Experience with the alpaka performance portability library in the CMS software

CHEP 2024 – October 21st, 2024

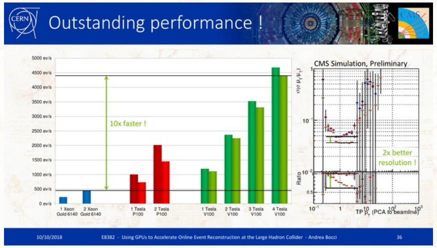
Andrea Bocci[1] for the CMS Collaboration

[1] CERN

version 1.0.3

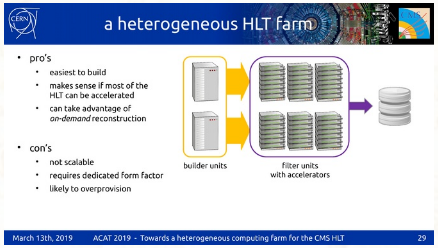
a brief history of GPUs at CMS

- 2016: first concrete interest in using (NVIDIA) GPUs for offloading reconstruction algorithms
- 2017: first CUDA code for Pixel local reconstruction
- 2018: continuous R&D activities
  - data structures, memory allocation strategies, caching and reuse
  - CUDA-based algorithms
- 2019: optimisations and debugging
  - more CUDA-based algorithms
  - first work on GPU-to-CPU code portability
- 2020: upstream integration
  - support for Run-3 and Phase-2 workflows
  - better integration with the HLT menu
  - automatic offloading to GPUs when available
- 2021: integration and adoption at HLT
- 2022: deployment in production
- 2023: migration to alpaka-based framework
  - improved data structures, automatic offloading
- 2024: alpaka-based framework and algorithms in production

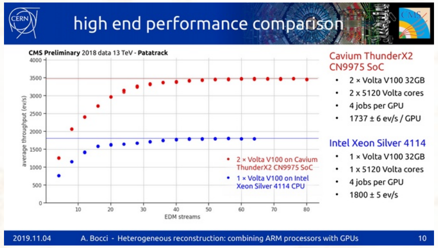


NVIDIA GTC (2018)

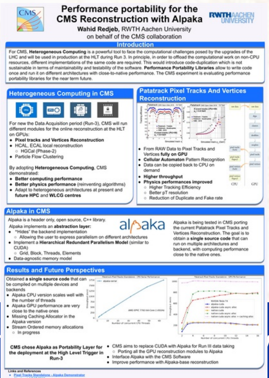
ACAT 2019



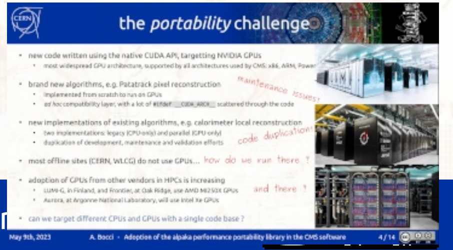
CHEP 2019



ACAT 2021



CHEP 2023



performance portability ?

OpenCL

OpenMP

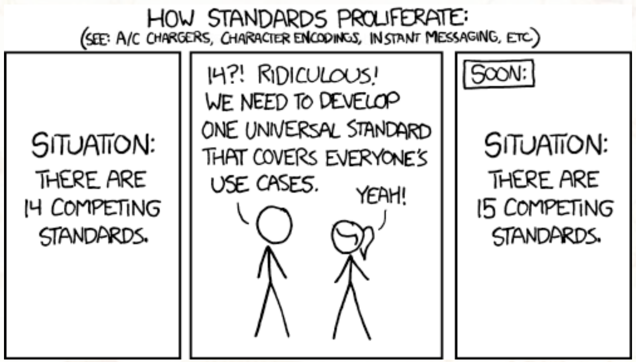
std::par

SYCL

OpenACC

Kokkos

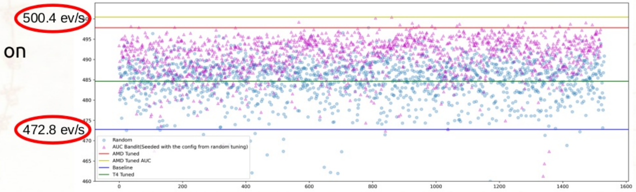
alpaka



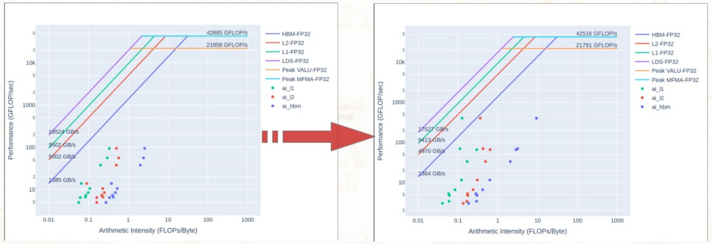
why *performance portability*

auto-tuining kernel parameters

- iterative approach
  - start from parameters "guessed" on NVIDIA hardware (V100 / T4)
  - first pass of auto-tuning using a random walk approach
  - second pass of auto-tuning using "bandit" approach

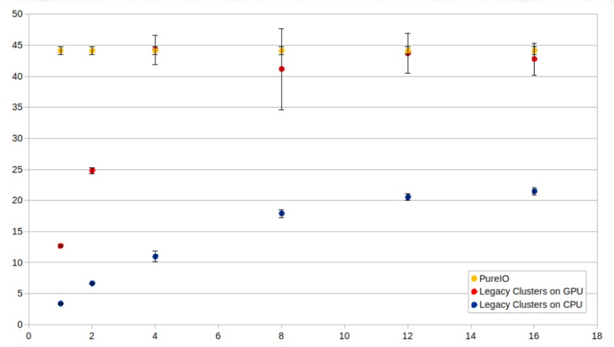


- 6% speed-up "for free" !



run transparently on different GPUs:
AMD MI250x at LUMI-G HPC

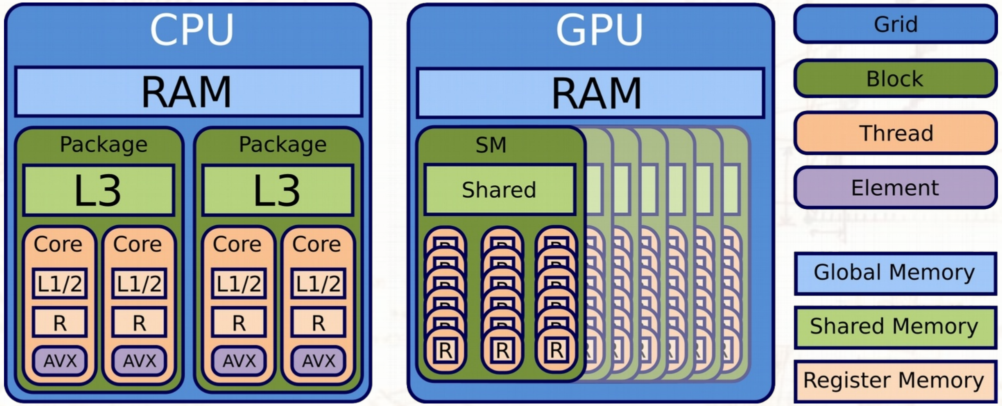
writing more kernels



- is now limited by the I/O
  - cannot read and decode the input data fast enough!

what is alpaka ?

