

Adoption of the alpaka performance portability library in the CMS software

CHEP 2023 – May 9th, 2023

Andrea Bocci¹, Eric Cano¹, Adriano Di Florio², Antonio Di Pilato³, Jakub Andrzej Gajownik⁴, Gabrielle Hugo¹, Vincenzo Innocente¹, Matti Kortelainen⁵, Shahzad Muzaffar¹, Breno Orzari⁶, Felice Pantaleo¹, Dimitrios Papagiannis¹, Wahid Redjeb^{1,7}, Thomas Reis⁴, Marco Rovere¹, Davide Valsecchi⁸

¹ CERN, ² INFN and University of Bari, ³ formerly at CASUS,

⁴ STFC, ⁵ FNAL, ⁶ UNESP, ⁷ RWTH, ⁸ ETH Zurich

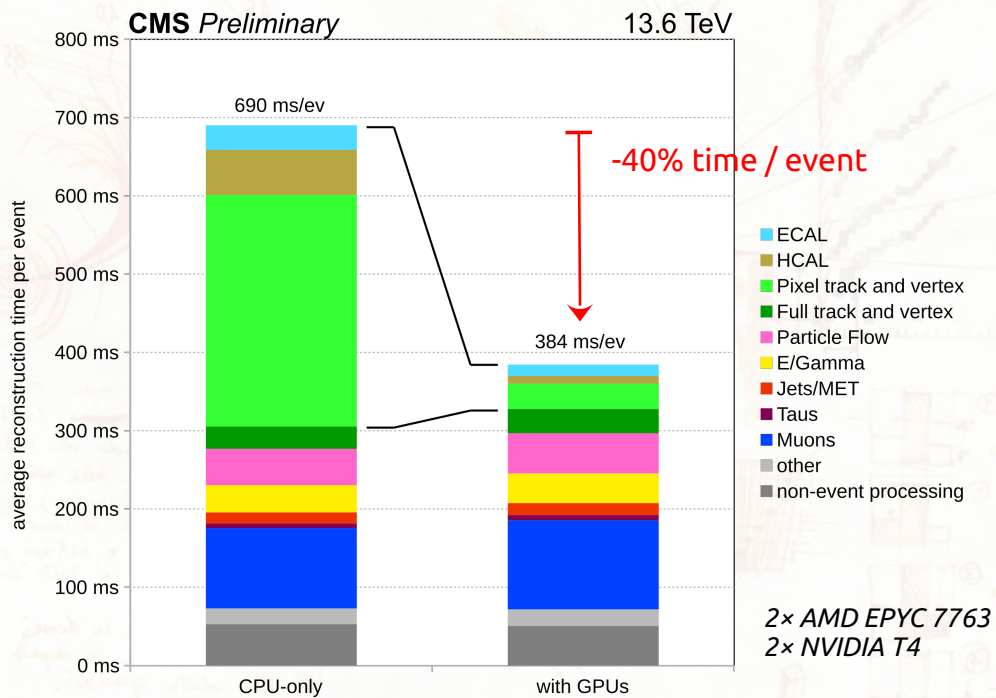


physics event reconstruction is a *very* parallel problem, at multiple levels

- many tasks can be expressed using parallel algorithms and data structures:
 - *e.g.* unpacking of detector data, signal fits and calibrations, clustering, track building and fitting, *etc.*
 - large parallelism → good candidates for GPU kernels
- many reconstruction steps are independent
 - run different algorithms in parallel → increase GPU occupancy and optimise CPU utilisation
- events are independent
 - reconstruct multiple events → *further* increase GPU occupancy and optimise CPU utilisation



