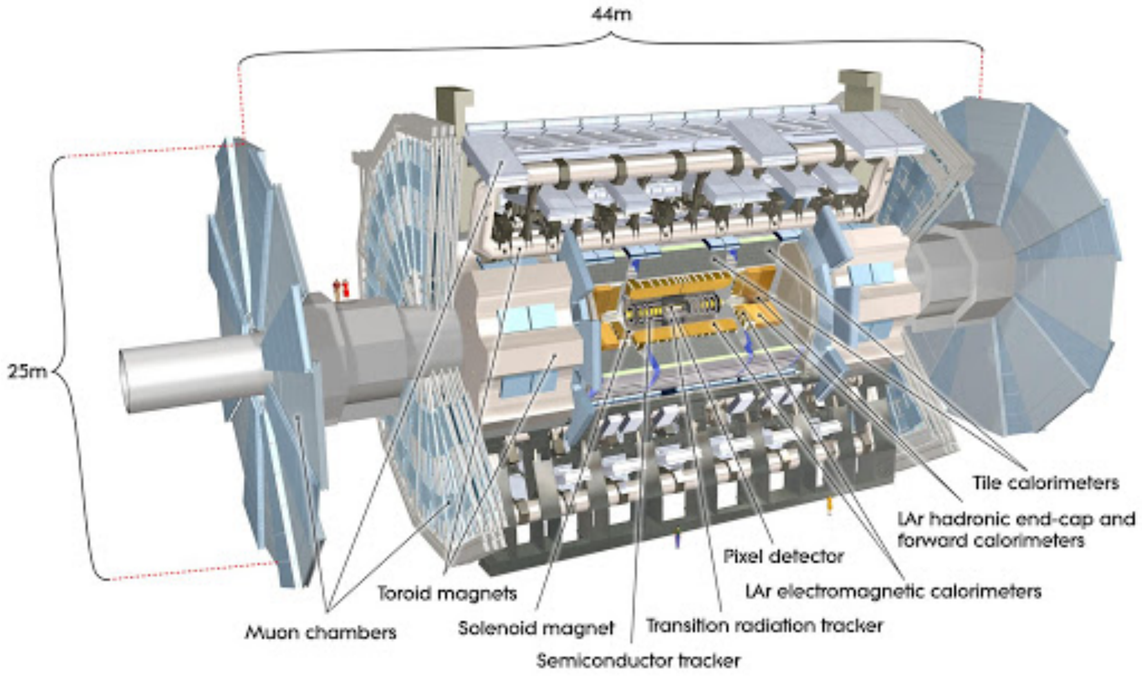
A New Era of Computing at the Large Hadron Collider

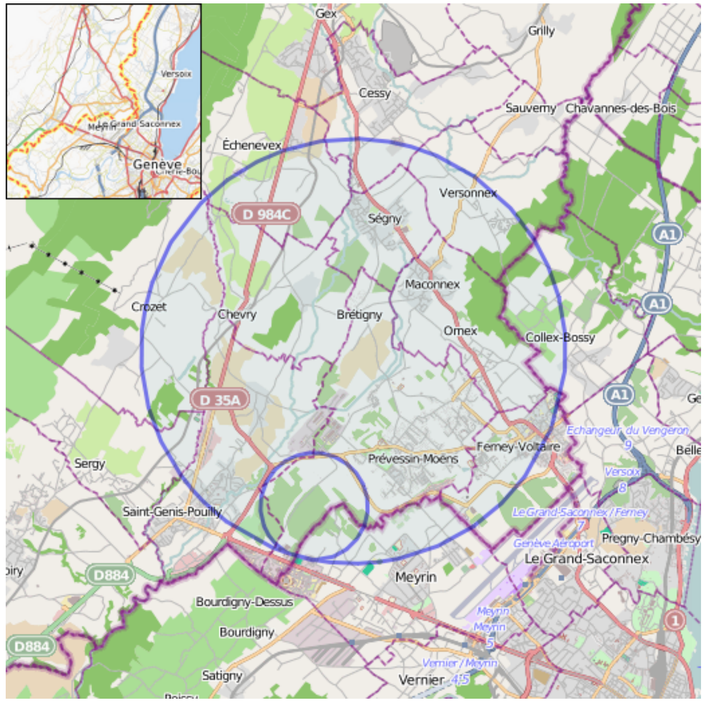
Mark Hodgkinson
University of Sheffield
17 March 2022

Contents

- Introduction to High Energy Physics at the Large Hadron Collider
- Porting code to GPU and portability layers for GPU.
- Machine Learning
- Software Training
- Conclusions

