

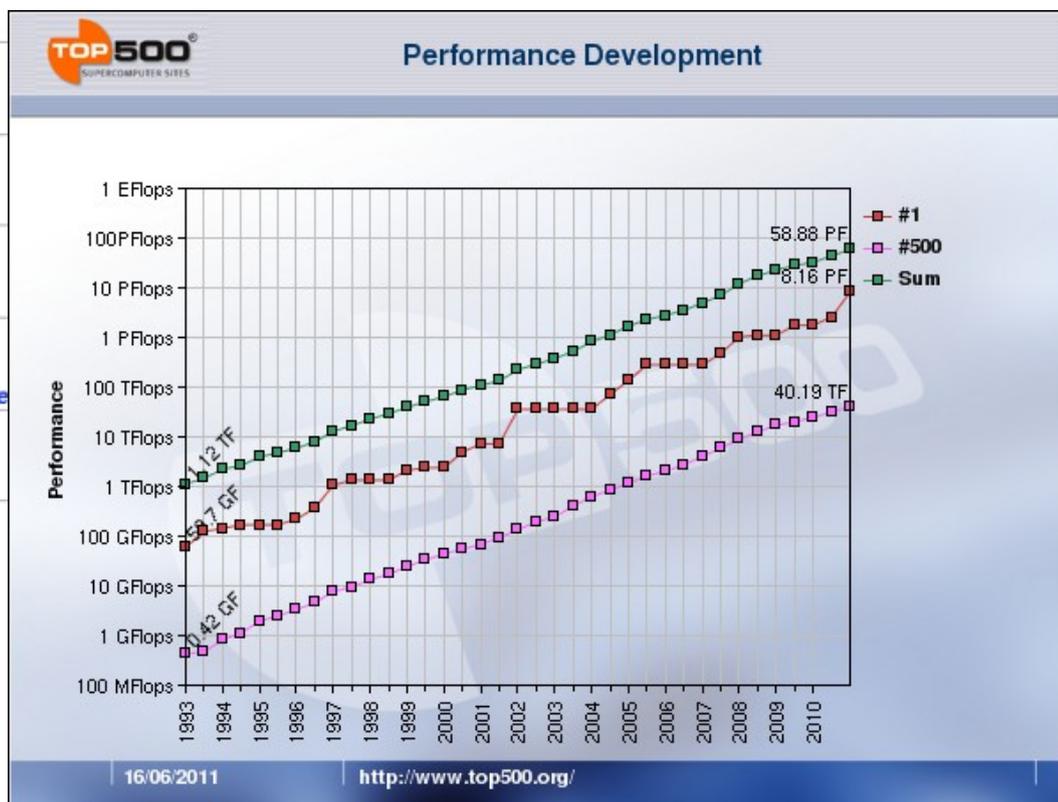
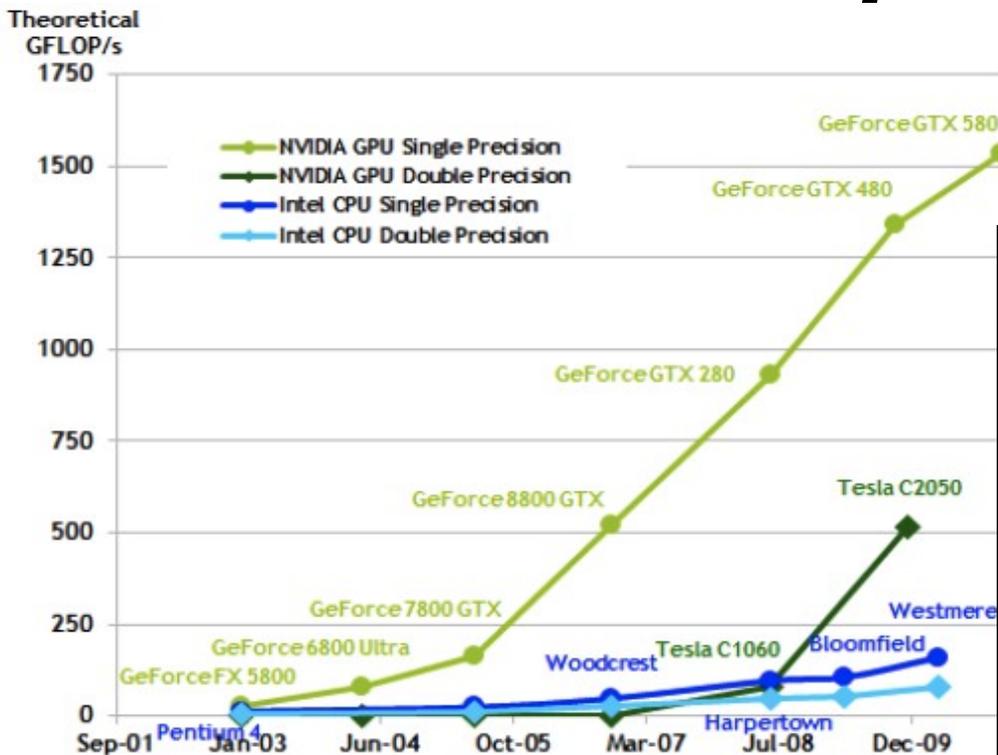


Crystallography "On-A-Chip": Using GPU for fast scattering computing

***Vincent.Favre-Nicolin@cea.fr
CEA / INAC & Université Joseph Fourier
Grenoble, France***



Computing Power

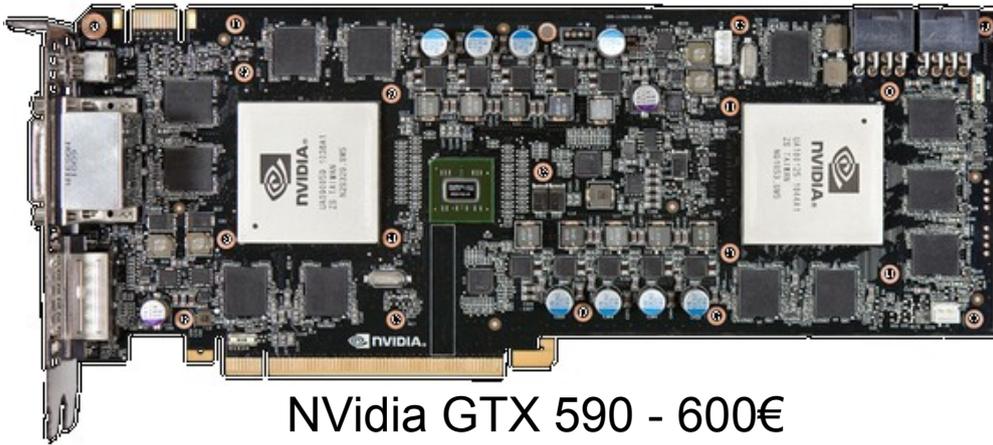


<http://developer.nvidia.com/category/zone/cuda-zone>

CPU cores can yield up to ~10 Gflop/s ... if you really tune your code ... usually closer to 1 Gflop/s

