Big Data Analytics at the MPCDF: GPU Crystallography with Python

Giuseppe Di Bernardo

Max Planck Computing and Data Facility (MPCDF)

giuseppe.di-bernardo@mpcdf.mpg.de

@jose_dibernardo

July 7, 2017

europython 2017

MPCDF Max Planck Computing & Data Facility

Outline

MPCDF Max Planck Computing & Data Facility

Outline

MPCDF Max Planck Computing & Data Facility

Atom Probe Tomography (APT) Microscopy

- Chemical analysis of 3D Volumes
- Atomic scale
- No need of composition calibrations
- Composition of interfaces and defects (dislocations, clusters ...)
- Can be directly compared to 3D simulations at the atomic scale

Figure: Ga grain boundary segregation in a nanocrystalline Al thin-film

MPCDF Max Planck Computing & Data Facility

Scattering computing from an atomistic model

Theory: X-ray and neutron scattering

- Scattering vector
- Scattering density, at position $\mathbf{r}$
- Scattered amplitude

$$A(\mathbf{S}) = \int_V \rho(\mathbf{r}) \exp(2i\pi\, \mathbf{S} \cdot \mathbf{r}) dV = FT[\rho(\mathbf{r})]$$

divide et impera

- FFT is very fast: $\mathcal{O}(N \log N)$
- coordinates in reciprocal space (hkl) are imposed by the electron density grid

Our approach: *Direct calculation*

- $A(\mathbf{s}) = \sum_{j=0}^{N-1} f_j(s) \exp(2i\pi \mathbf{S} \cdot \mathbf{r_j})$
- $\mathcal{O}(N \cdot N)$

ANCK
ING &
CILITY

GPU-accelerated analysis of large X-ray data

1. Direct calculation in 3D reciprocal space (hkl): **atomic positions** ($\mathbf{r_i}$) & scattering factors (f_i):
   - $$A(\mathbf{s}) = \sum_{j=0}^{N-1} f_j(s) e^{2i\pi \mathbf{S}\cdot\mathbf{r_j}}$$ 
   - computing any point in (hkl) from any structural model!
2. In crystallography usually we have:
   - many atoms, e.g. N_{atoms} $\mathcal{O}(\geq 1 \times 10^8)$
   - many reflections (N_{hkl}) in hkl, e.g. N_{khl} $\mathcal{O}(128 \times 128 \times 128)$
3. It requires **massive parallelism**
   - $N_{\text{flop}} \approx 10 \times N_{\text{atoms}} \times N_{\text{hkl}} : \mathcal{O}(1 \times 10^{14})$ *flop*
   - modern GPU: 10^{13} Flop/s, CPU server: 10^{12} Flop/s
   - algorithm well suited for GPU computations, scalable across multiple GPUs $\rightarrow \mathcal{O}(minutes)$

MPCDF Max Planck Computing & Data Facility

GPU-accelerated analysis of large X-ray data

1. Direct calculation in 3D reciprocal space (hkl): **atomic positions** ($\mathbf{r_i}$) & scattering factors (f_i):
   - $A(\mathbf{s}) = \sum_{j=0}^{N-1} f_j(s) e^{2i\pi \mathbf{S}\cdot\mathbf{r_j}}$ 
   - computing any point in (hkl) from any structural model!
2. In crystallography usually we have:
   - many atoms, e.g. N_{atoms} $\mathcal{O}(\geq 1 \times 10^8)$ 
   - many reflections (N_{hkl}) in hkl, e.g. N_{khl} $\mathcal{O}(128 \times 128 \times 128)$
3. It requires **massive parallelism**
   - $N_{\text{flop}} \approx 10 \times N_{\text{atoms}} \times N_{\text{hkl}} : \mathcal{O}(1 \times 10^{14})$ *flop*
   - modern GPU: 10^{13} Flop/s, CPU server: 10^{12} Flop/s
   - algorithm well suited for GPU computations, scalable across multiple GPUs $\rightarrow \mathcal{O}(minutes)$

MPCDF Max Planck Computing & Data Facility

GPU-accelerated analysis of large X-ray data

1. Direct calculation in 3D reciprocal space (hkl): **atomic positions** ($\mathbf{r_i}$) & scattering factors (f_i):
   - $A(\mathbf{s}) = \sum_{j=0}^{N-1} f_j(s) e^{2i\pi \mathbf{S}\cdot\mathbf{r_j}}$ 
   - computing any point in (hkl) from any structural model!
2. In crystallography usually we have:
   - many atoms, e.g. N_{atoms} $\mathcal{O}(\geq 1 \times 10^8)$ 
   - many reflections (N_{hkl}) in hkl, e.g. N_{khl} $\mathcal{O}(128 \times 128 \times 128)$
3. It requires **massive parallelism**
   - $N_{\text{flop}} \approx 10 \times N_{\text{atoms}} \times N_{\text{hkl}} : \mathcal{O}(1 \times 10^{14})$ *flop* 
   - modern GPU: 10^{13} Flop/s, CPU server: 10^{12} Flop/s 
   - algorithm well suited for GPU computations, scalable across multiple GPUs $\rightarrow \mathcal{O}(minutes)$

MPCDF Max Planck Computing & Data Facility

Outline

MPCDF Max Planck Computing & Data Facility

Big Data Analytics at the MPCDF

PyNX: Tools for **Nano-structures X-tallograhy**

It is an open-source library created by *Vincent Favre-Nicolin* at ESRF[1] with the following main modules:

[1] European Synchrotron Radiation Facility; favre@esrf.fr

[2] **New**: in PyNX 3.1.0, is called `pynx.scattering`

MPCDF Max Planck Computing & Data Facility

PyNX: Tools for **Nano-structures X-tallograhy**

It is an open-source library created by *Vincent Favre-Nicolin* at ESRF[1] with the following main modules:

1. `pynx.gpu`[2]: X-ray scattering computing using graphical processing units, allowing up to 4×10^{11} reflections/atoms/seconds (e.g., 2x nVidia GeForce GTX 980);

[1] European Synchrotron Radiation Facility; `favre@esrf.fr`

[2] **New**: in PyNX 3.1.0, is called `pynx.scattering`

MPCDF Max Planck Computing & Data Facility

PyNX: Tools for **Nano-structures X-tallograhy**

It is an open-source library created by *Vincent Favre-Nicolin* at ESRF[1] with the following main modules:

1. `pynx.gpu`[2]: X-ray scattering computing using graphical processing units, allowing up to 4×10^{11} reflections/atoms/seconds (e.g., 2x nVidia GeForce GTX 980);
2. `pynx.ptycho`: simulation and analysis of experiments using the ptychography technique;

[1] European Synchrotron Radiation Facility; favre@esrf.fr

[2] **New**: in PyNX 3.1.0, is called `pynx.scattering`

MPCDF Max Planck Computing & Data Facility

PyNX: Tools for **Nano-structures X-tallograhy**

It is an open-source library created by *Vincent Favre-Nicolin* at ESRF[1] with the following main modules:

1. `pynx.gpu`[2]: X-ray scattering computing using graphical processing units, allowing up to 4×10^{11} reflections/atoms/seconds (e.g., 2x nVidia GeForce GTX 980);
2. `pynx.ptycho`: simulation and analysis of experiments using the ptychography technique;
3. `pynx.wavefront`: X-ray wavefront propagation in the near, far field, or continuous;

[1] European Synchrotron Radiation Facility; favre@esrf.fr

[2] **New**: in PyNX 3.1.0, is called `pynx.scattering`

MPCDF Max Planck Computing & Data Facility

PyNX: Tools for **Nano-structures X-tallograhy**

It is an open-source library created by *Vincent Favre-Nicolin* at ESRF[1] with the following main modules:

1. `pynx.gpu`[2]: X-ray scattering computing using graphical processing units, allowing up to 4×10^{11} reflections/atoms/seconds (e.g., 2x nVidia GeForce GTX 980);
2. `pynx.ptycho`: simulation and analysis of experiments using the ptychography technique;
3. `pynx.wavefront`: X-ray wavefront propagation in the near, far field, or continuous;
4. `pynx.cdi`: Coherent Diffraction Imaging reconstruction algorithms using GPU.

[1] European Synchrotron Radiation Facility; `favre@esrf.fr`

[2] **New**: in PyNX 3.1.0, is called `pynx.scattering`

MPCDF Max Planck Computing & Data Facility

PyNX: Hardware requirements

PyNX aims to help computing scattering (X-ray or neutrons) maps for atomic structures, especially if they are *distorted or disordered*.

- High-performance computing with GPUs
  - nVidia's CUDA toolkit[3] and the pyCUDA library
  - OpenCL[4] language, along with pyOpenCL library

GPU-accelerated applications

PyNX provides fast parallel computation of *scattering from large assemblies of atoms* ($\gg 10^8$ atoms) and **1D**, **2D** or **3D** coordinates ($\gg 10^6$ grid points) in *reciprocal latticespace*.