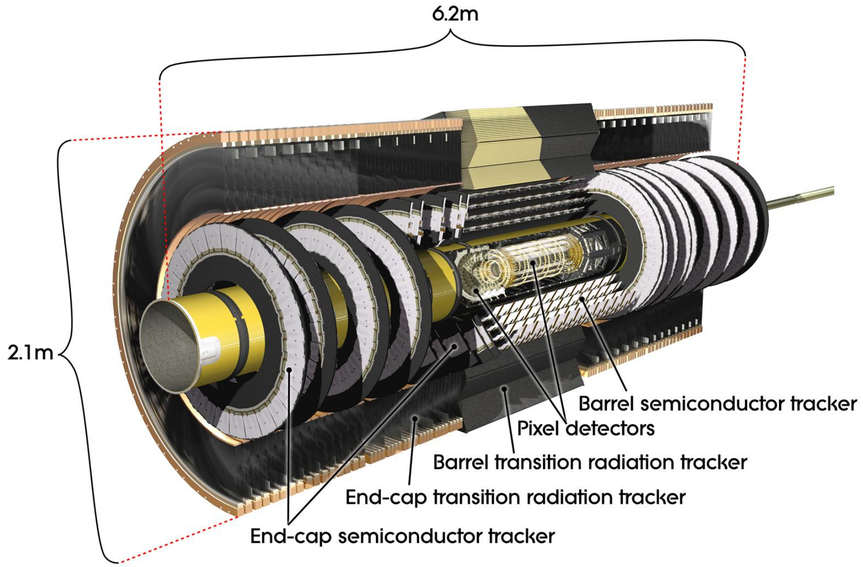
Introduction

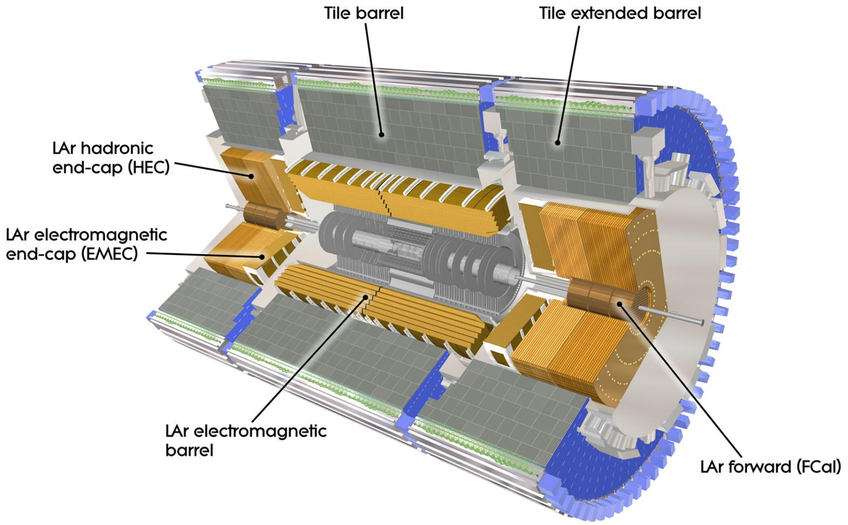




- ATLAS one of four main experiments at CERN Large Hadron Collider (LHC)
  - LHC collides bunches of protons together - the resulting interactions produce particles of interest such as the Higgs Boson for further study.
  - ATLAS uses a number of different particle detection techniques.

Particle Detection



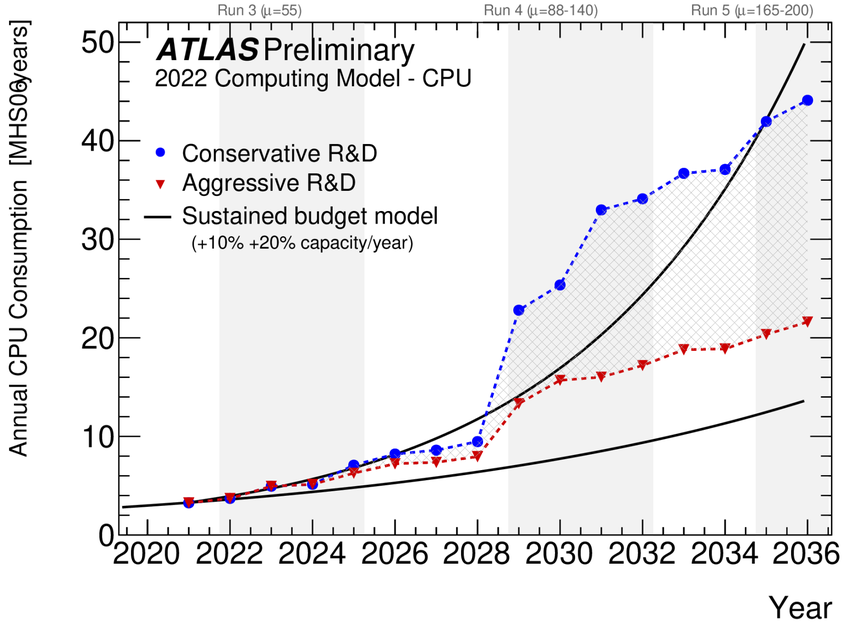


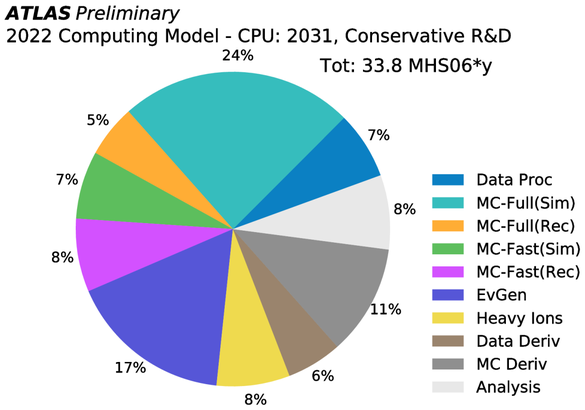
- Basic idea is particles pass through modules and “hit” them - software pattern recognition puts the “hits” together.
- Example of detecting an electron
  - Electrically charged particles interact with modules in the Inner Detector (ID), which is immersed in a magnetic field. Software looks for sets of hits that are consistent with a charged particle trajectory in that magnetic field.
  - All particles interact with modules in Calorimeter Detectors. Software looks for patterns of contiguous groups of Calorimeter Cells (modules) consistent with a particle interacting in that region. Can also detect an electron (or an electrically neutral photon).
- Such algorithms should run both “online” and “offline”.

ATLAS Software

- Can divide software up into different steps.
- We cannot process most of our data
  - Employ a “trigger” system which decides which data to keep and which to discard (irreversible decision!).
  - That which we keep is processed “offline” later on. The decision involves “online” software (often is the same c++ as offline, but with a different configuration chosen at runtime via python configuration system).
- We also simulate our data with Monte Carlo techniques
  - Simulation software is also a big resource user and new techniques can help here, as well as in our online/offline data processing software.

HL-LHC





- If we do nothing, CPU consumption needs will become inconsistent with hardware that is available.
  - Already introduced CPU level threading for LHC Run 3 at ATLAS (2022-2025).
  - After that you can see divergence in top right plot!
  - Active program to understand which techniques can solve this - for the most part this can be divided into porting “classical” algorithms from CPU to GPU and replacing “classical” algorithms with Machine Learning.
  - People also looking into porting onto FPGA and even Quantum Computing with Machine Learning.
  - LHC community recently published roadmap to HL-LHC software - includes many deliverables for software described in Christos talk (basically what to do beyond MT on CPU)

Porting Classical algorithms to GPU

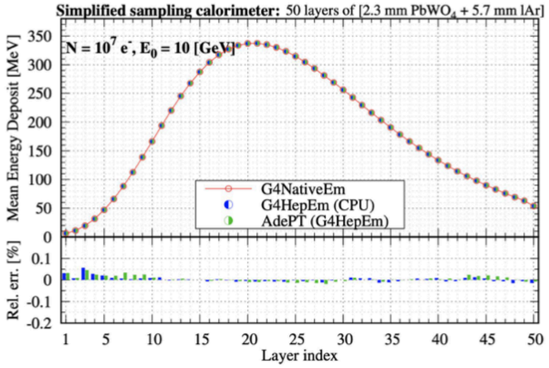
AdePT: Baseline performance

- TestEm3: 50 layer Pb/lAr sampling calorimeter
  - *Zero B-field, 10000 10GeV e-*
  - *Particles processed on GPU in batches of 26*
- AMD Ryzen 9 3900 (12C/24T), GeForce RTX2070 Super (J. Hanhfeld, SFT R&D Meeting, 20 April 2021)
  - *AdePT roughly same runtime as Geant4 with 24 Threads*
  - ***Caveat: AdePT/G4HepEM did not include "safety" geometric cut (MSC)***
  - ***Discussions with Celeritas on common benchmark problems, metrics, software/hardware setups***
- **VecGeom is significant limiter (similar observations in Celeritas)**
  - *Low device occupancy, as low as 10% in extreme cases*
  - *Large thread divergence, serialized calls to solids*
  - *Virtual function calls prevent compiler optimizations*
  - *Design, primarily virtual functions, not optimal for portability to, e.g. SYCL*

