High-performance computing using Python, GPUs & Modular approach to crystallographic software development

Vincent Favre-Nicolin
X-ray NanoProbe group, ESRF
OUTLINE OF MY TALK

• Modular, object-oriented programming
• Efficient programming in Python
• GPU computing from Python
• A few words about Cloud computing

```python
class CDI:
    """Reconstruction class for two or three dimensional projections.
    """
    def __init__(self, iobs, support, mask, lambda0, obj0, lambda1, pixel_size_object, pixel_size_detector):
        """Constructor. All arrays are assumed to be correct."
        self.iobs = iobs.astype(np.float32)

        if obj is None:
            self._obj = (np.random.uniform(0, 1, iobs.shape) * np.exp(1) * np.random.uniform(0, 2 * np.pi, iobs.shape)) * self._support.astype(np.complex64)
        else:
            if copy_obj is True:
                self._obj = obj.copy().astype(np.complex64)
            else:
                self._obj = obj.astype(np.complex64)
```
**MAIN COMPUTING PROJECTS**

- **Ab initio structure solution from powder diffraction:**
  FOX: [http://fox.vincefn.net](http://fox.vincefn.net)

- **Coherent diffraction Imaging using GPU**
  PyNX:
  - [https://software.pan-data.eu/software/102/pynx](https://software.pan-data.eu/software/102/pynx)
import numpy as np

# Linear programming
l, k, h = np.mgrid[-10:11, -10:11, -10:11]
n_atoms = 1000
x = np.random.uniform(0, 1, size=n_atoms)
y = np.random.uniform(0, 1, size=n_atoms)
z = np.random.uniform(0, 1, size=n_atoms)

fhkl = np.empty(h.shape, dtype=np.complex)
for i in range(h.size):
    fhkl[i] = np.exp(2j * np.pi * (h.flat[i] * x + k.flat[i] * y + l.flat[i] * z)).sum()
# Function

def calc_f_hkl(x, y, z, h, k, l):
    assert x.shape == y.shape == z.shape
    fhkl = np.empty(h.shape, dtype=np.complex)
    for i in range(h.size):
        fhkl[i] = np.exp(2j * np.pi * (h.flat[i] * x + k.flat[i] * y + l.flat[i] * z)).sum()
    return fhkl

fhkl = calc_f_hkl(x, y, z, h, k, l)

You can:
- Re-use the function
- Optimize it
class reciprocal_space:
    def __init__(self, h, k, l):
        assert h.shape == k.shape == l.shape
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()

rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == l.shape)
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None
    def get_f_hkl(self):
        return self._fhkl
    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()

rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
class reciprocal_space:
    def __init__(self, h, k, l):
        assert h.shape == k.shape == l.shape
        self.h = h
        self.k = k
        self.l = l
        self._f_hkl = None

    def get_f_hkl(self):
        return self._f_hkl

    def calc_f_hkl(self, x, y, z):
        self._f_hkl = np.empty(h.shape, dtype=np.complex)
        for i in range(h.size):
            self._f_hkl[i] = np.exp(2j * np.pi * (h.flat[i] * x + k.flat[i] * y + l.flat[i] * z)).sum()

rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()

---

**WHY OBJECT-ORIENTED**

- **Separate:**
  - Code (calculations)
  - Data (storage)
- **Abstract layer:**
  - No direct access to data
  - Methods must keep their signature (API)
  - Calculation/storage can be later optimized:
    - Limited or full-space hkl (FFT vs direct sum)
    - CPU or GPU?
- **Better code organization**
- **Easy to document**
- **Easy to share**
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == l.shape)
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x + self.k.flat[i] * y + self.l.flat[i] * z)).sum()

rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()
class reciprocal_space:
    def __init__(self, h, k, l):
        assert (h.shape == k.shape == l.shape)
        self.h = h
        self.k = k
        self.l = l
        self._fhkl = None

    def get_f_hkl(self):
        return self._fhkl

    def calc_f_hkl(self, x, y, z):
        self._fhkl = np.empty(h.shape, dtype=np.complex)
        for i in range(self.h.size):
            self._fhkl[i] = np.exp(2j * np.pi * (self.h.flat[i] * x +
                                             self.k.flat[i] * y +
                                             self.l.flat[i] * z)).sum()

rec = reciprocal_space(h, k, l)
rec.calc_f_hkl(x, y, z)
fhkl = rec.get_f_hkl()  # YES
fhkl = _fhkl             # NO- may trigger the end of the world

