GPUs at CMS

Sam Harper (STFC-RAL)

SWIFTHEP Meeting

GridPP/SwiftHEP Joint Meeting

March 29nd 2023

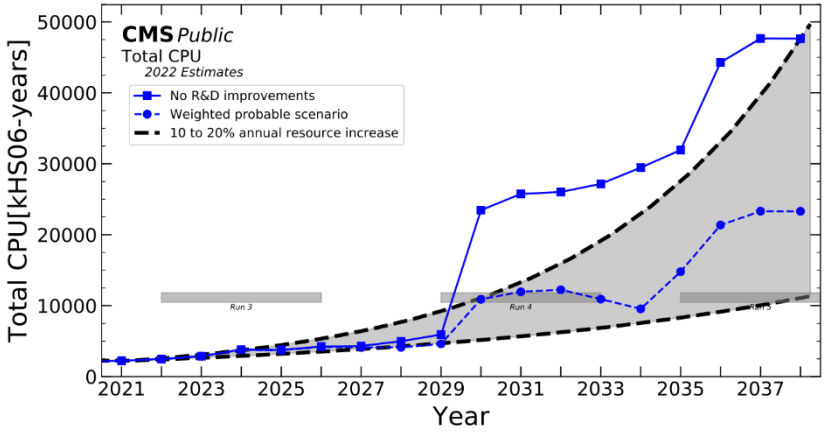
Introduction

- CMS has deployed a heterogenous HLT farm for 2022 data taking
- this talk is to give an account of that experience, current status and plans
- note: timing is good as CMS discussed much of this at ACAT 2022 (https://indico.cern.ch/event/1106990/)
  - GPU commissioning: https://indico.cern.ch/event/1106990/contributions/4991283/
  - remote GPUs: https://indico.cern.ch/event/1106990/contributions/5011939/
  - Alpaka: https://indico.cern.ch/event/1106990/contributions/5011939/

disclaimer: I was CMS trigger coordinator during the initial proposal and deployment and I have followed this topic closely but have not personally written a single line of GPU code for this project. The views here are also my own and do not necessarily represent those of CMS

CMS Heterogenous Strategy

- it is commonly recognised that CPUs alone will not allow us to meet computing requirements for HL-LHC



- CMS decided in 2019 to add GPUs to HLT farm to gain experience running in an heterogenous environment
  - **target was to be "break even"**, ie that the cost of the GPU is offset by the reduction in CPU required, which is ~30%
  - deliberately simple setup
  - the experience gained would be used to guide our Run4 computing strategy
- the HLT is the ideal test bed as HLT farm is 100% controlled by CMS

CMS GPU strategy

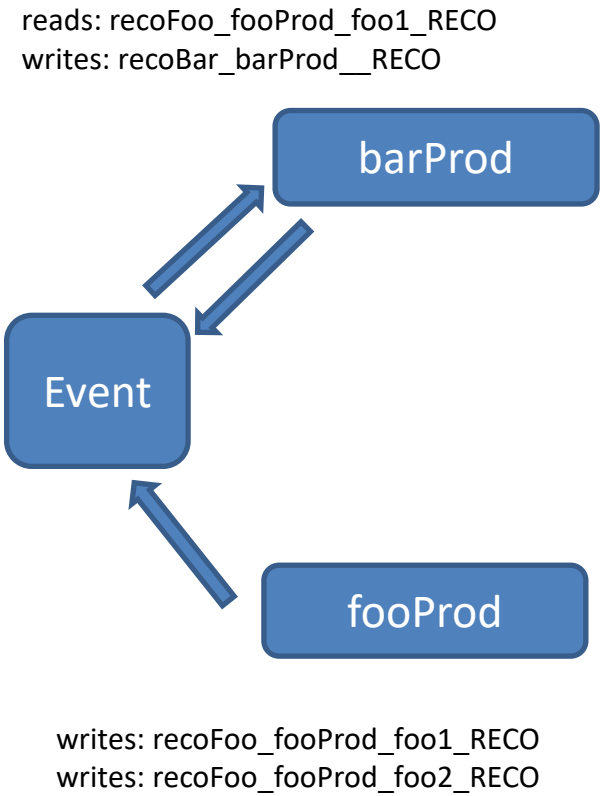
- focus on three key areas: ECAL , HCAL and pixel track reconstruction
  - ECAL and HCAL reconstruction is very similar
  - these three areas contribute about ~30% of the CPU time, represent large bang for buck *at the HLT*
- chosen technology: CUDA
  - only realistic choice at the time
  - ECAL/HCAL and pixel tracking algos re-written to be CUDA based
- support must be maintained for systems without GPUs
  - HLT will always be run at P5 on a GPU but also needs to be run for MC production on normal grid sites
  - so both CPU and GPU need to be supported