[3] https://developer.nvidia.com/cuda-toolkit
[4] https://www.khronos.org/opencl/

MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended).
Python interface ⇒ No need to learn CUDA

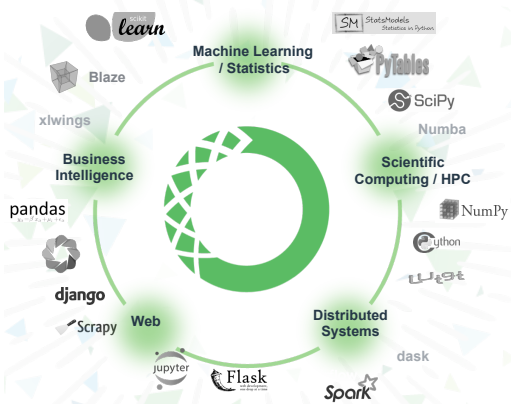
PyNX: Prerequisites

PyNX is available from:

- Mandatory: `numpy`, `matplotlib`, `pycuda`;
- Recommended[5]: `pyfftw` for cpu calculations;
- Optionally[6]: `cctbx` library, for *grazing incidence scattering*

```
$ conda install -c mx cctbx=20160309
```

Python is the Common Language



[6] https://pypi.python.org/pypi/pyFFTW
[7] http://cctbx.sourceforge.net/

MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended).
Python interface ⇒ No need to learn CUDA

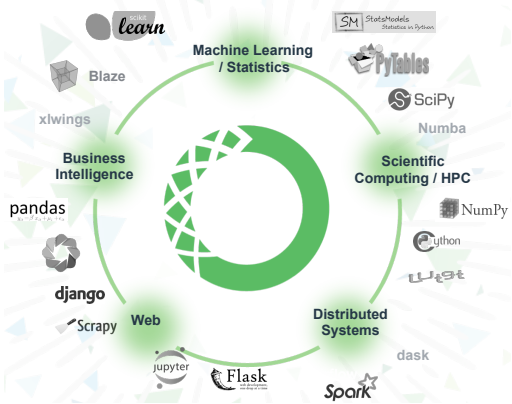
PyNX: Prerequisites

PyNX is available from:

- Mandatory: `numpy, matplotlib, pycuda`;
- Recommended[5]: `pyfftw` for cpu calculations;
- Optionally[6]: `cctbx` library, for *grazing incidence scattering*

```
$ conda install -c mx cctbx=20160309
```

Python is the Common Language



[6] https://pypi.python.org/pypi/pyFFTW
[7] http://cctbx.sourceforge.net/

MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

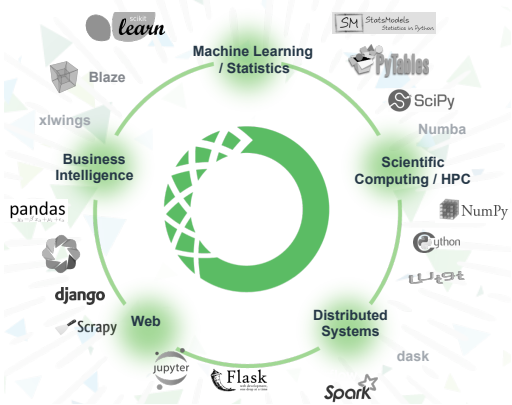
PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended).
Python interface ⇒ No need to learn CUDA

PyNX: Prerequisites

PyNX is available from:

- Mandatory: `numpy, matplotlib, pycuda`;
- Recommended[5]: `pyfftw` for cpu calculations;
- Optionally[6]: `cctbx` library, for *grazing incidence scattering*
  `$ conda install -c mx cctbx=20160309`

Python is the Common Language



[6] https://pypi.python.org/pypi/pyFFTW
[7] http://cctbx.sourceforge.net/

MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended) .
Python interface ⇒ No need to learn CUDA

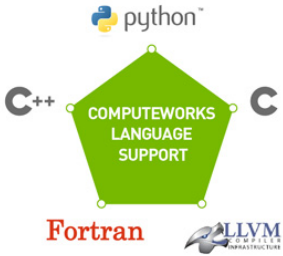
Install PyNX

PyNX is available from:

- `http://ftp.esrf.fr/pub/scisoft/PyNX/`
- `http://gitlab.esrf.fr/favre/PyNX`
- PyPI:

```
$ pip install pynx
```

Python is also a great glue language!



MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended) .
Python interface ⇒ No need to learn CUDA

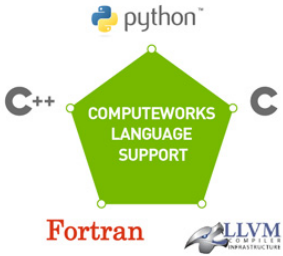
Install PyNX

PyNX is available from:

- `http://ftp.esrf.fr/pub/scisoft/PyNX/`
- `http://gitlab.esrf.fr/favre/PyNX`
- PyPI:

```
$ pip install pynx
```

Python is also a great glue language!



MPCDF Max Planck Computing & Data Facility

PyNX: Software requirements

PyNX supports Python version **2.7, 3.4** and above (Anaconda distribution is recommended) .
Python interface ⇒ No need to learn CUDA

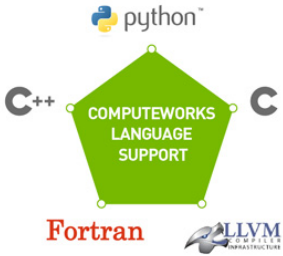
Install PyNX

PyNX is available from:

- `http://ftp.esrf.fr/pub/scisoft/PyNX/`
- `http://gitlab.esrf.fr/favre/PyNX`
- `PyPI`:

```
$ pip install pynx
```

Python is also a great glue language!



MPCDF Max Planck Computing & Data Facility

Why do Scripting for GPUs?

- GPUs are everything that *high-level scripting languages* **are not**.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
    $\rightarrow$ complement each other
- Scripting + GPU : A good combination

$$\Downarrow$$

- Python + CUDA[7] = **PyCUDA**
- Python + OpenCL = **PyOpenCL**

Figure: courtesy by A. Klöckner, Nvidia GTC 2010

[8] CUDA is a general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA graphics processing units (GPUs)

MPCDF Max Planck Computing & Data Facility

But before PyCUDA: why Python at all?

- general purpose
- simple to learn and use
- extensible and embeddable: Python C API
- science oriented too (NumPy, SciPy, mpi4py)
- great visualization tools (Bokeh, Matplotib)
- very well documented

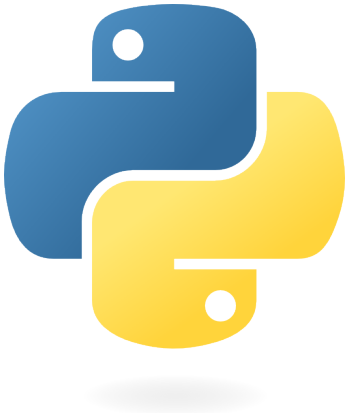
NumPy: package for large, N-dimensional array object

- `Vectors, Matrices, ...` array computing is fast
- `A+B, sin(A), dot(A,B)`
- `la.solve(A,b), la.eig(A)`
- `cube[:, :, n-k:n+k], cube+5`
- `FFT's`, tools for integrating `C/C++` and `Fortran` code

Scripting: Python[8]

One example of a scripting language: Python

- Mature
- Large and active community
- Emphasizes readability
- Written in widely-portable C
- A 'multi-paradigm' language
- Rich ecosystem of sci-comp related software



[8] Slide courtesy by A. Klöckner, Nvidia GTC 2010

MPCDF Max Planck Computing & Data Facility

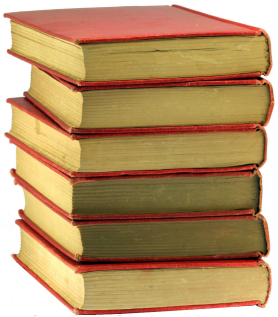
PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python.

MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

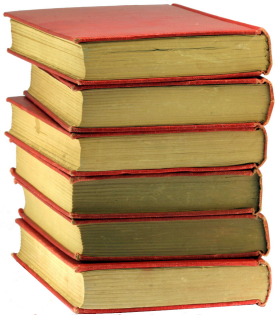


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking

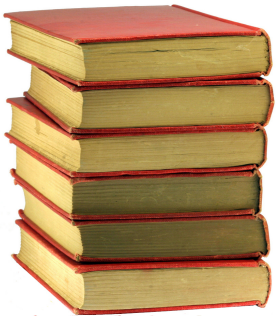


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking
- **Completeness**: automatic error checking

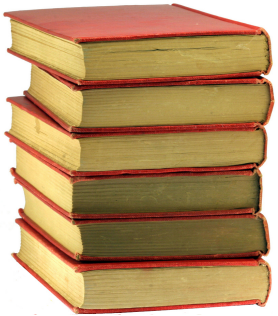


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking
- **Completeness**: automatic error checking
- **Convenience**: provide abstractions (comes with ready-made on-GPU linear algebra, reduction, scan. Add-on packages for FFT and LAPACK available.)

