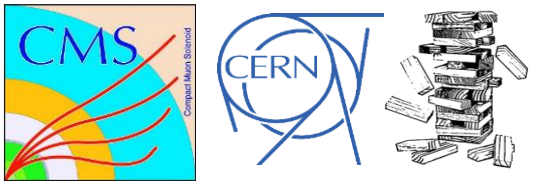


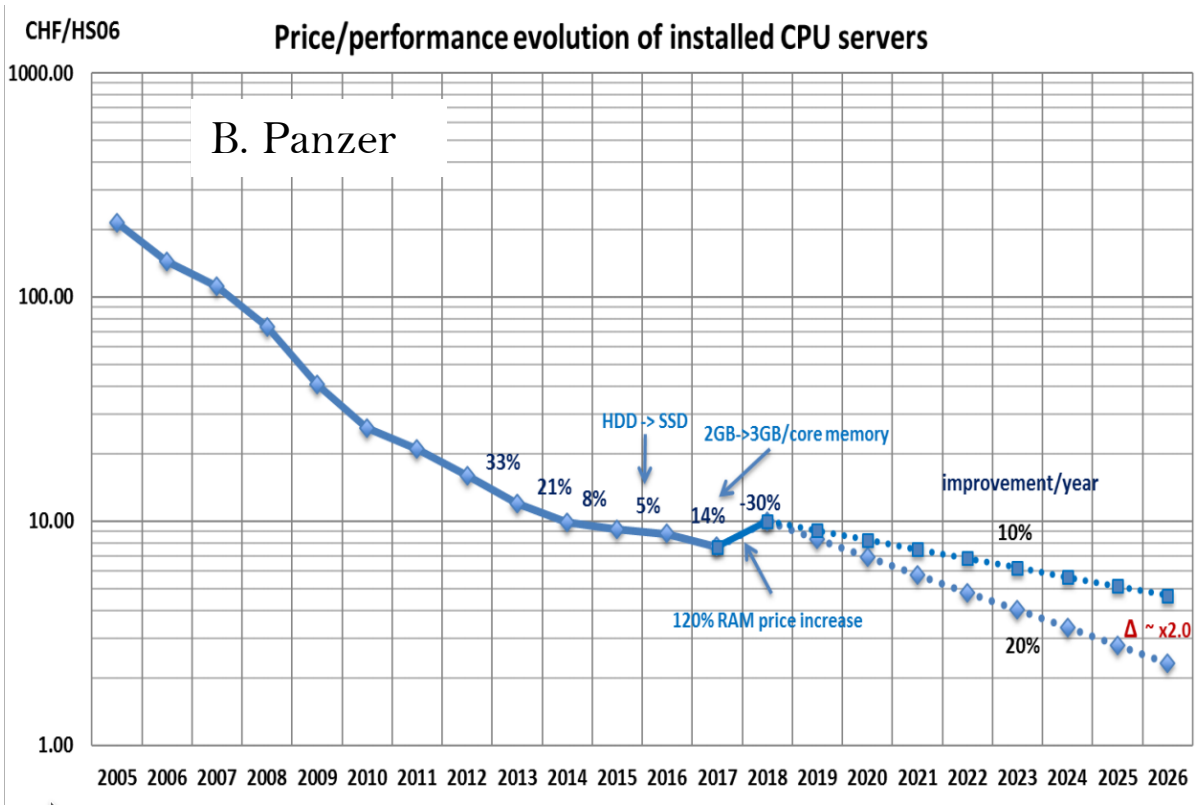
The Heterogeneous Computing Revolution

Felice Pantaleo

CERN - Experimental Physics Department

felice@cern.ch

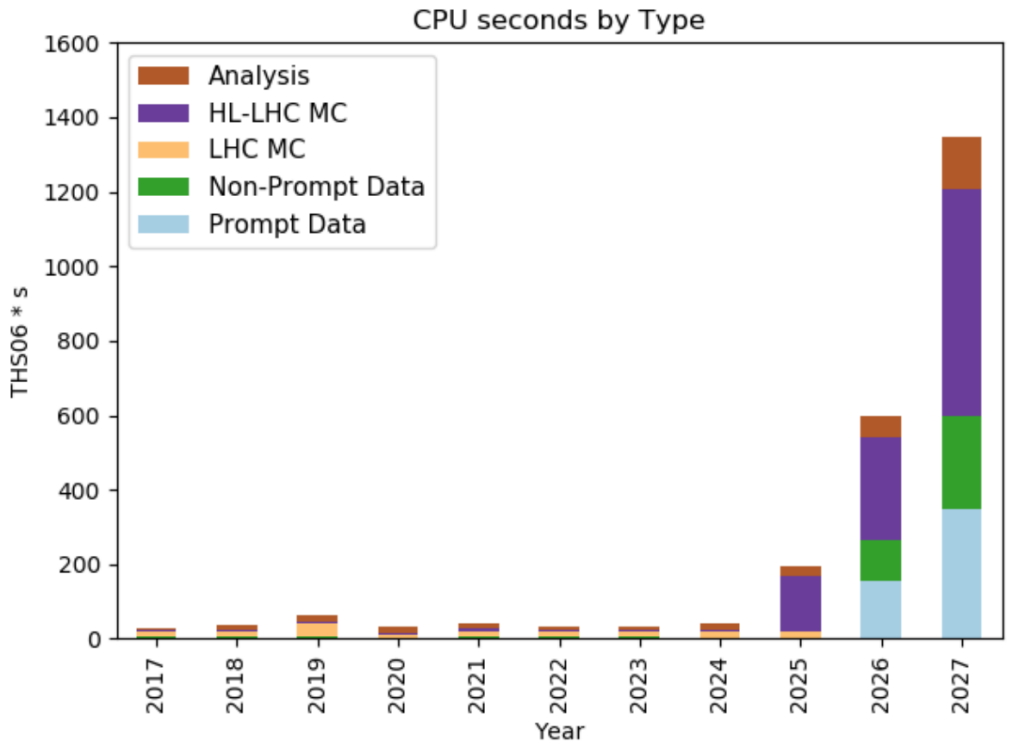
A reminder that…



CPU evolution is not able to cope with the increasing demand of performance

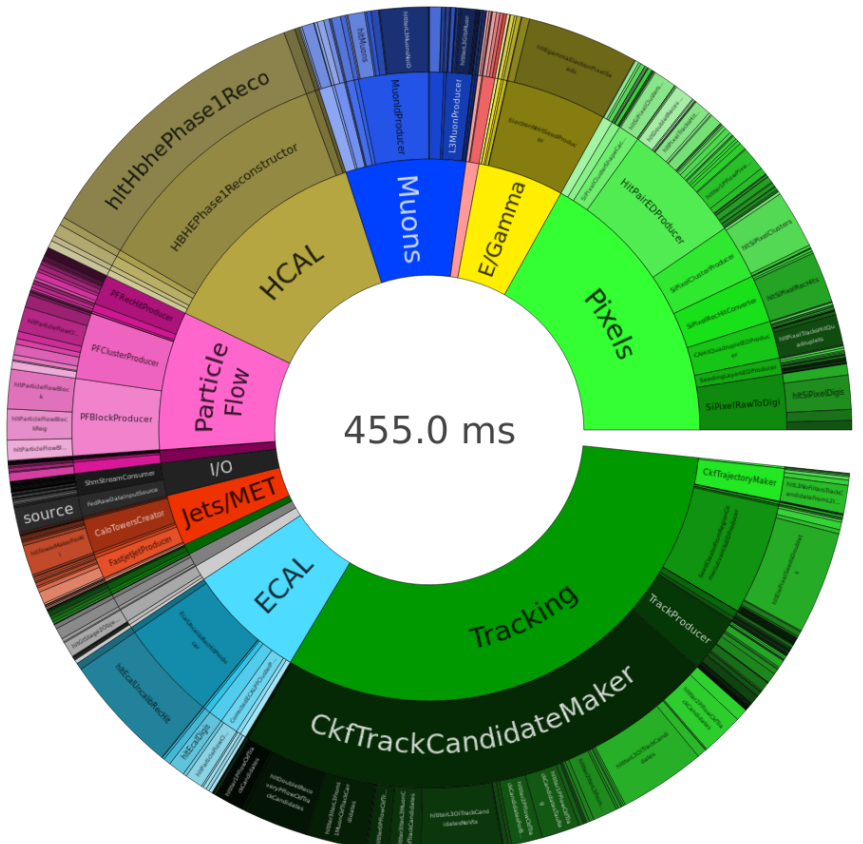
…CMS is in a computing emergency

- Performance demand will increase substantially at HL-LHC
- an order of magnitude more CPU performance offline and online



Today

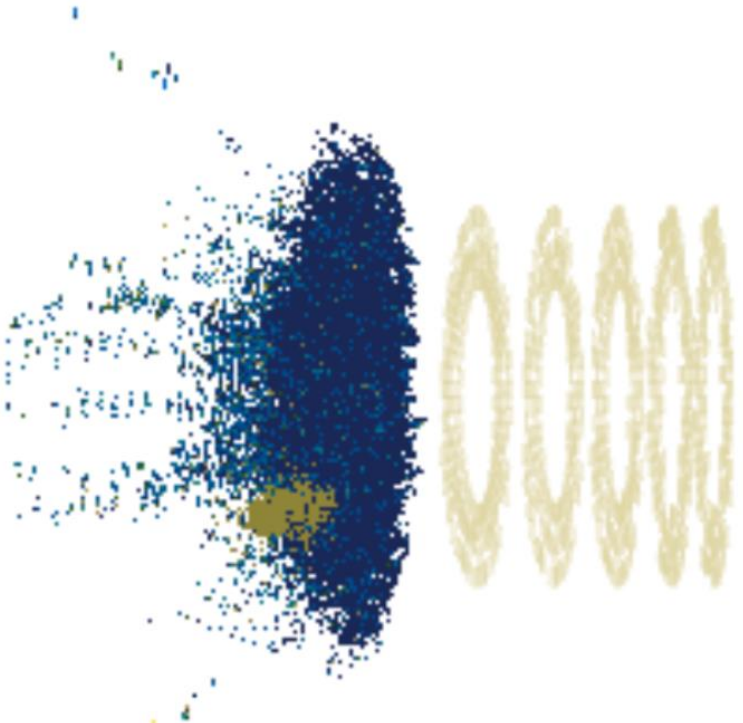
- High Level Trigger
  - readout of the whole detector with full granularity, based on the CMS software, running on 30,000 CPU cores
  - Maximum average latency is ~600ms with HT



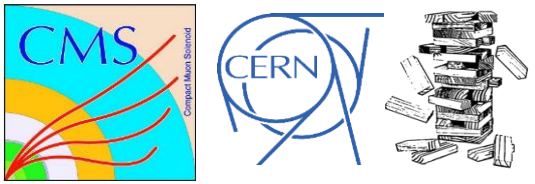
Circles

The CMS Trigger in Phase 2

- Level-1 Trigger output rate will increase to 750 kHz (7.5x)
- Pileup will increase by a factor 3x-4x
- The reconstruction of the new highly granular Calorimeter Endcap will contribute substantially to the required computing resources
- Missing an order of magnitude in computing performance



The Times They Are a-Changin'

