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

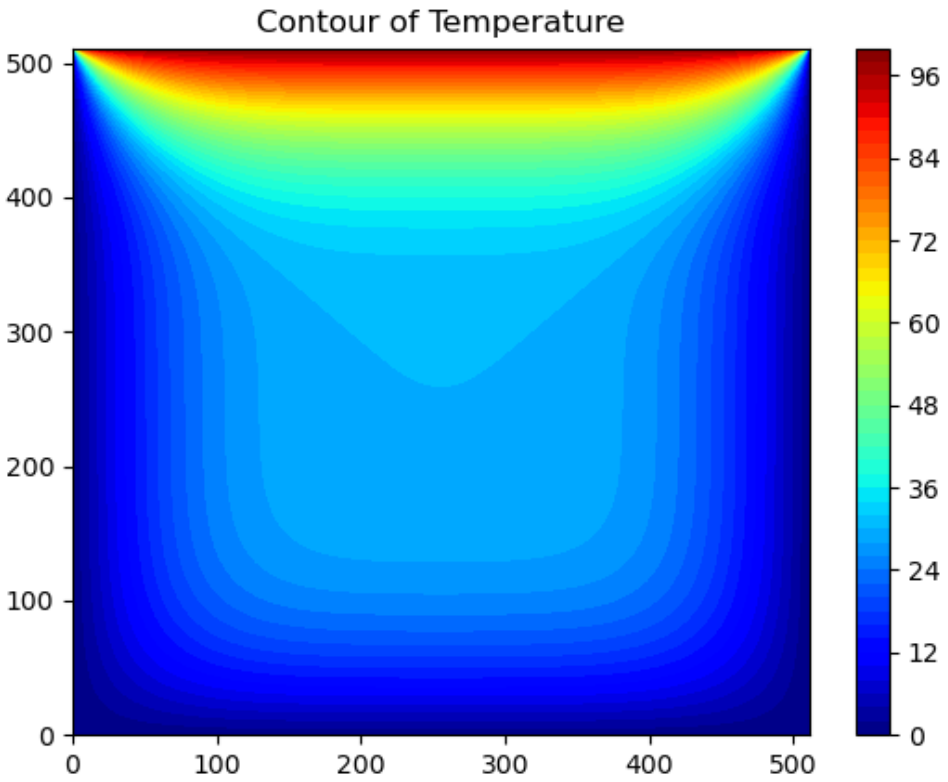
$$\nabla^2 u = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2}$$

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

```
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

```
# 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。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

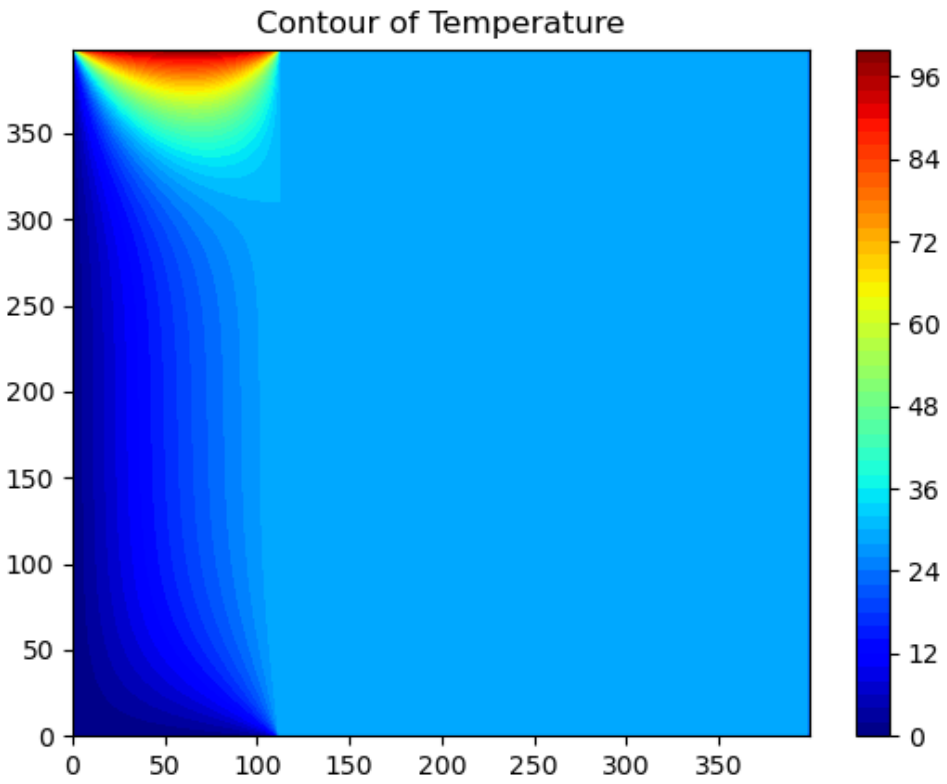
解决

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	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。