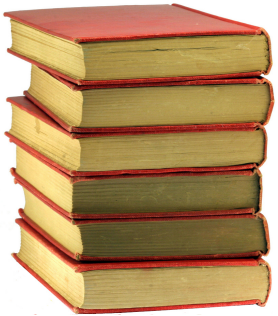


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking
- **Completeness**: automatic error checking
- **Convenience**: provide abstractions (comes with ready-made on-GPU linear algebra, reduction, scan. Add-on packages for FFT and LAPACK available.)
- Integrate tightly with **NumPy**

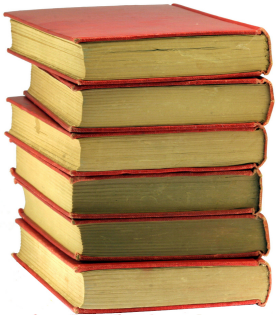


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking
- **Completeness**: automatic error checking
- **Convenience**: provide abstractions (comes with ready-made on-GPU linear algebra, reduction, scan. Add-on packages for FFT and LAPACK available.)
- Integrate tightly with **NumPy**
- **Speed**: PyCUDA's base layer is written in C++ (near-zero wrapping overhead)

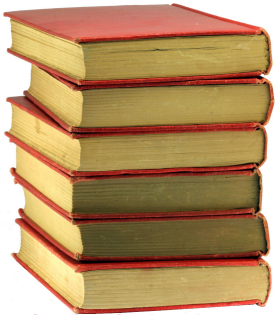


MPCDF Max Planck Computing & Data Facility

PyCUDA Philosophy: maps all CUDA into Python

PyCUDA lets you completely access to Nvidia's CUDA parallel computation API from Python. **Key Features**:

- **Robustness**: automatic management of object lifetimes and error checking
- **Completeness**: automatic error checking
- **Convenience**: provide abstractions (comes with ready-made on-GPU linear algebra, reduction, scan. Add-on packages for FFT and LAPACK available.)
- Integrate tightly with **NumPy**
- **Speed**: PyCUDA's base layer is written in C++ (near-zero wrapping overhead)
- Complete, helpful documentation



MPCDF Max Planck Computing & Data Facility

Whetting your appetite: SourceModule

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
a = np.random.randn(4,4).astype(np.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
# host-to-device
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule("""
  __global__ void multiply_by_two(float *a)
{
int idx = threadIdx.x + threadIdx.y*4;
a[idx] *= 2;
}
""")

func = mod.get_function("multiply_by_two")
func(a_gpu, bloc=(4,4,1))
```

Compute Kernel

MPCDF Max Planck Computing & Data Facility

Whetting your appetite: GPUArrays

```
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
a_gpu = gpuarray.to_gpu(np.random.randn(4,4).astype(np.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

GPUArrays: computational linear algebra

- element-wise algebraic operations (+, -, *, /, sin, cos, exp)
- tight integration with numpy
  - gpuarray.to_gpu(numpy_array)
  - numpy_array = gpuarray.get()
- mixed data types (int32 + float32 = float64)

PyCUDA Workflow: "Edit-Run-Repeat"

A two-fold aim:

1. usage of *existing* `CUDA C`
2. *on top* of the first layer, PyCUDA $\Rightarrow$ *abstractions*

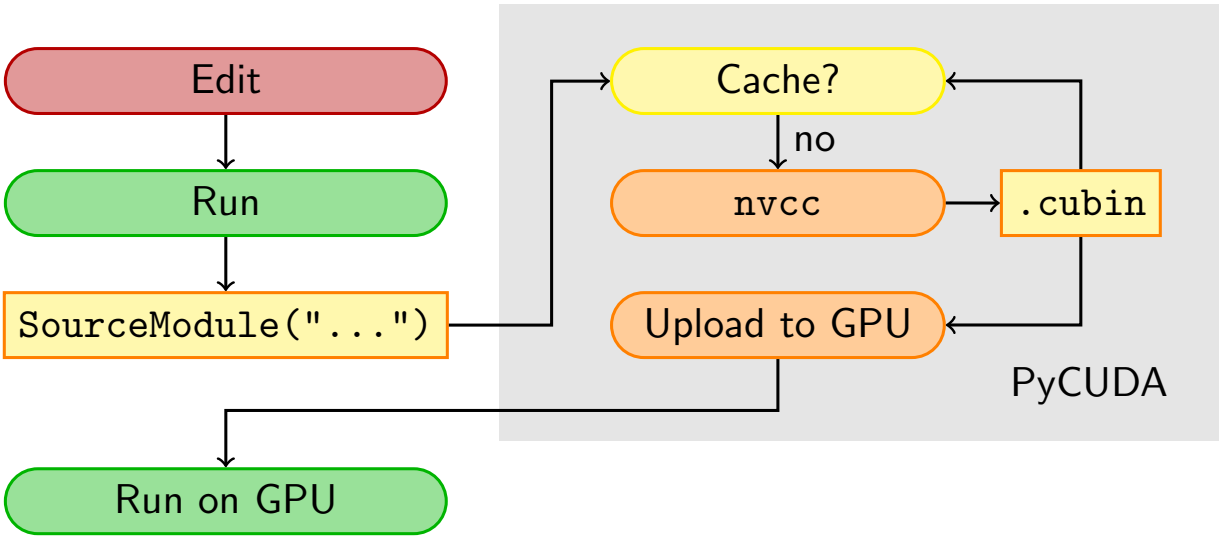


Figure: A. Klöckner et al. 2013, https://arxiv.org/abs/1304.5553

MPCDF Max Planck Computing & Data Facility

PyCUDA: Vital Information

- Availability: Freely down-loadable from this location
  - Open source
  - MIT Licensed
- requires **NumPy**, Python 2.4+ (Win/OS X/Linux)
- Support via mailing list
- For further information see:
  - Main PyCUDA page
  - PyCUDA Wiki
  - PyCUDA FAQ

Figure: courtesy by A. Klöckner, Nvidia GTC 2010

MPCDF Max Planck Computing & Data Facility

PyNX **kernel**: `__global__ void CUDA_fhkl()`

```
1  const unsigned long ix=threadIdx.x+blockDim.x*blockIdx.x; // Thread idx
2  const float h=twopi*vh[ix];
3  const float k=twopi*vk[ix];
4  const float l=twopi*vl[ix];
5  float fr=0,fi=0;
6  __shared__ float x[BLOCKSIZE]; // Shared memory between BLOCKSIZE %d
7  __shared__ float y[BLOCKSIZE]; // parallel threads
8  __shared__ float z[BLOCKSIZE];
9  long at=0;
10 for (;at<=(natoms-BLOCKSIZE);at+=BLOCKSIZE) {
11   __syncthreads();
12   x[threadIdx.x]=vx[at+threadIdx.x]; // "Coalesced" transfer
13   y[threadIdx.x]=vy[at+threadIdx.x]; // to shared memory
14   z[threadIdx.x]=vz[at+threadIdx.x];
15   __syncthreads();
16   for(unsigned int i=0;i<BLOCKSIZE;i++) {// Each thread computes
17      float s,c;                          // a single reflection
18      __sincosf(h*x[i] + k*y[i] + l*z[i] , &s,&c);
19      fr +=c; // fast, intrensic trigonometric function
20      fi +=s;
21   }__syncthreads();}
```

PyNX **kernel**: `__global__ void CUDA_fhkl()`

```
1  __syncthreads()
2  ;}
3
4  /* Take care of remaining atoms */
5  if(threadIdx.x<(natoms-at)) {
6      x[threadIdx.x]=vx[at+threadIdx.x];
7      y[threadIdx.x]=vy[at+threadIdx.x];
8      z[threadIdx.x]=vz[at+threadIdx.x];
9  }
10 __syncthreads();
11 for(long i=0;i<(natoms-at);i++) {
12     float s,c;
13     __sincosf(h*x[i] + k*y[i] + l*z[i] , &s,&c);
14     fr +=c;
15     fi +=s;
16 }
17 fhkl_real[ix]+=fr;
18 fhkl_imag[ix]+=fi;
19 }
```

PyNX **Python interface**: loading the data

```
1  import numpy as np
2  import time
3
4  @timeit
5  def do_work(*args, **kwargs):
6    def read_pos(file_name):
7        """Loads an APT.pos file
8           Columns: x, y, z, Da
9        """
10       print("reading in the data")
11       f = open(file_name, mode='rb') # Load orthonormal coord.
12       dt_type = np.dtype({'names':['x', 'y', 'z', 'Da'],
13                           'formats':['>f4', '>f4', '>f4', '>f4']})
14       pos = np.fromfile(f, dt_type, -1)
15       f.close()
16       print("the data contain: {0:.5e} atoms".format(pos.size))
17       return pos
```

PyNX Python interface: parsing the data

```
16      if args[1] is not None:
17          fname = args[1]
18
19      df = readpos(fname)
20      xEl = df['x'] # slicing columns by labels
21      yEl = df['y']
22      zEl = df['z']
23      lattice_parameter = 0.4045
24      xEl /= lattice_parameter # Convert to fractional coordinates
25      yEl /= lattice_parameter
26      zEl /= lattice_parameter
27      N = 128
28      h = np.linspace(-1.1, 1.1, num=N, endpoint=True) # HKL as 3D
29      k = np.linspace(-1.1, 1.1, num=N, endpoint=True)[:, np.
            newaxis]
30      l = np.linspace(-1.1, 1.1, num=N, endpoint=True)[:, np.
            newaxis, np.newaxis]
```

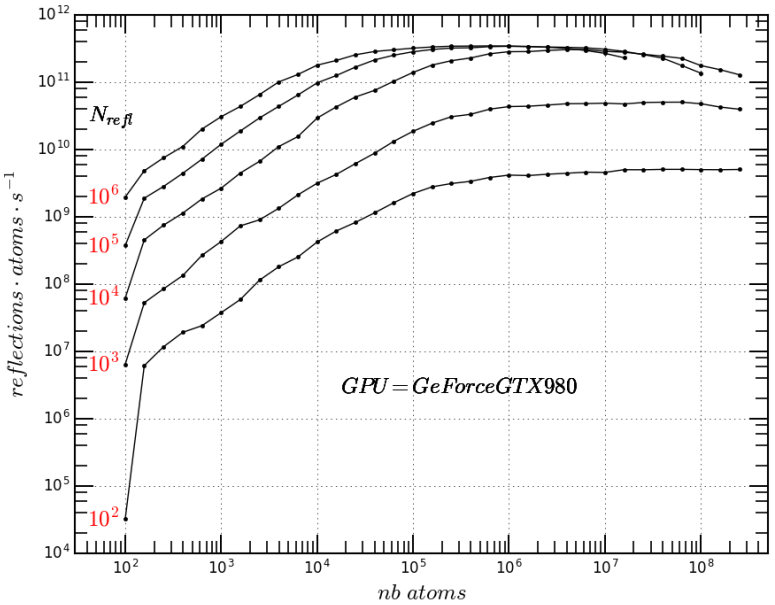
PyNX Python interface: The Fourier Transform

