GPU Ocean Workshop: OpenCL part II

Micro course on

High-performance simulation with high-level languages

André R. Brodtkorb, SINTEF

Outline

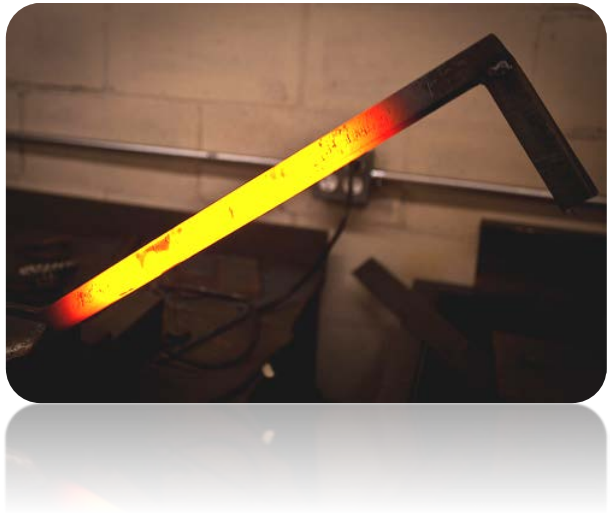
- Part 1a – Introduction
  - Motivation for going parallelParallel algorithm design
  - Programming GPUs with CUDA

- Part 1b – Solving conservation laws with pyopencl
  - The heat equation in 1D and 2D
  - The linear wave equation

The Heat Equation

- The heat equation is a prototypical PDE

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$



- u is the temperature, kappa is the diffusion coefficient, t is time, and x is space.

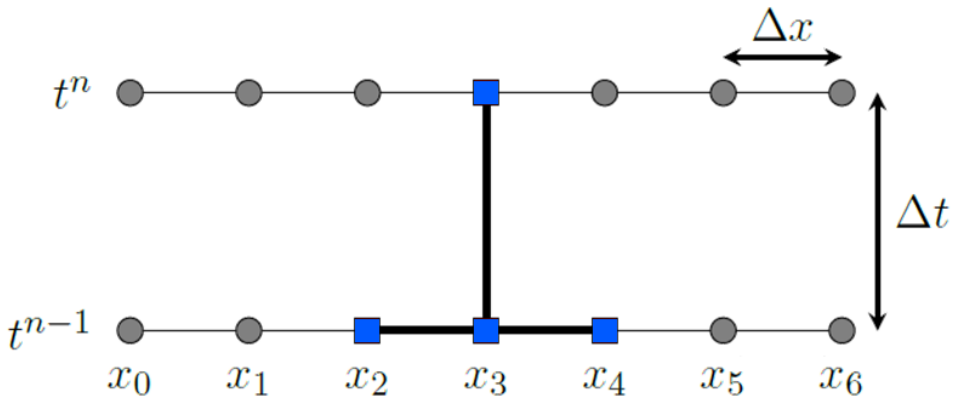
- It states that the rate of change in temperature over time is equal the second derivative of the temperature with respect to space multiplied by the heat diffusion coefficient

Solving the heat equation

- We can discretize this PDE by replacing the continuous derivatives with discrete approximations

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \quad \Rightarrow \quad \frac{1}{\Delta t}(u_i^{n+1} - u_i^n) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

- The discrete approximations use a set of grid points in space and time

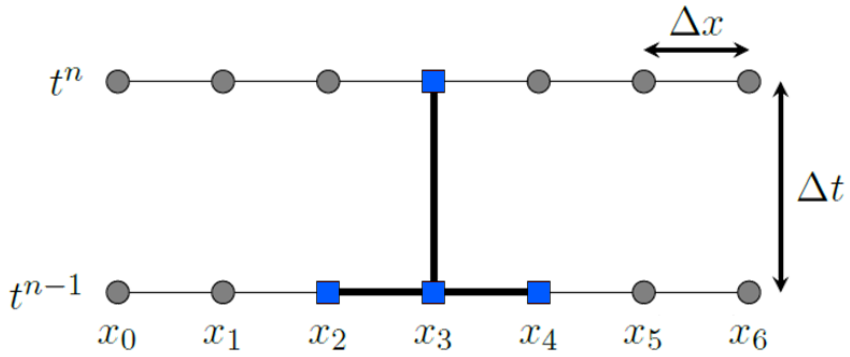


- The choice of discrete derivatives and grid points gives rise to different discretizations with different properties

Explicit scheme for the heat equation

- An explicit scheme for the heat equation gives us an explicit formula for the solution at the next timestep for each cell!
  - It is simply a weighted average of the two nearest neighbors and the cell itself

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n$$



- This is perfectly suited for the GPU: each grid cell at the next time step can be computed independently of all other grid cells!

