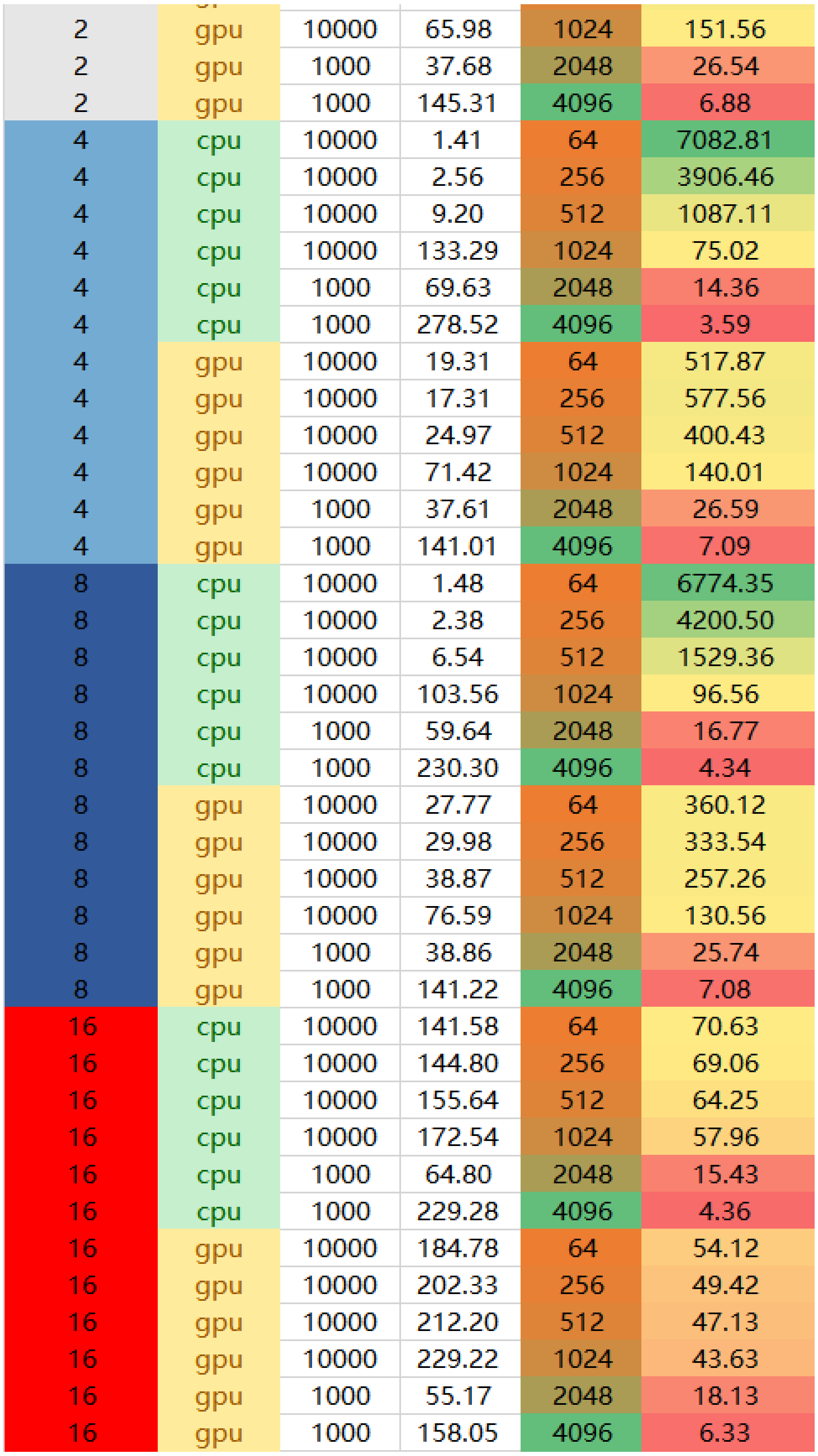
批处理测试

写了一个bat来分别对不同`线程数`，`设备`，`field形状大小`来进行时间的测试，由于`2048×2048`和`4096×4096`的数据量太大，所以`timestamps`变为了其他的`1/10`，最后设计的标准是`it/s`所以还是没有太大问题

```
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 2 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 2 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 2 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 2 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 2 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 2 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 4 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 4 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 4 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 4 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 4 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 4 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 8 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 8 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 8 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 8 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 8 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 8 python ../main.py -d gpu -t 1000 -i 100 -l 4096

mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 64
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 256
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 512
mpiexec -np 16 python ../main.py -d cpu -t 10000 -i 1000 -l 1024
mpiexec -np 16 python ../main.py -d cpu -t 1000 -i 100 -l 2048
mpiexec -np 16 python ../main.py -d cpu -t 1000 -i 100 -l 4096
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 64
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 256
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 512
mpiexec -np 16 python ../main.py -d gpu -t 10000 -i 1000 -l 1024
mpiexec -np 16 python ../main.py -d gpu -t 1000 -i 100 -l 2048
mpiexec -np 16 python ../main.py -d gpu -t 1000 -i 100 -l 4096
```

测试结果

