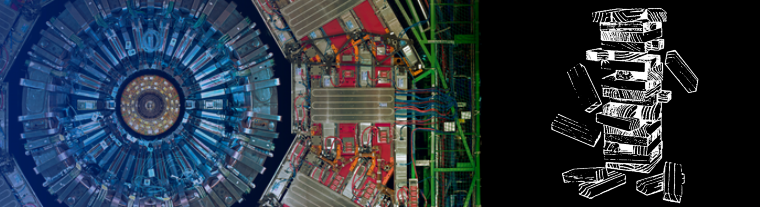
Adoption of the alpaka performance portability library in the CMS software

ACAT 2022 – October 27th, 2022

Andrea Bocci[1], Eric Cano[1], Antonio Di Pilato[2], Gabrielle Hugo[1], Vincenzo Innocente[1], Matti Kortelainen[3], Felice Pantaleo[1], Wahid Redjeb[1,4], Marco Rovere[1]

[1] CERN, [2] CASUS, [3] FNAL, [4] RWTH

motivation #1



why use GPUs for high energy physics event reconstruction ?

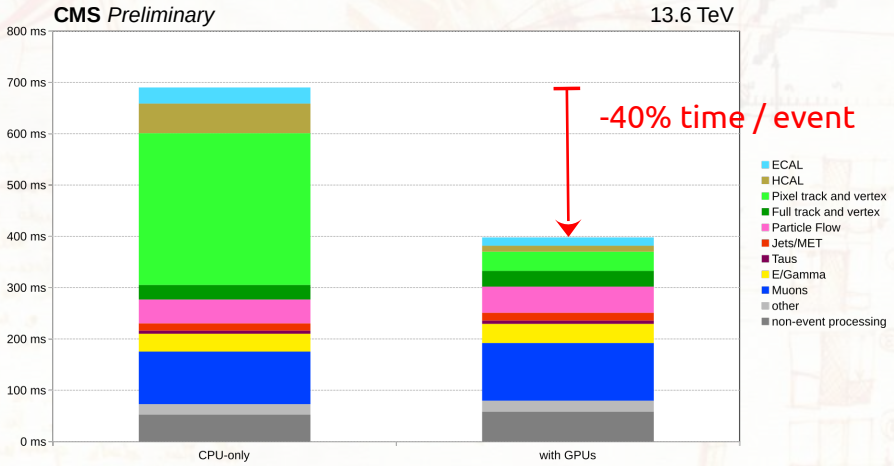
- event reconstruction is a *very* parallel problem, at multiple levels
  - events are independent: reconstruct multiple events
  - part of the reconstruction are independent: run different algorithms
  - many tasks can be expressed using parallel algorithms and data structures:
    - *e.g.* detector data "unpacking", fitting and calibration, clustering, track building and fitting, *etc.*

- parallel algorithms have some additional problems with respect to serial ones
  - more complicated to design and implement efficiently
    - *e.g.* divergences in the parallel execution may lead to suboptimal performance
  - undefined order of execution may produce results that are not fully reproducible
    - *e.g.* in combinatorial algorithms and reductions

a concrete application

- CMS High Level Trigger reconstruction
  - fully integrated in the CMSSW framework (can be reused also for offline reconstruction, though only for a small fraction of it)
  - validated offline on GPU-equipped nodes on CMS Tier-2s
  - deployed in production from the beginning of LHC Run-3 data taking
  - commissioned and optimised in the past few months
  - if you missed it, see the poster by Marc Huwiler, *Commissioning CMS online reconstruction with GPUs*

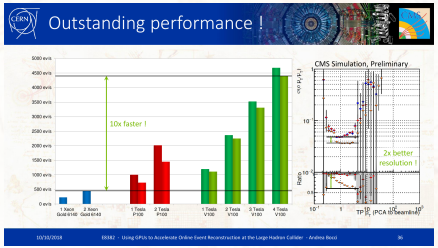
- with the deployment of a GPU-equipped HLT farm:
  - 70% better event processing throughput
  - 50% better performance per kW
  - 20% better performance per cost



