12th Int. Particle Acc. Conf. IPAC2021, Campinas, SP, Brazil JACoW Publishing
ISBN: 978-3-95450-214-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2021-THPAB190

OPTIMISING AND EXTENDING A SINGLE-PARTICLE TRACKING LIBRARY FOR HIGH PARALLEL PERFORMANCE

M. Schwinzerl*[1], R. De Maria, K. Paraschou, H. Bartosik, G. Iadarola, CERN, Geneva, Switzerland
A. Oeftiger, GSI, Darmstadt, Germany
[1]also at Institute of Mathematics and Scientific Computing, University of Graz, Austria

Abstract

SixTrackLib is a library for performing tracking simulations on highly parallel systems such as shared memory multi-core processors or graphical processing units (GPUs). Its single-particle approach fits very well to parallel implementations with reasonable base-line performance, making such a library an interesting building block for various use cases, including simulations covering collective effects. We describe the optimisations applied to SixTrackLib to improve its performance on its main target platforms and the associated performance gain. Furthermore we outline the technical interfaces and extensions implemented to allow its use in a wider range of applications and studies.

INTRODUCTION

SixTrackLib [1] is a clean-room re-implementation of the tracking component of the SixTrack [2, 3] simulation program in the form of a stand-alone library. It is suitable for stand-alone use in simulations or as a building-block in feature-rich tools and libraries. SixTrackLib is a *single-particle* tracking code, the idealised constituent particles of a beam only interacting with their environment via electro-magnetic fields but not directly with other particles. Particles p are described in a $6D$ (i.e. $4D$ transversal and $2D$ longitudinal) phase space in relation to the trajectory of a *reference particle* (cf. Fig. 1). The accelerator through

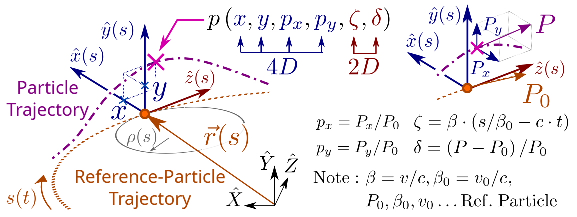

$p\left(x, y, p_x, p_y, \zeta, \delta\right)$
$4D$
$2D$
$\hat{y}(s)$
$\hat{x}(s)$
$\hat{z}(s)$
x
y
Particle Trajectory
$\rho(s)$
$\vec{r}(s)$
Reference-Particle Trajectory
$s(t)$
$\hat{X}$
$\hat{Y}$
$\hat{Z}$
P_y
P
P_x
P_0
$p_x = P_x/P_0 \quad \zeta = \beta \cdot (s/\beta_0 - c \cdot t)$
$p_y = P_y/P_0 \quad \delta = (P - P_0)/P_0$
Note : $\beta = v/c, \beta_0 = v_0/c,$
$P_0, \beta_0, v_0 \ldots$ Ref. Particle

Figure 1: $6D$ representation of a particle p in a circular accelerator (local bending radius $\rho(s)$). The coordinate system $\hat{x}(s) - \hat{y}(s) - \hat{z}(s)$ co-moves with the reference particle.

which p travels is represented by a *lattice*, i.e. a sequence of idealised and discrete *beam elements*. These elements represent the electro-magnetic (EM) field configurations encountered within the accelerator as a function of the spatial position. Thus solving the equations of motion for p turns into a problem of integrating each beam element's influence on p along it's path. In addition to conveying EM fields, SixTrackLib's implementation of beam element objects mark particles outside of admissible bounds as *lost*.