<http://www.top500.org>

Graphical Processing Units



NVidia GTX 590 - 600€
2.5 Tflop/s (single), 365W



AMD Radeon HD 6990 – 600€
5 Tflop/s (single), 375W



NVidia GTX 560 - <200€
1.2 Tflop/s (single)

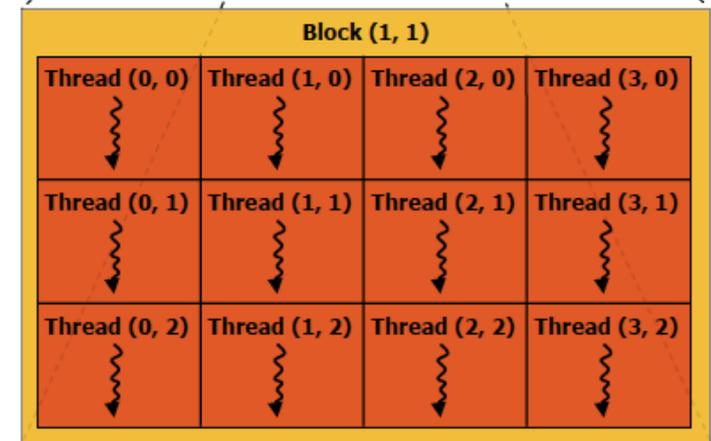
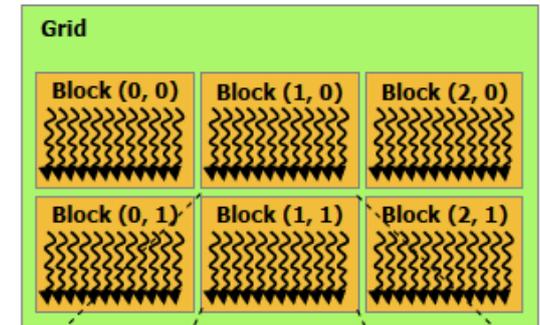
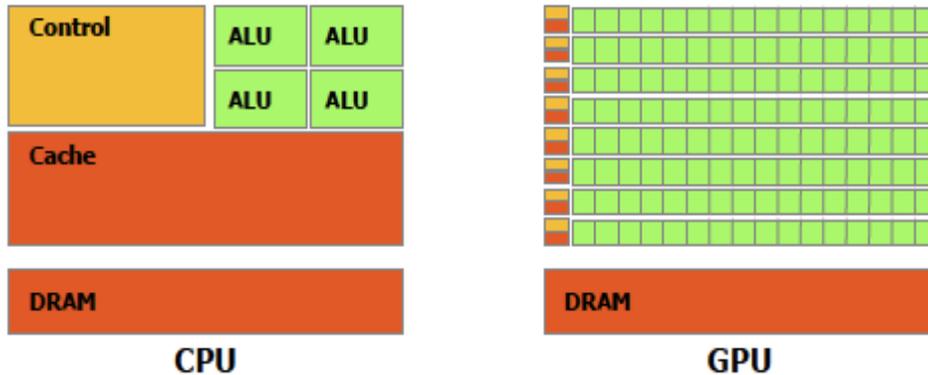
Most consumer graphic cards allow GPU computation, either using CUDA or OpenCL

<http://developer.nvidia.com/cuda-gpus>

http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units

GPU computing principles: Massive Parallelism



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

GPU are optimized for $> 10^4$ parallel threads, with more specialized instructions (fast exp, trigonometric functions,...)
GPU: typ. 768 active threads/multiprocessor, with 30 multiprocessors

All GPU threads must execute exactly the same code (on different data)
=> *no branching (if/else)*

GPU computing principles: Massive Parallelism

if/else Branching Workaround :

Not parallel:

```
if(a>1) b=2; else b=3;
```

Parallel:

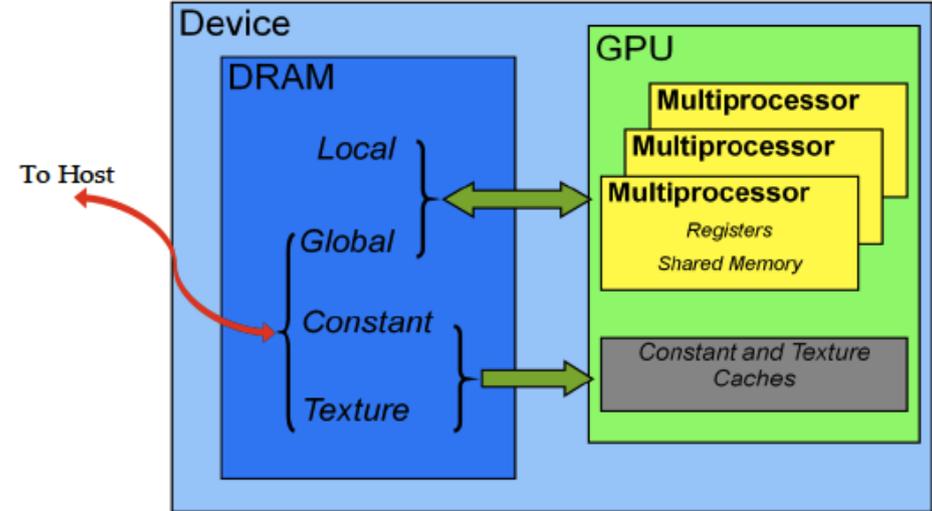
```
b=3-1*int(a>1);
```

**All GPU threads must execute exactly
the same code (on different data)
=> *no branching (if/else)***

GPU computing efficiency

On a GPU:
Computing is cheap
Memory transfers are (relatively) slow

A lot of development time is required to optimize *coordinated* memory transfers between threads in a block



To know if your algorithm can be GPU-optimized:

1) requires massive parallelism
($>10^3$, ideally $> 10^4$ simultaneous identical calculations)

2) $N_{\text{flop}} \gg N_{\text{memory transfers}}$

3) NB: if only 33% of your algorithm can be parallelized, you can gain at most a factor 3...