- fully integrated in the CMSSW framework
 - can be reused in the offline reconstruction
 - validated offline on GPU-equipped nodes on CMS Tier-1 and Tier-2s
 - commissioned and optimised over last year
 - deployed in production since the beginning of LHC Run-3
 - if you missed it, [check the talk earlier today by Ganesh on the "Run-3 Commissioning of CMS Online HLT reconstruction using GPUs"](#)
- with the deployment of a GPU-equipped HLT farm:
 - 70% better event processing throughput
 - 50% better performance per kW
 - 20% better performance per initial cost
- work is ongoing to rewrite more algorithms to run on GPUs:
 - particle flow clustering
 - seeding of the electron reconstruction
 - full primary vertex reconstruction - [check yesterday's talk on the "GPU-based algorithms for primary vertex reconstruction at CMS"](#) by Carlos and more, targeting also the Phase-2 reconstruction



the *portability* challenge

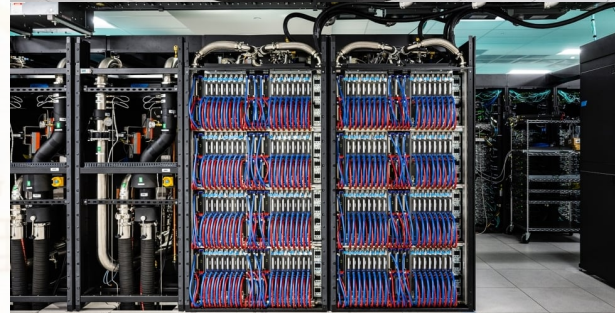


- new code written using the native CUDA API, targeting NVIDIA GPUs
 - most widespread GPU architecture, supported by all architectures used by CMS: x86, ARM, PowerPC
- brand new algorithms, e.g. Patatrack pixel reconstruction
 - implemented from scratch to run on GPUs
 - *ad hoc* compatibility layer, with a lot of `#ifdef __CUDA_ARCH__` scattered through the code
- new implementations of existing algorithms, e.g. calorimeter local reconstruction
 - two implementations: legacy (CPU-only) and parallel (GPU-only)
 - duplication of development, maintenance and validation efforts
- most offline sites (CERN, WLCG) do not use GPUs... *how do we run there ?*
- 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
- can we target different CPUs and GPUs with a single code base ?

maintenance issues!

code duplication!

and there ?



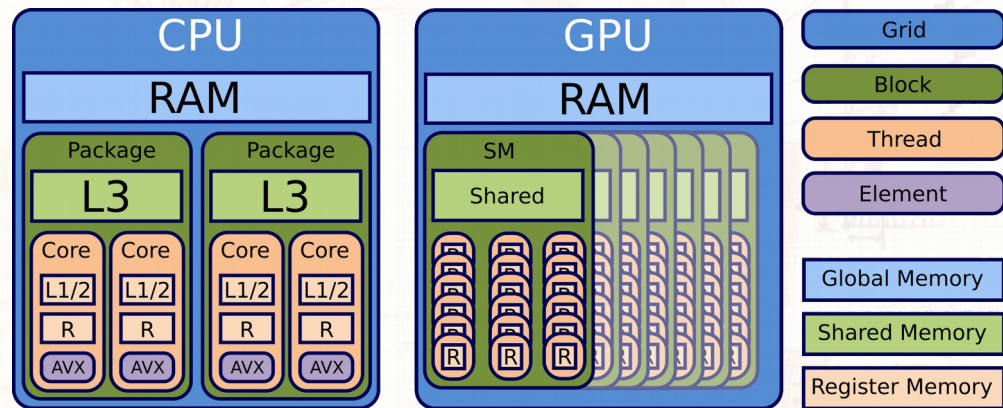
performance portability?



check Matti's talk "[Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code](#)"

