Collective Effects in Beam Tracking

GPU Parallelisation of Direct Space Charge Simulations in PyHEADTAIL and PyPIC

Adrian Oeftiger, Ph.D. student in

BE-ABP-HSC section / Space Charge Working Group

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

GPU Computing Meeting, CERN

12. January 2016

Outline

1. Collective Effects: Direct Space Charge
2. GPU accelerated: PyPIC and PyHEADTAIL
3. Further GPU Studies in BE-ABP

keywords: N-body simulations, particle-in-cell algorithm, PyCUDA

Introduction

collective beam dynamics $\longleftrightarrow$ N-body simulations

central question

stability of charged particle beams

some numbers:

- beams modelled with $\sim 10^6$ *macro-particles*
- in principle, 10^5 to 10^6 revolutions in accelerator ring
- usually 10^3 integration steps per revolution required

$\longrightarrow$ similar to models and dynamics in astrophysics, cosmology, plasma physics, ...

Motivation

physics issue

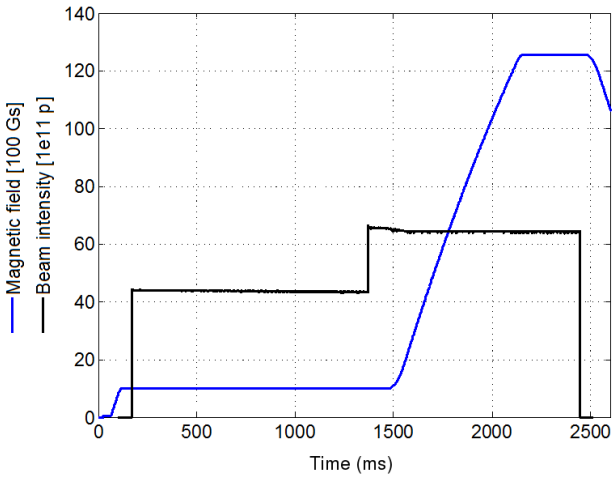
LHC Injectors Upgrade (LIU):

- increase beam intensity by **2x** for HL-LHC project

⟹ stronger beam self-fields (i.e. *space charge*)

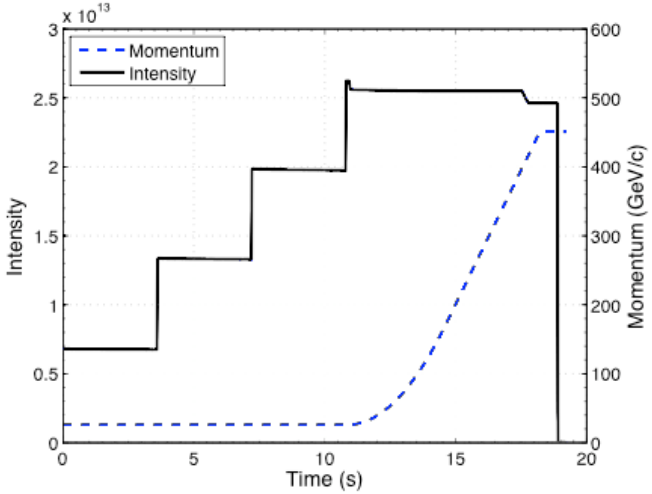
⟹ can lead to resonances: losses and beam degradation

⟹ PS and SPS injection plateaus affected



PS cycle

SPS cycle



Motivation

physics issue

LHC Injectors Upgrade (LIU):

- increase beam intensity by **2x** for HL-LHC project

software issues

expensive self-consistent space charge simulations:

- simulation execution time vs. simulated time

$\longrightarrow$ SPS: running $\mathcal{O}(1\text{ week})$ for 1 s cycle time, need 10.8 s

- growing artificial noise affects simulation validity

$\longrightarrow$ SPS: simulations with accessible numeric parameters are valid for $\mathcal{O}(10^4\,\text{turns})$, injection plateau $= 5 \times 10^5$ turns

Motivation

physics issue

LHC Injectors Upgrade (LIU):

- increase beam intensity by **2x** for HL-LHC project

software issues

expensive self-consistent space charge simulations:

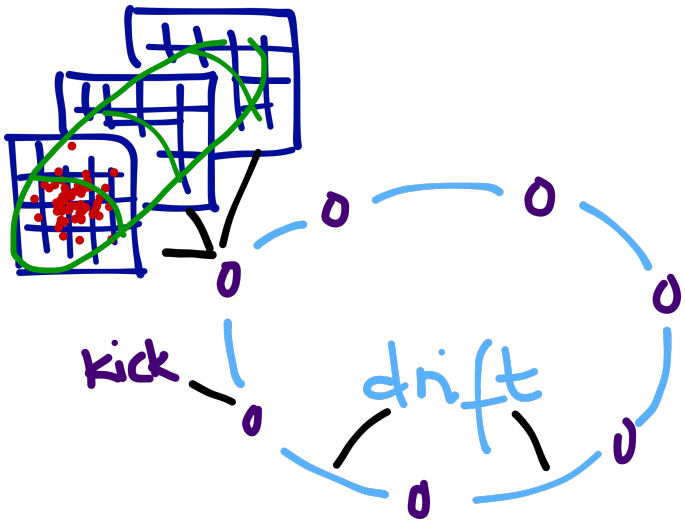
- simulation execution time vs. simulated time
- growing artificial noise affects simulation validity

our solution: PyHEADTAIL and PyPIC

GPU accelerated simulation framework addresses software issues and allows investigating physics at PS / SPS injection

Collective Effects

How-to: collective effects,
drift-kick model:



- treat single-particle separately from multi-particle dynamics
- push all single particles through "drift" to next interaction point
- evaluate multi-particle interaction integrated over drift
- apply interaction as "kick" to all particles

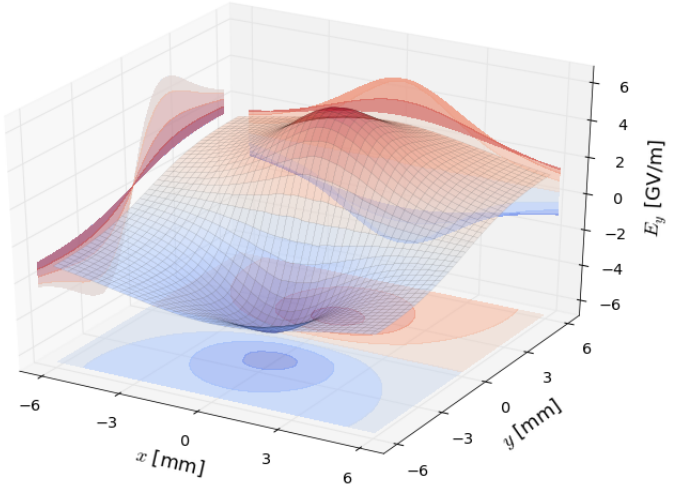
$\longrightarrow$ usually coarsen distribution to evaluate interaction strength (e.g. particle-mesh methods)

Direct Space Charge Modelling

Space charge in accelerator rings:

- Lorentz-boost to beam frame

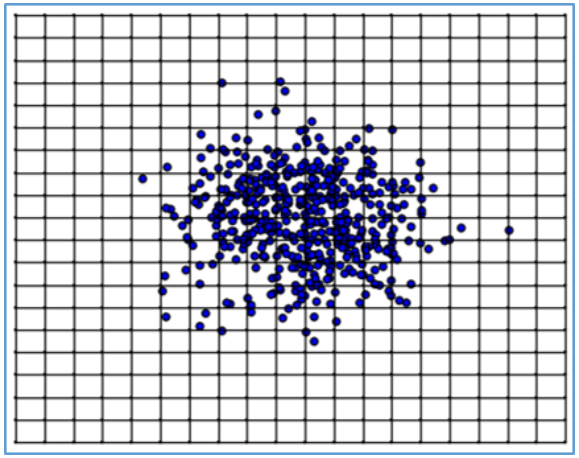
$\Longrightarrow$ electrostatic problem

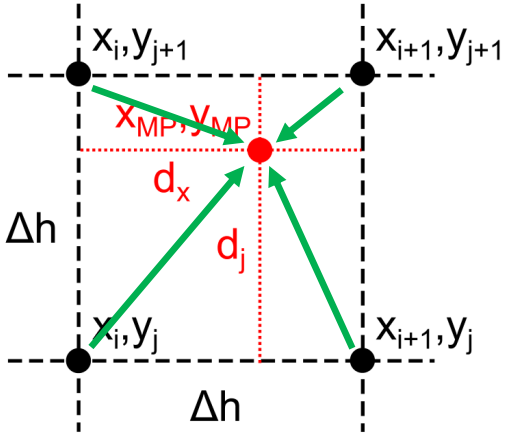


- evaluate beam fields by solving *Poisson's equation* for macro-particle distribution
  1. particle-to-particle: extremely slow
  2. Fast Multipole Method: exaggerated binary collisions of macro-particles need special care
  3. particle-mesh methods: **particle-in-cell** (PIC) algorithm is de facto standard (cell size = smoothing effect)
  4. (...)
- apply electric repulsion forces to all particles

