SHORT COURSE ON HIGH-PERFORMANCE SIMULATION WITH HIGH-LEVEL LANGUAGES

CONSERVATION LAWS ON GPUS

André R. Brodtkorb
Associate Professor, OsloMet – Oslo Metropolitan University
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital

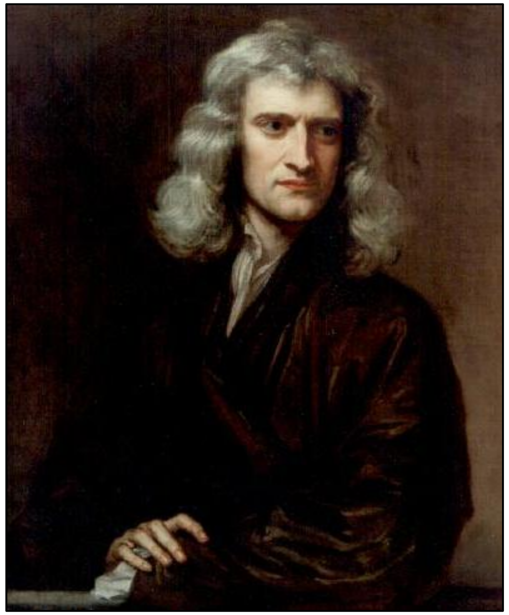
Outline

- Part 1a – Introduction
  - Motivation for going parallel
  - Multi- and many-core architectures
  - Parallel algorithm design
  - Programming GPUs with CUDA

- Part 1b – Solving conservation laws with pyopencl
  - Solving ODEs and PDEs on a computer
  - The heat equation in 1D and 2D
  - The linear wave equation

- Part 1c – Best practices for scientific software development
  - Challenges for scientific software development
  - Best practices for scientific software development

Conservation Laws



Isaac Newton, by Gottfried Kneller, public domain

- A conservation law describes that a quantity is conserved
- Comes from the physical laws of nature
- Example: Newtons first law: When viewed in an inertial reference frame, an object either remains at rest or continues to move at a constant velocity, unless acted upon by an external force.
- Example: Newtons third law: When one body exerts a force on a second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction on the first body.
- More examples: conservation of mass (amount of water) in shallow water, amount of energy (heat) in the heat equation, linear momentum, angular momentum, etc.
- Conservation laws are mathematically formulated as partial differential equations: PDEs

Ordinary Differential Equations (ODEs)

- Let us look at Newtons second law
  - The vector sum of the external forces F on an object is equal to the mass m of that object multiplied by the acceleration vector a of the object:
  - $\vec{F} = m \cdot \vec{a}$

- We know that acceleration, a, is the rate of change of speed over time, or in other words
  - $a = v' \ = \frac{dv}{dt}$

- We can then write Newtons second law as an ODE:
  - $F = m\frac{dv}{dt}$

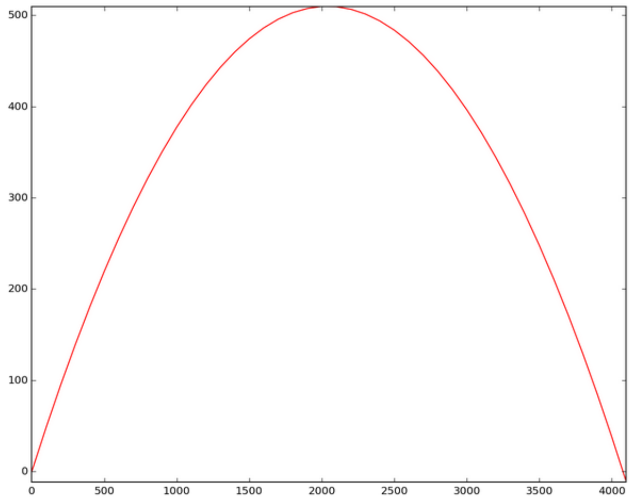
Trajectory of a projectile

- From Newton's second law, we can derive a simple ODE for the trajectory of a projectile

  - Acceleration due to gravity:
    - $\vec{a} = [0, 0, 9.81]$
  - Velocity as a function of time
    - $\vec{v}(t) = \overrightarrow{v_o} + t \cdot \vec{a}$
  - Change in position, p, over time is a function of the velocity
    - $\frac{d\vec{p}}{dt} = \vec{v}(t)$

  - We can solve this ODE analytically with pen and paper, but for more complex ODEs, that becomes infeasible

  - The term "computer" used to be the profession for those who (amongst other things) calculated advanced projectile trajectories (air friction etc.).



Solving a simple ODE numerically

- To solve the ODE numerically on a computer, we discretize it

- To discretize an ODE is to replace the continuous derivatives with discrete derivatives, and to impose a discrete grid.