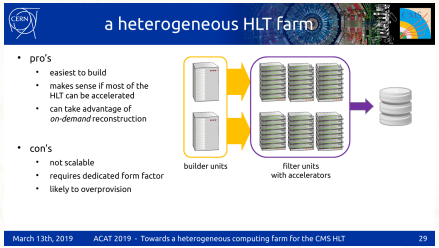
- work is ongoing to rewrite more algorithms to run on GPUs:
  - particle flow clustering: see the poster presented by Felice Pantaleo, *Particle Flow Reconstruction on Heterogeneous Architecture for CMS*
  - full primary vertex reconstruction: see the poster presented by Adriano Di Florio, *Primary Vertex Reconstruction for Heterogeneous Architecture at CMS*
  - *etc.*

how did we get there ?

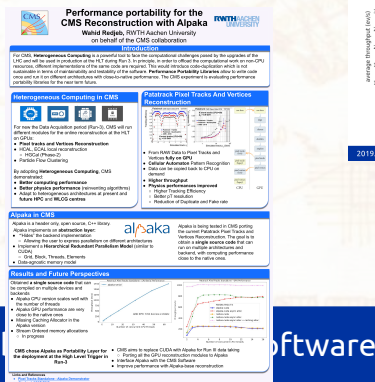
- 2016: first concrete interest in using (NVIDIA) GPUs for offloading reconstruction algorithms
- 2017: first CUDA code for Pixel local reconstruction
- 2018: continuous R&D activities
  - data structures, memory allocation strategies, caching and reuse
  - CUDA-based algorithms
- 2019: optimisations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



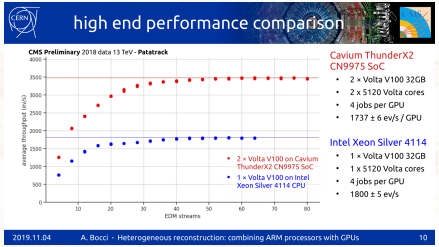
NVIDIA GTC (2018)



ACAT 2019



ACAT 2021



CHEP 2019

motivation #2

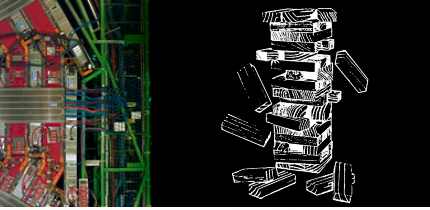
why do we care about *performance portability* ?

- on CPUs, we have been writing portable C++ code for a long time
  - single source, compiled for different architectures
  - target x86 CPUs (32 bit, then 64 bit), Power CPUs, ARM CPUs
- so far in CMS we have been using only NVIDIA GPUs, through the native CUDA API
  - how do we run code written for GPUs on CPUs, *e.g.* for running it offline where GPUs are not yet widespread ?
- present solutions …
  - in-house wrappers, and a lot of `#ifdef __CUDA_ARCH__` scattered through the code
  - good ol' code duplication
- adoption of GPUs from other vendors in HPCs is increasing
  - LUMI, in Kajaani, Finland, and Frontier, at Oak Ridge, US, use AMD MI250X GPUs
  - Aurora, at Argonne National Laboratory, US, will use Intel Xe GPUs
- can we target CPUs and GPUs from different vendors with a single code base ?

performance portability ?

- wide variety in computing architectures
  - CPUs (x86, ARM, Power, …)
  - GPUs (NVIDIA, AMD, soon Intel, …)
  - possibly FPGAs, or other dedicated hardware
- writing algorithms for each new backends requires a large investment
  - different programming tools (C++ vs CUDA vs ROCm vs SYCL vs …)
  - different algorithms (*e.g.* serial, branchy CPU code would have horrible performance on GPUs)
  - duplication of development, validation, and maintenance efforts
- a better approach: *performance portability* frameworks
  - abstract parallel programming models
  - expose the underlying details when necessary
  - (almost) native performance on different hardware

performance portability ?