GPU computing single vs. double precision

For nVidia cards:

Gflop/s (double) \sim $\frac{1}{2}$ Gflop/s (single)

.... for Tesla / Quadro cards (more expensive)

For consumer cards:

Gflop/s (double) \sim $\frac{1}{4} * \frac{1}{2}$ Gflop/s (single)

Also, the accuracy of some operations is relaxed w/r to IEEE specs

pyCUDA

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

```
func = mod.get function("twice")  
func(a gpu, block=(4,4,1))
```

```
a doubled = numpy.empty like(a)  
cuda.memcpy_dtoh(a doubled, a gpu)  
print a doubled  
print a
```

PyCUDA:
Compilation of the CUDA code is done on-the-fly
Memory transfer to/from GPU is simplified

Ability to perform meta-programming

<http://mathematician.de/software/pycuda>

GPU Crystallography

1) requires massive parallelism
($>10^3$, ideally $> 10^4$ simultaneous identical calculations)

2) $N_{\text{flop}} \gg N_{\text{memory transfers}}$

Crystallography:

Many atoms (N_a)

Many points/reflections (N_{hkl}) in reciprocal space

*$N_a * N_{hkl}$ calculations needed*

=> Ideal case ?

GPU Crystallography

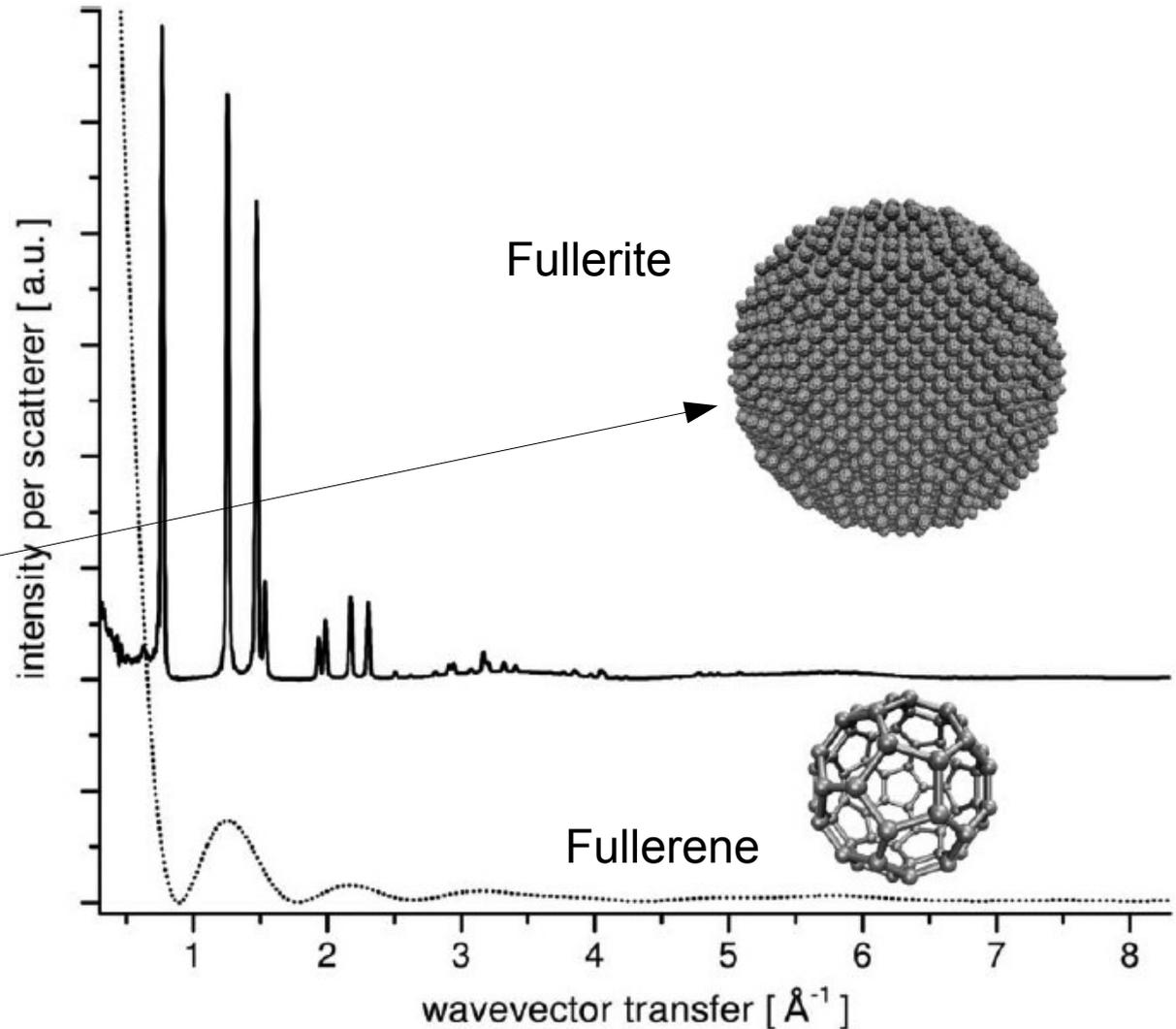
Powder Diffraction

$$I(Q) = \sum_{i=1}^N \sum_{j=1}^N f_i(Q) f_j(Q) \text{sinc}(Qr_{ij}).$$

Calculation of a powder pattern for nanocrystals

$$N_{\text{flop}} \propto N_{\text{atoms}}^2 \cdot N_{\text{points}}$$

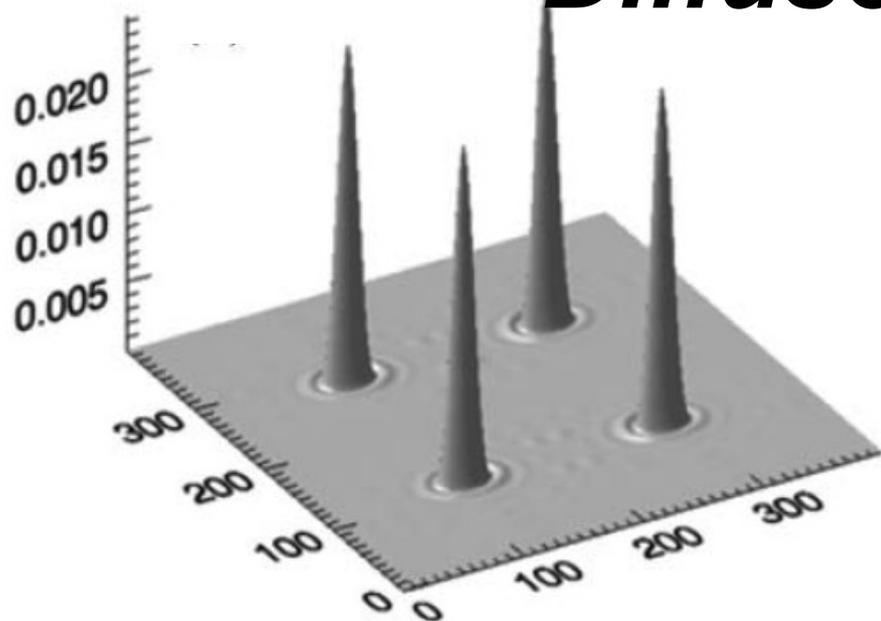
Ex: 30885 atoms,
~ 50s for 3000 points (GTX 295)



