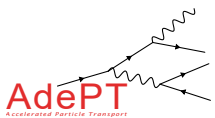
AdePT implementation overview

HSF Detector Simulation on GPU Community Meeting

AdePT Developers

May 3, 2022



Parallelization strategy

- GEANT4 tracks one particle at a time
  - Secondaries are pushed to a stack, tracked one after the other
  - For multi-threading: parallelization over events

Parallelization strategy

- GEANT4 tracks one particle at a time
  - Secondaries are pushed to a stack, tracked one after the other
  - For multi-threading: parallelization over events
- AdePT steps all active tracks at once
  - Much higher degree of parallelism & more uniform work for GPU

Parallelization strategy

- GEANT4 tracks one particle at a time
  - Secondaries are pushed to a stack, tracked one after the other
  - For multi-threading: parallelization over events
- AdePT steps all active tracks at once
  - Much higher degree of parallelism & more uniform work for GPU
- Different properties than simulation workflow of GEANT4
  - No "thread-local" state, everything associated with a track
  - At the same time: track must be as lightweight as possible
  - Data structures must not create bottlenecks (prefer atomics)

Track storage

- Properties stored per track: (see code for TestEm3)
  - Random number generator state
  - Kinetic energy
  - Position, direction, and current navigation state (volume)
  - State to be preserved across steps (number-of-interaction-left, MSC properties)

Track storage

- Properties stored per track: (see code for TestEm3)
  - Random number generator state
  - Kinetic energy
  - Position, direction, and current navigation state (volume)
  - State to be preserved across steps (number-of-interaction-left, MSC properties)

- Pre-allocate arrays of tracks per particle type (array of structures)
  - One for electrons, one for positrons, one for gammas
  - Advantage: can call spezialized kernels, potentially specialize stored properties
  - Atomic counter to hand out "slots" (recent addition to allow compaction)

Track storage

- Properties stored per track: (see code for TestEm3)
  - Random number generator state
  - Kinetic energy
  - Position, direction, and current navigation state (volume)
  - State to be preserved across steps (number-of-interaction-left, MSC properties)

- Pre-allocate arrays of tracks per particle type (array of structures)
  - One for electrons, one for positrons, one for gammas
  - Advantage: can call spezialized kernels, potentially specialize stored properties
  - Atomic counter to hand out "slots" (recent addition to allow compaction)

- Properties *not* stored per track:
  - Particle type / PDG number (instead implicit from array)
  - Charge, mass (can be inferred from particle type)

Random number generator: RANLUX++

- Based on the well-known RANLUX generator
  - Uses the equivalent LCG and therefore faster
  - Excellent statistical properties: inherited from RANLUX, only shared by MIXMAX (XORWOW used by default in cuRAND known to fail some statistical tests)

Random number generator: RANLUX++

- Based on the well-known RANLUX generator
  - Uses the equivalent LCG and therefore faster
  - Excellent statistical properties: inherited from RANLUX, only shared by MIXMAX (XORWOW used by default in cuRAND known to fail some statistical tests)
- Portable implementation available, written with GPUs in mind
  - See J. Hahnfeld, L. Moneta: A Portable Implementation of RANLUX++

Random number generator: RANLUX++

- Based on the well-known RANLUX generator
  - Uses the equivalent LCG and therefore faster
  - Excellent statistical properties: inherited from RANLUX, only shared by MIXMAX (XORWOW used by default in cuRAND known to fail some statistical tests)
- Portable implementation available, written with GPUs in mind
  - See J. Hahnfeld, L. Moneta: A Portable Implementation of RANLUX++
- Advantage over MIXMAX: smaller state
  - Even for `N = 17`, the default generator in GEANT4 (148 bytes of state)
  - Compared to 80 bytes for RANLUX++

Random number generator state per track

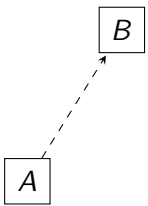
- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production

Random number generator state per track

- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production
- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track