* martin.schwinzerl@cern.ch

IMPLEMENTATION

Tracking Algorithm, Parallelisation Strategy

Consider a lattice with N_{elem} beam elements i.e., $E_i \equiv (\mathbf{E})[i]$ and a particle p with initial conditions $p(0)$. Rather than performing the numerical integration explicitly, SixTrackLib uses *symplectic maps* to sequentially update the state of p consistent with the piece-wise solutions to the *Hamiltonian equations* for each E_i:

$$p(i) \leftarrow E_i(p(i-1)) \quad i \in [0, N_{elem}). \tag{1}$$

Applying (1) for all i and repeating the operation until p has traversed until turn N in a circular accelerator is an inherently sequential operation with only very few opportunities for parallelisation[1]. While performing the simulation for a single p is useful, most real-world scenarios require to simulate an *ensemble* of N_p particles i.e., $p_j \equiv (\mathbf{Q})[j]$, $j \in [0, N_p)$.

Algorithm 1 Track all active particles in $(\mathbf{Q})$ over a lattice $(\mathbf{E})$ until all particles are in turn N or they are lost.

```
 1: procedure TRACK_UNTIL($(\mathbf{Q}), (\mathbf{E}), N$)
 2:     for $j \leftarrow 0$ to $N_p - 1$ do
 3:         $p_j \equiv (\mathbf{Q})[j]$
 4:         while ( not is_lost($p_j$) and get_at_turn($p_j$) < $N$ ) do
 5:             for $i \leftarrow 0$ to $N_{elem} - 1$ do
 6:                 $E_i \equiv (\mathbf{E})[i]$
 7:                 $p_j \leftarrow E_i(p_j)$
 8:                 if is_lost($p_j$) then
 9:                     break
10:             if not is_lost($p_j$) then
11:                 increment_at_turn($p_j$)
```

Note that particles need at least some additional state to keep track of the current turn and whether the particle has been lost. For $N_p \gg 1$, the loop over all particles (line 2 in Alg. 1) allows, due to the single-particle approach, a very efficient parallelisation.

Baseline Parallel Implementation

Typical use-cases for Alg. 1 have $N_p = 10^0$ to 10^7 particles, $N_{elem} = 10^1$ to 10^5 elements in a lattice, and $N = 10^0$ to $> 10^7$ turns. Given this wide range of scale, SixTrackLib

[1] I.e., OpenMP-style loop parallelisation, or *single-instruction, multiple-data* (SIMD) vectorisation, which both apply only to specific maps

supports sequential and multiple parallel computational back-ends (single-threaded and vectorised code for CPUs, and both OpenCL 1.2 [4] and CUDA [5], respectively). Source code for implementing the physics is shared between all three back-ends. A common implementation in particular for OpenCL and CUDA is possible because both abstract the target hardware in a similar way (i.e., "threads" organised on hierarchical grid, "teams" of threads operate in lockstep, segmented memory, etc.), and both support a similarly structured C99-like kernel language, allowing abstractions via C pre-processor macros.

In addition to the principal six degrees of freedom and the state variables to enable the "logistical" particle loss operations in Alg. 1, SixTrackLib uses 11 additional double-precision floating point and 2 additional integer attributes for each particle to cache frequently required auxiliary results and carry information about the reference particle. Ensembles of particles ($\mathbf{Q}$) are stored in *struct-of-array* arrangement. Storing a complete set of of attributes for each p_j in this fashion increases the memory footprint but provides a more flexible data model, eases SIMD-style vectorisation across particles, and results in neighbouring GPU threads accessing neighbouring sections of device memory (*coalesced* access, cf. sub-figure a) in Fig. 2). Caching intermediate results is also beneficial for numerical reproducibility reasons. Preparing a structure with pointer data-members on the host-side and transferring it to device memory on a GPU is non-trivial as the member pointers remain pointing to their original location. In order to remain consistent, all such pointers have to be "remapped" after any copy operation (cf. sub-figure b) of Fig. 2). To this end, SixTrackLib uses cobjects [6], a self-developed container library for exchanging data between host and GPU devices. The buffers provided by cobjects allow $\mathcal{O}(1)$ lookup of stored elements and zero-copy, zero-overhead read/write access on stored items[2]. For the base-line implementation, both particles and beam elements are stored using device *global memory*.

