Prototyping OpenCL with Python

1st Meetup Khronos Oslo Chapter

André R. Brodtkorb, Research Scientist, SINTEF ICT
Andre.Brodtkorb@sintef.no  http://babrodk.at.ifi.uio.no/
Important questions

- Who am I?
- Why am I here?
- Who are all of you guys???
Outline

- What I promised:

“OpenCL is the open standard for parallel programming of heterogeneous systems, André R. Brodtkorb will show how one can use Python and PyOpenCL to solve Partial Differential Equations.

Utilizing Python’s rapid development loop to prototype efficient OpenCL kernels.

André R. Brodtkorb is a research scientist in the Department of Applied Mathematics at SINTEF, a non-profit research organization in Norway with roughly 2000 researchers. His research interests include numerical simulation, accelerated scientific computing, image processing, and real-time scientific visualization.”

Oh and btw: There will be free pizza :)}
Outline – what I will actually go through

• Some motivation to use OpenCL

• Some introduction to the mathematics

• A live demo (but I'll have to cheat a bit to fit the schedule)

• Summary of my experiences teaching and using PyOpenCL
History lesson: development of the microprocessor 1/2

1942: Digital Electric Computer
(Atanasoff and Berry)

1947: Transistor
(Shockley, Bardeen, and Brattain)

1958: Integrated Circuit
(Kilby)

1971: Microprocessor
(Hoff, Faggin, Mazor)

1971- Exponential growth
(Moore, 1965)

1947- Exponential growth
(Moore, 1965)
History lesson: development of the microprocessor 2/2

1971: 4004, 2300 trans, 740 KHz

1982: 80286, 134 thousand trans, 8 MHz

1993: Pentium P5, 1.18 mill. trans, 66 MHz

2000: Pentium 4, 42 mill. trans, 1.5 GHz

2010: Nehalem, 2.3 bill. Trans, 8 cores, 2.66 GHz
End of frequency scaling

1970-2004: Frequency doubles every 34 months (Moore’s law for performance)
1999-2014: Parallelism doubles every ~30 months
What happened in 2004?

- Heat density approaching that of nuclear reactor core: **Power wall**
  - Traditional cooling solutions (heat sink + fan) insufficient
  - Industry solution: multi-core and parallelism!

Original graph by G. Taylor, “Energy Efficient Circuit Design and the Future of Power Delivery” EPEPS’09
Why Parallelism?

The power density of microprocessors is proportional to the clock frequency cubed:\(^1\)

\[ P_d \propto f^3 \]

---

1 Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010
Massive Parallelism: The Graphics Processing Unit

- 5-10 times as power efficient as CPUs!
- 7-10 times as powerful as CPUs
Why care about computer hardware?

- The key to performance, is to consider the full algorithm and architecture interaction.
- A good knowledge of both the algorithm and the computer architecture is required.

Graph from David Keyes, Scientific Discovery through Advanced Computing, Geilo Winter School, 2008
Examples of Early GPU Research at SINTEF

- Registration of medical data (~20x)
- Preparation for FEM (~5x)
- Self-intersection (~10x)
- Fluid dynamics and FSI (Navier-Stokes)
- Inpainting (~400x matlab code)
- Euler Equations (~25x)
- SW Equations (~25x)
- Marine aqoustics (~20x)
- Water injection in a fluvial reservoir (20x)

Matlab Interface

Linear algebra
Programming GPUs today

<table>
<thead>
<tr>
<th>Year</th>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>OpenGL</td>
<td>1st gen: Graphics APIs</td>
</tr>
<tr>
<td>2005</td>
<td>DirectX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DirectCompute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BrookGP U</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Embedded Meta-programming Language</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Brook+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMD CTM / CAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OpenCL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMD Brook+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAPIDMIND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OpenACC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NVIDIA CUDA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PGI Accelerator</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>C++ AMP</td>
<td>2nd gen: (Academic) Abstractions</td>
</tr>
<tr>
<td>2015</td>
<td>NVIDIA CUDA</td>
<td>3rd gen: C- and pragma-based languages</td>
</tr>
</tbody>
</table>

1st gen: Graphics APIs
2nd gen: (Academic) Abstractions
3rd gen: C- and pragma-based languages
OpenCL

• OpenCL is much like CUDA, but slightly more cumbersome to work with.
  • The benefit is that the same code can run on Intel CPUs, the Xeon Phi, NVIDIA GPUs, AMD GPUs, etc.