- In our ODE, we discretize in time, so that

$$\frac{d\vec{p}}{dt} = \vec{v}(t)$$

becomes

$$\frac{\vec{p}^{n+1} - \vec{p}^{n}}{\Delta t} = \vec{v}(n \cdot \Delta t)$$

Here, Δt is the grid spacing in time, and superscript n denotes the time step

Initial conditions

- Recall our discretization

$$\frac{\vec{p}^{n+1} - \vec{p}^{n}}{\Delta t} = \vec{v}(n \cdot \Delta t)$$

Rewriting so that n+1 is on the left hand side, we get an explicit formula

$$\vec{p}^{n+1} = \vec{p}^{n} + \Delta t \cdot \vec{v}(n \cdot \Delta t)$$

- Given initial conditions, that is the initial position, p^0, and the initial velocity, v^0, we can now simulate!

  - Example:

t	p	v
0	0.0	0.0
0.1	p0 + dt*v0 = 0.0	v0 - t*9.81 = -0.981
0.2	p1 - dt*v1 = -0.0981	v0 - t*9.81 = -1.962
0.2	…	…

Projectile trajectory in Python

- Python is an interpreted language that enables rapid prototyping
  - You can write the program as it is being interpreted: no compilation!
  - The simplest example:
    start Python
    enter the text
    print "hello world"
    and the console should reply "hello world"

- Ipython/Jupyter is an interactive shell for running Python with a very nice feature, the Notebook.
  - A web-based editor for Python with cells that you can evaluate one by one
  - We will use Ipython/Jupyter notebook in the lab session tomorrow

Projectile trajectory Python implementation

- Enable in-line plotting

```
%pylab inline
```

- Set initial conditions

```
v0 = np.array([200.0, 100.0])
p0 = np.array([0.0, 0.0])
dt = 0.1
nt = 100
a = np.array([0.0, -9.81])
```

- Create a for-loop with our time-stepping

```
for i in range(nt):
        t = ???
        v1 = ???
        p1 = ???
```

$$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \vec{p}^{\,n+1} = \vec{p}^{\,n} + \Delta t \cdot \vec{v}(n \cdot \Delta t)$$

```
        #Plot
        plot(p1[0], p1[1], 'x')

        #Swap p0 and p1
        p0, p1 = p1, p0
```

Projectile trajectory Python implementation

- Enable in-line plotting

```
%pylab inline
```

- Set initial conditions

```
v0 = np.array([200.0, 100.0])
p0 = np.array([0.0, 0.0])
dt = 0.1
nt = 100
a = np.array([0.0, -9.81])
```

- Create a for-loop with our time-stepping

```
for i in range(nt):
        t = n*dt
        v1 = v0+ t*a
        p1 = p0 + dt*v1

       #Plot
       plot(p1[0], p1[1], 'x')

       #Swap p0 and p1
       p0, p1 = p1, p0
```

$$\left.\begin{array}{l}\texttt{t = n*dt}\\ \texttt{v1 = v0+ t*a}\\ \texttt{p1 = p0 + dt*v1}\end{array}\right\} \vec{p}^{n+1} = \vec{p}^{n} + \Delta t \cdot \vec{v}(n \cdot \Delta t)$$

SINTEF

Particle trajectory results

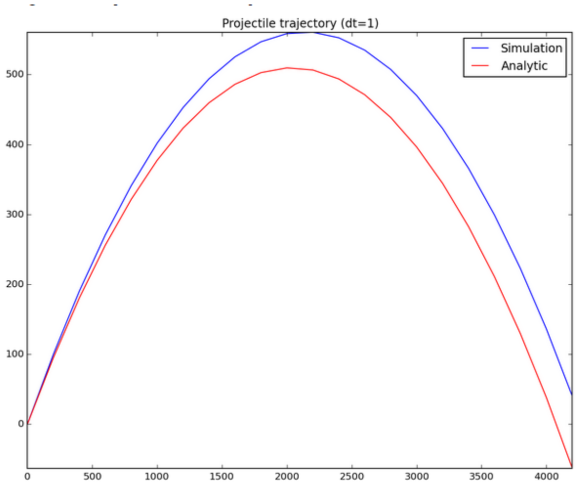
- When writing simulator code it is essential to check for correctness.

- The analytical solution to our problem is

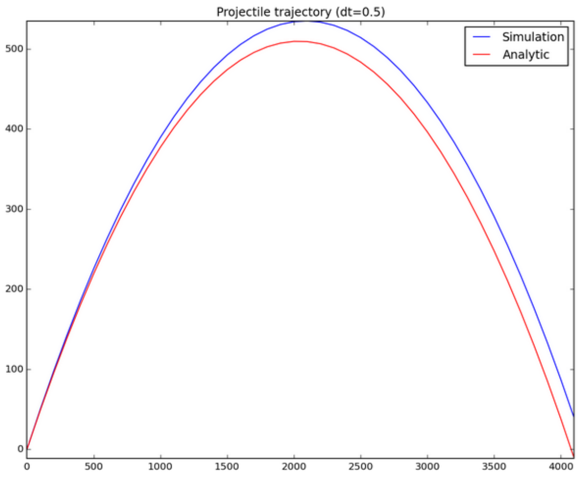
$$p(t) = \frac{1}{2}\vec{a}t^2 + t \cdot v^0 + p^0$$