Random number generator state per track

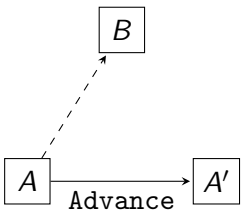
- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production
- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track



Input: state A
Output: states A' and B'

Random number generator state per track

- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production
- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track

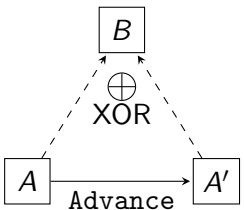


Input: state A
Output: states A' and B'

1. `Advance` to state A'

Random number generator state per track

- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production
- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track

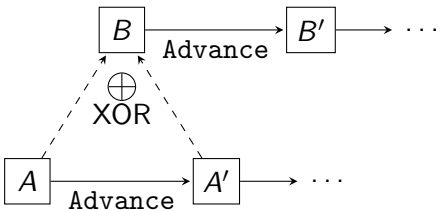


Input: state A
Output: states A' and B'

1. `Advance` to state A'
2. XOR bits in A and A' to get B

Random number generator state per track

- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production
- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track



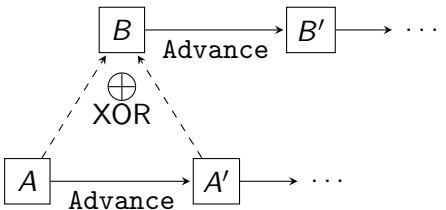
Input: state A
Output: states A' and B'

1. `Advance` to state A'
2. XOR bits in A and A' to get B
3. `Advance` state B to break correlations

Random number generator state per track

- For reproducibility, RNG state associated with each track
  - Identical results no matter the parallel execution order / kernel configuration
  - Essential for debugging, both during development and in production

- Need to initialize new RNG state for secondary particles
  - For the same reasons, must only depend on (parent) track
  - However, can re-use RNG state of dying track (annihilation, conversion)



Input: state A
Output: states A' and B'

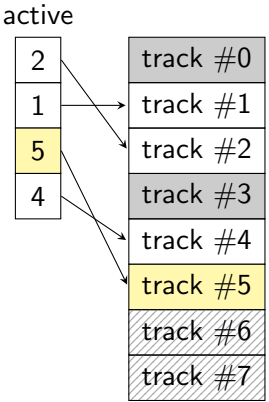
1. `Advance` to state A'
2. XOR bits in A and A' to get B
3. `Advance` state B to break correlations

Arrays of active and next tracks

track #0
track #1
track #2
track #3
track #4
track #5
track #6
track #7

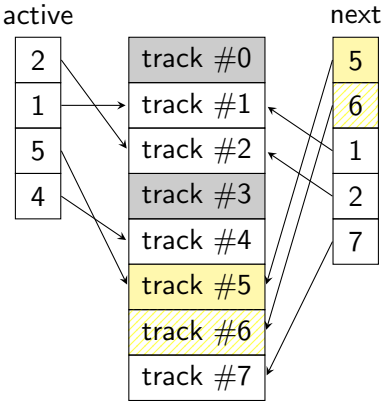
Arrays of active and next tracks

- Store indices of active tracks (per particle type)
  - For transportation, parallelize over these indices
  - Based on the index, locate the track and its state



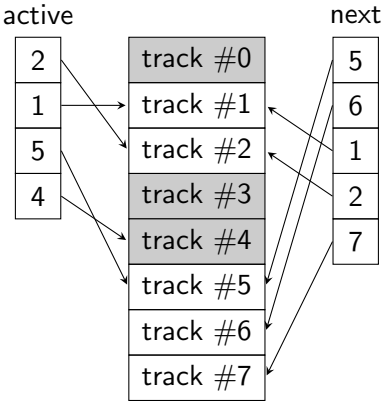
Arrays of active and next tracks

- Store indices of active tracks (per particle type)
  - For transportation, parallelize over these indices
  - Based on the index, locate the track and its state
- Queue indices for "next" active tracks
  - Both secondaries and "surviving" tracks
  - Implemented with atomic counter