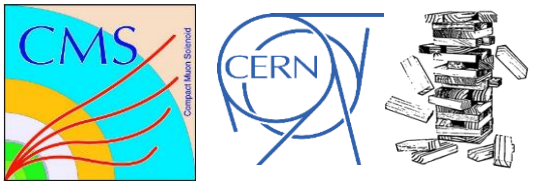


Achieving sustainable HEP computing requires change

Long shutdown 2 represents a good opportunity to embrace a paradigm shift towards modern heterogeneous computer architectures and software techniques:

- Heterogeneous Computing
- Machine Learning

Algorithms and Frameworks



The acceleration of algorithms with GPUs is expected to benefit:

- Online computing: decreasing the overall cost/volume of the event selection farm, or increasing its discovery potential/throughput
- Offline computing: enabling software frameworks to execute efficiently on HPC centers and saving costs by making WLCG tiers heterogeneous
- Volunteer computing: making use of accelerators that are already available on the volunteers' machines

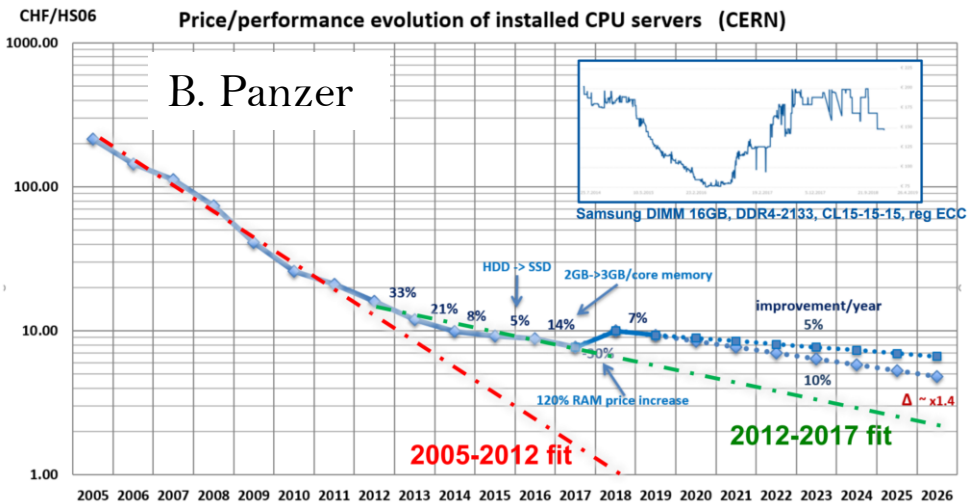
Patatrack

- Patatrack is a software R&D incubator
- Born in 2016 by a very small group of passionate people
- Interests: algorithms, HPC, heterogeneous computing, machine learning, software engineering
- Lay the foundations of the CMS online/offline heterogeneous reconstruction starting from 2020s

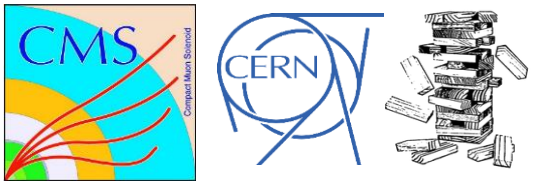
and it's growing fast

Why should our community care?

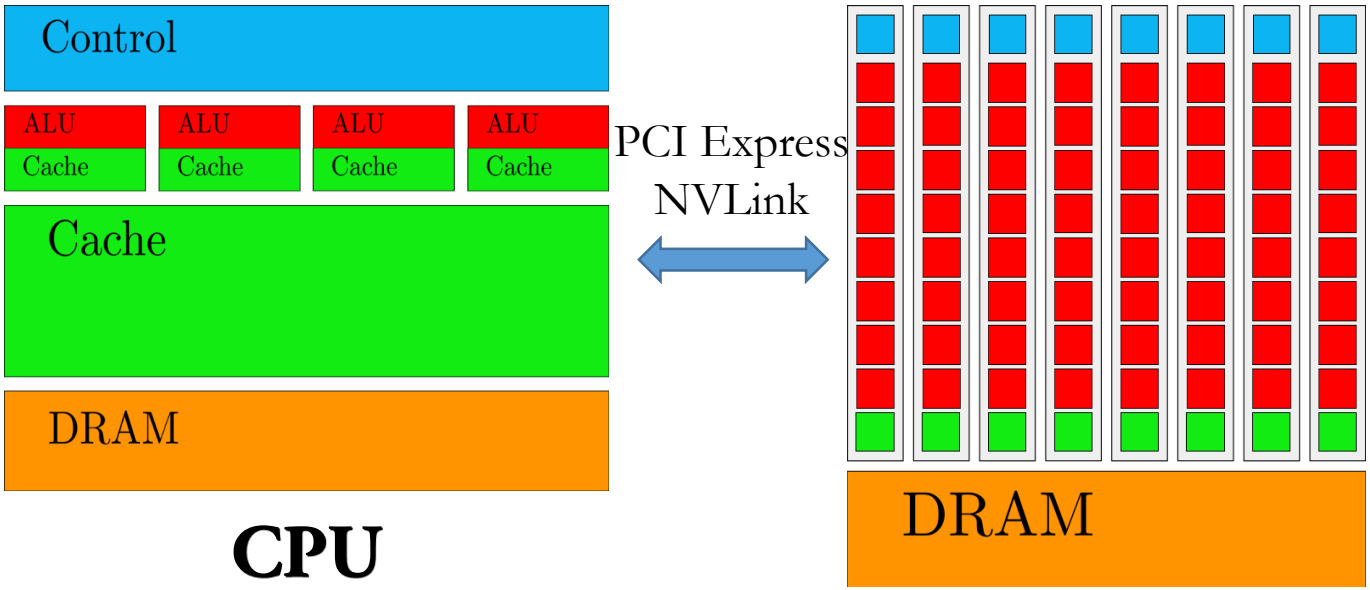
- Accelerators are becoming ubiquitous
  - Driven by more complex and deeper neural networks
  - Details hidden to the user by the FW
- Better Time-to-Solution, Energy-to-Solution, Cost-to-Solution
- Experiments are encouraged to run their software on Supercomputers
  - We are not using their GPUs
  - Summit: 190PFLOPS out of 200PFLOPS come from GPUs
- Training neural networks for production workflows is a negligible part
- Redesigning our algorithms and data structures to be well digested by a GPU can speed it up also when running on CPUs



Architectures

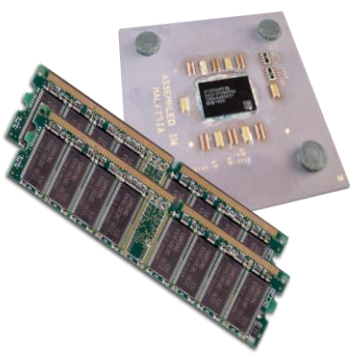


Control
ALU
Cache
DRAM



CPU

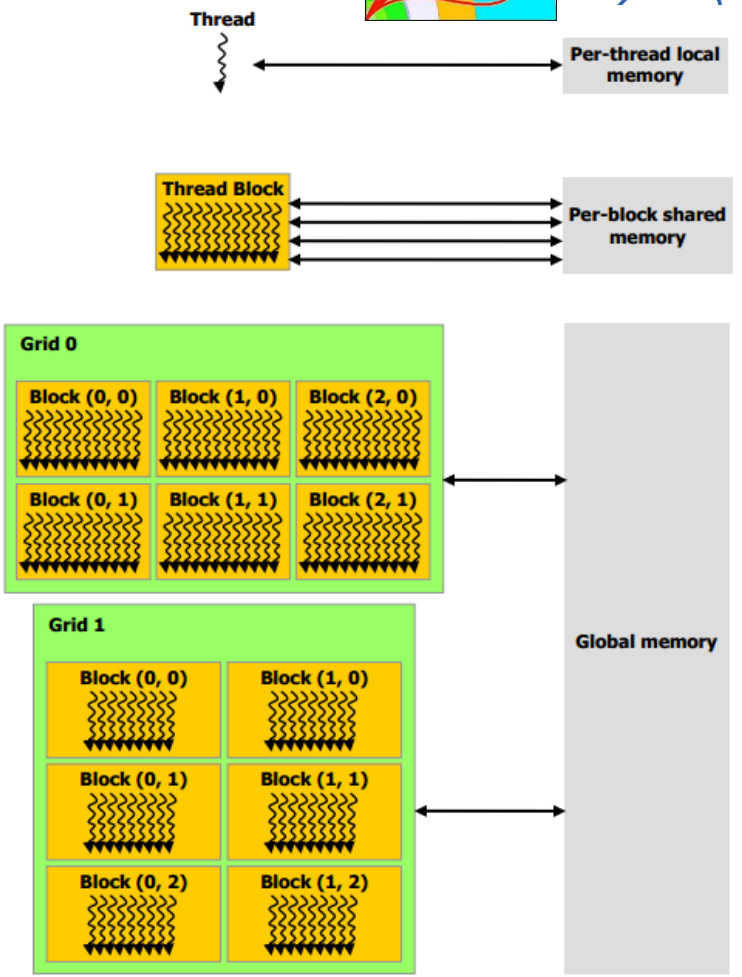
GPU



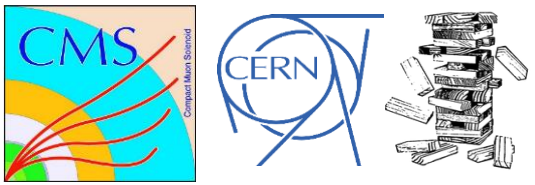
CUDA Programming model

A parallel kernel is launched on a grid of threads, grouped in blocks.

- All threads in the same block:
  - run on the same SM, in warps
  - can communicate
  - can synchronize

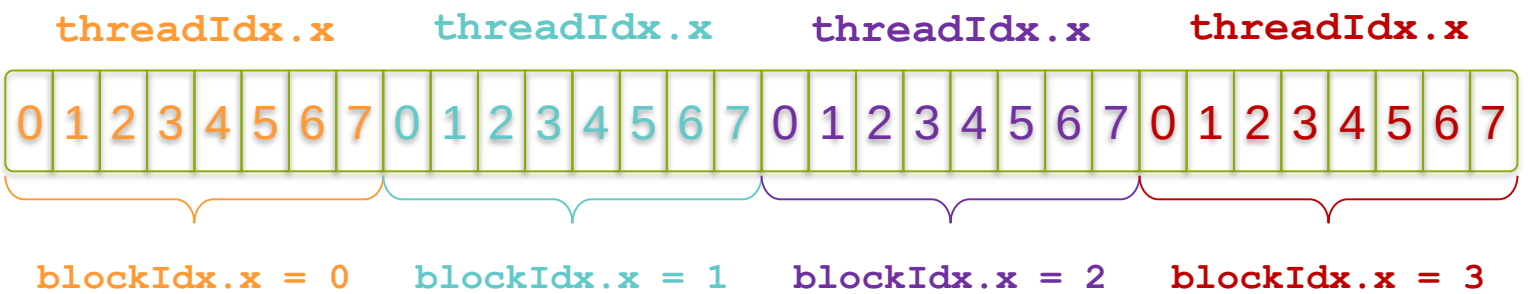


CUDA Kernels



Assign each thread a unique identifier and unroll the for loop.