10

Adept: G4HepEM Physics Validation

- TestEm3: 50 layer Pb/lAr sampling calorimeter
  - *B-field: 0T or 1T(*) constant*
- Comparison between
  - *Geant4 (CPU)*
  - *Geant4 w/G4HepEm (CPU)*
  - *AdePT+G4HepEM (GPU)*
- (*) couple parts-per-thousand discrepancy present in layers with highest energy deposition, number of particles
- Validation also in progress using CMS geometry, currently validated number of secondaries



M.Novak, Geant4 Workshop, Sep 2021
J. Hanhfeld, M. Novák, SFT R&D Meeting, 23 March 2021

8

- Studies of common simulation problem found blockers in some particular libraries (left)
  - Independent efforts (Adept and Celeritas) able to verify they see the same issues.
- Important to verify physics output matches existing output, already pretty close with current setup.
- Further details in B. Morgans talk (U. Warwick) at UK SWIFT-HEP Workshop, November 2021.

Porting Classical algorithms to GPU

```
__global__ void initialize_kn(ParamPointers const   params,
                              StatePointers const   states,
                              InitialPointers const init)
{
    // Grid-stride loop, see
    for (int tid = blockIdx.x * blockDim.x + threadIdx.x;
         tid < static_cast<int>(states.size());
         tid += blockDim.x * gridDim.x)
    {
        ParticleTrackView particle(
            params.particle, states.particle, ThreadId(tid));
        particle = init.particle;

        // Particles begin alive and in the +z direction
        states.direction[tid] = {0, 0, 1};
        states.position[tid]  = {0, 0, 0};
        states.time[tid]      = 0;
        states.alive[tid]     = true;
    }
}
```

```
using namespace alpaka;

//Define shortcuts for some alpaka items we will use
using Dim = dim::DimInt<1>;
using Idx = uint32_t;
//Define the alpaka accelerator to be Nvidia GPU
using Acc = acc::AccGpuCudaRt<Dim,Idx>;

struct initialize_alpaka{
  template <typename Acc>
  ALPAKA_FN_ACC void operator()(Acc const &acc,ParamPointers const params,StatePointers const states,InitialPointers const init) c
    for(int tid = idx::getIdx<Grid, Threads>(acc)[0];tid < static_cast<int>(states.size());tid += blockDim.x * gridDim.x){
      ParticleTrackView particle(params.particle, states.particle, ThreadId(tid));
      particle = init.particle;

      // Particles begin alive and in the +z direction
      states.direction[tid] = {0, 0, 1};
      states.position[tid]  = {0, 0, 0};
      states.time[tid]      = 0;
      states.alive[tid]     = true;
    }

  }
};
```

- People have been asking the question - what if we need to use GPU not made by NVIDIA?
  - Portability may be the answer - many choices exist such as Alpaka, Intel OneAPI etc
- At Sheffield I studied Alpaka, using simulation code in the Celeritas project, thanks to funding from the Excalibur program.
  - As you can see above Alpaka code is more verbose than CUDA code, though the concepts are the same (copy data to/from devices, run kernels etc).
  - Ran tests on VM in STFC cloud. Installation of needed packages was done in a docker container.

Porting Classical algorithms to GPU

Setup	Call	Time
1	iterate_kn	3.79 ± 0.04 ms
1	initialize_kn	4.54 ± 0.19 μs
1	cudaMalloc	277.41 ± 6.51 s
2	iterate_kn	77.80 ± 4.41 ms
2	initialize_kn	47.46 ± 0.29 μs
2	cudaMalloc	761.13 ± 302.91 μs
3	iterate_alpaka	92.35 ± 5.91 ms
3	initialize_alpaka	261.98 ± 2.02 μs
3	cudaMalloc	805.38 ± 348.72 μs

- Three setups were tested:
  - 1 - Nominal Celeritas, 2 - Nominal Celeritas + memory copying migrated to Alpaka and 3 - Nominal Celeritas + memory copying migrated to Alpaka + kernel execution migrated to Alpaka
- Uncertainty defined as fastest minus slowest of 5 runs
- Clearly there is an overhead when using Alpaka - not being a GPU expert I could not explain exactly why, but naively (to me) adding more layers of code might cause such an issue:
  - Representative of CMS collaboration (similar experiment to ATLAS on the LHC) confirmed in a more realistic example they have setup sees a factor of 2 or so slowdown w.r.t native cuda when using Alpaka (such differences are problem specific).
  - Real question may not be how much one slows down, but whether one gains enough over CPU only options and over CPU + GPU NVIDIA only options (given one then gets access to other vendor GPU resources).

Porting Classical algorithms to GPU

HEP-CCE

Patatrack : Results

- Versions:
  - Direct: CPU, CUDA, HIP
  - Kokkos: CPU (Serial, POSIX Threads), CUDA, HIP
  - Alpaka: CPU (Serial, TBB), CUDA
    - Developed by CERN group, first results shown in ACAT21 poster
- Snapshot of performance comparison of direct vs Kokkos versions on Cori GPU
  - Intel Xeon Gold 6148 (Skylake, 20 cores, 2 threads/core) + NVIDIA V100

Running 1-thread processes	Direct CPU	Kokkos Serial
1 process (node free)	25.2 ± 0.4 events/s	23.9 ± 0.4 events/s
40 processes (full socket)	460 ± 10 events/s	260 ± 10 events/s

1 process	Direct CUDA	Kokkos CUDA
1 concurrent event	891 ± 5 events/s	582 ± 6 events/s
3 concurrent events	1725 ± 4 events/s	996 ± 4 events/s
7 concurrent events	2202 ± 9 events/s	985 ± 1 events/s

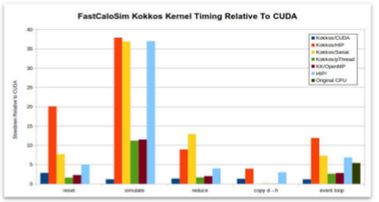
- Showed also in vCHEP21 that the throughput with CUDA Unified Memory was about 3x smaller than with explicit memory management

