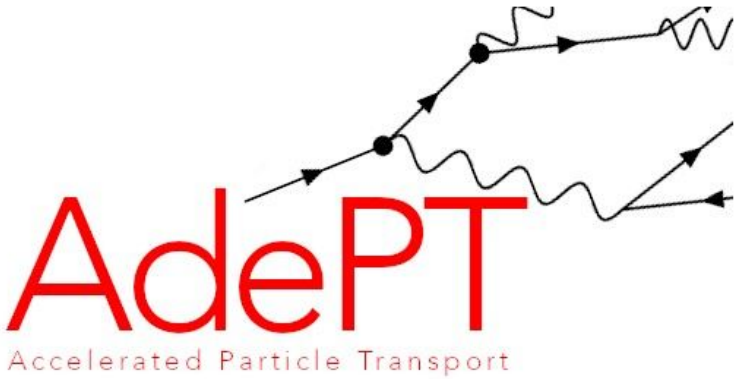


Accelerated demonstrator of electromagnetic Particle Transport

HSF WLCG Virtual Workshop 18-24
November 2020

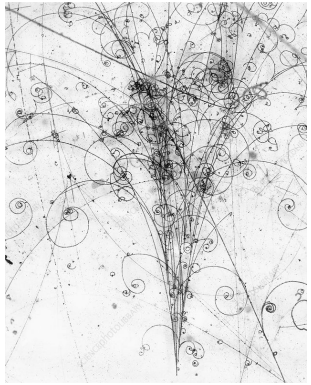
andrei.gheata@cern.ch

Motivation

- HEP detector simulation on accelerators: a hot topic
  - Can we efficiently use GPUs for more complex simulation applications?
- At least handing over part of Geant4 simulation to the GPU
  - EM calorimeter simulation seems a natural candidate
  - Can profit from massive parallelism at track level (e.g. EM showers) ?
  - Using simplified/specialized code + GPU-specific workflow
- Work needed/ongoing to enable the use of simulation components on GPU
  - Geometry demonstrator on GPU (VecGeom based ray-tracing)
  - Simplifying/adapting the physics framework
- Started a consolidated R&D effort on a prototype
  - Demonstrating a realistic simulation workflow on GPU
  - Try to assess the expectations for performance/investment for a large-scale project

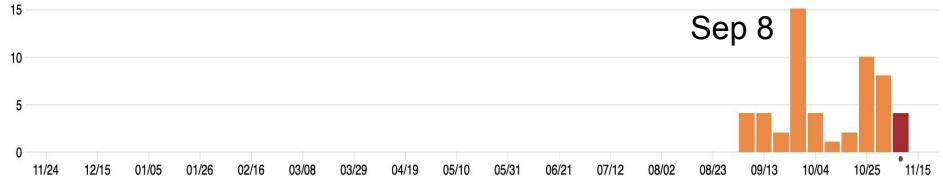
AdePT goals

- Demonstrate realistic EM shower simulation workflow on GPU
- Start with a basic “Fisher-Price” like workflow demonstrator
  - Single particle type carrying minimal state, two processes (energy loss and secondary generation), energy deposits per cell as output, dynamic track population
  - Allowing to develop a framework **controlling a dynamic track workflow**



- Evolve as $e^+/e^-/\gamma$ simulator in simple calorimeter setup
  - Magnetic field
  - VecGeom-based transport manager as first implementation
  - Gradually evolved physics processes allowing to simulate EM showers
  - Simple pre-configured “hits” as simulation result, transferred to host
- Maintain CPU compatibility for the entire simulation
  - Correctness validation, performance analysis, reference baseline