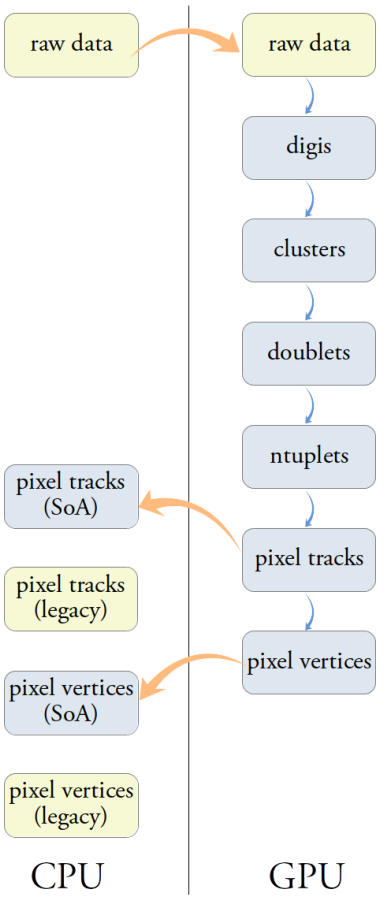
For example:




```
__global__ void add(const int *a, const int *b,
                    int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

Patatrack Pixel Reconstruction Workflow

- Full Pixel Track reconstruction in CMSSW
  - from Raw data decoding to Primary Vertices determination
- Raw data for each event is transferred to the GPU initially (~250kB/event)
- At each step data can be transferred to CPU and used to populate "legacy" event data
- The standard validation is fully supported
- Integer results are identical

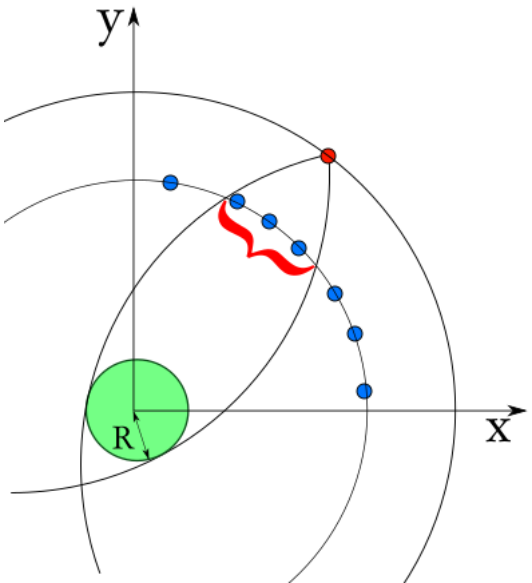


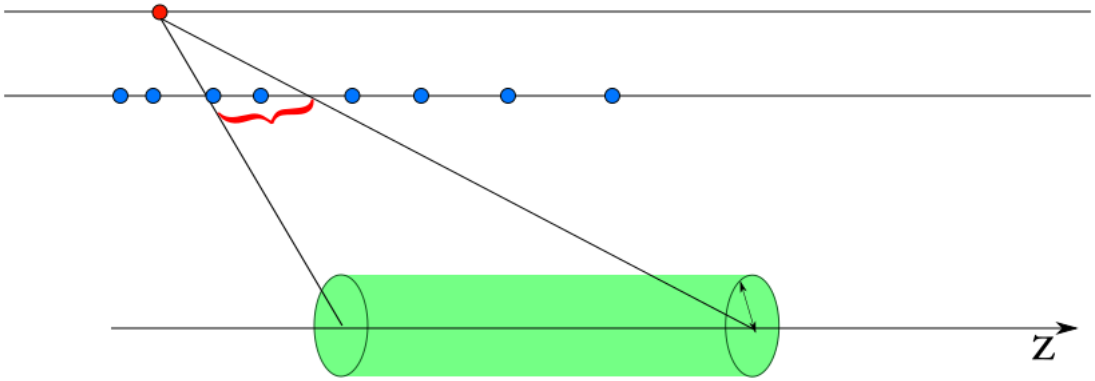
Data structures

SoA can be very well digested by GPUs (as well as CPUs)

Doublets

- The local reconstruction produces hits
- Doublets are created opening a window depending on the tracking region/beamspot and layer-pair
  - The cluster size along the beamline can be required to exceed a minimum value for barrel hits connecting to an endcap layer
- Hits within the bins are connected to form doublets if they pass further "alignment cuts" based on their actual position
- In the barrel the compatibility of the cluster size along the beamline between the two hits can be required
- The cuts above reduce the number of doublets by an order of magnitude and the combinatorics by a factor 50

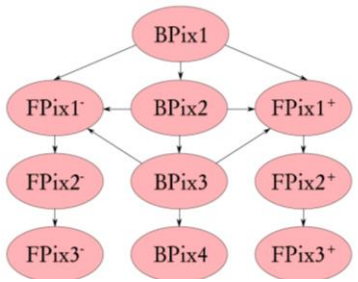


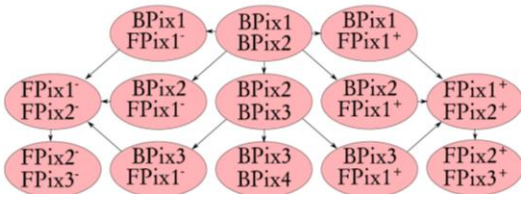


Cellular Automaton-based Hit Chain-Maker

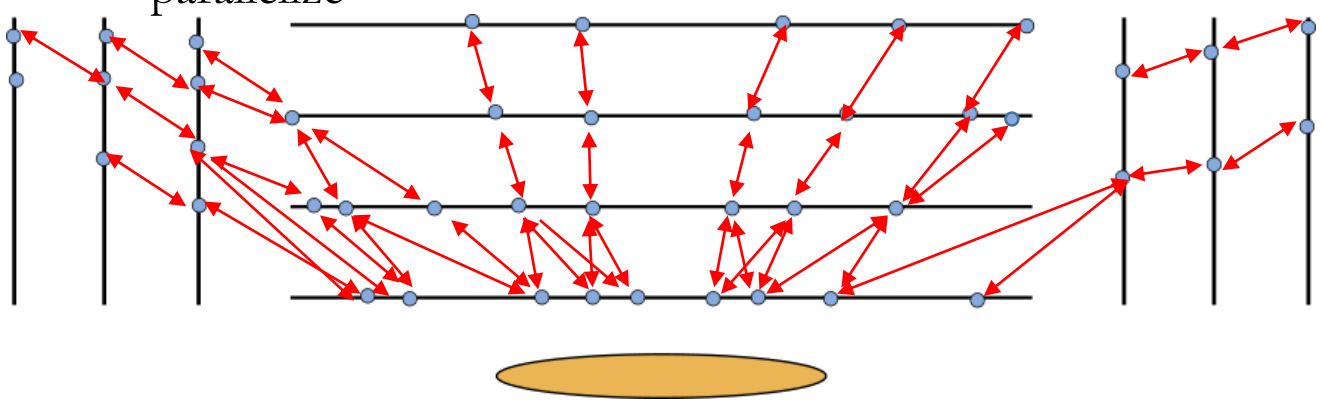
The CA is a track seeding algorithm designed for parallel architectures

It requires a list of layers and their pairings





- A graph of all the possible connections between layers is created
- Doublets aka Cells are created for each pair of layers, in parallel at the same time
- Fast computation of the compatibility between two connected cells, in parallel
- No knowledge of the world outside adjacent neighboring cells required, making it easy to parallelize

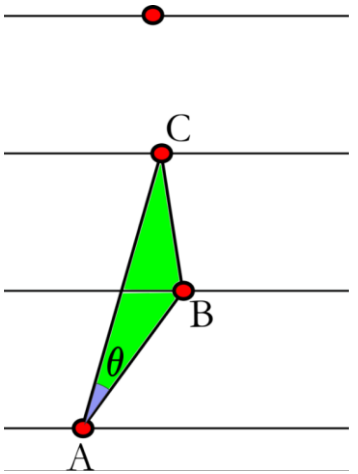


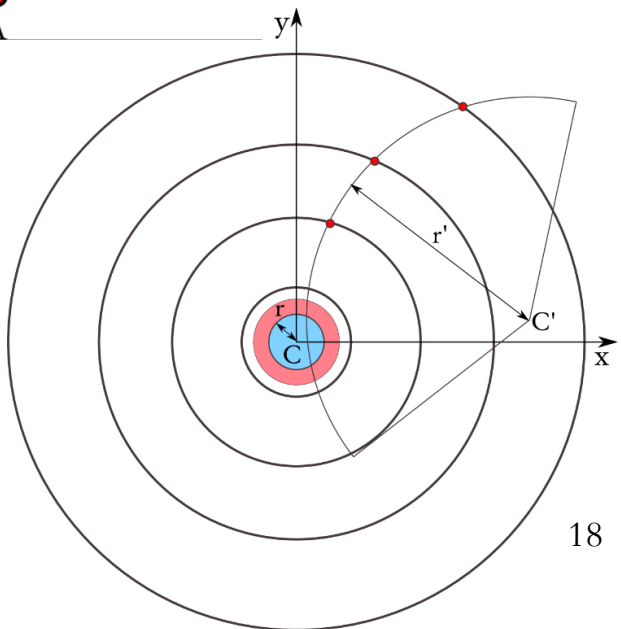
- Better efficiency and fake rejection wrt previous algo
- Since 2017 data-taking has become the default track seeding algorithm for all the pixel-seeded online and offline iterations

- In the following, at least four hits are required, but triplets can be kept to recover efficiency where geometric acceptance lacks one hit

CA compatibility cuts

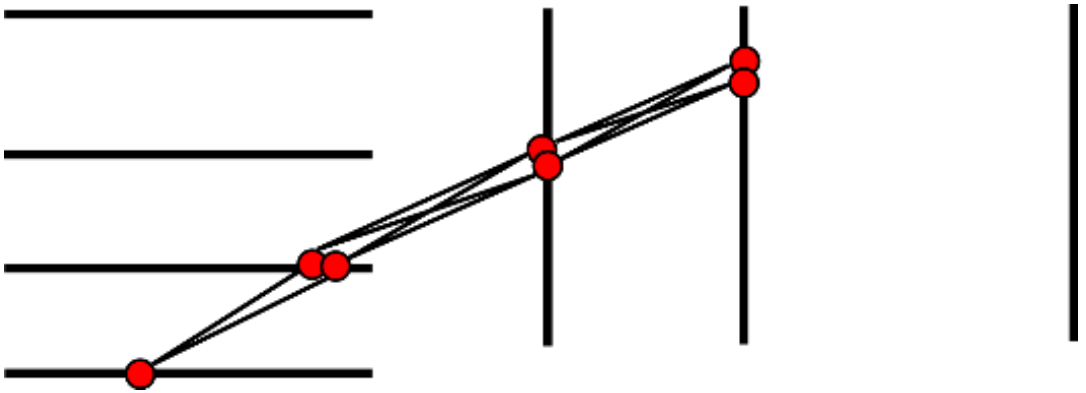
- The compatibility between two cells is checked only if they share one hit
  - AB and BC share hit B
- In the R-z plane a requirement is alignment of the two cells
- In the cross plane the compatibility with the beamspot region



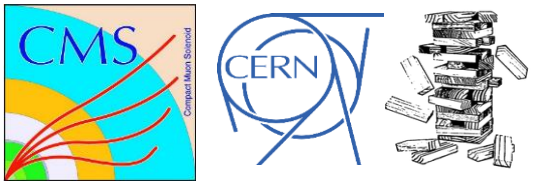


Fishbone

- After using the CA for producing N-tuplets, “fishbone” seeds can be produced to account for module/layer overlaps
- Only highest grade n-tuplet is fitted and duplicate doublets are filtered out



Fits



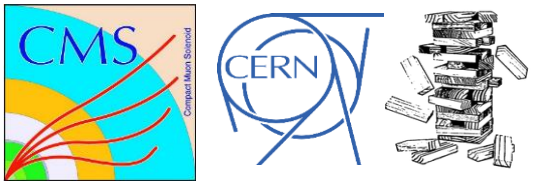
Pixel track "fit" at the HLT is still using 3 points for quadruplets and errors on parameters are loaded from a look-up table[eta][pT]

