# 2D Heat and Wave Equation with PYCUDA

Alfonso Conte

# 2D Heat Equation

### Discretization

The heat equation in 2D can be written as:

$$\frac{\delta u}{\delta t} = k\Delta^2 u = k[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}] \tag{1}$$

As usual, our goal is to derive an explicit scheme that will simplify the resolution of this equation by exploiting the characteristics and potential of the GPU. In fact the equation 1 is a PDE (partial differential equation) and in order to solve it numerically we need to linearize it.
By approximating the temporal derivative with a forward difference, the spatial derivative with a central difference, and gathering $u^n + 1$ on the left hand side and $u^n$ on the right, we obtain:

$$\frac{1}{\Delta t}(u_{i,j}^{n+1} - u_{i,j}^n) = k[\frac{1}{\Delta x^2}(u_{i-1,j}^n) - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{1}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n)] \tag{2}$$

ending up with:

$$u_{i,j}^{n+1} = u_{i,j}^n + k\frac{\delta t}{\delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n + k\frac{\delta t}{\delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n) \tag{3}$$

The discretization is stable if the following CFL conditions are met:

$$\frac{1}{2} < \frac{k\Delta t}{\Delta x^2}, \frac{1}{2} < \frac{k\Delta t}{\Delta y^2} \tag{4}$$

or equivalently:

$$\Delta t < min(\frac{\Delta x^2}{2k}, \frac{\Delta y^2}{2k}) \tag{5}$$

Given initial conditions, the equation can be solved.
So, using **n** as the time index, and **i, j** as the space-indeces we need the temperature at the previous time in order to calculate the ones for the current time. In order to simplify the management of calculations, it was decided to linearize the matrix and treat it as a simple one-dimensional vector.

## Code with pycuda

The language used was Pycuda which gives access to Nvidia's CUDA parallel computation API from Python. It has serveral advantages:

- Object cleanup tied to lifetime of objects. This idiom makes it much easier to write correct, leak- and crash-free code.
- Convenience. Abstractions like pycuda.driver.SourceModule and pycuda.gpuarray.GPUArray make CUDA programming even more convenient than with Nvidia's C-based runtime.
- Completeness. PyCUDA puts the full power of CUDA's driver API at disposal.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCUDA's base layer is written in C++, so all the niceties above are virtually free.

### Starting Phase

First of all we need to import the required packages for pycuda together with the packages necessary to manage vectors and plots.

```
import pandas as pd
import pycuda
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import pycuda.autoinit
```

```
import numpy as np
import matplotlib.pyplot as plt
```

In particular we have:

- **Pycuda**: with all its features.
- **NumPy**: is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- **Matplotlib**: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python

After that, we define the size of each block that will be executed. In this case it was chosen a block of (8,8) with 64 threads.

```
BLOCK_SIZE_X=8
BLOCK_SIZE_Y=8
```

At this point we are ready to create the function through which our kernel cuda will be called.

```
def heatEquationGPU(u0,kappa,dt,dx,dy,num_timesteps):
```

This function takes as parameters **u0** which rapresent the initial temperatures, **kappa** that is the material specific heat conduction constant, **dt**, **dx** e **dy** for the discretization in time and space and the number of timesteps indicating the number of timesteps for which the equation is to be solved.
In addition to the vector u0, as we said, we need 2 vectors because the "n+1" timestep of the simulation is computed based on timestep "n", so we need to make space on the GPU for u0 and also for another vector called u1, the vector that will contain the results of the computation. This is possible through the *cuda.mem_alloc* function. After that, we upload the value of u0 from the CPU to the GPU with the *cuda.memcpy_htod* function.

```
u1_gpu = cuda.mem_alloc(u0.nbytes)
u0_gpu=cuda.mem_alloc(u0.nbytes)
cuda.memcpy_htod(u0_gpu,u0)
```

Let's now compute the block and grid dimensionalities. As said the block are bidimensional (BLOCK SIZE X*BLOCK SIZE Y) previously defined, and the grid will also be bidimensional with the dimension in figure 1

```
#Compute block and grid size:
block=(BLOCK_SIZE_X, BLOCK_SIZE_Y,1)
grid=(int(np.ceil(nx/BLOCK_SIZE_X)),int(np.ceil(ny/BLOCK_SIZE_Y)),1)
```