• All attributes can be accessed in Python
• Members with a leading _ are a programmer’s hint that the method or data may be changed in the future
• Use public access methods
• Use _ for data/methods which may change
For any crystallographic computing, you can choose:

• Large classes with many methods
• A large number of small classes easily re-used
EXAMPLE PROBLEM: STRUCTURE SOLUTION

- Sample → Diffraction Data → Unit Cell, Spacegroup(s) → Structure Solution → Structure Refinement

Get atomic positions within 0.1-0.5 Å

- Least-Squares

Global Optimization in real space

- Modeling by atoms or atom groups
  - (more or less) random search in direct space from the model degrees of freedom

- Grid Search
- Monte-Carlo
- Genetic Algorithms

Reciprocal-space Methods

- Extracted |F(hkl)| → Phases
  - Fourier recycling
    - Structural model
    - Electronic density

! Need high-quality |F(hkl)|!
Parametrization
-> Degrees Of Freedom
(atomic positions, torsion angles, ..)

starting configuration → random change of parameters → evaluation of the new configuration: Cost (C)

keep configuration yes

Keep configuration with probability:

\[ P = e^{-\frac{\Delta C}{T}} \]

no

is configuration better? \( C_n < C_{n-1} \)

Hypersurface
Cost = f (DOF)

Generate a distribution of configurations following Boltzmann's law

Temperature of the algorithm

Need two classes:
• Object to optimize
• Algorithm
APPRAOCH 1: BIG CLASSES / OBJCRYST++

- ScatteringPower
  - scattering factor
  - anomalous scattering factor
  - temperature factor

- RefinableObj
  - List of parameters
  - Cost: log(likelihood)
  - Random moves
  - Derivatives

- Scatterer
  - (list of) position(s)
  - scattering power(s)
  - specific moves

- Crystal
  - unit cell
  - spacegroup
  - list of scatterers

- ScatteringPowerAtom

- OptimizationObj

- Monte-Carlo
  - Simulated Annealing
  - Parallel Tempering

- Least squares

- ScatteringData
  - crystal
  - reflections

- PowderPattern
  - Background
  - Crystalline Phases

- SingleCrystal

- Profiles
  - Pseudo-Voigt, TOF-

- Atom

- Molecule (restraints)
OBJCRYST++ API

Public Member Functions

- ReflectionProfile()
- ReflectionProfile(const ReflectionProfile&)
- ~ReflectionProfile()
- virtual ReflectionProfile*
- CreateCopy() const=0
- virtual CrystVector_REAL GetProfile(const CrystVector&)
- Get the reflection profile.
- virtual REAL GetFullProfileWidth(const ReflectionProfile&)
- Get the (approximate) full profile width.

See API from:
http://fox.vincefn.net
- Molecule
- UnitCell
- LeastSquares
OBJCRYST++ API: AVOID RE-COMPUTATIONS

Powder pattern calculation:
- Did crystal structure change?
  - No: re-use structure factors
  - Yes:
    - Re-compute structure factors
    - Re-use table of atomic scattering factors
- Did unit cell change?
  - No: keep reflection positions
  - Yes: re-compute reflection positions
- Did reflection profiles change?
  - No: re-use reflection profiles
  - Yes: re-compute reflection profiles

Note:
- top objects know nothing about derived objects implementation – all is hidden behind OO API
- Optimization algorithms know nothing about crystallography!
FOX/OBJCRYST++ SPEED