CMS HLT Farm

- 200 nodes, each node:
  - 2x AMD EPYC 7763 “Milan” 64 core processors
    - 128 physical cores, 256 threads
  - 2x Nividia T4 GPUs
    - 1.6 Ghz, 16GB
  - deployed since Run3 start (July 4th)
- GPU code runs at 90kHz
  - pixel : 88% of events
  - ECAL : 70% of events
  - HCAL : 65% of events

CMSSW: Key points

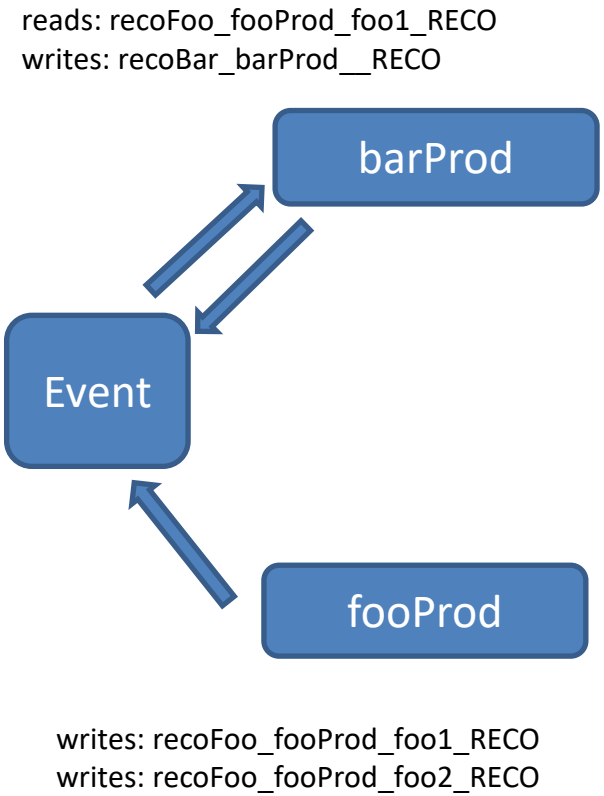
- CMSSW is the CMS software framework which runs all CMS workflows
- it consists of a series of independent c++ modules configured by a python config
  - modules communicate solely via reading/writing products to the event object
  - once is a product is written to the event, it immutable
- at construction each module registers
  - the products it consumes
  - the products it produces
- products have a unique name:
  - c++type_moduleLabel_instanceLabel_processLabel
  - a process label is the label of the job, ie HLT, RECO etc and the same process label can not be used in a later job
  - instance label is there incase a module writes more than one product of the same c++ type
- the scheduler only runs modules whose products are consumed



by requesting the recoBar_barProd product, you force both the barProd and fooProd to run

CMSSW: Key points (II)

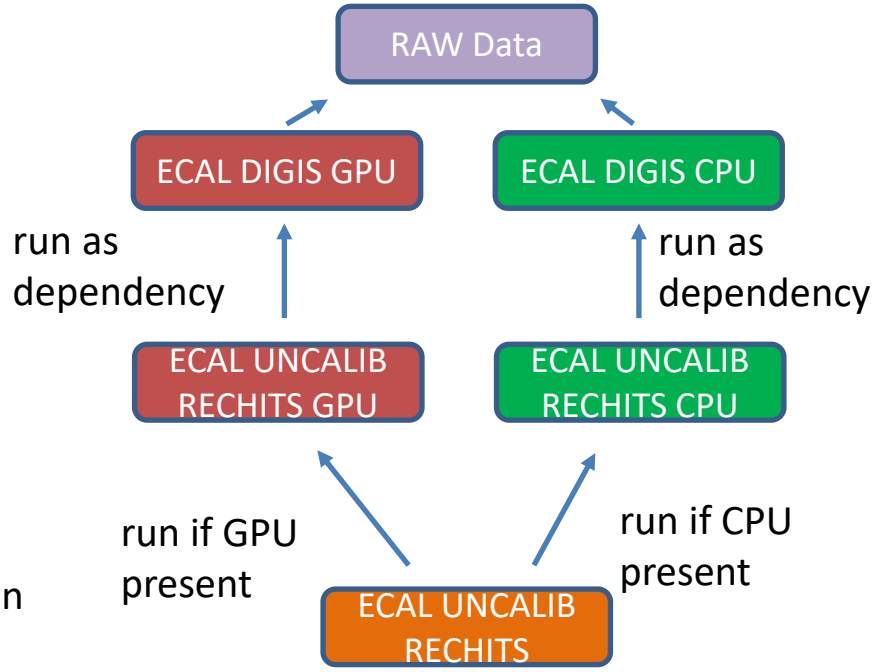
- HLT and RECO share the same release and share the same code wherever possible
- usually local reco modules (eg all ECAL, HCAL and Pixel local reco) are ~identical between HLT and RECO (or some minor config differences)
  - everything ported for GPU so far for the HLT can in theory be used in reconstruction jobs as well
- higher level reco modules differ but try and reuse algo code as much as possible
  - eg Electrons are built variable by variable to reject early at the HLT while RECO builds the entire electron in one go



by requesting the recoBar_barProd product, you force both the barProd and fooProd to run

