:
Recent development and status of beam physics codes for heterogenous platform

R. De Maria, G. Iadarola

Particles beam physics codes

Simulate particle trajectories inside accelerator structures

Physics of the simulations

Relativistic Newton Law: $\dot{p} = m\,\gamma F$

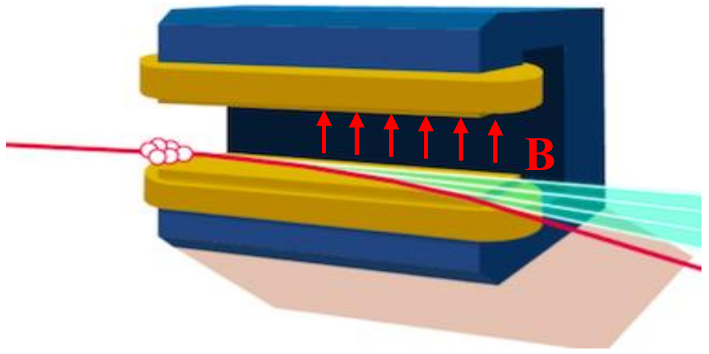
Lorenz force: $F = q(\vec{E} + v \times \vec{B})$

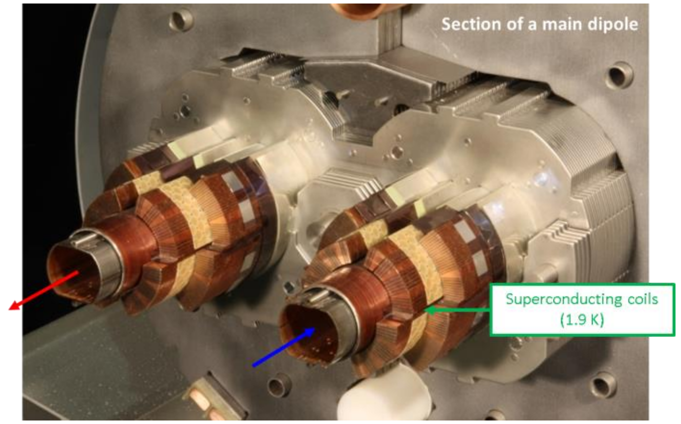
Maxwell Equations:

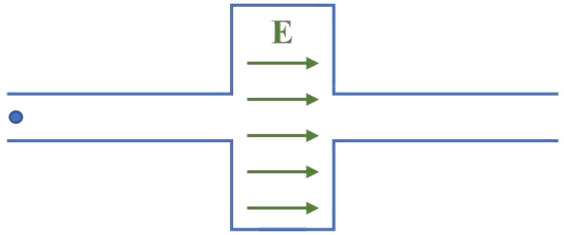
$\nabla \cdot \vec{\mathrm{B}} = 0; \nabla \cdot \vec{\mathrm{E}} = \rho/\epsilon_0;$

$\nabla \times \vec{E} + \frac{\partial \vec{B}}{\partial t} = 0; \nabla \times \vec{B} + \frac{1}{c^2}\frac{\partial \vec{E}}{\partial t} = \mu \vec{J}$

Synchrotron radiation…
Particle matter interactions…
Secondary electron emission…