The Patatrack Pixel reconstruction includes two Multiple Scattering-aware fits:

- Riemann Fit
- Broken Line Fit

They allow to better exploit information coming from our 4-layer pixel detector and improve parameter resolutions and fake rejection

Fits - Implementation

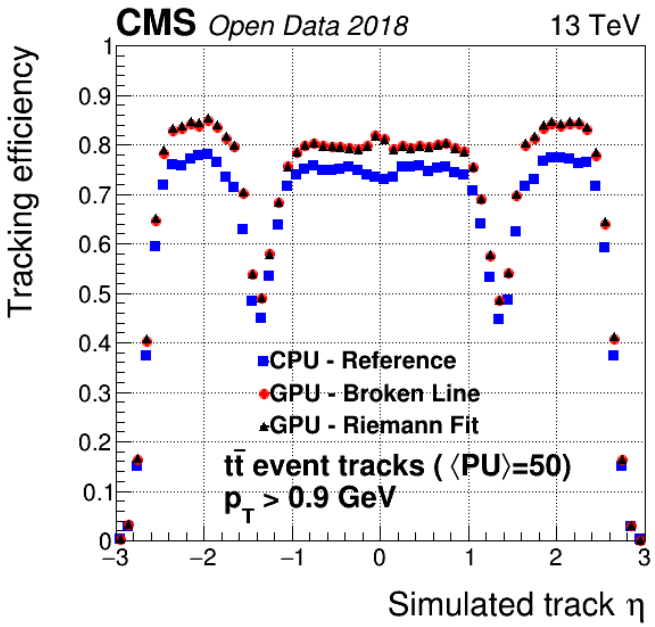


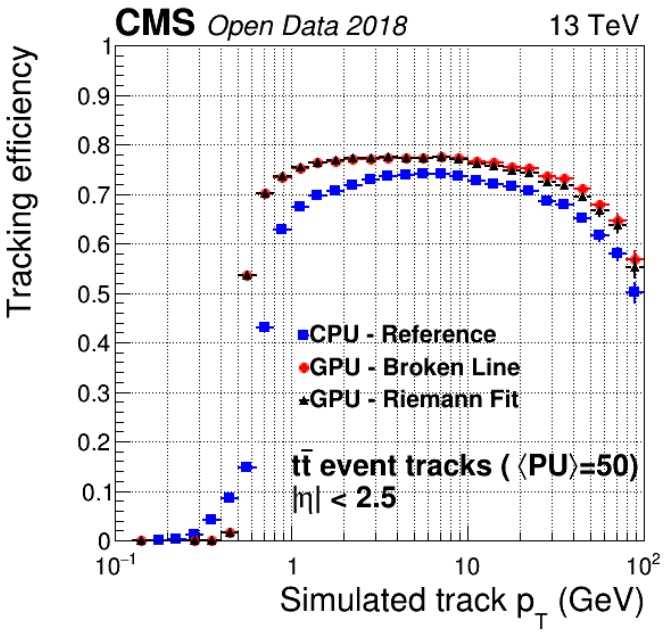
Both the Riemann and the Broken Line fits have been implemented using Eigen

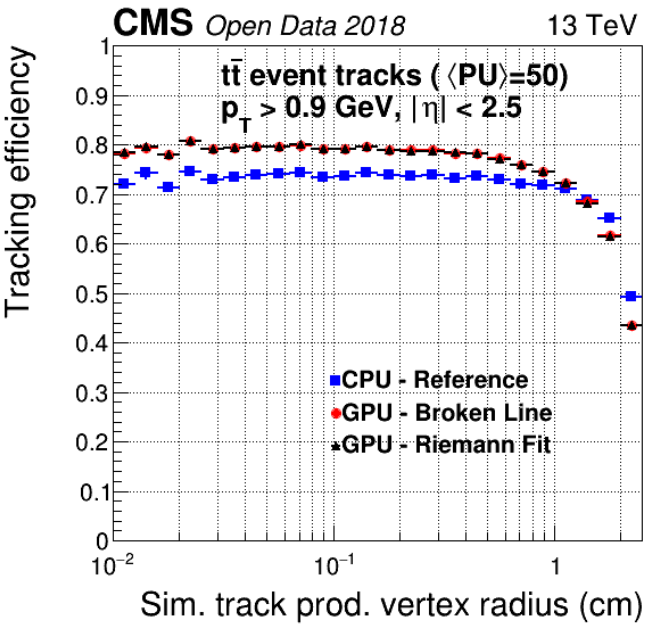
Eigen is a C++ template library for linear algebra, matrix and vector operations

This allows perfect code portability between CPU and GPU implementation and bitwise-matching of the results

Physics Performance - Efficiency

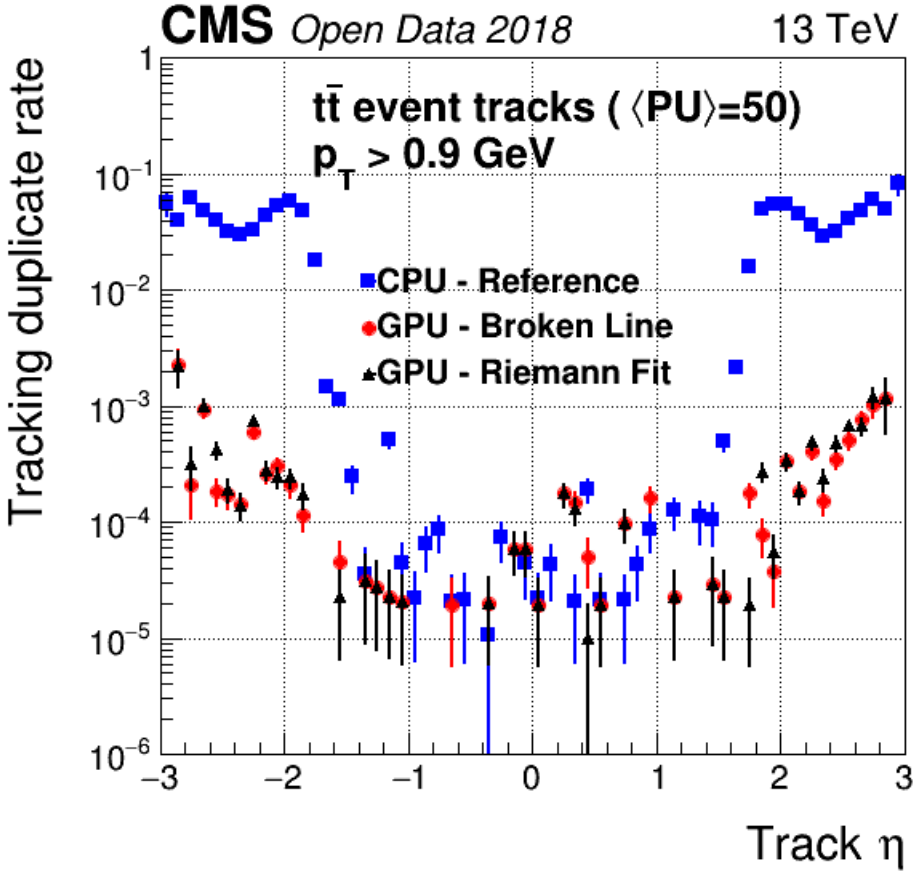


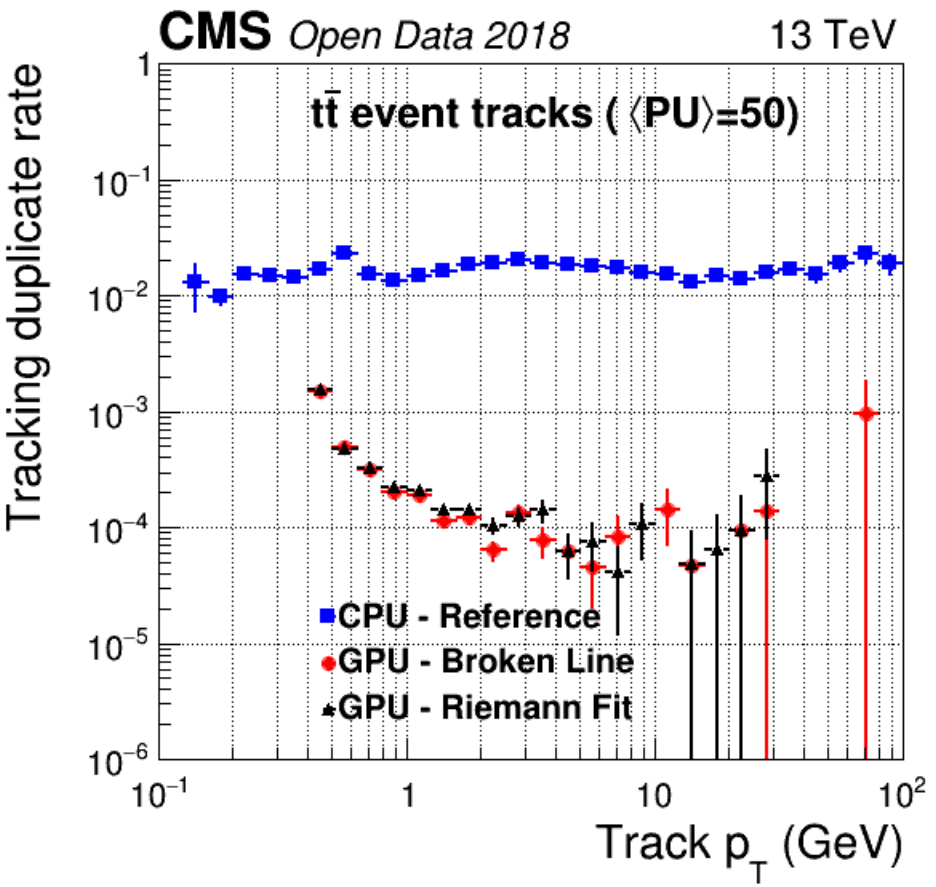




Track reconstruction efficiency as a function of simulated track η, p_T, and production vertex radius.

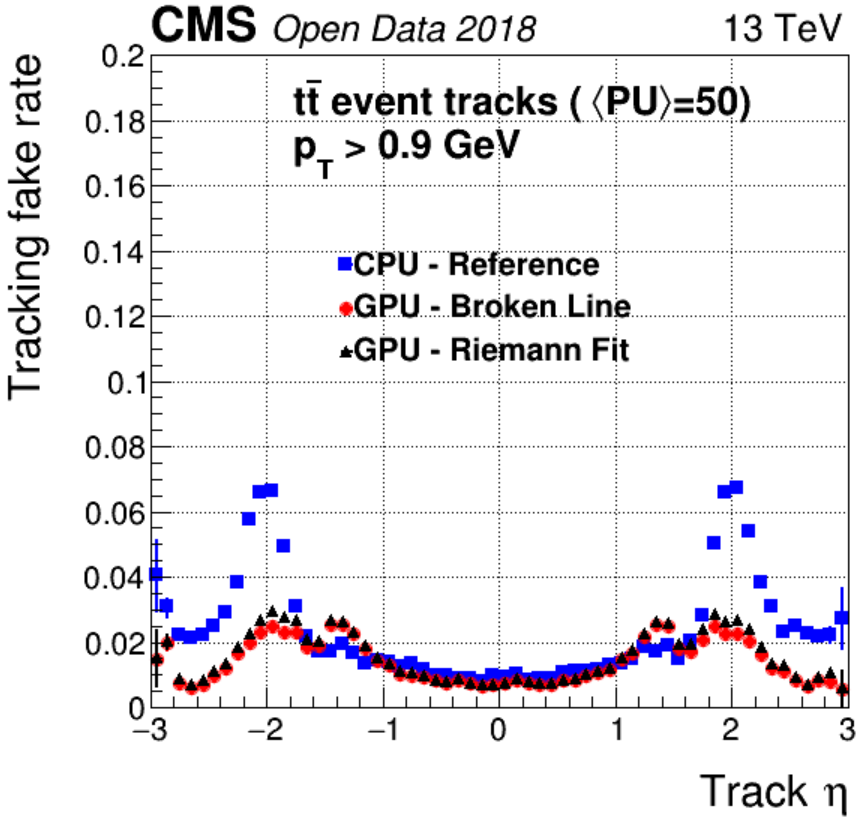
Physics performance - Duplicates

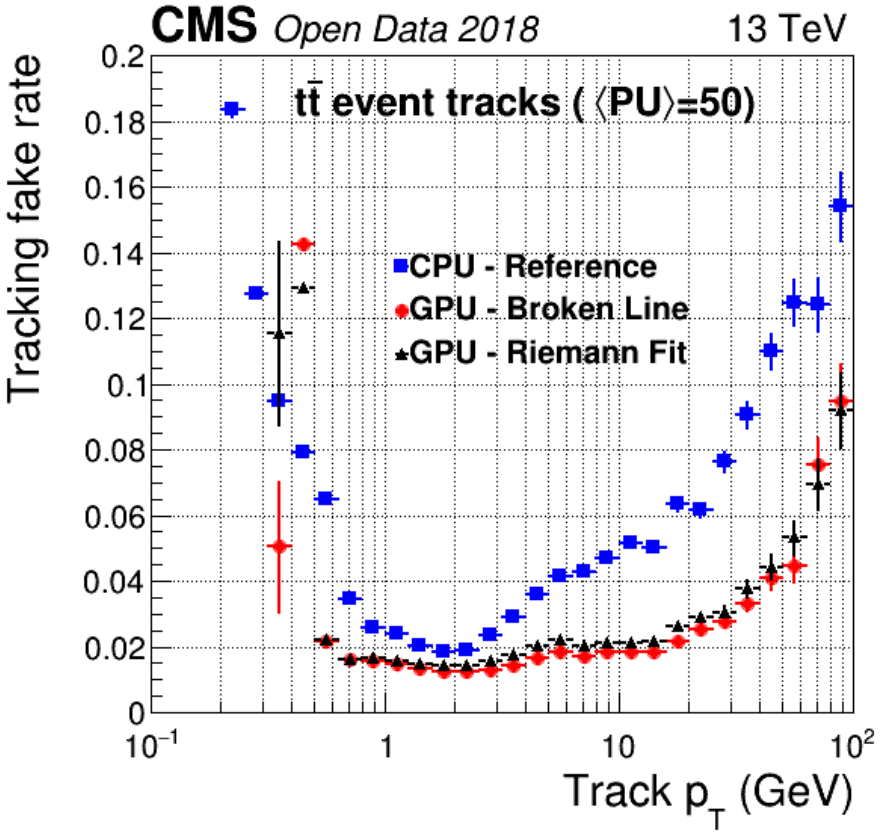




Track reconstruction duplicate rate as a function of reconstructed tracks η, p_T

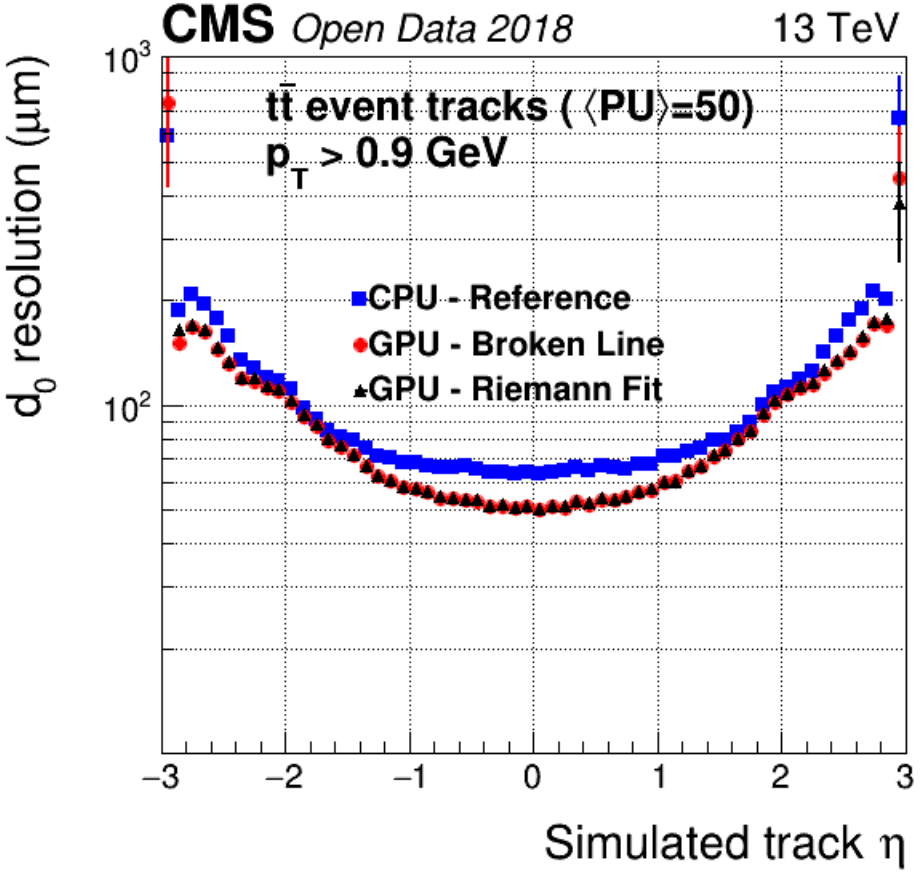
Physics performance – Fakes

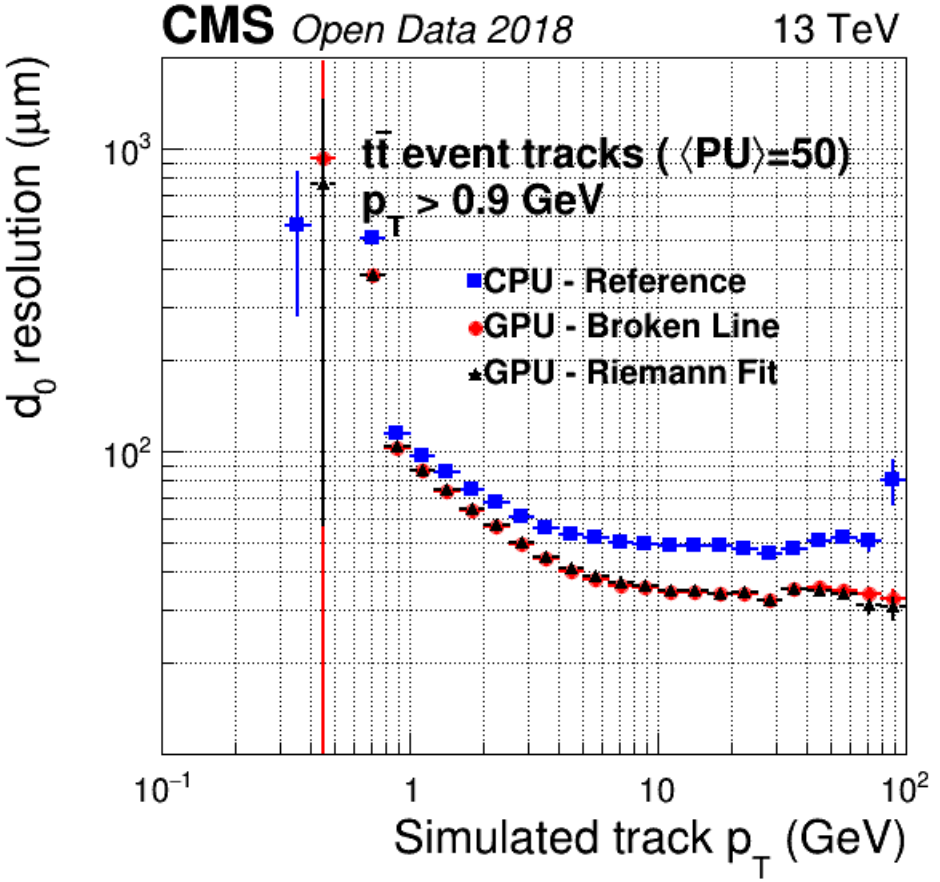




Track reconstruction fake rate as a function of reconstructed tracks η, p_T

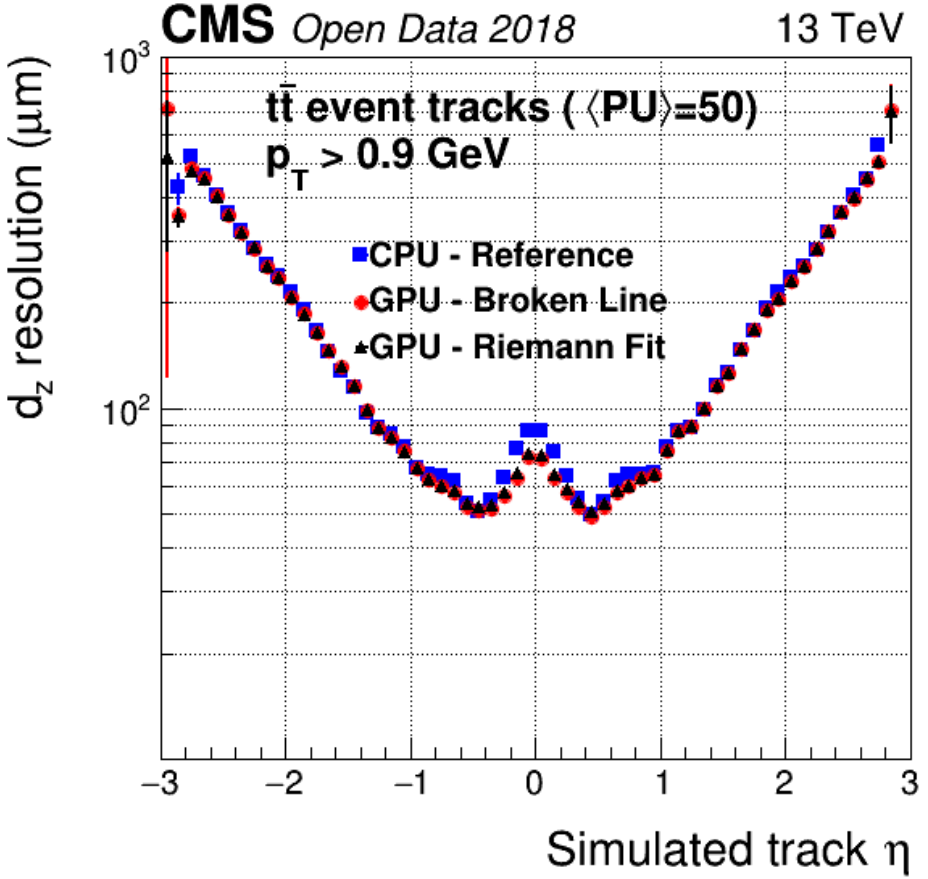
Physics Performance - Resolutions

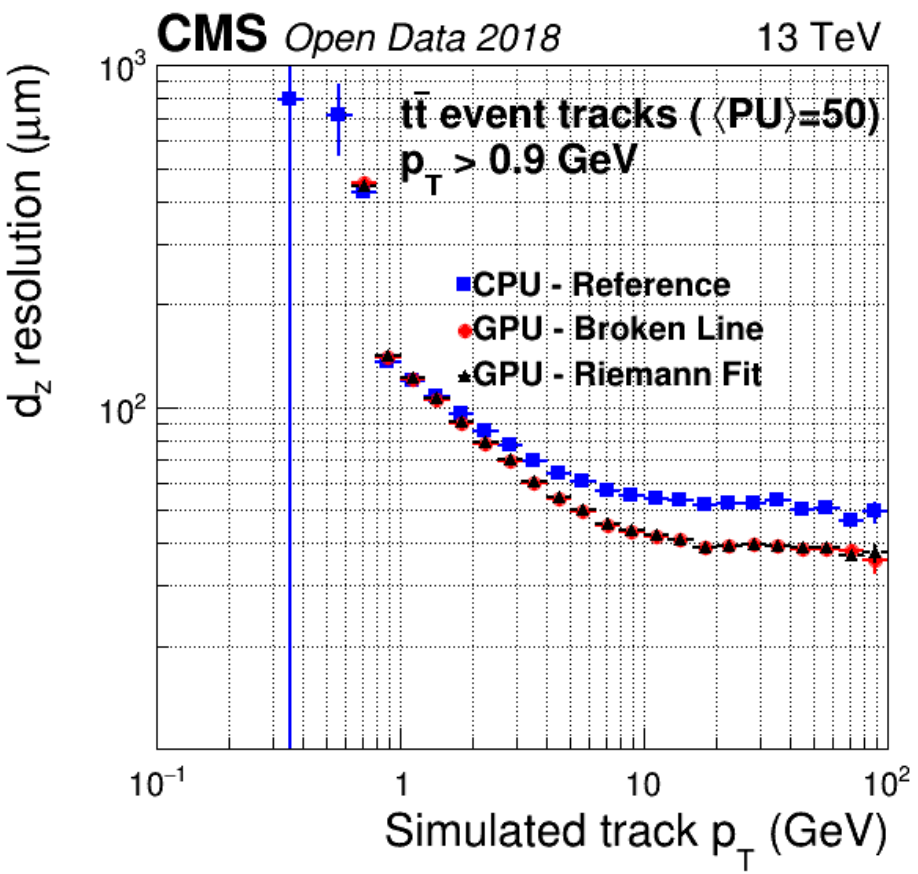




Track resolution of the transverse impact parameter as a function of simulated track η and p_T

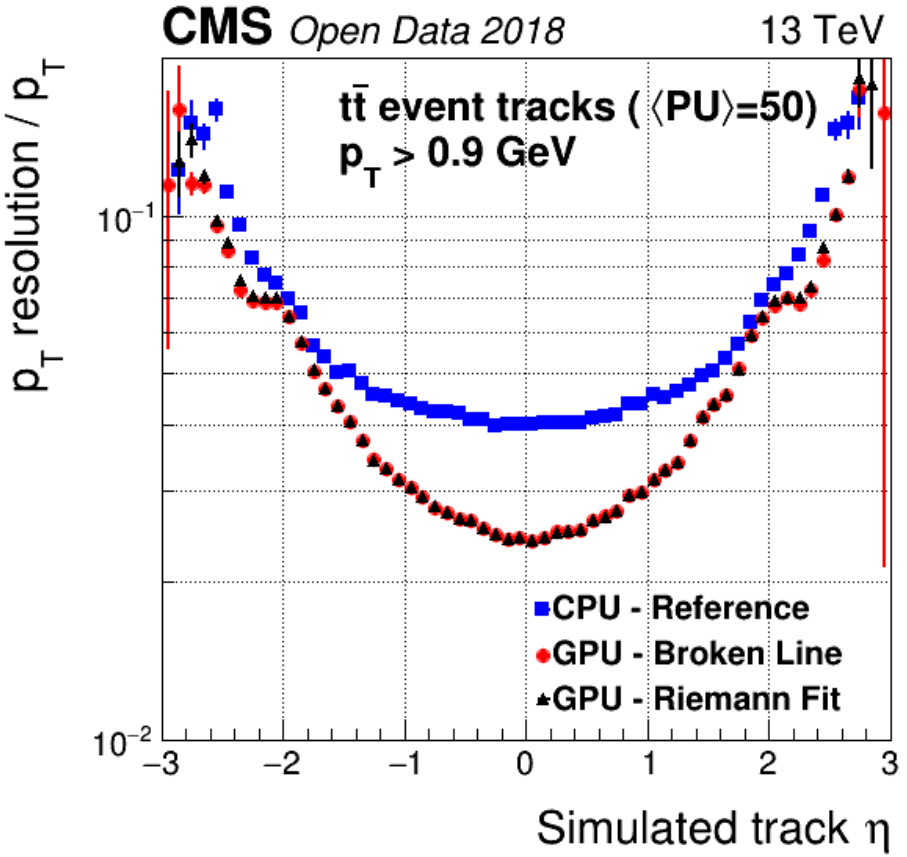
Physics Performance - Resolutions

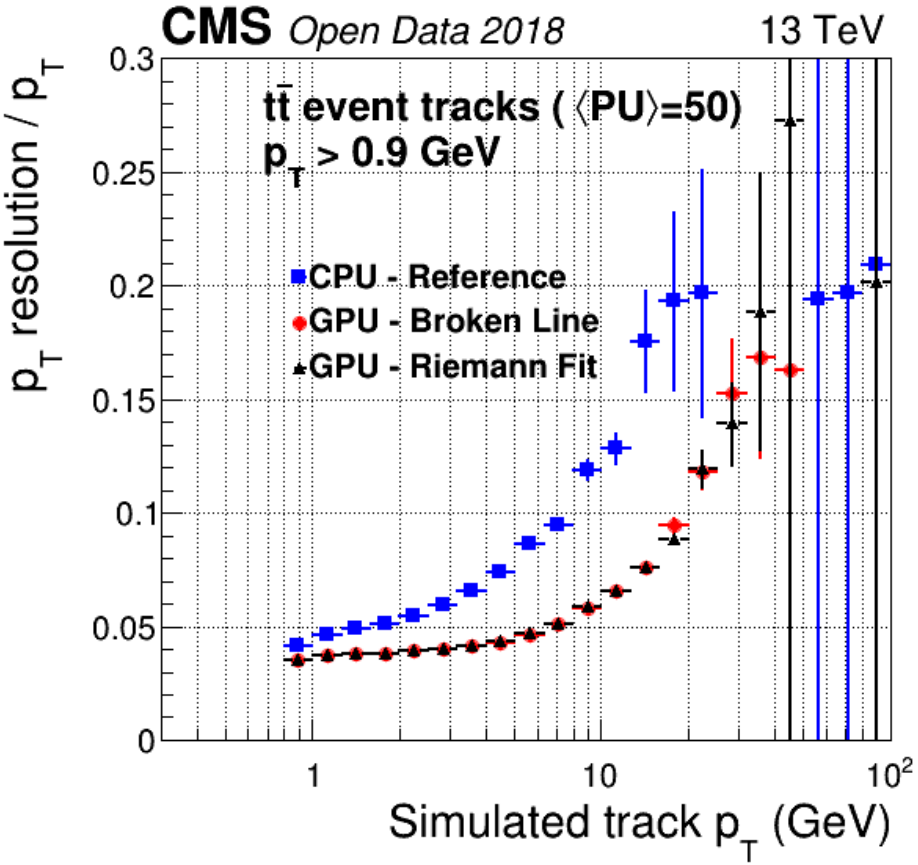




Track resolution of the longitudinal impact parameter as a function of simulated track η and p_T

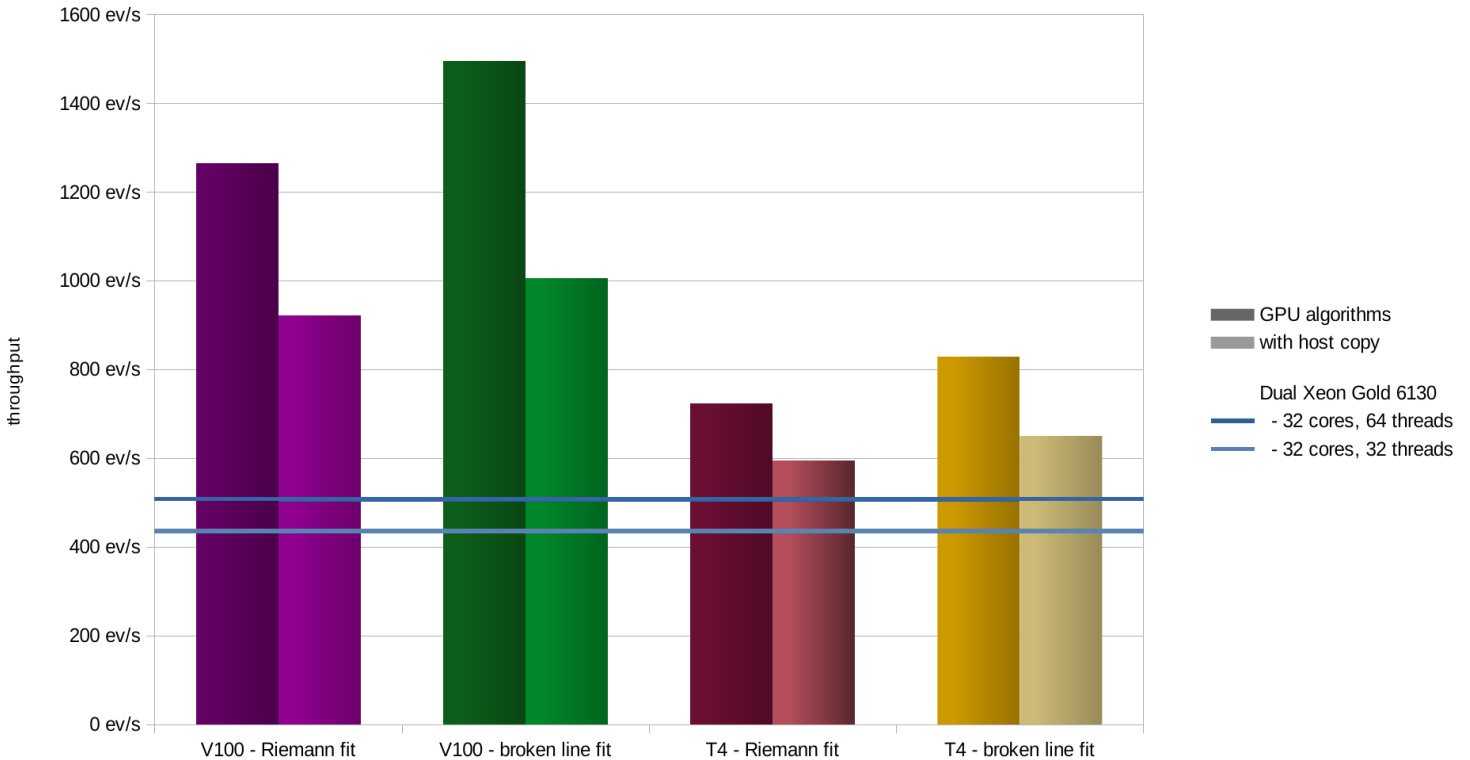
Physics Performance - Resolutions



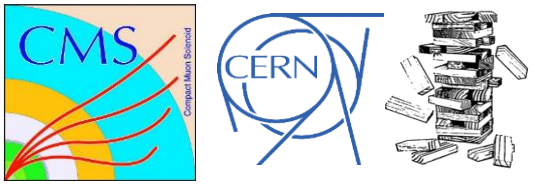


Track reconstruction resolution of p_T as a function of simulated track η and p_T

Computational Performance

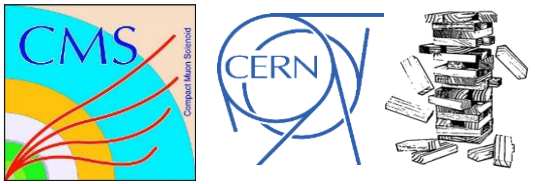


Pixel reconstruction consumers can either work directly on the GPU or ask for a copy of the tracks and vertices on the host



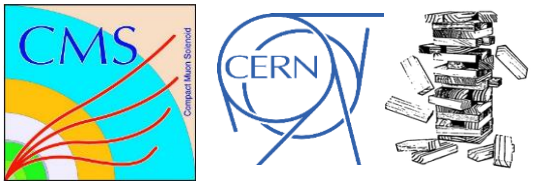
On Performance Portability

Why are we caring?



- The Patatrack team has demonstrated a complete CMS Pixel reconstruction running on GPU:
  - on a NVIDIA T4 can achieve 50% higher performance than a full Skylake Gold node
  - NVIDIA T4 costs approx. 1/5 of a node
  - It is fully integrated in CMSSW and supports standard validation
  - It is written in CUDA for the GPU part, C++ for the CPU part
- Maintaining and testing two codebases might not be the most sustainable solution in the medium/long term
  - Not a showstopper at the moment, but will become one when we will transfer ownership of the code to the collaboration
- In the long term other accelerators might appear

P != PP



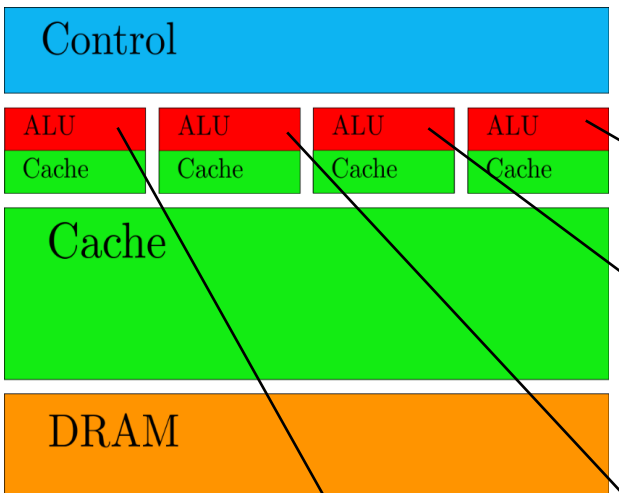
Portability could be achieved by blindly translating CUDA threads to, e.g., CPU threads or viceversa (plus some synchronization mechanism)

- You would not need to learn how a GPU works

Unfortunately, this is a terrible idea and will almost certainly lead you to poor performance

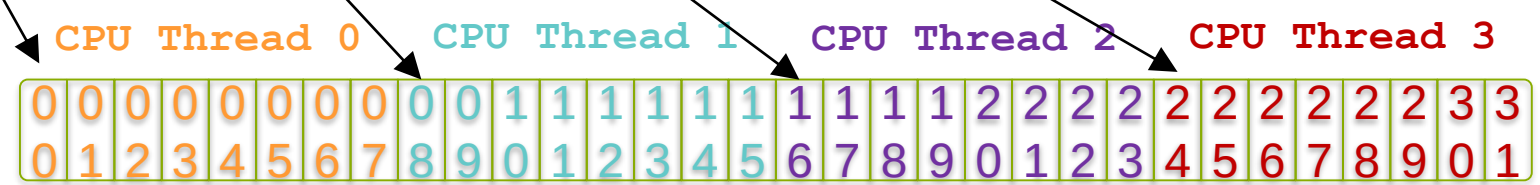