• The amount of code needed to do the exact same thing is larger in OpenCL
  • OpenCL is a C API
  • CUDA has C++ bindings, and supports templates

• CUDA has a much better development community for NVIDIA

• OpenCL has a lot of very nice tools available from AMD
Trivial example: OpenCL matrix addition

```c
__kernel void addMatricesKernel(__global float* c, __global float* a,
     __global float* b, unsigned int cols, unsigned int rows) {
    //Indexing calculations
    unsigned int global_x = get_global_id(0);
    unsigned int global_y = get_global_id(1);
    unsigned int k = global_y*cols + global_x;

    //Actual addition
    c[k] = a[k] + b[k];
}

void addFunctionOpenCL() {
    ... //More code here: Allocate data on GPU, copy CPU data to GPU
    //Set arguments
    clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&gpu_c);
    clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&gpu_a);
    clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&gpu_b);
    clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&cols);
    clSetKernelArg(ckKernel, 4, sizeof(cl_int), (void*)&rows);
    // Launch kernel
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &gws, &lws, 0, NULL, NULL);
    ... //More code here: Download result from GPU to CPU
}
```
iPython and Pyopencl

- OpenCL is a C API, which requires working in C, and possibly long compilation times.
- Even the simplest OpenCL example will require a lot of boilerplate code.
- Pyopencl solves this, by enabling access to OpenCL through Python.
- Ipython notebook gives us an interactive shell to try out OpenCL and prototype!
The caveat

- Installing can be slightly cumbersome
  - You need an OpenCL driver, which complicates things

- If you're on Ubuntu and have an NVIDIA graphics card, you're lucky
  - `sudo apt-get install ipython ipython-notebook python-pyopencl` and you're good to go

- If you're on any other platform or hardware, you need to do a bit of manual labour
  - But it still shouldn't take longer than half an hour.
Hello World PyOpenCL – add two vectors

```c
%%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];
}
```

```python
#Upload data to the device, allocate output data
...

#Execute program on device
add_kernel(cl_queue, a.shape, None, a_g, b_g, c_g)

#Allocate data on the host for result
c = np.empty_like(a)

#Download data from device to host
cl.enqueue_copy(cl_queue, c, c_g)
```
More mathematics: 5 minute crash course to PDEs
Conservation Laws

• A conservation law describes that a quantity is conserved
  • Comes from the physical laws of nature

• Example: Newton’s 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: Newton’s 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.

• Conservation laws can be mathematically formulated as partial differential equations: PDEs
Ordinary Differential Equations (ODEs)

• Let us look at Newton's second law
  • The vector sum of the external forces $\vec{F}$ on an object is equal to the mass $m$ of that object multiplied by the acceleration vector $\vec{a}$ 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 Newton's 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
    - $\vec{v}(t) = \vec{v}_0 + t \cdot \vec{a}$
  - Change in position, $p$, over time is a function of the velocity
    - $\frac{dp}{dt} = \vec{v}(t)$
  - We can solve this ODE analytically with pen and paper, but for more complex ODEs, that becomes infeasible
Example of a simple ODE

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

• We replace the continuous derivatives with discrete equivalents

• 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, \(\Delta t\) 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!

<table>
<thead>
<tr>
<th>t</th>
<th>p</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>( p_0 + dt \cdot v_0 = 0.0 )</td>
<td>( v_0 - dt \cdot 9.81 = -0.981 )</td>
</tr>
<tr>
<td>0.2</td>
<td>( p_1 - dt \cdot v_1 = -0.0981 )</td>
<td>( v_1 - dt \cdot 9.81 = -1.962 )</td>
</tr>
<tr>
<td>0.2</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Demo time
Projectile trajectory IPython 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
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

\[ \text{dt}=1 \quad \text{dt}=0.5 \quad \text{dt}=0.25 \]
More advanced problems

• Tracing a trivial projectile trajectory is not computationally challenging (by today's standards)

• Let's look at more advanced problems:
  • Linear waves (acoustic waves)
  • Shallow water waves (tsunami waves)
  • Gas dynamics (explosions)
Linear Wave Equation

- The linear wave equation can be written
  \[
  \frac{\partial^2 u}{\partial t^2} = c \nabla^2 u = c \left[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right]
  \]

- Here c is the wave propagation speed coefficient

- We can write a numerical scheme that approximates a solution 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

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

- In other words, if we have two initial timesteps, we can solve forward in time
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
__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];
  }
}

Internal cells

__kernel void linear_wave_2D(__global float* u2, global const float* u1, __global const float* u0, float kappa, float dt, float dx, float dy) {
    ....

    //Get position in grid
    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 = (j+1)*nx + i;     int south = (j-1)*nx + i; ... 

    //Process Internal cells
    if (i > 0 && i < nx-1 && j > 0 && j < ny-1) {
        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]);
    }
}
Demo time
Summary PyOpenCL

- iPython is a game-changer in terms of usability
  - PyOpenCL couples high-level Python with low-level OpenCL
  - Some difficulty with debugging and compilation errors
  - Students become productive within a couple of hours
  - Python plotting at your fingertips

- For maximum performance, low-level (C++) inner loop is still preferable

- For prototyping kernels and checking out different approaches, Python is king
Thank you for your attention!

Contact: Andre.Brodtkorb@sintef.no
Webpage: http://babrodtk.at.ifi.uio.no/