Software strategy for GPU hardware abstraction in ALICE

David Rohr

drohr@cern.ch

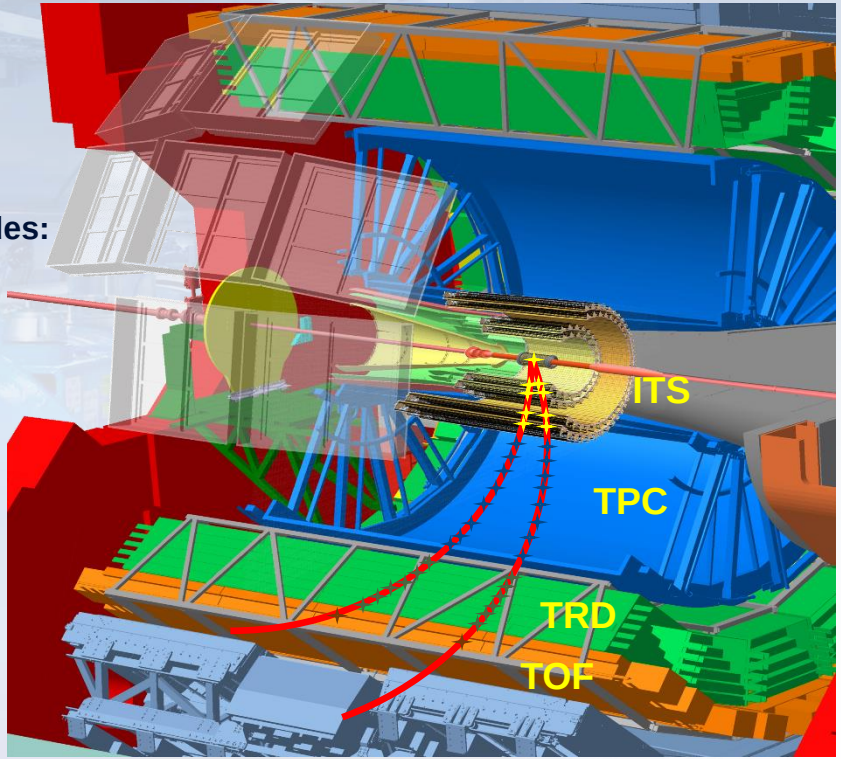
Compute Accelerator Forum

11.11.2020

The ALICE detector in Run 3

- **ALICE uses mainly 3 detectors for barrel tracking: ITS, TPC, TRD + (TOF)**
  - **7 layers ITS** (Inner Tracking System – silicon tracker)
  - **152 pad rows** TPC (Time Projection Chamber)
  - **6 layers TRD** (Transition Radiation Detector)
  - **1 layer TOF** (Time Of Flight Detector)

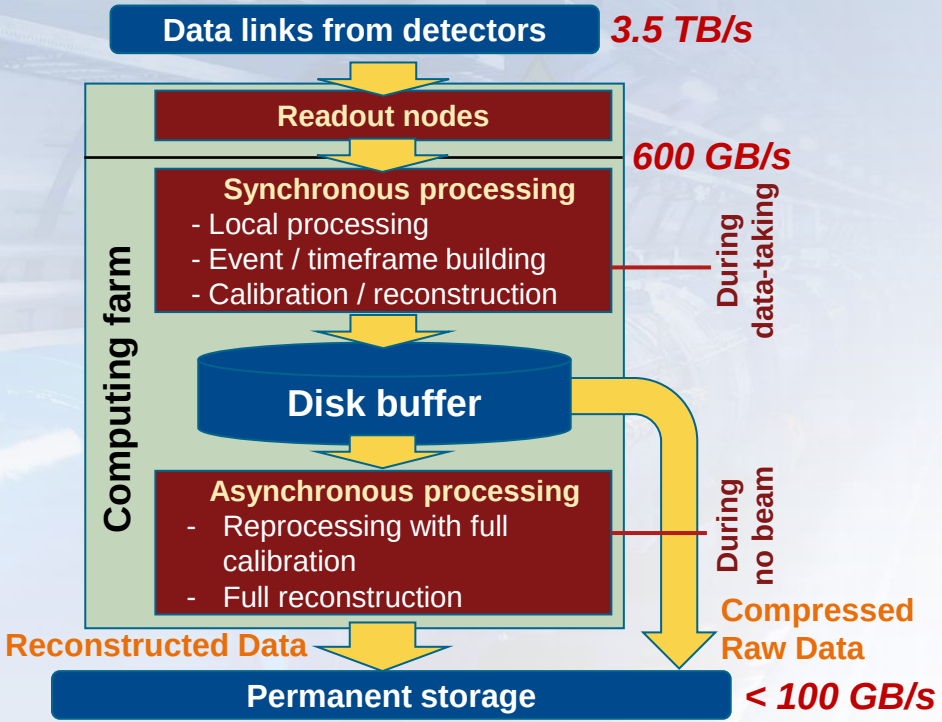
- **In the shutdown before Run 3, there are several major upgrades:**
  - The TPC is equipped with a GEM readout.
  - The ITS is completely replaced by 7 layers of silicon pixels.
  - **Major computing upgrade in the O² project.**



Run 3 data processing in a nutshell

- **Synchronous processing** while there is beam in the LHC and raw data is recorded.
- All data is compressed and stored on a **disk buffer**.
- When there is no beam in the LHC (or the computing farm is not fully loaded during pp data taking), the computing resources are used for **asynchronous reprocessing** of the data.
- Details on next slides.

Time Frames assembled in synchronous processing:
- **~10 ms of data**
- **Contains O(500) collisions**



GPU compatible workloads

- ALICE runs reconstruction in 2 phases during Run 3: synchronous (online) and asynchronous (~offline).

- Synchronous reconstruction dominated by TPC tracking (>90% of compute load).
  - Fully running on GPUs and defines capacity of online computing farm → 2000 GPUs.
  - Other steps might run on GPU (e.g. ITS tracking of few % of events), but GPUs mostly needed for TPC.