MPI_thread	device	timesteps	time(s)	shape	it/s
2	cpu	10000	1.48	64	6772.58
2	cpu	10000	3.41	256	2934.66
2	cpu	10000	35.64	512	280.57
2	cpu	10000	177.49	1024	56.34
2	cpu	1000	88.70	2048	11.27
2	cpu	1000	350.17	4096	2.86
2	gpu	10000	9.46	64	1056.79
2	gpu	10000	11.76	256	850.33
2	gpu	10000	21.98	512	455.02



2	gpu	10000	65.98	1024	151.56
2	gpu	1000	37.68	2048	26.54
2	gpu	1000	145.31	4096	6.88
4	cpu	10000	1.41	64	7082.81
4	cpu	10000	2.56	256	3906.46
4	cpu	10000	9.20	512	1087.11
4	cpu	10000	133.29	1024	75.02
4	cpu	1000	69.63	2048	14.36
4	cpu	1000	278.52	4096	3.59
4	gpu	10000	19.31	64	517.87
4	gpu	10000	17.31	256	577.56
4	gpu	10000	24.97	512	400.43
4	gpu	10000	71.42	1024	140.01
4	gpu	1000	37.61	2048	26.59
4	gpu	1000	141.01	4096	7.09
8	cpu	10000	1.48	64	6774.35
8	cpu	10000	2.38	256	4200.50
8	cpu	10000	6.54	512	1529.36
8	cpu	10000	103.56	1024	96.56
8	cpu	1000	59.64	2048	16.77
8	cpu	1000	230.30	4096	4.34
8	gpu	10000	27.77	64	360.12
8	gpu	10000	29.98	256	333.54
8	gpu	10000	38.87	512	257.26
8	gpu	10000	76.59	1024	130.56
8	gpu	1000	38.86	2048	25.74
8	gpu	1000	141.22	4096	7.08
16	cpu	10000	141.58	64	70.63
16	cpu	10000	144.80	256	69.06
16	cpu	10000	155.64	512	64.25
16	cpu	10000	172.54	1024	57.96
16	cpu	1000	64.80	2048	15.43
16	cpu	1000	229.28	4096	4.36
16	gpu	10000	184.78	64	54.12
16	gpu	10000	202.33	256	49.42
16	gpu	10000	212.20	512	47.13
16	gpu	10000	229.22	1024	43.63
16	gpu	1000	55.17	2048	18.13
16	gpu	1000	158.05	4096	6.33

可以发现

随着MPI线程数的增多shape越小的field运算速率越慢，而shape越大的运算速率越快，同样的gpu在shape较小的field中运算效率较低，而shape逐渐增大也与cpu的计算效率拉开差距，可以说明数据量更大的运算，就需要更多的MPI线程和gpu的参与。

若数据量较小的话，MPI线程越多可能会造成线程间通讯的浪费更多，GPU和host的通信也同样会有损失。

Advertising prediction based on CTR

数据集

数据集中的部分属性解释如下：

```
click: 是否点击,1表示被点击,0表示没被点击
hour: 广告被展现的日期+时间，格式为YYMMDDHH
banner_pos: 广告投放在屏幕上的位置
site_id: 投放站点id
site_domain: 站点域名
site_category: 站点分类
app_id: appId
app_domain: app域名
app_category: app分类
device_id: 设备Id
device_ip: 设备Ip
device_model: 设备型号
device_type: 设备型号
C1,C14-C21: 匿名分类变量
```