- alpaka is a header-only C++17 abstraction library for heterogeneous software development
  - it aims to provide *performance portability* across accelerators through the abstraction of the underlying levels of parallelism
  - *may* expose the underlying details when necessary
  - (almost) *native* performance on different hardware



- supports all platforms of interest to CMS
  - x86 and ARM CPUs
    - with serial and parallel execution
  - stable support for NVIDIA and AMD GPUs
    - with CUDA and ROCm backends
  - experimental support for Intel GPUs and FPGAs, based on SYCL and oneAPI

- developed at CASUS at HZDR, and at CERN
  - open source project, easy to contribute to: https://github.com/alpaka-group/alpaka/

- it is production-ready today !

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

alpaka is under active development

- alpaka 1.0.0 released on November 2023
  - experimental support for Intel oneAPI, with SYCL Unified Shared Memory model
  - support `std::mdspan` and Kokkos' `mdspan`
- alpaka 1.1.0 released on January 2024 ← used by CMS for 2024 releases
  - stable support for Intel oneAPI
  - implement additional math functions and warp-level functions
- alpaka 1.2.0 just released on October 2024
  - more complete support for Intel oneAPI
  - introduce helpers for writing parallel kernels ← already used in CMS software
  - last release to support c++17, keep the 1.2.x branch for long term support
- looking ahead: plans for alpaka 2.0.0
  - move to c++20 and introduce Concepts
  - make more device-side operations `constexpr`
  - improve memory buffers and views
  - support grid-wide synchronisation
  - support CUDA graphs / HIP graphs / TBB flow graphs

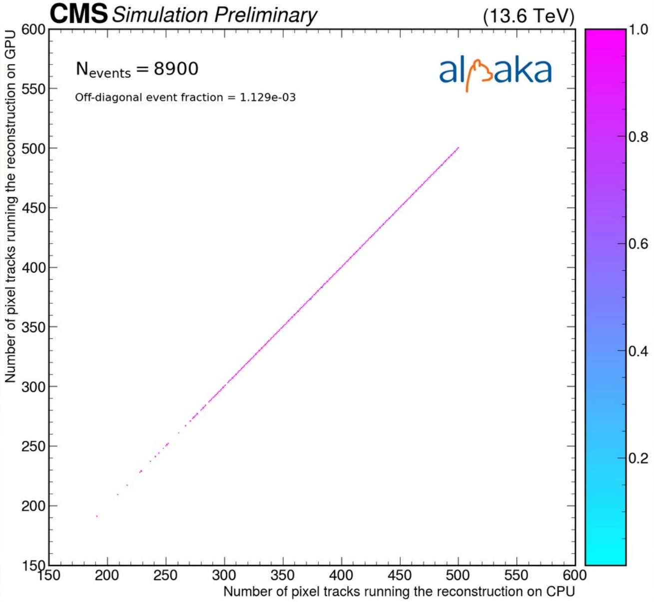
the alpaka migration

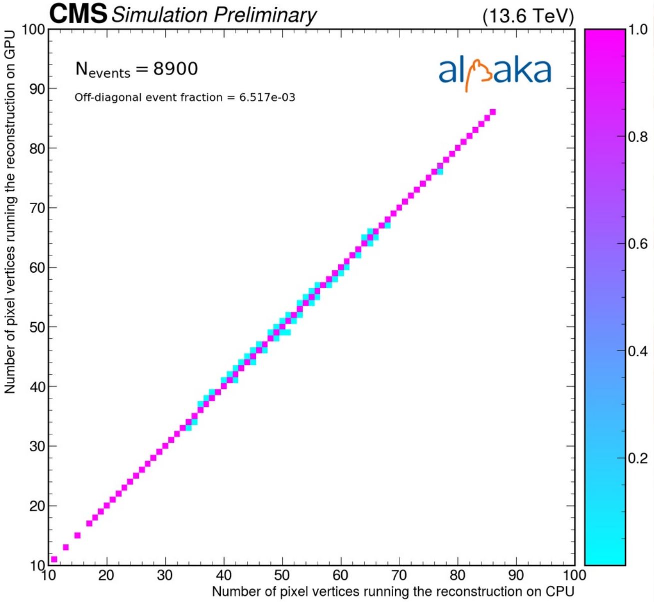
- adopt a *performance portability* library
  - reduce code duplication
- adopt a generic and consistent SoA approach for heterogeneous data structures
  - implement common optimisations and minimise memory operations
  - offer a common interface, and reduce the development and maintenance efforts
- adopt an improved version of the accelerator framework in CMSSW
  - automate data transfers from GPUs to host
  - support automatic selection of the “best” backend among the host and all available accelerators
- simplify the logic and the dependency among modules, reduce code duplication

benefits of alpaka as a portability library

- single code base targetting CPUs and GPUs
  - reduce code duplication and maintenance effort
  - implement a common interface to the data and algorithms
- modular builds
  - always build code to run on CPUs
  - build code to run on the GPUs as additional shared libraries, only if supported by the architecture
    - *e.g.* no HIP/ROCm on ARM, no CUDA on RISC-V
    - developers can enable only available backends to speed up local builds
  - load GPU-based libraries at runtime only if they are present on the machine
    - match available hardware to the environment and to the job's configuration
- uniform algorithms and data structures
  - framework can automatically schedule tasks on the CPU or on the GPUs
  - framework can automatically schedule copies (to and) from the GPUs

single code running on CPUs and GPUs





Note: each column is normalised to unity

- uniform algorithms and data structures
  - framework can automatically schedule tasks on the CPU or on the GPUs
  - framework can automatically schedule copies (to and) from the GPUs

Structures of Arrays

DataFormats/ParticleFlowReco/interface/PFRecHitSoA.h

```
using PFRecHitsNeighbours = Eigen::Matrix<int32_t, 8, 1>;

GENERATE_SOA_LAYOUT(PFRecHitSoALayout,
    SOA_COLUMN(uint32_t, detId),
    SOA_COLUMN(float, energy),
    SOA_COLUMN(float, time),
    SOA_COLUMN(int, depth),
    SOA_COLUMN(PFLayer::Layer, layer),
    SOA_EIGEN_COLUMN(PFRecHitsNeighbours, neighbours),
    SOA_COLUMN(float, x),
    SOA_COLUMN(float, y),
    SOA_COLUMN(float, z),
    SOA_SCALAR(uint32_t, size)
)

using PFRecHitSoA = PFRecHitSoALayout<>;
```

syntax similar to a struct

DataFormats/ParticleFlowReco/interface/PFRecHitHostCollection.h

```
using PFRecHitHostCollection =
    PortableHostCollection<PFRecHitSoA>;
```

use on CPU ...