- However, we must have much smaller time steps than in the implicit scheme

Timestep restriction

- Consider what would happen if you used a timestep of e.g., 10 hours for a stencil computation.
  - It is impossible, numerically, for a disturbance to travel more than one grid cell
  - Physically, however, the disturbance might have travelled half the domain
  - Using too large timesteps leads to unstable simulation results
    (too large timesteps in implicit schemes, you only loose accuracy)

- The restriction on how large the timestep can be is called the Courant-Friedrichs-Levy condition, or more commonly, the CFL condition
  - Find the fastest propagation speed within the domain, and the timestep is inversely proportional to this speed.
  - For the heat equation: $\frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}$

The heat equation in Ipython/Jupyter

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n \qquad r = \frac{\kappa \Delta t}{\Delta x^2} \qquad \frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}$$

- General setup

```
%pylab inline
import numpy as np
```

- Initial conditions

```
nx = 100
u0 = np.random.rand(nx)
u1 = np.empty(nx)
kappa = 1.0
dx = 1.0
dt = ???
nt = 500
```

- Simulation for loop for internal cells

```
for n in range(nt):
    for i in range(1, nx-1):
```

- Explicit heat equation

```
u1[i] = ???
        ???
        ???
```

- Boundary conditions

```
u1[0] = ???
u1[nx-1] = ???
```

- Swap u0 and u1

```
u0, u1 = u1, u0
```

The heat equation in Ipython/Jupyter

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n \qquad r = \frac{\kappa \Delta t}{\Delta x^2} \qquad \frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}$$

- General setup

```
%pylab inline
import numpy as np
```

- Initial conditions

```
nx = 100
u0 = np.random.rand(nx)
u1 = np.empty(nx)
kappa = 1.0
dx = 1.0
dt = 0.8 * dx*dx / (2.0*kappa)
nt = 500
```

- Simulation for loop for internal cells

```
for n in range(nt):
    for i in range(1, nx-1):
```

- Explicit heat equation

```
u1[i] = u0[i]
        + kappa*dt/(dx*dx)
        * (u0[i-1] - 2*u0[i] + u0[i+1])
```

- Boundary conditions

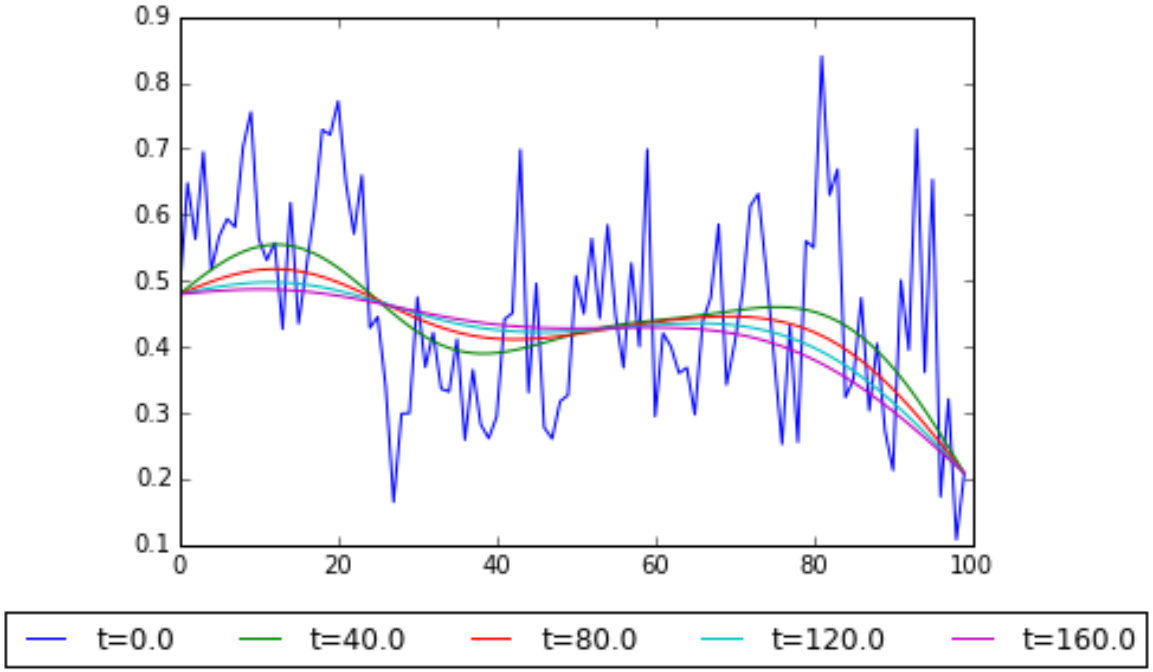
```
u1[0] = u0[0]
u1[nx-1] = u0[nx-1]
```