读取数据

```python
train_file = './data/data/DataSet/train.csv'

chunk_size = 10 ** 6   # 每次导入内存的数据量
num = 0
train_set = pandas.DataFrame()

# 获得训练集
for chunk in pandas.read_csv(train_file, chunksize=chunk_size):
    num += 1
    # 以0.2的概率加入train_set
    train_set = pandas.concat([train_set, chunk.sample(frac=.2, replace=False,
random_state=123)], axis=0)
    print('导入第'+str(num)+'块')
```

特征分析与数据清洗

缺失数据处理

首先，删去train_set的id属性

```python
train_set = train_set.drop(['id'], axis=1)
```

检测是否有空数据

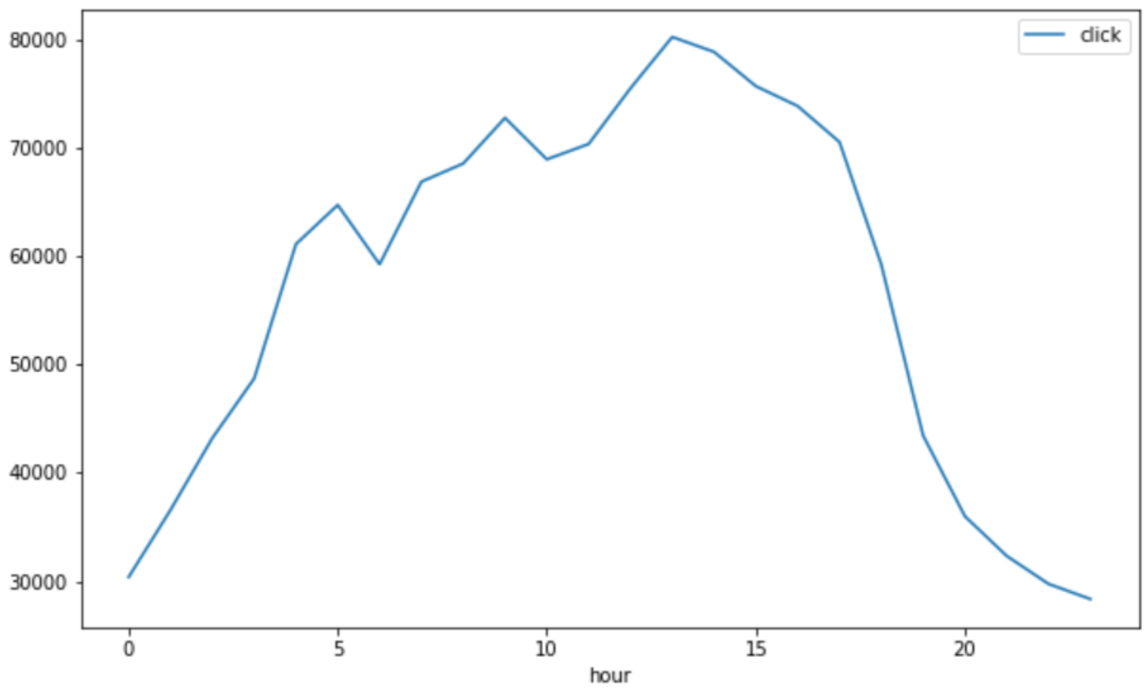
```python
print(train_set.isnull().sum())
```

对关键属性的分析

hour属性的格式为YYMMDDHH，对于广告点击率的变化，年月日属性对点击率的变化应该是没有大的影响的，而一个广告在一天之内投放的时间段，应该对广告的点击率是有较大影响的。 因此，对于hour属性而言，对点击率预测有帮助的应该是广告投放的时间段，我我把Hour属性的年、月、日属性去掉，只留下一天之内的时间段。

查看广告投放时间段与广告点击的关系

```
def tran_hour(x):
    return x % 100

train_set['hour'] = train_set.hour.apply(tran_hour)
# 投放时间段与广告点击的关系
train_set.groupby('hour').agg({'click':'sum'}).plot(figsize=(10,6))
```



特征类型的判定

```
train_set.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8085793 entries, 373315 to 40393381
Data columns (total 23 columns):
click               int64
hour                int64
C1                  int64
banner_pos          int64
site_id             object
site_domain         object
site_category       object
app_id              object
app_domain          object
app_category        object
device_id           object
device_ip           object
```

```
device_model         object
device_type          int64
device_conn_type     int64
C14                  int64
C15                  int64
C16                  int64
C17                  int64
C18                  int64
C19                  int64
C20                  int64
C21                  int64
dtypes: int64(14), object(9)
```