.../ParticleFlowReco/interface/alpaka/PFRecHitDeviceCollection.h

```
namespace ALPAKA_ACCELERATOR_NAMESPACE {
  using PFRecHitDeviceCollection =
    PortableCollection<::reco::PFRecHitSoA>;
}
```

... on GPU ...

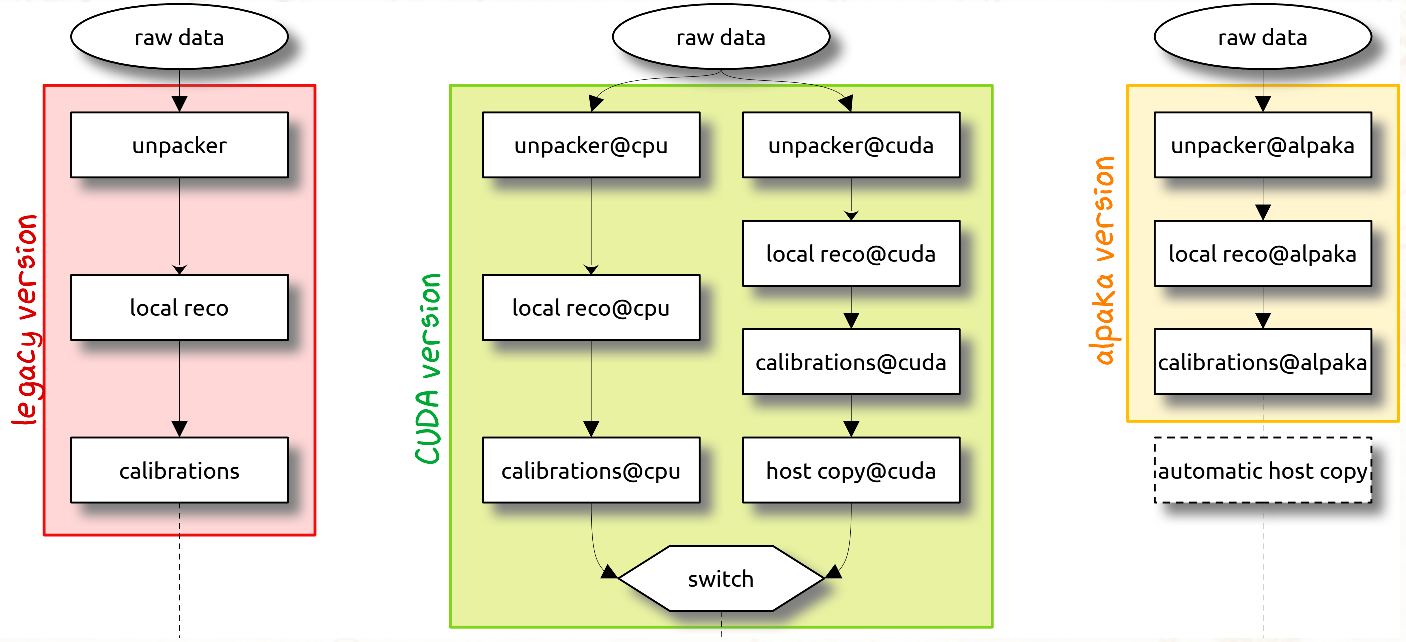
DataFormats/ParticleFlowReco/src/classes_serial.cc

```
SET_PORTABLEHOSTCOLLECTION_READ_RULES(
    PFRecHitHostCollection);
```

... in ROOT files

- uniform algorithms and data structures
  - framework can automatically schedule tasks on the CPU or on the GPUs
  - framework can automatically schedule copies (to and) from the GPUs