- Let us compare the solutions

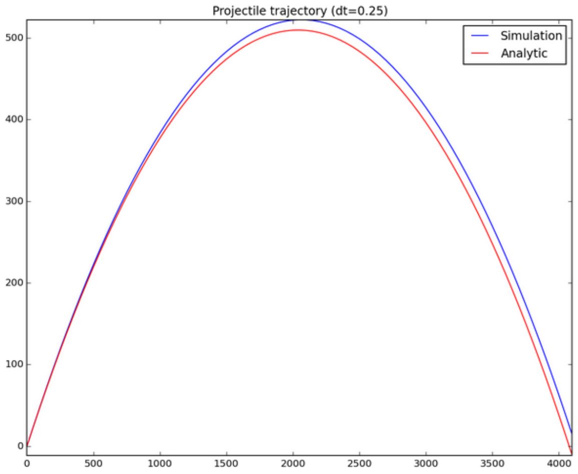
dt=1



dt=0.5



dt=0.25



More accuracy

- We have used a very simple integration rule (or approximation to the derivative)
  - Our rule is known as forward Euler

$$p^{n+1} = p^n + \Delta t \cdot \vec{v}$$

- We can get much higher accuracy with more advanced techniques such as Runge-Kutta 2

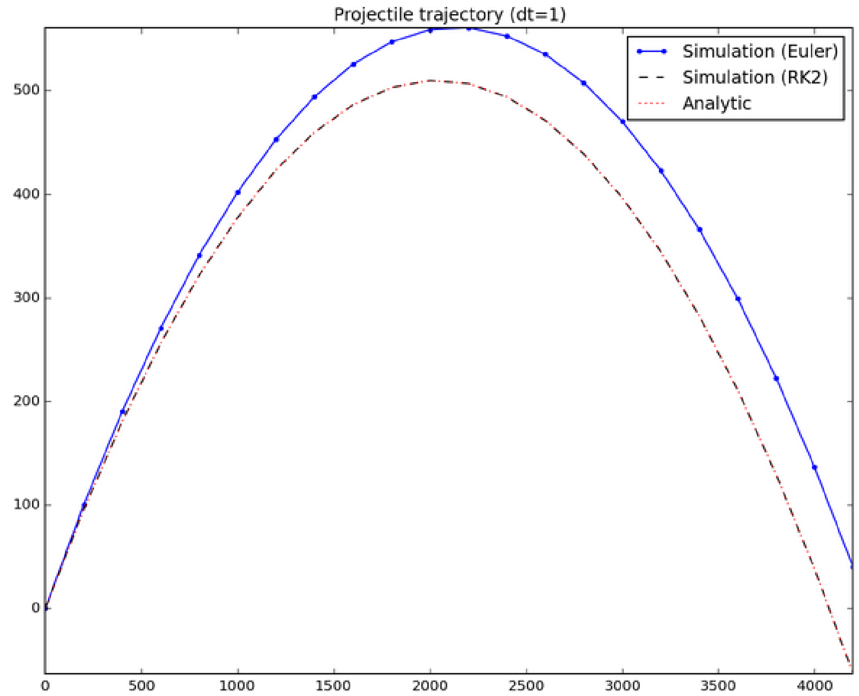
$$p^* = p^n + \Delta t \cdot \vec{v}(n \cdot \Delta t)$$

$$p^{**} = p^* + \Delta t \cdot \vec{v}\big((n+1) \cdot \Delta t\big)$$

$$p^{n+1} = \frac{1}{2}(p^n + p^{**})$$

- In summary, we need to think about how we discretize our problem!

dt=1



Partial Differential Equations (PDEs)

- Partial differential equations (PDEs) are much like ordinary differential equations (ODEs)

- They consist of derivatives, but in this case partial derivatives.

- Partial derivatives are derivatives with respect to *one* variable
  - Example:

$$f(x,y) = x \cdot y^2$$
$$\frac{\partial f(x,y)}{\partial x} = y^2$$
$$\frac{\partial f(x,y)}{\partial y} = 2 \cdot x \cdot y$$

- These are often impossible to solve analytically, and we must discretize them and solve on a computer.