12

U.S. DEPARTMENT OF ENERGY | Office of Science | Argonne NATIONAL LABORATORY | Brookhaven National Laboratory | Fermilab | BERKELEY LAB Bringing Science Solutions to the World

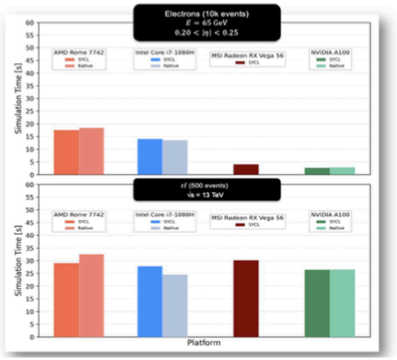
HEP-CCE

FastCaloSim : Results



Kokkos

- exercised all backends: CUDA, HIP, Intel, pThread, OpenMP
- Kokkos/CUDA performs similarly to pure CUDA
  - 5x faster event loop for 65GeV electrons
  - 40x faster for 4TeV electrons
- increased penalties from launch latencies and memory init
- hip/AMD considerably slower than CUDA
  - Kokkos/HIP performs similarly to HIP
- Kokkos/OMP has 2.5x perf of original CPU at 12 threads



SYCL

- Ten 'runs' of single-electron and top quark pair production simulations
  - AMD CPU host backend (TBB, on OpenCL)
  - Intel CPU with OpenCL backend
- ~4x faster than CPU for single electrons when executed using GPU offload
- Top quark simulations achieve no gains on GPU due to lack of inter event parallelism and run-time loading of parametrizations on host (more secondaries)

Same source runs on 4 different platform

17

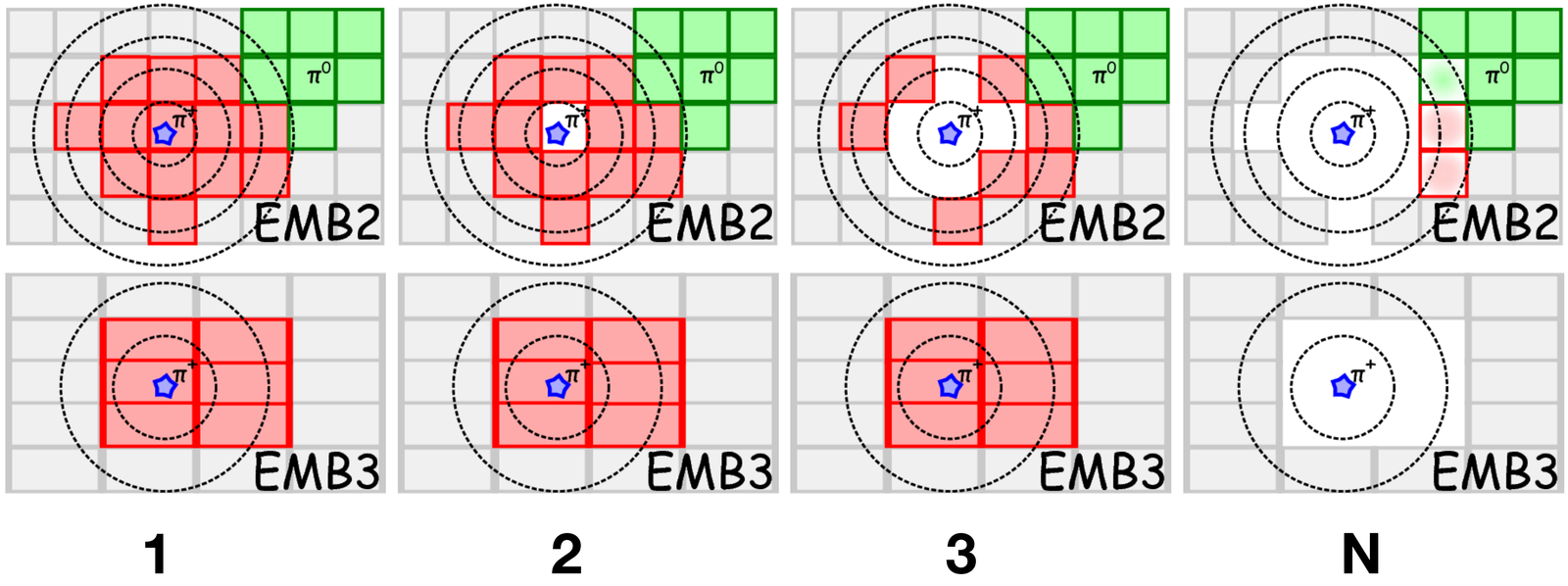
U.S. DEPARTMENT OF ENERGY | Office of Science | Argonne NATIONAL LABORATORY | Brookhaven National Laboratory | Fermilab | BERKELEY LAB Bringing Science Solutions to the World

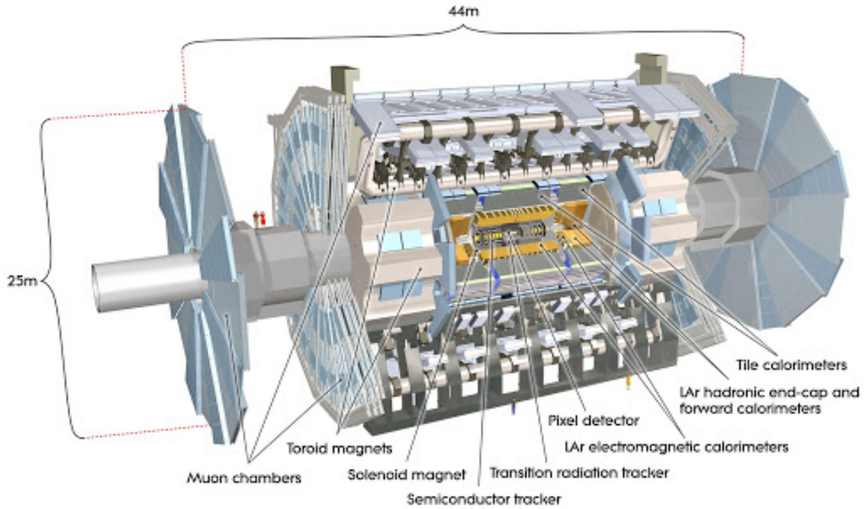
- Taken from C. Leggets (Lawrence Berkeley National Lab) talk at CERN Compute Accelerator Forum (HCAF), Feb 2022.
  - Performance depends on the task.
  - In general can see some performance penalties when using portability layers.
  - Pattern recognition problem in data processing (left) and simulation problems (right).