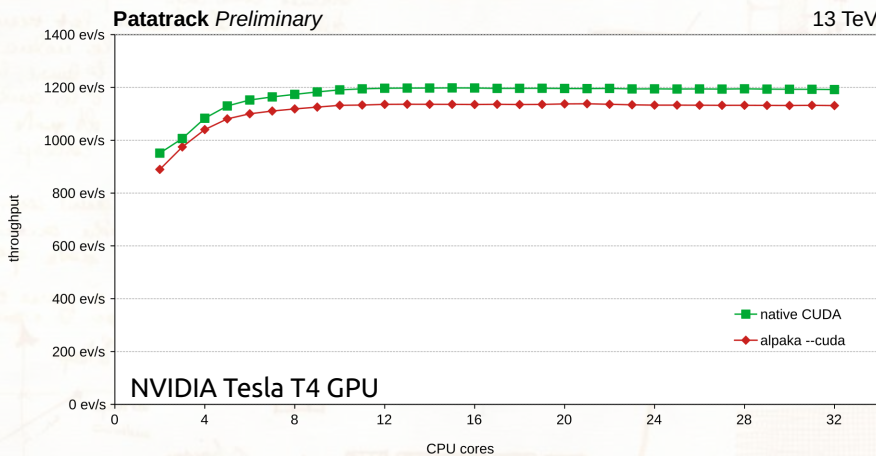
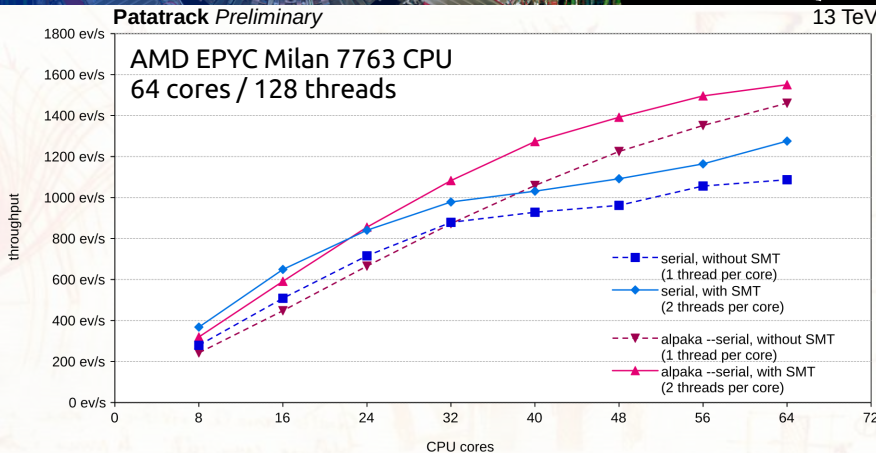


- alpaka is a header-only C++17 abstraction library for heterogeneous software development
 - it aims to provide *performance portability* across accelerators through the abstraction of the underlying levels of parallelism
 - may* expose the underlying details when necessary
 - (almost) *native* performance on different hardware
- supports all platforms of interest to CMS
 - x86, ARM and Power CPUs
 - with serial and parallel execution
 - NVIDIA and AMD GPUs
 - with CUDA and ROCm backends
 - support for Intel GPUs and FPGAs is *under development*, based on SYCL and oneAPI
- it is production-ready today !
 - open source project, easy to contribute to: <https://github.com/alpaka-group/alpaka/>





- evaluated on Run-3 algorithms within the Patatrack pixel-only standalone reconstruction
- good performance on current hardware
 - running on an AMD EPYC "Milan" 7763 CPU (64 cores / 128 threads SMT)
 - running on an NVIDIA Tesla T4 GPU
- 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
- low-level approach, very close to CUDA
 - easy to port code from CUDA, and to teach to students

