



ACAT: 29 November - 3 December, 2021

A vendor-agnostic, single-code based GPU tracking for the Inner Tracking System of the ALICE experiment

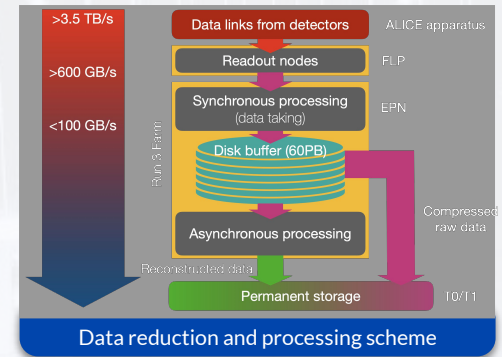
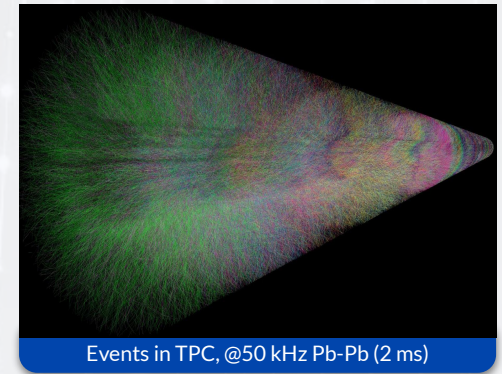
Matteo Concas, INFN Torino
on behalf of the ALICE collaboration

mconcas@cern.ch



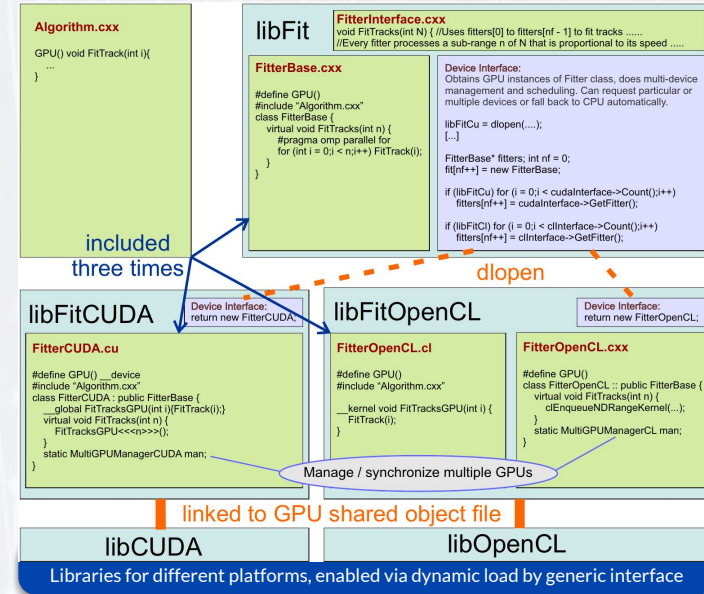
ALICE data-taking in Run 3

- Run 3: LHC will deliver **50 kHz in Pb-Pb** collisions
 - ALICE aims to record **>10 nb⁻¹** integrated luminosity
 - **x50 times** more minimum bias data wrt Run 2
- Triggerless approach: **continuous readout**
 - Data stream split in 10 ms *timeframes*
- **Online reconstruction** and calibration for data reduction
 - **Large computing power required** to operate online
 - Alice O² (Online-Offline) framework: software stack for Run 3
 - **Synchronous**: TPC reconstruction and calibration of all detectors
 - **Asynchronous**: during p-p collisions or technical stops: all data are reconstructed (EPN, Grid)
- Processing on dedicated farm at **experimental site**
 - **250x Event Processing Nodes (EPNs)** with x1 CPU with **64** cores
 - **8x AMD Graphic Processing Units (GPUs)**
 - **~1500 GPUs** required to process 50 kHz Pb-Pb collisions