- Swap u0 and u1

```
u0, u1 = u1, u0
```

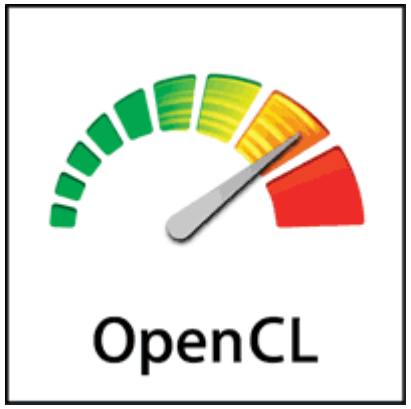
Heat equation results

- We see that given something with random heat inside, our implementation will smear the data, and interpolate the end points



Pyopencl and Jupyter

- Pyopencl enables us to directly access the GPU (or the CPU for that matter) through Python.
- It is a thin Python/C++ wrapper for opencl, and has been developed since 2009
- Has a set of wrappers for easy integration into Ipython/Jupyter



- Enables rapid prototyping of efficient GPU code

Getting started with Ipython and Pyopencl

- First, you need to install prerequisites
  - ipython notebook, numpy, pyopencl itself

- In addition you also need a driver for an OpenCL device
  - For relatively modern Intel CPUs that support SSE 4.1 and 4.2, you can install the Intel OpenCL driver https://software.intel.com/en-us/articles/opencl-drivers
  - For NVIDIA GPUs, the driver is automatically installed for you on Ubuntu!
  - For AMD GPUs, you can download and install drivers http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

- AMD has very good development tools
  NVIDIA has some support, but do not support OpenCL 2.0, only 1.2

Hello World OpenCL 1/3

- Ipython integration of pyopencl

```
%pylab inline
%load_ext pyopencl.ipython_ext
```

- Packages

```
import numpy as np
import pyopencl as cl
```

- Enable verbose compiler output

```
import os
os.environ["PYOPENCL_COMPILER_OUTPUT"] = "1"
```

- Create an OpenCL context and queue

```
cl_ctx = cl.create_some_context()
cl_queue = cl.CommandQueue(cl_ctx)
```

- Create an OpenCL kernel (note keyword)

```
%%cl_kernel
__kernel void add_kernel(
                        __global const float *a,
                        __global const float *b,
                        __global float *c) {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
}
```

Hello World OpenCL 2/3

- Create test input data

```
a = np.linspace(0.0, 2*np.pi).astype(np.float32)
b = np.linspace(0.0, 2*np.pi).astype(np.float32)

a = np.sin(a) + 1.0
b = b*b * 0.1
```

- Upload data to device

```
mf = cl.mem_flags
a_g = cl.Buffer(cl_ctx,
        mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_g = cl.Buffer(cl_ctx,
        mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
```