OpenCL™

OpenMP®

OpenACC

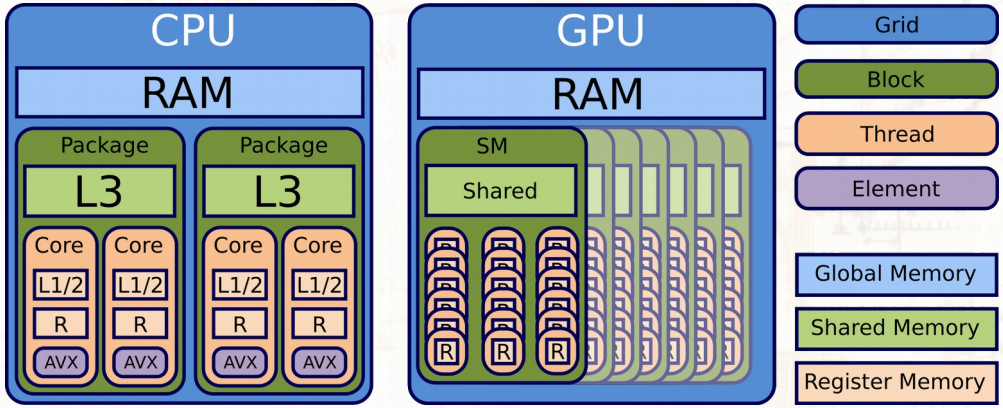
alpaka

SYCL™

okkos

what is alpaka ?

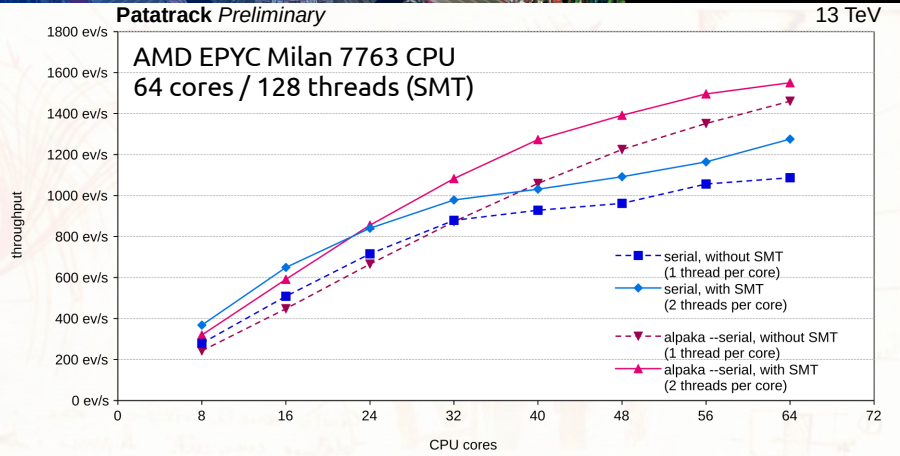
- alpaka is a header-only C++17 abstraction library for accelerator development
  - it aims to provide *performance portability* across accelerators through the abstraction of the underlying levels of parallelism
- it currently supports
  - CPUs, with serial and parallel execution
  - GPUs by NVIDIA, with CUDA
  - GPUs by AMD, with HIP/ROCm
  - support for Intel GPUs and FPGAs is *under development*, based on oneAPI

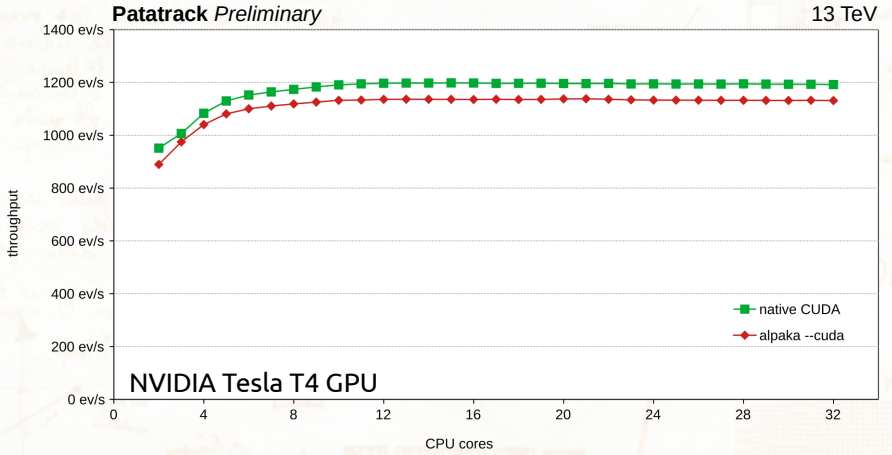


- it is easy to integrate with the CMSSW plug-in architecture
  - write code once, let the build system target multiple backends
  - a *single application* supports all these different backends *at the same time*
- for more information, see the poster by Jan Stephan, *Performance portability with alpaka*

why alpaka ?