- 20 independent atoms, 100 reflections
- $10^4$ to $5.10^4$ configurations/s

Drawback of the ‘big object’ approach: the library is optimized for algorithms needing large number of trials/s
class ERProj(CLOperatorCDI):
    """
    Error reduction.
    """

    def __init__(self, positivity=False):
        super(ERProj, self).__init__()
        self.positivity = positivity

    def op(self, cdi):
        if self.positivity:
            self.processing_unit.cl_er_real(cdi._cl_obj, cdi._cl_support)
        else:
            self.processing_unit.cl_er(cdi._cl_obj, cdi._cl_support)
        return cdi

class ER(CLOperatorCDI):
    """
    Error reduction cycle
    """

    def __new__(cls, positivity=False, calc_llk=False):
        return ERProj(positivity=positivity) * FourierApplyAmplitude(calc_llk=calc_llk)

Idea:

decompose the computing problem
in as many
independent snippets as possible
COHERENT DIFFRACTION IMAGING

- Illuminate a single crystal
- Scattering on detector is the Fourier Transform of the Crystal’s shape
- The phase of the oscillations is lost!

Nature 470, 73 (2011)
The algorithms can only converge if the problem is **over-determined**

→ need more than 1 point per fringe  
   “oversampling”

**Density modification:**
- Positivity (*)
- Finite support
ER: support projection

= set electronic density to zero outside a defined support
All operations on can be described as mathematical operators:

\[ A_{\text{calc}}(i+1) = F A_{\text{proj}} \times FT^{-1} \times ER_{\text{proj}} \times FT \times A_{\text{calc}}(i) \]
TABLE I. Summary of various algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iteration $\rho^{(n+1)} = $</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER</td>
<td>$P_m P_{m\beta}^{(n)}$</td>
</tr>
<tr>
<td>SF</td>
<td>$R_s P_{m\beta}^{(n)}$</td>
</tr>
</tbody>
</table>
| HIO       | \[
| r \in S   | \frac{P_m \rho^{(n)}(r)}{(I - \beta P_m) \rho^{(n)}(r)} |
| S         | \]                           |
| DM        | \[
|          | \frac{1}{2} \left[ R_s R_m + I \right] \rho^{(n)} |
| ASR       | $\frac{1}{2} \left[ R_s (R_m + (\beta - 1) P_m) + I + (1 - \beta) P_m \right] \rho^{(n)}$ |
| HPR       | $\frac{1}{2} \left[ R_s (R_m + (\beta - 1) P_m) + I + (1 - \beta) P_m \right] \rho^{(n)}$ |
| RAAR      | $\frac{1}{2} \beta (R_s R_m + I) + (1 - \beta) P_m \rho^{(n)}$ |

$P_m$:
- Fourier transform the object
- Impose magnitude in Fourier space from observed intensity
- Back-Fourier Transform

$P_s$:
- Replace density by zero outside of support

class Operator:
    """
    Base class for an operator, applying e.g. to a wavefront object.
    """

    def __init__(self):
        self.ops = [self]

    def __mul__(self, w):
        """
        Applies the operator to an object.
        """
        if isinstance(w, Operator):
            self.ops += w.ops
            return self
        return_ops_mul(w)

        def __mul__(self, w):
            """
            Applies the operator to an object.
            """
            if isinstance(w, Operator):
                self.ops += w.ops
                return self
            return_ops_mul(w)

            def apply_ops_mul(self, w):
                """
                To apply multiple operators:
                A = Op3 * Op2 * Op1 * A
                """
                return w

            Overload the __mul__ method:
            A = Op * A
            Is the same as A=Op.__mul__(A)
class FourierApplyAmplitude(CLOperatorCDI):
    """
    Fourier magnitude operator, performing a Fourier transform, the magnitude projection, and a backward FT.
    """
    def __new__(cls, calc_llk=False):
        return IFT() * ApplyAmplitude(calc_llk=calc_llk) * FT()

class ERProj(CLOperatorCDI):
    """""""Error reduction projection""""
    def op(self, cdi):
        self.processing_unit.cl_er(cdi._cl_obj, cdi._cl_support)
        return cdi

class ER(CLOperatorCDI):
    """""""Error reduction cycle""""
    def __new__(cls, positivity=False, calc_llk=False):
        return ERProj(positivity=positivity) * FourierApplyAmplitude(calc_llk=calc_llk)
Also overload the `__pow__` operator

Example full CDI reconstruction algorithm:

\[
\text{cdi} = \text{ER()}^{100} \times (\text{SupportUpdate()} \times \text{ER()}^{50} \times \text{HIO()}^{200})^{5} \times \text{cdi}
\]