- Allocate output data

```
c_g = cl.Buffer(cl_ctx, mf.WRITE_ONLY, a.nbytes)
```

- Execute kernel

```
add_kernel(cl_queue, a.shape, None, a_g, b_g, c_g)
```

- Copy result from device to host

```
c = np.empty_like(a)
cl.enqueue_copy(cl_queue, c, c_g)
```

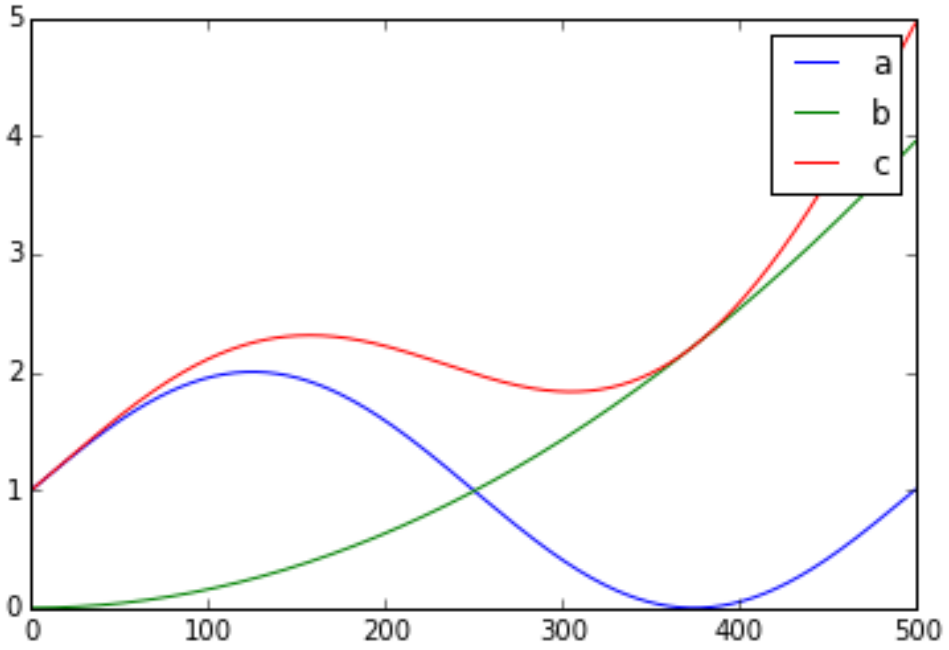
- Check result

```
c_ref = a + b
np.sum(np.abs(c - c_ref))
```

Hello World OpenCL 3/3

- Plot results

```
figure()
plot(a, label='a')
plot(b, label='b')
plot(c, label='c')
legend()
```



The heat equation in OpenCL

- Recall the discretized heat equation

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n \qquad r = \frac{\kappa \Delta t}{\Delta x^2} \qquad \frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}$$

- We also need initial conditions, and boundary conditions to be able to simulate

  Initial conditions
  - $u_i^0 = rand(\quad) \; \forall \; i$

  Boundary conditions (Fixed value, socalled Dirichlet boundary condition)
  - $u_0^n = u_0^0, \quad u_k^n = u_k^0 \quad \forall \; n$
  - k = nx = number of cells

- We see that every u_i^{n+1} can be computed independently for internal cells (i != 0, k)
  - $u_i^{n+1} = u_i^n + r(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

The heat equation in OpenCL

- The OpenCL kernel

```
%%cl_kernel
__kernel void heat_eq_1D(__global float *u1,
                         __global const float *u0,
                         float kappa, float dt, float dx) {
  int i = get_global_id(0);
  int nx = get_global_size(0); //Get total number of cells

  //Internal cells
  if (i > 0 && i < nx-1) {
    u1[i] = u0[i] + kappa*dt/(dx*dx) * (u0[i-1] - 2*u0[i] + u0[i+1]);
  }
  //Boundary conditions (socalled ghost cells)
  else {
    u1[i] = u0[i];
  }
}
```