Figure 1: Block and grid size

Finally, in a for loop, for the number of timesteps given as an input, we invoke the kernel *heatEqnGPU* in order to solve the equation. At the end of each execution it is necessary to swap the pointers of the two vectors **u0_gpu** e **u1_gpu** so the vector representing the state "i+1" will represent the state "i" in the next iteration

```
for n in range(num_timesteps):
    #Compute u1 from u0
    # A for-loop for the space-dimension
    #Loop over all "internal" cells.
    #Move to GPU!!!
    #Kernel function arguments:  float* u1,
    #float* u0,
    #float kappa,
    #float dx,
    #float dt,
    #float dy,
    #unsigned int nx
    #unsigned int ny

    heatEqnGPU(u1_gpu, u0_gpu, np.float32(kappa), np.float32(dt), np.float32(dx),
               np.float32(dy),np.uint32(nx),np.uint32(ny), block=block, grid=grid)
    #Swap the new and old temperatures
    u0_gpu, u1_gpu = u1_gpu, u0_gpu
    if(n%50==0):
        u2=np.empty_like(u0)
        cuda.memcpy_dtoh(u2, u0_gpu)
        plotting_function(u2)
```

After each 50 iteration, we also download the data and plot them to see the advances. At the end of the execution we can download the final values, transfer them from the GPU to the CPU and return.

```
u1 = np.empty_like(u0)
cuda.memcpy_dtoh(u1, u0_gpu)

#Return the updated temperatures
return u1
```

### Kernel

Let's now move on to the kernel implementation. Managing a bidimensional object we need to determine two indeces, unlike the heat equation in 1D. Since the blocks are in two dimensions, the index **i** is determined from the id of the block multiplied by its size along the x axis to which the id of the thread is added. The same is valid for the index **j**, for which simply are considered the dimensions along the y-axis.
The linearization of the matrix forces us to manipulate a simple vector that is logically treated as if it were two-dimensional. For this reason, 5 indices are determined corresponding to the values that logically are on the right, left on the top and bottom of the central element. They are essential to process the heat equation. In particular, the **j*nx** value allows us to logically scroll through the matrix rows sequentially, and then we add the **i** value moving along the x-axis to scroll the columns.

```
heat_eqn_kernel_src = """
__global__ void heatEqn2D(
            float* u1,
            float* u0,
            float kappa,
            float dt,
            float dx,
            float dy,
            unsigned int nx,
            unsigned int ny) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;

    int center = j*nx + i;
    int north =(j+1)*nx+i;
    int south=(j-1)*nx+i;
    int east=j*nx + (i+1);
    int west=j*nx + (i-1);

            if(i>0 && i<nx-1 && j>0 && j<ny-1){
                u1[center] = u0[center]
                + kappa*dt/(dx*dx) * (u0[west] - 2*u0[center] + u0[east])
                + kappa*dt/(dy*dy) * (u0[south]-2*u0[center]+u0[north]);
            }

            else{
                u1[center] = u0[center];
                }
}
"""
```

In the core of the kernel for the indices included in the range [1,nx-2] and [1,ny-2] is calculated the equation. In the kernel function are also setted the boundary conditions, describing how the solution should behave at the boundary of our domain. In this case when the index **i** is equal to **0 or nx-1** simply the value of the vector u1, containing the updated temperature, is made to be equal to u0, and the same happen when **j** is equal to **0 or ny-1**. These conditions are necessary and that's because for some elements of the matrix certain points are not defined (like the east points of the n-1 raw). In this case it is assumed that the temperature of the object at the beginning and at the end is fixed. Let's now define the source module

```
mod = SourceModule(heat_eqn_kernel_src)
heatEqnGPU = mod.get_function("heatEqn2D")
```

