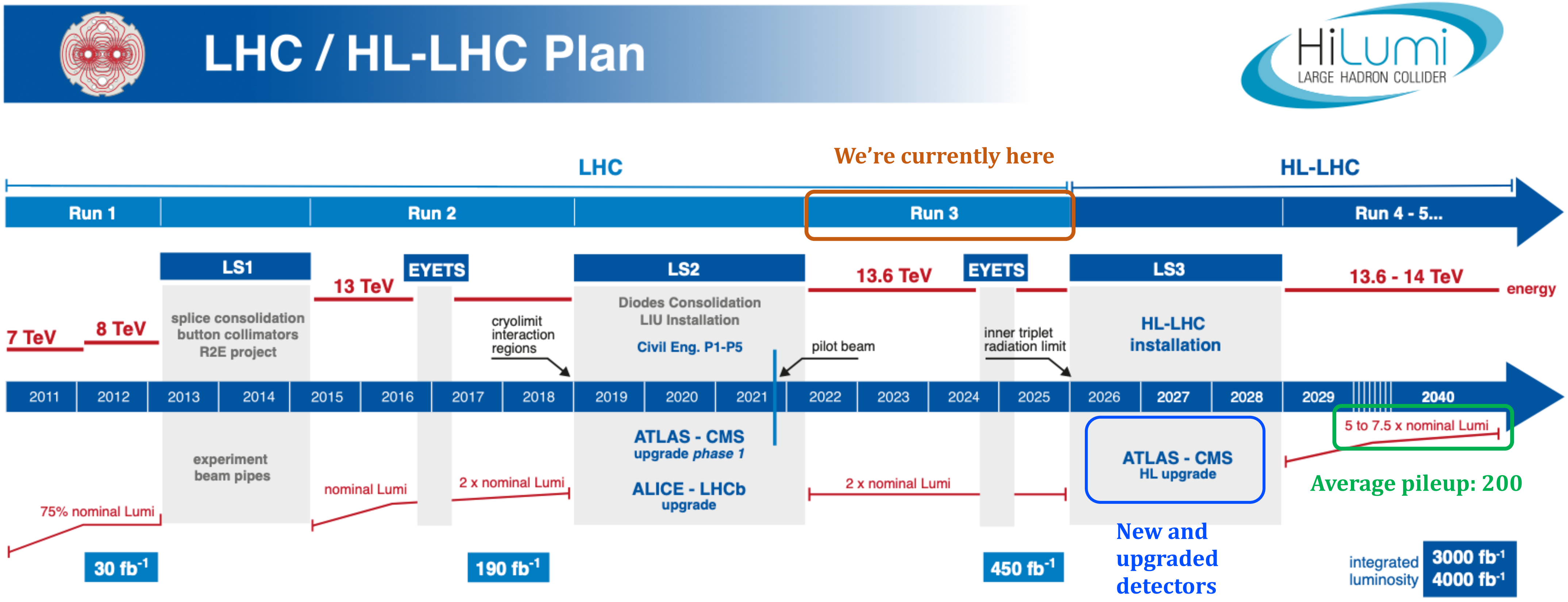
Performance study of the CLUE algorithm with the alpaka library

Andrea Bocci, Tony Di Pilato*, Luca Ferragina, Matti Kortelainen, Juan Jose Olivera Loyola, Felice Pantaleo, Aurora Perego, Marco Rovere, Wahid Redjeb