Gelasio et al, J. Appl. Cryst. **43** (2010), 647

GPU Crystallography

Diffuse Scattering



**Diffuse Scattering in
YBaCo₄O₇**
(magnetic neutron scattering)

Table 3

Comparison of execution times on the CPU and GPU for using 50 lots for YBaCo₄O₇ averaged over ten runs using single precision.

| Calculation type | Execution time (s) | Speedup |
|------------------|--------------------|---------|
| CPU exact | 4135.92 | 1.0 |
| CPU approximate | 1482.05 | 2.8 |
| GPU approximate | 33.978 | 121.7 |
| GPU exact | 16.935 | 244.2 |

Table 4

Comparison of execution times on the CPU and GPU for using 50 lots for YBaCo₄O₇ averaged over ten runs using double precision.

| Calculation type | Execution time (s) | Speedup |
|------------------|--------------------|---------|
| CPU exact | 4958.39 | 1.0 |
| CPU approximate | 976.80 | 5.1 |
| GPU approximate | 72.929 | 68.0 |
| GPU exact | 62.936 | 78.8 |

Gutmann, *J. Appl. Cryst.* **43** (2010), 250

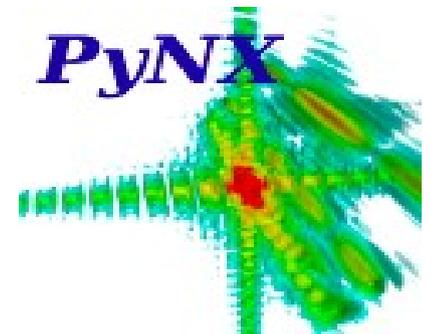
PyNX

GPU computing for nano-Xtallography

Goals:

- Scattering computation
- **Python** interface (no need to learn CUDA)
- Applications to:
 - Grazing-incidence diffraction for epitaxial nano-structures
 - Coherent Bragg Imaging on strained nano-structures

<http://pynx.sf.net>



Scattering computing: FFT vs Direct calculation

Scattering computation

1) Electronic density
within 1 unit cell on
a fine grid

2) FFT

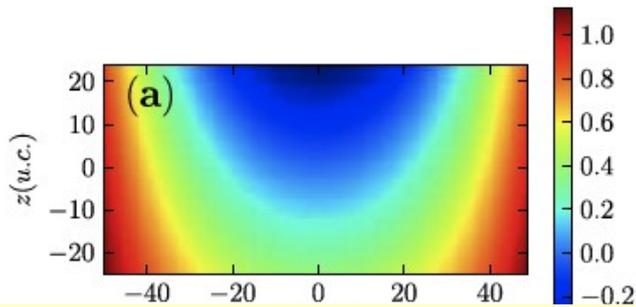
Direct calculation using
atomic positions +
scattering factors

$$A(\vec{s}) = \sum f_i(s) e^{2\pi\vec{s}\cdot\vec{r}_i}$$

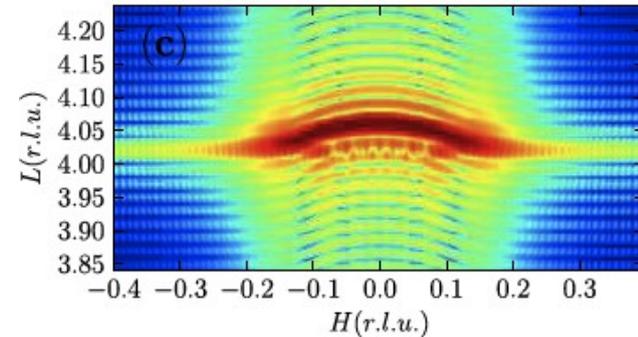
- Used mostly for proteins
- Density calculation is costly for small cells
- FFT is *very* fast
- Useful for large, complete set of reflections
- Coordinates in reciprocal space are imposed by the electron density grid

- For small unit cells
- Any point in reciprocal space can be calculated

Scattering computing: Strained Nano-Structures



Strained Silicon-on-insulator line: u_z
(2D: 50x100 unit cells)



Scattering near (004) reflection

Exact scattering formula:

$$A(\vec{s}) = \sum f_i(s) e^{2\pi \vec{s} \cdot (\vec{r}_i^0 + \vec{u}_i)}$$

\vec{u}_i : displacement w/r perfect lattice

Approximation:

$$A(\vec{s}) = FFT[\rho(i) e^{2i\pi \vec{s}_0 \cdot \vec{u}_i}]$$

Not valid if strongly strained object (hetero-structure)

PyNX: CUDA Code

```
const unsigned long ix=threadIdx.x+blockDim.x*blockIdx.x;
const float h=twopi*vh[ix];
const float k=twopi*vk[ix];
const float l=twopi*vl[ix];
float fr=0,fi=0;
__shared__ float x[BLOCKSIZE];
__shared__ float y[BLOCKSIZE];
__shared__ float z[BLOCKSIZE];
long at=0;
for (;at<=(natoms-BLOCKSIZE);at+=BLOCKSIZE)
{
    x[threadIdx.x]=vx[at+threadIdx.x];
    y[threadIdx.x]=vy[at+threadIdx.x];
    z[threadIdx.x]=vz[at+threadIdx.x];
    __syncthreads();
    for(unsigned int i=0;i<BLOCKSIZE;i++)
    {
        float s,c;
        __sincosf(h*x[i] + k*y[i] + l*z[i] , &s,&c);
        fr +=c;
        fi +=s;
    }
}
```

Thread index

Shared memory between 32 parallel threads
(Fast access, no latency)

“Coalesced” transfer
to shared memory

Each thread computes
a single reflection

Fast, intrinsic trigonometric function

PyNX: Python Interface

Python code:

```
from pynx import gpu  
gpu.Fhk1_thread(h, k, l, x, y, z, occ=occ, gpu_name="295") [0]
```

returns a
complex vector



Calculation distributed
on several GPU

h,k,l,x,y,z,occ:
numpy arrays

Identify GPU
(also works with "CPU", calculation
distributed between cores
using sse_mathfun.h)