Portability does not imply Performance Portability

Memory access patterns: cached



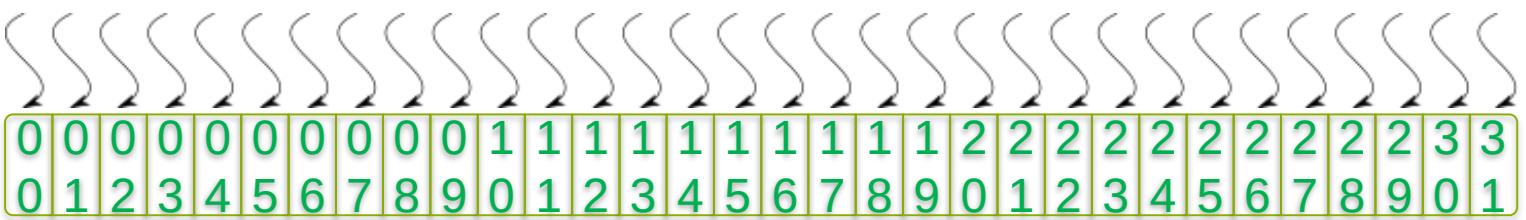
For optimal CPU cache utilization, the thread *a* should process element *i* and *i+1*

- stride=1

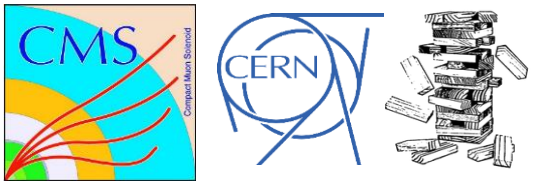


Inside a GPU SM: coalesced

- L1 data cache shared among ALUs
- ALUs work in SIMD mode in groups of 32 (warps)
- If a *load* is issued by each thread, they have to wait for all the loads in the same warp to complete before the next instruction can execute
- Coalesced memory access pattern optimal for GPUs: thread *a* should process element *i*, thread *a+1* the element and *i+1*
  - Lose an order of magnitude in performance if cached access pattern used on GPU



Portability frameworks



OpenMP and OpenACC

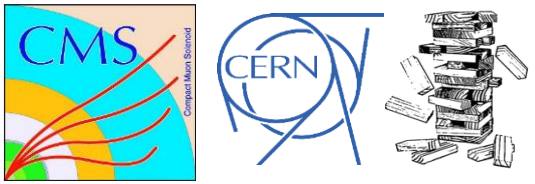
- Portability programming models based on compiler directives
- Sensitive to compiler support and maturity
- Difficult coexistence with a tbb-based framework-scheduler

OpenCL -> SYCL -> OneAPI

- Initially The promise for portability, then became framework for portability between GPUs from different vendors, now supporting FPGAs
- While OpenCL did not support the combination of C++ host code and accelerator code in a single source file, SYCL does
    - This is a precondition for templated kernels which are required for policy based generic programming
- SYCL enables the usage of a single C++ template function for host and device code
- At the moment, OneAPI is SYCL

For all the above, if you need portable performance you have to manage memory and its layout yourself

Performance Portability frameworks



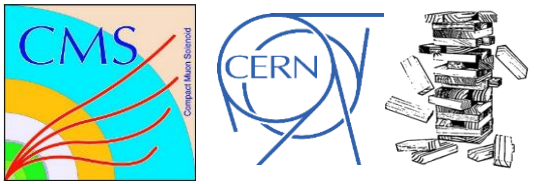
In the context of Patatrack R&D we have been recently looking into:

- Alpaka/Cupla: https://github.com/ComputationalRadiationPhysics/alpaka
  - Developed by Helmholtz-Zentrum Dresden – Rossendorf
    - Applications in Material science, XFEL, HPC
- Kokkos: https://github.com/kokkos/kokkos
  - Developed by Sandia National Lab, U.S. National Nuclear Security Administration

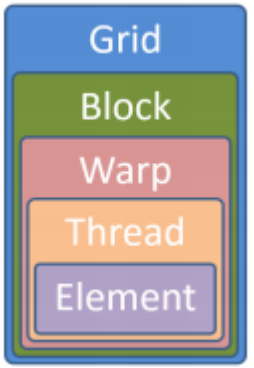
They provide an interface that hides the back-end implementation.

In the following, the assumption is that you already have a data-parallel code.

Alpaka abstraction hierarchy



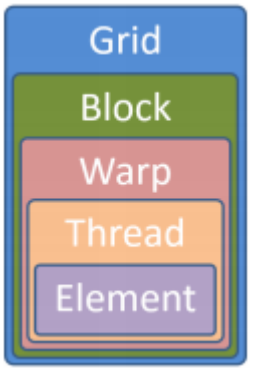
- multiple elements are processed per thread
- multiple threads are executed in lock-step within a warp
- multiple warps form independent blocks



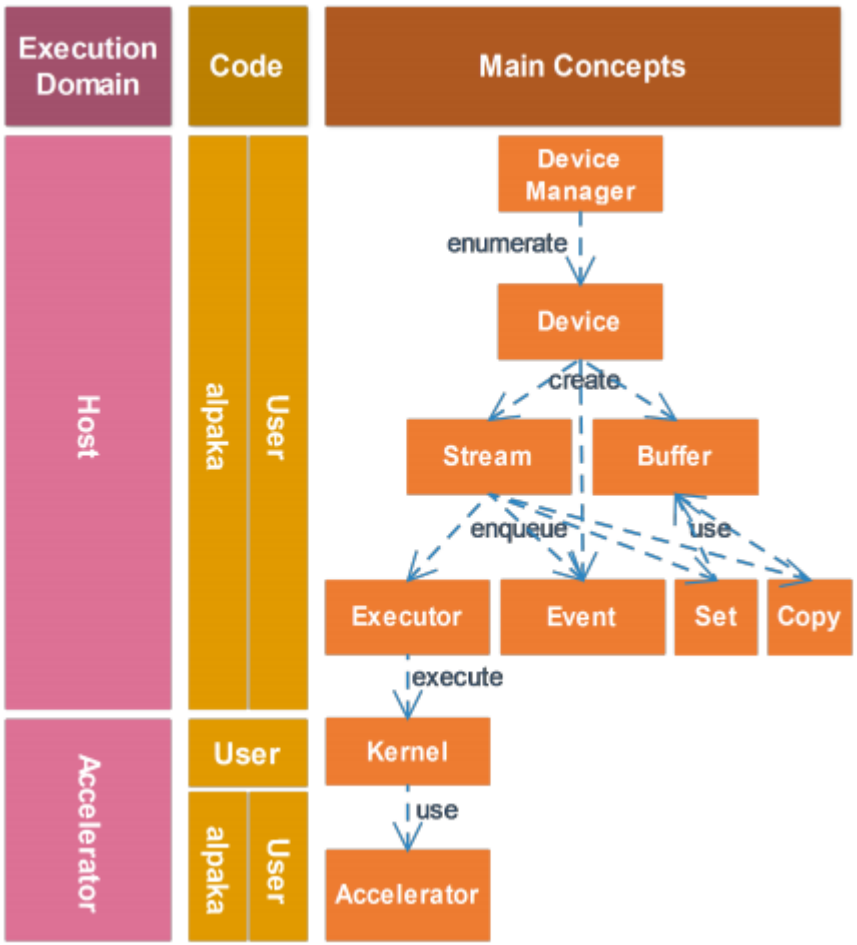
- Cupla was created because mapping the Alpaka's abstraction to CUDA is straightforward as the hierarchy levels are identical up to the element level.

Alpaka abstraction hierarchy to CPU

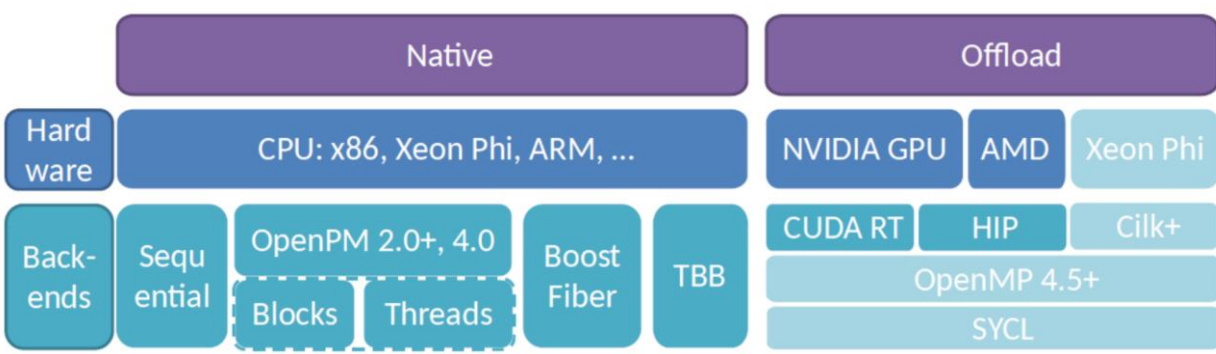
- On GPU, warps can handle branches with divergent control flows of multiple threads
  - There is no component on the CPU capable of this
  - 1to1 mapping of threads to warps
- Blocks cannot be mapped to the node nor socket
  - too much cache, memory, bus traffic
  - They are mapped to the cores
- Elements can be used to map CPU vector units



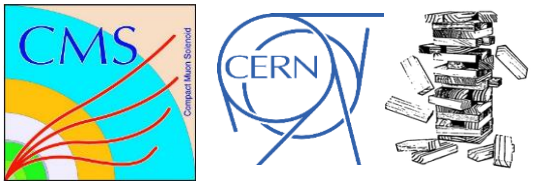
Alpaka/Cupla