Partial Differential Equations (PDEs)

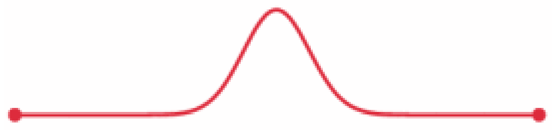
- Many natural phenomena can (partly) be described mathematically as such conservation laws
  - Magneto-hydrodynamics
  - Traffic jams
  - Shallow water
  - Groundwater flow
  - Tsunamis
  - Sound waves
  - Heat propagation
  - Pressure waves
  - …

"Magnificent CME Erupts on the Sun - August 31" by NASA Goddard Space Flight Center - Flickr: Magnificent CME Erupts on the Sun - August 31. Licensed under CC BY 2.0 via Wikimedia Commons

Example: The linear wave equation

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

- Can describes vibration of string (in 1D)



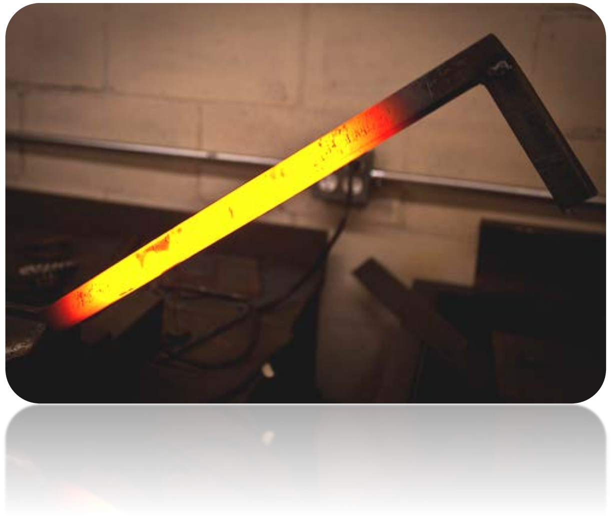
- u is the deflection of the string
- c is a material property (related to wave propagation speed)

Violin image CC0

The Heat Equation

- The heat equation is a prototypical PDE (partial differential equation)

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



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

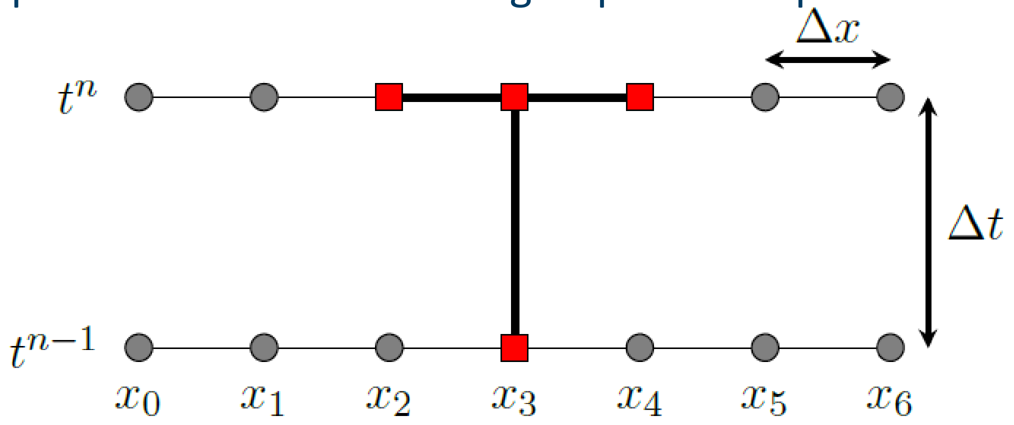
- It states that the rate of change in temperature over time is equal the second derivative of the temperature with respect to space multiplied by the heat diffusion coefficient

Solving the heat equation

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

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

- 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

Solving the heat equation

- From the discretized PDE, we can create a numerical scheme by reordering the terms

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

⇩

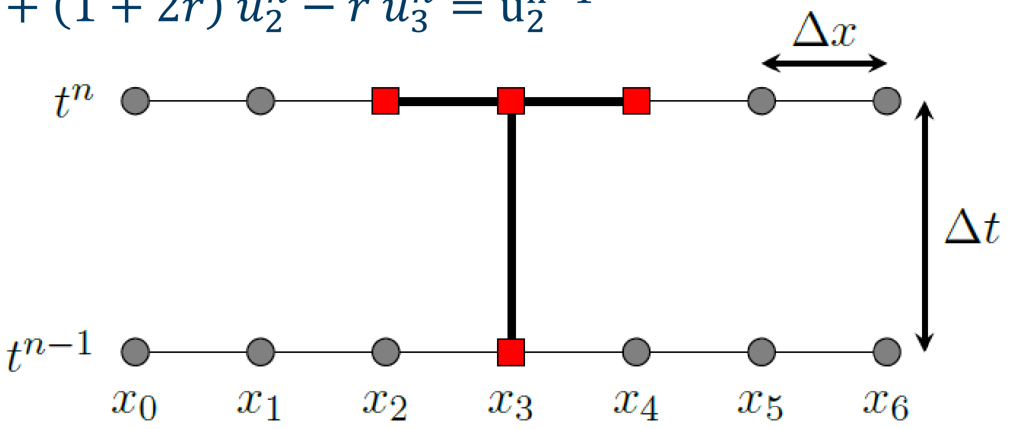
$$-ru_{i-1}^n + (1+2r)u_i^n - ru_{i+1}^n = u_i^{n-1}, \qquad r = \frac{\kappa \Delta t}{\Delta x^2}$$