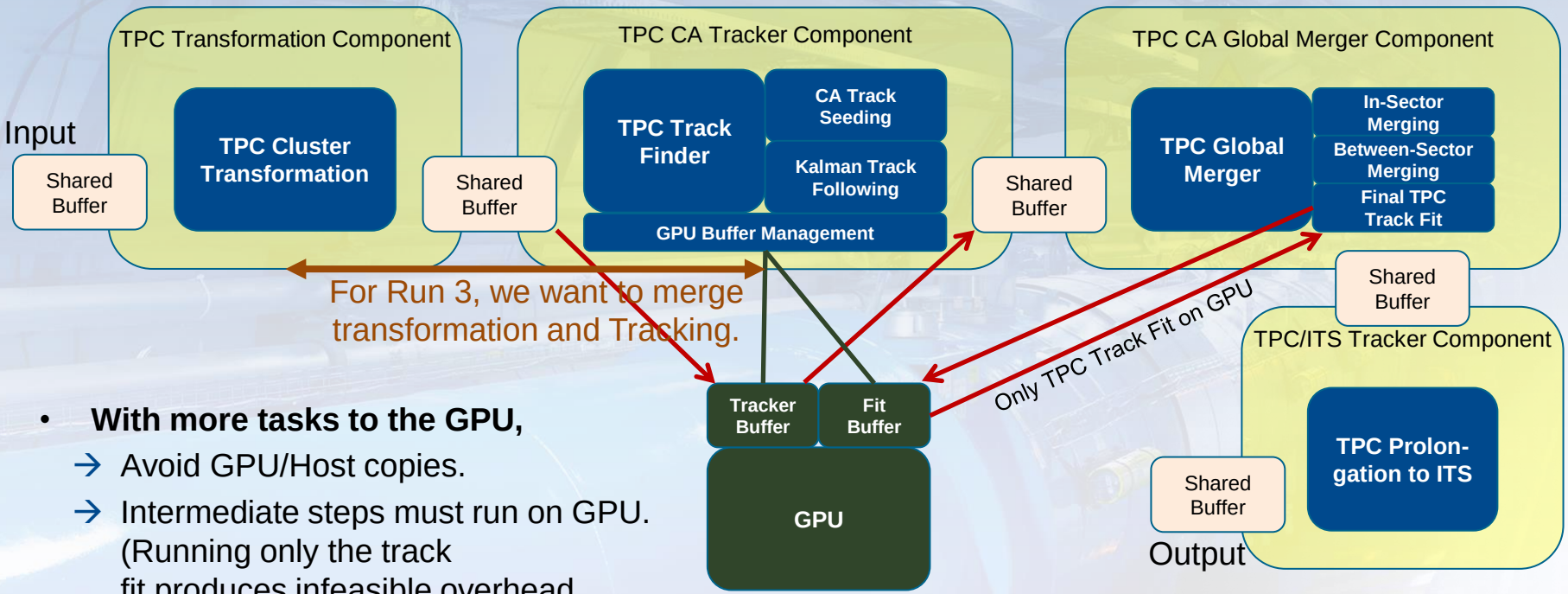
- GPUs are available during asynchronous reconstruction.
  - Not dominated by TPC tracking → Asynchronous reconstruction at O2 farm has GPU capacity available.
  - GPUs in the GRID not so clear, but makes sense to leverage whatever might become available.
  - ITS tracking large part of asynchronous reconstruction, and most steps supports GPUs already.

Baseline solution (almost available today): Most of sync. reco on GPU

Optimistic solution (what could we do in the ideal case): Run most of tracking + X on GPU.

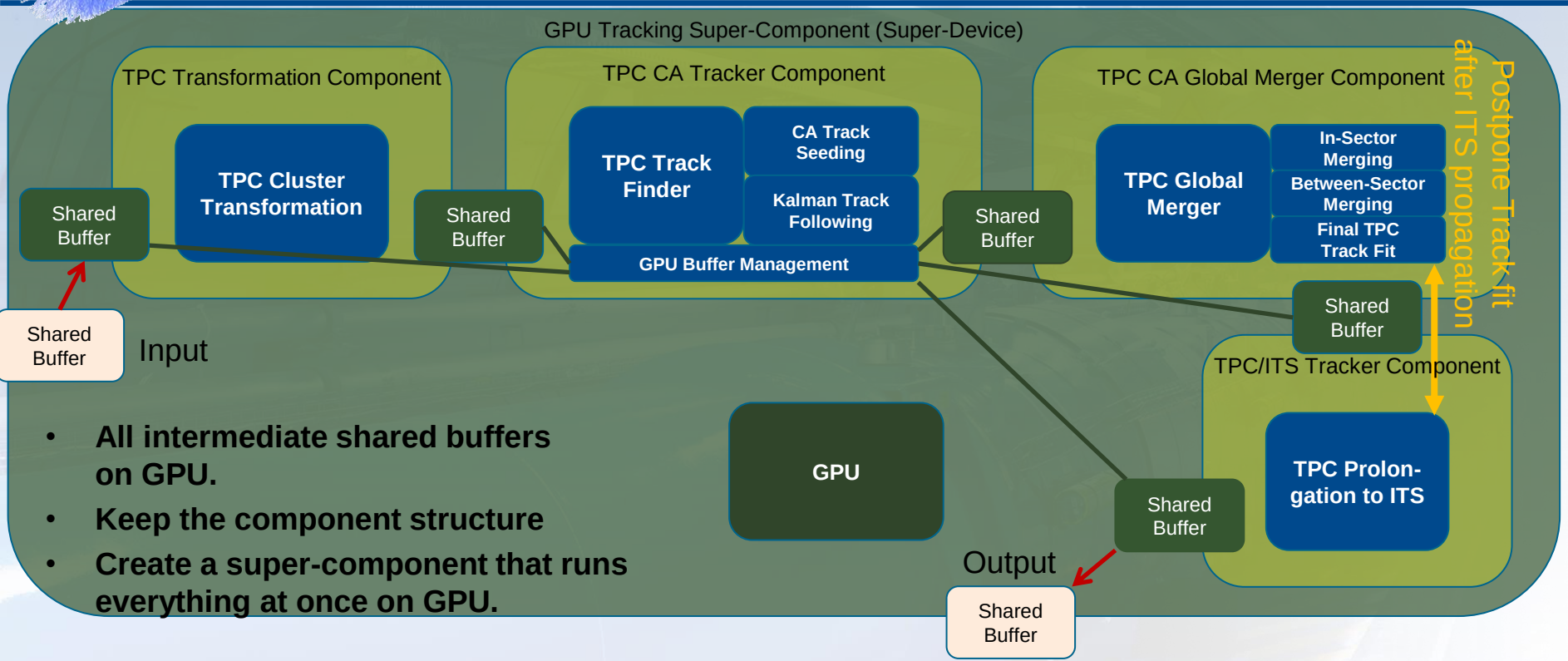
- **Candidate for extended GPU usage: Full Barrel Tracking + related.**
- **As long as GPU capacity is available, every work offloaded from the CPU frees CPU resources.**

