SixTrackLib: Design & Implementation of a GPU Accelerated Beam-Dynamics Simulation Library

Martin Schwinzerl

June 19th, 2020 :: BE Seminar :: CERN

Supervisors:

Riccardo De Maria (CERN)

Gundolf Haase
(University of Graz, Austria)

Supported by the Austrian
Doctoral Student Program @ CERN

Special Thanks & Acknowledgements:

Hannes Bartosik (CERN),
Massimo Giovannozzi (CERN),
Giovanni Iadarola (CERN),
Carlo Emilio Montanari (U. Bologna),
Adrian Oeftiger (GSI/FAIR),
Konstantinos Paraschou (AUTH, CERN)

Goals

- Introduce single-particle tracking, symplectic tracking and `SixTrackLibs` approach to data-parallelism
- Explain & motivate design decisions for `SixTrackLib`
- Provide a minimal API demonstration (Cf. accompanying jupyter-notebook)
- Give overview about the preliminary performance figures
- Showcase examples of real-world applications

Goals

- Introduce single-particle tracking, symplectic tracking and `SixTrackLibs` approach to data-parallelism
- Explain & motivate design decisions for `SixTrackLib`
- Provide a minimal API demonstration (Cf. accompanying jupyter-notebook)
- Give overview about the preliminary performance figures
- Showcase examples of real-world applications

Goals

- Introduce single-particle tracking, symplectic tracking and `SixTrackLibs` approach to data-parallelism
- Explain & motivate design decisions for `SixTrackLib`
- Provide a minimal API demonstration (Cf. accompanying jupyter-notebook)
- Give overview about the preliminary performance figures
- Showcase examples of real-world applications

Goals

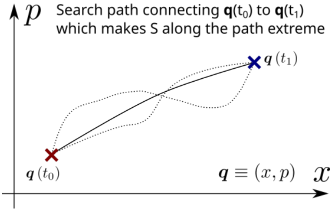
- Introduce single-particle tracking, symplectic tracking and `SixTrackLibs` approach to data-parallelism
- Explain & motivate design decisions for `SixTrackLib`
- Provide a minimal API demonstration (Cf. accompanying jupyter-notebook)
- Give overview about the preliminary performance figures
- Showcase examples of real-world applications

Goals

- Introduce single-particle tracking, symplectic tracking and `SixTrackLibs` approach to data-parallelism
- Explain & motivate design decisions for `SixTrackLib`
- Provide a minimal API demonstration (Cf. accompanying jupyter-notebook)
- Give overview about the preliminary performance figures
- Showcase examples of real-world applications

Introduction

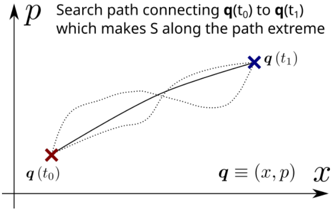
Hamiltonian Formulation



- $\boldsymbol{q} \equiv (x, p)$ conjugate coordinates
- for given start- and end-points in phase-space $\boldsymbol{q}(t_0)$ and $\boldsymbol{q}(t_1)$ and
- $S := \int_{t_0}^{t_1} dt \, [p \cdot \dot{x} - H(x, p, t)]$,
- find expressions for $\boldsymbol{q}$ so that $\delta S \to 0$

- We define a (Transfer) **Map** as a transformation that has the same effect as integrating $\dot{q}_i$ from $t_0 \mapsto t_1$, i.e. $\boldsymbol{q}(t_1) = \boldsymbol{M}_{t_0 \mapsto t_1}(\boldsymbol{q}(t_0))$

Hamiltonian Formulation



- $\boldsymbol{q} \equiv (x, p)$ conjugate coordinates
- for given start- and end-points in phase-space $\boldsymbol{q}(t_0)$ and $\boldsymbol{q}(t_1)$ and
- $S := \int_{t_0}^{t_1} dt\, [p \cdot \dot{x} - H(x, p, t)]$,
- find expressions for $\boldsymbol{q}$ so that $\delta S \to 0$

Hamilton Equations of Motion, Transfer Maps

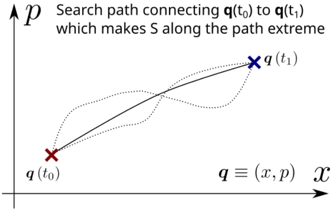
It can be shown, that if x, p, H, and t obey the equations

$$\dot{x} = \frac{dx}{dt} = \frac{\partial H}{\partial p}, \quad \dot{p} = \frac{dp}{dt} = -\frac{\partial H}{\partial x} \tag{1}$$

then $\delta S \to 0$ indeed is true. In physics, the *Hamiltonian* $H \equiv T + V$

- We define a (Transfer) **Map** as a transformation that has the same effect as integrating $\dot{q}_i$ from $t_0 \mapsto t_1$, i.e. $\boldsymbol{q}(t_1) = \boldsymbol{M}_{t_0 \mapsto t_1}(\boldsymbol{q}(t_0))$

Hamiltonian Formulation



- $\boldsymbol{q} \equiv (x, p)$ conjugate coordinates
- for given start- and end-points in phase-space $\boldsymbol{q}(t_0)$ and $\boldsymbol{q}(t_1)$ and
- $S := \int_{t_0}^{t_1} dt \, [p \cdot \dot{x} - H(x, p, t)]$,
- find expressions for $\boldsymbol{q}$ so that $\delta S \to 0$

Hamilton Equations of Motion, Transfer Maps

It can be shown, that if x, p, H, and t obey the equations

$$\dot{x} = \frac{dx}{dt} = \frac{\partial H}{\partial p}, \quad \dot{p} = \frac{dp}{dt} = -\frac{\partial H}{\partial x} \tag{1}$$

then $\delta S \to 0$ indeed is true. In physics, the *Hamiltonian* $H \equiv T + V$

- We define a (Transfer) **Map** as a transformation that has the same effect as integrating $\dot{q}_i$ from $t_0 \mapsto t_1$, i.e. $\boldsymbol{q}(t_1) = \boldsymbol{M}_{t_0 \mapsto t_1}(\boldsymbol{q}(t_0))$

Symplectic Transformation & Integration

- For $\delta t \to 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes
  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0}$
- Jacobian of the Transformation
  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \; = \; \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$
- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0 + 2\delta t} = J_{(t_0+\delta t) \mapsto (t_0 + 2\delta t)} \circ J_{t_0 \mapsto t_0 + \delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Symplectic Transformation & Integration

- For $\delta t \to 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes
  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0}$
- Jacobian of the Transformation
  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \; = \; \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$
- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0 + 2\delta t} = J_{(t_0+\delta t) \mapsto (t_0 + 2\delta t)} \circ J_{t_0 \mapsto t_0 + \delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Symplectic Transformation & Integration

- For $\delta t \rightarrow 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes
  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0}$
- Jacobian of the Transformation
  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \; = \; \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$

Definition: Symplectic Transformation

J is a symplectic transformation $:\Longleftrightarrow \; J^T \Omega J = \Omega$

- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0 + 2\delta t} = J_{(t_0+\delta t) \mapsto (t_0 + 2\delta t)} \circ J_{t_0 \mapsto t_0 + \delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Symplectic Transformation & Integration

- For $\delta t \to 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes

  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0}$
- Jacobian of the Transformation

  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \ = \ \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \frac{\partial^2 H}{\partial q_j \partial q_k}\Big|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$

Definition: Symplectic Transformation

J is a symplectic transformation $:\Longleftrightarrow \ J^T \Omega J = \Omega$

- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0+2\delta t} = J_{(t_0+\delta t) \mapsto (t_0+2\delta t)} \circ J_{t_0 \mapsto t_0+\delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Symplectic Transformation & Integration

- For $\delta t \to 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes
  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0}$
- Jacobian of the Transformation
  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \ = \ \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$

Definition: Symplectic Transformation

J is a symplectic transformation $:\Longleftrightarrow \ J^T \Omega J = \Omega$

- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0 + 2\delta t} = J_{(t_0+\delta t) \mapsto (t_0 + 2\delta t)} \circ J_{t_0 \mapsto t_0 + \delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Symplectic Transformation & Integration

- For $\delta t \to 0$, $q \equiv (q_0, q_1)$, we find $q_i(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \dot{q}_i$
- With (1) and $\Omega_{ik} = (\Omega)_{ik} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, this becomes
  $q(t_0 + \delta t) = q_i(t_0) + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0}$
- Jacobian of the Transformation
  $J_{ij} = \frac{\partial q_i(t_0+\delta t)}{\partial q_j(t_0)} \; = \; \delta_{ij} + \delta t \cdot \Omega_{ik} \cdot \left. \frac{\partial^2 H}{\partial q_j \partial q_k} \right|_{t=t_0} \Rightarrow J = I + \delta t \cdot \Omega \cdot \tilde{H}$

Definition: Symplectic Transformation

J is a symplectic transformation $:\Longleftrightarrow \; J^T \Omega J = \Omega$

- J as derived via (1) fulfills symplectivity condition
- Thus $J_{t_0 \mapsto t_0 + 2\delta t} = J_{(t_0+\delta t) \mapsto (t_0 + 2\delta t)} \circ J_{t_0 \mapsto t_0 + \delta t}$ also symplectic
- $\boldsymbol{M}_{t_0 \mapsto t_1}$ constructed from a composition of symplectic J is also symplectic (i.e. $\boldsymbol{M}$ is a **symplectic Map**)

Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map

Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map

Consequences (Hamiltonian EoM, Symplectic Maps)

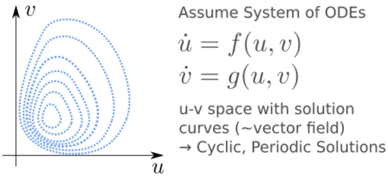
- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map

Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map

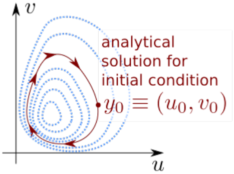
Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map
- Cyclic Motion



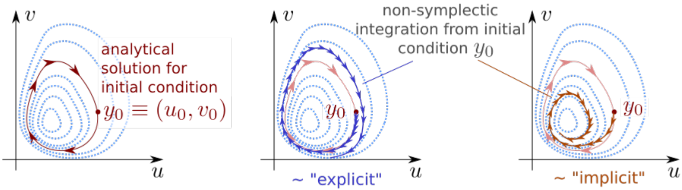
Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map
- Cyclic Motion