- Shrinks the support area
- Hybrid Input-Output
  Linear combination between current and previous crystal model (outside support)
BIG CLASSES VS TOOLKIT APPROACH

Big classes allow fine-tuning for specific applications
... allow more optimizations
... but are not very flexible

Small classes are much easier to re-use (cctbx)
... but can lead to a very large number of classes / function

```python
class FourierApplyAmplitude(CLOperatorCDI):
    
    """
    Fourier magnitude operator, performing a Fourier transform, the magnitude projection, and a backward FT.
    """

def __new__(cls, calc_llk=False):
    return IFT() * ApplyAmplitude(calc_llk=calc_llk) * FT()
```
class Operator:
    
    /*
    * Base class for an operator, applying e.g. to a wavefront object
    */
    def __mul__(self, w):
        
        /*
        * Applies the operator to an object
        */
        self.apply_ops_mul(w)
        return w

/** \brief Unit Cell class: Unit cell with */
class UnitCell {
   
   public:
       // Default Constructor
       UnitCell();
       /** \brief UnitCell Constructor (triclinic)
       *  \param a,b,c : unit cell dimension, in angstroems
       *  \param alpha,beta,gamma : unit cell angles, in radians.
       *  \param SpaceGroupId: space group symbol or number
       */
       UnitCell(const REAL a, const REAL b, const REAL c, const REAL alpha, const REAL beta, const REAL gamma, const string &SpaceGroupId);
**DOCUMENTATION: DOXYGEN/C++**

Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitCell ()</td>
<td>Default Constructor.</td>
</tr>
<tr>
<td>UnitCell (const REAL a, const REAL b, const REAL c)</td>
<td>Constructor (orthorombic) More...</td>
</tr>
<tr>
<td>UnitCell (const REAL a, const REAL b, const REAL c)</td>
<td>Constructor (triclinic) More...</td>
</tr>
</tbody>
</table>

Member Data Documentation

**CrystMatrix REAL ObjCryst::UnitCell::mBMatrix**

-B Matrix (Orthogonalization matrix for reciprocal space)-

\[
B = \begin{bmatrix}
  a^* & b^* \cos(\gamma^*) & c^* \cos(\beta^*) \\
  0 & b^* \sin(\gamma^*) & -c^* \sin(\beta^*) \cos(\alpha) \\
  0 & 0 & \frac{1}{c}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  k_x \\
  k_y \\
  k_z
\end{bmatrix}
\text{orthonormal} = B \times \begin{bmatrix}
  h \\
  k \\
  l
\end{bmatrix}
\text{integer}
\]

http://doxygen.org
Doxygen is a must-have tool for C++ documentation:

- Automatic documentation
- Html, pdf
- Dependencies, hierarchy, graphs

http://doxygen.org
PyNX: Python tools for Nano-structures Xtallography

Introduction

PyNX stands for Python tools for Nano-structures Xtallography.

1. `pynx.scattering`: X-ray scattering data and analysis (single nVidia Titan X) computer
2. `pynx.pycbo`: simulation of the behavior of OpenCL. Examples are provided as well as using or producing
3. `pynx.xraywavefront`: X-ray wavefronts
4. `pynx.cdi`: Coherent Data Information

In addition, it includes scripts and functions with documentation (pynx-id01pty.py, pynx-cgi.py)

Download

PyNX is available from:
- http://gitlab.esrf.fr/favre/PyNX (login required, site registration is open & free)
- All modules except `pynx.pycbo` (see http://ftp.esrf.fr/pub/scisoft/PyNX/README.txt) are also available:
  - PyPI (pip install pynx)
  - http://pynx.sf.net

Citation & Bibliography

- Also from inline documentation
- requires more specific documentation writing
- Plug-ins
CONTROL VERSION SYSTEM: GIT