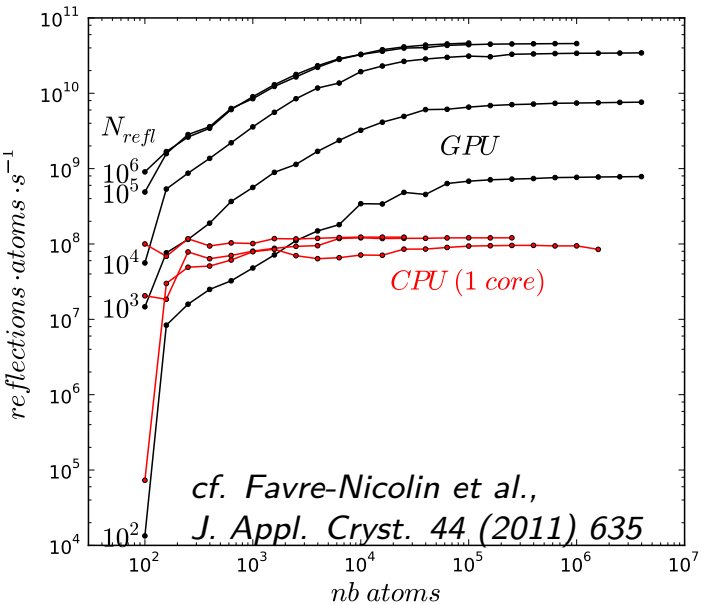
```
31      from pynx import gpu # main module for scattering computation
32      gpu_name = args[0] # identify GPU, e.g. GeForce GTX980
33
34      if args[0] is not None:
35          fhkl, dt = gpu.Fhkl_thread(h, k, l, xEl, yEl, zEl,
                gpu_name=args[0],verbose=True,language="CUDA",
                cl_platform="nvidia")
36          print("The FT computed in dt = {0:7.5f}:".format(dt))
37          print("{0:d} fourier number points".format(fhkl.size))
38
39      return fhkl # a numpy complex64 array
```

gpu.Fhkl_thread

- computes $F(hkl) = \sum_i \exp[2j\pi(x_i \times h + y_i \times k + z_i \times l)]$;
- distributes the scattering maps on **several GPU**.

PyNX: Performance

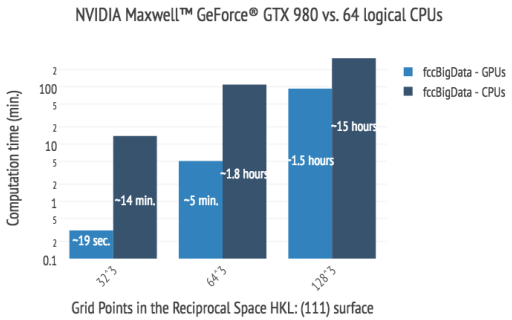




assesment of **GPU-accelerated python** software package PyNX

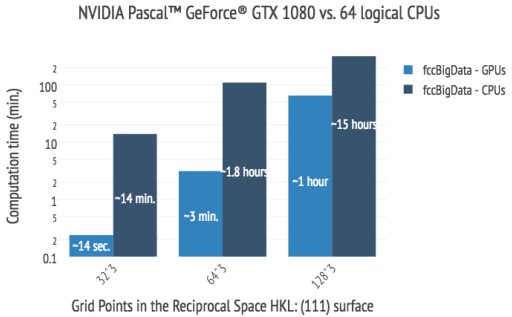
- Effective (singe precision) throughput per GPU:
  up to 3.5×10^{11} **reflections** $\cdot$ **atoms** $\cdot$ $\mathbf{s}^{-1}$

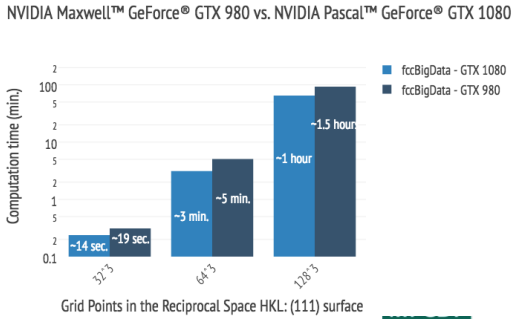
PyNX: benchmarks at the MPCDF



PyNX computing time at the MPCDF for 10^8 atoms

1. 2 × nVidia GTX 980
2. 2 × nVidia GTX 1080
3. interactive plots with Plotly





MAX PLANCK COMPUTING & DATA FACILITY

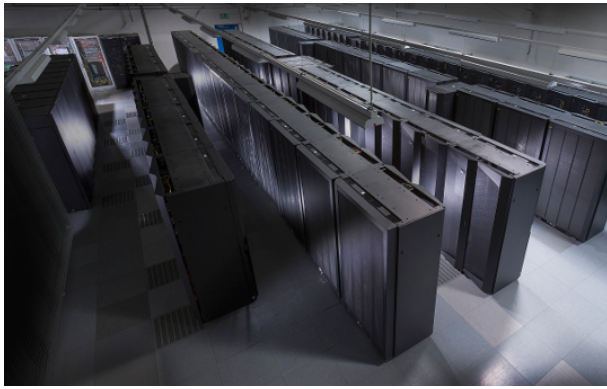
Deploying Data Science Projects - Clusters

```
  #!/bin/bash -l
#SBATCH -J pynx_run
# Queue:
#SBATCH -partition=gpu
# Node feature:
#SBATCH -constraint="gpu"
# N. of nodes and MPI tasks per node:
#SBATCH -nodes=1
#SBATCH -ntasks-per-node=32
# wall clock limit:
#SBATCH -time=02:00:00
echo "starting the job ..."
python crystAl.py -g="GeForce GTX 980"
echo "...done"
```

SLURM worload manager: basic usage

- qsub: submit your job;
- qstat: query queue/job status;
- qdel: delete your job

SUPERCOMPUTING DRIVES SCIENCE THROUGH SIMULATIONS



The MPG SuperComputer Hydra at the MPCDF

MPCDF MAX PLANCK COMPUTING & DATA FACILITY

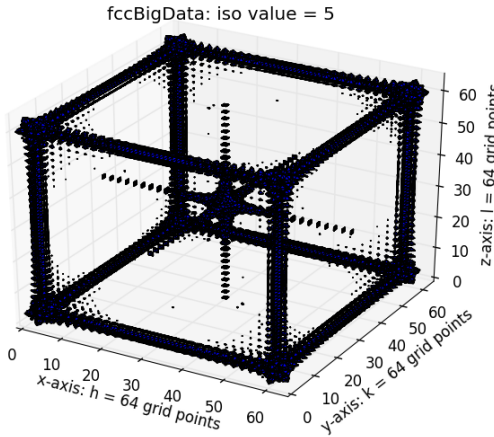
Outline

MPCDF Max Planck Computing & Data Facility

Image Processing in Python: Toolbox for SciPy

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import
Poly3DCollection
from mpl_toolkits.mplot3d import Axes3D
from skimage import measure
pow_spec = abs(np.log10(abs(fhkl)**2))
mean_iso = (pow_spec.max() + pow_spec.min())/2.
verts, faces =
measure.marching_cubes(pow_spec, mean_iso)
fig = plt.figure()
ax = plt.axes(projection="3d")
mesh = Poly3DCollection(verts[faces])
ax.add_collection3d(mesh)
plt.show()
```

scikit-image
image processing in python



Marching Cubes

1. MC is an algorithm to extract a 2D surface mesh from a 3D volume;
2. ‘verts[faces]‘ to generate a collection of triangles;
3. Display triangular mesh using Matplotlib.

MPCDF Max Planck Computing & Data Facility

NumPy to VTK