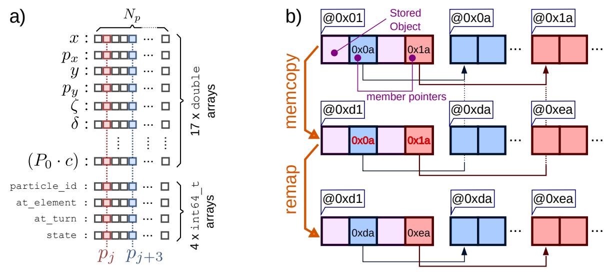


Figure 2: a) Struct-of-arrays storage model for sets of particles. b) Schematic principle for storing structured objects with member pointers in a cobjects buffer.

Feature Extensions & Interfaces

Using cobject buffers does not preclude direct manipulation of pointers on the device side. SixTrackLib provides tools to manipulate device-side pointers, so that sharing slices of data across several beam elements is feasible. An example is the calculation of symplectic kicks from electron cloud contributions via tri-cubic interpolation of a potential [7], which requires data-tables with approx. 10^8 to 10^9 Bytes each. Replicating these tables at each required lattice position would not be possible due to the cumulative global memory requirements. However, by allowing tables to be shared across all elements using the same data (cf. Fig. 3), intensive simulations of this type have successfully been performed using SixTrackLib. Similarly, SixTrackLib implements a *frozen space charge* model which, among others, features an interpolated line-charge density profile, allowing the discretised line-profile data to be shared among the beam elements [8]. It uses the same interfaces as the previous example but the main aim here is to simplify updates to the charge-density profile over the course of the simulation.

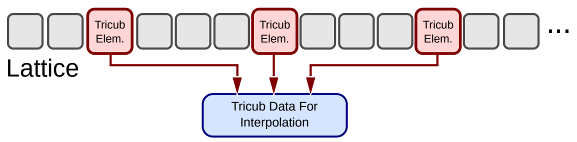


Figure 3: Multiple items in a cobjects buffer sharing data.

Finally, by implementing tracking modes that traverse only a subset of a turn in addition to Alg. 1 and exposing the particle data stored in device memory, SixTrackLib allows a seamless integration and hand-off of tracking duties with other algorithms and libraries. This enables for example seamless integration between PyHEADTAIL [9] and SixTrackLib. Here, SixTrackLib handles the tracking of sections that are suitable for efficient single-particle tracking code and hands-off to PyHEADTAIL for sections of the machine representing collective effects. These extensions and additional interfaces increase the versatility and usefulness of SixTrackLib, but also put additional constraints on some strategies to improve the numerical performance.