Handing CPU and GPU modules

- CMSSW has GPU modules and CPU modules
  - a GPU module can only run on GPU, will crash if one not present
- to allow the same cfg file to run on CPU+GPU and CPU, the concept of a **SwitchProducer** was introduced
  - has two modules nested in it, a module to run when there is a GPU present and a module to run when no GPU is present
  - can be extended to other devices
- only the end product needs a switch producer, intermediate dependences are handled by on demand execution
- **single HLT config file: device availability on node will determine if it runs the GPU or CPU modules**

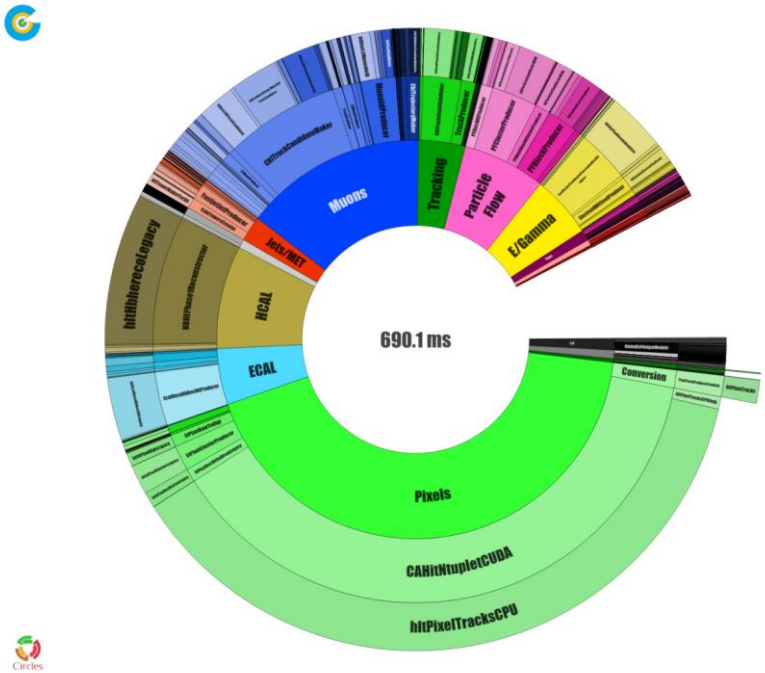


switch producer, decides which module to run based on available devices

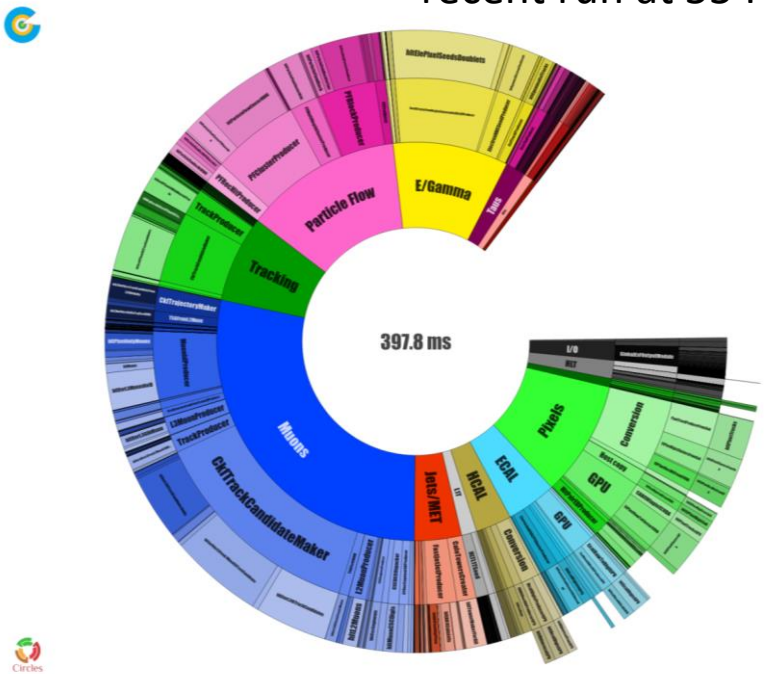
```
hltEcalUncalibRecHit = SwitchProducerCUDA(
 cpu = cms.EDAlias(
    hltEcalUncalibRecHitLegacy = cms.VPSet(
      cms.PSet(  type = cms.string( "*" )        )
    )
 ),
cuda = cms.EDAlias(
    hltEcalUncalibRecHitFromSoA = cms.VPSet(
      cms.PSet(  type = cms.string( "*" )        )
    )
 ),
)
```

Performance

recent run at 55 PU



CPU only



CPU+ GPU

- 42% reduction in CPU processing time from offloading ECAL, HCAL, and Pixels
  - copy and conversion to legacy CPU formats is a significant fraction of remaining time