The heat equation in OpenCL

- Uploading initial conditions

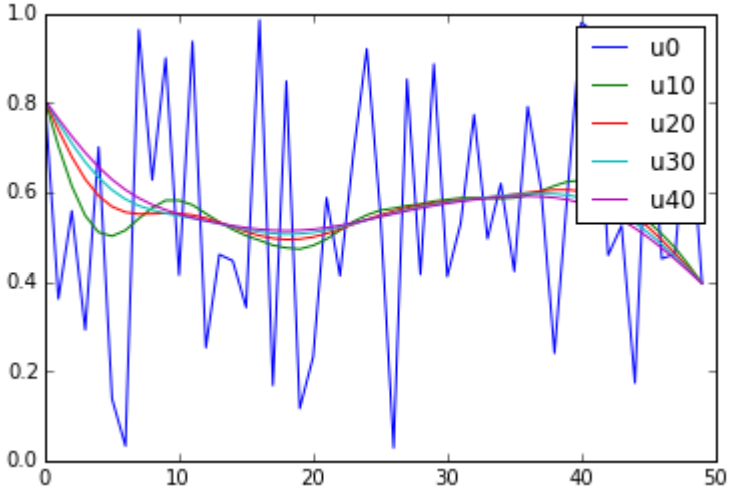
```
#CPU data
u0 = np.random.rand(50).astype(np.float32)

#Number of cells
nx = len(u0)

mf = cl.mem_flags

#Upload data to the device
U0_g = cl.Buffer(cl_ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=u0)

#Allocate output buffers
U1_g = cl.Buffer(cl_ctx, mf.READ_WRITE, u0.nbytes)
```

The heat equation in OpenCL

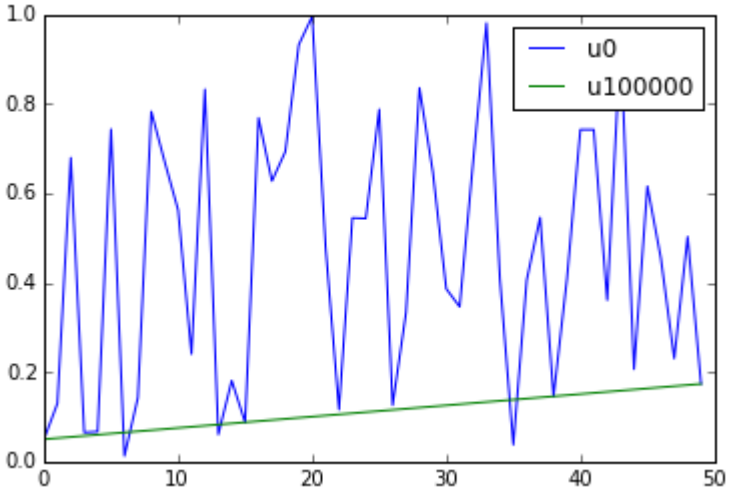
```
#Set number of timesteps
nt = 50

#Calculate timestep size from CFL condition
dt = 0.8 * dx*dx / (2.0*kappa)

#Loop through all the timesteps
for i in range(nt):
    #Execute kernel on device with nx threads
    heat_eq_1D(cl_queue, (nx,1), None, u1_g, u0_g,
            numpy.float32(kappa),  numpy.float32(dt),  numpy.float32(dx))

    #Download and plot solution every fifth iteration
    if (i % 10 == 0):
        u1 = np.empty(nx, dtype=np.float32)
        cl.enqueue_copy(cl_queue, u0_g, u1)
        plot(u1, label="u_"+str(i))

    #Swap variables
    u0_g, u1_g = u1_g, u0_g
```

The heat equation in OpenCL

- The kernel smooths the input data as expected, and the boundary values remain unchanged



- If we run a huge amount of iterations, the boundary conditions (end points) dictate the solution



Two dimensions

- In two dimensions, the heat equation can be written

$$\frac{\partial u}{\partial t} = \kappa \nabla^2 u$$
$$= \kappa \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right]$$

- This simply adds the second or der partial derivative of u with respect to the y dimension.

- For the code, we have to now solve in 2 dimensions, not only one!