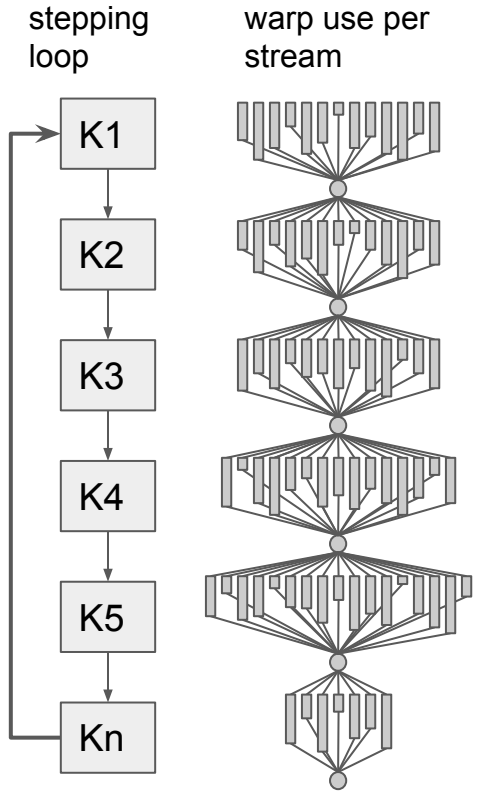
Bootstrapping AdePT



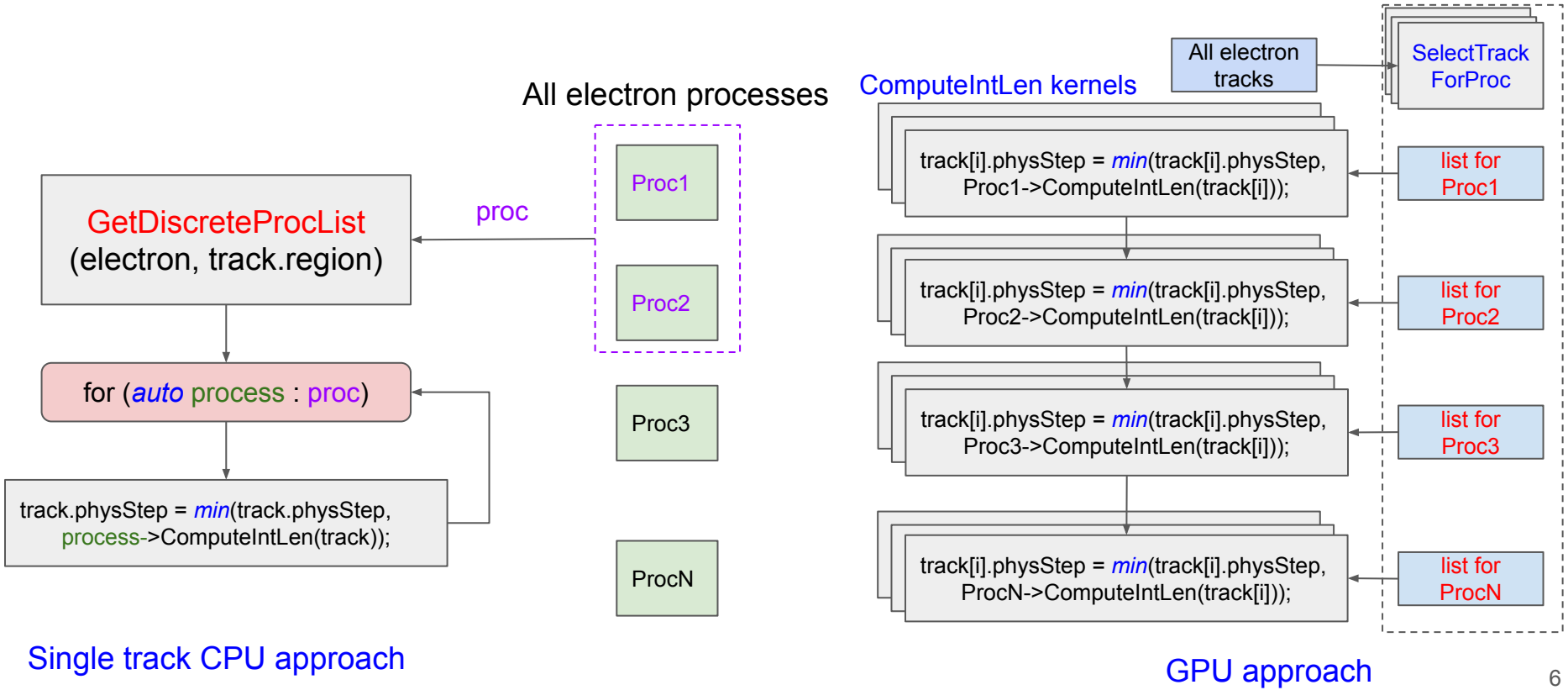
- GitHub repository + basic files
  - Readme, License, Contributing, … - September 2020
- Work so far mostly focused on creating basic infrastructure
  - Core types/macros/abstractions/kernel launchers reusable outside AdePT separated into an externalizable library for GPU (***CopCore***)
  - Concurrent containers/helpers for handling a dynamic population of tracks
  - Simple examples evolving with the rest of the infrastructure
  - Utilities and examples aiming to understand different portability strategies
- Rapid evolution in several directions
  - Contributors, collaborators, contacts with groups doing similar efforts
  - Work sites: adding standalone work items
  - Moving from workflow ideas to implementation