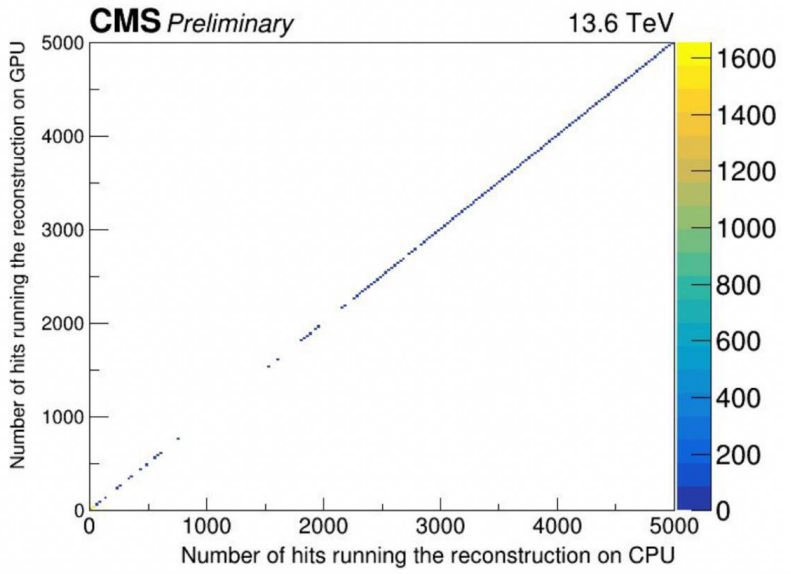
from ATAC 2022
https://indico.cern.ch/event/1106990/contributions/4991283/attachments/2533710/4360071/HLT_GPU_Poster_MH_final.pdf

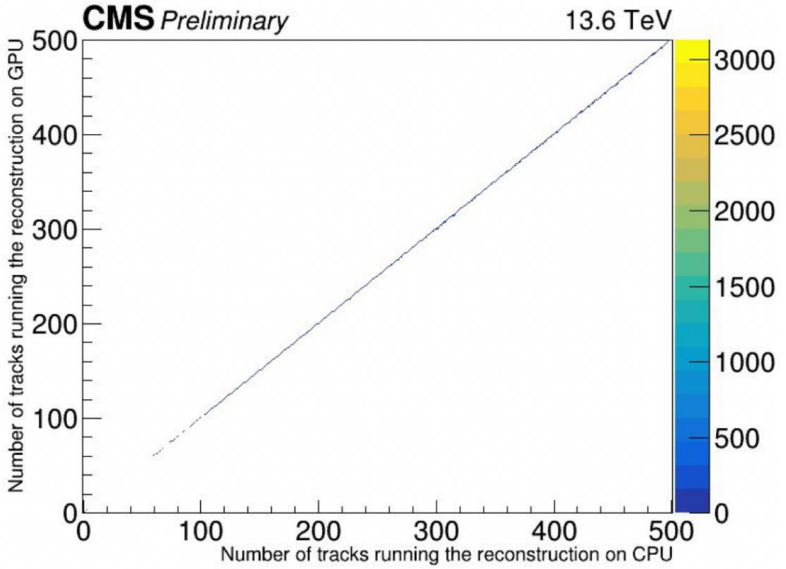
GPU vs CPU comparisons

- for a small fraction of events, HLT runs both the CPU and GPU reco algos and compares result
- the next few plots are taken from a pp run in Oct 22