- This discretization gives us one equation per grid point which we must solve

Solving the heat equation

- We can write up the equation for each cell as follows:

- Cell ui: $-r\, u_{i-1}^{n} + (1 + 2r)\, u_i^n - r\, u_{i+1}^n = \mathrm{u}_{\mathrm{i}}^{\mathrm{n-1}}$
- Cell u1: $-r\, u_0^n + (1 + 2r)\, u_1^n - r\, u_2^n = \mathrm{u}_1^{\mathrm{n-1}}$
- Cell u2: $-r\, u_1^n + (1 + 2r)\, u_2^n - r\, u_3^n = \mathrm{u}_2^{\mathrm{n-1}}$



- Problem:
  - Cell u-1 does not exist (outside domain!)
  - Cell u7 does not exist (outside domain!)
  - These are called boundary conditions (what the temperature is at the boundary)

Solving a PDE

- We organize all the equations we have into a matrix equation Ax=b
  - We gather the *coefficients* in A
  - We gather the unknowns (u^n) in the vector x
  - We gather the known state (u^{n-1})in the vector b

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

$$\left[\begin{array}{ccccccc} \multicolumn{2}{c}{???} & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & \multicolumn{2}{c}{???} \end{array}\right] \begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix} = \begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}$$

- For the first and last equations, we need boundary conditions!

Boundary conditions

- Boundary conditions describe how the solution should behave at the boundary of our domain

- Different boundary conditions give very different solutions!

- A simple boundary condition to implement is "fixed boundaries" / Dirichlet boundaries
  - This simply sets the temperature at the end points to a fixed value

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1{+}2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1{+}2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1{+}2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1{+}2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1{+}2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix} = \begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}$$

Solving the heat equation

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{A}} \underbrace{\begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix}}_{\text{x}} = \underbrace{\begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}}_{\text{b}}$$

- We now have a well-formed problem, if we give some initial heat distribution, u^0
- We can solve the matrix equation $Ax = b$ using linear algebra solvers (Gaussian elimination, conjugate gradients, tri-diagonal solvers, etc.)
- Choosing the right solver is often key to performance: CUBLAS, CUSPARSE, CUSP, …

The Heat Equation on the GPU

- The example so far is quite inefficient and boring…
    - It solves only in 1D
        - Many real-world problems require 2D or 3D simulations
    - It does not utilize any knowledge about the matrix A or the solution
        - A is tridiagonal: we are storing and computing n^2elements, whilst we only need to store the $3n$ non-zero elements
    - It uses a regular grid
        - Non-regular grids give us local refinement where we need it

- Adding more features gives a more complex picture
    - The matrix A quickly gets more complex with more features (2D/3D/non-regular grids/etc.)
    - More complex problems have more equations, and the A matrix must often be re-calculated for each simulation step (non-constant coefficients)

The Heat Equation on the GPU

- The presented numerical scheme is called an *implicit* scheme

- Implicit schemes are often sought after
  - They allow for large time steps,
  - They can be solved using standard tools
  - Allow complex geometries
  - They can be very accurate
  - …

- However…
  - Solution time is often a function of how long it takes to solve Ax=b and linear algebra solvers can be **slow and memory hungry**, especially on the GPU
  - for many time-varying phenomena, we are also interested in the temporal dynamics of the problem

Explicit scheme for the heat equation

- For problems in which disturbances travel at a finite speed, we can change the time derivative from a backward to a forward difference.

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

$$\Downarrow$$

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

- This gives us an *explicit* numerical scheme (compared to the *implicit* scheme already shown)

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

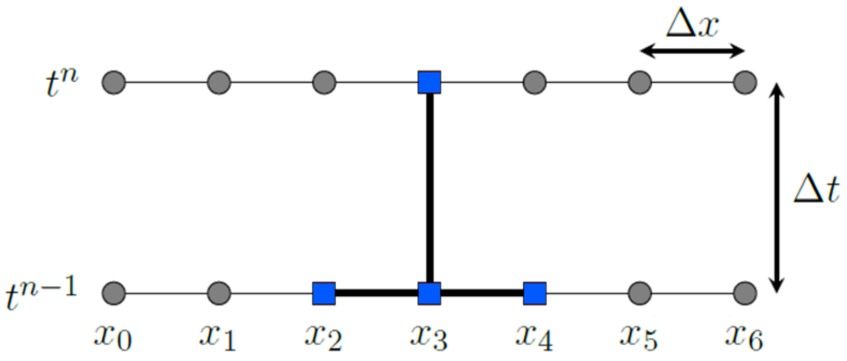
$$\Downarrow$$

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

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

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



- 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 Python / Jupyter

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

- 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 Python / 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