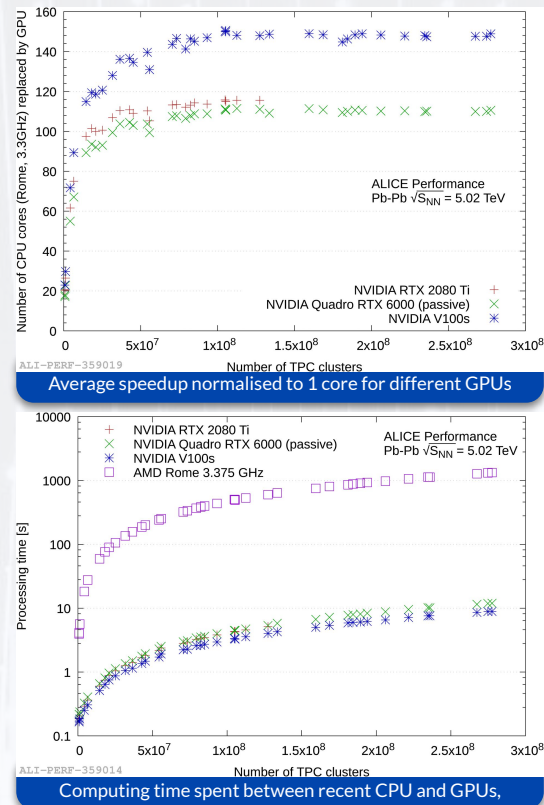
Using GPUs in ALICE in Run 2

- TPC reconstruction ran on the **NVIDIA** GPUs on the High-Level Trigger (HLT) farm for data compression
- Included support for **multiple architectures** and paradigms:
 - Parallelism on CPU with **OpenMP**
 - **OpenCL** 1.2&2 implementation for non-NVIDIA accelerators
- A **single codebase** to support all the options
 - Advanced, high-level, **all-encompassing** implementation
 - **>90% of shared code** vs <10% of specific architecture macros!
 - Fix and developments of core routines **processor-independent**
 - **Architecture autodetection** → **shared libraries** produced and dynamically loaded at runtime



Using GPU in Run 3 with ALICE O²

- Operate part of reconstruction on **GPU is mandatory***
 - TPC is the main actor and uses whole cards in synchronous reco
 - **Efficient usage** of all computing resources on EPNs **is desired**
 - **Other detectors** developing solutions for the asynchronous pass



*GPU reconstruction of TPC on GPU results in the lowest cost/performance ratio for the farm

Using GPU in Run 3 with ALICE O²

- Operate part of reconstruction on **GPU is mandatory***
 - TPC is the main actor and uses whole cards in synchronous reco
 - **Efficient usage** of all computing resources on EPNs is desired
 - **Other detectors** developing solutions for the asynchronous pass
- GPU codebase evolved into a **common framework**
 - Steer and centralise reconstruction on graphics devices
 - Advanced and **centralised device memory management**
 - **Useful features exposed** for external components (e.g. automatic dynamic load of required libraries)
 - Different strategies to **minimize codebase** (metaprogramming, wrappers, syntactic sugar)

```
#if defined(__HOST_CODE__)
#define GPUD() // Omit keyword on host code
...
#elif defined(__OPENCL__)
#define GPUD() // Omit keyword for OpenCL
...
#elif defined(__CUDAACC__) || defined(__HIPCC__)
#define GPUD() __device__ // Define keyword for CUDA and HIP
...
#endif

namespace o2::gpu::std {
#ifdef __GPU_CODE__
template <typename T, size_t N>
struct array {
    GPUD() T& operator[](size_t i) { return m_internal_V__[i]; }
    ...
    T m_internal_V__[N];
};
#else
template <typename T, size_t N>
using array = std::array<T, N>;
#endif
}

GPUD() void my_function(...) { // works on host and device code
    o2::gpu::std::array<float, 3> arr;
    ...
}
```

Possible code to showcase uniform definitions with single macros

*GPU reconstruction of TPC on GPU results in the lowest cost/performance ratio for the farm