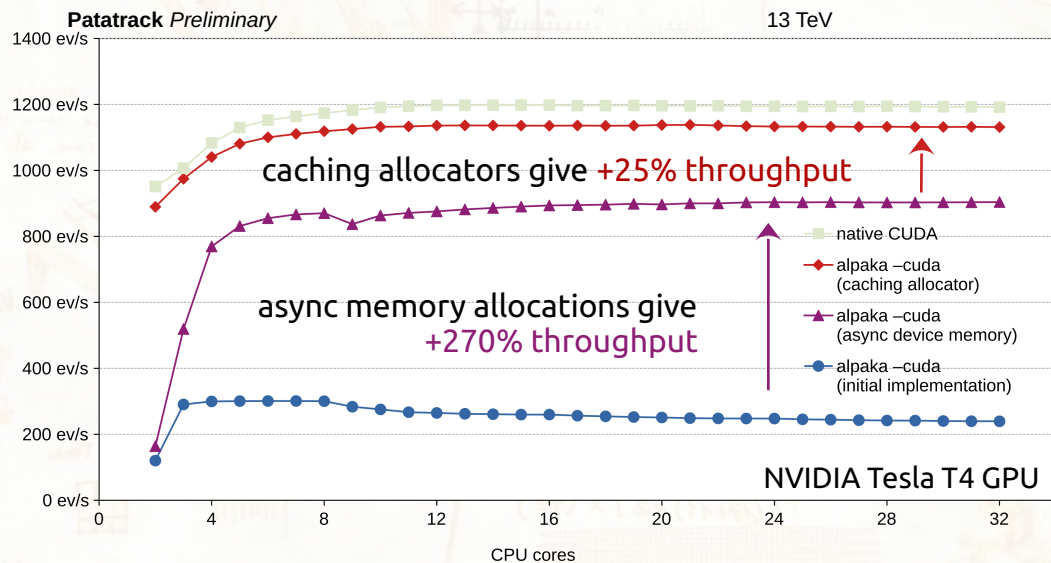




- fruitful collaboration with the Alpaka development team
 - improve efficiency for CMS workflows
 - contribute support for new features and architectures
 - students' projects !

- more flexible support for CUDA and HIP
 - support for CUDA and HIP APIs in the host compiler
 - support for CUDA and HIP targets in a single build
- asynchronous memory allocations, on backends that support them
 - `cudaMallocAsync()/cudaFreeAsync()`
 - CUDA \geq 11.2, ROCm \geq 5.4, CPUs
- caching of GPU resources
 - streams and events
 - device and host memory buffers
- contribute to the SYCL implementation
 - support for USM memory model in oneAPI
- more efficient atomic operations
- improved memory buffer and kernel syntax
- bug fixes, improvements to the tests, *etc.*

in progress





- 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 and 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
```




- to support the compilation of **alpaka**-based plugins and libraries for multiple backends, we have introduced a new directory structure and 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
    
```



- 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 10.2**
 - what is available:
 - standard C++ functionality, 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 11.5**
 - 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



- migration of CUDA code in CMSSW to Alpaka is ongoing
 - implemented build rules, core framework, unit tests
 - support for CPUs, NVIDIA GPUs via CUDA, AMD GPUs via ROCm
- opportunity to review various design choices taken in the past years
 - 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
- aim to deploy an Alpaka-based version of the HLT during this year’s data taking
 - eventually, drop support for native CUDA code



Questions ?

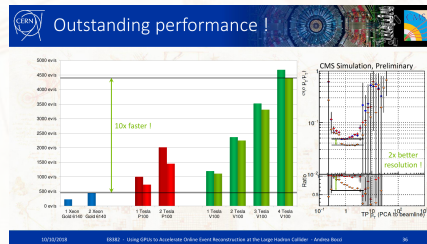




a brief history of GPUs at CMS



- 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 (“cudacompact”)
- 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
- 2022: deployment in production



NVIDIA GTC (2018)

a heterogeneous HLT farm

- pro's
 - easiest to build
 - makes sense if most of the HLT can be accelerated
 - can take advantage of on-demand reconstruction
- con's
 - not scalable
 - requires dedicated form factor
 - likely to overprovision

builder units → filter units with accelerators

ACAT 2019

March 13th, 2019 ACAT 2019 - Towards a heterogeneous computing farm for the CMS HLT