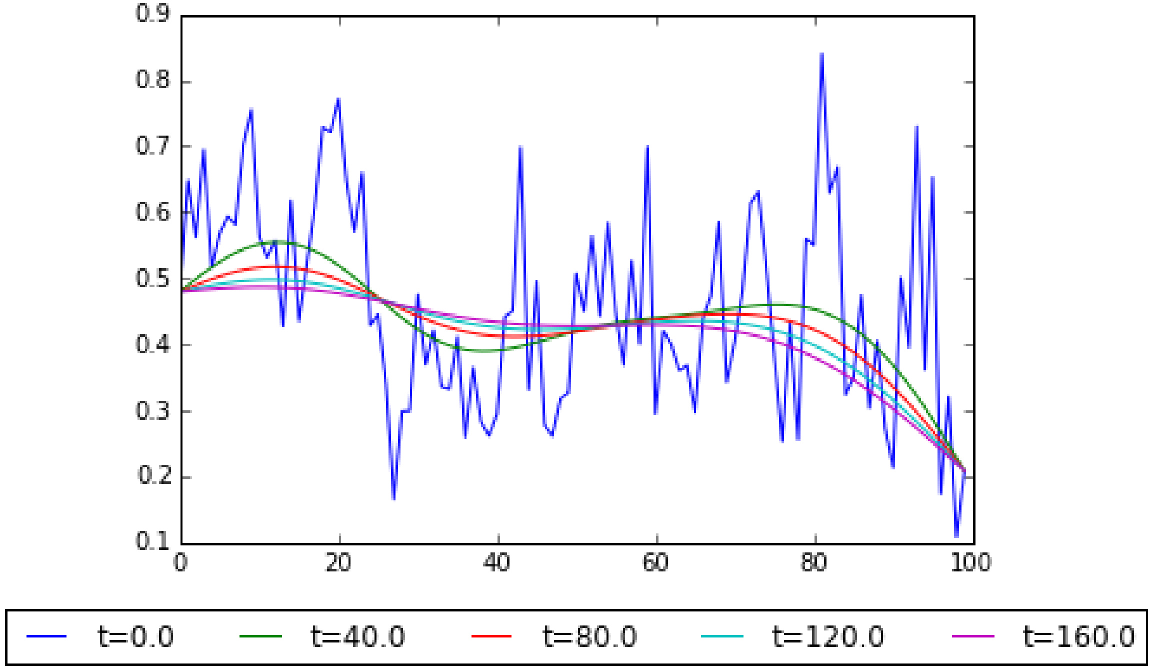
```
u1[0] = u0[0]
u1[nx-1] = u0[nx-1]
```

- 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 Ipyhon/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)
```

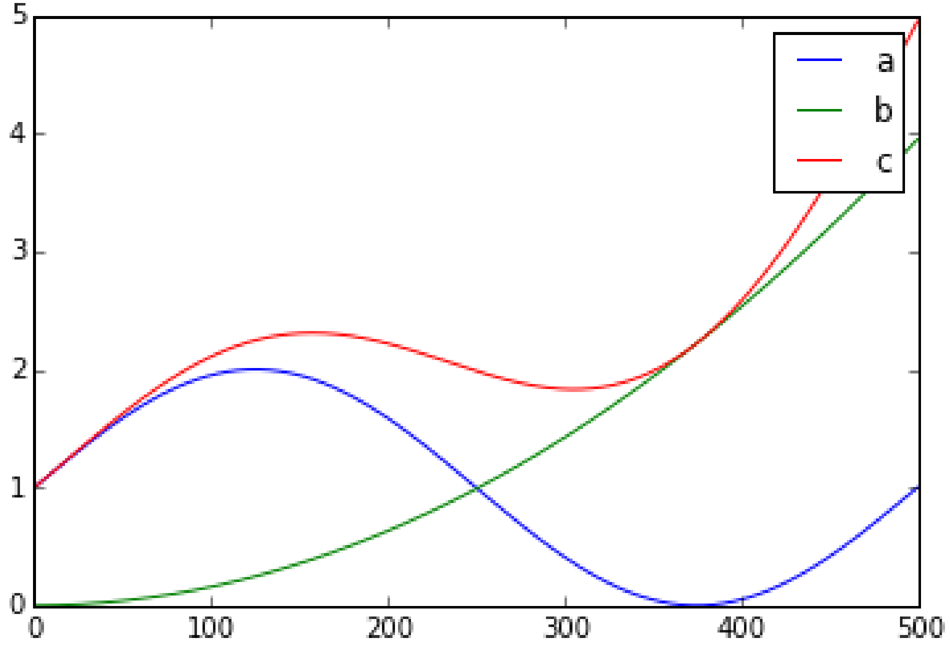
- Check result