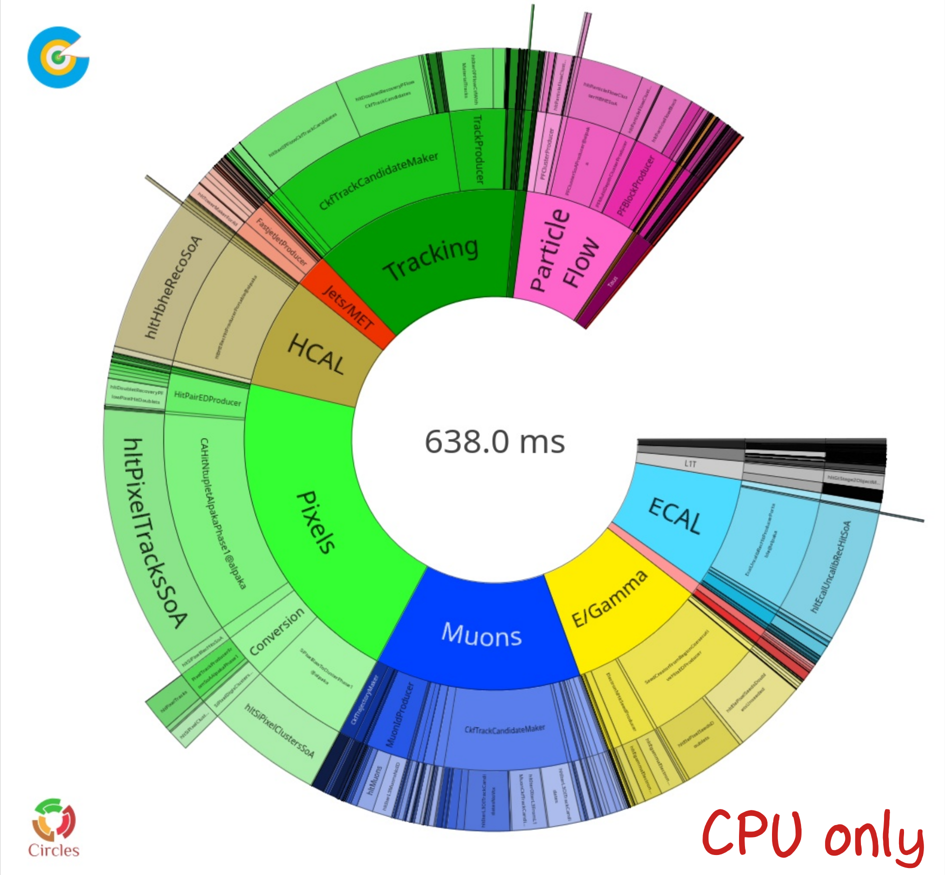
improved heterogeneous framework



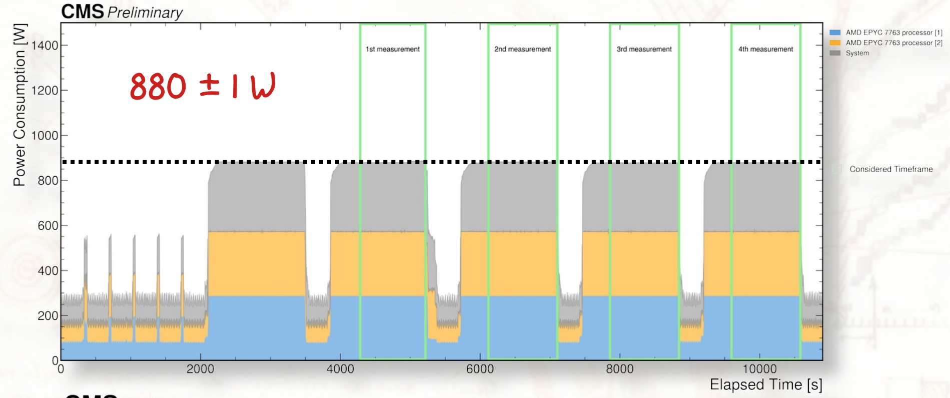
- uniform algorithms and data structures
  - framework can automatically schedule tasks on the CPU or on the GPUs
  - framework can automatically schedule copies (to and) from the GPUs