Approach of Run 2 HLT TPC / ITS Tracking Components



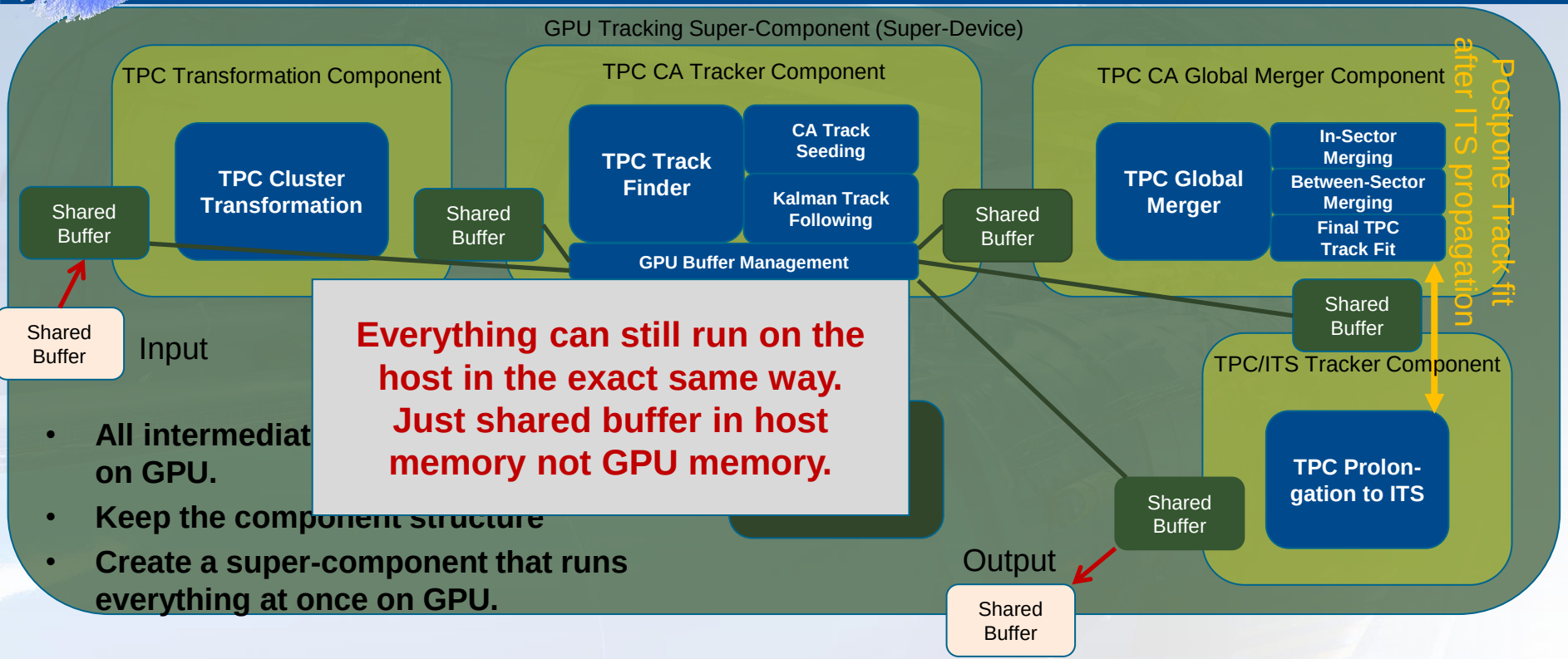
- **With more tasks to the GPU,**
  - → Avoid GPU/Host copies.
  - → Intermediate steps must run on GPU. (Running only the track fit produces infeasible overhead.

Approach for Run 3



- **All intermediate shared buffers on GPU.**
- **Keep the component structure**
- **Create a super-component that runs everything at once on GPU.**

Approach for Run 3

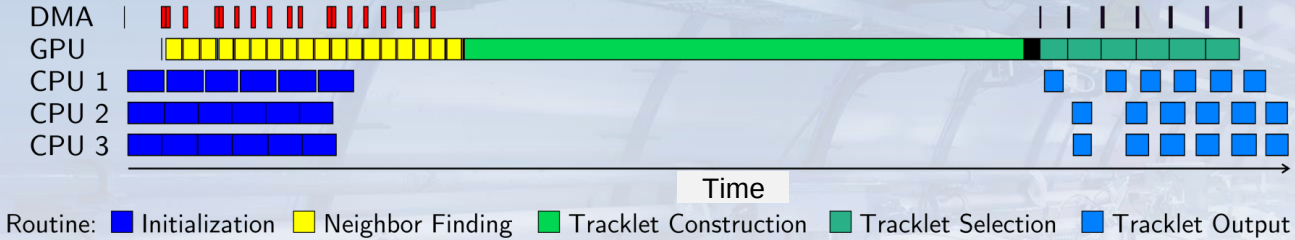


Everything can still run on the host in the exact same way. Just shared buffer in host memory not GPU memory.

- **All intermediat… on GPU.**
- **Keep the component structure**
- **Create a super-component that runs everything at once on GPU.**

Pipelined processing (from Run 2)

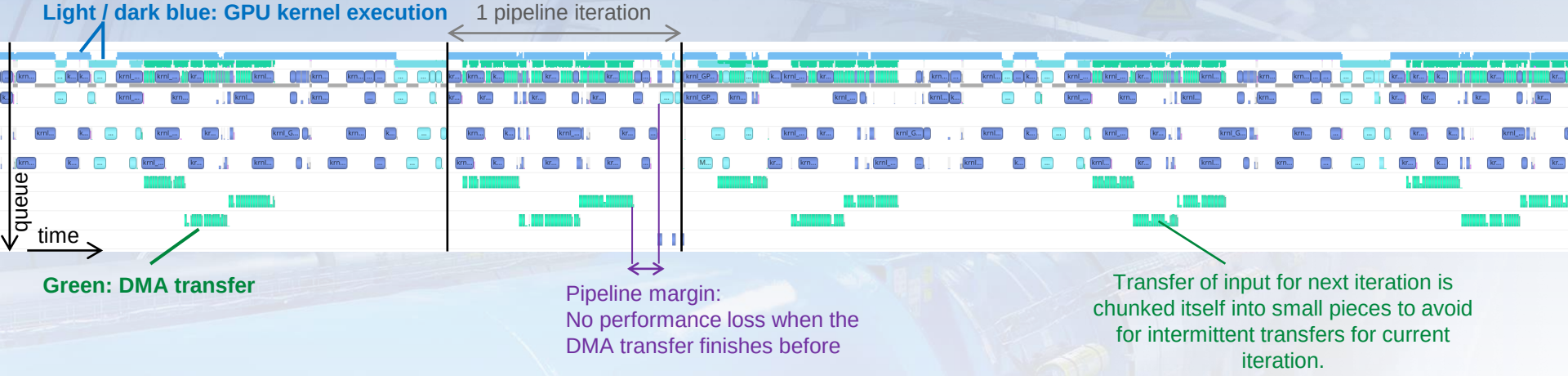
- **Handling of asynchronous computation / data transfers**
  - **1st iteration (Run 1 HLT)**: Split event in chunks, to pipeline CPU processing, GPU processing, and PCIe transfer.