Machine Learning

- HEP has long used machine learning, though recently has it become far more prevalent.
  - Even 20 years ago people looked into used neural nets (via CERN ROOT TMVA software), though often disfavoured because people did not like “black boxes” (this was my experience during my PhD).
  - Boosted Decision Trees were very popular for binary classification tasks in the 2010’s.
  - Nowadays many people using keras etc to train deep neural nets etc.

Particle Flow

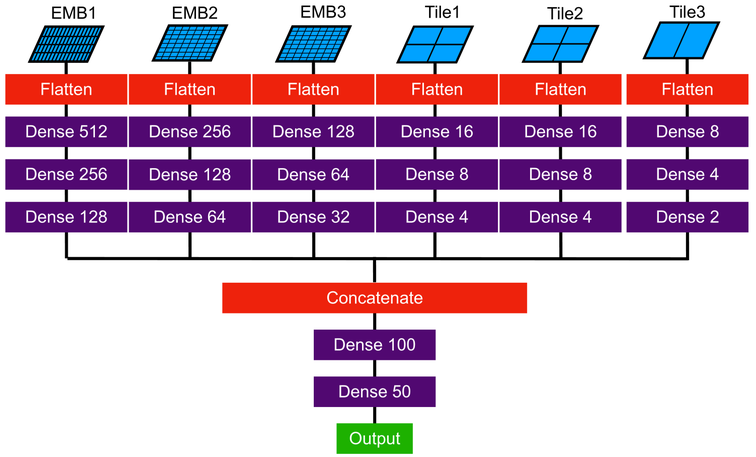




- In ATLAS Technical Design Report (1999) it was envisaged jets would be measured only using the calorimeters.
  - Particle Flow improves this measurement by matching the Inner Detector (ID) and Calorimeter measurements of charged particles - precision on ID measurement is better and ID allows to match charged particle measurements with a specific collision within the proton bunch and hence can reject charged particles from the “wrong” collision - typically ATLAS sees one interesting collision (“hard scatter” which might e.g produce a Higgs Boson) and many other collisions (“pileup”) at the same time.
  - Particle Flow uses knowledge of how charged pions interact in the calorimeters (e.g shape and symmetry of energy deposits) to swap the measurement for the ID measurement.
  - When Calorimeter is unfolded in angular coordinates this looks like a 2D image with pixels.