impact on the HLT farm

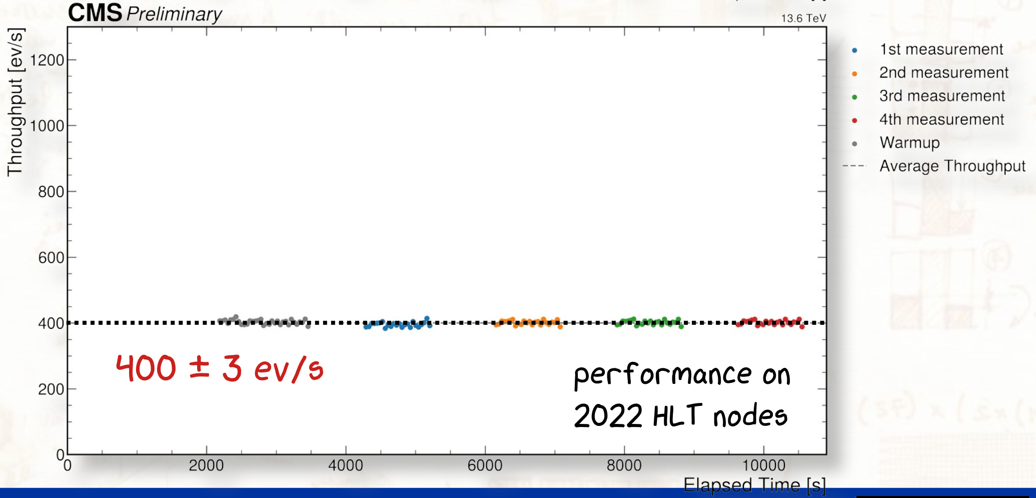
the HLT reconstruction with CPUs



CPU only



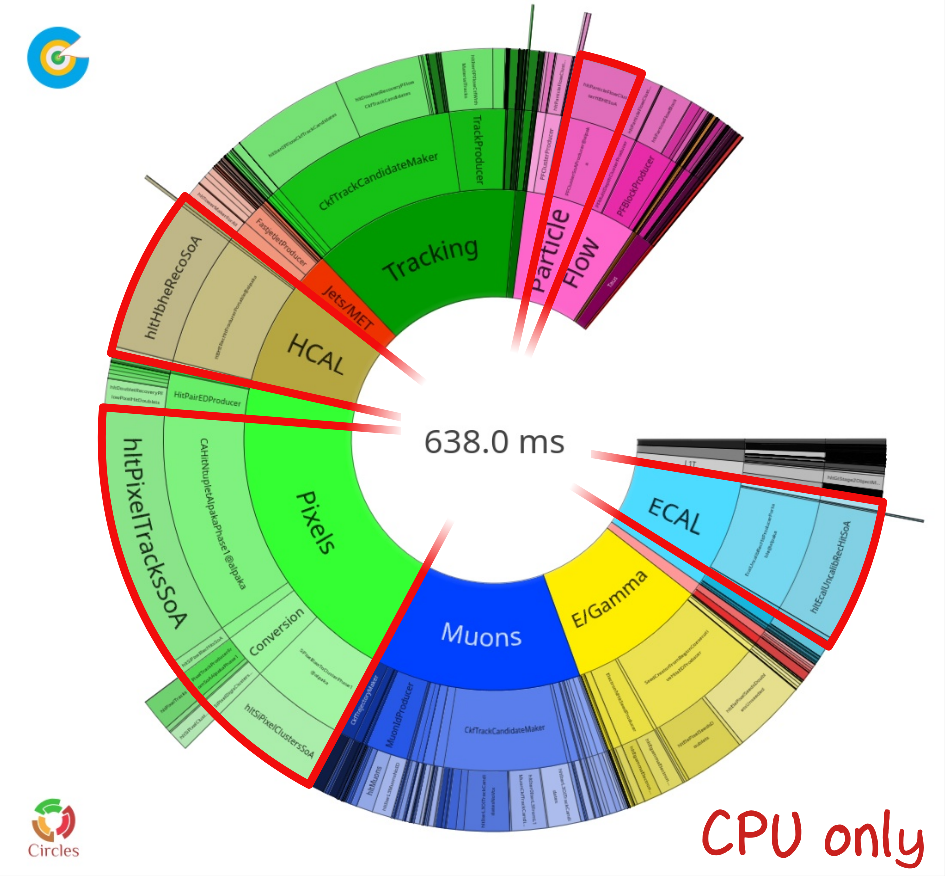
880 ± 1 W



400 ± 3 ev/s

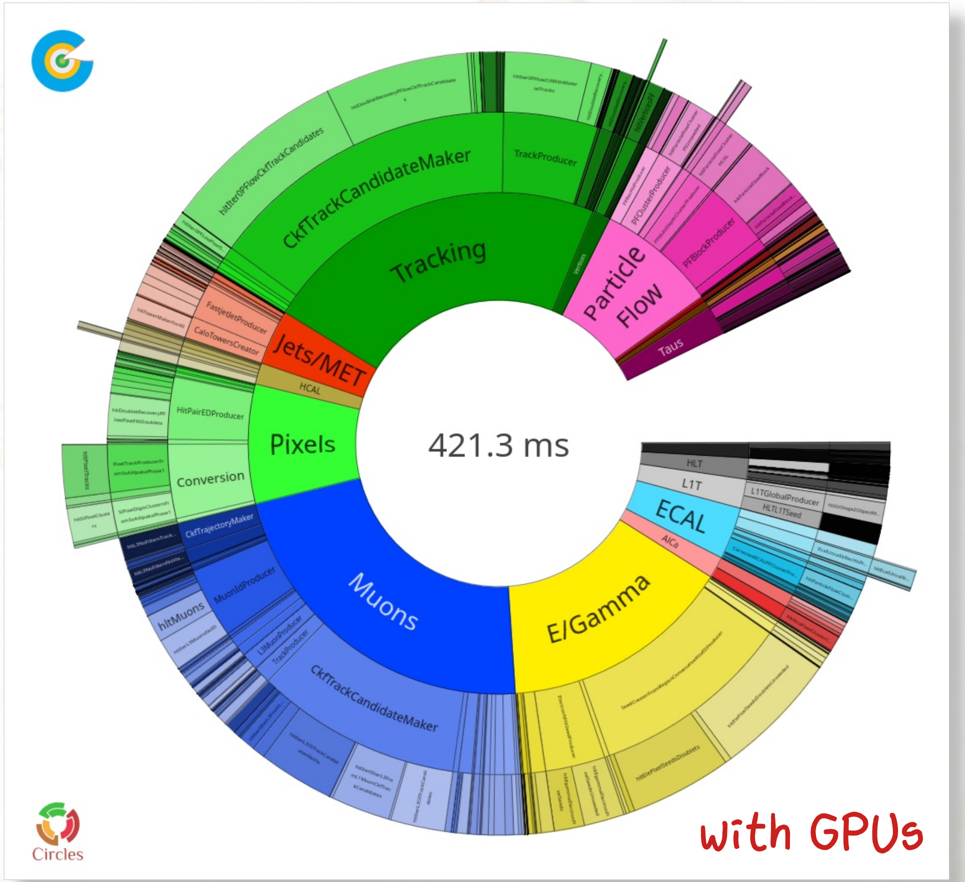
performance on 2022 HLT nodes

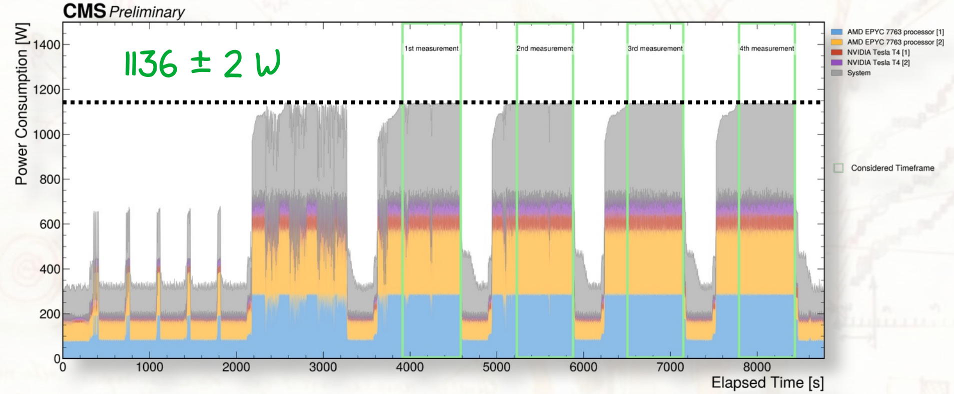
port time-consuming algortihms to GPUs

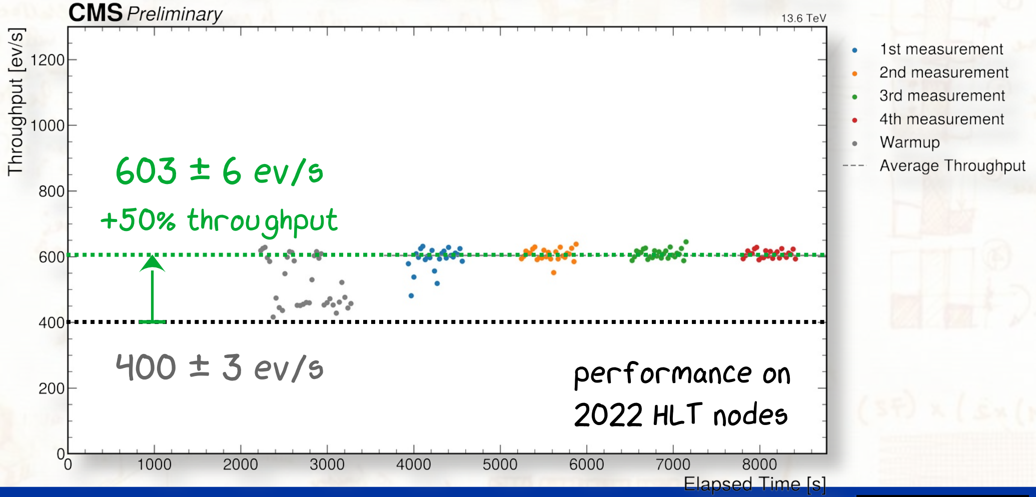


- HLT algorithms running on GPUs in 2024
  - pixel local reconstruction
  - ntuplet reconstruction, tracks and vertex fitting
    - see the talk by Daniele about the offline validation
  - ECAL unpacking and local reconstruction
  - HCAL local reconstruction
    - see the poster by Martin
  - HCAL Particle Flow clustering
    - see the poster by Jonathan
- GPU implementation under development
  - ECAL local calibrations
  - electron seeding
    - see the talk by Charis on Wednesday
  - full primary vertex reconstrution