  - **2nd iteration (Run 2 HLT)**: Processing of two events in parallel on the GPU concurrently in addition.
    - ~20% faster than first version – GPUs have become wider and this exploits the parallelism better.
    - Not possible during Run 1 due to GPU limitations at that time.
    - We still kept the pipeline-scheme within each event, to maximize performance.
  - **3rd iteration (Run 3)**: Go back to the old scheme from Run 1 – with time frames instead of events.
    - Time frames are large → avoid keeping multiple in memory.
    - Enough parallelism inside one time frame.

Pipelined GPU processing (Run 3)

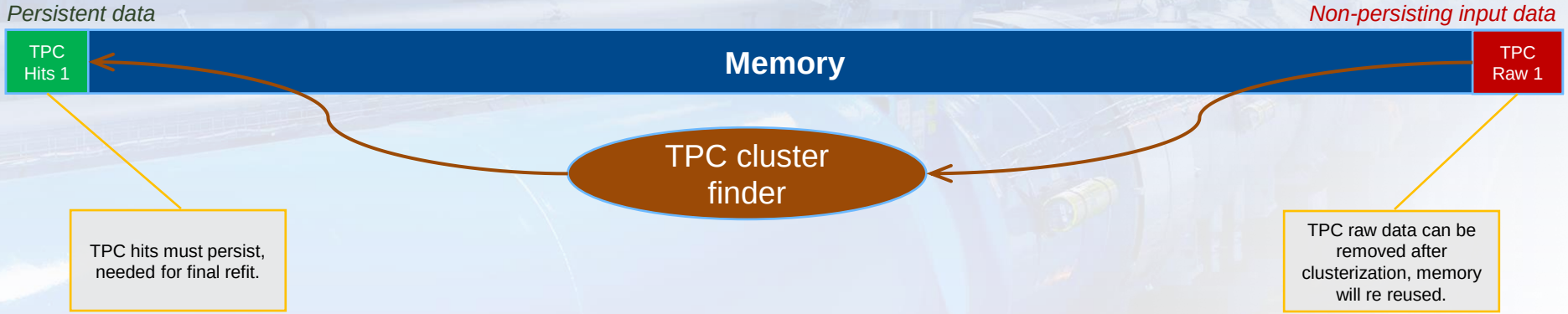
- Plot shows TPC Clusterization stage (part with **heaviest DMA transfers**).



- **Full profile of 3 time frames: 100% GPU kernel execution:**

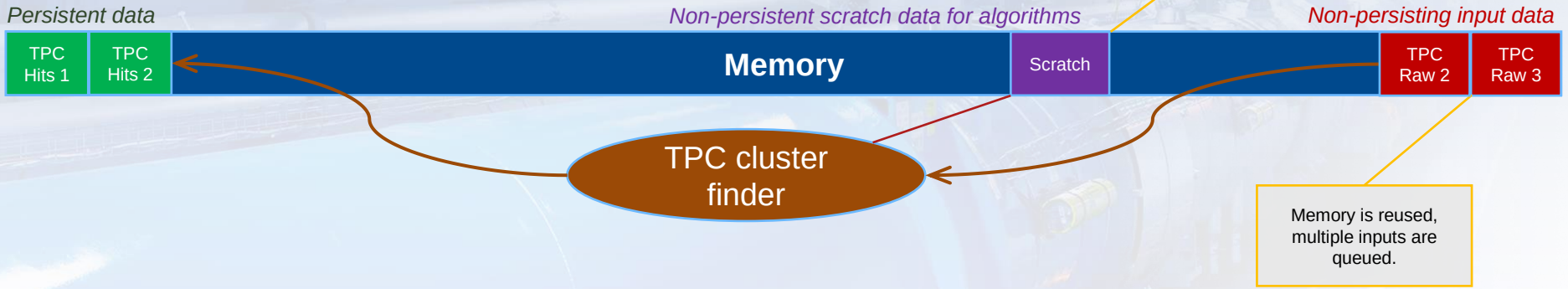
Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



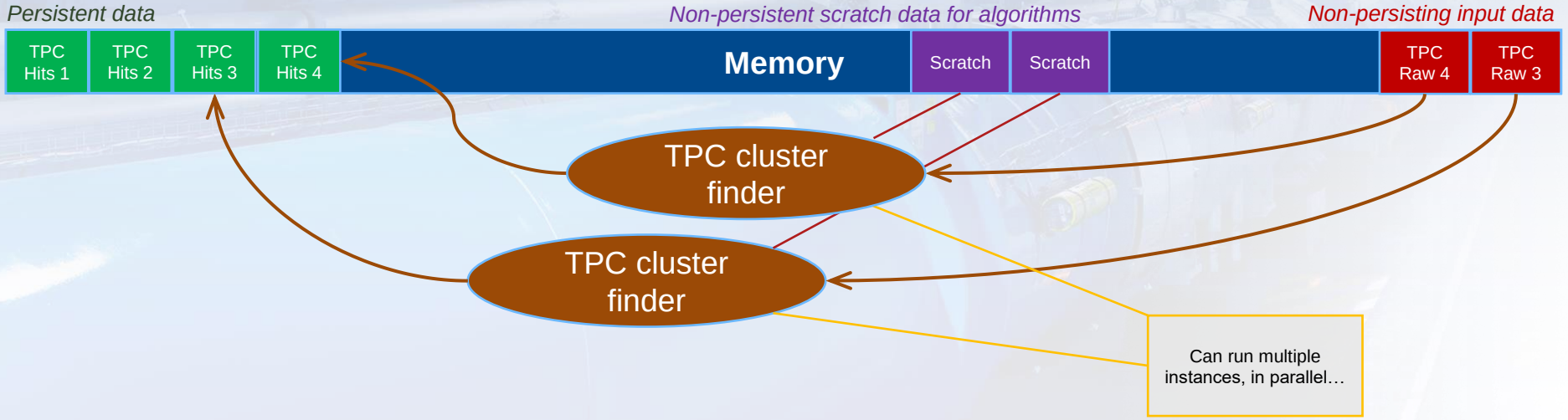
Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