Rationale

- Refactor the stepping loop operations as sequence of kernels
- Move from: “list of actions to do for a track” to “list of tracks doing the same action”
- Partially mitigate kernel synchronization overheads and thread divergence by overlapping stepping loop execution in multiple streams
- Partially mitigate memory latency issues by coalescing accesses for reused state data
  - Using *SOA* in the track model, prefetching to shared memory, colocating particles prone to execute same models

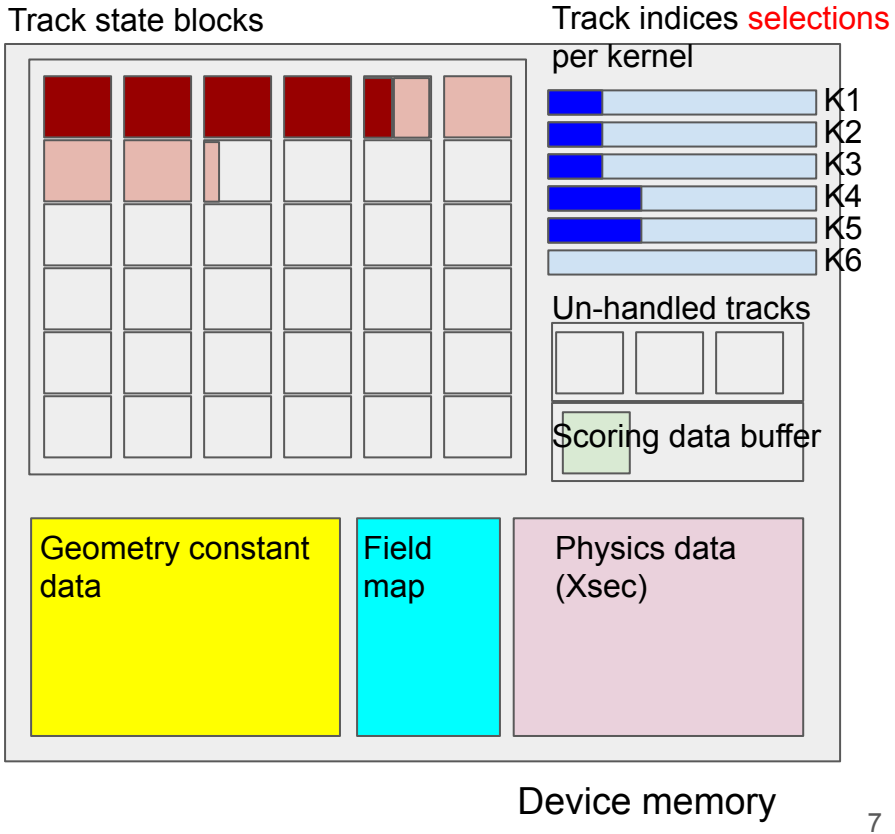


Example: sampling the step length

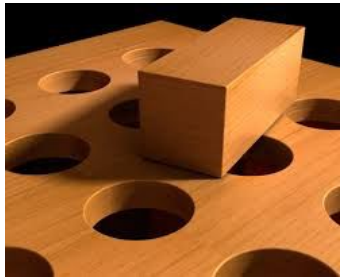


Workflow under investigation

- Pre-allocate data on device
  - constant data, state data (tracks), output data (hits)
- Fill tracks from host, start stepping loop
  - Manage dynamically selections for the “next” kernel
  - Manage dynamically holes produced by killed tracks
- Control flow using watermarks
  - New data blocks, track priorities to avoid memory bloat
- Fill pre-formatted output “hits”