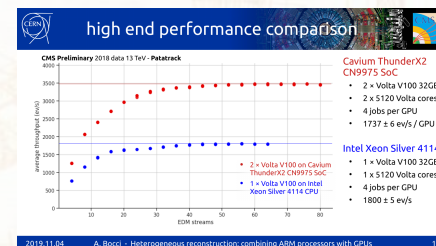
Performance portability for the CMS Reconstruction with Alpaka

Wolfgang Riegler, RWTH Aachen University on behalf of the CMS collaboration

Alpaka is a heterogeneous computing framework that allows to port parallel applications to different hardware architectures. It is designed to be easy to use and to integrate with existing code. It provides a high-level API for writing parallel applications and a low-level API for writing performance-critical code. It is designed to be easy to use and to integrate with existing code. It provides a high-level API for writing parallel applications and a low-level API for writing performance-critical code.

Alpaka is a heterogeneous computing framework that allows to port parallel applications to different hardware architectures. It is designed to be easy to use and to integrate with existing code. It provides a high-level API for writing parallel applications and a low-level API for writing performance-critical code. It is designed to be easy to use and to integrate with existing code. It provides a high-level API for writing parallel applications and a low-level API for writing performance-critical code.

ACAT 2021



CHEP 2019





- 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



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_ptrs`, and should be passed by value or by `const&`



Queues and Events

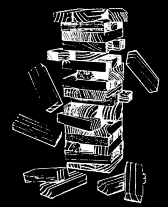
- queues identify a work queue where tasks (memory ops, kernel executions, ...) are executed in order
 - for example, a queue could represent an underlying CUDA stream or a CPU thread
- queues can be sync(hronous or blocking) or async(hronous or non-blocking)
 - work submitted to a sync queue is executed 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

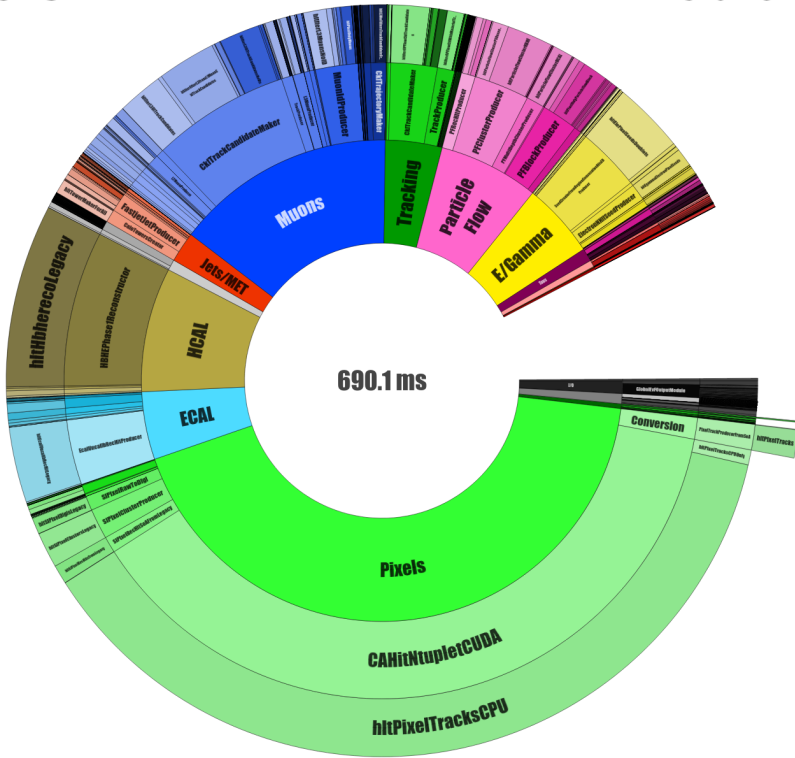


CMS HLT reconstruction breakdown



CMS

13.6 TeV



CMS

13.6 TeV