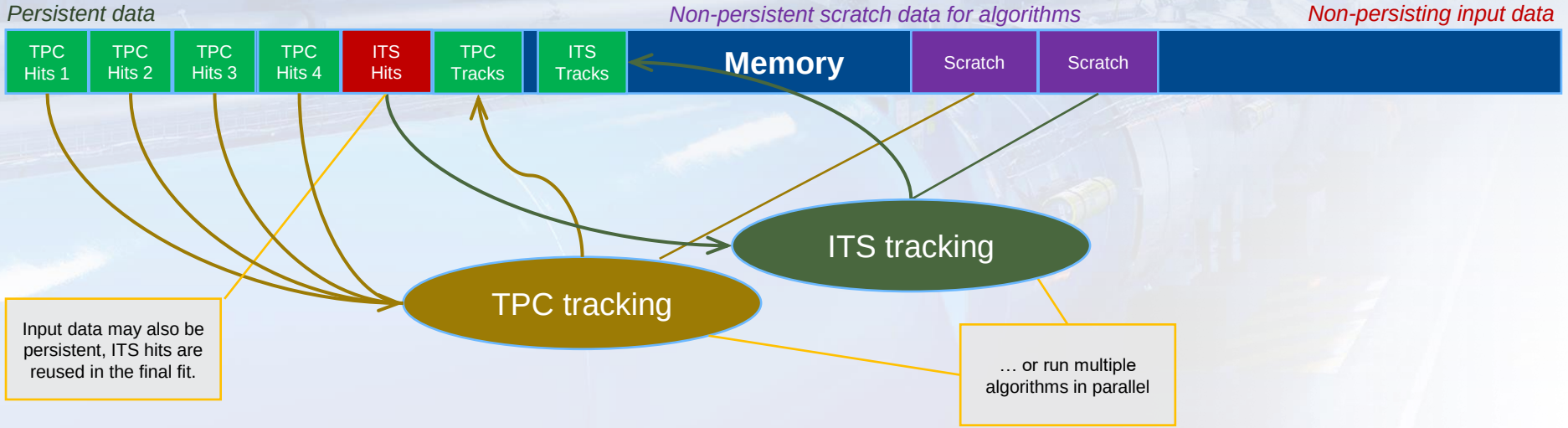
Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



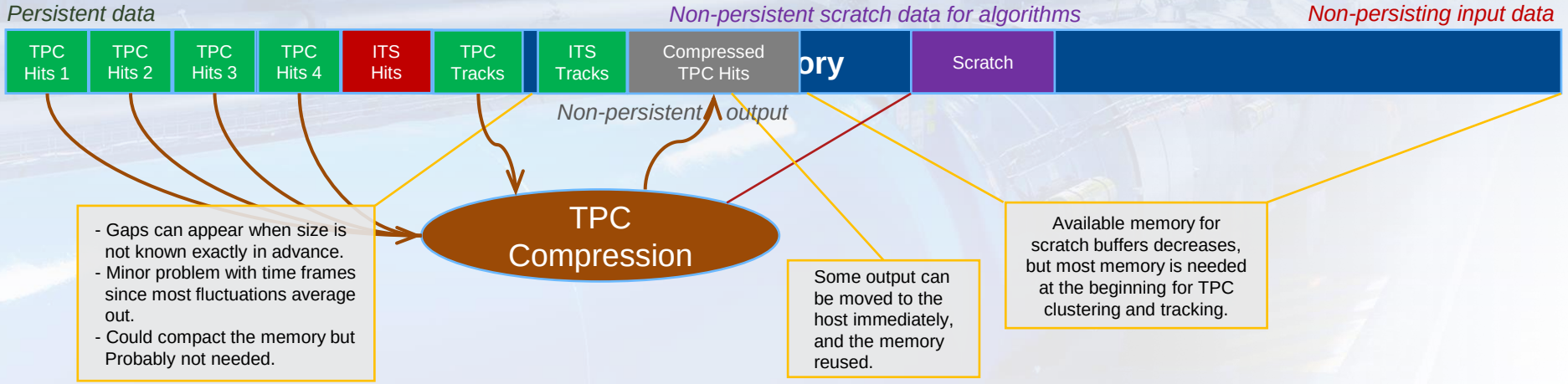
Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



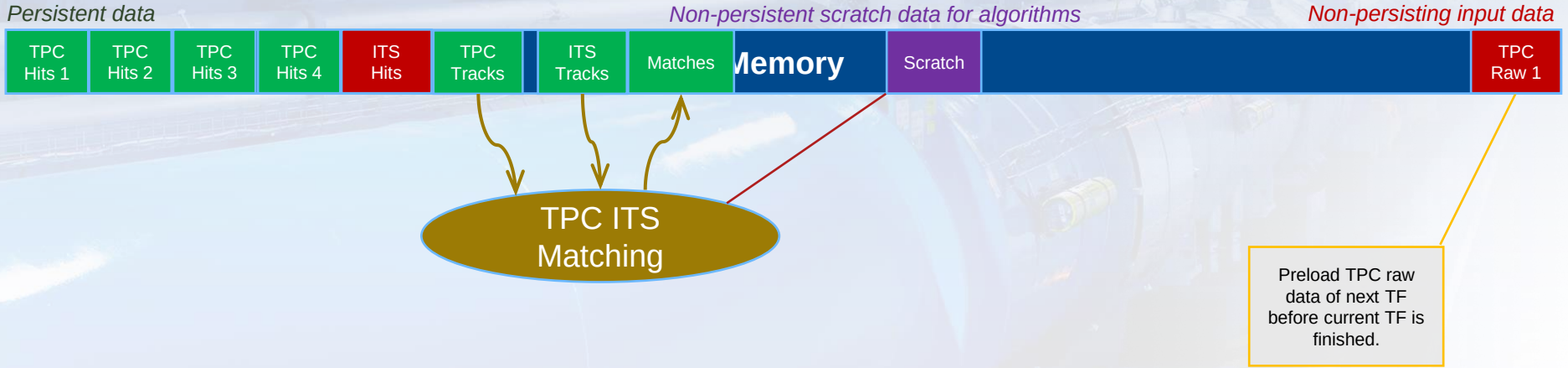
Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



Memory requirements

- **ALICE reconstructs timeframes (TF) independently: ~10 ms of continuous data / ~500 Pb-Pb collisions @ 50 kHz.**
  - TF must be processes more or less as a whole, at least in some phases all data must be available.
  - → Timeframe should fit in GPU memory. Otherwise could use a ring-buffer but that would complicate things.
  - → Need efficient memory usage.
- **Custom allocator: grabs all GPU memory, gives out chunks manually, memory will be reused when possible.**
  - Classically: reuse memory between events, collisions are not that large.
  - ALICE reuses memory between different algorithms in a TF, possibly also between independent collisions.
  - Some memory must persist during timeframe processing.



Shared source code between CPU and GPU

- **We employ a single source-code for CPU and GPU.**
  - It can parallelize on the CPU via OpenMP.
  - We support GPUs via CUDA, OpenCL, HIP.