```
128  struct kernel_compute_histogram {
129    template <typename T_Acc>
130    ALPAKA_FN_ACC
131    void operator()(T_Acc const &acc, LayerTilesCupla<T_Acc> *d_hist,
132        PointsPtr d_points, int numberOfPoints) const {
133      int i = blockIdx.x * blockDim.x + threadIdx.x;
134      if (i < numberOfPoints) {
135        // push index of points into tiles
136        d_hist[d_points.layer[i]].fill(d_points.x[i], d_points.y[i], i);
137      }
138    }
139  };
```



Kokkos



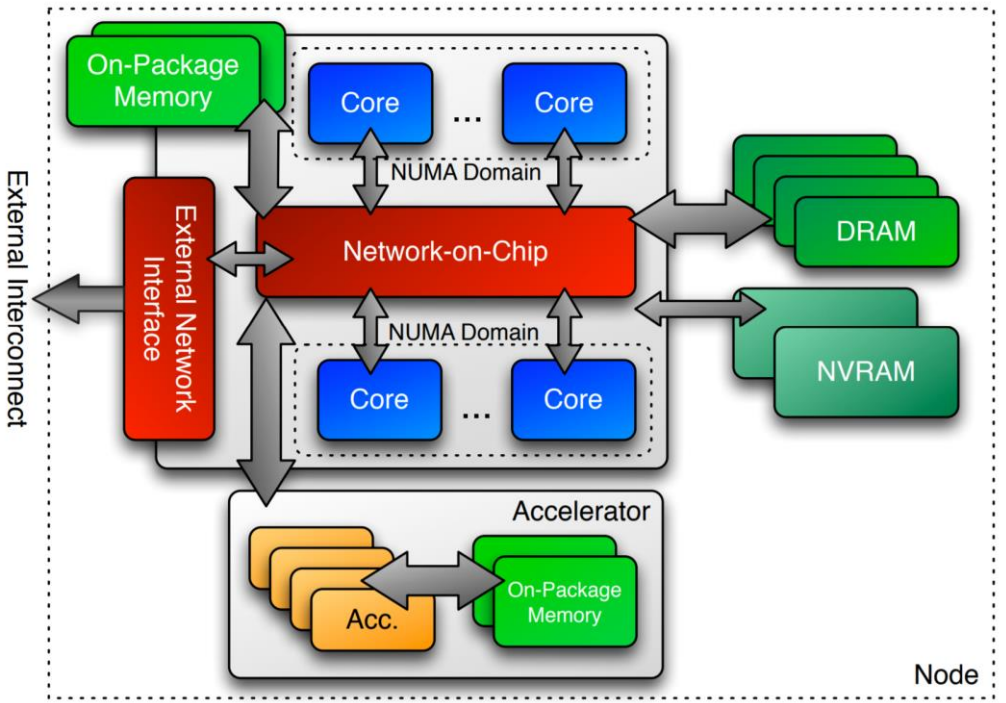
- Provides an abstract interface for portable, performant shared-memory programming

Supported backends:

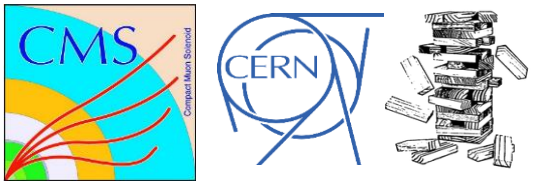
- std::threads, OpenMP, Intel tbb
- CUDA, ROCm
- Offers `parallel_for, parallel_reduce, parallel_scan, task` to describe the pattern of the parallel tasks
- Multidimensional arrays with a neutral indexing and an architecture dependent layout are available
- Thread-safety issues: the most portable approach is for only one (non-Kokkos) thread of execution to control Kokkos

Kokkos Machine Model

- Kokkos assumes an *abstract machine model* , in which multiple processing devices can coexist and might share memory space



Kokkos Execution Policy



An execution policy determines how the threads are executed:

- sizes of blocks of threads
- static, dynamic scheduling

Range Policy: execute an operation once for each element in a range

Team Policy: *teams* of threads form a *league*

- *sync and shared memory* in same team
- Different teams can run different execution patterns (parallel_for, scan etc)
- Policies can be nested

You decide where to run the parallel kernel by specifying an Execution Space