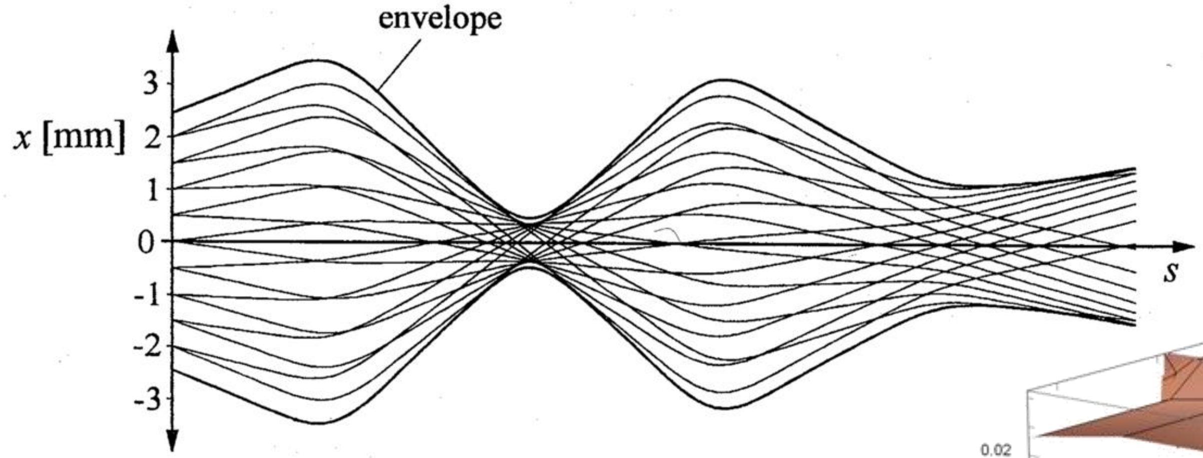


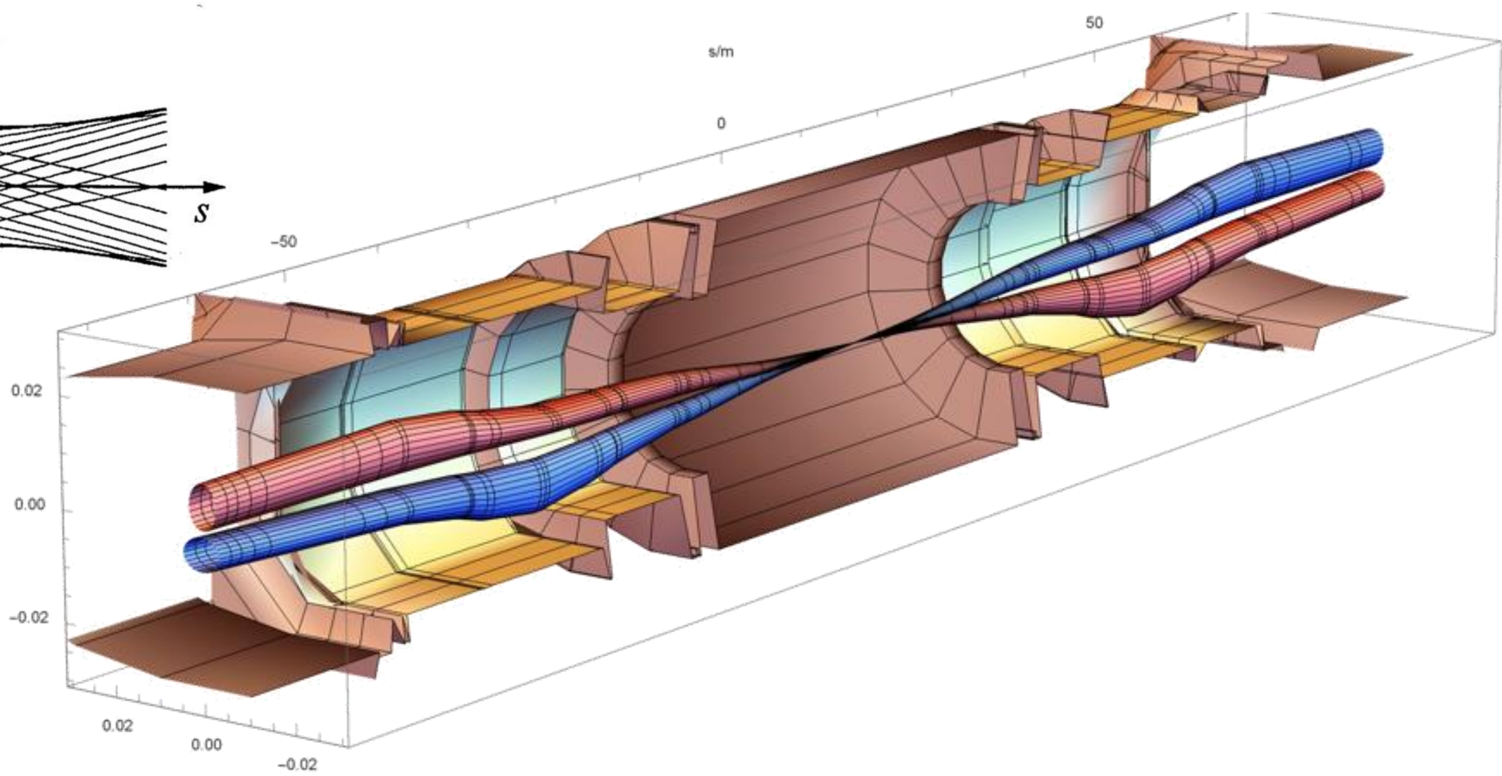


Accelerating structure

Type of simulations (1)