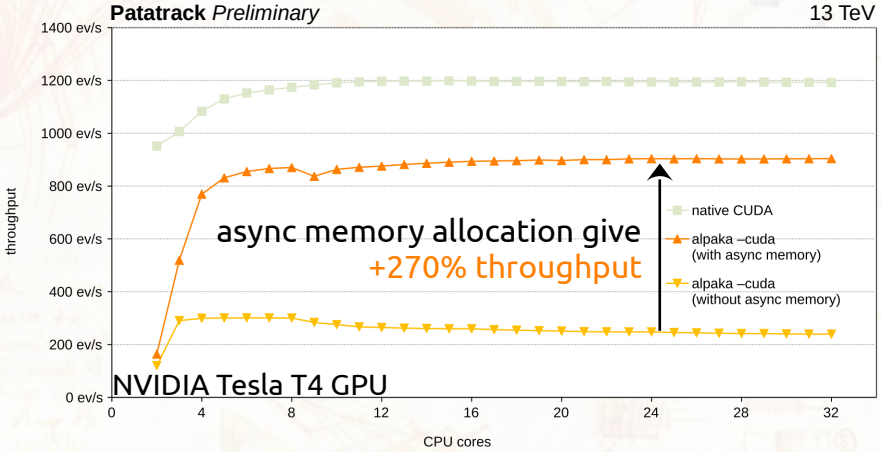
- it is production-ready today !
  - support all hardware CMS consider supporting
  - open source project, easy to contribute to
- header-only library, easy to integrate in the CMS framework
  - support multithreading in the host application
  - support multiple targets in a single build
    - GPUs from different vendors and different generations
    - CPUs with different execution modes, *e.g.* parallel execution using TBB
- evaluated in the Patatrack pixel-only standalone reconstruction
  - running on an AMD EPYC “Milan” 7763 CPU (64 cores / 128 threads SMT)
  - running on an NVIDIA Tesla T4 GPU
- good performance on the current hardware: CPUs and NVIDIA GPUs
  - today’s software: Pixel reconstruction
  - Phase-2 reconstruction - check Monday's talk by Tony Di Pilato, *Performance study of the CLUE algorithm with the alpaka library*





CMS contributions to alpaka

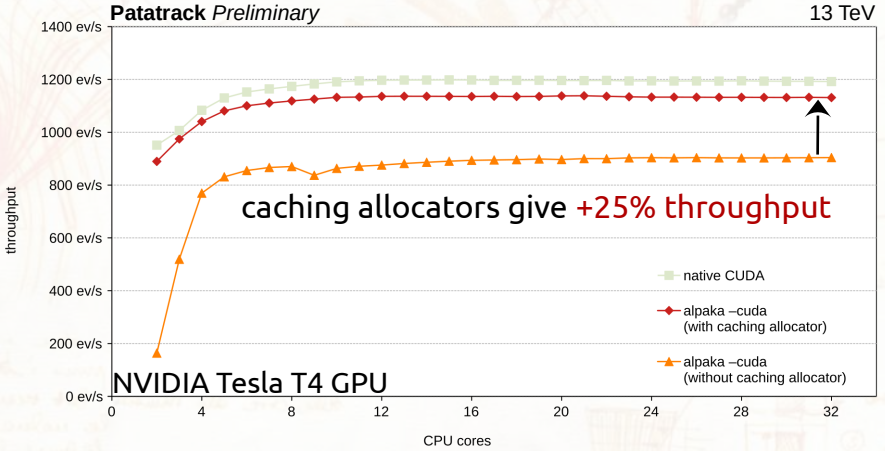
- contribute to the upstream project:
  - asynchronous memory allocations, on backends that support them
    - `cudaMallocAsync()`/`cudaFreeAsync()`
    - CUDA ≥ 11.2, CPUs
  - more efficient atomic operations on CPUs
    - `boost::atomic_ref` vs `std::mutex`
  - support for scalar buffers with a single element
  - user friendly syntax for accessing buffers' content:
    - `*buffer`, `buffer->member`, `buffer[i]`
  - more flexible support for CUDA and HIP
    - support for CUDA and HIP host APIs with a standard compiler
    - support for CUDA and HIP targets in a single build
  - bug fixes, improvements to the tests, *etc.*