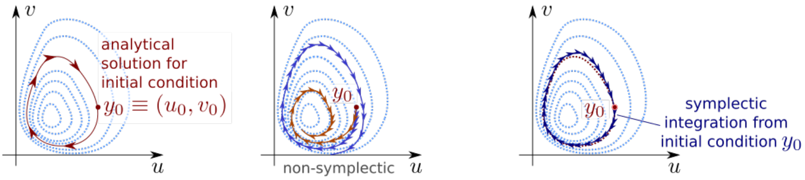
Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map
- Cyclic Motion



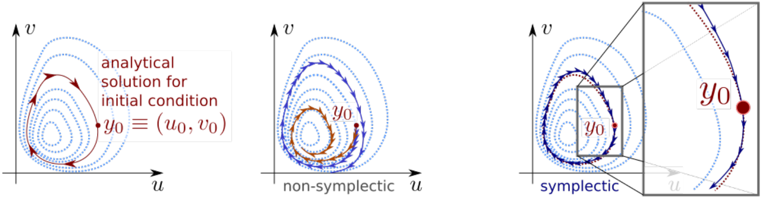
Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map
- Cyclic Motion $\longrightarrow$ Closed Orbit



Consequences (Hamiltonian EoM, Symplectic Maps)

- J is symplectic $\Longrightarrow det(J) = \pm 1$
  Preservation of phase space volume (Liouville Theorem)
- Hamiltonian formalism allows algebraic transformation of independent variable from $t \longrightarrow s$ (i.e. distance from start of turn)
- $\Longrightarrow$ Allows to approximate the effect of "beam-elements" located at spatial position s with sequence of symplectic maps
- Composition of beam-element maps $\rightarrow$ symplectic one-turn-map
- Cyclic Motion $\longrightarrow$ Closed Orbit
  (approximations/truncations $\rightarrow$ deviations; but: orbit still closed!)

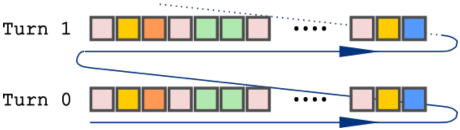


Single Particle Tracking, Parallelism

- Accelerator $\sim$ sequence of discrete beam-elements ("lattice")
- Tracking a particle over $N \geq 10^4 \ldots 10^8$ turns $\rightarrow$ numerically expensive & challenging (non-linear, on-setting chaos,...)
- If any two particles P_i and P_j $i \neq j \in [0, N_P)$ do not interact $\rightarrow$ "single-particle tracking"

Single Particle Tracking, Parallelism

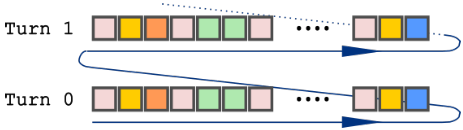
- Accelerator ∼ sequence of discrete beam-elements ("lattice")
- Tracking a single particle over a lattice $\Longrightarrow$ sequential operation



- Tracking a particle over $N \geq 10^4 \ldots 10^8$ turns → numerically expensive & challenging (non-linear, on-setting chaos,...)
- If any two particles P_i and P_j $i \neq j \in [0, N_P)$ do not interact → "single-particle tracking"

Single Particle Tracking, Parallelism

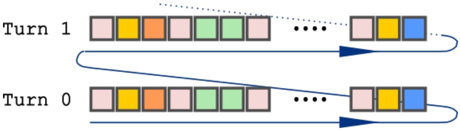
- Accelerator $\sim$ sequence of discrete beam-elements ("lattice")
- Tracking a single particle over a lattice $\Longrightarrow$ sequential operation



- Tracking a particle over $N \geq 10^4 \ldots 10^8$ turns $\rightarrow$ numerically expensive & challenging (non-linear, on-setting chaos,...)
- If any two particles P_i and P_j $i \neq j \in [0, N_P)$ do not interact $\rightarrow$ "single-particle tracking"

Single Particle Tracking, Parallelism

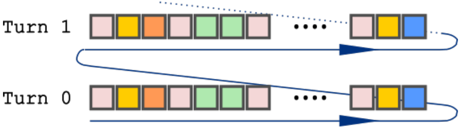
- Accelerator $\sim$ sequence of discrete beam-elements ("lattice")
- Tracking a single particle over a lattice $\Longrightarrow$ sequential operation



- Tracking a particle over $N \geq 10^4 \ldots 10^8$ turns $\rightarrow$ numerically expensive & challenging (non-linear, on-setting chaos,...)
- If any two particles P_i and P_j $i \neq j \in [0, N_P)$ do not interact $\rightarrow$ "single-particle tracking"

Single Particle Tracking, Parallelism

- Accelerator $\sim$ sequence of discrete beam-elements ("lattice")
- Tracking a single particle over a lattice $\Longrightarrow$ sequential operation



- Tracking a particle over $N \geq 10^4 \ldots 10^8$ turns $\rightarrow$ numerically expensive & challenging (non-linear, on-setting chaos,...)
- If any two particles P_i and P_j $i \neq j \in [0, N_P)$ do not interact $\rightarrow$ "single-particle tracking"
- Single-Particle Tracking + $N_P >> 1 \longrightarrow$ "embarrassingly parallel problem" (data-parallelism)

Bringing It All Together: SixTrackLib

SixTrackLib is a

1. Parallel
2. Single-Particle
3. Symplectic Tracking
4. Library

- Re-implementation of the core functionality of SixTrack, focusing only on tracking
- Under development for > 2 years
- https://github.com/SixTrack/sixtracklib

- Numerical accuracy, stability & reproducibility
- Wide range of supported hardware $\rightarrow$ multiple parallel backends
- Good scalability towards $N_P >> 1$ (parallel processors, GPUs)
- High code efficiency for $N_P \sim 1$ (CPU)
- Strict separation between "physics" and "business logic" code
- Single code base, bindings to multiple languages

Bringing It All Together: SixTrackLib

SixTrackLib is a

1. Parallel
2. Single-Particle
3. Symplectic Tracking
4. Library

- Re-implementation of the core functionality of SixTrack, focusing only on tracking
- Under development for > 2 years
- https://github.com/SixTrack/sixtracklib

- Numerical accuracy, stability & reproducibility
- Wide range of supported hardware $\rightarrow$ multiple parallel backends
- Good scalability towards $N_P >> 1$ (parallel processors, GPUs)
- High code efficiency for $N_P \sim 1$ (CPU)
- Strict separation between "physics" and "business logic" code
- Single code base, bindings to multiple languages

Bringing It All Together: SixTrackLib

SixTrackLib is a

1. Parallel
2. Single-Particle
3. Symplectic Tracking
4. Library

- Re-implementation of the core functionality of SixTrack, focusing only on tracking
- Under development for > 2 years
- https://github.com/SixTrack/sixtracklib

Requirements:

- Numerical accuracy, stability & reproducibility
- Wide range of supported hardware $\rightarrow$ multiple parallel backends
- Good scalability towards $N_P >> 1$ (parallel processors, GPUs)
- High code efficiency for $N_P \sim 1$ (CPU)
- Strict separation between "physics" and "business logic" code
- Single code base, bindings to multiple languages

Bringing It All Together: `SixTrackLib`

`SixTrackLib` is a

1. Parallel
2. Single-Particle
3. Symplectic Tracking
4. Library

- Re-implementation of the core functionality of `SixTrack`, focusing only on tracking
- Under development for > 2 years
- `https://github.com/SixTrack/sixtracklib`

Requirements:

- Numerical accuracy, stability & reproducibility
- Wide range of supported hardware $\rightarrow$ multiple parallel backends
- Good scalability towards $N_P >> 1$ (parallel processors, GPUs)
- High code efficiency for $N_P \sim 1$ (CPU)
- Strict separation between "physics" and "business logic" code
- Single code base, bindings to multiple languages

Bringing It All Together: `SixTrackLib`

`SixTrackLib` is a

1. Parallel
2. Single-Particle
3. Symplectic Tracking
4. Library

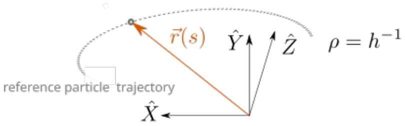
- Re-implementation of the core functionality of `SixTrack`, focusing only on tracking
- Under development for > 2 years
- `https://github.com/SixTrack/sixtracklib`

Requirements:

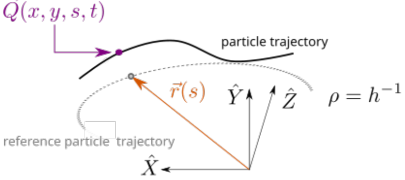
- Numerical accuracy, stability & reproducibility
- Wide range of supported hardware $\rightarrow$ multiple parallel backends
- Good scalability towards $N_P >> 1$ (parallel processors, GPUs)
- High code efficiency for $N_P \sim 1$ (CPU)
- Strict separation between "physics" and "business logic" code
- Single code base, bindings to multiple languages

Implementation, Design & Basic Usage

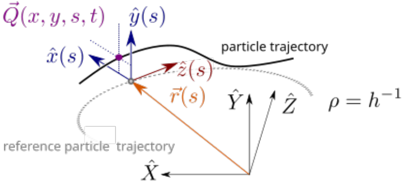
Modelling the State Of The Particles



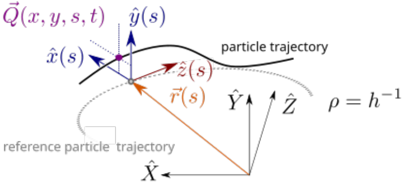
Modelling the State Of The Particles



Modelling the State Of The Particles



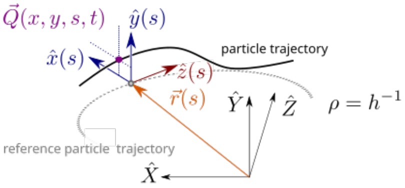
Modelling the State Of The Particles



6 Main Degrees Of Freedom

- x, y [m]
- $p_x = P_x / P_0$, $p_y = P_y / P_0$ [rad]
- $\zeta = \beta \cdot (s/\beta_0 - c \cdot t)$ [m]
- $\delta = (P - P_0)/P_0$