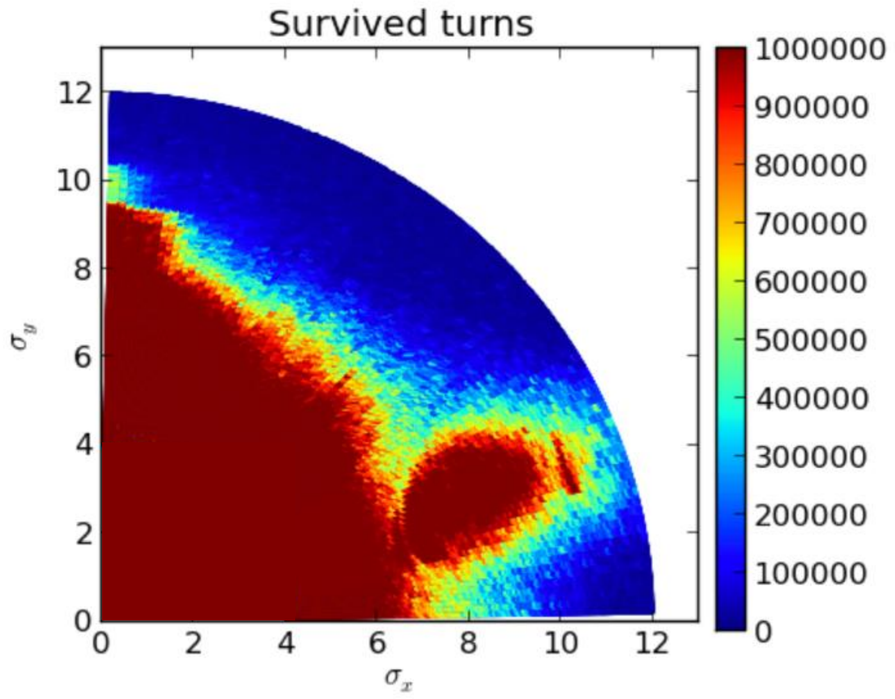
<u>Single particle for one turn</u>: follows one particle and studies the perturbation of the motion.





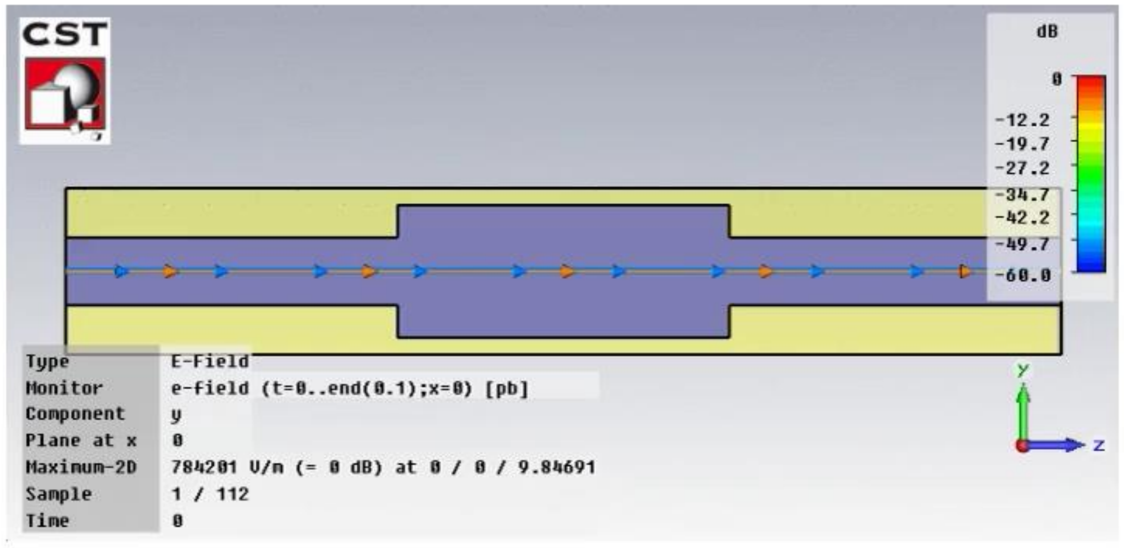
Type of simulations (2)

Many single particles: tracks many particles for many turns to understand the long-term stability and the beam lifetime.



Type of simulations

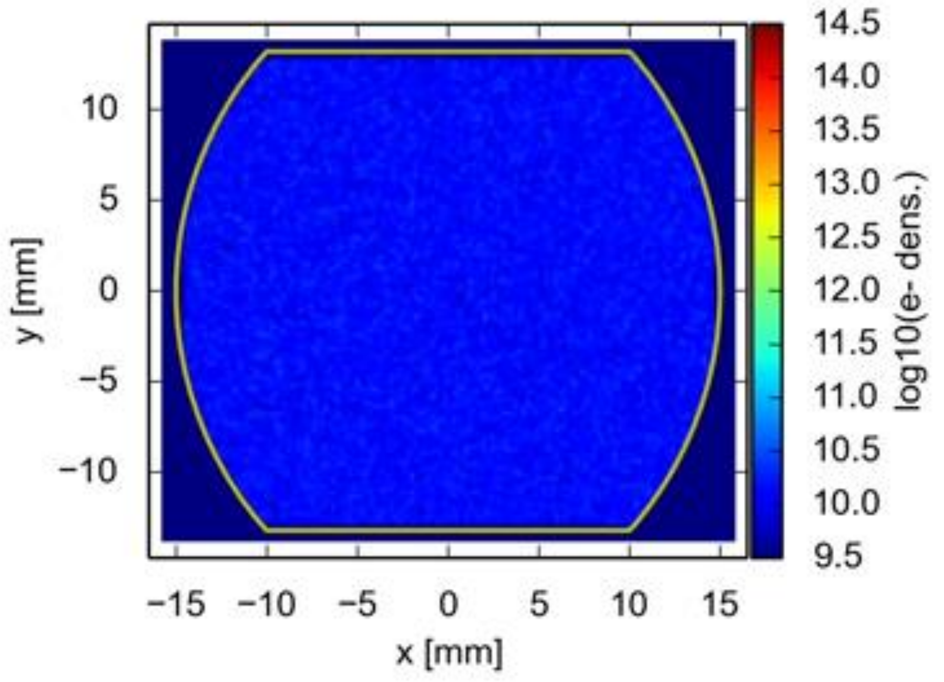
<u>Many interacting particles</u>: tracks many particles and compute the interaction between particles and with the e.m. field surrounding them.