Impact of using stream-ordered, asynchronous memory operations, running on an NVIDIA Tesla T4 GPU.

CMS developments on top of alpaka

- caching and runtime improvements on top of alpaka
  - caching memory allocator inspired by CUB's `cub::CachingDeviceAllocator`
    - implement queue-ordered, asynchronous semantics also for backends that do not support it natively (HIP, SYCL)
    - support global device memory and pinned host memory
  - caching and reuse of queues (*e.g.* CUDA streams) and events, to avoid the expensive construction and destruction of the underlying objects
  - various `parallel_for`-like wrappers using either indices or lambdas for *strided* block and grid loops
  - more efficient host tasks for CUDA callbacks



Impact of using a caching allocator for global device memory and pinned host memory, running on an NVIDIA Tesla T4 GPU.

- currently, these utilities are integrated in CMSSW code
  - once they are stable, we will consider what makes sense to keep in CMSSW, upstream to Alpaka, or move to a separate library

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::PltfCpu;
    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::PltfCudaRt;
  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/**, **plusings/** 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_serial.h
    │   └── classes_serial_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
```

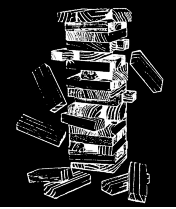
`alpaka/` directories

- all code under the `…/{src,plugins,test}/alpaka/` directories is compiled multiple times
  - into a separate shared library for a each backend
    - 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 10.2`**
  - what is available:
    - standard C++ functionality, *e.g.* ROOT and CMSSW framework
    - the host side API of the selected accelerator:
      e.g. `cudaMemcpyAsync(…)`
  - what is not allowed:
    - device code:
      e.g. `__global__ void kernel() { … }`
    - kernel launches:
      e.g. `kernel<<<blocks, threads>>>(…);`

- **`*.dev.cc`** files by the *device compiler*
  - for example, **`nvcc 11.5`**
  - what is available:
    - the host side API of the selected accelerator:
      e.g. `cudaMemcpyAsync(…)`
    - device code:
      e.g. `__global__ void kernel() { … }`
    - kernel launches:
      e.g. `kernel<<<blocks, threads>>>(…);`
  - what is discouraged
    - access to ROOT and the full CMSSW framework

goal: move to alpaka for 2023 data taking



A solid plan ahead !

- September 2022:
  - Alpaka framework and generic data structures in CMSSW
  - kickstart the migration
- November 2022:
  - Pixel and Tracking code ported to Alpaka
- December 2022 – next CMSSW release:
  - ECAL code ported to Alpaka
  - Pixel and Tracking configuration and validation updated
  - *central validation of the Pixel and Tracking migration*

at the same time, adopt a common set of data structures

- see the poster by Eric Cano, *Implementation of generic SoA data structure in the CMS software*

- January 2023:
  - HCAL code ported to Alpaka
  - ECAL configuration and validation updated
  - *central validation of the ECAL migration*
- February 2023 - CMSSW release for data taking:
  - HCAL configuration and validation updated
  - *central validation of the HCAL migration*
- March 2023
  - final validation, switch to Alpaka by default
- following release cycle:
  - port other GPU code targeting Run-3 to Alpaka
  - drop CUDA code, keep only Alpaka code, at least for Run-3

11th Patatrack Hackathon

… kickstarted by a dedicated hackathon !

Questions ?

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 cannot be instantiated, and are only used as a type
- 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&`

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

Accelerator

- encapsulates 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