4 modules in PyNX:

- **pynx.gpu:** main module for scattering computation
 - Fhk1_thread
- **pynx.gid:** Grazing Incidence Diffraction (uses cctbx)
 - Wave, DistortedWave, Fhk1DWBA4, Fhk1DWBA5
- **pynx.fthomson:** thomson scattering factors
- **pynx.test:** test module

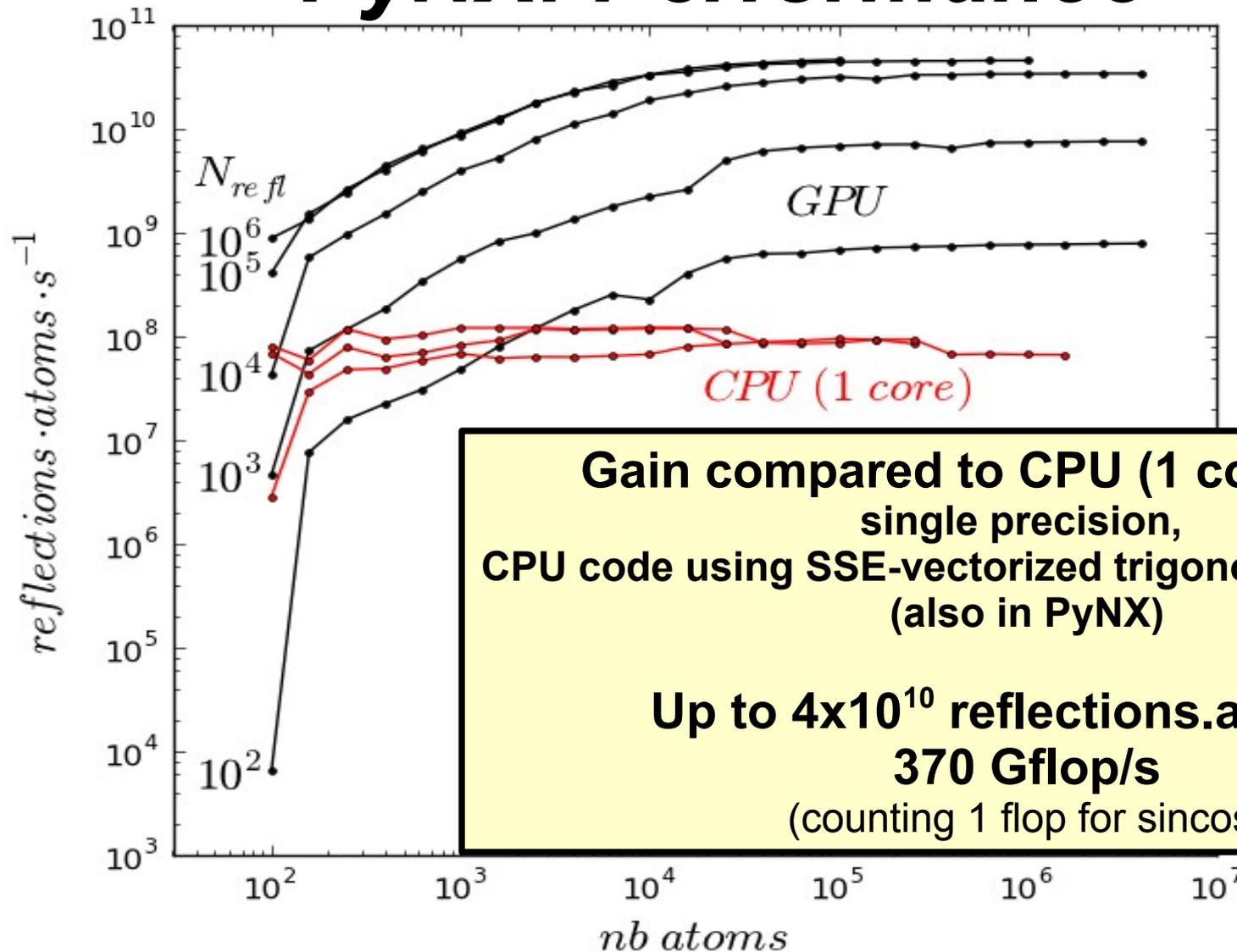
License: **CeCILL-B license** (similar to BSD)