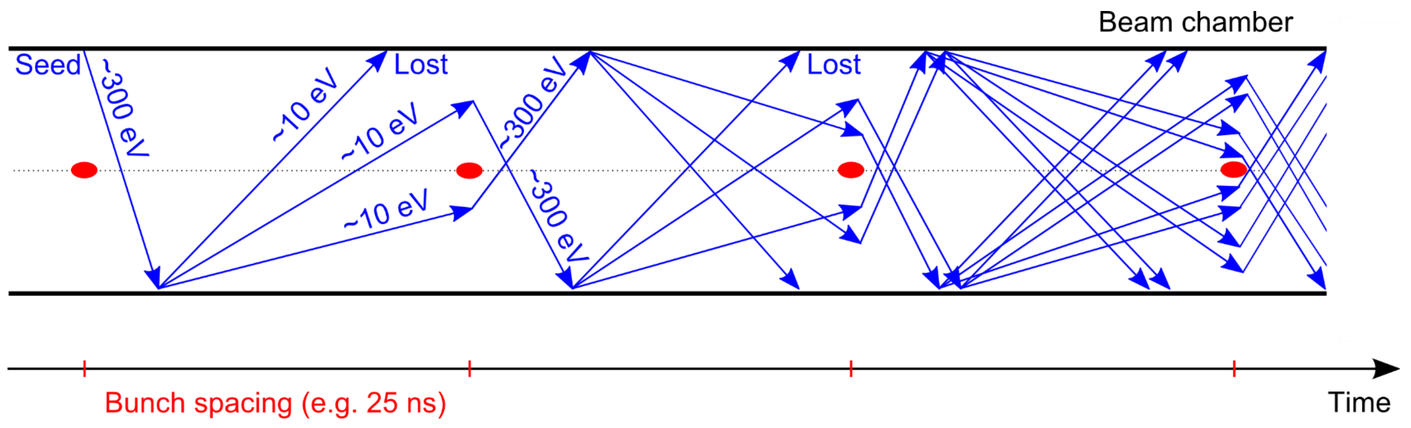


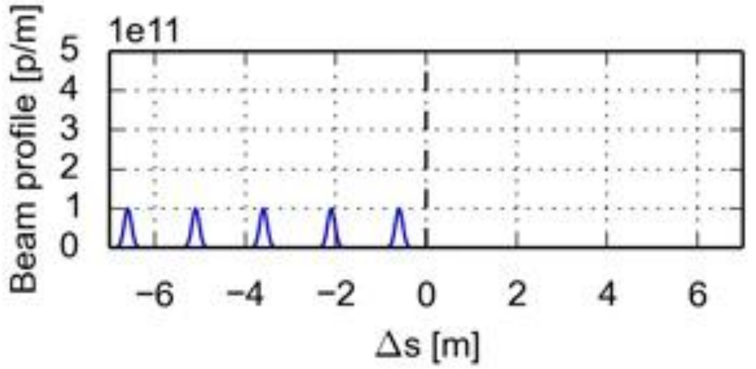
```
Type          E-Field
Monitor       e-field (t=0..end(0.1);x=0) [pb]
Component     y
Plane at x    0
Maximum-2D    784201 V/m (= 0 dB) at 0 / 0 / 9.84691
Sample        1 / 112
Time          0
```

Type of simulations

Particle matter interactions: track many particle and compute the interaction with matter such as electron extraction from surfaces, elastic and inelastic interaction with collimators or residual gasses