```
1 from pyevtk.hl import gridToVTK
2 h=k=l = np.linspace(-1.1, 1.1, num=N, endpoint=True, dtype='
      float64')
3 gridToVTK("/viz/vtk/" + filename, h, k, l, pointData = {"pow_spec
      " : pow_spec})
```

PyEVTK: a great little package by a Paulo Herrera

1. `$ pip install pyevtk`
2. save **NumPy** arrays straight to different types of VTK XML-based
3. visualize and process your **NumPy** arrays with any of the flagship VTK applications such as ParaView, VisIt, Mayavi

MPCDF Max Planck Computing & Data Facility

What is ParaView?

- ParaView is an **open-source application** and **architecture display** and analysis of scientific data sets;
- ParaView has been demonstrated to process billions of unstructured cells and to process over a trillion structured cells;
- ParaView's parallel framework has run on over 100,000 processing cores (from notebooks to world's largest supercomputers);
- ParaView's **key features** are:
  - An open-source scalable, multi-platform for visualizing **2D/3D data** (excels at traditional scientific vis qualitative 3D rendering);
  - An extensible, modular **architecture** based on open standards

    e.g. Custom apps, plugins, `Python` scripting, `ParaViewWeb`, `Catalyst`
  - Support for distributed computation models to process **large data sets**;
  - An open, extensible, and **intuitive user interface**.

MPCDF Max Planck Computing & Data Facility

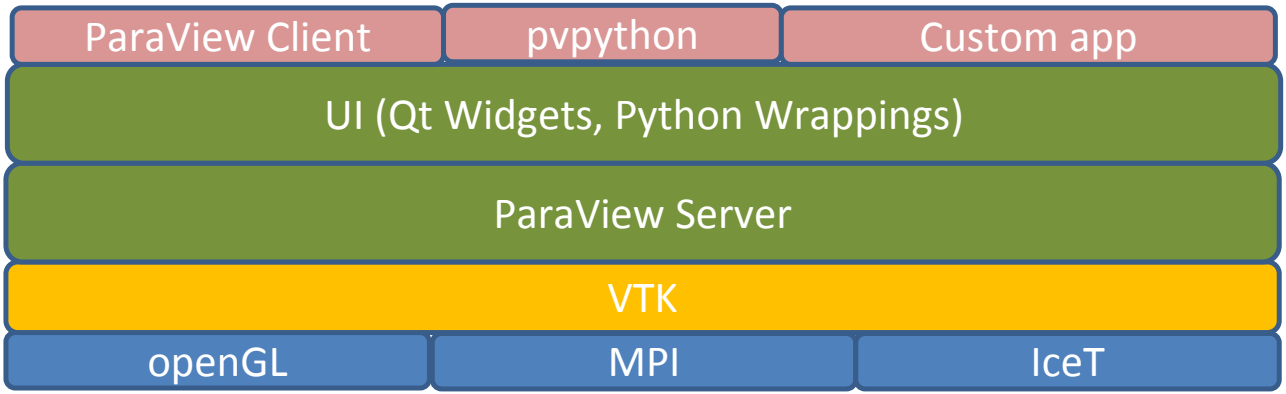
ParaView: a standard de-facto

- ParaView is primarily developed and published by Kitware Inc.
  - A flexible BSD 3-clause license;
  - Commercial maintenance and support;
- ParaView is used by many academic, government, and commercial institutions all over the world;
- ParaView is downloaded roughly 100,000 times every year;
- ParaView also won the HPCwire Readers' Choice Award and HPCwire Editors' Choice Award for Best HPC Visualization Product or Technology.

MPCDF Max Planck Computing & Data Facility

ParaView: The architecture

The application most people associate with ParaView is really just a small client application built on top of a **tall stack** of libraries that provide ParaView with its functionality.