Particle-in-cell Algorithm

1. **particles to mesh**: deposit all macro-particle charges onto (regularly distributed) mesh nodes
2. **solve** discretised Poisson equation on the mesh, options:
   - direct solving, e.g. via sparse matrices
   - spectral methods
   - Hockney's algorithm $\Longrightarrow$ 'cheap' FFT algorithm
3. **gradient** of potential yields electric fields
4. **mesh to particles**: interpolate mesh fields to particles





Hockney's Algorithm

Poisson's equation

$$\Delta\phi(\vec{x}) = \rho(\vec{x})$$

can be solved via the Green's function method

$$G: \ \Delta G(\vec{x}) = \delta(\vec{x}) \quad .$$

Trick: mirroring $G(\vec{x})$ for each plane $\Longrightarrow$ periodicity!
Formal solution with convoluted Green's function

$$\varphi(\vec{x}) = \int d^3y \ \rho(\vec{x})\, G(\vec{x},\vec{y})$$

can be expressed as Fourier transform ($\Longrightarrow$ **FFT**!),

$$\varphi(\vec{x}) = \mathscr{F}\left\{\mathscr{F}\{\rho\}\,\mathscr{F}\{G\}\right\} \quad .$$

Implementation

- ⟶ direct space charge algorithm implemented in PyPIC[1)], an effort to share PIC algorithms with a python interface
- ⟶ integrated into PyHEADTAIL[2)], a collective effects library

[1)] https://github.com/PyCOMPLETE/PyPIC

[2)] https://github.com/PyCOMPLETE/PyHEADTAIL

Python. Python?

Why python?

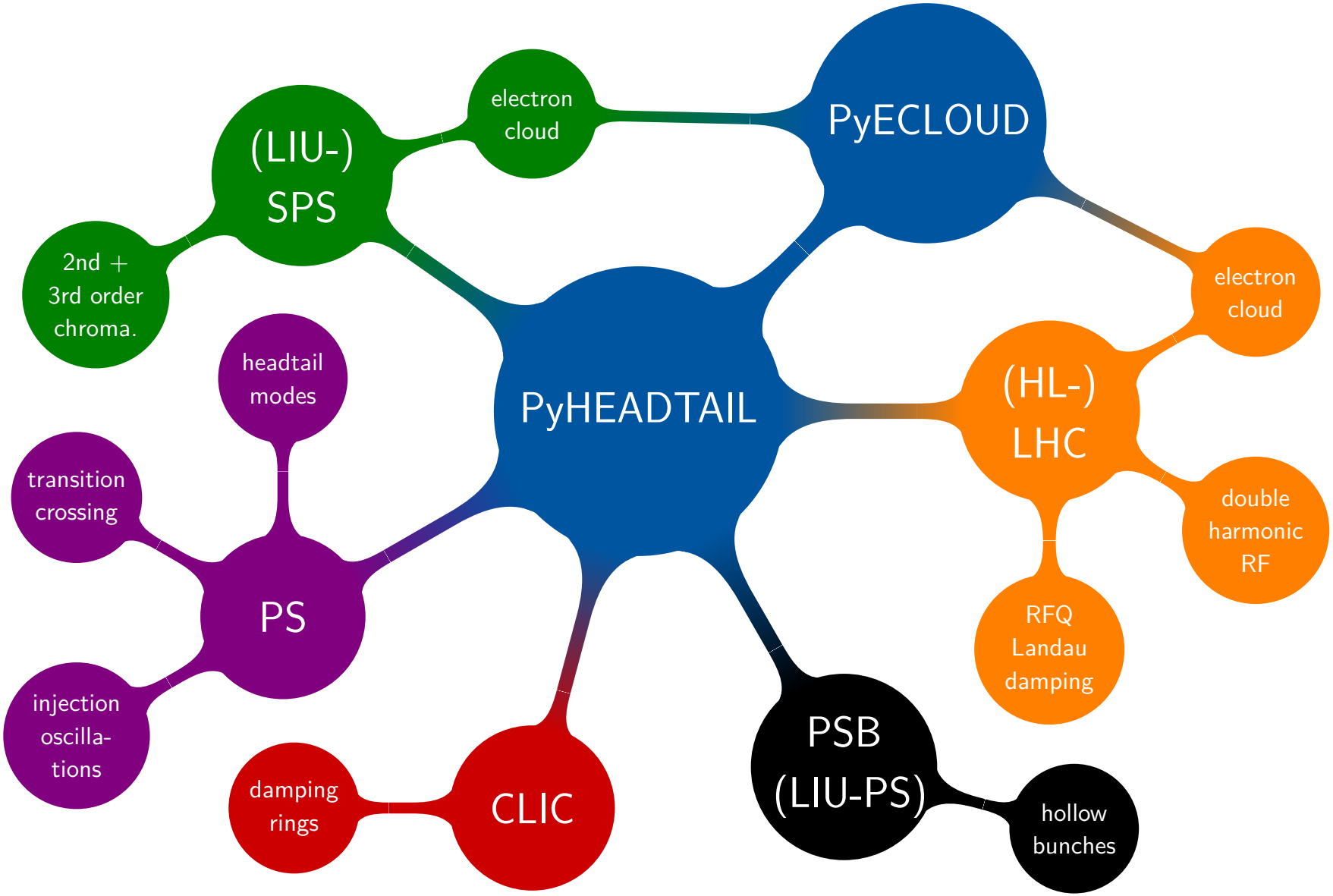
- very active development of PyHEADTAIL
- required to be easily extensible
- sped up development process, rapid prototyping
- dynamic developer community
- enhanced code legibility $\Longrightarrow$ maintainability
- data processing becomes trivial, flexible