J. Appl. Cryst. **44** (2011),635

<http://pynx.sf.net>

PyNX: Performance

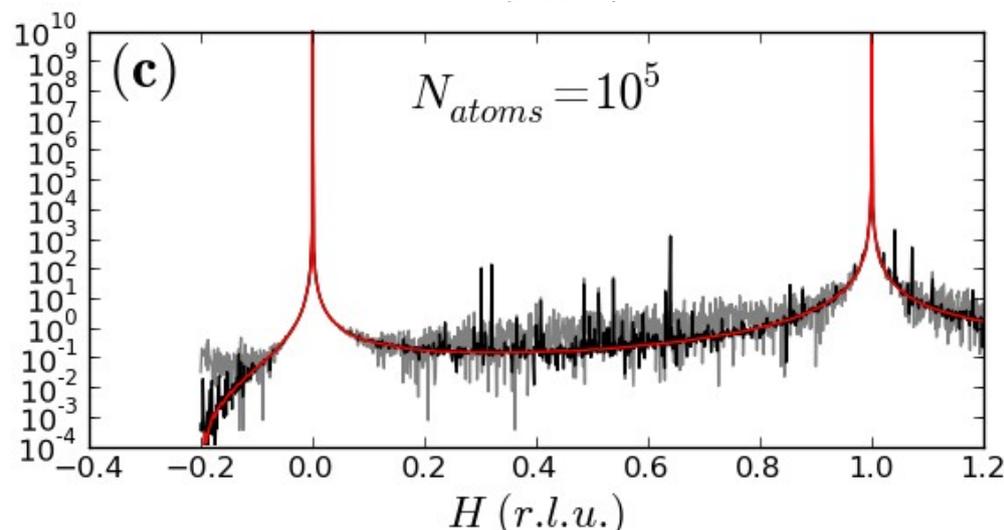
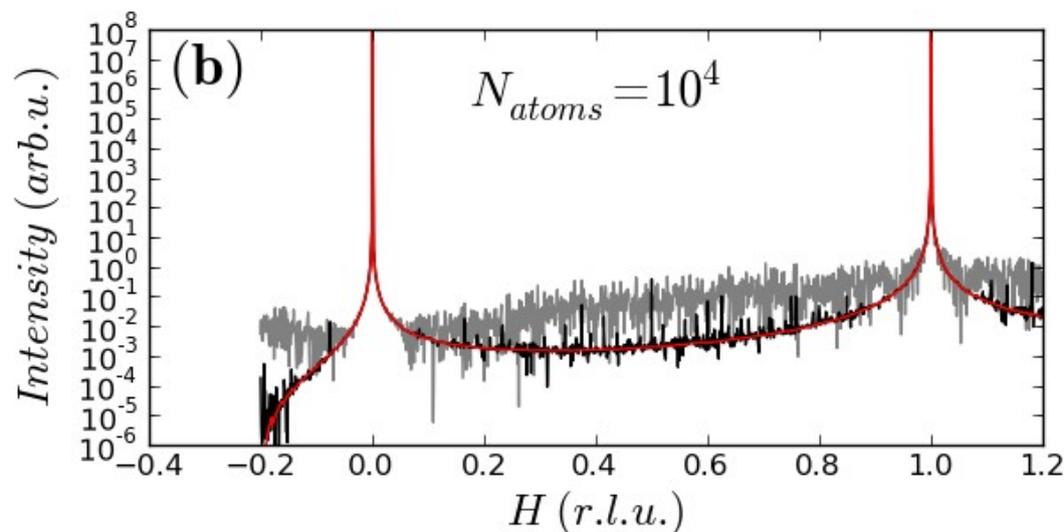
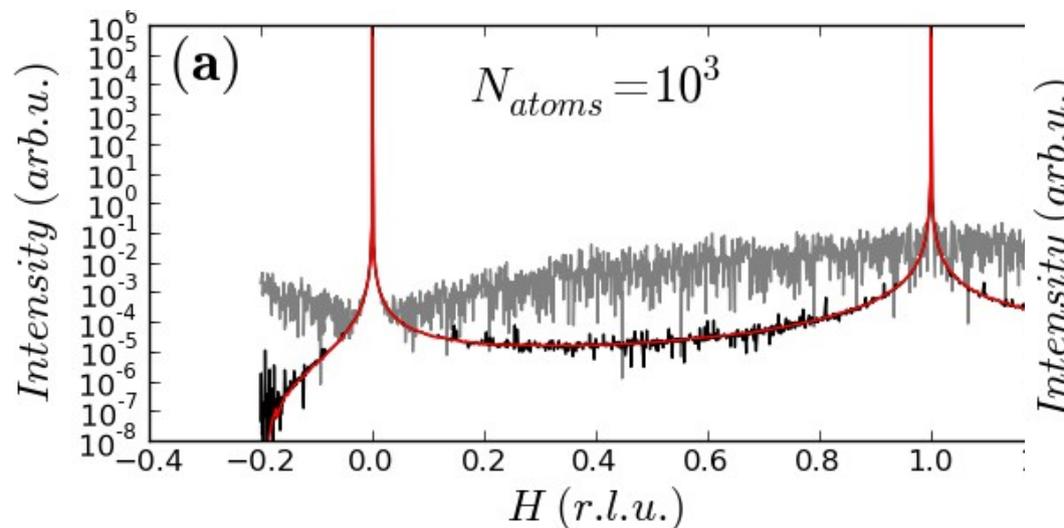
GTX 295



J. Appl. Cryst. **44** (2011),635

<http://pynx.sf.net>

PyNX: Accuracy



Scattering of a linear chain of atoms:
analytical formula
GPU calculation
calculation with atoms randomly
displaced with a sigma=1e-3

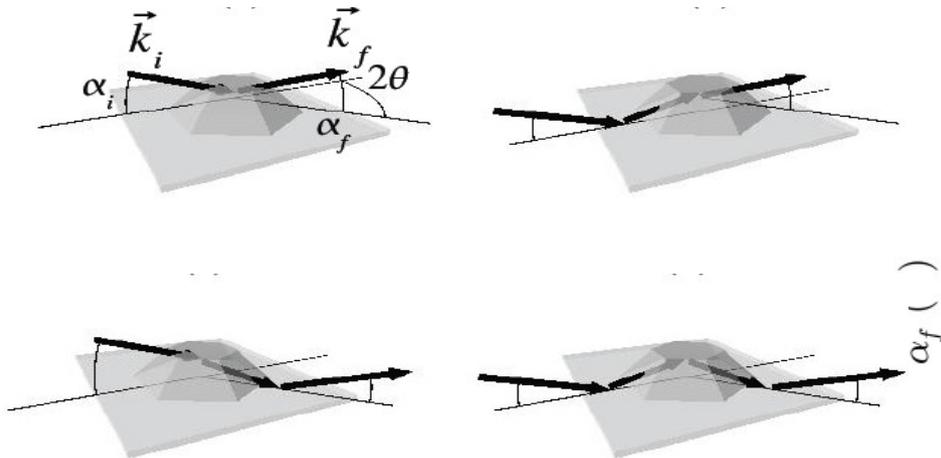
=> Errors due to the single precision
floating point calculations are visible.
... But for 'real' crystals this will
probably be below the background/TDS