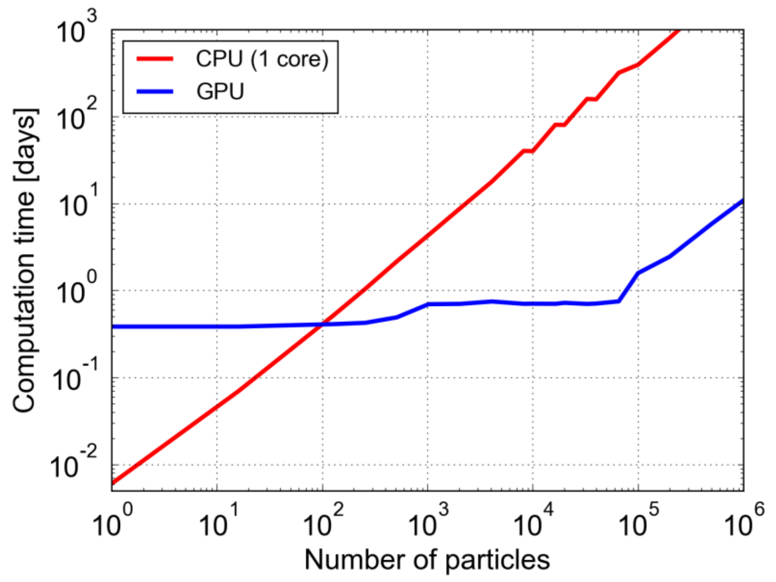


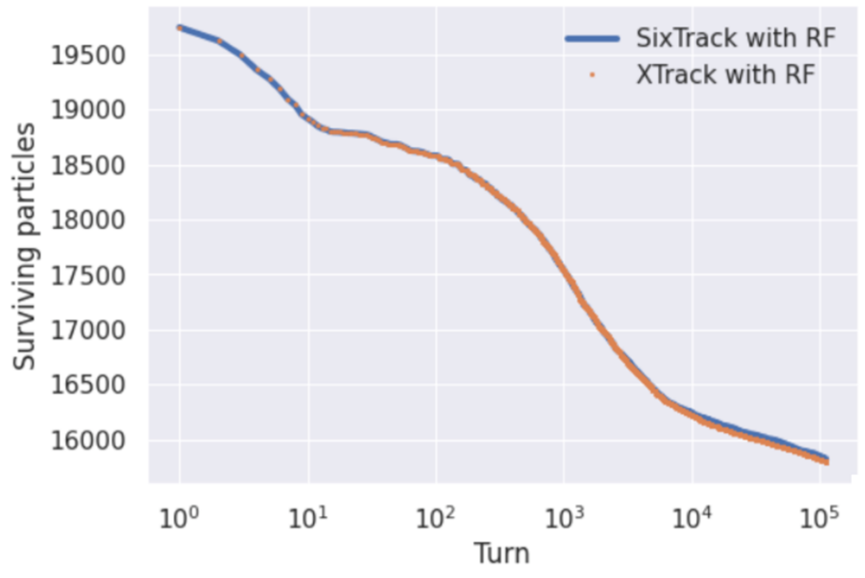


GPU computing on Beam Physics codes

GPU are very attractive because many simulations are embarrassingly parallel and often without branching.

Even in case of interactions, electromagnetic solvers (e.g. FFT based) can be also highly efficient on GPUs.





Test run (10 turns)

Setup	Tracking time [s]
SixTrack without K2	40
XTrack without K2 (GPU)	0.3

Realistic study (parametric scan)

	Simulated time interval	Number of jobs	Time needed*
Xtrack (GPU)	10s	400	~ 24 h
SixTrack	1s	40000	~ 7 days

* CPUs and GPUs in HTCondor

Code development needs in ABP

Context

- Codes to be continuously tailored for specific studies (generic codes would be too slow)
- Staff develop very often part-time, students/fellows often with no strong computing background, numerical methods in the field not well documented.
- High throughput is needed to study large parameter spaces.
- Low latency needed for decision taking studies.
- GPUs provide more resources and/or enable computations otherwise not possible.

Requirements

- Rapid developing environment based on popular and simple programming language with large library ecosystem.
- Code immediately runnable on CPU and NVidia, AMD, Intel GPU (hardware is available today in user machine and datacenters).
- First implementation needs to achieve close to maximum performance. Often no time left for optimizations.

Code structure and technology choices

- Code organized in Python packages.
- User interactions via Python scripts.
- Data structures described, allocated and fully exposed (r/w) in Python including GPU (avoid unnecessary copy).
- Performance critical code written in C with using automatically generated C-API from Python.
- Code compiled at run time and on-demand:
    - Allows problem specific optimizations essential on GPU
    - Reduce writing, testing, executing cycle
- Dependencies:
    - numpy: allocate and exchange memory
    - cffi (and a C compiler): generates binary Python modules that can be imported at run time and prepare arguments
    - cupy (and a cuda driver): implements rich numpy-like array on device and compiles cuda kernels
    - pyopencl (and OpenCL drivers): wraps OpenCL API and implement a basic numpy-like array

OpenCL status

Support of OpenCL is one of the reason we write C instead of C++ code (other being simplicity).