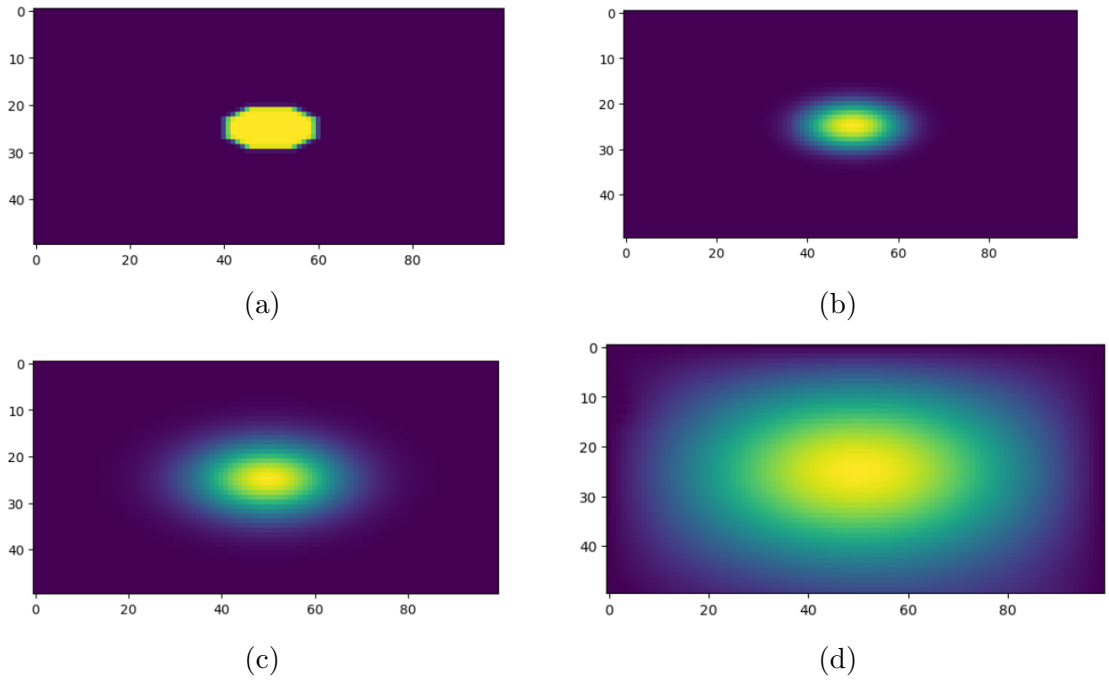
### Run and Plot

```
nx=100
ny=50
initial_temp=np.zeros((ny,nx)).astype(np.float32)
kappa=1
dx=1
dy=2
dt = 0.4*min(dx**2 / (2.0*kappa), dy**2 / (2.0*kappa))
for j in range(ny):
    for i in range(nx):
        x = (i - nx/2.0) * dx
        y = (j - ny/2.0) * dy
        if (np.sqrt(x**2 + y**2) < 10*min(dx, dy)):
            initial_temp[j, i] = 10.0
initial_temp=initial_temp.flatten()
u1 = heatEquationGPU(u0=initial_temp.copy(), kappa=kappa,dt=dt,dx=dx,dy=dy,num_timesteps=200)
```

After setting the problem dimensionality in terms of x-axis **nx** and y-axis **ny** and all the parameters necessary for solving the heat equation, we initialize the initial values and call the *heatEquationGPU* function saving the result in the variable u1. It should be noted that the time interval is computed to validate the Courant-Friedrichs-Levy condition for the particular issue at hand. Specifically, by taking the $min(\frac{\Delta x^2}{2k}, \frac{\Delta y^2}{2k})$ and then multiplying it by 0.4, we guarantee that the condition in equation 5 is satisfied.

Below we also report the plot function and the results obtained from the execution

```
def plotting_function(u):
    u=u.reshape(ny,nx)
    plt.imshow(u.squeeze())
    fig = plt.figure()
```



(a) (b) (c) (d)

# Wave Equation

### Discretization

The linear wave equation in 2D can be written:

$$\frac{\delta^2 u}{\delta t^2} = c\Delta^2 u = c[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}] \tag{6}$$

where **c** is the wave propagation speed coefficient.

The equation, following the same reasoning done for the heat equation, can be discretized as:

$$\frac{1}{\Delta t^2}(u_{i,j}^{n+1} - 2u_{i,j}^n + u_{i,j}^{n-1}) = \frac{c}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^2) + \frac{c}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n) \tag{7}$$