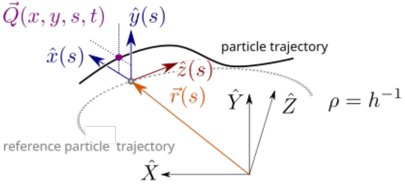
Modelling the State Of The Particles



6 Main Degrees Of Freedom

- x, y [m]
- $p_x = P_x / P_0$, $p_y = P_y / P_0$ [rad]
- $\zeta = \beta \cdot (s/\beta_0 - c \cdot t)$ [m]
- $\delta = (P - P_0)/P_0$

Modelling the State Of The Particles



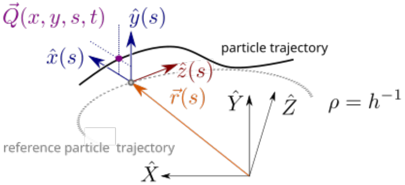
6 Main Degrees Of Freedom

- x, y [m]
- $p_x = P_x/P_0$, $p_y = P_y/P_0$ [rad]
- $\zeta = \beta \cdot (s/\beta_0 - c \cdot t)$ [m]
- $\delta = (P - P_0)/P_0$

4 Logical Coordinates

- `particle_id`
- `at_element`
- `at_turn`
- `state`
  (0 == lost, 1 == active)

Modelling the State Of The Particles



6 Main Degrees Of Freedom

- x, y [m]
- $p_x = P_x / P_0$, $p_y = P_y / P_0$ [rad]
- $\zeta = \beta \cdot (s/\beta_0 - c \cdot t)$ [m]
- $\delta = (P - P_0)/P_0$

4 Logical Coordinates

- `particle_id`
- `at_element`
- `at_turn`
- `state`
  (0 == lost, 1 == active)

6 Auxiliary Attributes

- s [m]
- $p_\sigma = (E - E_0)/(\beta_0 \cdot P_0 \cdot c)$
- $r_{pp} = P_0/P$, $r_{vv} = \beta/\beta_0$
- `charge_ratio` $= q/q_0$
- $\chi = (q/q0)/(m/m_0)$

Modelling the State Of The Particles II

- Additionally, we have

5 Attributes Describing The Reference Particle "0"

- charge q_0 ($[q_0] = 1$ proton charge)
- mass m_0 ($[m_0] = 1eV/c^2$
- $\beta_0 = v_0/c$
- $\gamma_0 = (1 - \beta_0^2)^{-1/2}$
- $\mathrm{p0c} = (P_0 \cdot c)\ [\mathrm{p0c}] = eV$

- In total: **21** attributes ($\sim$ 168 Bytes/particle)
- Store N_p particles in one structure: `struct-of-arrays`
- $\Longrightarrow$ replicate q_0, m_0, β_0, γ_0, p0c for all N_p particles! Why?
  1. Consistency: tracking is asynchronous and can update ref.
  2. Performance: vectorisation, burstable loads
  3. Flexibility: allow different ref. parameters per particle

Modelling the State Of The Particles II

- Additionally, we have

5 Attributes Describing The Reference Particle "0"

- charge q_0 ($[q_0] = 1$ proton charge)
- mass m_0 ($[m_0] = 1eV/c^2$
- $\beta_0 = v_0/c$
- $\gamma_0 = (1 - \beta_0^2)^{-1/2}$
- $\mathrm{p0c} = (P_0 \cdot c)\ [\mathrm{p0c}] = eV$

- In total: **21** attributes ($\sim$ 168 Bytes/particle)
- Store N_p particles in one structure: `struct-of-arrays`
- $\Longrightarrow$ replicate q_0, m_0, β_0, γ_0, p0c for all N_p particles! Why?
  1. Consistency: tracking is asynchronous and can update ref.
  2. Performance: vectorisation, burstable loads
  3. Flexibility: allow different ref. parameters per particle

Modelling the State Of The Particles II

- Additionally, we have

5 Attributes Describing The Reference Particle "0"

- charge q_0 ($[q_0] = 1$ proton charge)
- mass m_0 ($[m_0] = 1eV/c^2$
- $\beta_0 = v_0/c$
- $\gamma_0 = (1 - \beta_0^2)^{-1/2}$
- $\mathrm{p0c} = (P_0 \cdot c)$ $[\mathrm{p0c}] = eV$

- In total: **21** attributes ($\sim$ 168 Bytes/particle)
- Store N_p particles in one structure: `struct-of-arrays`
- $\Longrightarrow$ replicate q_0, m_0, β_0, γ_0, p0c for all N_p particles! Why?
  1. Consistency: tracking is asynchronous and can update ref.
  2. Performance: vectorisation, burstable loads
  3. Flexibility: allow different ref. parameters per particle

Modelling the State Of The Particles II

- Additionally, we have

5 Attributes Describing The Reference Particle "0"

- charge q_0 ($[q_0] = 1$ proton charge)
- mass m_0 ($[m_0] = 1eV/c^2$
- $\beta_0 = v_0/c$
- $\gamma_0 = (1 - \beta_0^2)^{-1/2}$
- $\mathrm{p0c} = (P_0 \cdot c)$ $[\mathrm{p0c}] = eV$

- In total: **21** attributes ($\sim$ 168 Bytes/particle)
- Store N_p particles in one structure: `struct-of-arrays`
- $\Longrightarrow$ replicate q_0, m_0, β_0, γ_0, $\mathrm{p0c}$ for all N_p particles! Why?
  1. Consistency: tracking is asynchronous and can update ref.
  2. Performance: vectorisation, burstable loads
  3. Flexibility: allow different ref. parameters per particle

Modelling the State Of The Particles II

- Additionally, we have

5 Attributes Describing The Reference Particle "0"

- charge q_0 ($[q_0] = 1$ proton charge)
- mass m_0 ($[m_0] = 1eV/c^2$
- $\beta_0 = v_0/c$
- $\gamma_0 = (1 - \beta_0^2)^{-1/2}$
- $\mathrm{p0c} = (P_0 \cdot c)$ $[\mathrm{p0c}] = eV$

- In total: **21** attributes ($\sim$ 168 Bytes/particle)
- Store N_p particles in one structure: `struct-of-arrays`
- $\Longrightarrow$ replicate q_0, m_0, β_0, γ_0, p0c for all N_p particles! Why?
  1. Consistency: tracking is asynchronous and can update ref.
  2. Performance: vectorisation, burstable loads
  3. Flexibility: allow different ref. parameters per particle

Lattice & Beam Elements

In General: Similar to `SixTrack`

- `Drift`, `DriftExact`
- `Multipole` (incl. Dipoles, Quadrupoles, Sextupoles, etc.)
- `Cavity`
- `RFMultipole`
- `XYShift`: transversal shift
- `SRotation`: rotation in the transversal plane
- `DipoleEdge`
- `BeamMonitor`: programmable dump of particle state
- `BeamBeam4D`, `BeamBeam6D`
- `SpaceChargeCoasting`, `SpaceChargeBunched`[1]
- `LimitRect`, `LimitEllipse`, `LimitRectEllipse`: aperture checks

[1], `SpaceChargeBunched` → `SpaceChargeQGaussian`

Lattice & Beam Elements II

There are different approaches to build a new or import an existing lattice for `SixTrackLib`:

1. Build manually, element by element
2. Import from `pysixtrack`
3. Import from `MAD-X` (via `pysixtrack` and `cpymad`)
4. Import from `SixTrack` (via `pysixtrack` and `sixtracktools`)
5. Load from binary dump

Related Python-Centric Projects Under the `SixTrack` Umbrella:

- `pysixtrack`: `https://github.com/SixTrack/pysixtrack`
- `sixtracktools`: `https://github.com/SixTrack/sixtracktools`
- `cobjects`: `https://github.com/SixTrack/cobjects`

Cf. accompanying jupyter notebook + data samples!

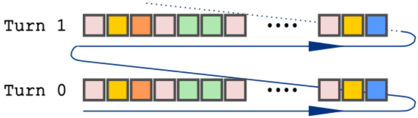
Example: Simple Tracking Example

```python
import sixtracklib as st
import numpy as np

# Create an initial particle distribution:
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)

# Load the lattice containing all the beam-elements in sequence from a prepared file
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")

# Most users will only interact with the so called "Track Job"
# Setup an instance:
job = st.TrackJob( lattice, beam )

# Track *until* all active particles arrive in turn 100
job.track_until( 100 )

# Actively mark a specific particle as lost
p.state[0] = 0 # 0 == lost, 1 == active

# Track *until* all active particles arrive in turn 200
job.track_until( 200 )

# Print the result to verifiy the success of the operation
if p.num_particles <= 16:
    print( f"at_element after tracking for 200 turns: {p.at_element}" )
    print( f"at_turn    after tracking for 200 turns: {p.at_turn}" )
    print( f"state      after tracking for 200 turns: {p.state}" )
    print( f"x          after tracking for 200 turns: {p.x}" )
```

```
at_element after tracking for 200 turns: [0 0 0 0 0 0 0 0 0 0]
at_turn    after tracking for 200 turns: [100 200 200 200 200 200 200 200 200 200]
state      after tracking for 200 turns: [0 1 1 1 1 1 1 1 1 1]
x          after tracking for 200 turns: [-9.99845051e-07 -7.77491911e-07 -5.55303462e-07 -3.33123094e-07
 -1.10949224e-07  1.11219787e-07  3.33385573e-07  5.55549726e-07
  7.77713872e-07  9.99879665e-07]
```

Modes & Logistics Of Tracking

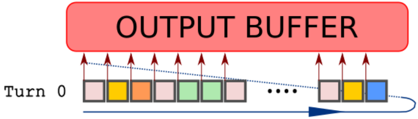
1. `track_until` Mode:



Track all active particles until they reach at_turn N

`job.track_until( N )`

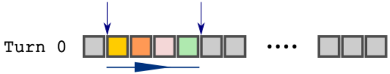
2. `track_elem_by_elem` Mode:



Like track_until(), but dump (i.e. copy) the particle state to an external buffer before each beam-element

`job.track_elem_by_elem( N )`

3. `track_line` Mode:

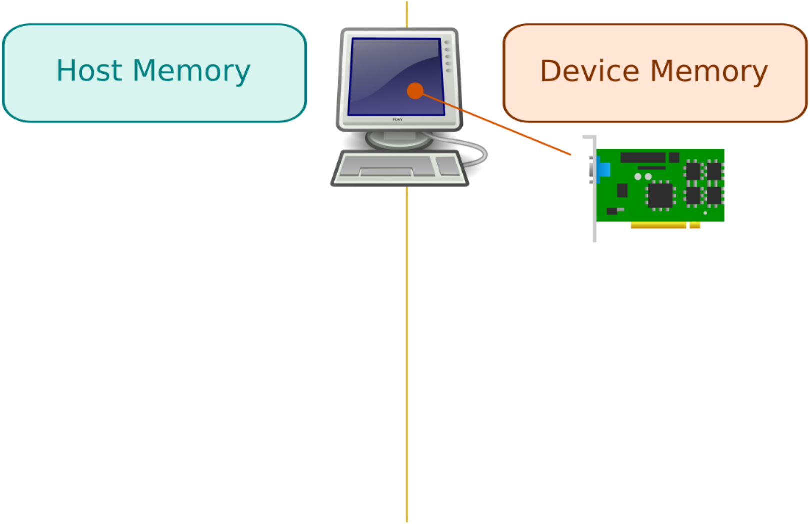


Track over subset of lattice [begin, end)

```
job.track_line( begin, end,
    end_turn=False )
```

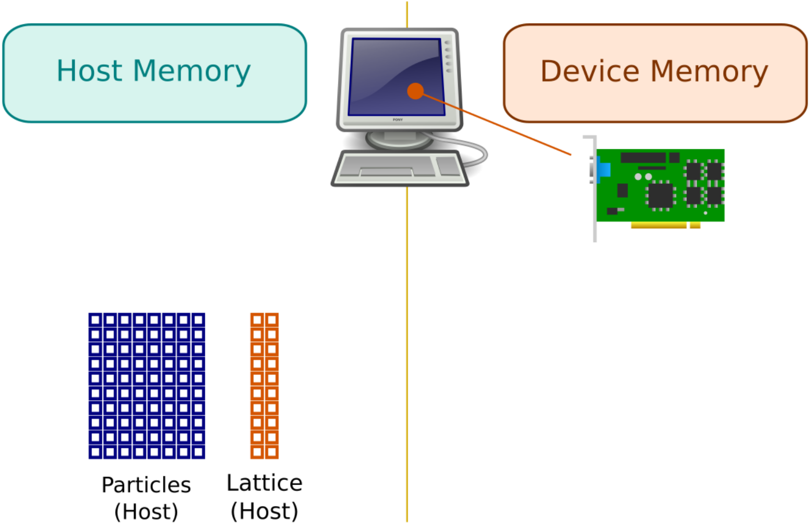
Using SixTrackLib On A GPU

(Simplified) Workflow Of A GPU Accelerated Program



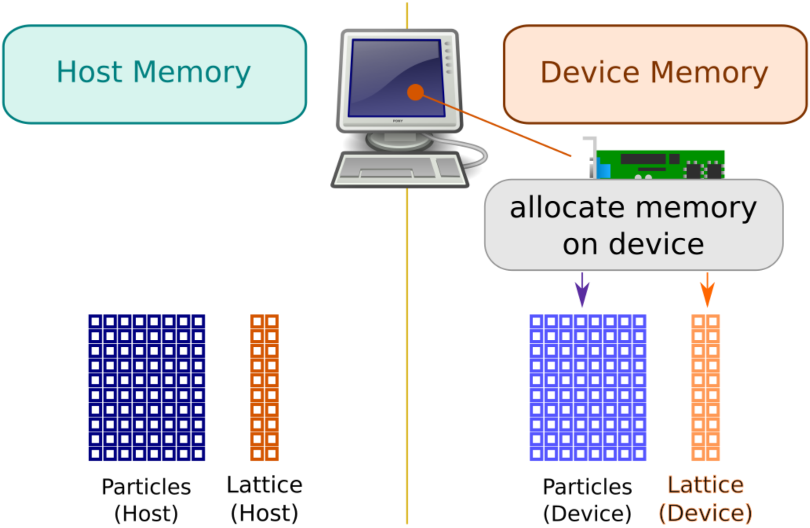
Icons: https://openclipart.org - License: Public Domain

(Simplified) Workflow Of A GPU Accelerated Program



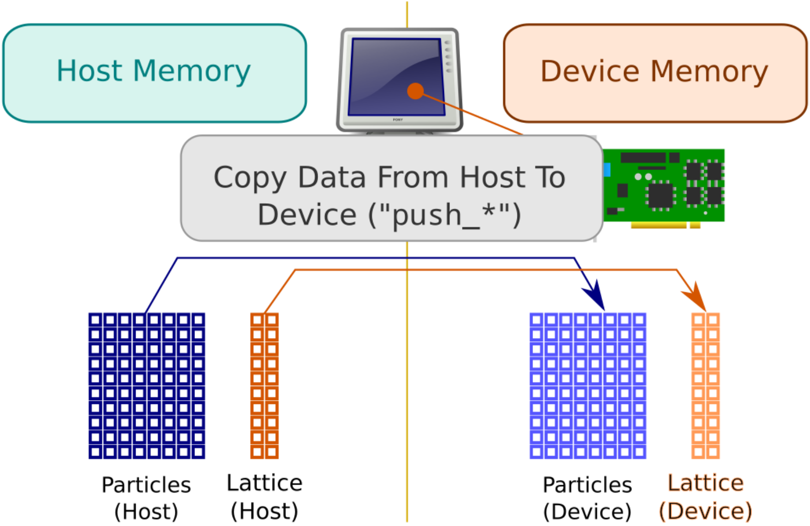
Icons: https://openclipart.org - License: Public Domain

(Simplified) Workflow Of A GPU Accelerated Program



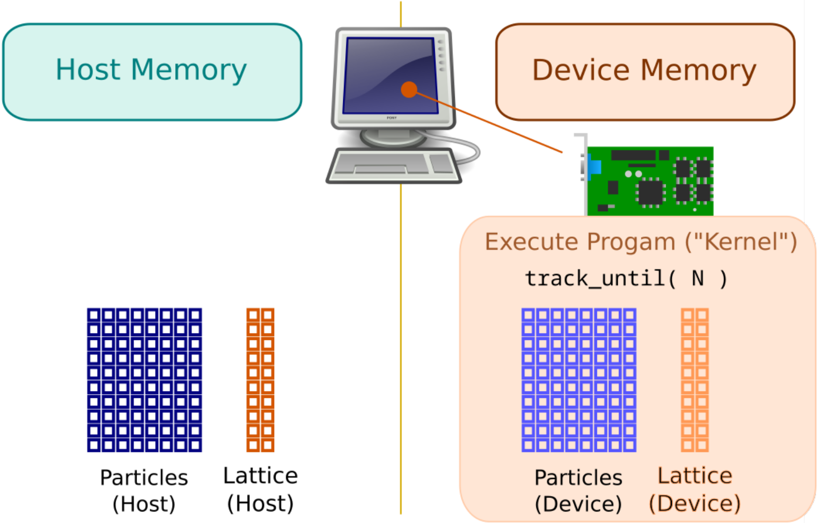
Icons: https://openclipart.org - License: Public Domain

(Simplified) Workflow Of A GPU Accelerated Program



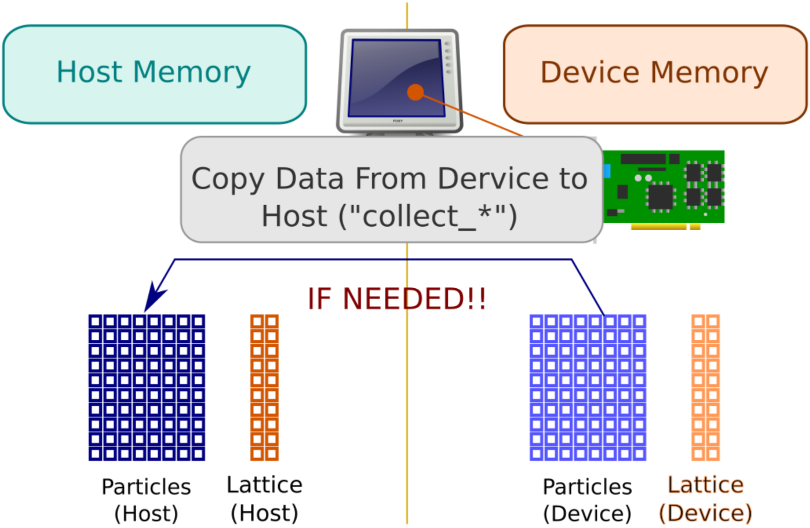
Icons: https://openclipart.org - License: Public Domain

(Simplified) Workflow Of A GPU Accelerated Program



Icons: https://openclipart.org - License: Public Domain

(Simplified) Workflow Of A GPU Accelerated Program



Icons: https://openclipart.org - License: Public Domain

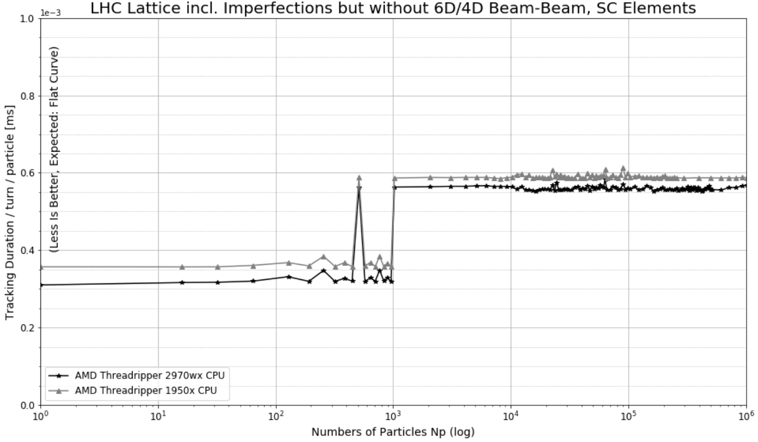
Example: Tracking Code Working On CPUs & GPUs (With Minimal Changes)

```
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")

#device=None  # Or:
device="opencl:0.0" #for GPU

job = st.TrackJob( lattice, beam, device=device )
print( f"Architecture of the track job: {job.arch_str}")

job.track_until( 100 )
job.collect_particles()

p.state[0] = 0 # Mark particle 0 explicitly as lost
job.push_particles()

job.track_until( 200 )
job.collect_particles()

if p.num_particles <= 16:
    print( f"at_element after 200 turns : {p.at_element}" )
    print( f"at_turn    after 200 turns : {p.at_turn}" )
    print( f"state      after 200 turns : {p.state}" )
```

```
Architecture of the track job: opencl
at_element after 200 turns : [0 0 0 0 0 0 0 0 0 0]
at_turn    after 200 turns : [100 200 200 200 200 200 200 200 200 200]
state      after 200 turns : [0 1 1 1 1 1 1 1 1 1]
```