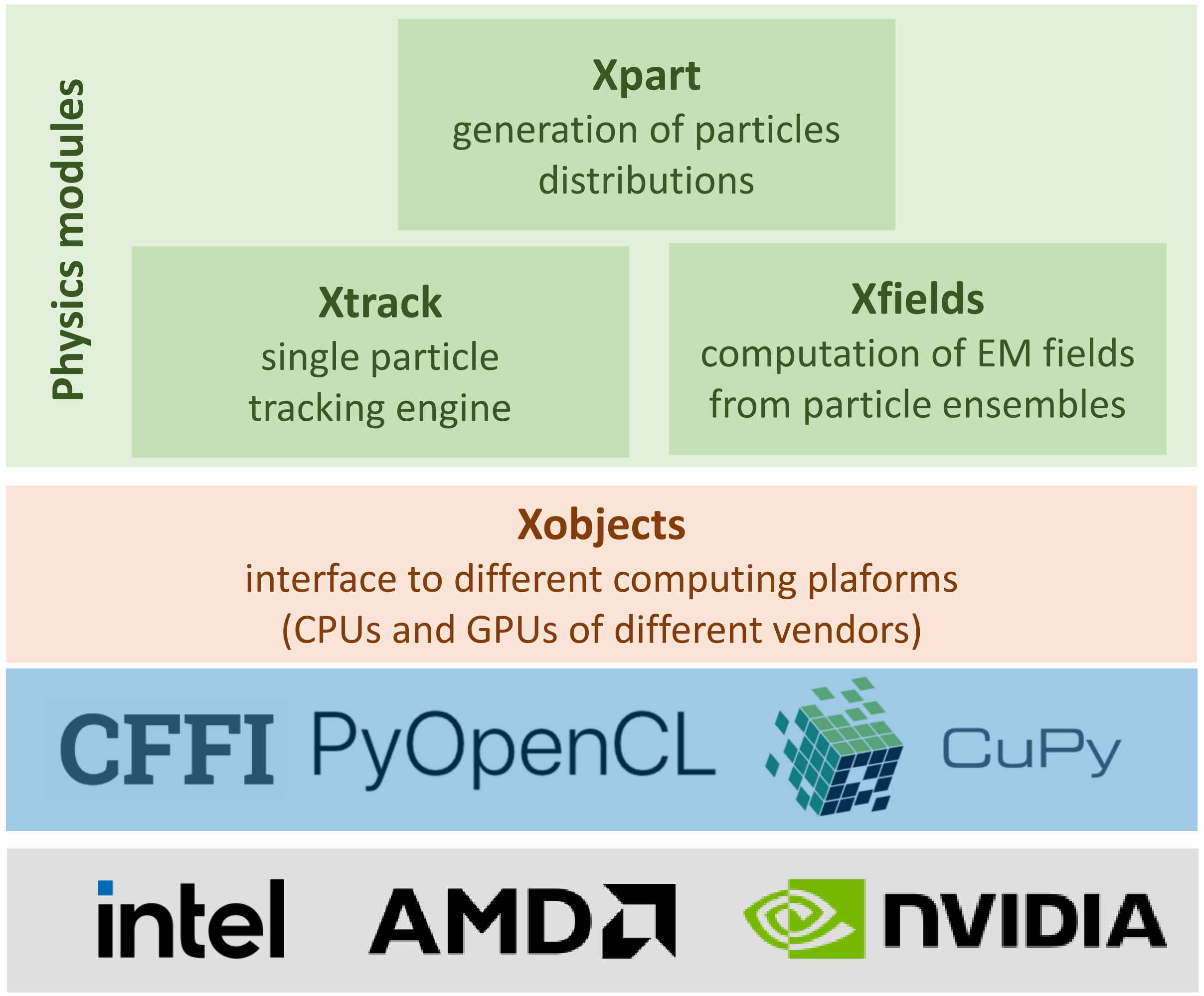
OpenCL 1.2 has been stable and supported since 2011.

Practically any hardware has both a proprietary and an open-source implementation.

Device language	Cuda	HIP	Sycl	OpenCL	Python
Intel/AMD CPU	🟥 No	🟥 No	🟩 OneAPI	🟩 POCL, OneAPI	🟩 Numba
Intel GPU	🟥 No	🟥 No	🟩 OneAPI	🟩 Intel-compute, mesa	🟥 No
Nvidia GPU	🟩 Yes	🟨 Rocm	🟨 OneAPI, computeCPP	🟩 Nvidia-driver, mesa, POCL	🟩 Numba
AMD GPU (navi)	🟥 No	🟥 Promised	🟥 No	🟩 AMDGPU, mesa	🟥 No
AMD GPU (vega, cdna)	🟨 hipify	🟩 Rocm	🟨 hipsycl	🟩 AMDGPU, ROCM, mesa	🟨 Numba
Qualcom Adreno, ARM Mali, PowerVR	🟥 No	🟥 No	🟥 No	🟩 Android	🟥 No

Despite many claims, developments continue well in 2021: 1.2 support in mesa, 3.0 support intel, nvidia.
OpenCL does not look is going away, best protection against vendor lock-in…