MPCDF Max Planck Computing & Data Facility

ParaView: The big picture

The application most people associate with ParaView is really just a small client application built on top of a **tall stack** of libraries that provide ParaView with its functionality.

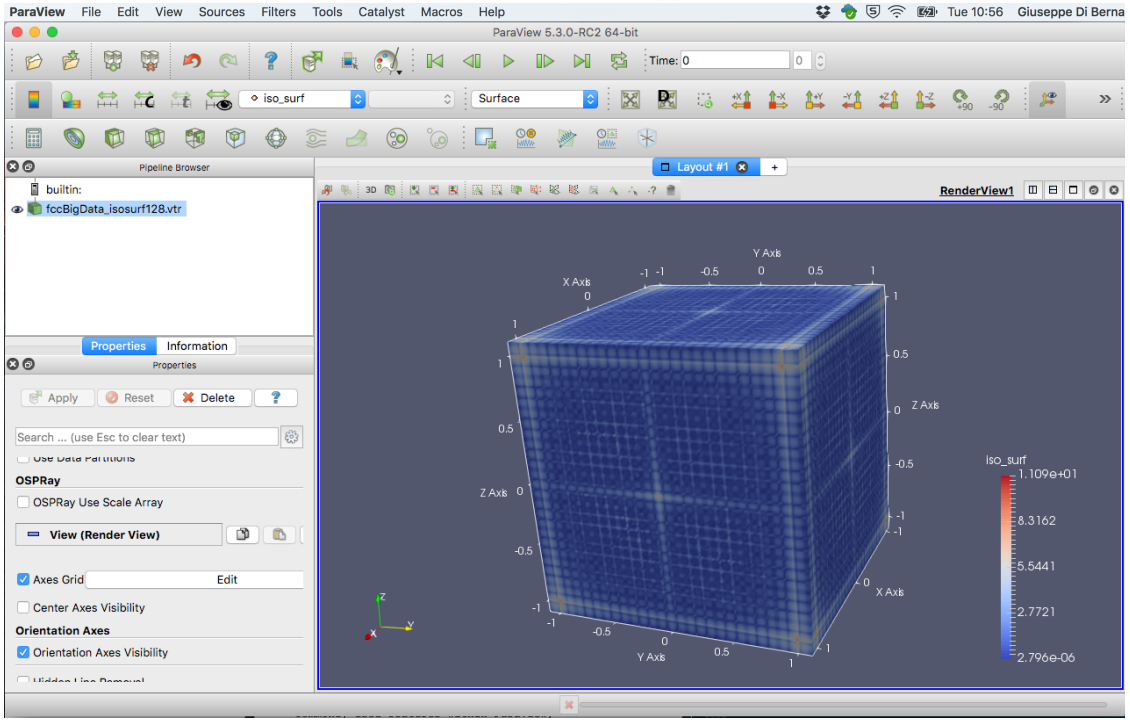
Full Stack

- ParaView comes with a `pvpython` application that allows you to automate the visualization and post-processing with `Python` scripting;
- ParaView Server library `pvserver` provides the abstraction layer necessary for running parallel, interactive visualization. It relieves the client application from most of the issues concerning if and how ParaView is running in parallel;
- The **Visualization Toolkit** (VTK) provides the basic visualization and rendering algorithms $\Rightarrow$ **Data-Flow Paradigm**

ParaView: Loading, filtering and rendering

GUI elements

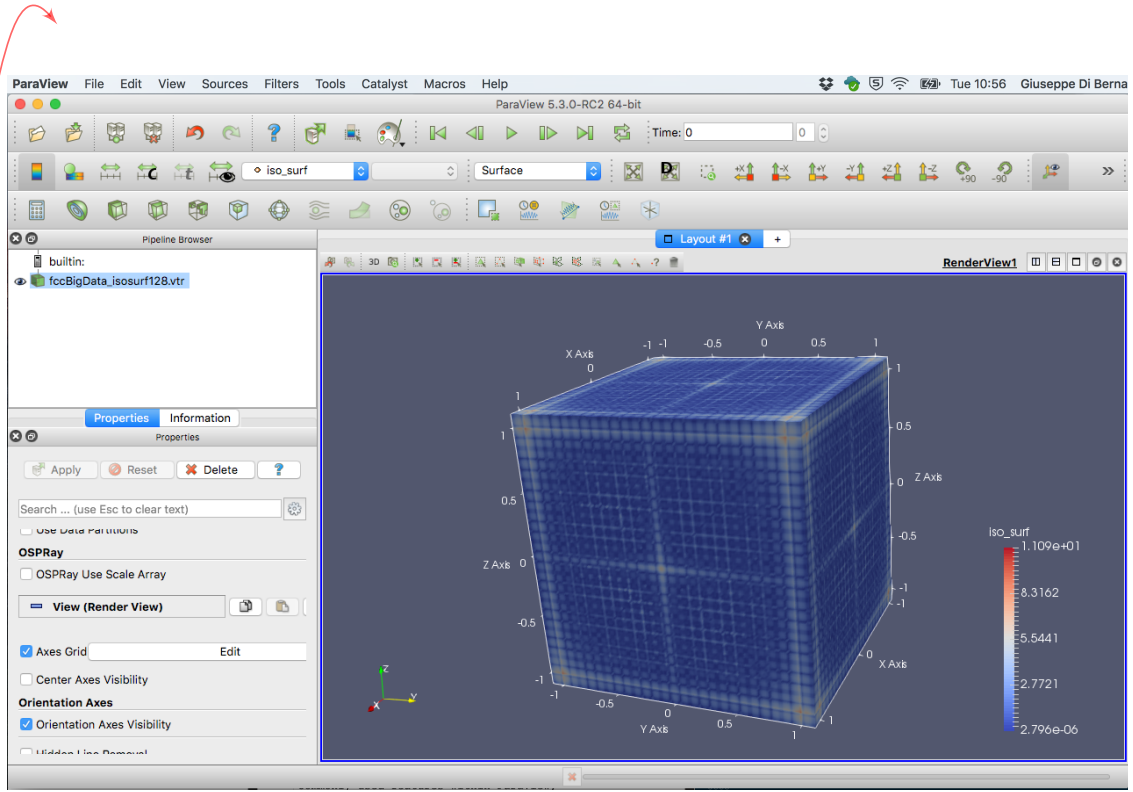
- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views



ParaView: Loading, filtering and rendering

GUI elements

- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views

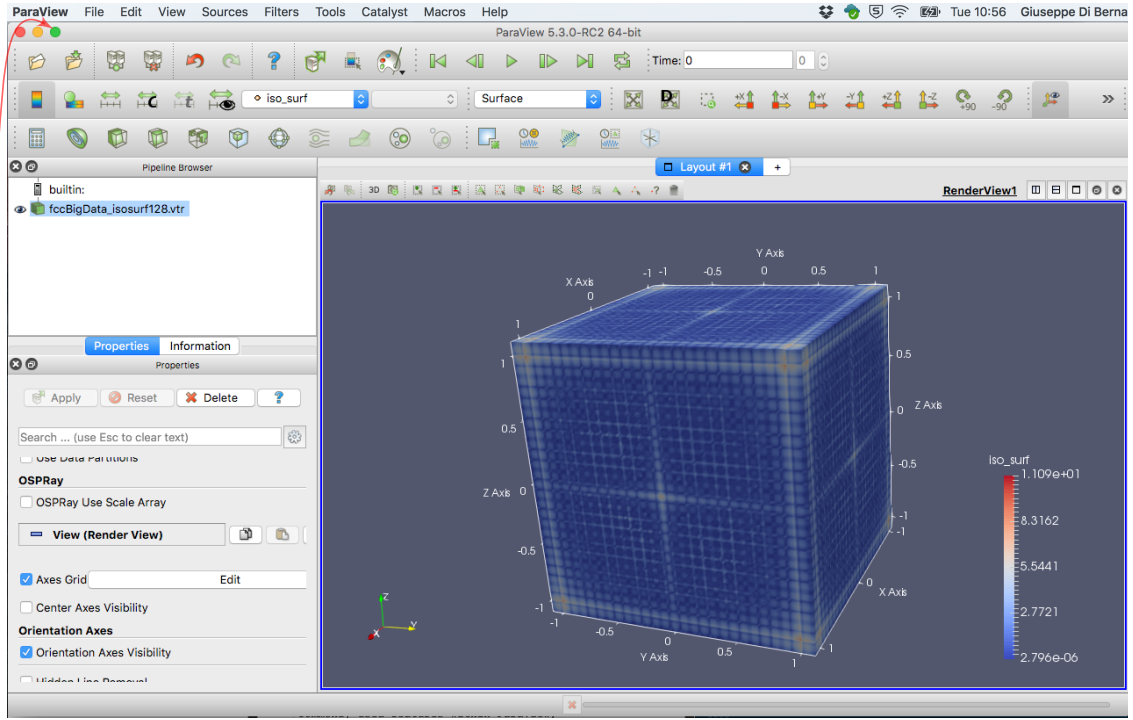


MPCDF Max Planck Computing & Data Facility

ParaView: Loading, filtering and rendering

GUI elements

- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views

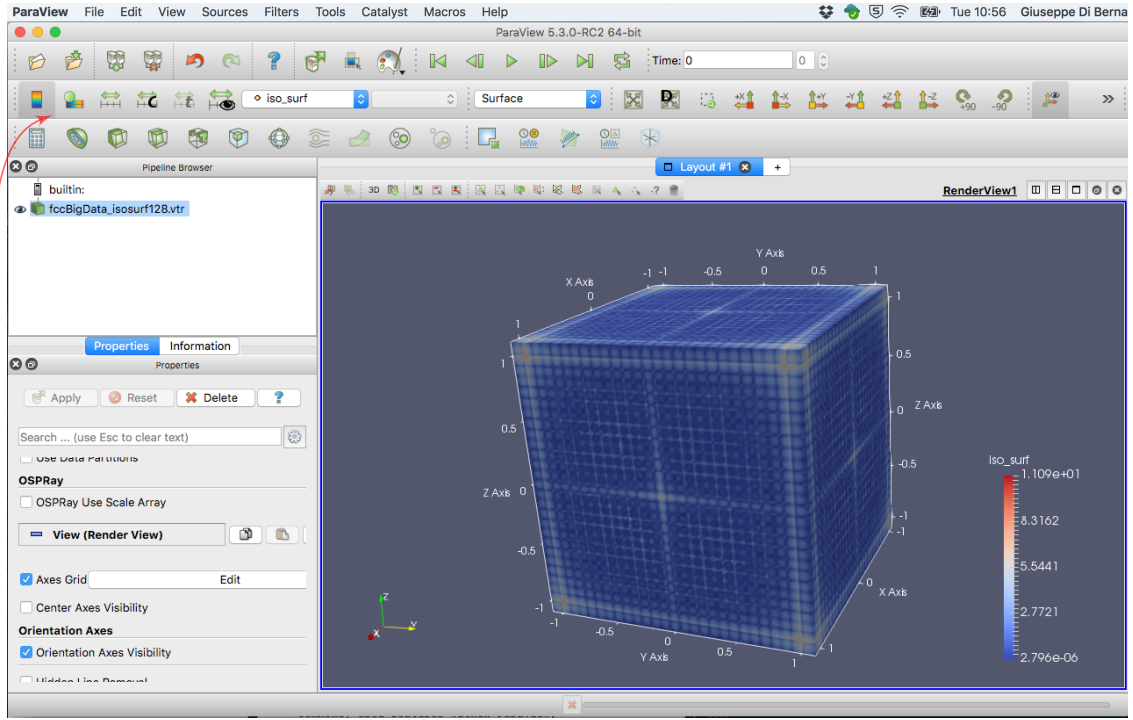


MPCDF Max Planck Computing & Data Facility

ParaView: Loading, filtering and rendering

GUI elements

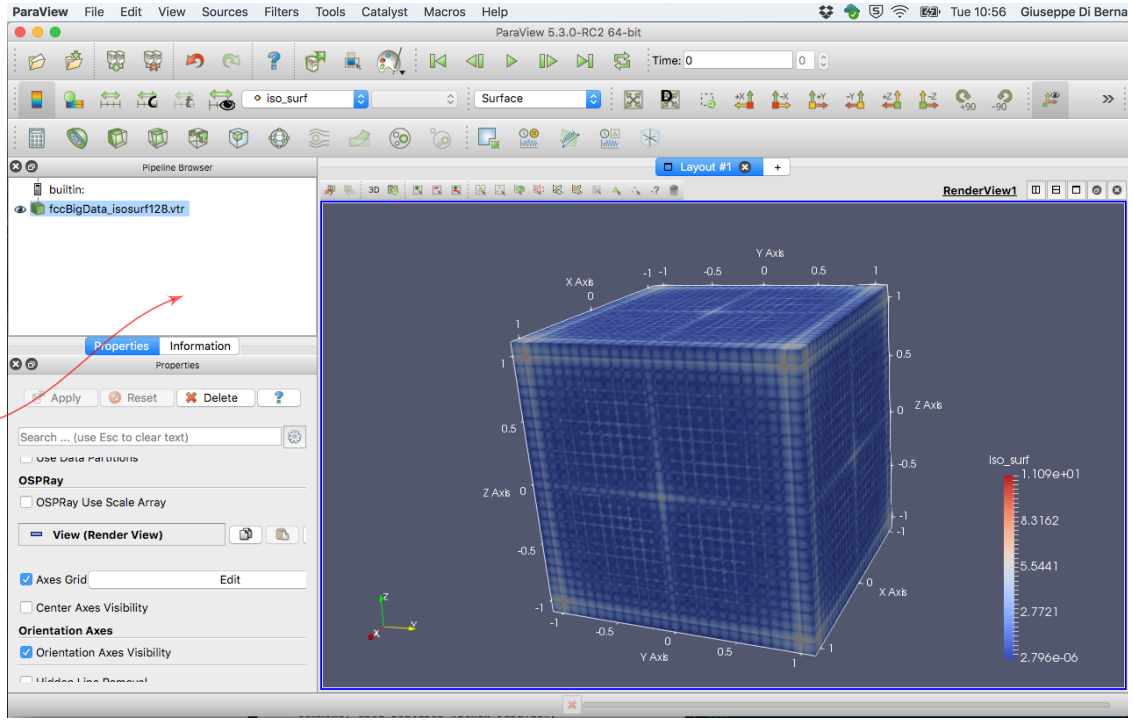
- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views



ParaView: Loading, filtering and rendering

GUI elements

- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views

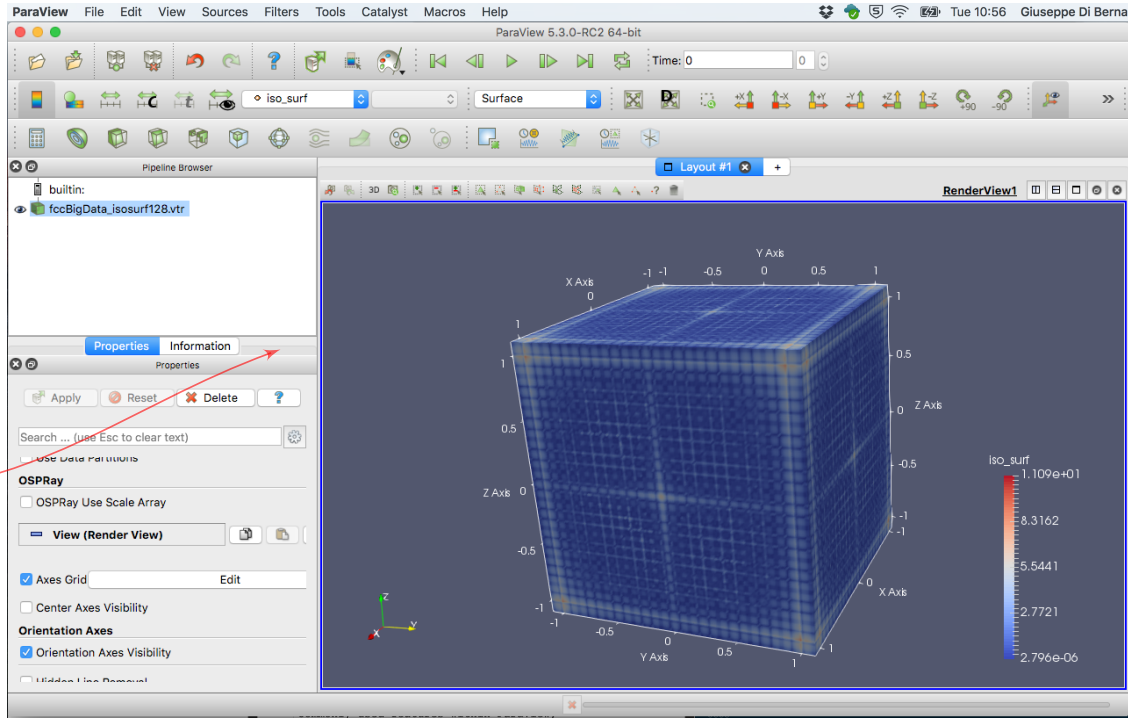


MPCDF Max Planck Computing & Data Facility

ParaView: Loading, filtering and rendering

GUI elements

- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views

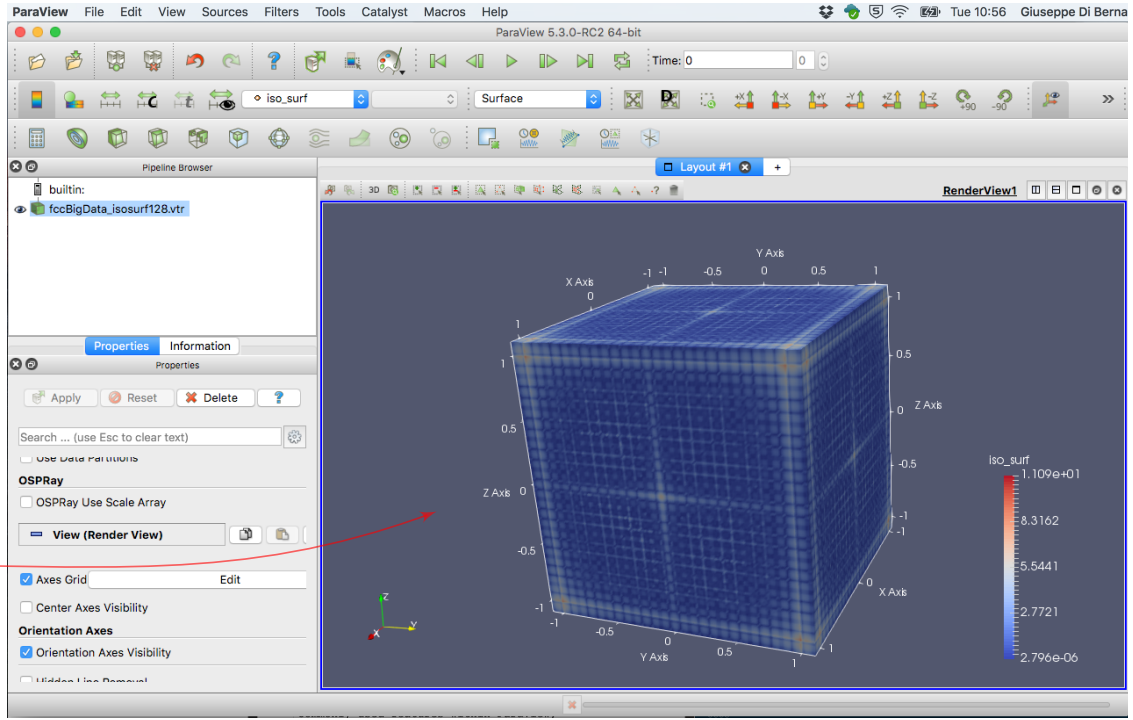


MPCDF Max Planck Computing & Data Facility

ParaView: Loading, filtering and rendering

GUI elements

- Menu
- Toolbars
- Pipeline
- Inspector
- Help
- Views



MPCDF Max Planck Computing & Data Facility