Quantifying Parallel Performance

Performance CPU TrackJob)



Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) → renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) ⟶ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight → less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed → again, t_s increases

Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Limiting Factors For Parallel Performance

(**Remember**: single-particle tracking, "embarrassingly" parallel program)

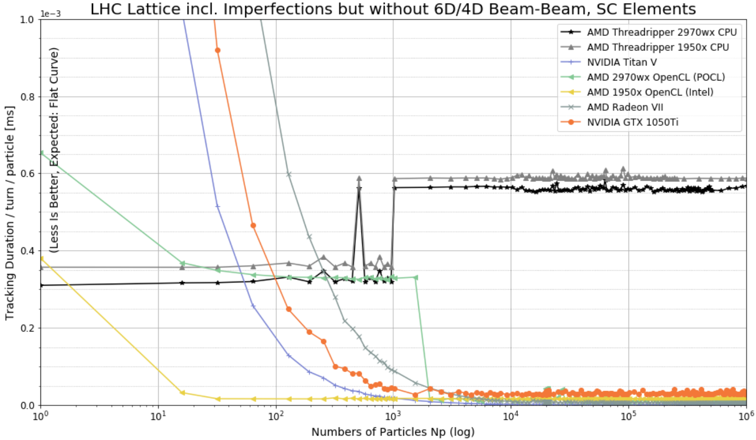
- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Limiting Factors For Parallel Performance

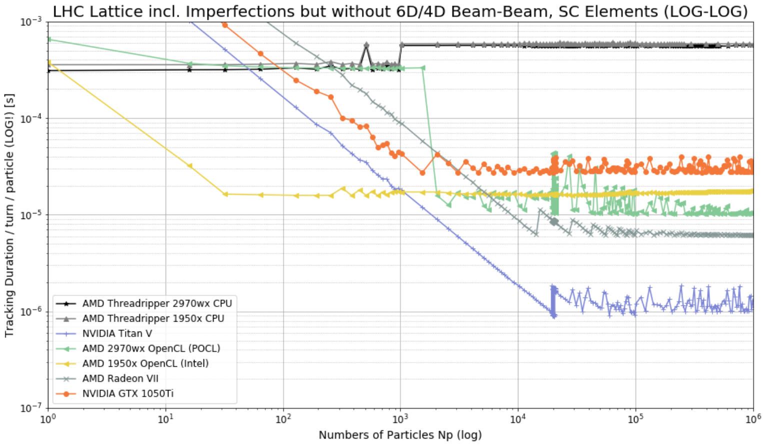
(**Remember**: single-particle tracking, "embarrassingly" parallel program)

- Sequential portion of the run-time t_s
  - Data-dependent branching in kernels (SPMD/SIMD) $\rightarrow$ renders data-dependent code-paths sequential
  - Limited bandwidth and finite latency in `collect_*` and `push_*` calls
  - Latency in starting kernels / waiting until kernel execution is finished
- Individual threads can not be scheduled on GPUs - code execution in multiples of **warp** / **wavefront** sizes (32/64 threads)
- Limited Available Resources (Registers, Shared Memory, etc.) $\longrightarrow$ number of threads that can be executed / scheduled concurrently is reduced
- Reduced number of warps/wavefronts in flight $\rightarrow$ less opportunity to mitigate I/O blocks and other latency issues by switching from a stalled to a warp/wavefront that can be executed $\rightarrow$ again, t_s increases

Performance Parallel Environment (GPUs & CPUs))



Performance Parallel Environment (GPUs & CPUs))



Using and Extending SixTrackLib: Real-World Scenarios

Usage & Integration Strategies For `SixTrackLib`

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_*`, `collect_*`, `push_*` API (C, C++, Python)
2. Compile And Launch Custom Kernel via `SixTrackLib` Infrastructure + use `track_line` to hand-off/take over from the custom kernel(Currently only OpenCL, C99; CUDA with NVRTC possible)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into `SixTrackLib` (C99, C++, Python)
5. Directly include C99 header-only subset of SixTrackLib into application kernel **or** link application against C99 or C++ API of `SixTrackLib` (C99 + Most Other Languages)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use track_*, collect_*, push_* API (C, C++, Python)
2. Compile And Launch Custom Kernel via SixTrackLib Infrastructure + use track_line to hand-off/take over from the custom kernel(Currently only OpenCL, C99; CUDA with NVRTC possible)
3. Share particles state "in-place" with other applications (zero-copy) together with track_line (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into SixTrackLib (C99, C++, Python)
5. Directly include C99 header-only subset of SixTrackLib into application kernel **or** link application against C99 or C++ API of SixTrackLib (C99 + Most Other Languages)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_*`, `collect_*`, `push_*` API (C, C++, Python)
2. Compile And Launch Custom Kernel via SixTrackLib Infrastructure + use `track_line` to hand-off/take over from the custom kernel(Currently only OpenCL, C99; CUDA with NVRTC possible)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into SixTrackLib (C99, C++, Python)
5. Directly include C99 header-only subset of SixTrackLib into application kernel **or** link application against C99 or C++ API of SixTrackLib (C99 + Most Other Languages)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use track_*, collect_*, push_* API (C, C++, Python)
2. Compile And Launch Custom Kernel via SixTrackLib Infrastructure + use track_line to hand-off/take over from the custom kernel(Currently only OpenCL, C99; CUDA with NVRTC possible)
3. Share particles state "in-place" with other applications (zero-copy) together with track_line (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into SixTrackLib (C99, C++, Python)
5. Directly include C99 header-only subset of SixTrackLib into application kernel **or** link application against C99 or C++ API of SixTrackLib (C99 + Most Other Languages)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_*`, `collect_*`, `push_*` API (C, C++, Python)
2. Compile And Launch Custom Kernel via `SixTrackLib` Infrastructure + use `track_line` to hand-off/take over from the custom kernel(Currently only OpenCL, C99; CUDA with NVRTC possible)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into `SixTrackLib` (C99, C++, Python)
5. Directly include C99 header-only subset of SixTrackLib into application kernel **or** link application against C99 or C++ API of `SixTrackLib` (C99 + Most Other Languages)

A Selection Of Usage Examples

1. Dynamic Aperture (DA), Beam-Stability, Resonances
   Carlo Emilio Montanari (Università di Bologna), Massimo Giovannozzi
2. Symplectic Kicks From An Electron Cloud
   Konstantinos Paraschou (AUTH,CERN), Giovanni Iadarola, et al
3. Simulating Beam-Beam Interactions & Space-Charge Effects
   Hannes Bartosik, Giovanni Iadarola, et al
4. Integrating SixTrackLib with PyHEADTAIL
   Adrian Oeftiger (GSI/FAIR)

1 Dynamic Aperture (DA), Beam-Stability, Resonances

- Study uses `SixTrackLib` directly to perform tracking for N turns
- Performs analysis and evaluation between turns on the host
- "Simple" use case - no extension and customisation was required

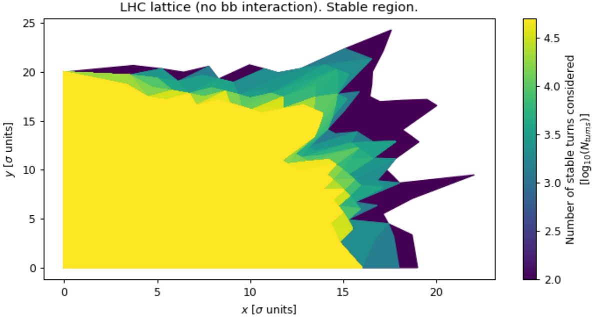


Figure: Sampling stable region via radial scans over N_{turns}

1 Dynamic Aperture (DA), Beam-Stability, Resonances

- Visualising $4D$ space ($r, \alpha, \Theta_1, \Theta_2$ is challenging - `SixTrackLib` helps with creating interactive views by being embeddable into parameterised visualisations

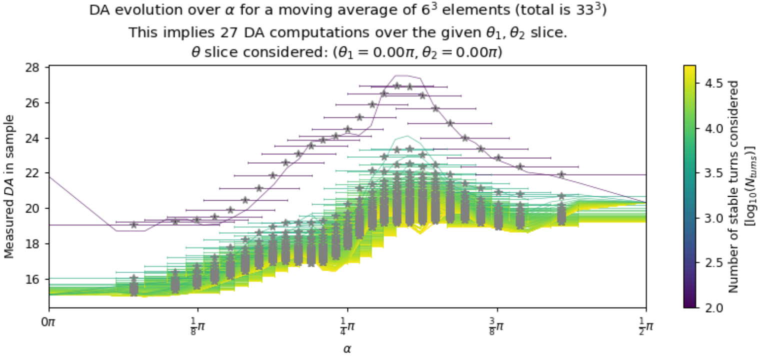


Figure: Evolution of r over α for a given Θ_1,Θ_2 slice over N_{turns}

1 Dynamic Aperture (DA), Beam-Stability, Resonances

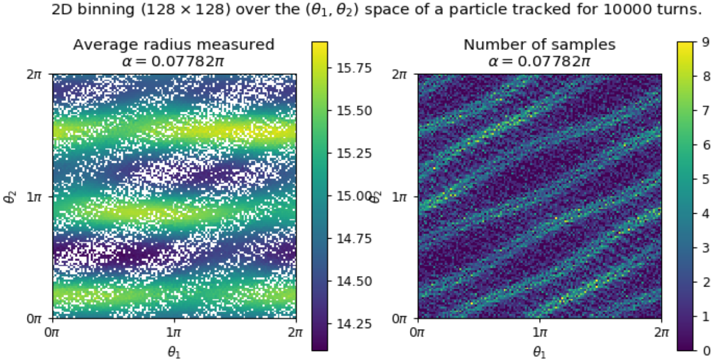


Figure: Histogram and average measured r over $_1$, Θ_2 plane in dependence of initial value for α