PERFORMANCE ANALYSIS & OPTIMISATION

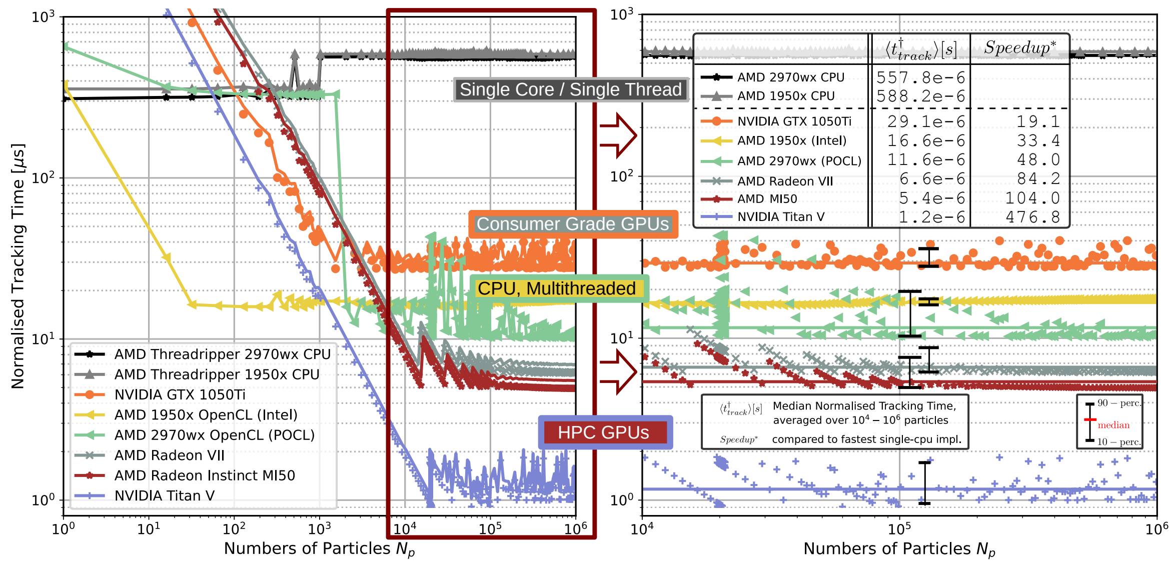
Baseline Performance

The implementation described in the previous section corresponds to version 0.5 of SixTrackLib [1]. Using an example lattice from CERN's LHC with imperfections but without beam-beam or space charge effects. Figure 4 shows the normalised tracking time $t_{track} = t_{elapsed}/(N_p \cdot N)$ as a function of N_p, for a representative set of target systems.

With increasing N_p, parallelisation overheads and latencies should eventually be amortised, resulting in approx. constant t_{track}. All presented systems show this expected behaviour. Even with grid dimensions adapted to preferable warp/wave-front sizes, run-time performance can vary considerably for similar N_p. Therefore, the median t_{track} over the range most interesting for GPUs i.e., $N_p = 10^4$ to 10^6 (cf. detail in right half of Fig. 4) is used to calculate the

[2] At the cost of increased complexity and reduced flexibility during the creation / initial arrangment of items in a cobjects buffer

ISBN: 978-3-95450-214-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2021-THPAB190



	$\langle t^{\dagger}_{track} \rangle [s]$	Speedup*
AMD 2970wx CPU	557.8e-6	
AMD 1950x CPU	588.2e-6	
NVIDIA GTX 1050Ti	29.1e-6	19.1
AMD 1950x (Intel)	16.6e-6	33.4
AMD 2970wx (POCL)	11.6e-6	48.0
AMD Radeon VII	6.6e-6	84.2
AMD MI50	5.4e-6	104.0
NVIDIA Titan V	1.2e-6	476.8

Figure 4: Normalised tracking time (i.e., elapsed wall-time per particle and turn, lower numbers are better) as a function of N_p on a selection of hardware targets for the CERN LHC lattice. $(P_0 \cdot c) = 6.5$ TeV, starting conditions have been chosen such that no particles are lost.

speed-ups. For the presented (quite representative) scenario, even lower-end consumer GPUs with poor double precision performance ratios (i.e., 1:32 for the GTX1050 Ti) yield median speedups of $> \mathcal{O}(10)$ compared to sequential tracking on the fastest tested CPU. Using more powerful GPUs with a 1:2 double precision performance ratio, speedups of $\geq \mathcal{O}(100)$ are easily reachable. Using multi-threaded OpenCL parallelisation on CPUs typically yields performance in between consumer- and HPC-grade GPUs. For these systems, optimal scaling (i.e. constant t_{track}) sets in at lower $N_p = 10^1 \dots 10^3$, depending on the parallel runtime and CPU). The sequential, single-core CPU back-end of SixTrackLib also performs approx. $> \mathcal{O}(10)$ better than recent versions of MAD-X ($< 10^{-3} s$ vs. $\simeq 0.01 s$ for the LHC [10])[3], further demonstrating the competitiveness of SixTrackLib.

Optimisation Strategy

In order to improve the run-time performance, the following optimisation options have been implemented compared to Alg. 1: a) create a private copy of p_j for each thread (i.e., change line 3) and write back after finishing tracking, b) eliminate the nested loop over all elements (cf. line 5) by introducing a special lattice terminating beam element which handles the roll-over into the next turn, thus simplifying the logic and aiding the compilers in optimisation, and c) reduce the number of thread-local variables to reduce register pressure and avoid *spilling*.