Status after 1 year of development


Physics modules
Xpart
generation of particles distributions
Xtrack
single particle tracking engine
Xfields
computation of EM fields from particle ensembles
Xobjects
interface to different computing plaforms (CPUs and GPUs of different vendors)
CFFI
PyOpenCL
CuPy
intel
AMD
NVIDIA

Main goal:

- fast development cycle
- easy to experiment with different data structures
- student friendly

Resource invested in 2021:

- Infrastructure: 2 devs for 0.5 FTE
- Code porting: 2 devs+ about 5 experts for 0.5 FTE

Results obtained:

- Porting about 20 users to new codes.
- Converting 150 kLOC into 10 kLOC (code) + 10kLOC (test + examples)

http://github.com/xsuite
https://indico.cern.ch/event/1076583/contributions/4527801

Xobject structure

Types: define memory layout, allocate data in buffer, used by C-API generator	C-API generator Struct, Array, Ref, Union, UnionRef: Building blocks to define custom data structures		
Contexts: create buffers and kernels from C annotated sources	ContextCpu: Holds compiler settings	ContextPyopencl: Holds OpenCL context	ContextCupy: Hold cuda device
Kernels: prepare function arguments, run code, convert results	KernelCpu: Holds cffi functions	KernelPyopencl: Holds OpenCL kernels	KernelCupy: Hold Cuda kernels
Buffers: growable byte blob with memory allocator	BufferNumpy: int8 numpy array	BufferPyopencl: OpenCL buffer	BufferCupy: int8 cupy array

Xobjects – data manipulation in Python

The main features of Xobjects can be illustrated with a simple **example** (Xsuite physics packages are largely based on the features illustrated here)

A **Xobjects Class** can be defined as follows:

```python
import xobjects as xo

class DataStructure(xo.Struct):
    a = xo.Float64[:] # Dynamic Array
    b = xo.Float64[:] # Dynamic Array
    c = xo.Float64[:] # Dynamic Array
    s = xo.Float64    # Scalar
```

Nested classes, tagged unions and references also supported…
Dynamic shapes frozen after allocation.

An **instance of our class** can be instantiated on CPU or GPU by passing the appropriate context

```python
# ctx = xo.ContextCpu()
ctx = xo.ContextCupy() # for NVIDIA GPUs

obj = DataStructure(_context=ctx,
                    a=[1,2,3], b=[4,5,6],
                    c=[0,0,0], s=0)
```

Encourage data allocation on device only. Not unusual a 2 GB host to control 40 GB GPU

Independently on the context, the **object is accessible in read/write directly from Python**. For example:

```python
print(obj.a[2]) # gives: 3
obj.a[2] = 10
print(obj.a[2]) # gives: 10
```

Essential for debugging, shortcut for one-time configuration before expensive calculations

Xobjects – data access from C

The definition of a Xobject class in Python, **allows the generation of a set of functions (C-API)** that can be used in C code to access the data.

They can be inspected by:

```
print(DataStructure._gen_c_decl(conf={}))
```

which gives (without the comments):

```
// ...

// Get the length of the array DataStructure.a
int64_t DataStructure_len_a(DataStructure obj);

// Get a pointer to the array DataStructure.a
ArrNFloat64 DataStructure_getp_a(DataStructure obj);

// Get an element of the array DataStructure.a
double DataStructure_get_a(const DataStructure obj, int64_t i0);

// Set an element of the array DataStructure.a
void DataStructure_set_a(DataStructure obj, int64_t i0, double value);

// get a pointer to an element of the array DataStructure.a
double DataStructure_getp1_a(const DataStructure obj, int64_t i0);

// ... similarly for b, c and s
```

```
# From before
class DataStructure(xo.Struct):
    a = xo.Float64[:]
    b = xo.Float64[:]
    c = xo.Float64[:]
    s = xo.Float64

# ctx = xo.ContextCpu() # CPU
ctx = xo.ContextCupy()  # GPU

obj = DataStructure(_context=ctx,
            a=[1,2,3], b=[4,5,6],
            c=[0,0,0], s=0)
```

Concise, stable and user-friendly API:

```
<typename>_<op>_<name1>_...(<root>,<index1>, [<value>])
```

More to come…

Xobjects – writing cross-platform C code

A **C function that can be parallelized when running** on GPU is called "Kernel".