the HLT reconstruction with GPUs



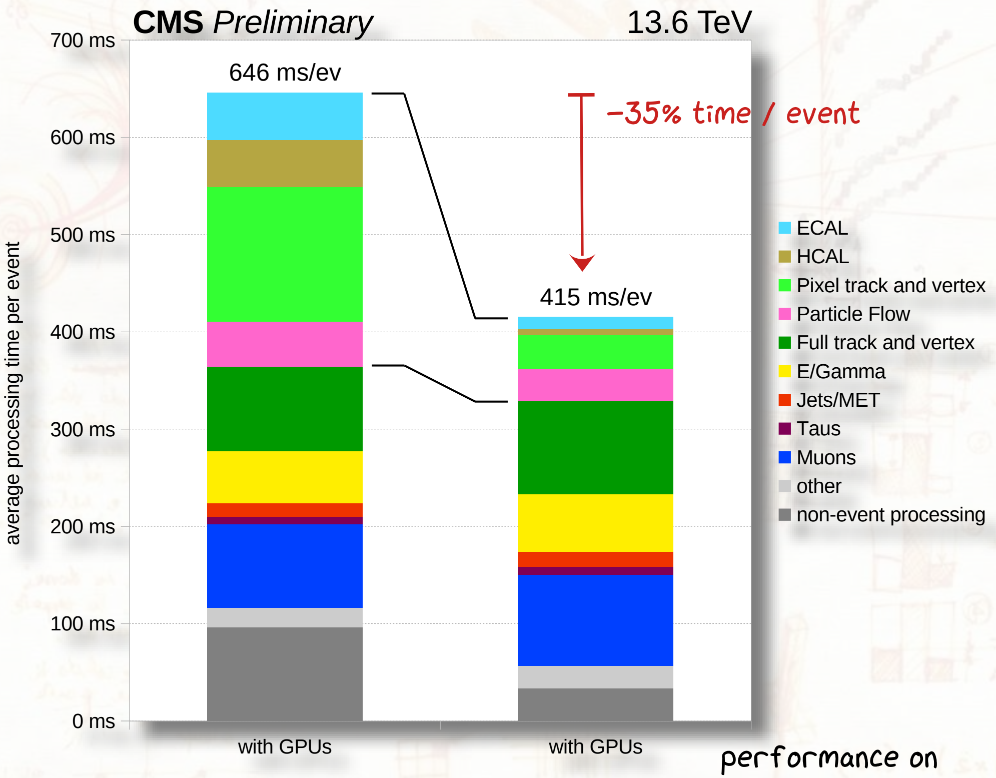




impact on the HLT farm

CMS Run 3 GPU-equipped HLT farm

- 200 nodes:
  - 2 × AMD EPYC "Milan" 7763 processors
  - 2 × NVIDIA Tesla T4 GPUs
- +20% extention in 2024 with 18 nodes:
  - 2 × AMD EPYC "Bergamo" 9754 processors
  - 3 × NVIDIA L4 GPUs
- thanks to the use of GPUs
  - 50% better event processing throughput
  - 35% less processing time per event
  - 15% - 20% better performance at initial cost
  - 15% - 25% better performance per kW



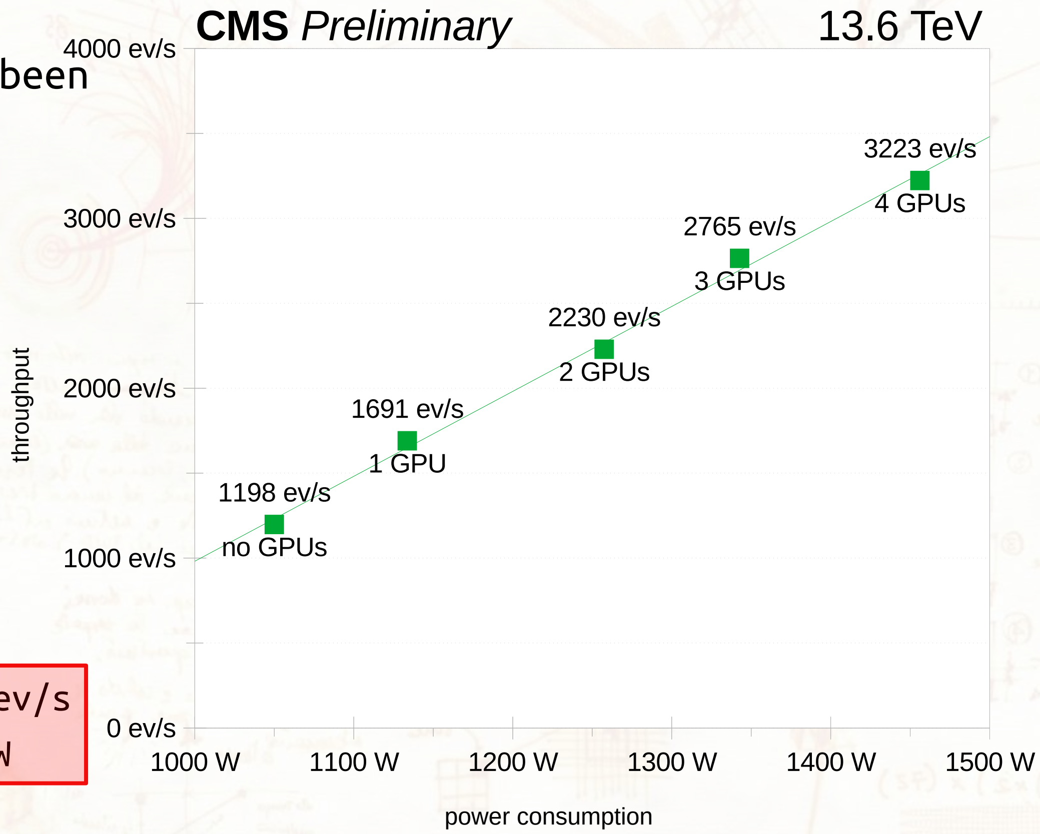
performance on 2022 HLT nodes

looking ahead

how well can GPU usage scale ?

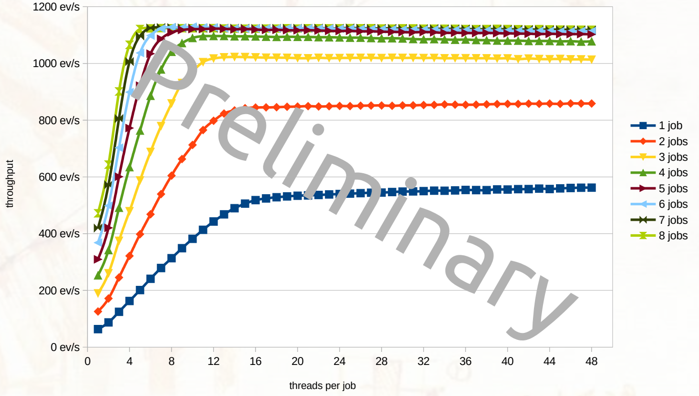
- ideal test case
  - consider only the fraction of the HLT that has been ported to alpaka
  - can run on CPUs (x86, ARM)
  - can run almost entirely on GPUs
- baseline
  - performance of a CPU-only setup
  - 2 × AMD EPYC "Bergamo" 9754 processors
- scaling
  - add 1×, 2×, 3×, 4× NVIDIA L4 GPUs
  - subtract the baseline
- results

			each GPU		baseline
throughput:	N_{GPUs}	×	512.3 ev/s	+	1196.5 ev/s
power draw:	N_{GPUs}	×	102.1 W	+	1043.8 W

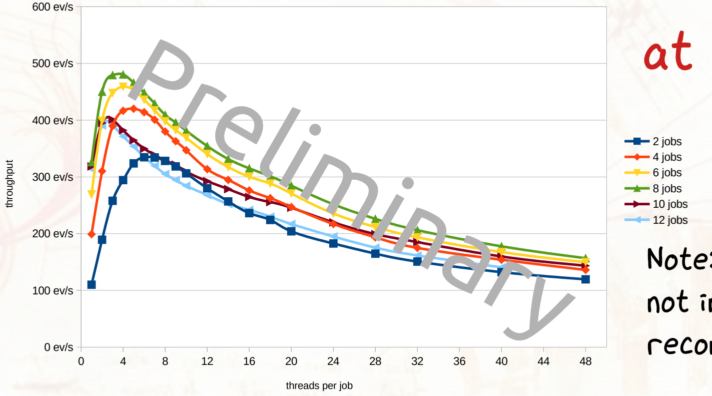


what about oher architectures ?