results

- identical number of pixel rec-hits

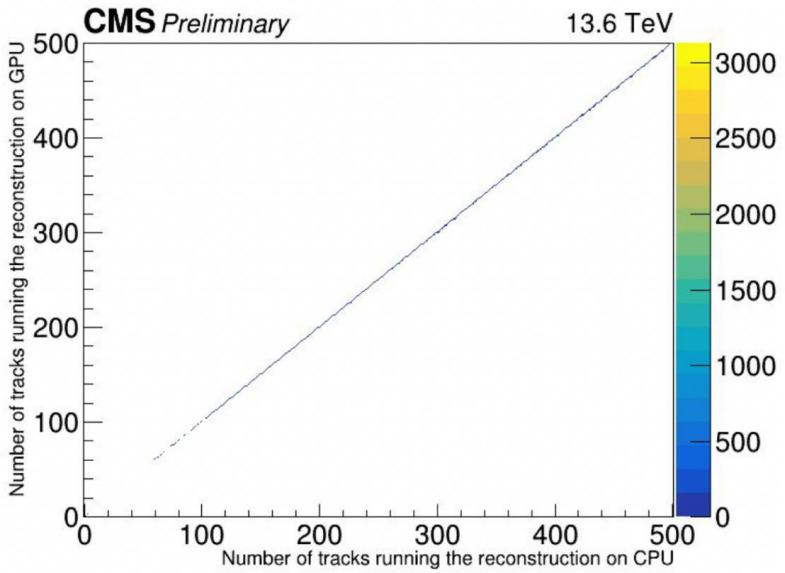


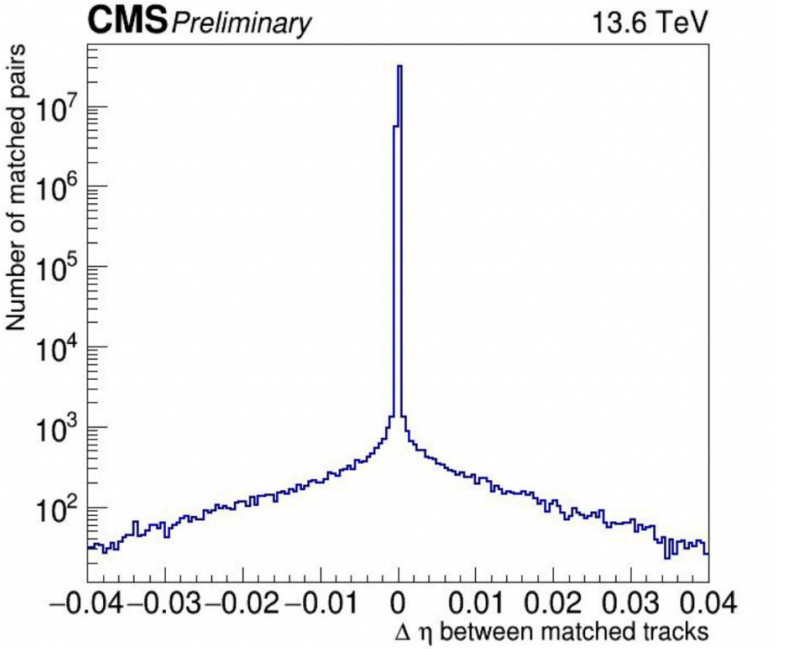


from ATAC 2022
https://indico.cern.ch/event/1106990/contributions/4991283/attachments/2533710/4360071/HLT_GPU_Poster_MH_final.pdf

GPU vs CPU comparisons

- small mismatch in number of tracks and track eta
- thought to be differences due to float (GPU) vs double (CPU)

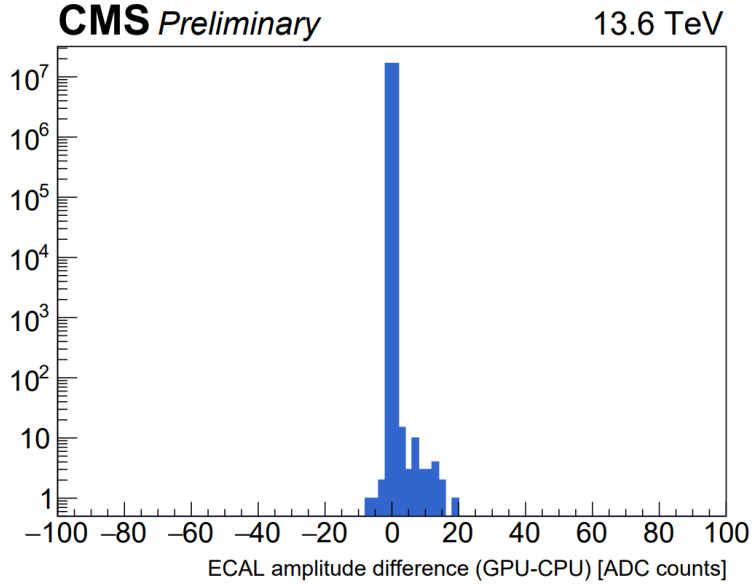


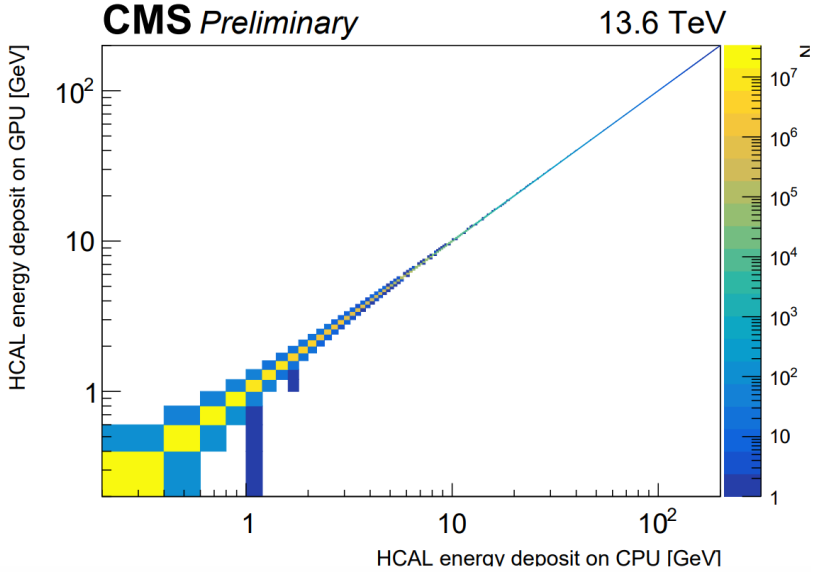


from ATAC 2022
https://indico.cern.ch/event/1106990/contributions/4991283/attachments/2533710/4360071/HLT_GPU_Poster_MH_final.pdf