Example: C function that computes obj.c = obj.a * obj.b

```
src = '''
/*gpukern*/
void myprod(DataStructure ob, int nelem){
   for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem
      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);
   }//end_vectorize
}
'''
```

(Comments in red are Xobjects annotation, defining how to parallelize the code on GPU)

```
# From before
class DataStructure(xo.Struct):
    a = xo.Float64[:]
    b = xo.Float64[:]
    c = xo.Float64[:]
    s = xo.Float64

# ctx = xo.ContextCpu() # CPU
ctx = xo.ContextCupy()  # GPU

obj = DataStructure(_context=ctx,
          a=[1,2,3], b=[4,5,6],
          c=[0,0,0], s=0)
```

The Xobjects context compiles the function from Python:

```
ctx.add_kernels(
    sources=[src],
    kernels={'myprod': xo.Kernel(
                args = [xo.Arg(DataStructure, name='ob'),
                        xo.Arg(xo.Int32, name='nelem')],
                n_threads='nelem')
            } )
```

Potentially unnecessary step if we bother parsing C (pycparser could help but…)

The kernel can be easily called from Python and is executed on CPU or GPU based on the context:

```
# obj.a contains [3., 4., 5.] , obj.b contains [4., 5., 6.]
ctx.kernels.myprod(ob=obj, nelem=len(obj.a))
# obj.c contains [12., 20., 30.]
```

Xobjects – code specialization

Before compiling, Xobjects **specializes the code** for the chosen computing platform.

- Specialization and compilation of the C code are **done at runtime** through Python

Code written by the user

```
/*gpukern*/ void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  }//end_vectorize
}
```

Code specialized for CPU

```
void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //autovectorized

     double a_ii = DataStructure_get_a(ob, ii);
     double b_ii = DataStructure_get_b(ob, ii);
     double c_ii = a_ii * b_ii;
     DataStructure_set_c(ob, ii, c_ii);

  }//end autovectorized
}
```

Could use also OpenMP

Code specialized for GPU (OpenCL)

```
__kernel void myprod(DataStructure ob, int nelem){

  int ii; //autovectorized
  ii=get_global_id(0); //autovectorized

     double a_ii = DataStructure_get_a(ob, ii);
     double b_ii = DataStructure_get_b(ob, ii);
     double c_ii = a_ii * b_ii;
     DataStructure_set_c(ob, ii, c_ii);

  //end autovectorized
}
```

Xobjects – code specialization

Before compiling, Xobjects **specializes the code** for the chosen computing platform.

- Specialization and compilation of the C code are **done at runtime** through Python

Code written by the user

```
/*gpukern*/ void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  }//end_vectorize
}
```

Code specialized for CPU

```
void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //autovectorized

     double a_ii = DataStructure_get_a(ob, ii);
     double b_ii = DataStructure_get_b(ob, ii);
     double c_ii = a_ii * b_ii;
     DataStructure_set_c(ob, ii, c_ii);

  }//end autovectorized
}
```

Could use also OpenMP

Code specialized for GPU (Cuda)

```
__global__ void myprod(DataStructure ob, int nelem){
    int ii; //autovectorized
    ii=blockDim.x * blockIdx.x + threadIdx.x; //au
    if (ii<nelem){

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

   }//end autovectorized
}
```

Xobjects – Data types and API

Array of array of struct

```python
class Point(xo.Struct):
    x = xo.Float64
    y = xo.Float64

class Triangle(Point[3]):
    pass

class Mesh(Triangle[:]):
    pass
```

C-API stable against some data layout changes.

Array of struct of array

```python
class Triangle(xo.Struct):
    x = xo.Float64[3]
    y = xo.Float64[3]

class Mesh(Triangle[:]):
    pass
```

Struct of array of array

```python
class Mesh(xo.Struct):
    x = xo.Float64[:,3]
    y = xo.Float64[:,3]
```

Struct of array of array

```python
class Mesh(xo.Struct):
    x = xo.Float64[3:1,::0]
    y = xo.Float64[3:1,::0]
```

API generated

```c
/*gpufun*/
 double Mesh_get_x(
        const Mesh/*restrict*/ obj,
        int64_t i0,
        int64_t i1){
  int64_t offset=0;
  offset+=16+i0*48;
  offset+=i1*16;
  return *(/*gpuglmem*/double*)((/*gpuglmem*/char*) obj+offset);
}
```

```c
/*gpufun*/
 double Mesh_get_x(
        const Mesh/*restrict*/ obj,
        int64_t i0,
        int64_t i1){
  int64_t offset=0;
  offset+=16+i0*72;
  offset+=i1*8;
  return *(/*gpuglmem*/double*)((/*gpuglmem*/char*) obj+offset);
}
```

Conclusions

GPUs open computing capacity and enable new computation otherwise prohibitive.

Developing portable code is extremely challenging:

- Multiple incompatible languages and framework, vendors not helping, but rather increasing fragmentation

In ABP we decided to:

- Leverage Python and ecosystem for user API, and low-level code C generation.
- Support OpenMP, Cuda and OpenCL allows to cover and exploit existing resources.
- In 2021 we invested half time in framework and half in porting code and we obtained performance not lower than (our) hand made code and functionality coverage.

Challenges:

- Will the approach be future proof?
- Will we manage to keep the cost of the framework low for more complex code?
- Are we maximizing performance given the effort?