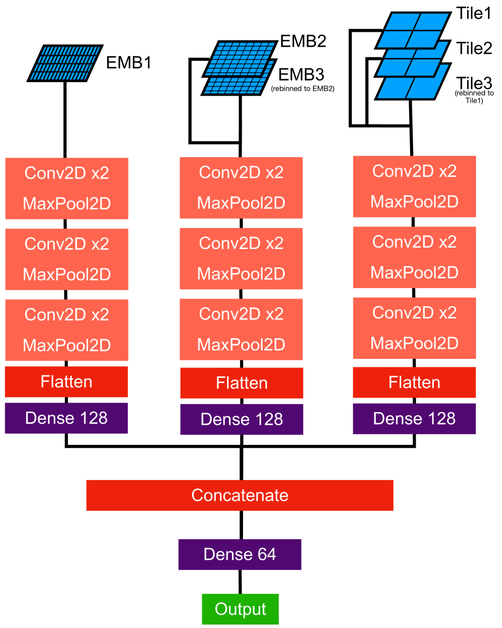
Machine Learning

DNN Classifier

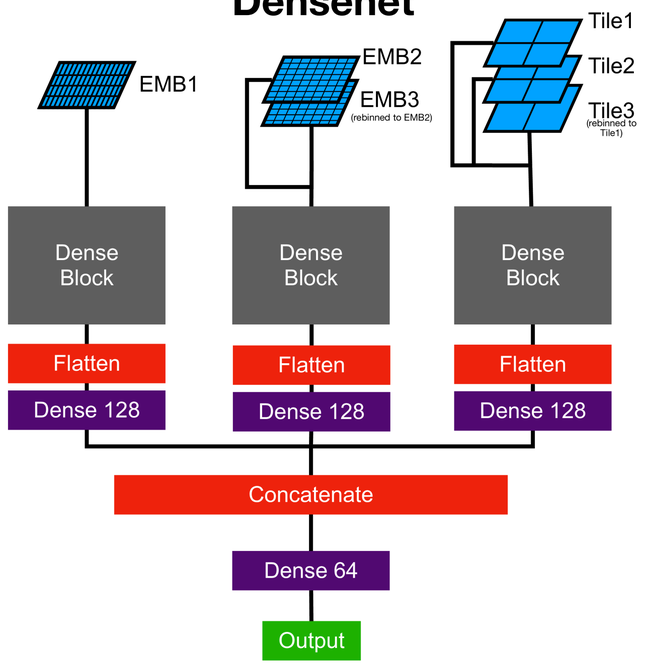


ATL-PHYS-PUB-2020-018

CNN Classifier



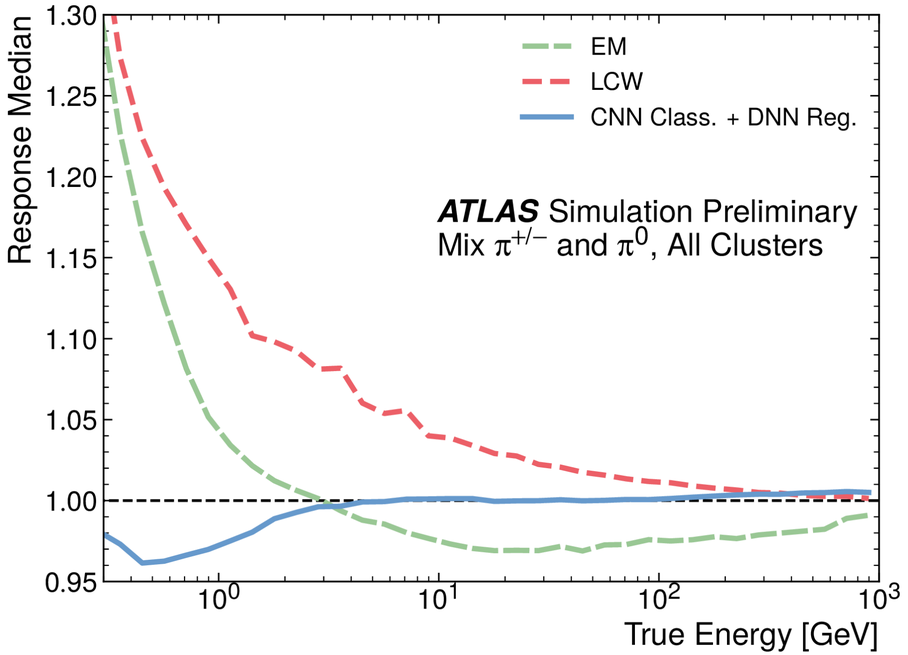
Densenet

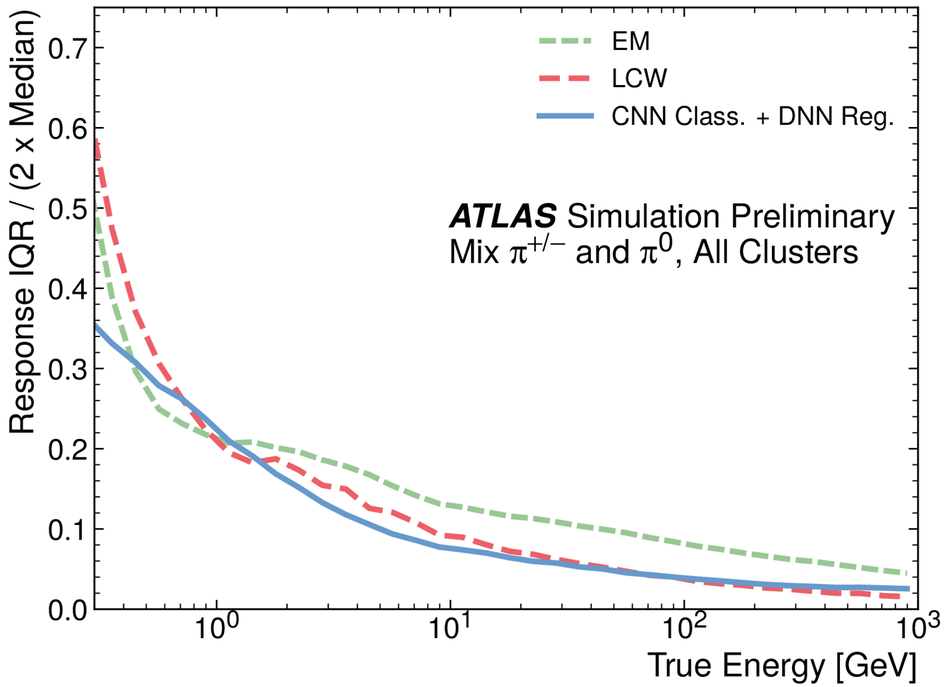


- Calibration - correct energy measurements in calorimeter for things we cannot measure (neutrino particles do not interact, interactions in “dead material” where we run cabling to read out calorimeter modules, etc).
- Deep Neutral Network (DNN), Convolutional Neural Network (CNN) and Densely Connected Convolution Network (DenseNet) have been studied.
- Currently ATLAS LCW Calibration scheme uses a Likelihood:
  - Classification step using Likelihood ratio, making use of the cluster energy, eta position, longitudinal depth and average cell energy density.
  - Calibration step deploys calorimeter cell signal weighting which depend on cluster energy and location.
  - The Machine Learning schemes also do both classification and regression.

Machine Learning

ATL-PHYS-PUB-2020-018





- Combined classification and regression test:
  - Compare LCW to combination of CNN Classifier (best) and DNN regression (best)
  - High performance of CNN classifier ensures that the correct energy regression is applied in this mixed particle sample.
  - Good performance means a value of 1 in the left plot and the lowest possible value in the right plot.
    - Response is ratio of measured energy to the known true energy deposit in a simulation (ideally exactly 1).
    - Resolution is a measure of the precision of this measurement (ideally zero - i.e we always measure a value of 1).
- Currently we (Sheffield) are working with Lancaster to add these techniques into the particle flow algorithm we developed, which ATLAS currently uses.
  - Funded through STFC IRIS. Unfortunately ATLAS has not yet produced public results, so can't show any details.
  - Other architecture such as Graph Neural Network (GN) under study.

Quantum Computing with Machine Learning

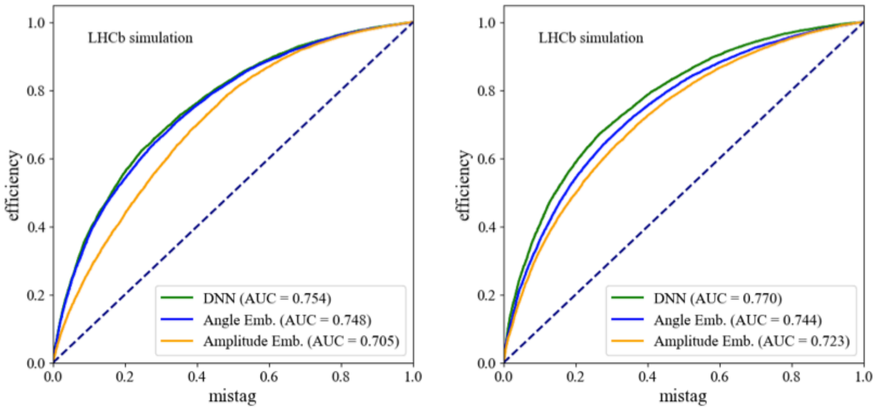


Figure 5: ROC distributions and AUC score for DNN (green), Angle Embedding (blue) and Amplitude Embedding circuits (yellow) for the *muon dataset* (a) and the *complete dataset* (b). The dashed line represents a random classifier.

- Standard DNN (green) comparable with QML (blue)
- Proof that in principle you can get the same performance as DNN
  - Presumably then the question is what you gain from QML over ML on GPU?
  - The paper emphasises fewer training events were needed on the QML to get the same performance.

Let's start with a simple example circuit that generates a two-qubit entangled state, then evaluates the expectation value of the Pauli-Z operator on the first wire.