1 Dynamic Aperture (DA), Beam-Stability, Resonances

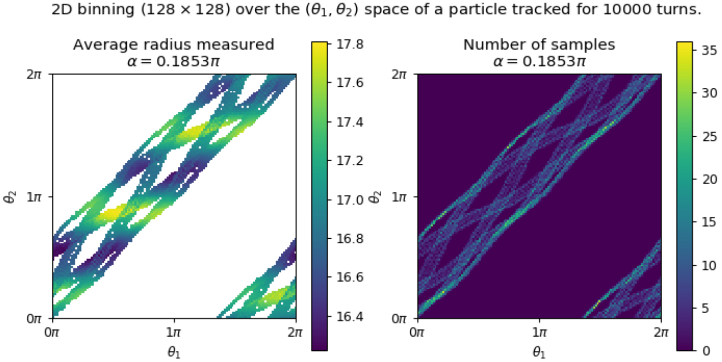
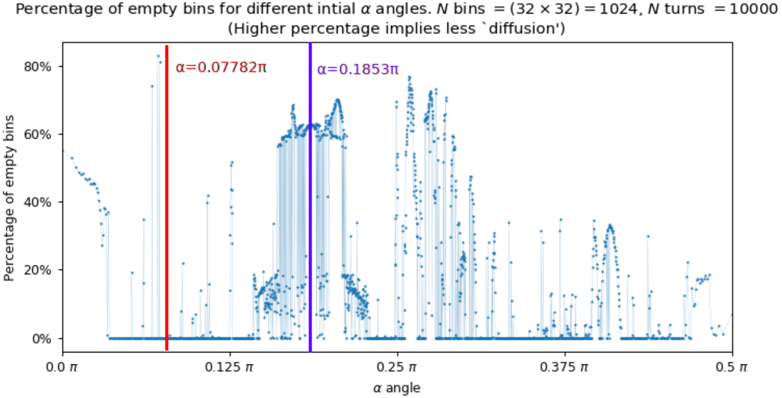


Figure: Histogram and average measured r over $_1$, Θ_2 plane in dependence of initial value for α

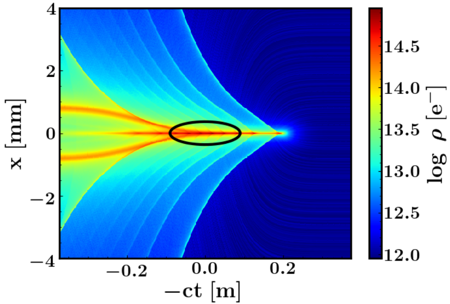
1 Dynamic Aperture (DA), Beam-Stability, Resonances



2 Symplectic Kicks From An Electron Cloud

For various reasons and under certain conditions (fulfilled in the LHC), there exists a complex distribution of electrons within the vacuum chamber that interacts with the beam called "**Electron Cloud**".
Distribution strongly depends on x, y and time! (as bunch passes through the electron cloud)

Example PyECLOUD simulation:



Particles with an amplitude of 1 beam-σ oscillate within the black line

Under usual approximations[0] the interaction can be written as a thin-lens through the Hamiltonian:

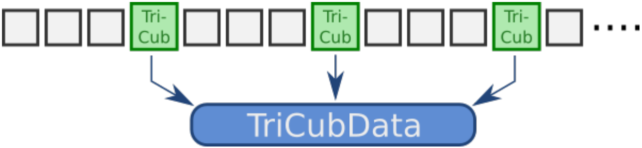
$$H(x, y, \tau; s) = \frac{qL}{\beta_0 P_0 c} \phi(x, y, \tau)\, \delta(s)$$

where ϕ is the scalar potential describing the electron cloud.

[0]see G. Iadarola, CERN-ACC-NOTE-2019-0033.

2 Symplectic Kicks From An Electron Cloud

- PyECLOUD would produce ϕ on a discrete grid (x, y, time)
  $\rightarrow$ ϕ should be **interpolated**
- To study slow effects, interpolation should produce symplectic kicks
  $\rightarrow$ Tricubic Interpolation: $\phi(x, y, \tau) = \sum_{i,j,k=0}^{3} a_{ijk} x^i y^j \tau^k$
- Add custom beam-element `TriCub` to implement the map
- N^3 coefficients with typically $N \sim \mathcal{O}(10^2)$ per `TriCub` element
  $\Rightarrow$ $\mathcal{O}(10^3)$ MByte of data for each `TriCub`
- But: interpolation data can be shared between many beam-elements (e.g. All focusing quadrupole magnets have similar Electron Cloud)
- **Idea:** implement infrastructure to store data externally from `TriCub` elements and assign & share coefficient data



2 Symplectic Kicks From An Electron Cloud

- In principle, TriCub element general enough to describe any interaction whose Hamiltonian can be discretized on a grid of (x,y,τ)
- GPUs: large global memory (4-16 GByte), adequate memory bandwidth $\rightarrow$ perfect environment for simulations with TriCub beam-elements.

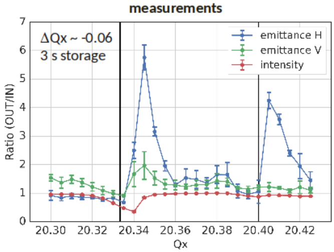
3 Beam-Beam Interactions & Space-Charge Effects

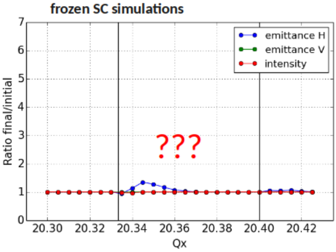
- SixTrackLib implements $4D$ and $6D$ beam-beam (BB) interactions using a weak-strong beam formulation[2]
- Frozen Space-Charge (SC) beam-elements share infrastructure with the BB implementation
  - Coasting `SpaceChargeCoasting`
  - Bunched `SpaceChargeQGaussianProfile`
  - Bunched `SpaceChargeInterpolatedProfile` using linear and cubic spline longitudinal interpolation (under development)
- `SpaceChargeInterpolatedProfile` uses API to assign external data to a number of beam-elements to share profile samples and interpolation parameters between SC elements

[2] G. Iadarola et al. CERN-ACC-NOTE-2018-0023 "6D beam-beam interaction step-by-step

Observations from CERN SPS experiment

- **Benchmark experiment**
  - Horizontal 3rd order resonance at Qx = 20.33 deliberately excited
  - Additional resonance observed at Qx = 20.40 (space charge driven)

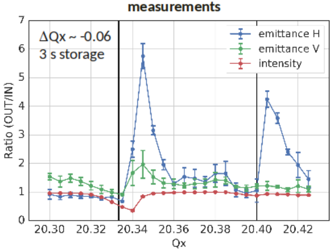




[2] H. Bartosik, F. Schmidt "Studies on Tune Ripple", 4th ICFA Mini-Workshop on SpaceCharge 2019, https://indico.cern.ch/event/828559/contributions/3528378

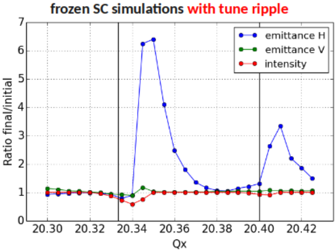
Observations from CERN SPS experiment

- **Benchmark experiment**
  - Horizontal 3rd order resonance at Qx = 20.33 deliberately excited
  - Additional resonance observed at Qx = 20.40 (space charge driven)
  - Simulations with **frozen potential** far from experiment unless SPS **tune ripple** from quadrupole power converters is taken into account



With CPU only impl. 4: 5e3 Particles ~ 4 Days

With SixTrackLib: 20e3 Particles ~ 4 Hours



Simulation over 130000 turns

After each turn: collect, update quadrupoles, push!

[2] H. Bartosik, F. Schmidt "Studies on Tune Ripple", 4th ICFA Mini-Workshop on SpaceCharge 2019, https://indico.cern.ch/event/828559/contributions/3528378

4 Integrating SixTrackLib with PyHEADTAIL

Beyond the single-particle treatment within SixTrackLib, model collective effects as "true" interaction between macro-particles via PyHEADTAIL[3]:

- accelerated on the GPU via (Py)CUDA
- self-consistent models for (e.g. 3D PIC/particle-in-cell) space charge, wake fields and feedback systems

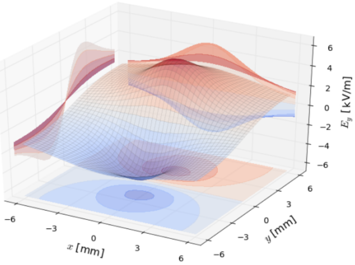


Figure: PIC space charge

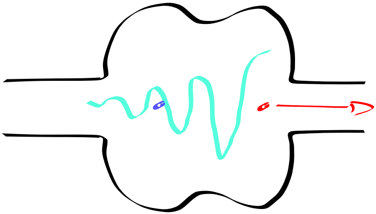


Figure: wake fields

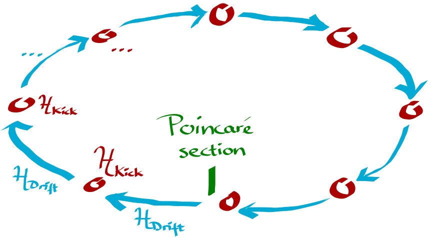
[3] https://github.com/PyCOMPLETE/PyHEADTAIL

4 Integrating SixTrackLib with PyHEADTAIL

Share particle memory between `SixTrackLib` and `PyHEADTAIL`:

1. use SixTrackLib's `track_line` API to advance particles through parts of accelerator lattice
2. expose particle coordinates on GPU via SixTrackLib's `get_particle_addresses` interface to apply kick in PyHEADTAIL

$\Longrightarrow$ alternating single- and multi-particle physics while *remaining* on GPU device memory!

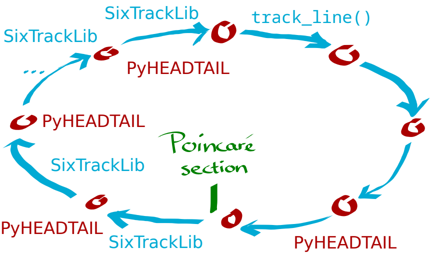


4 Integrating SixTrackLib with PyHEADTAIL

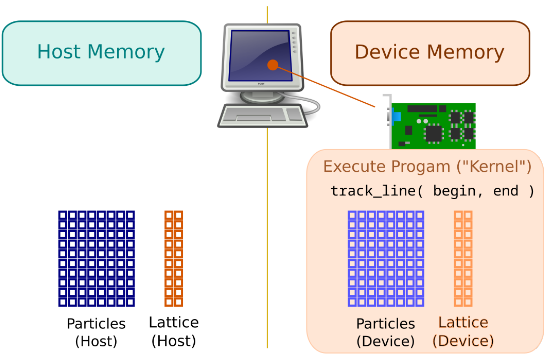
Share particle memory between SixTrackLib and PyHEADTAIL:

1. use SixTrackLib's track_line API to advance particles through parts of accelerator lattice
2. expose particle coordinates on GPU via SixTrackLib's get_particle_addresses interface to apply kick in PyHEADTAIL

$\Longrightarrow$ alternating single- and multi-particle physics while *remaining* on GPU device memory!