查看各个特征的可能取值个数

对于int 类型的特征，我们可以通过value_count来查看每个特征的取值情况，通过len函数来查看各个特征的可能取值个数。

```
len_of_feature_count = []
for i in train_set.columns[2:23].tolist():
    print(i, ':', len(train_set[i].astype(str).value_counts()))
    len_of_feature_count.append(len(train_set[i].astype(str).value_counts()))
```

```
C1 : 7
banner_pos : 7
site_id : 3822
site_domain : 5151
site_category : 24
app_id : 5914
app_domain : 382
app_category : 30
device_id : 872531
device_ip : 2650597
device_model : 6922
device_type : 5
device_conn_type : 4
C14 : 2503
C15 : 8
C16 : 9
C17 : 431
C18 : 4
C19 : 67
C20 : 169
C21 : 60
```

把int看作str类型

```
# 需要更改为str类型的特征
feature_tran_str = train_set.columns[2:4].tolist() +
train_set.columns[13:23].tolist()

# 将特征类型转变为str类型
for feature in feature_tran_str:
    train_set[feature] = train_set[feature].astype(str)
```

特征取值的限制

同时，可以发现，一些特征的可能取值个数特别多，有上百万个可能取值，这无疑对接下来的建模预测造成麻烦。所以在此将每个属性的可能取值个数进行限制，一旦取值个数超过10，则进行以下操作： 计算每个取值对于点击率的影响程度，将这种程度作为新的属性取值。即依据属性的点击率区分为10个档次，分别取 0，1，2，3，4，5，6，7，8，9。数值越高代表其点击率越高。

```
need_clean_features = []
for i in range(len(len_of_feature_count)):
    if len_of_feature_count[i] > 10:
        need_clean_features.append(train_set.columns[2:23].tolist()[i])

need_clean_features
```

点击率与对应的取值之间的关系

点击率	取值
< m-0.04	0
m-0.04 ~ m-00.3	1
m-0.03 ~ m-0.02	2
m-0.02 ~ m-0.01	3
m-0.01 ~ m	4
m ~ m+00.1	5
m+0.01 ~ m+0.02	6
m+0.02 ~ m+0.03	7
m+0.03 ~ m+0.04	8
> m+0.04	9

平均点击率

```
# 获得数据集的平均点击率
mid_click_rate = train_set.describe().loc['mean','click']
print("平均点击率为: ", mid_click_rate)
```

平均点击率为： 0.170001

为上述需要变动的值进行转换

```
rate_0 = mid_click_rate-0.04
rate_1 = mid_click_rate-0.03
rate_2 = mid_click_rate-0.02
rate_3 = mid_click_rate-0.01
rate_4 = mid_click_rate
rate_5 = mid_click_rate+0.01
rate_6 = mid_click_rate+0.02
```

```
rate_7 = mid_click_rate+0.03
rate_8 = mid_click_rate+0.04

# 定义一个函数用于执行特征取值点击率与新取值之间的映射关系
def obj_clean(X):

    # 用于取得点击率的函数
    def get_click_rate(x):
        temp = train_set[train_set[X.columns[0]] == x]
        result = round((temp.click.sum() / temp.click.count()), 3)
        return result

    # 获取属性新取值的函数
    def get_type(V, str):
        type_.append(V[V[str] <= rate_0].index.tolist())
        type_.append(V[(V[str] > rate_0) & (V[str] <= rate_1)].index.tolist())
        type_.append(V[(V[str] > rate_1) & (V[str] <= rate_2)].index.tolist())
        type_.append(V[(V[str] > rate_2) & (V[str] <= rate_3)].index.tolist())
        type_.append(V[(V[str] > rate_3) & (V[str] <= rate_4)].index.tolist())
        type_.append(V[(V[str] > rate_4) & (V[str] <= rate_5)].index.tolist())
        type_.append(V[(V[str] > rate_5) & (V[str] <= rate_6)].index.tolist())
        type_.append(V[(V[str] > rate_6) & (V[str] <= rate_7)].index.tolist())
        type_.append(V[(V[str] > rate_7) & (V[str] <= rate_8)].index.tolist())
        type_.append(V[V[str] > rate_8].index.tolist())

    # 根据属性的点击率，返回转换后的属性取值
    def clean(x):
        for i in range(10):
            if x in type_[i]:
                return str(i)
        return str(5)

    // obj_clean从此开始执行
    print('执行：', X.columns[0])
    # 频率列表
    frequence = X[X.columns[0]].value_counts()

    # 理论上，所有属性的频率都被映射到这10个取值中，可得到最佳效果
    # 但是，为了节约执行时间，舍去频率排名低于1000的取值
    if len(frequence) > 1000:
        frequence = frequence[:1000]

    frequence = pandas.DataFrame(frequence)
    frequence['new_column'] = frequence.index
    # 调用get_click_rate函数
    frequence['click_rate'] = frequence.new_column.apply(get_click_rate)

    type_ = []
    get_type(frequence, 'click_rate')

    # 返回转换结果
    return X[X.columns[0]].apply(clean)

# 对每个属性执行函数
for i in need_clean_features:
    train_set[[i]] = obj_clean(train_set[[i]])
```