... but isn't that extremely slow?!

- identify bottlenecks by profiling

$\Longrightarrow$ translate relevant parts into high-performance languages

Studies with PyHEADTAIL



GPUs and PyHEADTAIL / PyPIC

on-going efforts to parallelise PyHEADTAIL on GPU:

$\longrightarrow$ master thesis of Stefan Hegglin

ingredients to PyHEADTAIL on the GPU:

- PyCUDA library by Andreas Kloeckner: exploit NumPy's vectorisation model
- in future: PyOpenCL?
- scikit-cuda library by Lev Givon for cuFFT, cuSOLVER, ...

$\Longrightarrow$ straight-forward porting of PyHEADTAIL to GPU, compromise on speed

- sparse matrix solving library cuSOLVER tested

$\longrightarrow$ parallelisation unsuccessful: Poisson matrix too serial

$\Longrightarrow$ FFT based approach most successful

2.5D vs. 3D Model

2.5D "slice-by-slice" model:

- slice bunch into n slices
- solve n independent 2D transverse Poisson equations
- approximation: bunch very long

$\longrightarrow$ CPU: serial

$\longrightarrow$ GPU: treat slices in parallel

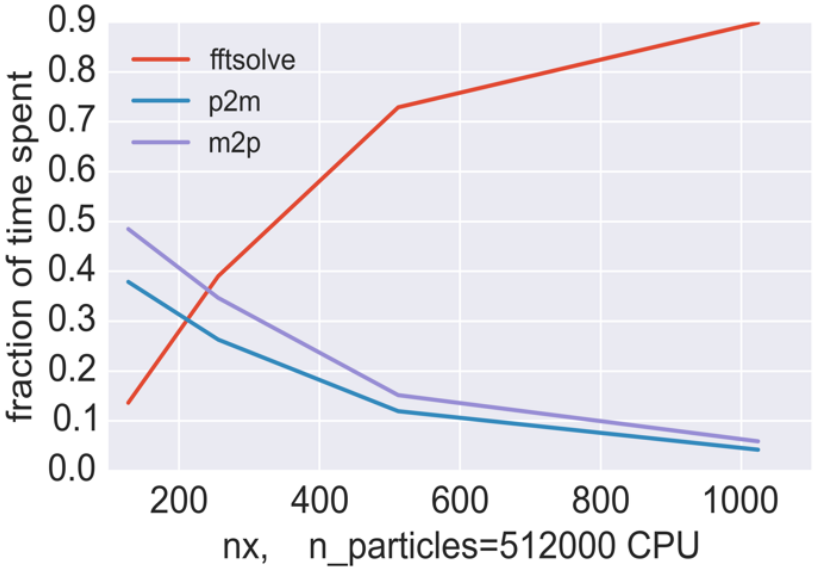
3D model:

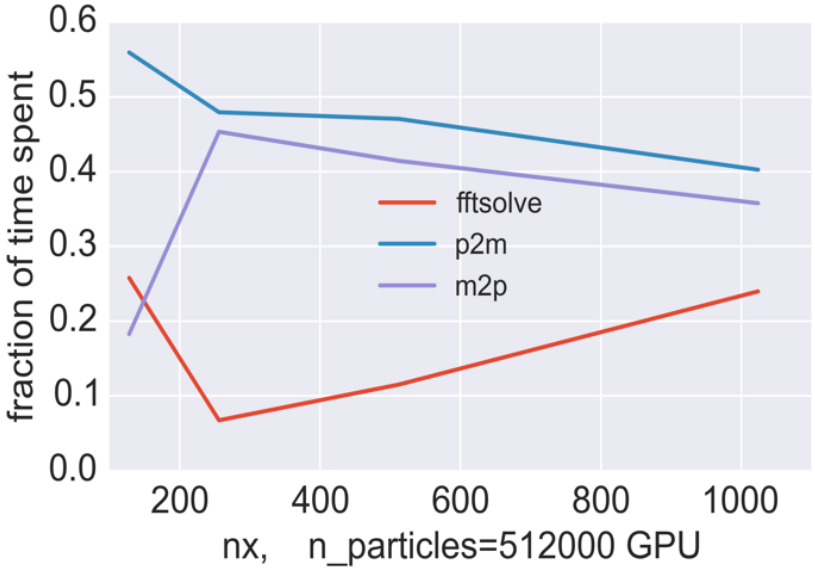
- solve the full 3D bunch on a 3D grid

$\longrightarrow$ CPU: too slow to be practical due to one more FFT

$\longrightarrow$ GPU: large memory requirements due to Hockney's algorithm

Bottleneck CPU vs. GPU

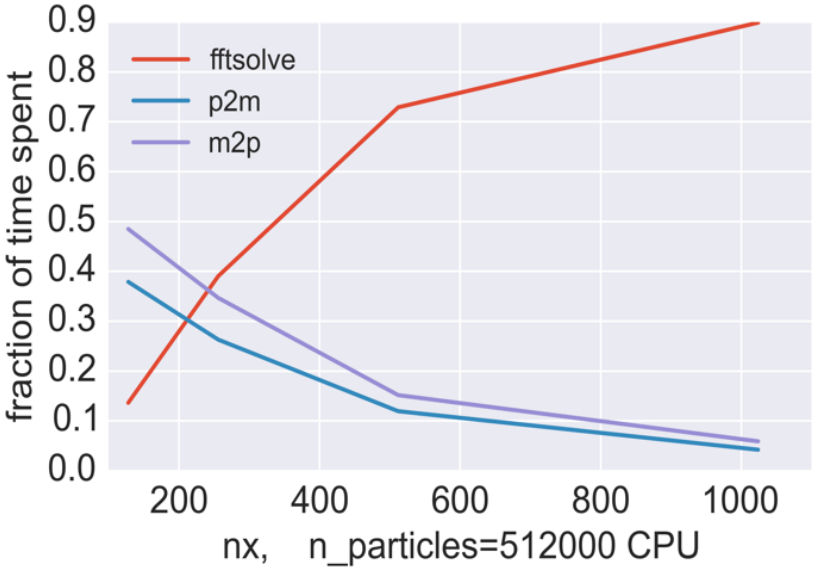


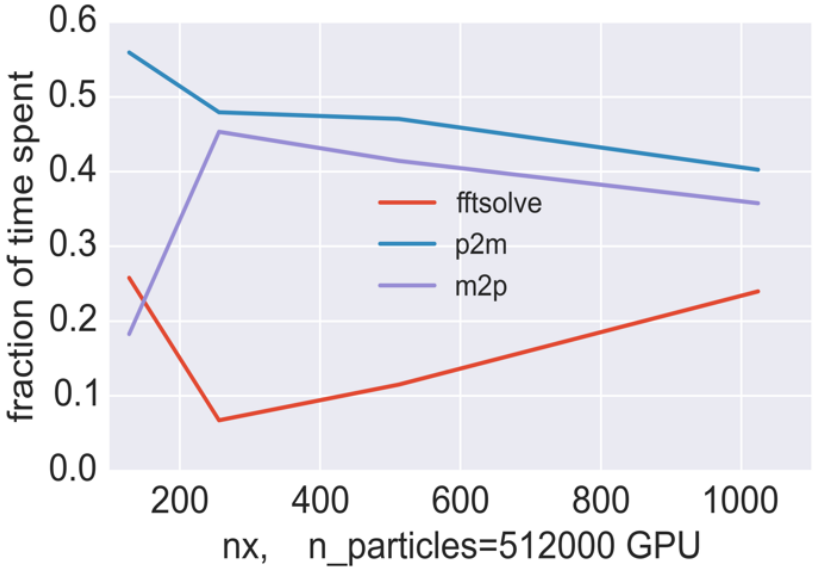


profiling of 2D implementation reveals

- on CPU, FFT solving dominates
- cuFFT on GPU: $\sim$ 30x faster
- mesh deposition bottleneck on GPU, memory-bound

Bottleneck CPU vs. GPU





⟶ implemented K. Ahnert et al.'s molecular dynamics algorithm from Numerical Computations with GPUs

1. sort particles by cell ID
2. determine cell boundary indices
3. 1 thread ⟷ 1 cell: add contributions per cell