NVIDIA L40S GPU

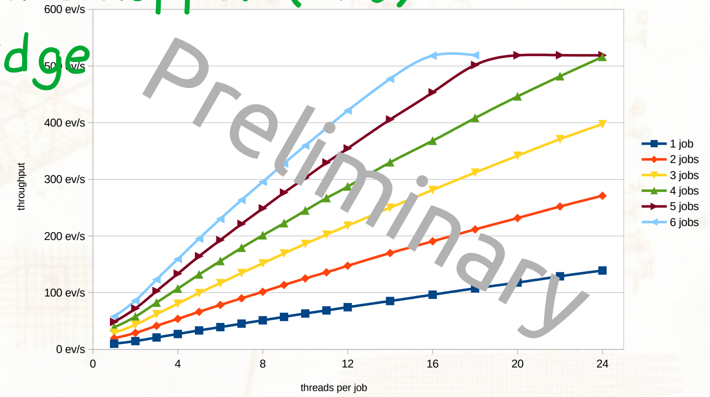


full AMD Instinct MI250X GPU at LUMI HPC

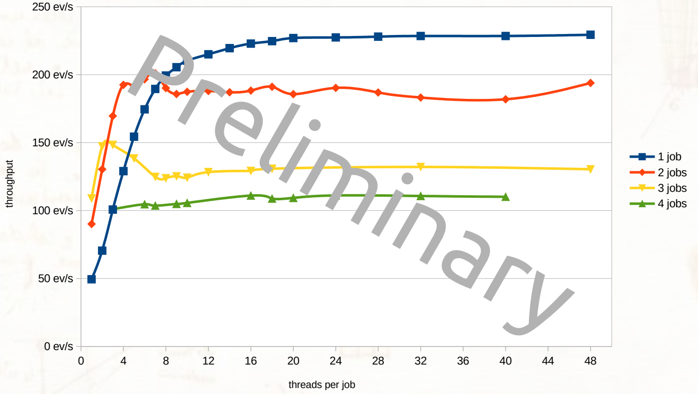


Note: MI250X does not include the pixel reconstruction

NVIDIA Grace Hopper (GPU) at Oak Ridge



AMD Radeon Pro W7800 GPU



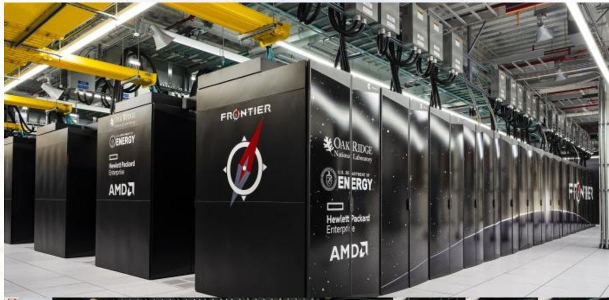
conclusions

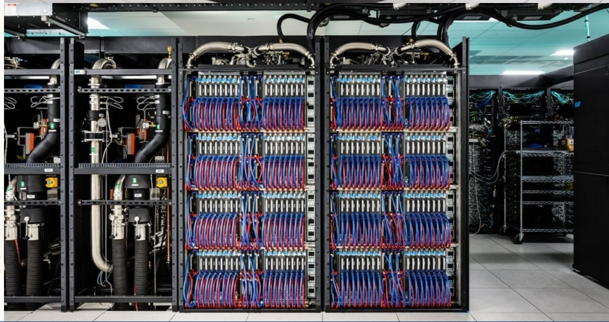
conclusions

- lessons learned
  - writing new reconstruction algorithms takes effort
    - whether they run only on CPU or on heterogeneous hardware
  - code duplication is Bad™
    - duplicate effort to add the same features and fix the same bugs
    - introduce more bugs
  - a portability framework
    can help minimise these efforts

alpaka

- looking ahead
  - GPUs can achieve impressive performance
    - if used for a large enough fraction of the algorithms
  - optimising the performance of heterogenous hardware is complicated!
  - need to gain more experience with non-NVIDIA hardware





Questions ?

the *portability* challenge

- new code written using the native CUDA API, targetting NVIDIA GPUs
  - most widespread GPU architecture, supports x86 and ARM
  - no RISC V yet ?
- develop new algorithms to run on GPUs
  - *ad hoc* compatibility layer
  - a lot of `#ifdef __CUDA_ARCH__` scattered through the code

maintenance issues!

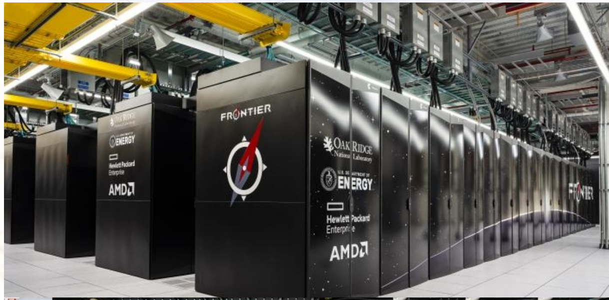
- port existing algorithms to run on GPUs
  - two implementations: legacy (CPU-only) and parallel (GPU-only)
  - duplication of development, maintenance and validation efforts

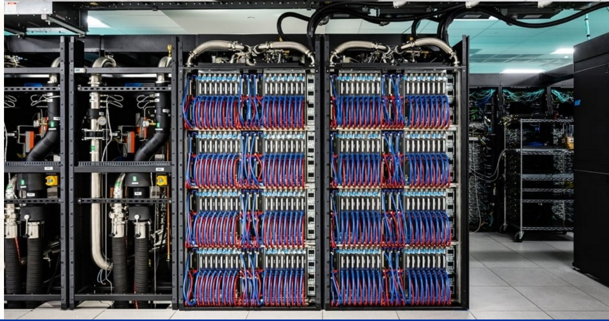
code duplication!