Heat Equation in 2D

- In 1D, we started with

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

- And ended up with the numerical scheme

$$u_i^{n+1} = u_i^n + k\frac{\Delta t}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

- In 2D, we start with

$$\frac{\partial u}{\partial t} = \mathrm{k}\nabla^2 u = k\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right]$$

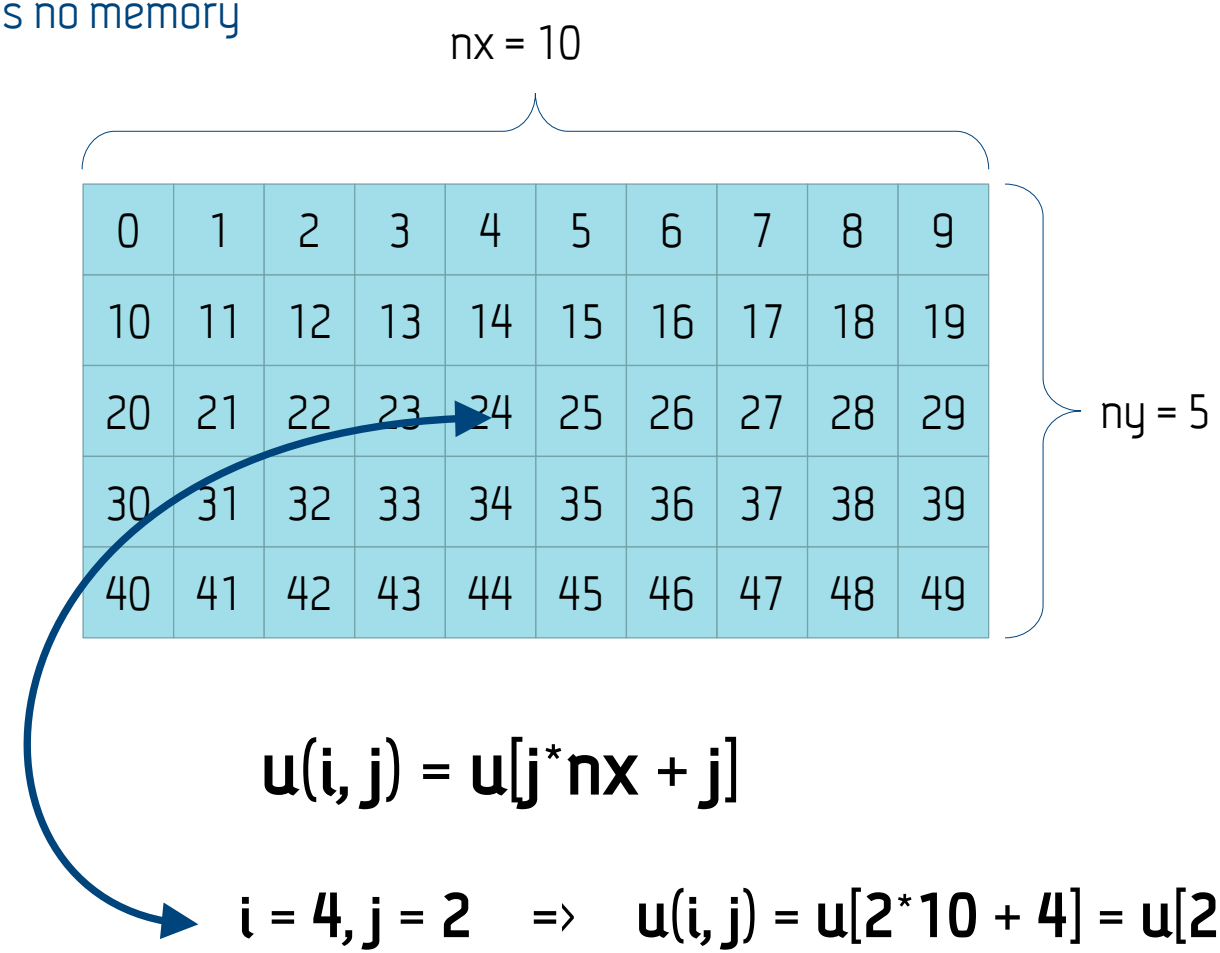
- And end up equivalently with

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

- All we have done, is add a second index, j, and the second order partial derivative of u with respect to y.