⟹ distributes memory access and avoids stalls (speed-up ~3x)

Resources

BE-ABP simulations carried out at

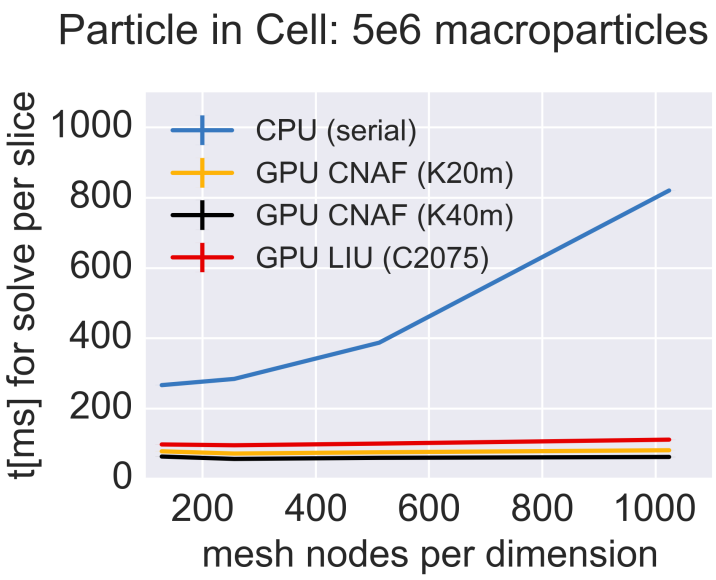
- CERN: LIU-PS-GPU server
  - ⟶ 4x NVIDIA Tesla C2075 cards (mid 2011)
- CNAF (Bologna): high performance cluster
  - ⟶ 7x NVIDIA Tesla K20m (early 2013)
  - ⟶ 8x Tesla K40m (late 2013)

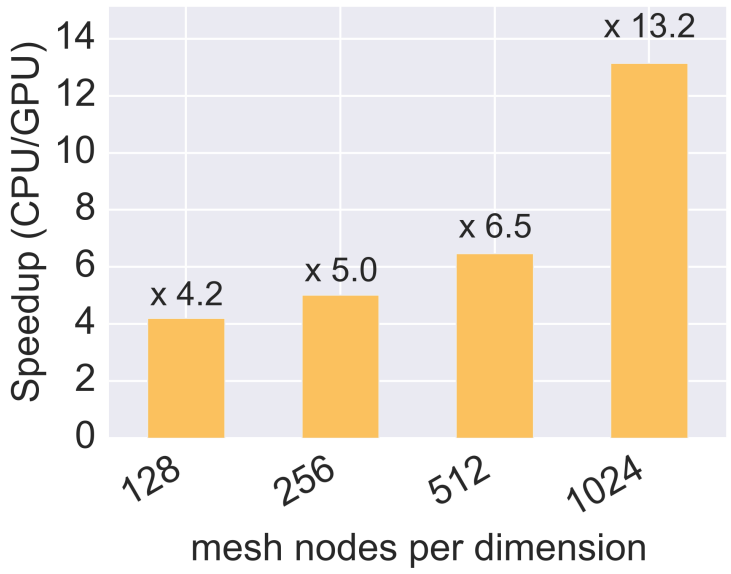
(relevant specifications in appendix)

Speed-up Results

multi-particle tracking, direct space charge: $\mathbf{S \leq 13.2}$

$\Longrightarrow$ interaction between particles, memory-bound situation





cf. single-particle tracking study, longitudinal plane: $\mathbf{S \leq 428}$ [3)]

$\Longrightarrow$ "embarrassingly parallel" computationally-bound situation

[3)] `www.oeftiger.net/parallelisation-longitudinal-tracking/`

Current Applications

- injection oscillations: beam envelope frequencies shifted by space charge (beam resonances?)