Using GPU in Run 3 with ALICE O²

- Operate part of reconstruction on **GPU is mandatory***
 - TPC is the main actor and uses whole cards in synchronous reco
 - **Efficient usage** of all computing resources on EPNs **is desired**
 - **Other detectors** developing solutions for the asynchronous pass
- GPU codebase evolved into a **common framework**
 - Steer and centralise reconstruction on graphics devices
 - Advanced and **centralised device memory management**
 - **Useful features exposed** for external components (e.g. automatic dynamic load of required libraries)
 - Different strategies to **minimize codebase** (metaprogramming, wrappers, syntactic sugar)
 - **Kernel abstractions** to support CPU (serial and parallel) and GPU with single class

```
class MyKernel : public GPUKernelTemplate // base kernel
  abstraction class
{
public:
  GPUd() static void Thread(int nBlocks,
                            int nThreads,
                            int iBlock,
                            int iThread,
                            ...);

  ...
};

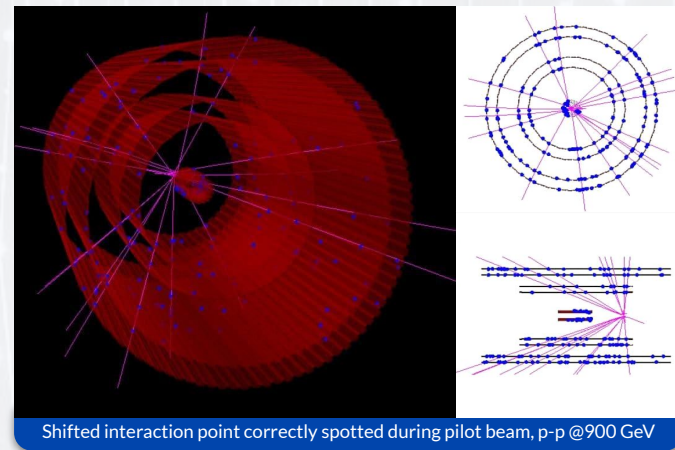
template <>
GPUd() inline void MyKernel::Thread<0>(int nBlocks,
                                       int nThreads,
                                       int iBlock,
                                       int iThread, ...) {
  ... // macros to correctly treat different platforms
}
```

Simplified pseudocode for kernel abstraction: base class provides ad-hoc utilities. Thread() function contains the routine to be performed

*GPU reconstruction of TPC on GPU results in the lowest cost/performance ratio for the farm

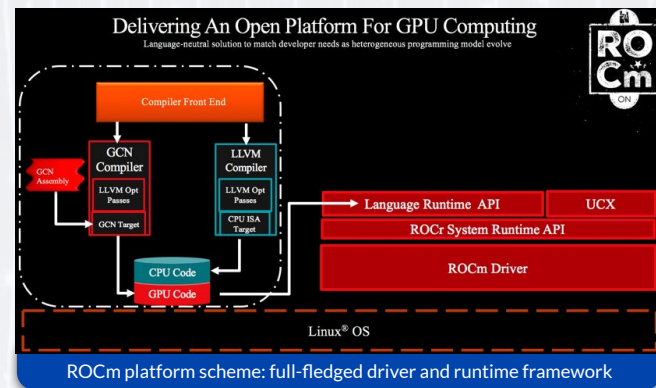
ITS reconstruction

- Run 3: ITS is the innermost **7 silicon layers** detector
 - Charged particles leave **clusters of fired pixels** on layers
- Reconstruction algorithms for **vertexing** and **tracking**
 - CPU code is in production and tested, **GPU to scale the solution** parallelising at multiple levels
 - Complexity is dominated by **combinatorial** matches between
 - Mostly an **embarrassingly parallel** problem to address, very few spots where more sophisticated solutions are required
- Running on **GPUs during asynchronous phase**
 - faster execution and a efficient usage of GPU resources
 - CPUs to remain available for rest of the reconstruction
 - Possible to exploit GPU resources on Grid pledges



GPU usage in ITS reconstruction

- GPU code is a **new development** for Run 3
 - Design choices based on present **state of the art**
- Choose tools to suit two targets: **NVIDIA** and **AMD** devices
 - Compute Unified Device Architecture (CUDA)
 - Heterogeneous-Compute Interface for Portability (**HIP**) using **ROCm**
- Coding prototypes and proofs of concept for **both platforms**?
 - New developers and students are familiar with mainstream tools
 - Adopt ALICE-specific approach can show a steep learning curve
- **Integration** with existing GPU framework
 - **Both vendors platforms are supported** and mainstream in O^2
 - Ok to plug-in external **independent GPU code** (libs) to the system
 - Integration with native framework can happen in a second time



Heterogeneous-Compute Interface for Portability (HIP)