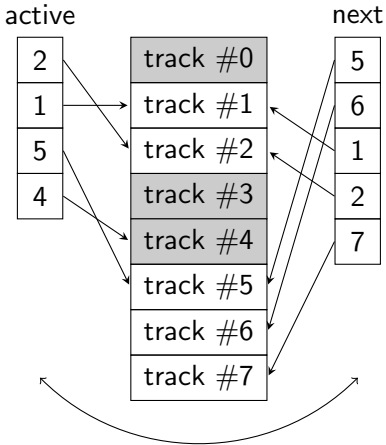
Arrays of active and next tracks

- Store indices of active tracks (per particle type)
  - For transportation, parallelize over these indices
  - Based on the index, locate the track and its state
- Queue indices for "next" active tracks
  - Both secondaries and "surviving" tracks
  - Implemented with atomic counter
  - Tracks are killed by not enqueuing



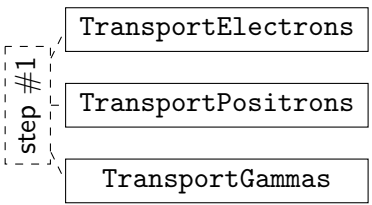
Arrays of active and next tracks

- Store indices of active tracks (per particle type)
  - For transportation, parallelize over these indices
  - Based on the index, locate the track and its state
- Queue indices for "next" active tracks
  - Both secondaries and "surviving" tracks
  - Implemented with atomic counter
  - Tracks are killed by not enqueuing
- After each step, swap the two arrays
  - And clear the array of "next next" active tracks



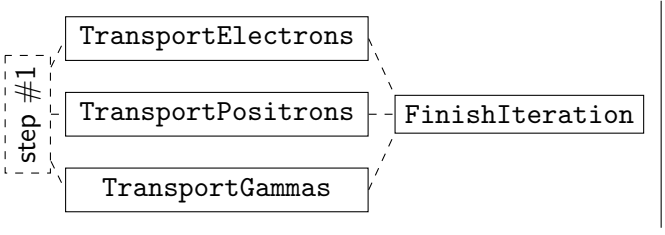
Overview of stepping workflow

- Transport of all particle types and active tracks is independent!
  - Can start kernels for particle types in parallel



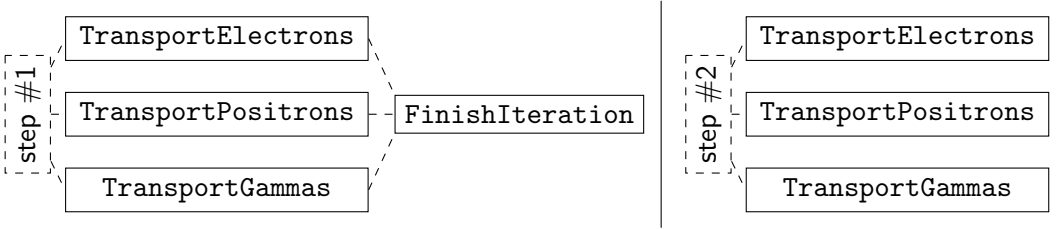
Overview of stepping workflow

- Transport of all particle types and active tracks is independent!
  - Can start kernels for particle types in parallel
- Synchronization means overhead:
  - Synchronize on the GPU via CUDA events
  - Synchronize with host once at the end of the step



Overview of stepping workflow

- Transport of all particle types and active tracks is independent!
  - Can start kernels for particle types in parallel
- Synchronization means overhead:
  - Synchronize on the GPU via CUDA events
  - Synchronize with host once at the end of the step



Conclusion

- AdePT steps all active tracks at once

Conclusion

- AdePT steps all active tracks at once
- Requires different data structures compared to GEANT4
  - In particular RNG state per track for reproducibility

Conclusion

- AdePT steps all active tracks at once
- Requires different data structures compared to GEANT4
  - In particular RNG state per track for reproducibility
- Active tracks managed via arrays of indices

Conclusion

- AdePT steps all active tracks at once
- Requires different data structures compared to GEANT4
  - In particular RNG state per track for reproducibility
- Active tracks managed via arrays of indices
- Transport kernels started in parallel, only synchronized once per step