```
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

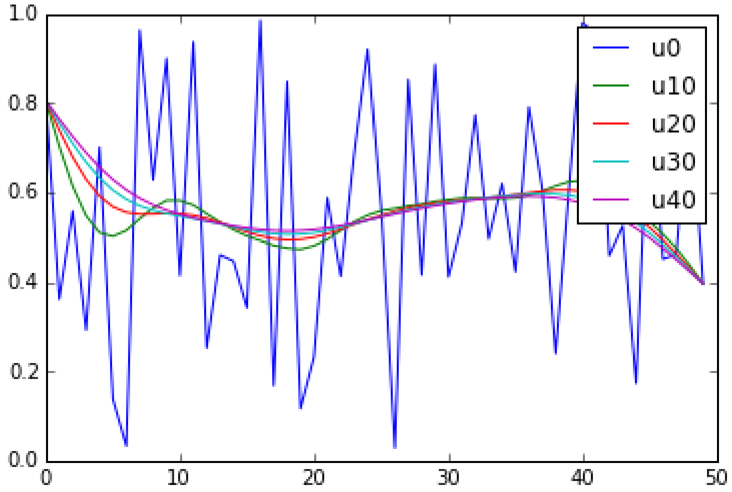
```
#CPU data
u0 = np.random.rand(50).astype(np.float32)

#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

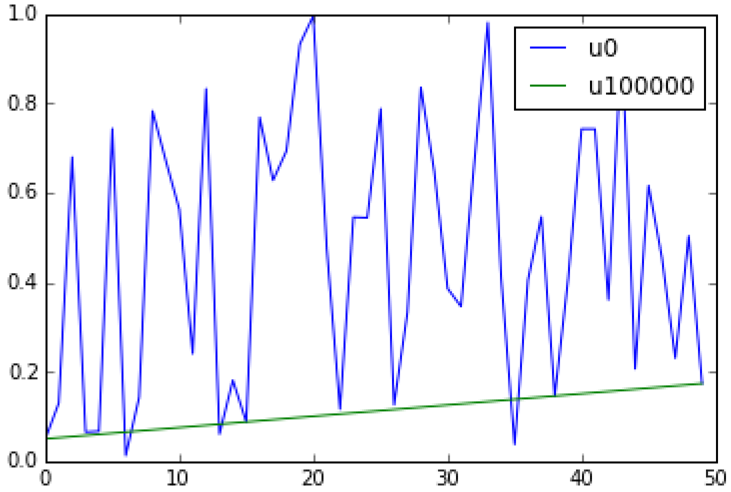
```
#Set number of timesteps
nt = 50

#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

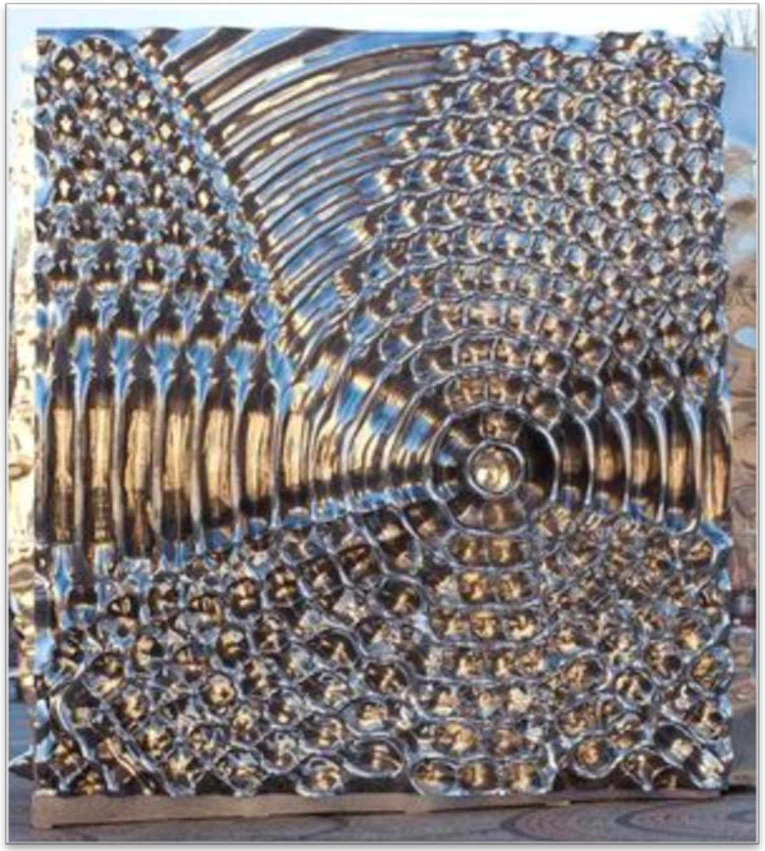
- In two dimensions, the heat equation can be written

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

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

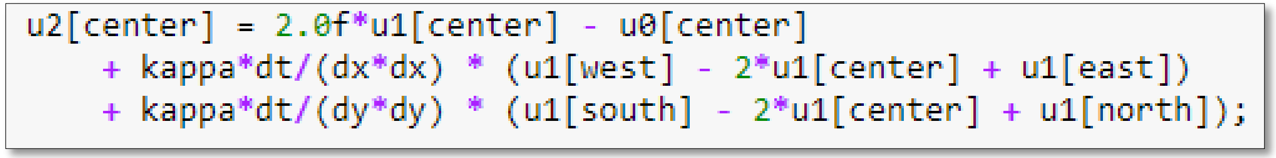
- For the code, we have to now solve in 2 dimensions, not only one!