```
parallel_for(
  RangePolicy< ExecutionSpace >(0,numberOfIntervals),
  [=] (const size_t i) {
    /* ... body ... */
  });
```

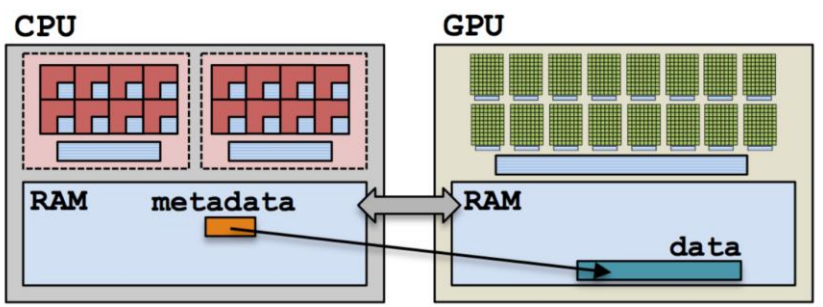
Kokkos Views

Multi-dimensional array of 0 or more dimensions, with sizes set at compile or run time

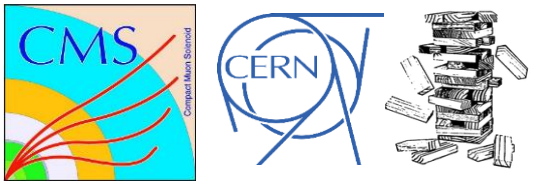
```
View<double ***, MemorySpace> data("label" , N0 , N1 , N2 ); 3 run, 0 compile
View<double **[N2], MemorySpace> data("label" , N0 , N1 ); 2 run, 1 compile
View<double *[N1][N2], MemorySpace> data("label" , N0 ); 1 run, 2 compile
View<double [N0][N1][N2], MemorySpace> data("label" ); 0 run, 3 compile
```

Specify MemorySpace to choose where to allocate the payload of the View

- HostSpace, CudaSpace, CudaUVMSpace…
- Mirroring/deep copy from one space to another possible
- Layout (row-/column-major) depends on the architecture for coalesced/cached memory access

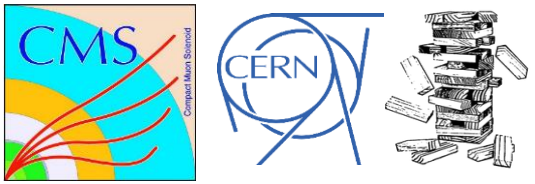


How Kokkos code looks like