Portability



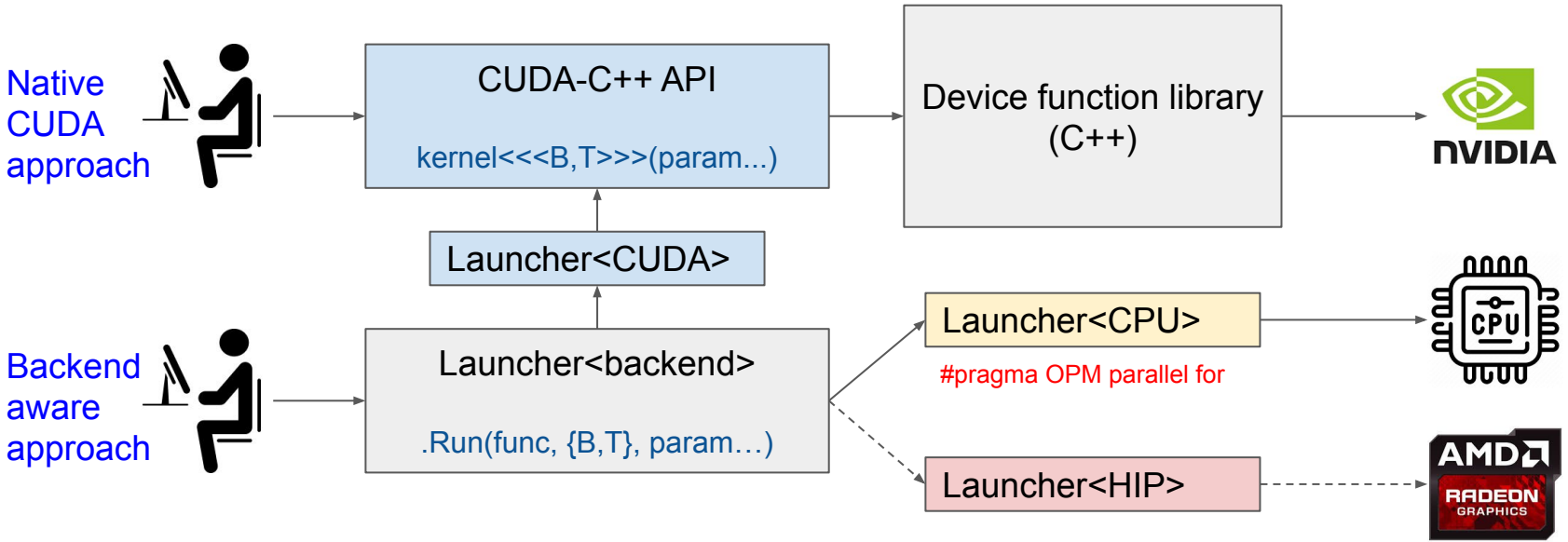
- Several directions being tried out
  - Portability libraries: Alpaka, OneAPI, …
  - Redefining CUDA keywords + Allen-like framework kernel launch
  - Macro-based decoration (à la VecGeom) + template specialization of the launch API
- Target: maintaining a single code base running on CPU/GPU
  - Common algorithmic part, possibly decorated with macros
  - Backend-specific launch infrastructure
  - Getting same/compatible results on CPU/GPU
- NOT trying to solve the general problem in AdePT!
  - Adopt/develop a thin layer requiring minimum dev effort & maintenance
  - Providing just the functionality that we need
  - Using a programming model friendly to CUDA-C++ features

Kernel launchers

- The idea is to “hide” the CUDA-C++ API behind a backend
  - Function/variable decoration can be hidden by using macros(VecGeom)/symbol redefinitions(Allen)
  - Kernel launches and CUDA types/allocations are a bit more tricky to hide
- Can we write a minimal C++ abstraction of this API ?
  - Specializing for other backends for the subset of features we use
  - Lightweight “re-invention of the wheel” for functionality provided by portability frameworks
  - Allows implementing the stepping loop as chained kernel launches, templated on the backend

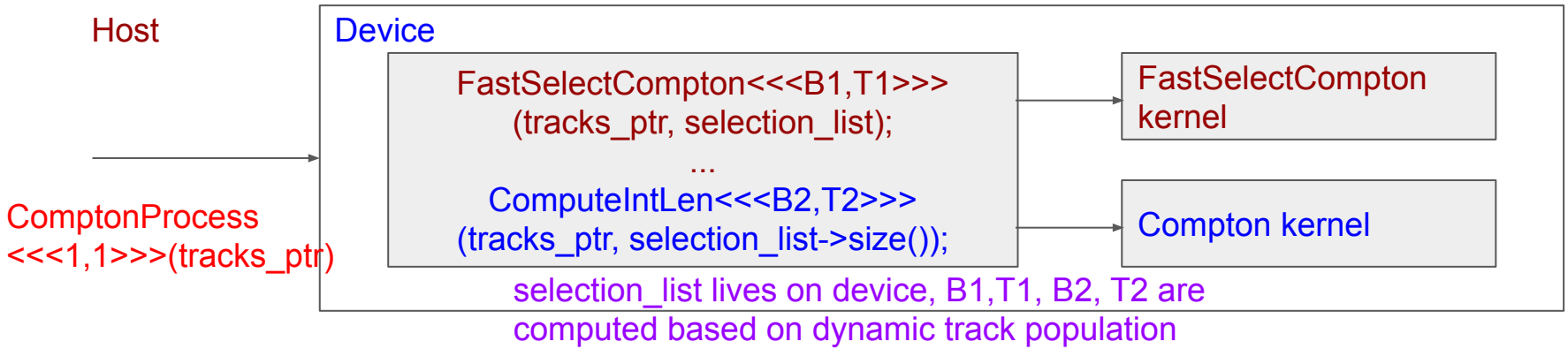
Kernel launchers

- Template specialized launchers
  - Abstracting part of the functionality provided by the CUDA-C++ launch API
  - Basically mapping GPU kernel parallelism to a OMP *parallel for* on CPU