- **C++ Runtime API** and Kernel Language for portable AMD and NVIDIA applications from single source code
 - It **resembles** very closely CUDA APIs, **by design**
 - Computing capabilities unique for each platform not supported
- ROCm has tools to automatically translate CUDA to HIP
 - **hipify-clang**: based on Clang, **actual code translation**
 - **hipify-perl**: script for **line-by-line code conversion**
- Notorious CUDA libraries like **Thrust** and **CUB**, find their HIP counterparts in the ROCm stack
- Goal maintaining both solutions: **eliminate redundant code**
 - **Automatically generate it at some point!**

```
// CUDA code
cudaMalloc(&A_d, Nbytes);
cudaMalloc(&C_d, Nbytes);
cudaMemcpy(A_d, A_h, Nbytes, cudaMemcpyHostToDevice);

vector_square <<<512, 256>>> (C_d, A_d, N);
cudaMemcpy(C_h, C_d, Nbytes, cudaMemcpyDeviceToHost);

// HIP code, translated
hipMalloc(&A_d, Nbytes);
hipMalloc(&C_d, Nbytes);
hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);

hipLaunchKernelGGL(vector_square, 512, 256, 0, 0, C_d, A_d, N);
hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```

Minimal code to compute the square of an array of loaded data on a GPU.
CUDA and HIP API calls have a 1:1 correspondence

HIP code on-the-fly generation

- The **O² compilation process** is steered by **CMake**, providing
 - **Platform autodetection** and production of corresponding target libraries
 - Customisable configuration and compilation stages that can execute **custom commands** (i.e. the translator) and **establish dependencies between targets**, to be re-generated upon source file changes
- To generate HIP code in place is a **straightforward process**
 - Build target libraries and executables parsing CUDA files and generating HIP ones
 - Currently based on **hipify-perl**: is run on all **.cu** files to produce HIP code
- Headers files used for both compilations are the same
 - Very little fraction of dedicated code to cope with some small architectural differences
 - Overall **negligible boilerplate** (<0.1% of LoCs in two instances where this approach is present)
- Currently **two main components** in O² successfully rely on this approach