GUI elements definition

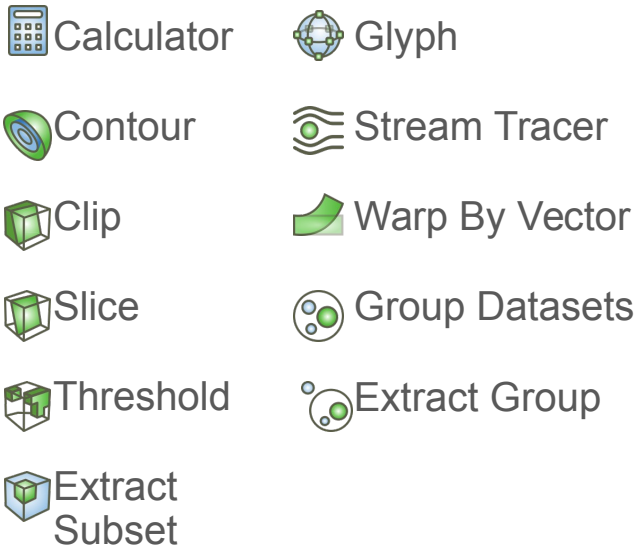
1. **Menu Bar**: allows you to access the majority of features;
2. **Toolbars**: provide quick access to the most commonly used features within ParaView;
3. **Pipeline Browser**: ParaView manages the reading and iterating of data with a pipeline. A convenient list of pipeline objects with an indentation style that shows the pipeline structure;
4. **Properties Panel**: to view and change the parameters of the current pipeline object. The properties are by default coupled with an information tab that shows a basic summary of the data produced by the pipeline object;
5. **3D View**: to present the data so that you may view, interact with, and explore your data.

MPCDF Max Planck Computing & Data Facility

Filters Menu

...some of them

1. **Calculator**: evaluates a user-defined expression on a *per-point* or *per-cell* basis
2. **Contour**: extracts the points, curves, or surfaces where a scalar field is equal to a user-defined value. This surface is often also called *iso-surface*
3. **Threshold**: extracts cells that lie within a specified range of a scalar-field



MPCDF Max Planck Computing & Data Facility

Other Filters

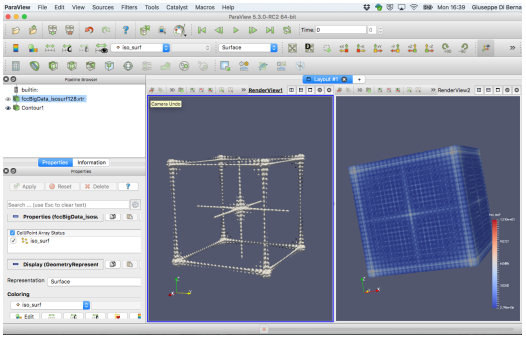
...some of them

- Recent
- AMR (adaptive mesh refinement)
- CTH
- Common (the same list available in the filters toolbars)
- Cosmology

- **Data Analysis**: quantitative values from the data
- Material Analysis
- **Statistics**: descriptive statistics of the data (primarily in tabular form)
- Temporal
- Alphabetical

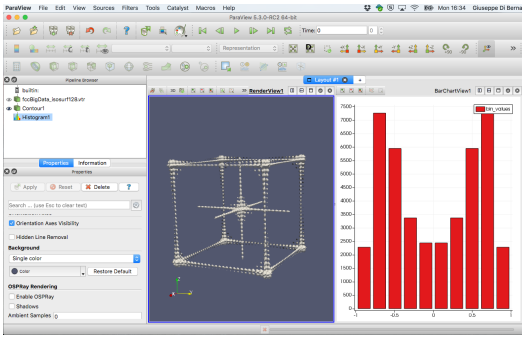
MPCDF Max Planck Computing & Data Facility

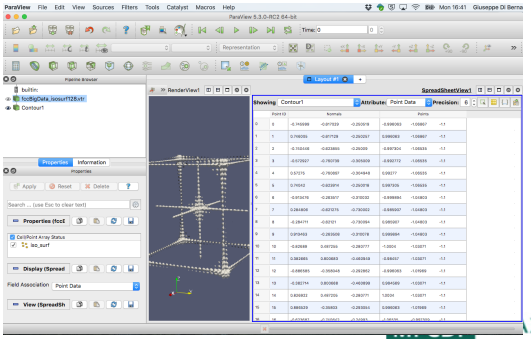
ParaView: Multiple views



Some examples

1. 3d + 3d
2. 3d + Histograms
3. 3d + table





ParaView: The dataset types (sampling structures)

ParaView was designed primarily to handle data with spatial representation. Thus the **data types** used in ParaView are meshes

Uniform Rectilinear
(vtkImageData)

Non-Uniform Rectilinear
(vtkRectilinearData)

Curvilinear
(vtkStructuredData)

Polygonal
(vtkPolyData)

Unstructured Grid
(vtkUnstructuredGrid)