- most offline sites do not use GPUs yet… how do we run here ?
- adoption of GPUs from other vendors in HPCs is increasing
  - LUMI-G, in Finland, and Frontier, at Oak Ridge, use AMD MI250X GPUs
  - Aurora, at Argonne National Laboratory, will use Intel Xe GPUs

and here ?

- can we target different CPUs and GPUs with a single code base ?





alpaka core concepts

`Platform` and `Device`

- identify the type of hardware (*e.g.* NVIDIA GPUs) and individual devices (*e.g.* each single GPU) present on the machine
- the `DevCpu` device serves two purposes:
  - as the "host" device, for managing the data flow (*e.g.* perform memory allocation and transfers, run EDProducer, etc.)
  - as an "accelerator" device, for running heterogeneous code (*e.g.* to run an algorithm on the CPU)
- platforms and devices should be created at the start of the program and used consistently

owning `Buffer` and non-owning `View`

- point to a scalar or a N-dimensional array in host or device memory
- scalars and 1-dimensional arrays can be accessed with the pointer `*`, `->` and array `[]` operators
- on device that support it, the buffer allocations/deallocations can use a queue-ordered semantic

nota bene: all Alpaka objects behave like `shared_ptr`s, and should be passed by value or by `const&`

alpaka core concepts

Queues and Events

- queues identify a work queue where tasks (memory ops, kernel executions, …) are executed in order
  - for example, a queue could represent an underlying CUDA stream or a CPU thread
- queues can be sync(hronous or blocking) or async(hronous or non-blocking)
  - work submitted to a sync queue is executed immediately, before returning to the caller
  - work submitted to an async queue is executed in the background, without waiting for its completion
- events identify points in time along the work queue
  - can be used to query or wait for the readiness of a task submitted to a queue
- queues and events are always associated to a specific device

Tags and Accelerators

- tags describe all possible accelerators
- accelerators encapsulate the execution policy on a specific device
  - N-dimensional work division (1D, 2D, 3D, …)
  - on CPU: serial vs parallel execution of the “blocks” (single thread, multi-threads, TBB tasks, …)
- accelerators are created any time a kernel is executed, and can be used in device code to extract the execution configuration

alpaka in CMSSW: backends

- in CMSSW we tie together the `Device`, `Queue`, `Event` and `Accelerator` types in a "backend"
- each backend is associated to a namespace
  - synchronous execution on the CPU, with a single thread:

```
namespace alpaka_serial_sync {
    using Platform = alpaka::PlatformCpu;
    using Device = alpaka::DevCpu;
    using Queue = alpaka::QueueCpuBlocking;
    using Event = alpaka::EventCpu;
    template <typename TDim> using Acc = alpaka::AccCpuSerial<TDim, uint32_t>;
}
```

  - asynchronous execution on a GPU, with a grid of blocks and threads:

```
namespace alpaka_cuda_async {
  using Platform = alpaka::PlatformCudaRt;
  using Device = alpaka::DevCudaRt;
  using Queue = alpaka::QueueCudaRtNonBlocking;
  using Event = alpaka::EventCudaRt;
  template <typename TDim> using Acc = alpaka::AccGpuCudaRt<TDim, uint32_t>;
}
```

files and directory structure

- to support the compilation of alpaka-based plugins and libraries for multiple backends, we have introduced a new directory structure ad a new file type:
  - `alpaka/` subdirectories under `interface/`, `src/`, `plugins/` or `test/`
  - `*.dev.cc` files

```
DataFormats/PortableTestObjects/
├── BuildFile.xml
├── README.md
├── interface/
│   ├── TestHostCollection.h
│   └── TestSoA.h
│
└── src/
    │
    │
    ├── classes.h
    └── classes_def.xml
```

```
HeterogeneousCore/AlpakaTest/
├── plugins/
│   ├── BuildFile.xml
│   └── TestAlpakaAnalyzer.cc
│
└── test/
    ├── BuildFile.xml
    ├── reader.py
    ├── testHeterogeneousCoreAlpakaTestWriteRead.sh
    └── writer.py
```

files and directory structure

- to support the compilation of alpaka-based plugins and libraries for multiple backends, we have introduced a new directory structure ad a new file type:
  - `alpaka/` subdirectories under `interface/`, `src/`, `plugins/` or `test/`
  - `*.dev.cc` files

```
DataFormats/PortableTestObjects/
├── BuildFile.xml
├── README.md
├── interface/
│   ├── TestHostCollection.h
│   ├── TestSoA.h
│   └── alpaka/
│       └── TestDeviceCollection.h
└── src/
    ├── alpaka/
    │   ├── classes_cuda.h
    │   └── classes_cuda_def.xml
    ├── classes.h
    └── classes_def.xml
```

```
HeterogeneousCore/AlpakaTest/
├── plugins/
│   ├── BuildFile.xml
│   ├── TestAlpakaAnalyzer.cc
│   └── alpaka/
│       ├── TestAlgo.dev.cc
│       ├── TestAlgo.h
│       ├── TestAlpakaProducer.cc
│       └── TestAlpakaTranscriber.cc
└── test/
    ├── BuildFile.xml
    ├── reader.py
    ├── testHeterogeneousCoreAlpakaTestWriteRead.sh
    └── writer.py
```

alpaka/ directories

- all code under the `…/{src,plugins,test}/alpaka/` directories is compiled multiple times
  - into a separate shared library for each back-end
    - isolate compile-time and run-time dependencies, minimise code loaded at runtime
  - defining the `ALPAKA_ACCELERATOR_NAMESPACE` **macro** to the corresponding backend **namespace**
    - automate using the correct types, avoid symbol clashes

`*.cc` files by the *host compiler*

- for example, **`gcc 12.3`**
- what is available:
  - standard C++, *e.g.* ROOT and CMSSW framework
  - the host side API of the selected accelerator:
    e.g. `alpaka::memcpy(queue,dest, source)`
- what is not allowed:
  - device code:
    e.g. `ALPAKA_FN_ACC void func(TAcc const& acc, …) { … }`
  - kernel launches:
    e.g. `alpaka::exec<Acc1D>(queue, workDiv, kernel{}, …);`

`*.dev.cc` files by the *device compiler*

- for example, **`nvcc 12.2`** or **`hipcc 5.6`**
- what is available:
  - the host side API of the selected accelerator:
    e.g. `alpaka::memcpy(queue,dest, source)`
  - device code:
    e.g. `ALPAKA_FN_ACC void func(TAcc const& acc, …) { … }`
  - kernel launches:
    e.g. `alpaka::exec<Acc1D>(queue, workDiv, kernel{}, …);`
- what is discouraged
  - access to ROOT and the full CMSSW framework

CMS HLT reconstruction breakdown