 - A **GPU benchmarking** tool developed *ad-hoc* to test graphics card I/O performance
 - **ITS primary vertex finder**: is fully operational on NVIDIA and AMD cards

Conclusions and outlook

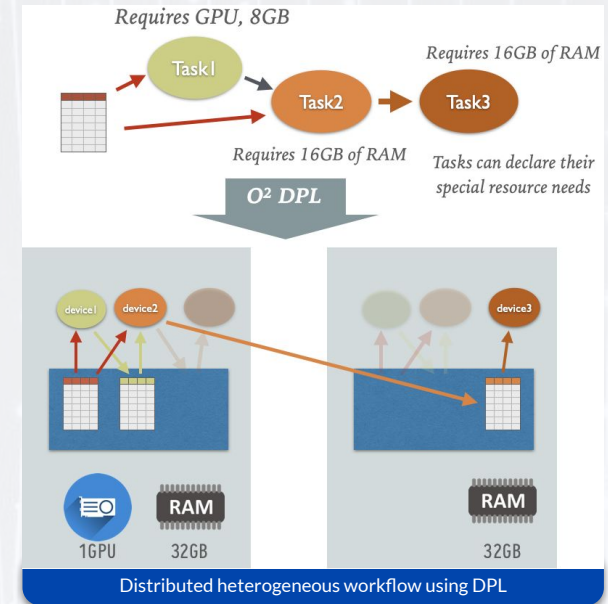
- During Run 3 data taking the **ALICE** experiment is using **GPUs to enable online reconstruction**
 - Main use case, the TPC reconstruction has been **successfully tested during the pilot beam** in October
 - **Asynchronous reconstruction**: plan is to **extend** the usage of graphics cards **to other use cases**
- **ITS aims** at having a working **GPU version of the tracking algorithms**
 - CUDA and HIP platforms **served by a single code base**, duplicated at build time
 - **Primary vertexer is already functional**, with visible speedup: up to x12 times in a not optimized implementation on CUDA and x5 on AMD.
 - GPU tracking prototype in place, to be updated to latest developments
- The **automatic translation** of the CUDA code allows us to maintain and develop a single code
 - **Little additional code** other than what already present in O² for GPU
 - Does **not require external** dedicated compatibility and portability **libraries**

Backup



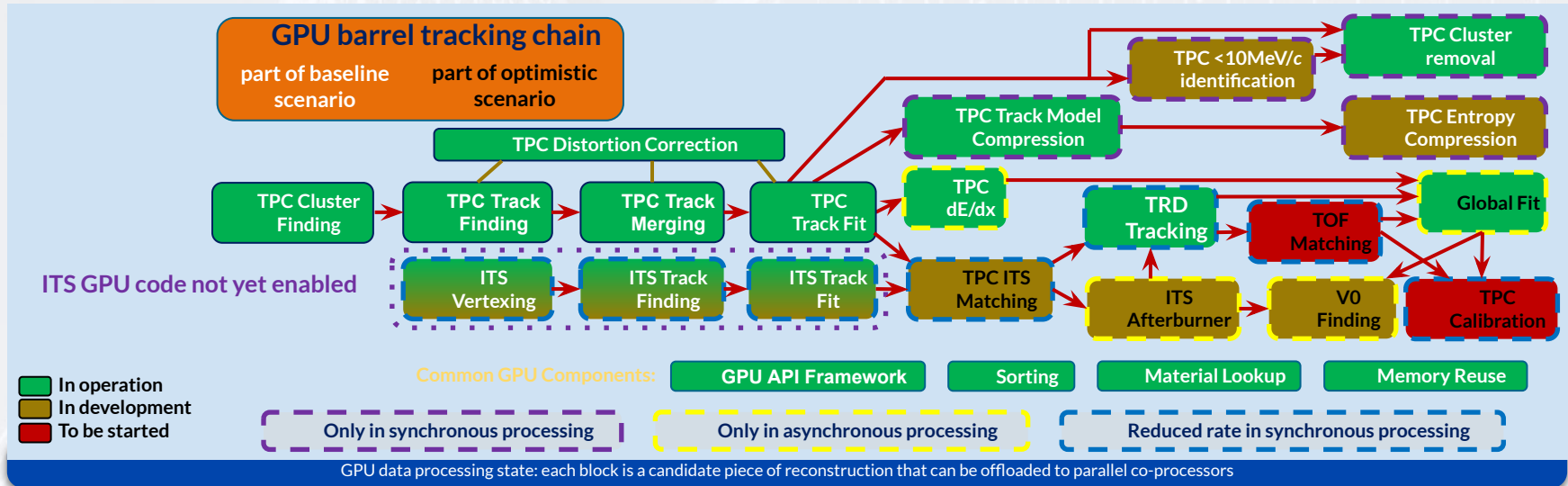
GPU integration in O²

- O²: Single software framework for **online and offline computing**
 - Implicit workflow description translated by data processing layer (DPL) to a topology of processes
 - Communication across tasks is done via message passing: works locally and distributed across computing nodes
- DPL devices **support GPU executions**
 - Multiple processes can share the same GPU and run in parallel
 - Multiple tasks running on different TFs simultaneously: mask the asynchronous behaviour of GPUs
 - GPU reconstruction framework is included a device to steer the reconstruction on graphics cards



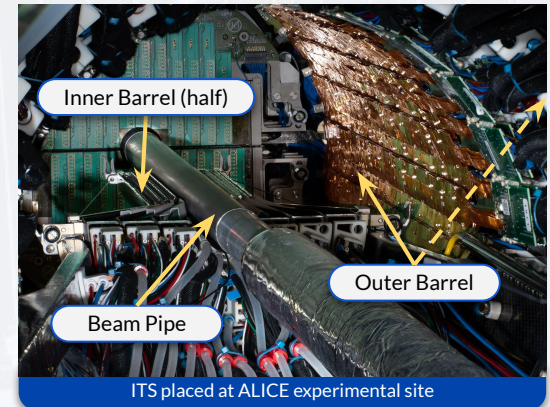
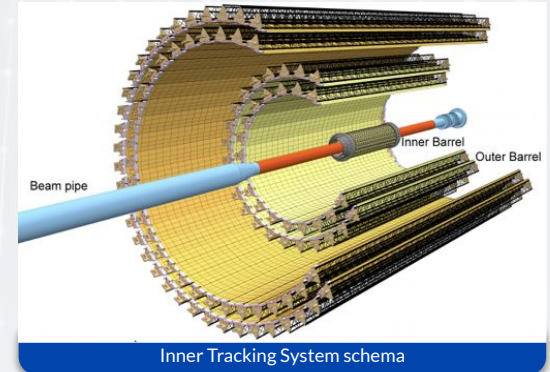
ALICE GPU processing: state of the art

- Two scenarios:
 - Baseline:** ready and commissioned, fully operative and tested, also during pilot beam
 - Optimal:** diverse state, some pieces are ready, will be ideal to efficiently use available resources