- **CPU and GPU tracker (CUDA, OpenCL, …) share common source files.**
- **Specialist wrappers for CPU and GPU exist, that include these common files.**

common.cpp:
```
__DECL FitTrack(int n) {
….
}
```

cpu_wrapper.cpp:
```
#define __DECL void
#include ``common.cpp``

void FitTracks() {
  for (int i = 0;i < nTr;i++) {
    FitTrack(n);
  }
}
```

cuda_wrapper.cpp:
```
#define __DECL __ device void
#include ``common.cpp``

__global void FitTracksGPU() {
  FitTrack(threadIdx.x);
}

void FitTracks() {
  FitTracksGPU<<<nTr>>>();
}
```

Opencl_wrapper.cl
```
#define __DECL void
#include ``common.cpp``

__kernel void FitTracksGPU() {
  FitTrack(threadIdx.x);
}
```

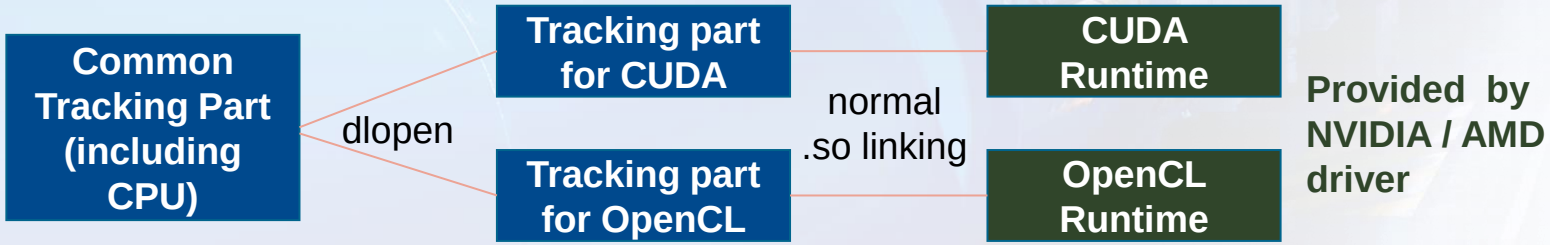
Opencl_wrapper.cxx
```
void FitTracks() {
  clEnqueueNDRangeKernel(…
  FitTracksGPU, ….);
}
```

→ **Same source code for CPU and GPU version**
- The macros are used for API-specific keywords only.
- The fraction of common source code is above 90%.

Distribution of same software to CPU and non-GPU nodes

- **A lesson learned in the HLT farm: we should have a common software package:**
  - The HLT farm consisted of GPU-equipped nodes and nodes without GPU.
  - GPU code that links to the CUDA runtime requires the CUDA runtime library to be present.
  - Just installing the library on non-GPU nodes is insufficient, as the library fails to load when the kernel module is not loaded.
  - Back then, the kernel module failed to load without an NVIDIA device present.

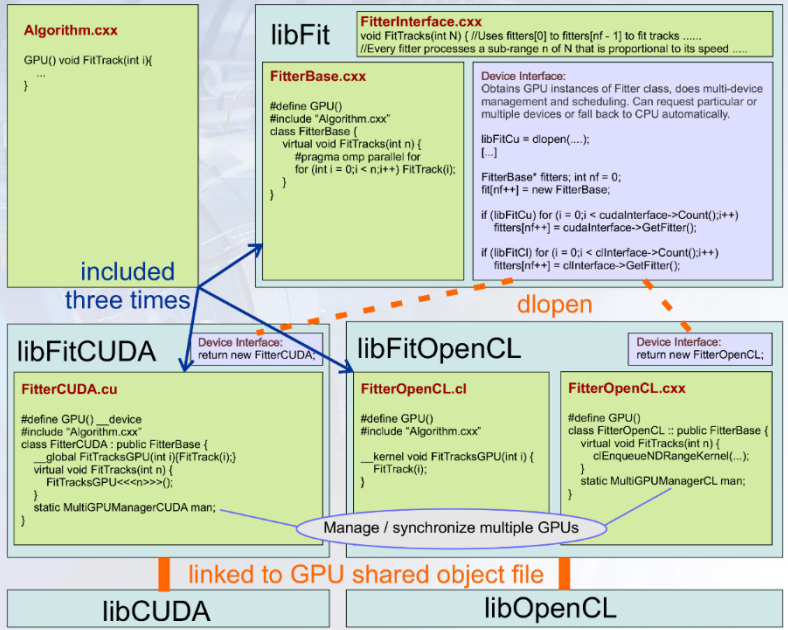
- **Still, we do not want to ship different software packages.**
  - To facilitate software distribution, we have one binary package that contains all versions.
    - The GPU versions of the code are contained in special GPU-tracking libraries.
    - These GPU-tracking libraries are accessed via dlopen.
    - Only the GPU-tracking libraries link to the GPU driver / runtime.
      - → The tracking software (without GPU acceleration) runs on all compute nodes, irrespective of the presence of a driver.



Compatibility with several GPU frameworks

- **Generic common C++ Code compatible to CUDA, OpenCL, HIP, and CPU (with pure C++, OpenMP, or OpenCL).**
  - OpenCL needs clang compiler (ARM or AMD ROCm) or AMD extensions (TPC track finding only on Run 2 GPUs and CPU for testing).
  - Certain worthwhile algorithms have a vectorized code branch for CPU using the Vc library.
  - All GPU code swapped out in dedicated libraries, same software binaries run on GPU-enabled and CPU servers.

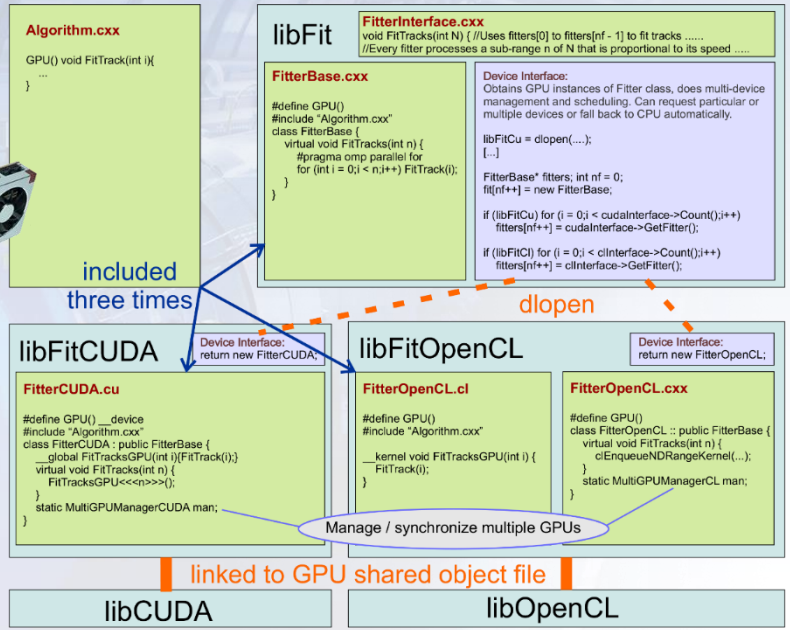
Still simplified example, e.g.
GPU measurement and tracking
are in fact independent libraries…