Selected Results

Applying optimisations a) to c) to the same configuration, numerical experiments conducted at CERN and GSI reveal performance gains across all studied GPU systems. Figure 5 shows the improvements for the AMD Radeon VII and NVIDIA Titan-V cards, with typical improvement factors around 2× (cf. [11]).

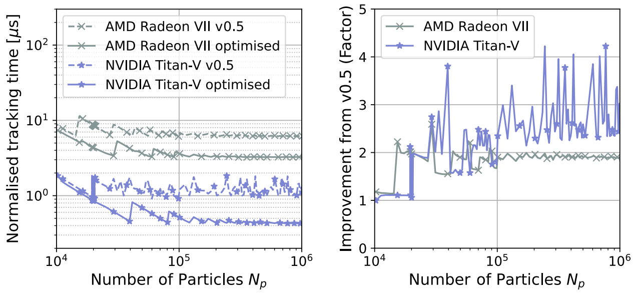


Figure 5: Same lattice and particle configuration as before but with optimisations, run-time performance improves for the presented GPU systems, typically by about a factor of 2.

CONCLUSIONS & OUTLOOK

Writing a tracking library with good parallel performance across a large range of simulated particles N_p and for a diverse set of hardware is feasible. The presented optimised implementation provides satisfactory performance: sorted by increasing speedup, simulations profit from parallelisation on consumer-grade GPUs, multi-core high-end CPUs and finally high-end GPUs. Further investigations about the contributing factors to the scaling behaviour are needed. Of particular interest are lattices containing beam elements contributing significantly to the register pressure. Scenarios where the extended interfaces for sharing data are not required could benefit from moving the lattice or parts of the particle data to constant or shared device memory, warranting further investigations. The presented optimisations will be part of the upcoming version 1.0 of SixTrackLib.

[3] Comparisons are only meaningful up to orders of magnitude due to the significantly wider range of capabilities and the large number of parameters influencing the performance of MAD-X.

REFERENCES

[1] SixTrackLib source code repository, https://github.com/SixTrack/sixtracklib

[2] R. De Maria *et al.*, "SixTrack V and runtime environment", *International Journal of Modern Physics A*, vol. 34, no. 36, p. 1942035, 2019. doi:10.1142/S0217751X19420351

[3] SixTrack source code repository, https://github.com/SixTrack/sixtrack

[4] J. E. Stone *et al.* "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems", *Computing in Science & Engineering*, vol. 12, no. 3, p. 66–72, 2010. doi:10.1109/mcse.2010.69

[5] J. Nickolls *et al.*, "Scalable Parallel Programming with CUDA", *Queue*, vol. 6, no. 2, p. 40 – 53, 2008. doi:10.1145/1365490.1365500

[6] CObjects source code repository, https://github.com/SixTrack/cobjects

[7] K. Paraschou and G. Iadarola, "Incoherent electron cloud effects in the Large Hadron Collider", *CERN Yellow Rep. Conf. Proc.*, vol. 9, pp. 249–255, Dec. 2020. doi:10.23732/CYRCP-2020-009.249

[8] A. Oeftiger *et al.*, "Simulation study of the space charge limit for the FAIR Heavy-ion Synchrotron SIS100", 2021, to be published.

[9] PyHEADTAIL source code repository, https://github.com/PyCOMPLETE/PyHEADTAIL

[10] T. H. B. Persson, H. Burkhardt, L. Deniau, A. Latina, and P. K. Skowronski, "MAD-X for Future Accelerators", presented at the 12th Int. Particle Accelerator Conf. (IPAC'21), Campinas, Brazil, May 2021, paper WEPAB028.

[11] ipac21_sixtracklib: Complementary repository containing raw timing data and information about the presented hardware systems, https://github.com/martinschwinzerl/ipac21_sixtracklib