Example: The 2D wave equation



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

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



```
u2[center] = 2.0f*u1[center] - u0[center]
    + kappa*dt/(dx*dx) * (u1[west] - 2*u1[center] + u1[east])
    + kappa*dt/(dy*dy) * (u1[south] - 2*u1[center] + u1[north]);
```

SINTEF

Høy C, ©Bård Breivik/BONO. Foto: Terje Heiestad. UiO

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

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

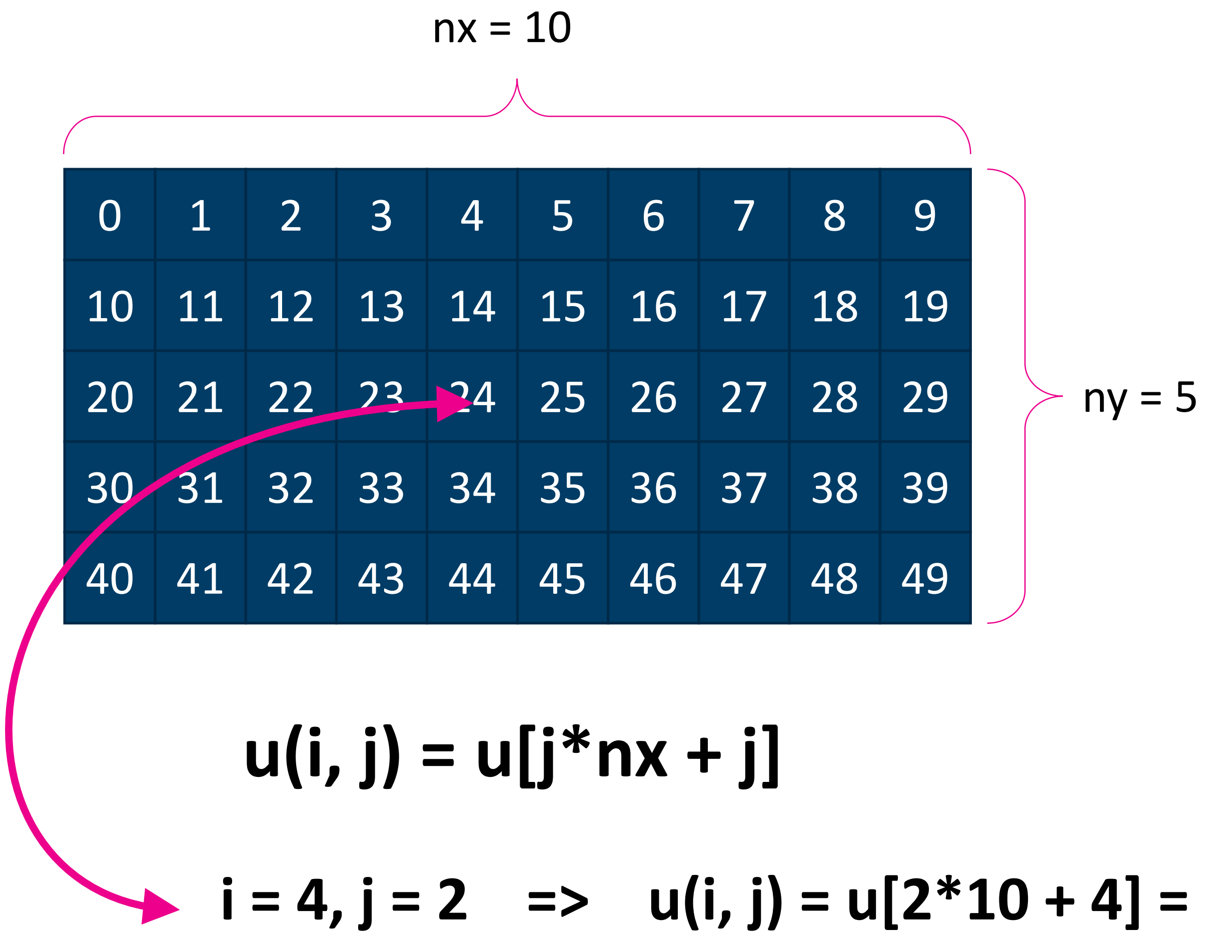
- And end up equivalently with

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

- 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



nx = 10

ny = 5

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

$$u(i, j) = u[j*nx + j]$$

$i = 4, j = 2$ => $u(i, j) = u[2*10 + 4] = u[24]$

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

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

- 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

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

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 or download from https://goo.gl/mssYsT
  - The username / password is opencl

- 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