GPU vs CPU comparisons

- ECAL amplitude: differ in 1 / 10^6 ecal rec hits
- also tiny difference in HCAL energy deposits
- disagreement is thought to be due to double vs float issues





from ATAC 2022
https://indico.cern.ch/event/1106990/contributions/4991283/attachments/2533710/4360071/HLT_GPU_Poster_MH_final.pdf

next steps:

- CMS is rapidly iterating on its GPU models
  - we are getting better at defining common tasks
    - eg SoA formats now much easier to produce and have wrappers which make it more developer friendly
- CMS is actively trying to port new things algorithms to GPU
  - E/gamma algorithms, Particle flow
- improving infrastructure support
  - now can submit jobs via crab (cms grid software) to run on GPU nodes

next steps : performance portability

- the current approach of having separate CPU and CUDA code does not scale well
  - support for additional device types likely will be also needed, thus further adding to maintenance burden
    - eg effort on FPGAs already underway
- solution : portability/abstraction layers -> write once, run anywhere
- many options on the market: SYCL, Kokkos, alpaka…
- **CMS HLT is moving to alpaka** (https://github.com/alpaka-group/alpaca)
  - group is actively collaborating with CMS developers, infact many CMS developers are now listed as authors of Alpaka
  - very close to being deployed in production for patatrack code
    - PRs are either there or very close
    - ECAL code is coming along
  - note: it has been much less painful moving CUDA to alpaka than it was re-writing the CPU code base for CUDA as they share the same concepts
- note: the final approach for Phase-II is not decided, however alpaka now has first mover advantage in CMS
  - in my opinion, other approaches will need to demonstrate considerable advantages over alpaka if CMS would switch to them

More on Alpaka

- header only c++17 library

Its aim is to provide performance portability across accelerators through the abstraction (not hiding!) of the underlying levels of parallelism.

It is platform independent and supports the concurrent and cooperative use of multiple devices such as the hosts CPU as well as attached accelerators as for instance CUDA GPUs and Xeon Phis (currently native execution only). A multitude of accelerator back-end variants using CUDA, OpenMP (2.0/4.0), Boost.Fiber, std::thread and also serial execution is provided and can be selected depending on the device. Only one implementation of the user kernel is required by representing them as function objects with a special interface. There is no need to write special CUDA, OpenMP or custom threading code. Accelerator back-ends can be mixed within a device queue. The decision which accelerator back-end executes which kernel can be made at runtime.

The abstraction used is very similar to the CUDA grid-blocks-threads division strategy. Algorithms that should be parallelized have to be divided into a multi-dimensional grid consisting of small uniform work items. These functions are called kernels and are executed in parallel threads. The threads in the grid are organized in blocks. All threads in a block are executed in parallel and can interact via fast shared memory. Blocks are executed independently and can not interact in any way. The block execution order is unspecified and depends on the accelerator in use. By using this abstraction the execution can be optimally adapted to the available hardware.