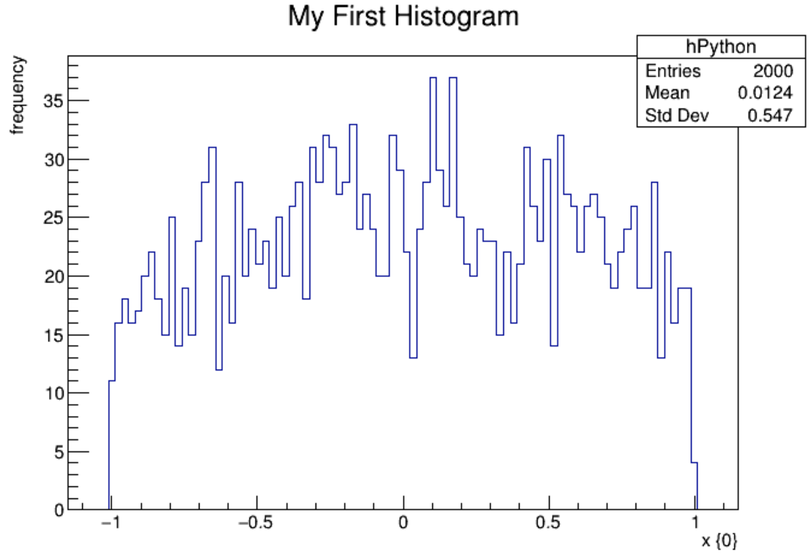
```
@qml.qnode(dev, interface="jax")
def circuit(param):
    # These two gates represent our QML model.
    qml.RX(param, wires=0)
    qml.CNOT(wires=[0, 1])

    # The expval here will be the "cost function" we try to minimize.
    # Usually, this would be defined by the problem we want to solve,
    # but for this example we'll just use a single PauliZ.
    return qml.expval(qml.PauliZ(0))
```

- Doubtful (my opinion) this is something we will be using in 10 years, but I include it given a few groups have started to study usage.
  - e.g this preprint uses LHCB (another large scale LHC experiment) data to compare existing NN with NN on simulated quantum computers for a task called "b-tagging" - essentially a binary classification task. Is a measurement of a quark traversing the detector of one type (b) or another (light)?
  - They used pennylane to simulated quantum circuits and google jax to do the ML with that quantum circuit. Looking at the pennylane tutorials it appears straightforward to use.
  - Standard DNN uses keras.

University Training

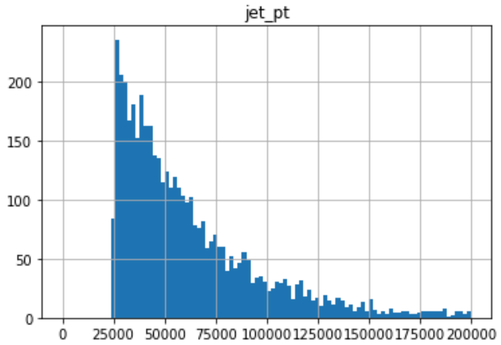
```
[4]: #Now lets try the same in python
#In the python case we have to explicitly ask to import the classes (TH1F, TCanvas) that we will use.
#We did not have to do anything equivalent in the C++ example above, executed in CLING c++ - CLING C++ is aware
#of the ROOT libraries by default, python is not aware by default (hence the need for import statements).
from ROOT import TH1F, TCanvas
histPython = TH1F("hPython","My First Histogram;x {0}; frequency",100,10,10)
histPython.FillRandom("gaus",2000);
canPython = TCanvas()
histPython.Draw()
canPython.Draw()
```

```
Info in <TH1F::FillRandom>: Using function axis and range [-1,1]
```



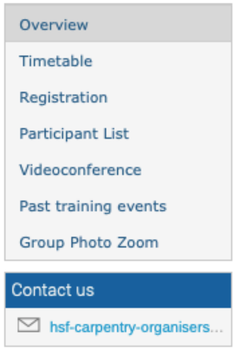

```
[10]: miniTreeTTbar = uproot.open("http://opendata.atlas.cern/release/samples/MC/mc_117049.ttbar_had.root")["mini"]
df_ttbar = miniTreeTTbar.arrays(["alljet_n","jet_pt"],library='pd',entry_stop=1000)
df_ttbar.hist("jet_pt",bins=100,range=[0,200000])
df_ttbar.hist("alljet_n",bins=10)
```

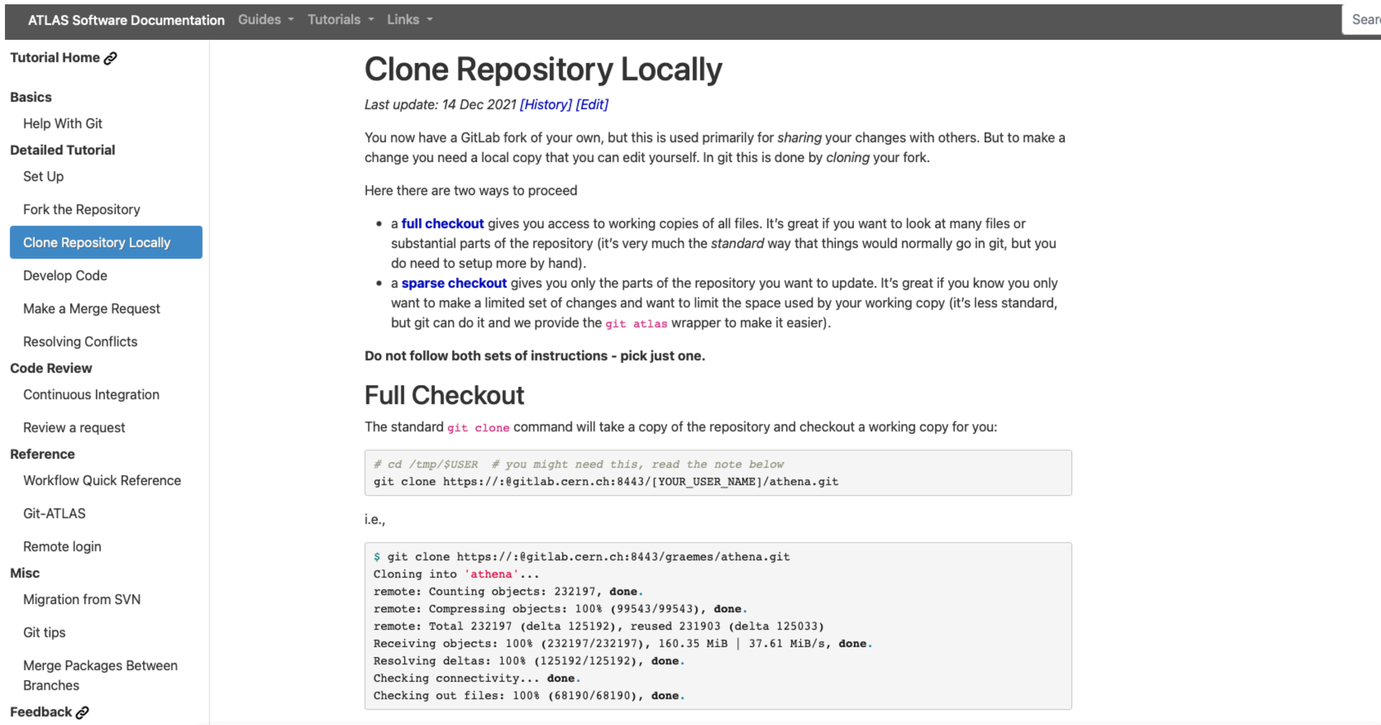
```
[10]: array([[<AxesSubplot:title={'center':'alljet_n'}>]], dtype=object)
```