Dynamic parallelism

- Due to the stochastic nature of simulation, kernel launches need to be dynamic
  - Adapting to the dynamic population of tracks per kernel
  - Moving data from device to host to compute kernel launch parameters is expensive
- Make launches from the device using dynamic parallelism:



Moving forward

- Splitting the work in more standalone tasks important for keeping up the momentum
  - Many such items already identified in several areas: geometry improvements, enabling magnetic field, data model, memory management, physics models
- Gradually adding features to our examples and implementing more complex workflows
  - Stepping management in geometry setup, constant field, actual physics processes using cross section data, ...
- Evolving the core part to provide device-friendly infrastructure for physics
  - Physics/math constants, materials & cuts, RNG support, efficient reductions for competing processes, …
  - Track model, data management (e.g. cross section tables & magnetic field)
- Adding tools for performance evaluation and benchmarking

Outlook

- The AdePT initiative followed an exploration period and discussions started already in 2019
  - Trying to put together a number of out-of-the-box ideas for particle transport “seen” from GPU perspective
- Exploration still important and good, we’re still in the “learning” phase
  - Several projects having similar goals, but no de-facto standard yet
  - Trying different approaches and techniques and learning from other projects
- Merging the scarce community resources in this R&D area will become essential for engaging a large-scale project
  - We cannot afford developing/maintaining several different GPU simulation applications
  - ***Excalibur*** developers are also co-developing in AdePT
  - We initiated discussions with ***Celeritas*** to touch base on commonalities and concurring approaches

Backup

Kernel launchers, the user side

main.cpp

```
void RunOnGPU();   // forward declare CUDA entry function
                   // (compiler-dependent macros in headers)
int main() {
  RunOnCPU();   // dispatch RunSim<CPU>
  RunOnGPU();   // dispatch RunSim<CUDA>
}
```

Device function lib

```
// thread id passed to user function
VECCORE_ATT_HOST_DEVICE
void myFunc(int id, param...) {...}

COPCORE_CALLABLE_FUNC(myFunc)
```

RunSim.hpp

```
template <copcore::BackendType backend>
void RunSim() {
  // We need a local variable to hold a copy of the device function address
  COPCORE_CALLABLE_DECLARE(myFuncPtr, myFunc);
  // Allocate the data
  copcore::Allocator<MyType, backend> myAlloc; // optional device id in ctor
  MyType *data = myAlloc.allocate(1024, p…); // allocates 1024 elements passing p… to the ctor

  // Create a stream (do nothing on CPU)
  copcore::StreamType<backend> stream;
  StreamType<backend>::CreateStream(stream);
 … continued ...
```

Kernel launchers, the user side

RunSim.hpp

```
template <copcore::BackendType backend>
void RunSim() {
   … continued …
  // Create a launcher in the selected stream for this backend
  copcore::Launcher<backend> myLauncher(stream);

// Launch on a user-defined grid or internally optimized one, calling the user function in a grid size loop
  myLauncher.Run(myFuncPtr,   // function to be launched
                 1024,        // number of elements (#calls to the user function)
                 {32, 32},    // grid launch parameters to be used. {0,0} triggers internally-optimized grid
                 params…);    // parameters to be passed to the user function
  …
  // De-allocate the data
  myAlloc.deallocate(data, 1024);  // will call also the destructor for the allocated elements if 1024 not omitted
}
```

Launching functors

RunSim.hpp

```
template <copcore::BackendType backend>
void RunSim() {
   … continued …
  // Create a launcher in the selected stream for this backend
  copcore::Launcher<backend> myLauncher(stream);

// Launch on a user-defined grid or internally optimized one
  myLauncher.Run([] VECCORE_ATT_DEVICE (int thread_id, params…) { … }, // --extended-lambda needed
                 1024,          // number of elements (calls to the user function)
                 {32, 32},      // grid launch parameters to be used. {0,0} triggers internally-optimized grid
                 params…);      // parameters to be passed to the user function
  …
  // De-allocate the data
  myAlloc.deallocate(data, 1024);  // will call also the destructor for the allocated elements if 1024 not omitted
}
```

… could be useful for simple tasks (e.g. reductions, selections, …)