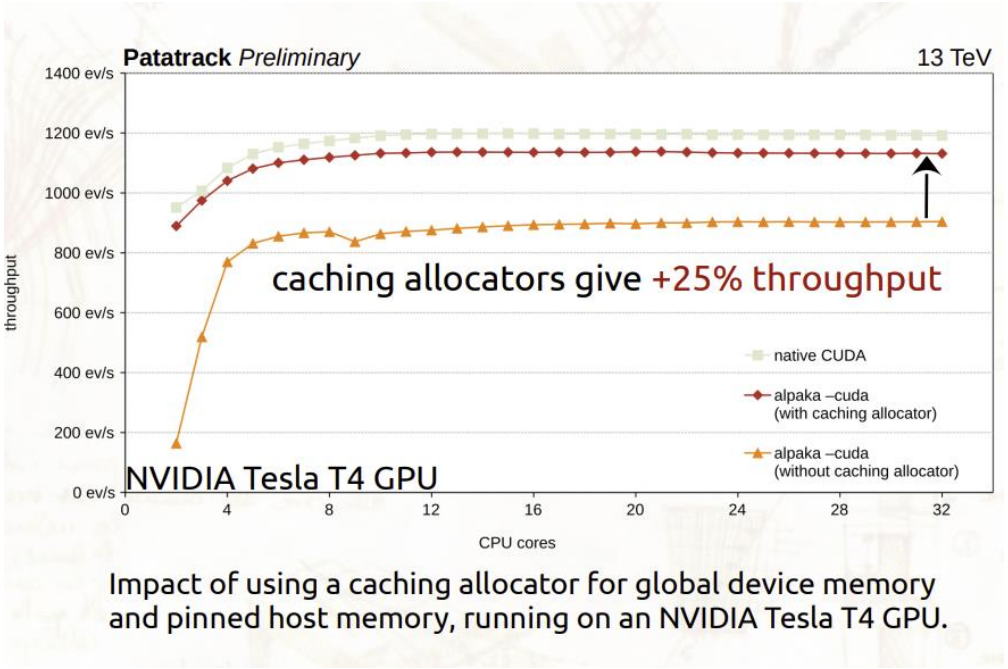
Accelerator Back-ends

Accelerator Back-end	Lib/API	Devices	Execution strategy grid-blocks	Execution strategy block-threads
Serial	n/a	Host CPU (single core)	sequential	sequential (only 1 thread per block)
OpenMP 2.0+ blocks	OpenMP 2.0+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
OpenMP 2.0+ threads	OpenMP 2.0+	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
OpenMP 5.0+	OpenMP 5.0+	Host CPU (multi core)	parallel (undefined)	parallel (preemptive multitasking)
		GPU	parallel (undefined)	parallel (lock-step within warps)
OpenACC (experimental)	OpenACC 2.0+	Host CPU (multi core)	parallel (undefined)	parallel (preemptive multitasking)
		GPU	parallel (undefined)	parallel (lock-step within warps)
std::thread	std::thread	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
Boost.Fiber	boost::fibers::fiber	Host CPU (single core)	sequential	parallel (cooperative multitasking)
TBB	TBB 2.2+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
CUDA	CUDA 9.0+	NVIDIA GPUs	parallel (undefined)	parallel (lock-step within warps)
HIP(clang)	HIP 4.0+	AMD GPUs	parallel (undefined)	parallel (lock-step within warps)

- https://github.com/alpaka-group/alpaka

Alpaka Performance

- orange is “default” alpaca
- red is CMSSW extensions adding caching and run time improvements
  - caches and reuses queues and events to avoid expensive construction and destruction of underlying objects
- can see CMSSW alpaka implementation close to native CUDA performance



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

A. Bocci, ACAT22
https://indico.cern.ch/event/1106990/contributions/4991273/

other areas of R&D

- ML inference on GPU
  - so far there has been interest but nothing has run in production
  - closest to production is the SONIC project (mostly US based) which provides ML inference as a service
    - Services for Optimized Network Inference on Coprocessors
    - https://iopscience.iop.org/article/10.1088/2632-2153/abec21
    - https://www.computer.org/csdl/proceedings-article/h2rc/2020/235400a038/1pVHdDr0PzG
- extension to other accelerators
  - eg FPGAs , AMD cards etc
  - ExternalWorker which CMS uses for GPU in theory can be anything external…
    - in fact I just got it to run a calculation on a FPGA…
- GPU/FPGA as a service
  - currently CMS farm each node has a GPU
  - in the future this may not be the optimum solution and there may be a dedicated GPU server which nodes can submit jobs to
    - offline, it is certain some farms will be setup this way
  - SONIC natively has this capability; other approaches are being researched too

Deployment Outside the HLT

- offline reco and online reco (HLT) share the same code wherever possible at CMS
- in particular calorimeter reco and clustering is identical
  - nothing is stopping us using the GPU versions of this code in offline reco but we don't currently use them
  - I suspect its because the modules we have ported are a significant fraction of the HLT CPU budget but not as significant for offline budget (offline always run expensive algos while HLT rarely runs them)
- there is a growing push to have GPU enabled reco code "available" and there is work going forward to this
  - E/gamma pixel seeding is a natural candidate we want to move
  - I suspect by the end of Run3 we will be opportunistically using GPU resources in our offline jobs
- also note: for the HLT, same config is used for data and MC so for MC jobs, a small part can already be run on the GPU if a GPU is available

Summary

- CMS has deployed a heterogenous farm for Run3
  - three areas ported (ECAL, HCAL, pixel reco)
  - further areas are being worked on (E/gamma algos, Particle Flow)
  - currently at 42% reduction of CPU, beating original 30% target
- system is performing well
  - some teething problems but to be expected with new code base
  - good CPU-GPU algorithm agreement
- infrastructure to support this in place
  - GPU dev machines, GPU-CPU validation workflows, tool upgrades, framework support all done
- next step: port GPU algos to Alpaka
  - patatrack pixel tracks ready / close to ready on this

CMSSW code

- framework for heterogenous code
  - https://github.com/cms-sw/cmssw/tree/master/HeterogeneousCore
- ECAL uncalibrated rechit producer (example of GPU module)
  - https://github.com/cms-sw/cmssw/blob/master/RecoLocalCalo/EcalRecProducers/plugins/EcalUncalibRecHitProducerGPU.cc
  - https://github.com/cms-sw/cmssw/blob/master/RecoLocalCalo/EcalRecProducers/plugins/EcalUncalibRecHitMultiFitAlgoGPU.cu