- Full development history
- Branching, merging
- De-centralized (every copy is a full copy)
- Collaborative development
- Accountability
- Command-line or GUI
- Also useful for articles

https://git-scm.com
UNIT TESTING

ValueError
/Users/favre/dev/pynx-private/pynx-
19 if __name__ == '__main__':
 20 try:
 21 w = CDIRunnerID10(sys.argv, params, CDIRunnerScanID10)
 22 w.process_scans()
 23 except CDIRunnerException:

import unittest
class TestCrystalStructureFactor(unittest.TestCase):
  def test_centric(self):
    cryst = Crystal(a=4,b=5,c=6, spacegroup='P-1')
cryst.add_random_atoms(nb=10)
l, k, h = np.mgrid[-10:11, -10:11, -10:11]
self.assertTrue(np.allclose(cryst.get_structure_factor(h=h,k=k,l=l), 1e-6)

Source of errors:
- New code bugs
- MacOS, Linux(es), Windows
- Python 2.7, 3.3…3.6
- Different hardware
- …

Use ‘unit tests’ which can be run automatically
AUTOMATED TESTING

ValueError
/Users/favre/dev/pynx-private

19  if __name__ == '__main__':
20      try:
21          w = CDIRunnerID10 ...
22          w.process_scans()
23      except CDIRunnerException as ex:

import unittest
import numpy as np

class TestCrystalStructureFactor(unittest.TestCase):
    def test_centric(self):
        cryst = Crystal(a=4,b=5,c=6, spacegroup='P1-1')
        cryst.add_random_atoms(nb=10)
        l, k, h = np.mgrid[-10:11, -10:11, -10:11]
        self.assertTrue(np.allclose(cryst.get_structure_factor(h=h, k=k, l=l), 1e-6))

Ideally:
• Automatic testing every time new code is pushed to a server
• Always test against different platforms, libraries version…
• Lots of unit tests
• Continuous integration
Which languages are fast?

- Compiled vs interpreted vs just-in-time
- Development vs execution speed
- Command-line interpreter
import timeit
import numpy as np
nb=100000

# With a python loop
nbiter=10
a=np.arange(nb)  # loop
b=np.arange(nb)
c=np.arange(nb)
t1=timeit.default_timer()
for i in range(nbiter):
    for i in range(nb):
        c[i]=a[i]+b[i]

    t2=timeit.default_timer()
    mflops=(nb*nbiter)/(t2-t1)/1e6
print("Boucle : %6.3f Mflops"%(mflops))

2.7 Mflop/s (on this laptop, 2.5GHz Intel i7)
AVOIDING LOOPS: NUMPY

- Numpy is optimized, so use operations on vectors!
- *Never* use a loop on individual values!

All lengthy operations should be delegated to python libraries

```python
import timeit
import numpy as np
nb=100000

# Using numpy operations
nbiter=1000
a=np.arange(nb) # loop
b=np.arange(nb)
c=np.arange(nb)
t1=timeit.default_timer()
for i in range(nbiter):
    c=a+b

t2=timeit.default_timer()
mflops=(nb*nbiter)/(t2-t1)/1e6
print("Speed : %6.3f Mflops"%(mflops))

1.3 Gflop/s (on this laptop, 2.5GHz Intel i7)
Speedup x500
```
import numpy as np

nb = 1000
a = np.random.uniform(0, 1, size=(nb, nb))

# operation on a sub-array

# Count values > 0.7
(a > 0.7).sum()

# Extract values > 0.7 (result is flattened)
d = a[a > 0.7]

# double values > 0.7
a[a > 0.7] *= 2  # version 1
a += a * (a > 0.7)  # version 2

• Use operations on vectors
• Also for:
  • sub-arrays
  • conditional access

Only random walks can hardly be vectorized
python -m cprofile -o log run.py

Use the profiler from python: cProfile

Visualization:
- Either using the pstats module
- Or: visualize with runsnake

Or simply use %timeit (within ipython/jupyter)
def fib(n):
    """Print the Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a + b

from distutils.core import setup
from Cython.Build import cythonize
setup( ext_modules=cythonize("fib.pyx"), )

$ python setup.py build_ext --inplace

import fib
fib.fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
In [1]: # TODO: make pynx-cxity.py lasses importable instead of just embedded in the script
def cxity import PtychoRunnerScanCXI
def cxity import params_generic as params
pylab.rcParams['figure.figsize'] = (12, 8)
print('Import OK')