2D array indexing

- We typically treat 2D arrays using an interpretation of a 1D array
- It is fast, and wastes no memory



OpenCL Kernel

```
__kernel void heat_eq_2D(__global float *u1, __global const float *u0,
                         float kappa, float dt, float dx, float dy) {
  //Get total number of cells
  int nx = get_global_size(0);
  int ny = get_global_size(1);
  int i = ???;   int j = ???;

  //Calculate the four indices of our neighboring cells
  int center = j*nx + i;
  int north = (j+1)*nx + i;   int south = ???   int east = ???   int west = ???

  //Internal cells
  if (i > 0 && i < nx-1 && j > 0 && j <ny-1) {
    u1[center] = u0[center] + ???
  }
  //Boundary conditions (ghost cells)
  else {
    u1[center] = u0[center];
  }
}
```

Initial conditions

```
nx = 100
ny =nx
kappa = 1.0
dx = 1.0
dy = 1.0
dt = 0.4 * min(dx*dx / (2.0*kappa), dy*dy / (2.0*kappa))
u0 = np.random.rand(ny, nx).astype(np.float32)

mf = cl.mem_flags

#Upload data to the device
u0_g = cl.Buffer(cl_ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=u0)

#Allocate output buffers
u1_g = cl.Buffer(cl_ctx, mf.READ_WRITE, u0.nbytes)
```

Execute kernel

```
nt = 500
for i in range(0, nt):
    #Execute program on device
    heat_eq_2D(cl_queue, (cl_data.nx, cl_data.ny), None,
            u1_g, u0_g,
            numpy.float32(kappa), numpy.float32(dt), numpy.float32(dx), numpy.float32(dy))

    #Swap the two timesteps
    u0_g, u1_g = u1_g, u0_g

    #Plot results
    if (i % 50 == 0):
        figure()
        u0 = np.empty((nx, ny), dtype=np.float32)
        cl.enqueue_copy(cl_queue, u0, u0_g)
        pcolor(u0)
```

Linear Wave Equation

- The heat equation can be written

$$\frac{\partial u}{\partial t} = \mathrm{k}\nabla^2 u = k\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right]$$

  which gave the numerical scheme

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

- The linear wave equation can be written

$$\frac{\partial^2 u}{\partial t^2} = \mathrm{c}\nabla^2 u = c\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right]$$

  which only changes the left hand side. Here c is the wave propagation speed coefficient
  We can write the numerical scheme as

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

Linear Wave Equation

- Rewriting

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

  We get

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

- The major difference with the heat equation is that we now need two timesteps of u to compute the next timestep!

Simulation loop

```
for i in range(0, nt):
    #Execute program on device
    linear_wave_2D(cl_queue, (nx,ny), None,
            u2_g, u1_g, u0_g,
            numpy.float32(c), numpy.float32(dt), numpy.float32(dx), numpy.float32(dy))

    #Impose boundary conditions
    linear_wave_2D_bc(cl_queue, (nx, ny), None, u2_g)

    #Swap variables
    u0_g, u1_g, u2_g = u1_g, u2_g, u0_g
```

Boundary conditions

```
__kernel void linear_wave_2D_bc(__global float* u) {
   int nx = get_global_size(0);    int ny = get_global_size(1);
   int i = get_global_id(0);    int j = get_global_id(1);

   //Calculate the four indices of our neighboring cells
   int center = j*nx + i;
   int north = …;    int south = …;    int east = …;    int west = …;

   if (i == 0) {
      u[center] = u[east];
   }
   else if (i == nx-1) {
      u[center] = u[west];
   }
   else if (j == 0) {
      u[center] = u[north];
   }
   else if (j == ny-1) {
      u[center] = u[south];
   }
}
```

Exercises

- Install Virtualbox
  - https://www.virtualbox.org/wiki/Downloads
    or apt-get install on ubuntu
  - Import the virtualbox image from USB

- Implement the Heat Equation in 2D
  - Most of the code is in the slides, but you have to do a bit of work
  - Look at the Heat Equation in 1D example notebook to get started

- Implement the linear wave equation in 2D
  - Start with the heat equation in 2D, and change the implementation so that it solves the linear wave equation