- HEP has a lot of training resources available.
  - The first one most people (as PhD students) would encounter are lectures/tutorials from their own university HEP groups.
  - At Sheffield for example we use Jupyter notebooks (available in GitHub) that I wrote to illustrate how to use ROOT (CERN software for making histograms from HEP specific ROOT file format), standard python tools (such as pandas, numpy) and to get started in ML (with python keras).
  - Run them in my mybinder.org (ok for lightweight tasks to learn technical details of how to do things - e.g to learn usage of NN have to massively restrict amount of data to train on if you want reasonable turnaround).
  - Make use of ATLAS open data to teach.

HSF Training





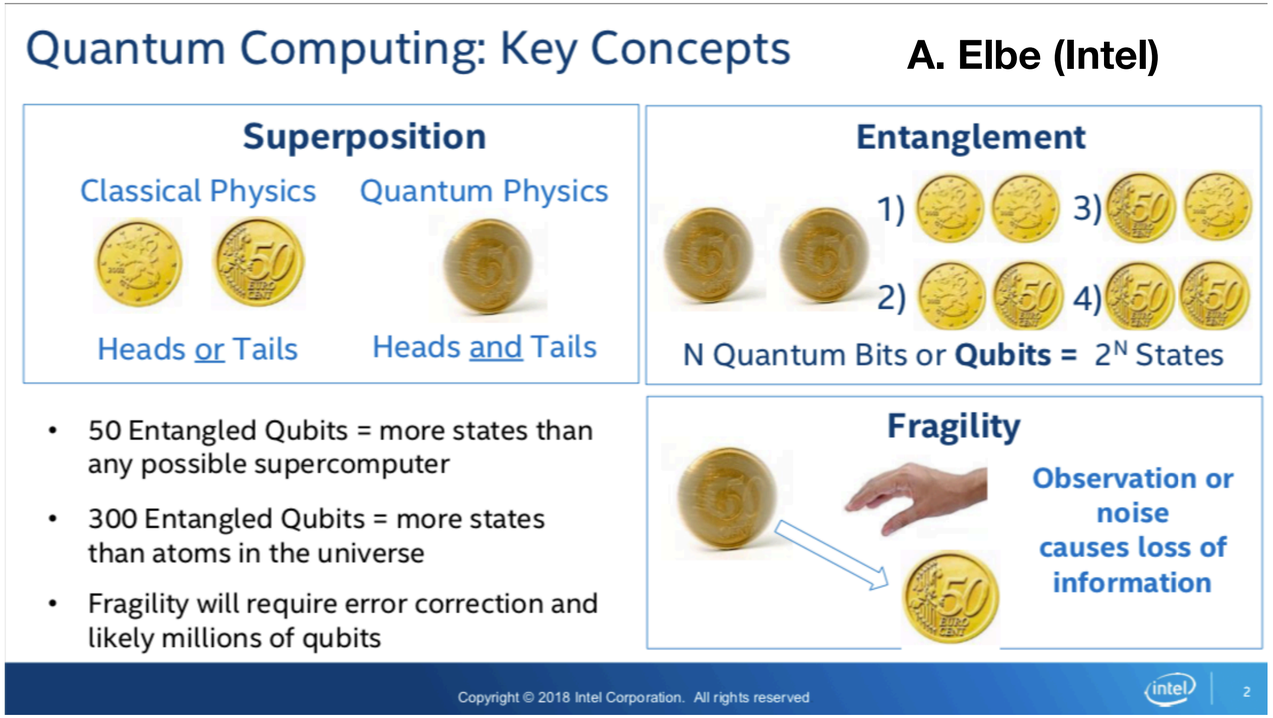
- HEP Software Foundation (HSF) provides general training material (left) for people to use.
- Students also attend annual Rutherford Appleton Laboratory lectures on computing technologies and ML (started a few years ago).
- On ATLAS they get a week of tutorials about how our C++ software works, how to use git etc.
  - ATLAS also maintains public documentation on many typical tasks ATLAS git workflow (right).
- CERN Openlab provides general training from third parties, e.g this upcoming tutorial on usage of Intel Software Tools.
- Also make use of Sheffield RSE courses (e.g I have attended their Nvidia ML course some years ago).

Conclusions

- We saw an overview of the software challenges for the future HL-LHC
- Many R&D studies underway to port algorithms to GPU (and FPGA) and study usage of portability layers.
- Machine Learning already extensively used in HEP, potentially many more gains to be had in HL-LHC by moving more and more classical algorithms to ML based approaches.
  - People even thinking very long term and starting to study ML on quantum computers.
- Extensive training provided within the HEP community.

Extras

Quantum Circuits



- Build circuits from qubits, create quantum logic gates - some qubits reserved for error correction - how many depends on the hardware
- Many commercial systems (using different technologies) - IBM, Intel, Google, Microsoft etc
- D-Wave has some nice explanations of how Quantum Computing (a particular type called annealing) works.
  - Classical algorithms may search for lowest valley in a minimisation problem, but might not reach the global minimum.
  - Quantum annealing allows to occupy many coordinates simultaneously, to quantum tunnel between valleys and to use quantum entanglement to see correlations that lead to the deepest valley.

CCE/PPS: Software Support Chart

HEP-CCE

	OpenMP Offload	Kokkos	dpc++ / SYCL	HIP	CUDA	Alpaka
NVIDIA GPU			*codeplay and intel/llvm*			
AMD GPU		*experimental (feature complete)*	*via hipSYCL and intel/llvm*			
Intel GPU		*prototype*		HIPLZ: *very early development*		*prototype*
CPU						
Fortran						
FPGA						*possibly via SYCL*

Supported
Under Development
3rd Party
Not Supported

time to add python!

Platform support still a moving target: this chart is updated often!

9

U.S. DEPARTMENT OF ENERGY | Office of Science | Argonne NATIONAL LABORATORY | Brookhaven National Laboratory | Fermilab | BERKELEY LAB Bringing Science Solutions to the World

- Taken from C. Leggets talk at CERN HCAF, Feb 2022.