In [2]:
params['cxifile'] = 'data/data.cxi'
params['probe'] = 'focus,50e-6x200e=6,0.09'
params['gpa'] = 'K80'
params['algorithm'] = '20DM'
params['object'] = 'random,0.8,2,0,0.5'
params['verbose'] = 5
params['liveplot'] = True

In [5]: ws = PtychoRunnerScanCXI(params, 0)
In [5]: ws.load_data() # Load 1025 frames from a maxipix detector using CXI/HDF5 data
In [5]: ws.prepare()
In [5]: ws.run()
In [5]: ws.run_algorithm('20AF')
In [5]: ws.run_algorithm('20ML')
In [5]: ws.run_algorithm('nbprobe=3,20AP')
In [5]: ws.run_algorithm('20ML')
Python #1 Mistakes: Copy by Reference

Python defaults to copy-by-reference
- Modifying a shallow copy of an object also changes the original object
- This saves memory!
- Memory is deleted after last referencing object is deleted (garbage collection)

Whenever a real copy is needed:
- Use the ‘copy.deepcopy’ function
- For numpy array: a = b.copy()

```
In [1]: import numpy as np
In [2]: a=np.arange(8)
In [3]: b=a
In [4]: print(a)
[0 1 2 3 4 5 6 7]
In [5]: b[5]=90
In [6]: print(a)
[0 1 2 3 4 90 6 7]
```
Consumer-grade Graphical Processing Units can yield > 1 Gflop/s

http://docs.nvidia.com/cuda/cuda-c-programming-guide/
GPU COMPUTING ARCHITECTURE

CPU use a few computing cores, with a general-purpose instruction set

GPU are optimized for > $10^4$ parallel threads, with optimized instructions (fast exp, trigonometric functions, ...)

Ex. GPU: 768 active threads/multiprocessor, with 30 multiprocessors

All GPU threads must execute the same code (on different data)

http://docs.nvidia.com/cuda/cuda-c-programming-guide/
Replace loops with functions (a **kernel** ) executing at each point in a problem domain

E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

**Traditional loops**

```c
void mul(const int n,
         const float *a,
         const float *b,
         float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

**Data Parallel OpenCL**

```c
__kernel void mul(__global const float *a,
                 __global const float *b,
                 __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```

// many instances of the kernel,
// called **work-items**, execute
// in parallel

Slide from https://github.com/HandsOnOpenCL/Lecture-Slides
GPU EFFICIENCY & MEMORY

- Computing is cheap (fast)
- Memory access is expensive
- (Avoid tests)
- Test against different GPUs

https://github.com/HandsOnOpenCL/Lecture-Slides
\begin{verbatim}
__kernel __attribute__((reqd_work_group_size(%(block_size)d, 1, 1)))
void Fhkl(__global float *fhkl_real, __global float *fhkl_imag,
         __global float *vx, __global float *vy, __global float *vz, const long natoms,
         __global float *vh, __global float *vk, __global float *vl)
{
    #define BLOCKSIZE %(_block_size)d
    #define twopi 6.2831853071795862f

    // Block index
    int bx = get_group_id(0);
    int by = get_group_id(1);

    // Thread index
    int tx = get_local_id(0);

    const unsigned long ix = tx + (bx + by * get_num_groups(0)) * BLOCKSIZE;
    const float h = twopi * vh[ix];
    const float k = twopi * vk[ix];
    const float l = twopi * vl[ix];

    float fr = 0, fi = 0;
    __local float x[BLOCKSIZE];
    __local float y[BLOCKSIZE];
    __local float z[BLOCKSIZE];

    for(unsigned int i = 0; i < BLOCKSIZE; i++)
    {
        const float tmp = h * x[i] + k * y[i] + l * z[i];
        fi += native_sin(tmp);
        fr += native_cos(tmp);
    }

    fhkl_real[ix] += fr;
    fhkl_imag[ix] += fi;
}
\end{verbatim}

$$A(\hat{s}) = \sum_i f_i(s) e^{2\pi \hat{s} \cdot \hat{r}_i}$$
OPENCL: FAST STRUCTURE FACTOR

- Up to $2.5 \times 10^{11}$ refl. atoms/s
- 8 flop per atom-refl pair
  => 2 Tflop/s