J. Appl. Cryst. **44** (2011), 635

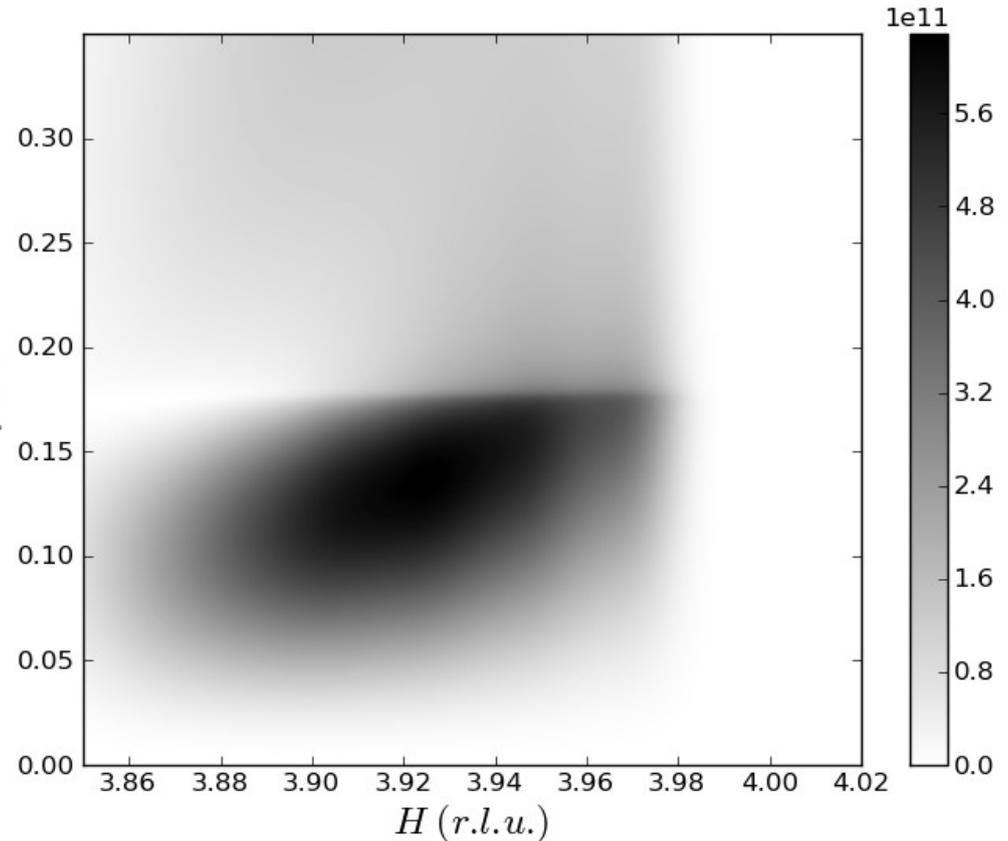
<http://pynx.sf.net>

PyNX: Example

Grazing Incidence Diffraction

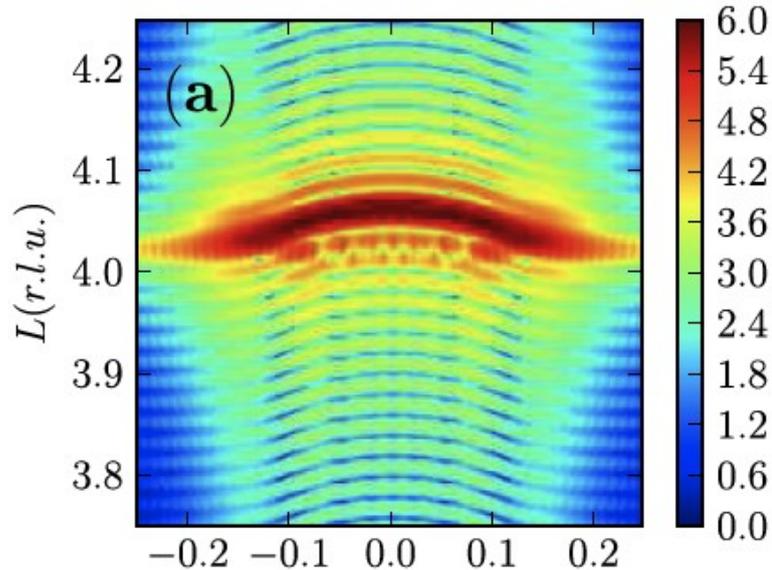


Distorted Wave Born Approximation
=> sum 4 paths
(electronic density of material calculated
using cctbx)

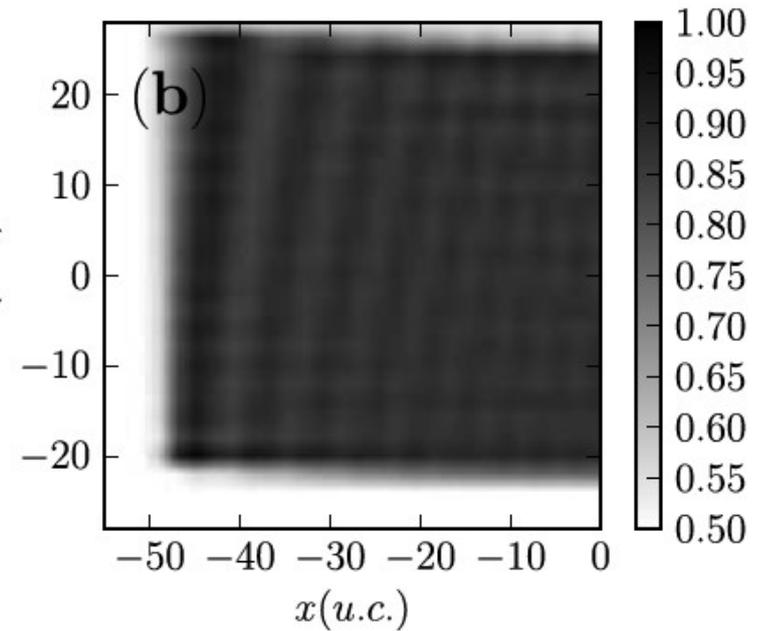
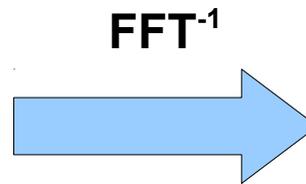


J. Appl. Cryst. **44** (2011),635
<http://pynx.sf.net>

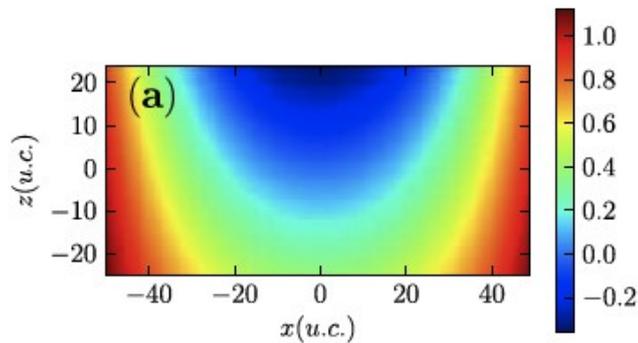
Coherent Bragg Imaging



Scattering around (004)

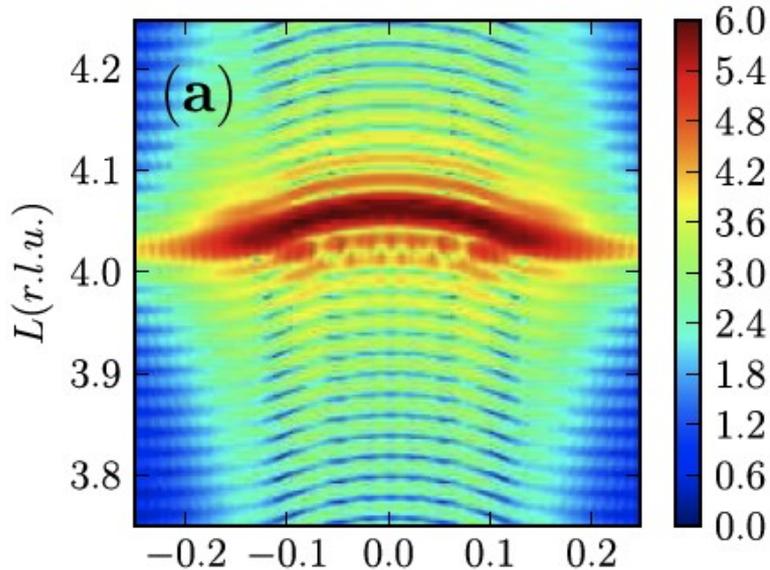


Using a FFT creates 'ripples'