Compatibility with several GPU frameworks

- **Generic common C++ Code compatible to CUDA, OpenCL, HIP, and CPU (with pure C++, OpenMP, or OpenCL).**
  - OpenCL needs clang compiler (ARM or AMD ROCm) or AMD extensions (TPC track finding only on Run 2 GPUs and CPU for testing).
  - Certain worthwhile algorithms have a vectorized code branch for CPU using the Vc library.
  - All GPU code swapped out in dedicated libraries, same software binaries run on GPU-enabled and CPU servers.

- **Screening different platforms for best price / performance.**
  (including some non-competitive platforms for cross-checks and validation.)
  - **CPUs** (**AMD Zen**, **Intel Skylake**)
    C++ backend with **OpenMP**, AMD **OCL**
  - **AMD GPUs**
    (**S9000** with **OpenCL 1.2**, **MI50** / **Radeon 7** / **Navi** with **HIP** / **OCL 2.x**)
  - **NVIDIA GPUs**
    (**RTX 2080** / **RTX 2080 Ti** / **Tesla T4** with **CUDA**)
  - **ARM Mali GPU** with **OCL 2.x**
    (Tested on dev-board with Mali G52)



A quick word on compilation:

- For CUDA / HIP, normally just use the compiler via CMake (nvcc / hipcc).
  - Can also just use other compilers, e.g. clang for CUDA.
- Initially, wanted to avoid OpenCL run time compilation:
  - → Small binary that is compiled in the Cmake process, compiles the program, and dumps the binary code.
  - → In the meantime we have just replaced this by clang and llvm-spirv converter (still, spirv support in the backend is disappointing…).
- In the end, OpenCL run time compilation is not that bad…
  - …preprocess all sources into a single .cl file, include the source in our software, compile at runtime.
  - This actually allows additional optimization, e.g. we support in some places replacing runtime-constant variables by constexpr.
- That said, run time compilation really is not that bad.
  - In the meantime, have implemented RTC support for NVIDIA (tried nvrtc but ran into some issues, now just using nvcc with fatbin output).
  - Working on RTC with HIP.
- In any case: the executable by itself will only contain the CPU code.
  - GPU code in optional libraries, loaded at runtime, defined by command line argument / env variable / etc.
  - RTC can be enabled optionally and replaces the compile-time compiled code (but delays startup significantly).

Programming framework

- **For ALICE: everything more or less manual, CMake, C++, and some preprocessor magic.**
  - Not beautiful but it works.

- **Why not a modern framework like Alpaka?**
  - Primary reason: historically – we started like this in Run 1, it works, never change a running system.
  - Our approach gives access to all API specific features if protected by #ifdef.
  - Actually the framework is rather lightweight:
    - Started with few 100 lines of code – OK, in the meantime it has grown to 5k lines today.
    - But this contains a lot of general framework code for synchronization, memory management, etc.
    - Includes the backends for CPU / OpenMP / CUDA / HIP / OpenCL.
    - Still quite manageable.
  - One problem with general frameworks:
    - Abstraction layers solve all complexity problems except for thing with too many layers of abstraction…
    - Not clear if Alpaka would be fully sufficient for us, so perhaps we would still add something on top / below.
- **Would we do it again this way?**
  - I would certainly look into some established frameworks first (Don't reinvent the wheel for general principle).
  - But to be honest, not sure what would be the outcome.

Code optimization

- **Try to keep as much code shared as possible.**
  - Using some #ifdefs to have some shortcuts for serial CPU version, or use special GPU optimizations.
  - Some compiler macros, e.g. to cache data in GPU shared memory.
- **Can use different depth of abstraction:**

Less API-specific code

Baseline:
- **Kernels are common and called by wrappers**

Less complicated abstraction

More abstraction:
- **Also kernel calls, outer loops, and general workflow common (used in TPC)**
- **Exact same algorithm for CPU / GPU**

More flexibility:
- **Common code for base classes, virtual class with "traits" for CPU and GPU steer the workflow. Main tracking class is common and uses the traits. (used in ITS)**

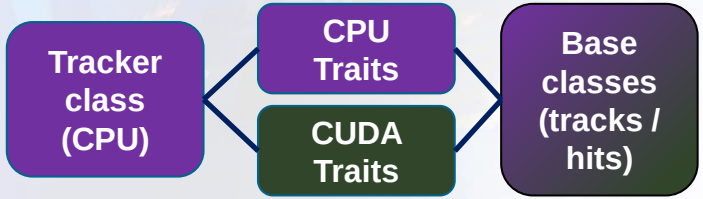
```
runKernel<krnlTrackFit>({nBlocks, nThreads}, {&eventWait, &eventRecord}, nTracks);
```

Number of threads/blocks on GPU/GPU *Events for synchronization*

Kernel arguments, variable number, passed as Variadic macro

Actual kernel is static class function, selected via template parameter

General kernel call, will dispatch to CUDA / OpenCL / CPU API via virtual interface



Some words on the technical implementation

- Examplary definition of an algorithm.

krnlName, as krnlTrkFit in the last slide.

Give the related reconstruction task a name, just for accounting during debugging / benchmarking.

Algorithm can consist of multiple consecutive kernels, so give them names.

This algorithm requires some shared memory, and derives from some common predefined shared memory algorithms.

Main processing function, called nBlocks * nThreads times, in the obvious GPU grid configuration. Additional parameters forwarded via variadic template.

Processor is a struct that contains the context for the algorithm (can also be passed in via additional variadic template parameters).

```
class GPUTPCCompressionKernels : public GPUKernelTemplate
{
 public:
  GPUhdi() constexpr static GPUDataTypes::RecoStep GetRecoStep() { return GPUDataTypes::RecoStep::TPCCompression; }

  enum K : int {
    step0attached = 0,
    step1unattached = 1,
  };

  struct GPUSharedMemory : public GPUKernelTemplate::GPUSharedMemoryScan64<int, 256> {
    GPUAtomic(unsigned int) nCount;
    unsigned int lastIndex;
    unsigned int sortBuffer[512];
  };

  template <int iKernel = defaultKernel>
  GPUd() static void Thread(int nBlocks, int nThreads, int iBlock, int iThread, GPUsharedref() GPUSharedMemory& GPUrestrict() smem,
    processorType& GPUrestrict() processors, ...);
```

Some words on the technical implementation

- Examplary definition of a context.