1 nVidia Titan X card
(1300 EUR, 250W)
1) Electronic density within 1 unit cell on a fine grid
2) FFT

F(hkl) computation

Direct calculation using atomic positions + scattering factors

\[ A(\mathbf{s}) = \sum f_i(s) e^{2\pi \mathbf{s} \cdot \mathbf{r}_i} \]

- FFT will always be faster than direct calculations
- .. But only for a full-Fourier space calculation and many reflections
- Efficient for partial Fourier space calculations
- Nano-structures: many atoms, scattering around a single reflection
CUDA/OpenCL require:

- Compiling GPU and C/C++
- Creating and transferring data between CPU (host) and GPU (device) memories
- Initializing a computing context and queue
- ... it's hard maintenance
import numpy as np
import pyopencl as cl

da_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

cntx = cl.create_some_context()
queue = cl.CommandQueue(cntx)

mf = cl.mem_flags
a_g = cl.Buffer(cntx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(cntx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)
res_g = cl.Buffer(cntx, mf.WRITE_ONLY, a_np.nbytes)

prg = cl.Program(cntx, ""
__kernel void sum(
    __global const float *a_g, __global const float *b_g, __global float *res_g)
{
    int gid = get_global_id(0);
    res_g[gid] = a_g[gid] + b_g[gid];
}
"").build()

prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)
import numpy as np
import pyopencl as cl
from pyopencl.elementwise import ElementwiseKernel

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
n = 10
a_np = np.random.randn(n).astype(np.float32)
b_np = np.random.randn(n).astype(np.float32)

a_g = cl.array.to_device(queue, a_np)
b_g = cl.array.to_device(queue, b_np)

lin_comb = ElementwiseKernel(ctx,
    "float k1, float *a_g, float k2, float *b_g, float *res_g",
    "res_g[i] = k1 * a_g[i] + k2 * b_g[i]",
    "lin_comb")

res_g = cl.array.empty_like(a_g)
lin_comb(2, a_g, 3, b_g, res_g)

print((res_g - (2 * a_g + 3 * b_g)).get()) # Check result
Compute a single result from GPU arrays: sum, Chi^2,…

```python
a = pyopencl.array.arange(queue, 400, dtype=numpy.float32)
b = pyopencl.array.arange(queue, 400, dtype=numpy.float32)

krnl = ReductionKernel(ctx, numpy.float32, neutral="0",
                       reduce_expr="a+b",
                       map_expr="x[i]* x[i] - y[i] *y[i]",
                       arguments="__global float *x, __global float *y")

chi2= krnl(a, b).get()
```
Elementwise, reduction kernels (and others: scan,..):  
- Hide all the memory handling complexity  
- Are simple enough that they are naturally optimised  

And most important:  
- A large number of crystallographic operations are simple vector operations: perfect candidate for GPU
• FFT supplied by clFFT (AMD) / gPyFFT
• Size with prime decomposition up to 13
• Performance up to a few 100 Gflop/s

```python
cl_psi = cla.zeros(cl_queue, (256, 256), np.complex64)
gpyfft_plan = gpyfft.FFT(cl_ctx, cl_queue, cl_psi, None)
for ev in gpyfft_plan.enqueue(forward=True): ev.wait()
for ev in gpyfft_plan.enqueue(forward=False): ev.wait()
```
In Ptychography

1025 frames of 400x400 pixels
0.24 s for one cycle with:
• Forward & backward 2D FFT
• ~2 elementwise kernels
GPU COMPUTING EFFICIENCY

- Computing is cheap, memory access expensive
- *Chain* all calculations on the GPU (pipeline)
- GPU calculations are *asynchronous*: python execution continues before GPU commands are done. So the next GPU command can be prepared in the GPU queue
- ‘Crystallography on-a-chip’ approach

- Amdahl’s rule: if you can optimize only fraction f of the execution time, the maximum speedup is $1/f$
- Faster is *not* always better
CLOUD COMPUTING

Computing trends:
- Less desktop computers
- More laptops
- Tablets
- SmartPhones

In institutes:
- Clusters
- CPU and GPU
- Updates complicated (software, hardware)