The discretization is stable if the following CFL conditions are met:

$$\frac{1}{2} < \frac{c\Delta t}{\Delta x^2}, \frac{1}{2} < \frac{c\Delta t}{\Delta y^2} \tag{8}$$

or equivalently:

$$\Delta t < min(\frac{\Delta x^2}{2c}, \frac{\Delta y^2}{2c}) \tag{9}$$

The reasoning followed in this case is practically analogous to that already seen for the resumption of the heat equation. The main difference is that two vectors are no longer sufficient to solve the equation, infact the "n+1" timestep of the simulation is computed based on "n" and "n-1" timesteps, so we will have 3 vectors, respectively u0 u1 and u2. In particular u0 contains the initial state, and is used for the time step "n-1". The state vector, u1, is utilized for the time step "n" and finally, the state vector, u2, is employed

for the time step "n+1".
This means that we need to reserve more space on the GPU using the *mem_alloc* function

```
def waveEquationGPU(u0,c,dt,dx,dy,num_timesteps):

    assert(u0.dtype == np.float32)
    u2_gpu = cuda.mem_alloc(u0.nbytes)
    u1_gpu = cuda.mem_alloc(u0.nbytes)
    u0_gpu = cuda.mem_alloc(u0.nbytes)
    cuda.memcpy_htod(u0_gpu,u0)
    cuda.memcpy_htod(u1_gpu,u0)
```

The rest of the implementation is practically the same as the one already seen for the heat equation. Once again it is considered a linearized version of the matrices suitable to contain the results of the equation and at each resolution cycle for a certain timestep it is necessary to exchange the pointers of the three vectors mentioned before. Infact at the completion of a timestep, u2 refers to the state that has been updated and we can swap the pointers such that the vector representing the state "i+1" will represent the state "i" in the next iteration.

```
for n in range(num_timesteps):
    waveEqnGPU(u2_gpu,u1_gpu,u0_gpu, np.float32(c), np.float32(dt), np.float32(dx),
               np.float32(dy),np.uint32(nx),np.uint32(nx), block=block, grid=grid)
    #Swap the new and old pointers
    u0_gpu, u1_gpu, u2_gpu = u1_gpu, u2_gpu, u0_gpu
```

### Kernel

The main difference with the heat equation is obviously the kernel implementation. We will still have two indices **i** and **j** to scroll through the matrix and given the linearization we will determine the points of north, south, east and west in the same way as the heat equation. Clearly the discretized equation will be different together with the boundary conditions. In fact, when i and j reach the boundary values 0 and ny -1 or ny-1 the vector u2 is updated with the near defined point (for example when i is equal to 0 we update the u2 value at the center with the one in vector u1 defined at the east).

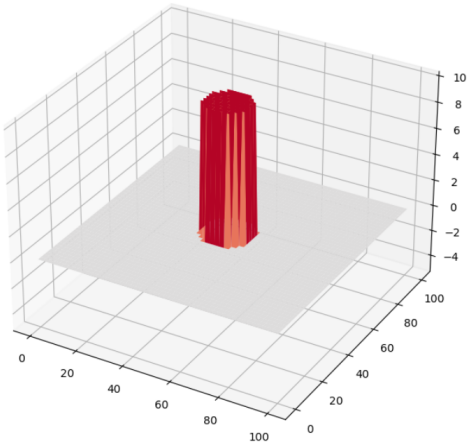
```
wave_eqn_kernel_src = """
__global__ void waveEqn2D(
            float* u2,
            float* u1,
            float* u0,
            float c,
            float dt,
            float dx,
            float dy,
            unsigned int nx,
            unsigned int ny) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;

    int center = j*nx + i;
    int north =(j+1)*nx+i;
    int south=(j-1)*nx+i;
    int east=j*nx + (i+1);
    int west=j*nx + (i-1);

            if(i>0 && i<nx-1 && j>0 && j<ny-1){
                u2[center] = 2.0f*u1[center] - u0[center]
                + (c*dt*dt)/(dx*dx) * (u1[west] - 2.0f*u1[center] + u1[east])
                + (c*dt*dt)/(dy*dy) * (u1[south]-2.0f*u1[center]+u1[north]);
            }
            if (i==0) {
                u2[center] = u1[east];
            }
            else if (i == nx-1) {
                u2[center] = u1[west];
            }
            else if (j == 0) {
                u2[center] = u1[north];
            }
            else if (j == ny-1) {
                u2[center] = u1[south];
            }
```