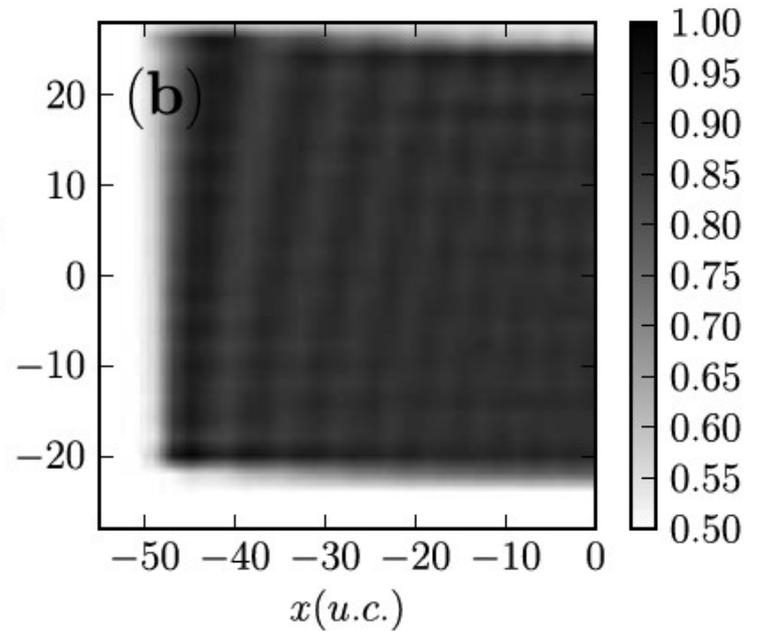
Strained silicon line

Coherent Bragg Imaging

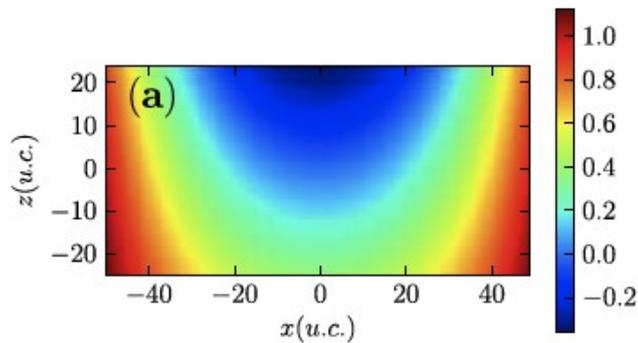


Scattering around (004)

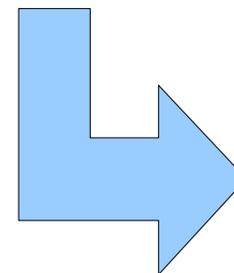
FFT⁻¹



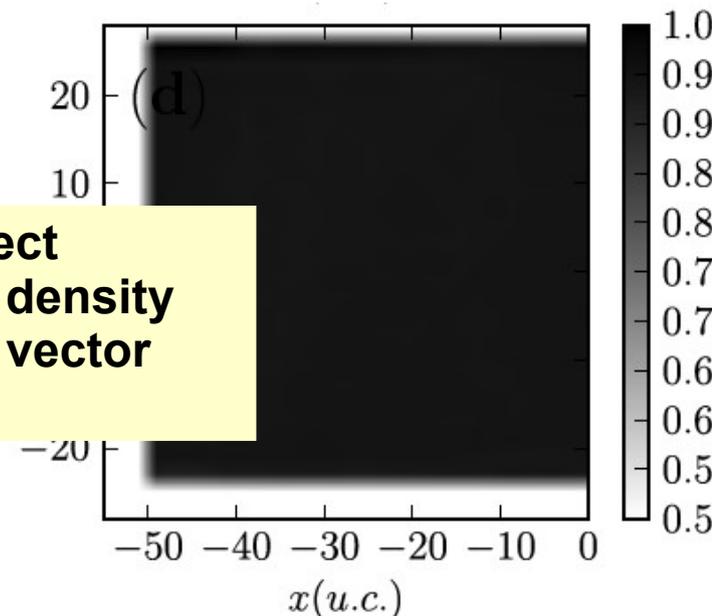
Using a FFT creates 'ripples'



Strained silicon line



By using a direct minimization of the density And displacement vector using GPU



Outlook: Dependency vs Vendor ?

Outlook: Dependency vs Vendor ?

=> OpenCL

- **GPU computing depends on graphic cards vendor to provide the API for GPU computation**
- **Originally, only nVidia cards (CUDA toolkit)**
- **Now also AMD Radeon cards**

- ***Is the language durable ?***

- **New language: OpenCL** <http://www.khronos.org/opencl/>
 - **Compatible with nVidia *and* AMD cards**
 - **Standard**
 - **PyOpenCL interface :-)**
 - **Same performance as CUDA**

Outlook: Crystallography "On-A-Chip"

Parallel code, 48 kb data ?

Develop code exploiting GPU, *without* any memory transfer ?

For Fermi nVidia cards:

- **Amount of shared memory per multiprocessor: 48 kb**
(+16kb L1 cache & 768kb L2 cache)
- **Number of registers per multiprocessor: 32k**

=> Ab initio Structure solution in real space ?

=> other applications ?

Acknowledgements & Bibliography

Beamlines:

- **BM02 (DAFS, GIXS, GI-MAD): JF Bérrar, N. Boudet, B. Caillot, S. Arnaud**
- **BM32 (GIXS, GI-MAD): F. Rieutord, JS Micha, O. Ulrich**
- **ID01 (GIXS, GI-MAD): T. Metzger, B. Krause, D. Carbone, O. Bikondoa**

Main contributors to the resonant experiments:

- **CEA Grenoble / INAC: B. Daudin, T. Schüllli, G. Renaud**
- **Grenoble Inp: H. Renevier**
- **U. Zaragoza (Spain): MG Proeitti**
- **Inst. Néel: J. Coraux**
- **INSA Rennes: A. Létoublon**
- **Univ. Marseille: M-I Richard**