Cloud computing is the future (and present)

- On-demand availability:
  - Start machine in minutes
  - Start clusters
- Numerous configurations:
  - CPU, GPU
  - Memory
  - Storage latency
- Virtual machines:
  - Create one image per application
  - Stop distributing software / code!
  - Distribute images
### AMAZON EC2 (ELASTIC COMPUTING)

<table>
<thead>
<tr>
<th>vCPU</th>
<th>Memory (GiB)</th>
<th>Instance Storage (GB)</th>
<th>Linux/UNIX Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Optimized - Current Generation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.large</td>
<td>8</td>
<td>3.75</td>
<td>EBS Only</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>16</td>
<td>7.5</td>
<td>EBS Only</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>31</td>
<td>15</td>
<td>EBS Only</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>62</td>
<td>30</td>
<td>EBS Only</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>132</td>
<td>60</td>
<td>EBS Only</td>
</tr>
<tr>
<td>c3.large</td>
<td>7</td>
<td>3.75</td>
<td>2 x 16 SSD</td>
</tr>
<tr>
<td>c3.xlarge</td>
<td>14</td>
<td>7.5</td>
<td>2 x 40 SSD</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>28</td>
<td>15</td>
<td>2 x 80 SSD</td>
</tr>
<tr>
<td>c3.4xlarge</td>
<td>55</td>
<td>30</td>
<td>2 x 160 SSD</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>108</td>
<td>60</td>
<td>2 x 320 SSD</td>
</tr>
<tr>
<td><strong>GPU Instances - Current Generation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2.xlarge</td>
<td>12</td>
<td>61</td>
<td>EBS Only</td>
</tr>
<tr>
<td>p2.8xlarge</td>
<td>94</td>
<td>488</td>
<td>EBS Only</td>
</tr>
<tr>
<td>p2.16xlarge</td>
<td>188</td>
<td>732</td>
<td>EBS Only</td>
</tr>
<tr>
<td>g2.2xlarge</td>
<td>26</td>
<td>15</td>
<td>60 SSD</td>
</tr>
<tr>
<td>g2.8xlarge</td>
<td>104</td>
<td>60</td>
<td>2 x 120 SSD</td>
</tr>
<tr>
<td>g3.4xlarge</td>
<td>47</td>
<td>122</td>
<td>EBS Only</td>
</tr>
<tr>
<td>g3.8xlarge</td>
<td>94</td>
<td>244</td>
<td>EBS Only</td>
</tr>
<tr>
<td>g3.16xlarge</td>
<td>188</td>
<td>488</td>
<td>EBS Only</td>
</tr>
</tbody>
</table>
CLOUD COMPUTING FOR DATA ANALYSIS

- Synchrotron & neutron facilities
- More accessible tools
- On-demand availability
- EU initiatives:
  - Helix Nebula Science cloud
  - European Open Science Cloud
  - http://pan-data.eu
- Open Data policies: more data will be made public

https://eoscpilot.eu

http://www.helix-nebula.eu
Choose wisely the approach when programming (big/small)
Document from the start!
Python is fast if well used
Python relies on optimised libraries – use VECTOR operations
Learn to work with cloud computing and virtual machines

Things I did not talk about:
Supercomputing
MPI
Popular Licenses

The following OSI-approved licenses are popular, widely used, or have strong communities:

- Apache License 2.0
- BSD 3-Clause "New" or "Revised" license
- BSD 2-Clause "Simplified" or "FreeBSD" license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- …

http://opensource.org

- Code is ALWAYS used much longer than you’d expect
- Choose a LICENSE!
- Otherwise, it will become unusable legally
- Preferably open-source!
- Note: ’public domain’ does not exist everywhere
- Discussing with your legal department may be annoying, but it has do be done just once
For a small group

Hands-on (with laptops, python, internet):

- SQL access to Crystallography Open Database
- Global optimization algorithms
- GPU computing basics + scattering calculations
- pyObjcryst: structure solution from powder pattern, using python (experimental)
- CDI and Ptychography using GPU with PyNX
- Basics of python/jupyter notebooks

Demo: ab initio structure solution from powder diffraction / FOX