### Run and Plot

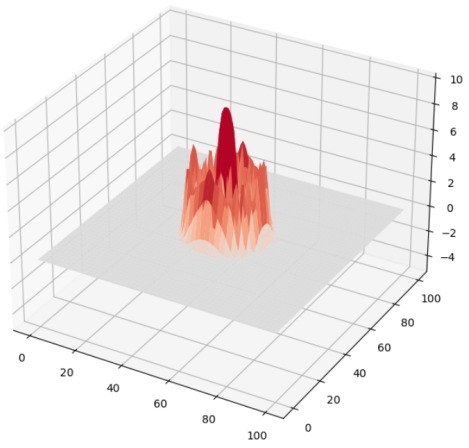
Below we report the code for the initialization of the parameters necessary to solve the equation, the plot function and the results obtained from the execution with increasing timesteps.

```
nx=100
ny=50

c=1
dx=1
dy=2
dt = 0.2 * min(dx / (2.0*c), dy / (2.0*c))
start=np.zeros((ny,nx)).astype(np.float32)
for j in range(ny):
    for i in range(nx):
        #start[j,i]=1.0
        x = (i - nx/2.0) * dx
        y = (j - ny/2.0) * dy
        if (np.sqrt(x**2 + y**2) < 10*min(dx, dy)):
            start[j, i] = 10.0

start=start.flatten()
print("Plot of the initial state")
funzione_plot(start)
print("Plot of the updated state")
u_final=waveEquationGPU(u0=start.copy(), c=c,dt=dt,dx=dx,dy=dy,num_timesteps=500)
```

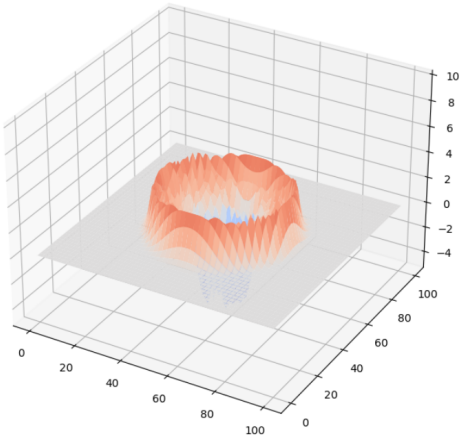
```
def funzione_plot(u_final):
    fig = plt.figure(figsize=(12, 8))
    updated = u_final.reshape(ny,nx)
    ax = fig.add_subplot(111, projection = '3d')
    X = np.linspace(0, nx*dx, nx)
    Y = np.linspace(0, ny*dy, ny)
    X, Y = np.meshgrid(X, Y)
    surf = ax.plot_surface(X, Y, updated[:,:], cmap=cm.coolwarm, linewidth=0, antialiased=True, vmin=-5, vmax=5)
    ax.set_zlim(-5,10)
    fig = plt.figure()
    plt.show()
    plt.close()
```



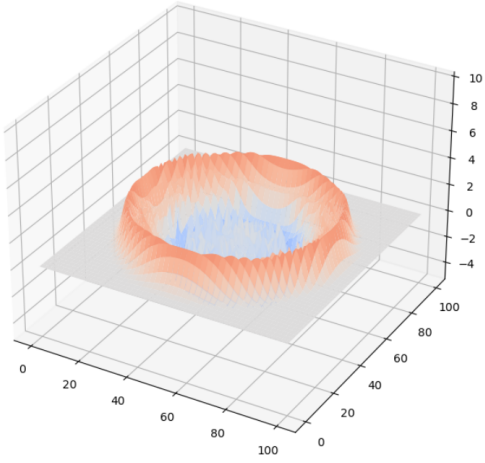

(a)




(b)




(c)




(d)

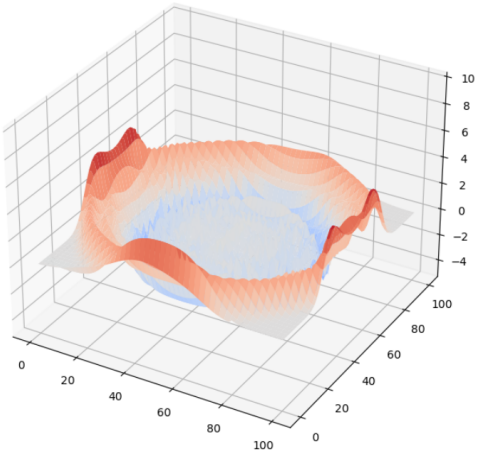



Figure 4: (e)