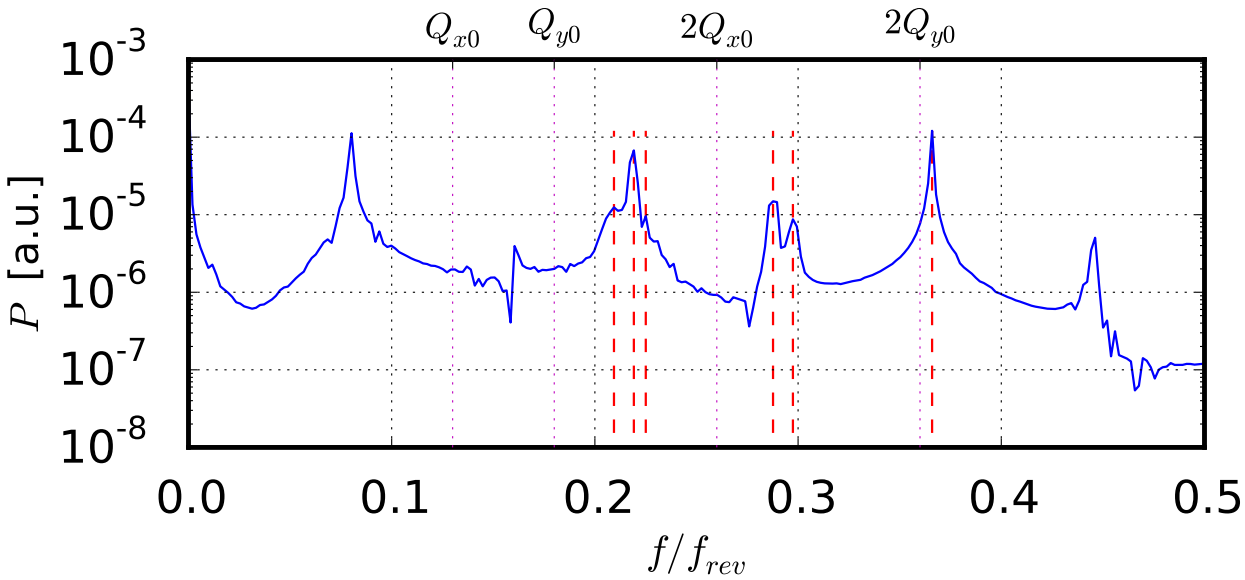
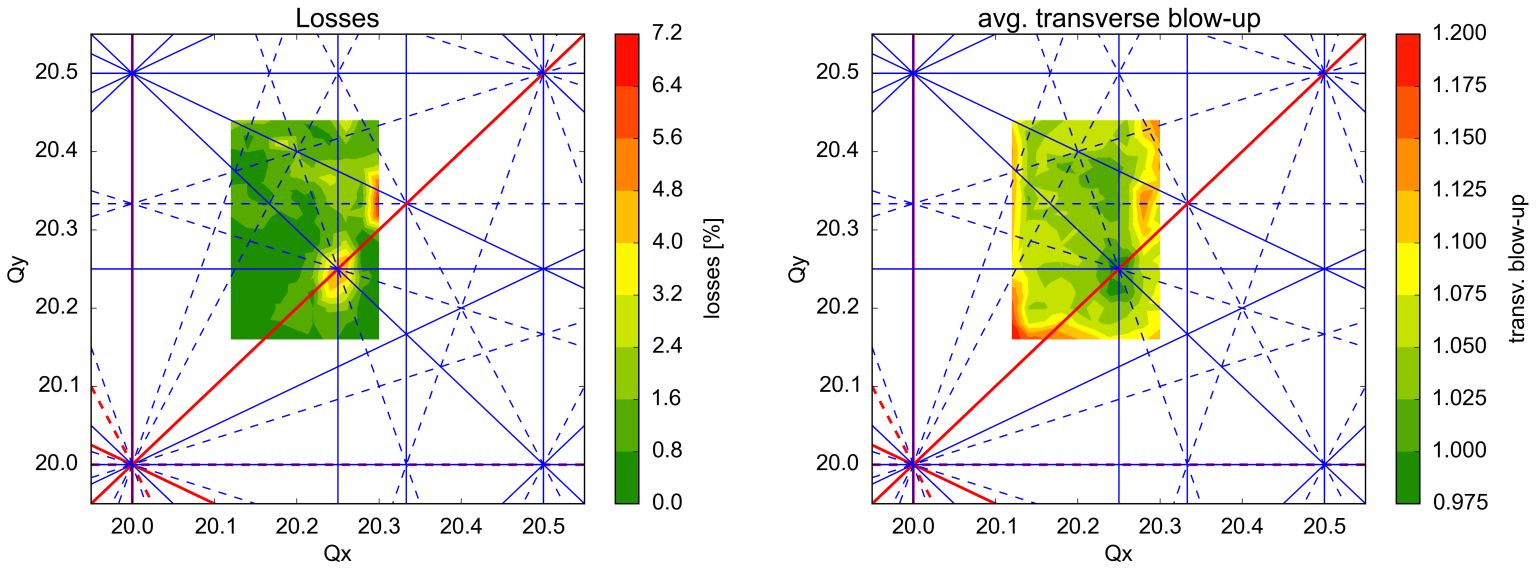


Figure : quadrupolar pickup spectrum for injection oscillations of beam envelope in SPS