The framework will create 2 instances of the processor:

- One in host memory.
- One in constant GPU memory.
- Processor is copied there before algorithms starts.
- Memory is allocated twice, once on host, once on GPU.
  - Allocations defined in the callback, sizes can be derived from variables (e.g. nMaxTracks).
  - Processors points to correct memory location.
  - Flags in the callback can suppress allocation on either host or GPU, if the memory is not needed there.

→ Nice debugging feature: If all memory is allocated on both memory spaces and has the same size, it can easily be copied forth and back.

→ Execution can be interrupted between every kernel, moved between host or device, and continued on the other side.

```
struct GPUTPCCompression : public GPUProcessor
{
#ifndef GPUCA_GPUCODE
  void InitializeProcessor();              // Callbacks only available if compiled for the host
  void RegisterMemoryAllocation();
#endif

  static constexpr unsigned int P_MAX_QMAX = 1 << 10;
  // […]

  // Ptrs to GPU / host memory
  memory* mMemory = nullptr;
  unsigned int* mAttachedClusterFirstIndex = nullptr;
  unsigned char* mClusterStatus = nullptr;

  // some (runtime) constants.
  unsigned int nMaxTracks;
}
```

Usage of this semantic is basically optional, all ptrs can be passed in as additional variadic arguments.
Just give the developers the freedom they want:

- Better provide optional features than force people into constraints they might not like.

Avoiding #ifdef

- **As said, we allow different code paths for different architectures, if there is a real benefit.**
  - Many things can be done without #ifdef, but e.g. with templates.
  - And try to keep the mess out of the actual algorithm.
  - (OK, one could also encapsulate reading from / writing to the charge array in a class applying the factor automatically…. but just to demonstrate the idea….)

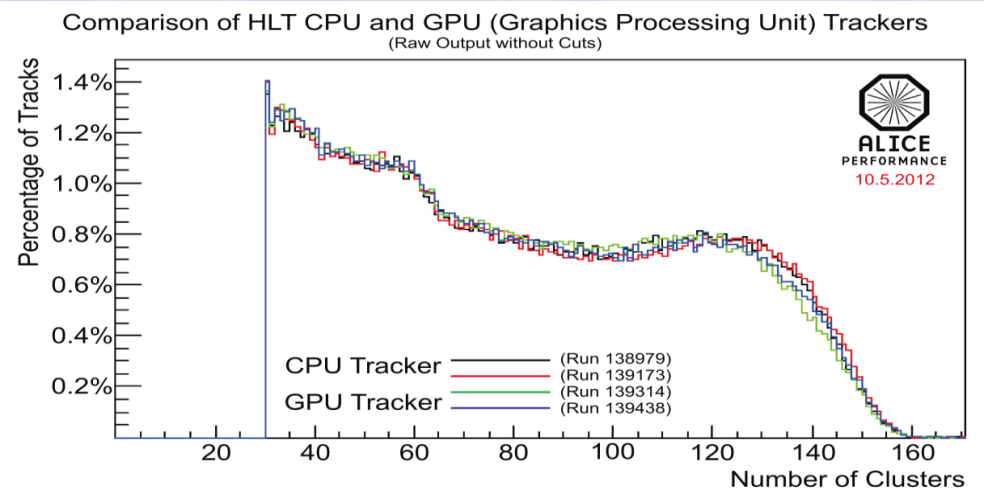
```
// Architecture namespace defined via #ifdefs / different includes

Namespace {
template <typename T, typename fake = void> struct scalingFactor;
template <typename fake> struct scalingFactor<unsigned short, fake> {
   static constexpr float factor = 4.f;
};
template <typename fake> struct scalingFactor<float, fake> {
   static constexpr float factor = 1.f;
};
template <typename fake> struct scalingFactor<half, fake> {
   static constexpr float factor = 1.f;
};
}

struct someAlgorithm { // (here dE/dx energy loss)
   using chargeDataType = architecture::dataTypes::lowPrecisionFloat;
   chargeDataType chargeArray[128];
   […]
   chargeArray[i] = cl.getCharge() * scalingFactor<chargeDataType>::factor;
   […]
};
```

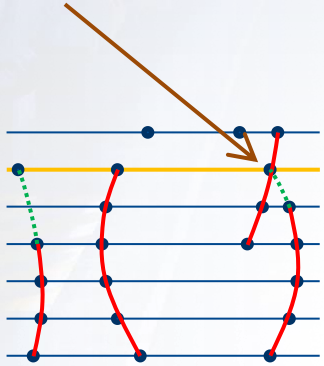
Consistency of Tracking Results

- **First version: Inconsistent results between CPU / GPU**
- **Root causes:**
  - Concurrency
  - Non-associative floating point arithmetic



- Problem: Cluster to track assignment was depending on the order of the tracks.
  - Each cluster was assigned to the longest possible track. Out of two tracks of the same length, the first one was chosen.
  - Concurrent GPU tracking processes the tracks in an undefined order.
- Solution: Need a continuous (floating point) measure of the track quality.
  - Two 32-bit floats can still be identical, but that is unlikely.

- In general: If the order doesn‘t matter, just still take something that is deterministic (e.g. random spatial coordinate).



Comparison of tracking results

- **To simplify Q/A, GPU and CPU results should be as close as possible!**
- **100% identical result on binary level is difficult.**
  - Prevents all fast-match optimization → >10% performance loss.
  - Difficult to kill concurrency issue completely:
    - 32-bit float gives ca 25 bit of entropy (do not use full range of exponent).
    - 1000 Events of ~ 10000 tracks per time frame → collision probability non-zero.
    - Could use double / 64-bit integer…

- **Good measure for us:**
  - Do not compare track parameters (float values might differ).
  - Compare cluster to track association.
    - Almost binarily compatible: 0.00024% of clusters assigned differently.

Summary

- GPUs acquired mostly for synchronous TPC processing (impossible without / prohibitively expensive).
  - Since we have them, use them in as many places as possible.
- Software runs as independent algorithms operating on data in shared buffers (can be in GPU memory or in host memory).
- Pipeline processing hides data transfers.
- Efficient memory usage is critical to process large time frames.
- Algorithms (majority of the code) written in generic C++, special keywords come in via preprocessor marcros.
- Two approaches for workflow:
  - Fully generic code, common workflow on CPU and GPU.
  - Traits for CPU and GPU that implement parts of the algorithm, more flexibility but more code duplication.
- Can run on CPU / CUDA / HIP / OpenCL.
- Consistent and reproducible results make your live much easier.