Multi-block

Hierarchical Adaptive Mesh Refinement (AMR)

Hierarchical Uniform AMR

Octree

MPCDF Max Planck Computing & Data Facility

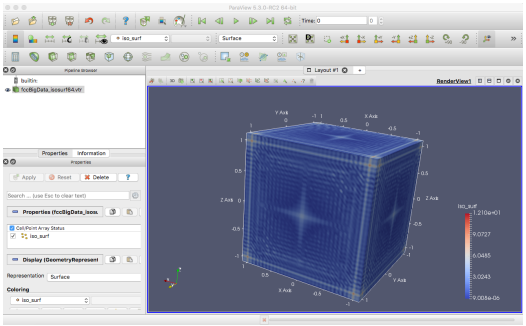
Data Selection

The goal of visualization is often to find the importance details within a large body of information. ParaView's selection abstraction is an important simplification of this process.
Selection is the act of identifying a subset of some dataset. More specifically the subset identifies particular select points, cells, or blocks within any single data set.
In ParaView, selection can take place at any time, and the program maintains a current selected set that is linked between all views.

1. The most direct means to create a selection is via the `Find Data` dialog 
2. Another way of creating a selection is to pick elements right inside the 3D view: **rubber-band selection**

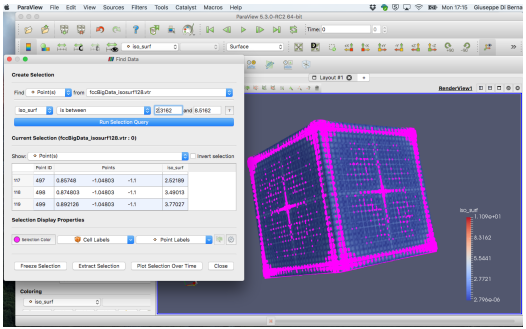
MPCDF Max Planck Computing & Data Facility

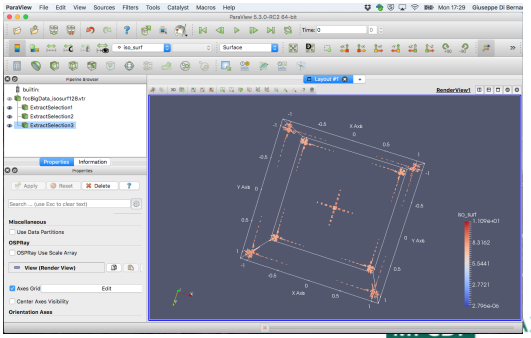
Query Data by Attribute Values - Find Data dialog



Some examples

1. load, filter and display your volume data   
2. find data matching your criteria 
3. extract selection (`Run Selection Query`)





Query Data Spatially - Selection

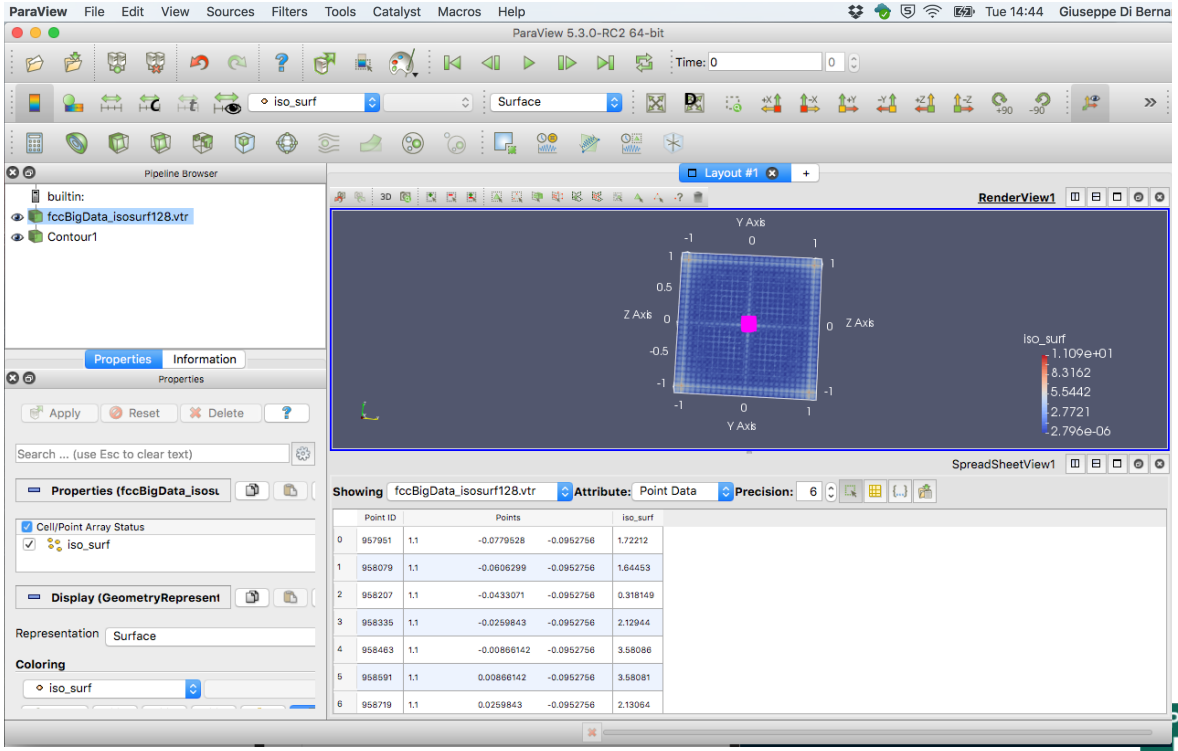
...by clicking and dragging the mouse in the 3D view

- **Select Cells On (Surface)**: cells that are visible in the view under a rubber band
- **Select Points On (Surface)**: points that are visible in the view under a rubber band
- **Select Cells Through (Frustum)**: cells that exist under a rubber band
- **Select Points Through (Frustum)**
- **Select Cells with Polygon**: you draw a polygon
- **Select Points with Polygon**
- **Select Blocks**: Select blocks in a multiblock dataset
- **Int. Select Cells**
- **Int. Select Points**

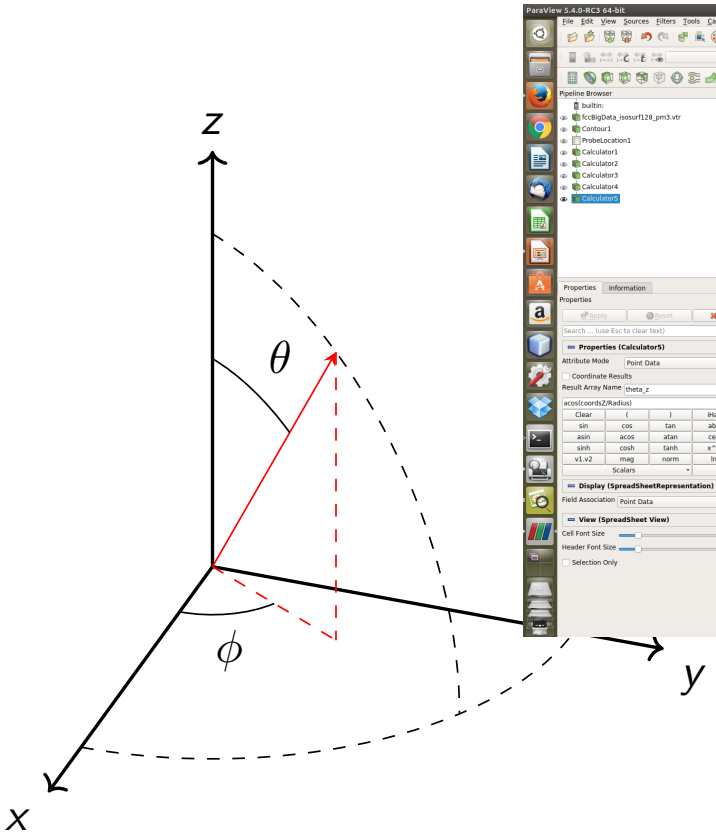
MPCDF Max Planck Computing & Data Facility

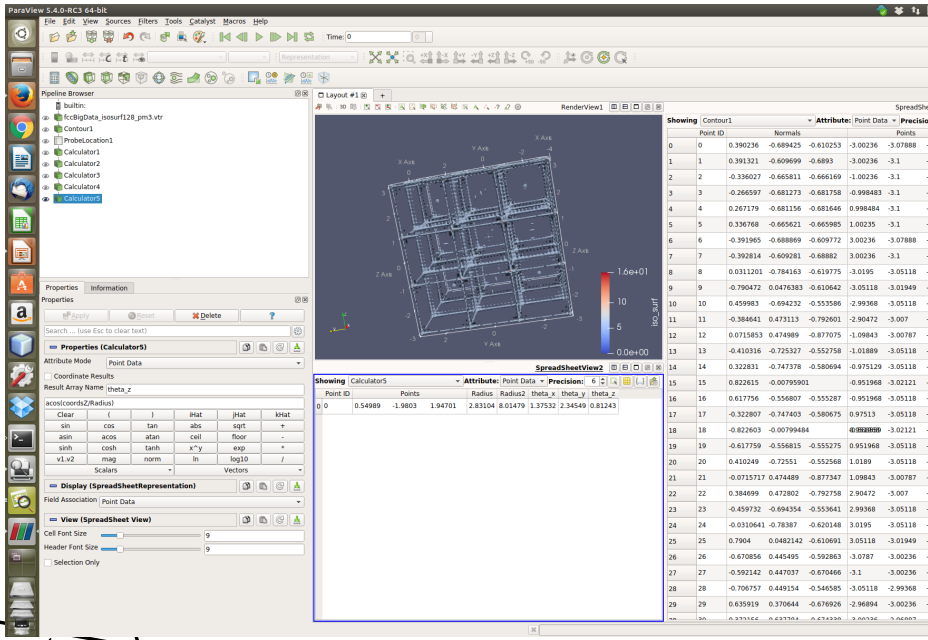
Query Data Spatially - Selection

...by clicking and dragging the mouse in the 3D view



Hands-On: extract the tuple of radius and angles





The End

MPCDF Max Planck Computing & Data Facility