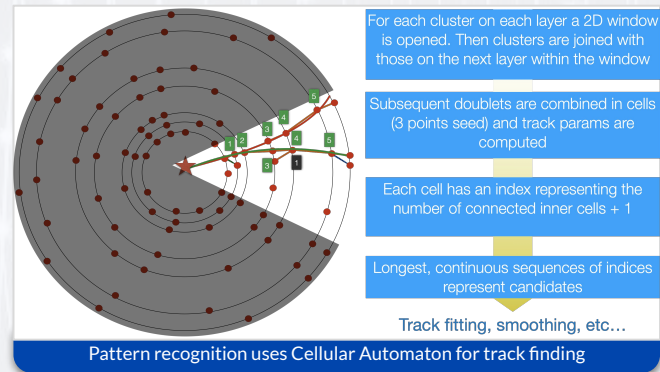
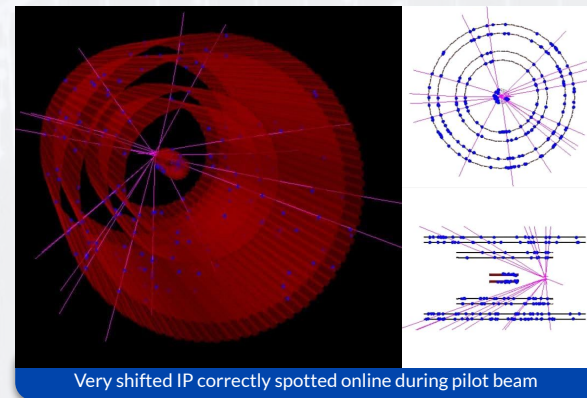
Upgraded ITS for Run 3

- Cylindrical detector with 7 layers of silicon **pixel sensors** (MAPS) sensible to charged particles
 - Their passage leaves **clusters** of fired pixels
- **Low material** budget ($X_0 < 1\%$)
 - Improved spatial and momentum resolution
- Operates at **continuous readout** regime
 - Up to 1MHz for pp collisions and 50 kHz in PbPb
- New detector \Rightarrow new algorithms for vertexing and tracking



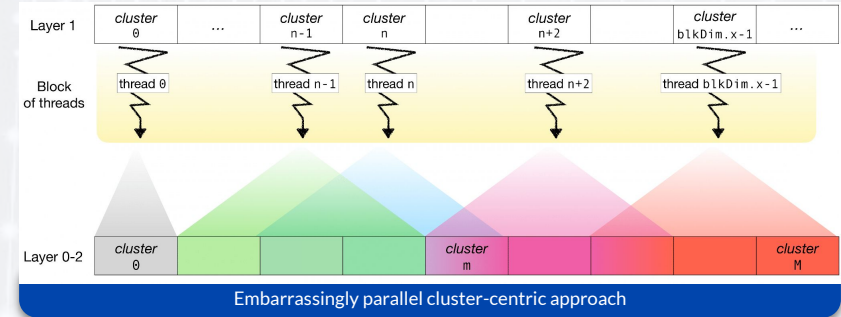
ITS tracking algorithms

- **Primary vertex finder** (seed for tracking)
 - Clusters from Inner Barrel-only (3 layers)
 - Brute-force combinatorial with some educated guesses
 - Linear extrapolation of tracks and refit of their origin to estimate the interaction point (IP)
 - Detects pile-up of collisions in the same time unit
- **Tracker finder and fitter**
 - Run multiple iterations with specific selections
 - Combinatorial and pattern recognition based on Cellular Automaton (CA)
 - Kalman filter track fitting of candidates
 - Posterior selection on quality of the fits and promote tracks to final



GPU parallelization strategies

- Both cases show native parallel structure
 - Pure combinatorial and track fitting are *embarrassingly parallel* problems
 - Some spots require some more refined parallelisation approaches (map-reduce, sorting algorithms, histograms), no need for reinventing the wheel
 - High throughput is achievable by processing whole timeframe at the same time



Online reconstruction and the EPN farm

- **Online reconstruction on the EPN farm**
 - Data acquisition splitted in time frames (TFs)
 - TFs then processed on the EPN farm, on scalable pipelines with configurable topologies
- **Synchronous phase:** during data taking:
 - Time Projection Chamber (TPC) data reconstruction and compression
 - 100% TPC standalone tracking runs on GPUs
 - Inner tracking system (ITS) and Transition Radiation Detector (TRD) tracking on fraction of events → needed for calibration
- **Asynchronous phase:** during no-beam or pp runs:
 - Full reconstruction including all detectors
 - Reconstruction on GPU for ITS and TRD