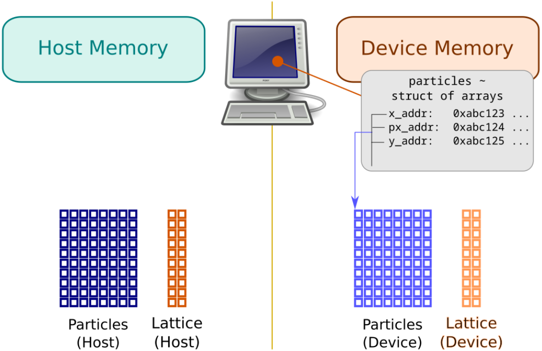


SideBar: How Does Address Sharing Work?



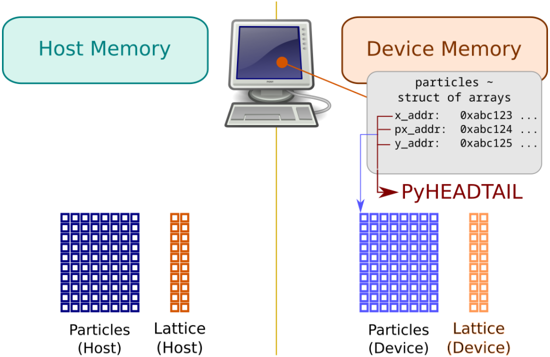
Icons: https://openclipart.org - License: Public Domain

SideBar: How Does Address Sharing Work?



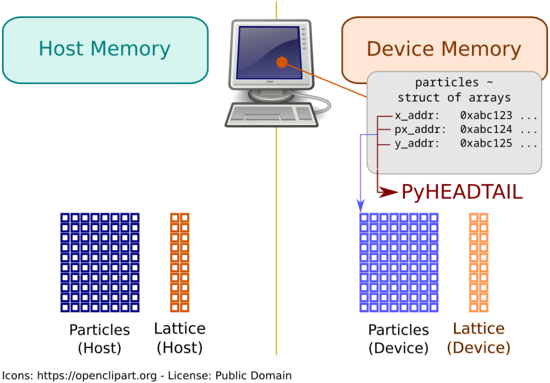
Icons: https://openclipart.org - License: Public Domain

SideBar: How Does Address Sharing Work?



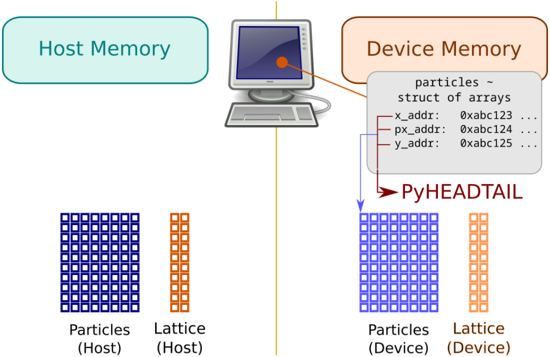
Icons: https://openclipart.org - License: Public Domain

SideBar: How Does Address Sharing Work?



- CUDA: Memory is managed via raw pointers → works
- But: Resource Management, Lifetime Management, Context & Device Selection → very difficile
- OpenCL: memory is managed via `cl_mem` Objects → more challenging
- Idea: Use OpenCL 2.x feature SVM → pointers again
- We are working on a proof of concept implementation for OpenCL

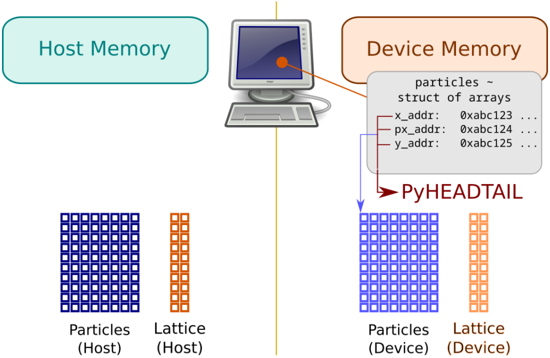
SideBar: How Does Address Sharing Work?



Icons: https://openclipart.org - License: Public Domain

- CUDA: Memory is managed via raw pointers → works
- But: Resource Management, Lifetime Management, Context & Device Selection → very difficile
- OpenCL: memory is managed via `cl_mem` Objects → more challenging
- Idea: Use OpenCL 2.x feature SVM → pointers again
- We are working on a proof of concept implementation for OpenCL

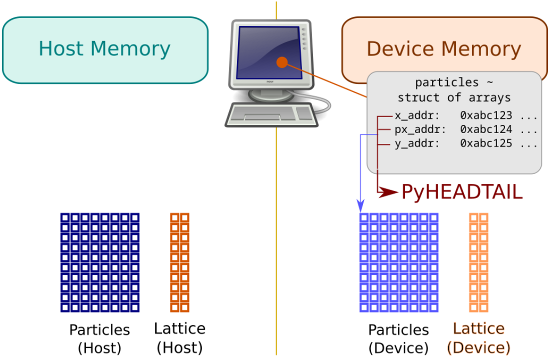
SideBar: How Does Address Sharing Work?



Icons: https://openclipart.org - License: Public Domain

- CUDA: Memory is managed via raw pointers → works
- But: Resource Management, Lifetime Management, Context & Device Selection → very difficile
- OpenCL: memory is managed via `cl_mem` Objects → more challenging
- Idea: Use OpenCL 2.x feature SVM → pointers again
- We are working on a proof of concept implementation for OpenCL

SideBar: How Does Address Sharing Work?



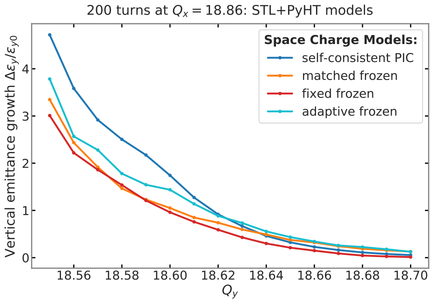
Icons: https://openclipart.org - License: Public Domain

- CUDA: Memory is managed via raw pointers → works
- But: Resource Management, Lifetime Management, Context & Device Selection → very difficile
- OpenCL: memory is managed via `cl_mem` Objects → more challenging
- Idea: Use OpenCL 2.x feature SVM → pointers again
- We are working on a proof of concept implementation for OpenCL

4 Integrating SixTrackLib with PyHEADTAIL

Space Charge Model Benchmarking

Comparison between realistic (computationally demanding) PIC and approximative frozen (fast) space charge models for half-integer stop-band:



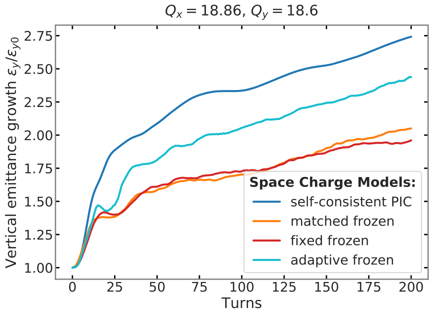


Figure: ICFA Beam Dynamics Newsletter #79, SIS100 contribution

$\Longrightarrow$ choose from variety of space charge models for identical lattice

Applications of SixTrackLib + PyHEADTAIL

90 deg stop-band

Interplay of coherent vs. incoherent resonances driven by space charge

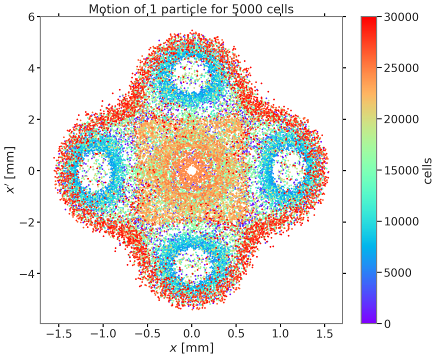


Figure: running 3D PIC in FODO

FAIR synchrotron SIS100

Beam loss studies with space charge and nonlinear magnet imperfections

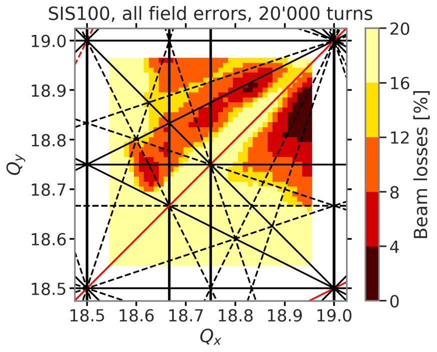


Figure: frozen SC in SIS100 lattice

Applications of SixTrackLib + PyHEADTAIL

90 deg stop-band

Interplay of coherent vs. incoherent resonances driven by space charge

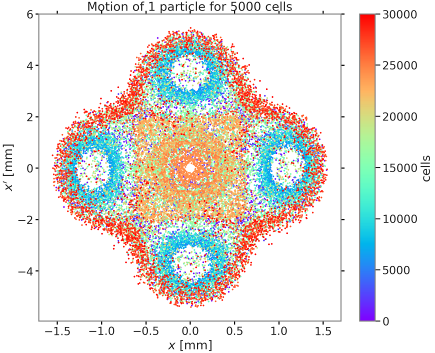


Figure: running 3D PIC in FODO

FAIR synchrotron SIS100

Beam loss studies with space charge and nonlinear magnet imperfections

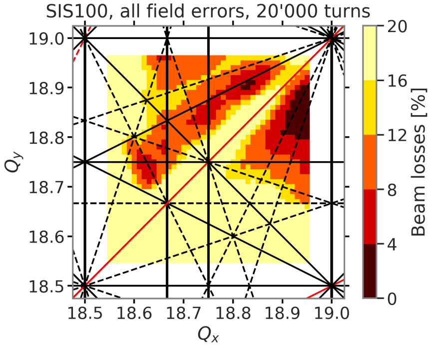
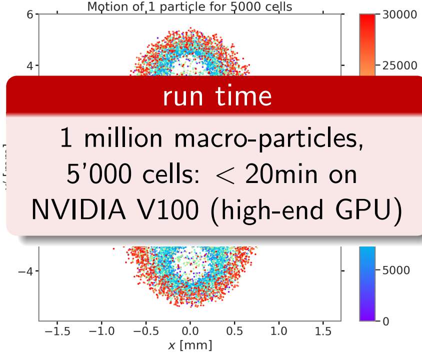