Bibliography

- talk on CMSSW for non CMS folks (C. Jones)
  - https://indico.cern.ch/event/1038551/contributions/4390441/attachments/2256302/3828631/CMSSW%20for%20DUNE%20Framework%20Workshop.pdf
- M Huwiler poster ACAT 2022
  - https://indico.cern.ch/event/1106990/contributions/4991283/
- M. Kortelainen HSF WLCG Workshop, May 2022
  - https://indico.cern.ch/event/908146/contributions/3826757/subcontributions/305916/attachments/2035394/3410634/20200512-HSF_Workshop_Heterogeneous_CMSSW.pdf
- patatrack project:
  - https://patatrack.web.cern.ch/patatrack/index.html
  - https://github.com/cms-patatrack

CPU & GPU running

- try and keep data on the device as much as possible
  - copies back and forth kill performance
- data formats have to be adjusted for GPU
  - struct of arrays (SoA) rather array of structures (AoS), traditionally used in HEP
  - must be converted back to traditional object approach to be consumed by CPU code

```
#ifndef CUDADataFormats_EcalRecHitSoA_interface_EcalUncalibratedRecHit_h
#define CUDADataFormats_EcalRecHitSoA_interface_EcalUncalibratedRecHit_h

#include <array>
#include <vector>

#include "CUDADataFormats/CaloCommon/interface/Common.h"
#include "CUDADataFormats/EcalRecHitSoA/interface/RecoTypes.h"
#include "DataFormats/EcalDigi/interface/EcalDataFrame.h"

namespace ecal {

  template <typename StoragePolicy>
  struct UncalibratedRecHit : public ::calo::common::AddSize<typename StoragePolicy::TagType> {
    UncalibratedRecHit() = default;
    UncalibratedRecHit(const UncalibratedRecHit&) = default;
    UncalibratedRecHit& operator=(const UncalibratedRecHit&) = default;

    UncalibratedRecHit(UncalibratedRecHit&&) = default;
    UncalibratedRecHit& operator=(UncalibratedRecHit&&) = default;

    typename StoragePolicy::template StorageSelector<reco::ComputationScalarType>::type amplitudesAll;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type amplitude;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type amplitudeError;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type chi2;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type pedestal;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type jitter;
    typename StoragePolicy::template StorageSelector<reco::StorageScalarType>::type jitterError;
    typename StoragePolicy::template StorageSelector<uint32_t>::type did;
    typename StoragePolicy::template StorageSelector<uint32_t>::type flags;

    template <typename U = typename StoragePolicy::TagType>
    typename std::enable_if<std::is_same<U, ::calo::common::tags::Vec>::value, void>::type resize(size_t size) {
      amplitudesAll.resize(size * EcalDataFrame::MAXSAMPLES);
      amplitude.resize(size);
      amplitudeError.resize(size);
      pedestal.resize(size);
      chi2.resize(size);
      did.resize(size);
      flags.resize(size);
      jitter.resize(size);
      jitterError.resize(size);
    }
  };

}  // namespace ecal

#endif  // CUDADataFormats_EcalRecHitSoA_interface_EcalUncalibratedRecHit_h
```

Personal Observations on GPU commissioning

- we have had more crashes with GPU code enabled, there has been a long line of bug fixes
  - note, “more” is still negligible, in Run2 the CMS HLT crashed very rarely
- most crashes involve data that is in some way bad
  - I suspect its just teething problems which you would get with any new code rather anything fundamental about GPU coding style
- there has been a lot of work to try and get the GPU and CPU algorithms in agreement (and keep them that way)
  - this is continuously monitored both at P5 and and in offline validation
  - sometimes it trying to reproduce somewhat arbitrary decisions by the original code
- MC validation was often not sufficient to catch issues
  - many issues were data only
- in general this was harder than the Run2 multithreading migration and has been a major focus of the CMS HLT group
  - although that migration was pretty easy from a CMS users perspective
- but has been easier than commissioning a new detector (eg new pixels in 2017)
  - still a huge amount of work has been done for this, it has been a massive project

Experience of Alpaka

RAL (through Thomas Reis) is currently porting ECAL local reconstruction from CUDA to Alpaka

pros:

- very similar to CUDA, easy migration path
- easy to copy data/back forth on the GPU
  - cmssw has introduced “portable collections” where you can write things in an AoS way which is then converted to SoA using alpaca
- case study: a masters student starting from ~zero managed to port the ECAL weights algo to Alpaka

cons:

- alpaka still under development and has bugs (which are promptly fixed when pointed out, situation is rapidly improving
- being a templated library, c++ error messages are “not fun”
  - a missing const on a function spat out a lot of test and took some time to actually pinpoint

in general, highly positive experience