```
Kokkos::View<Input, Kokkos::CudaSpace> input_d{"input_d"};
Kokkos::View<Input, Kokkos::CudaSpace>::HostMirror input_h = Kokkos::create_mirror_view(input_d);
std::memcpy(input_h.data(), &input, sizeof(Input));

Kokkos::View<Output, Kokkos::CudaSpace> output_d{"output_d"};
Kokkos::View<Output, Kokkos::CudaSpace>::HostMirror output_h = Kokkos::create_mirror_view(output_d);

auto start = std::chrono::high_resolution_clock::now();
Kokkos::deep_copy(input_d, input_h);

Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(0, input.wordCounter),
                     KOKKOS_LAMBDA (const size_t i) {
                       kokkos::rawtodigi(input_d, output_d, wordCounter,
                                         true, true, false, i);
  });
Kokkos::fence();
Kokkos::deep_copy(output_h, output_d);
Kokkos::fence();

auto stop = std::chrono::high_resolution_clock::now();
```

Conclusion



- Heterogeneous computing is a reality: better physics performance, better computational performance, better energy efficiency, lower cost
- Portable code is key for long-term maintainability, testability and support for new accelerator devices
  - Many possible solutions, not so many viable ones, even less production ready
  - Alpaka and Kokkos are very active teams and discussions/pull requests are ongoing
- Starting from a CUDA code makes life **much** easier