Figure: frozen SC in SIS100 lattice

Applications of SixTrackLib + PyHEADTAIL

90 deg stop-band

Interplay of coherent vs. incoherent resonances driven by space charge



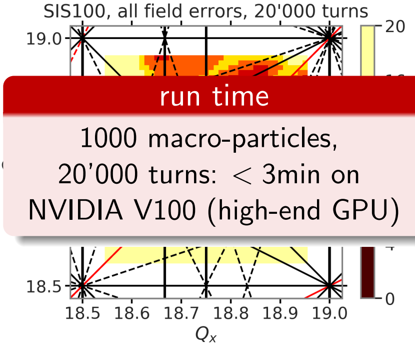
run time

1 million macro-particles, 5'000 cells: $<$ 20min on NVIDIA V100 (high-end GPU)

Figure: running 3D PIC in FODO

FAIR synchrotron SIS100

Beam loss studies with space charge and nonlinear magnet imperfections



run time

1000 macro-particles, 20'000 turns: $<$ 3min on NVIDIA V100 (high-end GPU)

Figure: frozen SC in SIS100 lattice

Summary & Outlook

- Delivering scalable single-particle tracking on massively parallel systems to users without GPU programming Know-How is possible :-)
- Retaining symplectivity is crucial for studying effects over $N >> 1$ turns
- `SixTrackLib` is still under heavy development but already useful in controlled settings with early adopters
- Still a lot of work to do, especially concerning optimisation and numerical stability & reproducibility

Summary & Outlook

- Delivering scalable single-particle tracking on massively parallel systems to users without GPU programming Know-How is possible :-)
- Retaining symplectivity is crucial for studying effects over $N >> 1$ turns
- `SixTrackLib` is still under heavy development but already useful in controlled settings with early adopters
- Still a lot of work to do, especially concerning optimisation and numerical stability & reproducibility

Summary & Outlook

- Delivering scalable single-particle tracking on massively parallel systems to users without GPU programming Know-How is possible :-)
- Retaining symplectivity is crucial for studying effects over $N >> 1$ turns
- `SixTrackLib` is still under heavy development but already useful in controlled settings with early adopters
- Still a lot of work to do, especially concerning optimisation and numerical stability & reproducibility

Summary & Outlook

- Delivering scalable single-particle tracking on massively parallel systems to users without GPU programming Know-How is possible :-)
- Retaining symplectivity is crucial for studying effects over $N >> 1$ turns
- `SixTrackLib` is still under heavy development but already useful in controlled settings with early adopters
- Still a lot of work to do, especially concerning optimisation and numerical stability & reproducibility

Thank You For Your Attention!

Extra Slides

2 Symplectic Kicks From An Electron Cloud

```c
/* From be_tricub/be_tricub.h */

typedef struct NS(TriCub)
{
    NS(be_tricub_real_t)    x_shift           SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    y_shift           SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    tau_shift         SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dipolar_kick_px   SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dipolar_kick_py   SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dipolar_kick_ptau SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    length            SIXTRL_ALIGN( 8 );
    NS(buffer_addr_t)       data_addr         SIXTRL_ALIGN( 8 );
}
NS(TriCub);
```

"Pointer" to external TriCubData

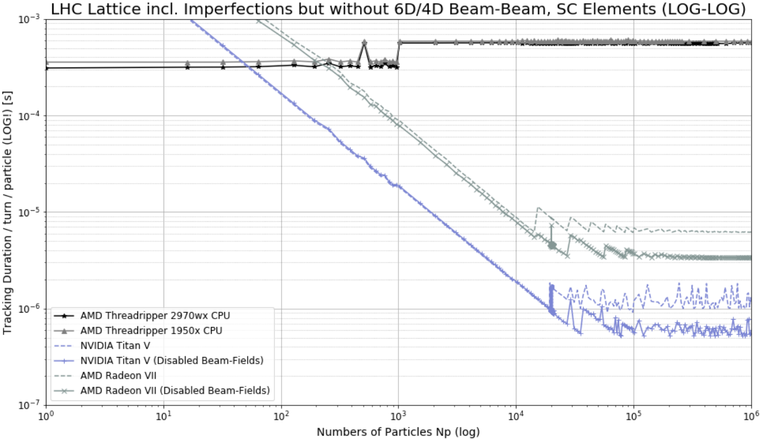
```c
/* From be_tricub/track.h */

SIXTRL_INLINE NS(track_status_t) NS(Track_particle_tricub)(
    SIXTRL_PARTICLE_ARGPTR_DEC NS(Particles)* SIXTRL_RESTRICT particles,
    NS(particle_num_elements_t) const ii,
    SIXTRL_BE_ARGPTR_DEC const struct NS(TriCub) *const SIXTRL_RESTRICT tricub )
{
    typedef NS(be_tricub_real_t) real_t;
    typedef NS(be_tricub_int_t)  int_t;

    SIXTRL_BUFFER_DATAPTR_DEC NS(TriCubData) const* tricub_data =
        NS(TriCub_const_data)( tricub );

    real_t const length = NS(TriCub_length)( tricub );

    real_t const x_shift = NS(TriCub_x_shift)( tricub );
    real_t const y_shift = NS(TriCub_y_shift)( tricub );
    real_t const z_shift = NS(TriCub_tau_shift)( tricub );

    // method = 1 -> Finite Differences for derivatives (Do not use)
    // method = 2 -> Exact derivatives
    // method = 3 -> Exact derivatives and mirrored in X, Y

    real_t const inv_dx = 1./( NS(TriCubData_dx)( tricub_data ) );
    real_t const inv_dy = 1./( NS(TriCubData_dy)( tricub_data ) );
    real_t const inv_dz = 1./( NS(TriCubData_dz)( tricub_data ) );

    real_t const x0 = NS(TriCubData_x0)( tricub_data );
    real_t const y0 = NS(TriCubData_y0)( tricub_data );
    real_t const z0 = NS(TriCubData_z0)( tricub_data );

    real_t const zeta  = NS(Particles_get_zeta_value)( particles, ii );
    real_t const rvv   = NS(Particles_get_rvv_value)( particles, ii );
    real_t const beta0 = NS(Particles_get_beta0_value)( particles, ii );

```

```c
typedef struct NS(TriCubData)
{
    NS(be_tricub_real_t)    x0          SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dx          SIXTRL_ALIGN( 8 );
    NS(be_tricub_int_t)     nx          SIXTRL_ALIGN( 8 );

    NS(be_tricub_real_t)    y0          SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dy          SIXTRL_ALIGN( 8 );
    NS(be_tricub_int_t)     ny          SIXTRL_ALIGN( 8 );

    NS(be_tricub_real_t)    z0          SIXTRL_ALIGN( 8 );
    NS(be_tricub_real_t)    dz          SIXTRL_ALIGN( 8 );
    NS(be_tricub_int_t)     nz          SIXTRL_ALIGN( 8 );

    NS(be_tricub_int_t)     mirror_x    SIXTRL_ALIGN( 8 );
    NS(be_tricub_int_t)     mirror_y    SIXTRL_ALIGN( 8 );
    NS(be_tricub_int_t)     mirror_z    SIXTRL_ALIGN( 8 );

    NS(buffer_addr_t)       table_addr  SIXTRL_ALIGN( 8 );
}
NS(TriCubData);

```

"Pointer" managed by CObjects Buffer
→ tricubic coefficients stored in
linear array

- Implemented in external branch kparasch_master
  https://github.com/martinschwinzerl/sixtracklib/tree/kparasch_master
- To be merged into SixTrack/sixtracklib:master (PR#123)

Impact of Kernel Complexity On Parallel Performance))



Impact of Kernel Complexity On Parallel Performance

- Calculation of field components (according to a Gaussian distribution) and the complex error function (Faddeeva function) is shared between BB and SC elements

```
/* From: be_beamfields/faddeeva_cern.h */
SIXTRL_INLINE void cerrf( SIXTRL_REAL_T in_real, SIXTRL_REAL_T in_imag,
    SIXTRL_ARGPTR_DEC SIXTRL_REAL_T* SIXTRL_RESTRICT out_real,
    SIXTRL_ARGPTR_DEC SIXTRL_REAL_T* SIXTRL_RESTRICT out_imag )
{
    /* This function calculates the SIXTRL_REAL_T precision complex error fnct.
    based on the algorithm of the FORTRAN function written at CERN by K. Koelbig
    Program C335, 1970. See also M. Bassetti and G.A. Erskine, "Closed
    expression for the electric field of a two-dimensional Gaussian charge
    density", CERN-ISR-TH/80-06; */

    int n, nc, nu;
    SIXTRL_REAL_T a_constant = 1.12837916709551;
    SIXTRL_REAL_T xLim = 5.33;
    SIXTRL_REAL_T yLim = 4.29;
    SIXTRL_REAL_T h, q, Saux, Sx, Sy, Tn, Tx, Ty, Wx, Wy, xh, xl, x, yh, y;
    SIXTRL_REAL_T Rx [33]; } !!!
    SIXTRL_REAL_T Ry [33];

    x = fabs(in_real);
    y = fabs(in_imag);

    if (y < yLim && x < xLim){
        q = (1.0 - y / yLim) * sqrt(1.0 - (x / xLim) * (x / xLim));
        h  = 1.0 / (3.2 * q);
        nc = 7 + (int) (23.0 * q);
        xl = pow(h, (SIXTRL_REAL_T) (1 - nc));
        xh = y + 0.5 / h;
        yh = x;
        nu = 10 + (int) (21.0 * q);
        Rx[nu] = 0.;
        Ry[nu] = 0.;
        for (n = nu; n > 0; n--){
            Tx = xh + n * Rx[n];
            Ty = yh - n * Ry[n];
            Tn = Tx*Tx + Ty*Ty;
            Rx[n-1] = 0.5 * Tx / Tn;
            Ry[n-1] = 0.5 * Ty / Tn;
            }
        /* .... */
```

Large amount of thread-local data required → Effects Execution of Kernel even if the function is not actually called!

Data-Dependent Branching