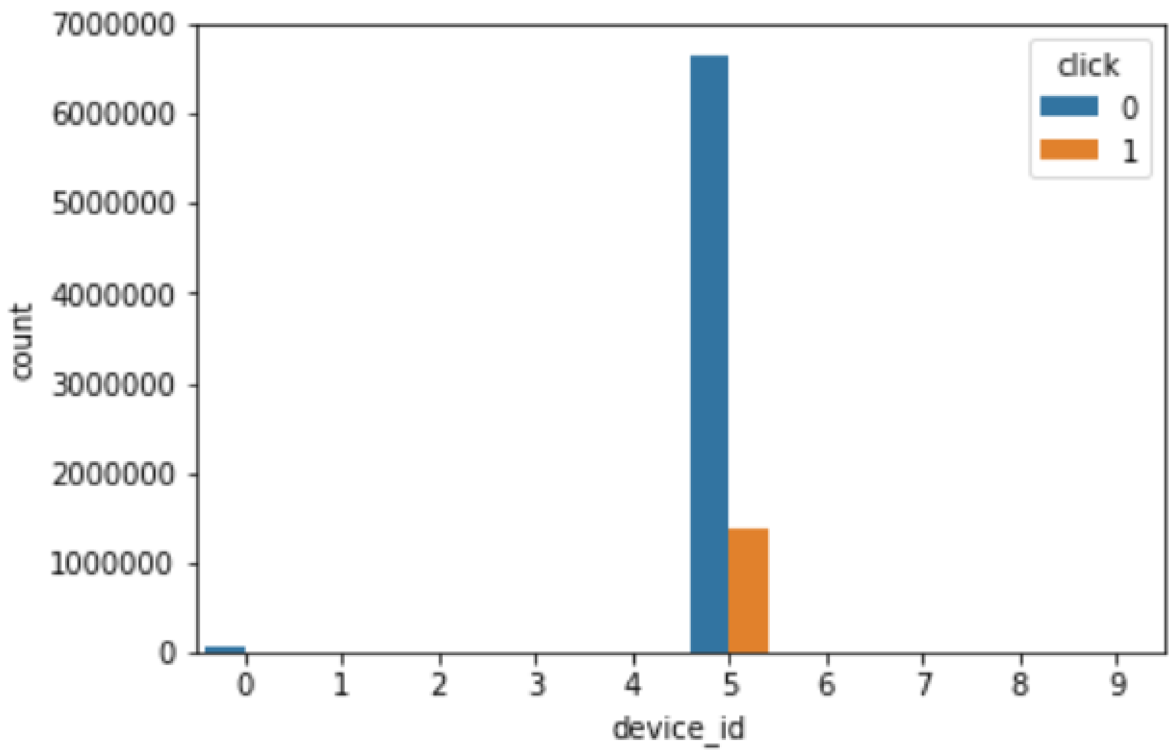
删除无用特征

对数据进行上述处理之后，还需要确认每个特征的取值对用户是否有影响。

对于那些只有一个取值的特征，应该删除

```
# 确认所有特征的取值情况以及对应的点击广告情况
import seaborn as sns
for i in train_set.columns:
    sns.countplot(x = i, hue="click", data=train_set)
    plt.show()
```

device_id属性可近似看作只有一种value



独热编码

```
'''
独热编码的优点
在回归，分类，聚类等机器学习算法中，特征之间距离的计算或相似度的计算是非常重要的，而我们常用的距
离或相似度的计算都是在欧式空间的相似度计算，计算余弦相似性，基于的就是欧式空间。
而我们使用one-hot编码，将离散特征的取值扩展到了欧式空间，离散特征的某个取值就对应欧式空间的某个
点。
将离散型特征使用one-hot编码，确实会让特征之间的距离计算更加合理。
'''

# 对所有特征进行独热编码
train_set = pandas.get_dummies(train_set)

# 将处理过后的数据集导出
train_set.to_csv('./work/DataSet/new_train.csv', index=False)
```