Current Applications

- injection oscillations: beam envelope frequencies shifted by space charge (beam resonances?)
- identification and characterisation of relevant resonances during SPS injection plateau



(a) beam losses (b) beam emittance blowup

Figure : measured tune diagram during 3 s of SPS injection plateau

Further Reading

relevant presentations on PyHEADTAIL and PyPIC:

- PyHEADTAIL space charge suite, presented at Oxford Space Charge 2015: `https://eventbooking.stfc.ac.uk/uploads/spacecharge15/oeftiger-pyheadtail.pdf`
- overview GPU parallelisation, presented in BE-ABP-HSC section meeting: `https://espace.cern.ch/be-dep/ABP/HSC/Meetings/GPUFFT.pptx`
- parallelisation approach and some physics details, presented in (former BE-ABP-HSC) Space Charge Working Group: `http://frs.web.cern.ch/frs/Source/space_charge/Meetings/meeting67_29.10.2015/`

Conclusion: Space Charge

Successfully parallelised direct space charge on GPU

- 3D model made accessible for simulations
- large mesh sizes and high #macro-particles $\Longrightarrow$ $S \leq 13$

$\Longrightarrow$ large resolutions also address mitigating noise effects (artefacts such as numerical emittance blow-up)

$\Longrightarrow$ improved validity for long-term simulations

Take-home message:

- Python allows rapid development for changing demands
- PyCUDA greatly simplifies concurrent GPU development

$\longrightarrow$ minimal code maintenance, less duplicate code

$\longrightarrow$ reasonable compromise in speed-up

Further GPU Studies: SixTrack

SixTrack: Single Particle Tracking Code (`cern.ch/sixtrack`)

- 70k lines written in Fortran 77/90
- numerically portable across OS and compilers
- used in the volunteer computing project LHC@Home with 200k registered users

GPU porting is being explored in the context of LHC@Home to use volunteer GPU:

$\longrightarrow$ heterogeneous hardware and software hard to test and fully deploy, many low-end GPU expected (low FP64 FLOPS count)

- D. Mikushin (Applied Parallel Computing LLC) (`indico/event/450856`) demonstrated deploying with CUDA + additional compilation stages + code annotations + special compiler software (numerically ok without FMAC instructions, no benchmark available)

Riccardo de Maria

Further GPU Studies: SixTrack

SixTrack: Single Particle Tracking Code (`cern.ch/sixtrack`)

- standalone tracking library (SixTrackLib) to be used with other codes (including SixTrack itself):
  - lightweight code being written in C/OpenCL for flexibility/portability (CERN&GSoC'14-'15)

  $\Longrightarrow$ speed-up of **250x** w.r.t. single i7 core with AMD-280X (1TFLOPS FP64, 300CHF) on first tests driven by PyOPENCL.
- if substantial and well controlled hardware/software resources are available, there could be an interest to deploy the SixTrack using GPU (provided reserving some time in SixTrack team).
  Hardware for single particle simulation: high FP64 FLOPS.
  Memory bandwidth and memory size less important.

Riccardo de Maria

Thank you for your attention!

Acknowledgements to PyHEADTAIL + PyPIC team:
Hannes Bartosik, Stefan Hegglin, Giovanni Iadarola, Kevin Li,
Annalisa Romano, Giovanni Rumolo, Michael Schenk
https://github.com/PyCOMPLETE/

CPU Machine – Specifications

CERN BE-ABP "LIUPSGPU" machine:

CPU	2× Intel Xeon E5-2630 (v1)
CPU cores	2×6
RAM	256 GB DDR3
CPU clock rate	2.30 GHz
CPU L3 cache	15 MB
instruction set	Intel AVX
32bit floating-point performance	0.1 TFLOPS

Table : Relevant CPU Machine Specifications

GPU Machines – Specifications

available machines at CNAF:

`http://wiki.infn.it/strutture/cnaf/clusterhpc/home`

	CERN BE-ABP	CNAF	
GPU	NVIDIA Tesla C2075	NVIDIA Tesla K20	NVIDIA Tesla K40
avail. GPU devices	4	7	8
avail. GPU DDR5 RAM (per device)	5.3 GB	5.1 GB	12.3 GB
GPU clock rate	1.15 GHz	0.7 GHz	0.75 GHz
CUDA cores per device	448	2496	2880
max. no of threads per block	$1024 \times 1024 \times 64$	$1024 \times 1024 \times 64$	$1024 \times 1024 \times 64$
CUDA computing capability	2.0	3.5	3.5
32bit floating-point performance	1.0 TFLOPS	3.5 TFLOPS	4.3 TFLOPS

Table : Relevant GPU Machine Specifications