建模与预测

平衡正反标签

资料集十分庞大，而且正向标签仅占全部资料的17%左右，为了缩短决策树的时间，从负向标签的资料中取样，与正向标签的资料拼成一份各占50%的资料集，来平衡权重问题。

划分特征与分类变量

```
# 平衡正反标签
pre_X = train_set[train_set['click'] ==
0].sample(n=len(train_set[train_set['click'] == 1]), random_state=111)

# 合并正反标签的数据
pre_X = pandas.concat([pre_X, train_set[train_set['click'] ==
1]]).sample(frac=1)

# 特征和分类变量的划分
pre_y = pre_X[['click']]  # 特征
pre_X.drop(['click'], axis=1, inplace=True)  # 分类变量
```

将数据集划分为训练集与测试集

训练集占比为80%， 测试集占比为20%

```
from sklearn.model_selection import train_test_split
train_X, test_X, train_y, test_y = train_test_split(pre_X, pre_y,test_size=0.20)
```

xgboost进行建模

```
raw_model = xgb.XGBClassifier()
model.fit(train_X,train_y.values.ravel())
y_pred = model.predict(test_X)
accuracy_score(test_y, y_pred)
```

未进行调参的模型准确度 `0.780527`

Grid Search调参

```
'''
n_estimators：弱学习器的数量
gamma：默认是0，别名是 min_split_loss，在节点分裂时，只有在分裂后损失函数的值下降了（达到
gamma指定的阈值），才会分裂这个节点。gamma值越大，算法越保守（越不容易过拟合）；[0，∞]
max_depth：默认是6，树的最大深度，值越大，越容易过拟合；[0，∞]
'''
# n_estimators
cv_params = {'n_estimators': numpy.linspace(100, 1000, 10, dtype=int)}
regress_model = xgb.XGBRegressor()
gs = GridSearchCV(regress_model, cv_params, verbose=2, refit=True, cv=5,
n_jobs=-1)
gs.fit(train_X, train_y)
# 性能测评
```

```
print("参数的最佳取值：", gs.best_params_)
print("最佳模型得分:", gs.best_score_)
>>> 参数的最佳取值：: {'n_estimators': 100}
>>> 最佳模型得分: 0.1448992153955665

# gamma
cv_params = {'gamma': numpy.linspace(0, 1, 10)}
regress_model = xgb.XGBRegressor()
gs = GridSearchCV(regress_model, cv_params, verbose=2, refit=True, cv=5,
n_jobs=-1)
gs.fit(train_X, train_y)
# 性能测评
print("参数的最佳取值：", gs.best_params_)
print("最佳模型得分:", gs.best_score_)
>>> 参数的最佳取值：  {'gamma': 1.0}
>>> 最佳模型得分: 0.1810817949977097

# max_depth
cv_params = {"max_depth":range(1,20)}
regress_model = xgb.XGBRegressor()
gs = GridSearchCV(regress_model, cv_params, verbose=2, refit=True, cv=5,
n_jobs=-1)
gs.fit(train_X, train_y)
# 性能测评
print("参数的最佳取值：", gs.best_params_)
print("最佳模型得分:", gs.best_score_)

>>> 参数的最佳取值：  {'max_depth': 2}
>>> 最佳模型得分: 0.180575252255935
```

最佳参数

可以看到，最佳参数为{'n_estimators':100, 'gamma': 1.0,'max_depth': 2}

```
# 使用xgboost 建模，并指定先前调参得到的最佳参数
model = XGBClassifier(n_estimators=100, gamma=1.0, max_depth=2)
model.fit(train_X,train_y.values.ravel())
y_pred = model.predict(test_X)
accuracy_score(test_y, y_pred)
```

正确率:0.814047