ACAT 2022

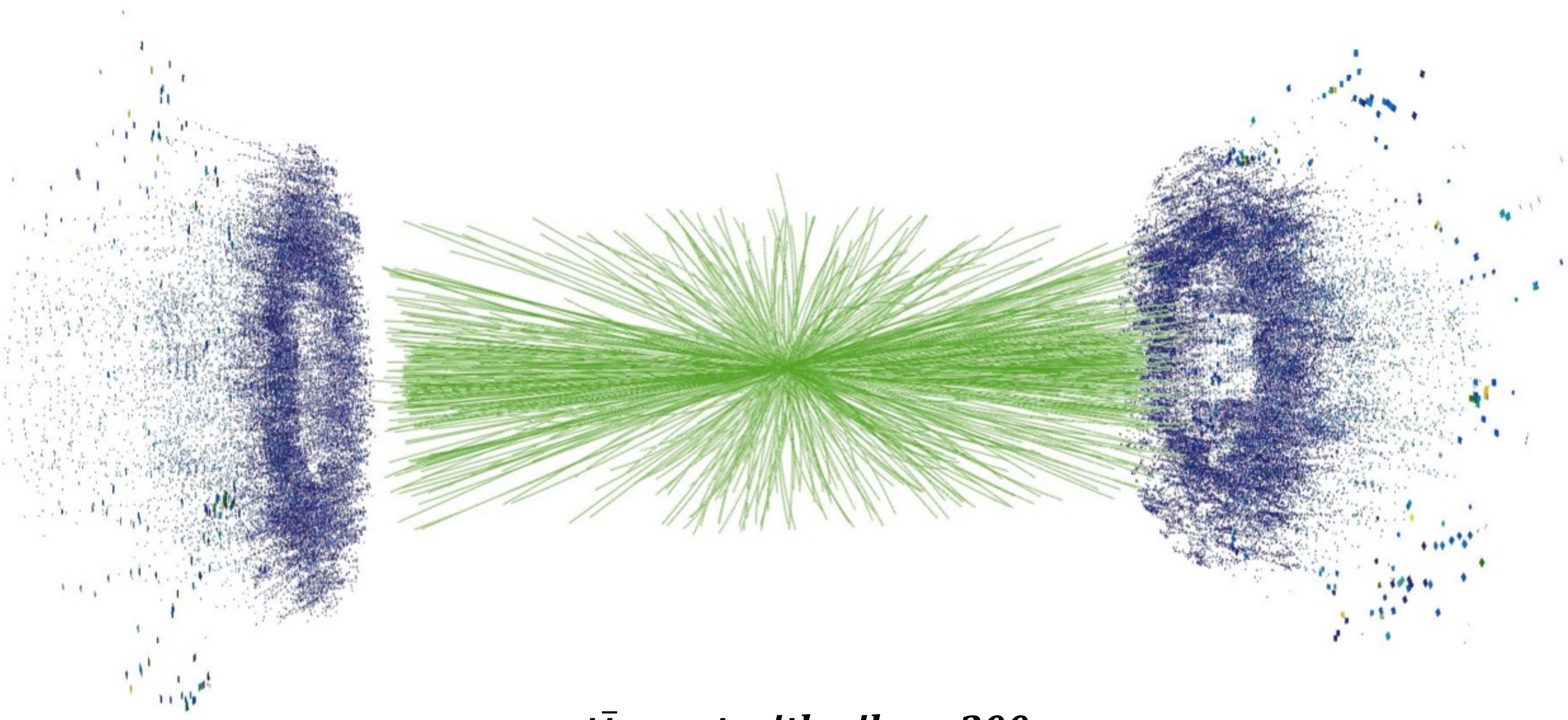
BARI

High Luminosity LHC

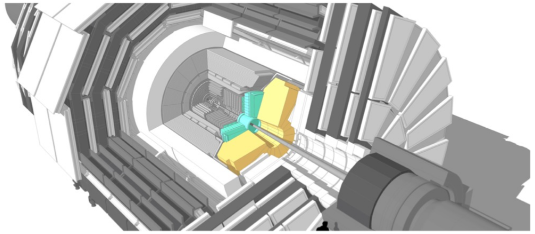


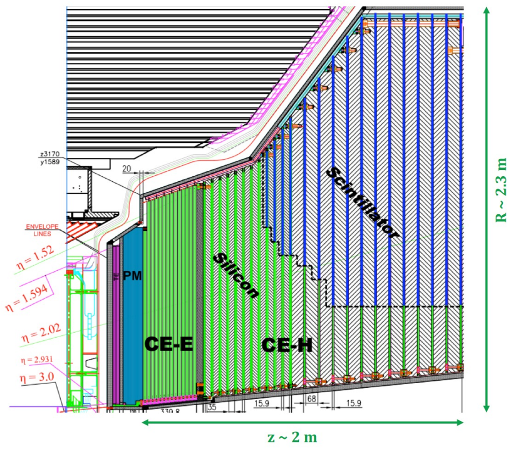
Discovering new physics and performing more accurate measurements due to the improved sensitivity level…

The CMS Phase-2 challenge



$t\bar{t}$ *event with pileup 200*





Future High Granularity Calorimeter for the CMS experiment

The software reconstruction challenge

Software reconstruction: digital signals in each detector must be processed to provide information about particles produced in the proton-proton collisions and successive decays and interaction with the absorber material.

- In the PU200 scenario, such a task becomes much **harder**
  - **Massive amount of computing resources** required
    - Advent of **heterogeneous computing**!

The heterogeneous computing scenario

Modern computing farms and data centers rely on ***heterogeneous architectures***

- CPU
- GPUs → ***hardware accelerators***

➢ ***HEP approach: offloading part of the reconstruction to GPUs for parallel execution***

❖ **Many vendors → many programming languages → many versions of the same code!!!**

NVIDIA CUDA · AMD ROCm · OpenCL

Performance portability with alpaka

- Performance portability libraries have become an interesting solution
  - Write code once
  - Compile for different backends
  - Execute on target platform
  - Not all the technologies provide close-to-native backend performance
- Portable code can be ***easily maintained*** and support new accelerators
- ***CMS choice for Run 3:*** alpaka

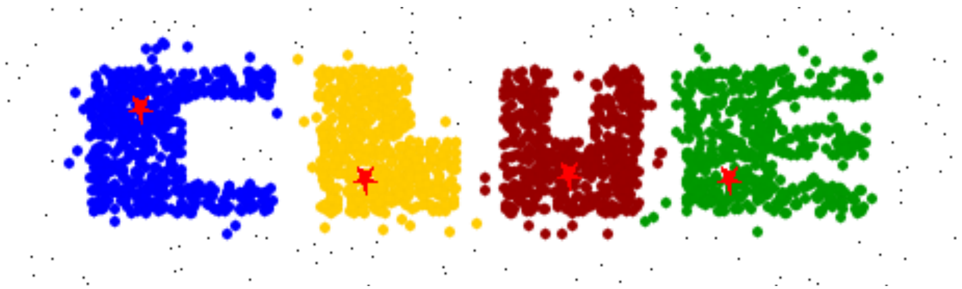
Alpaka

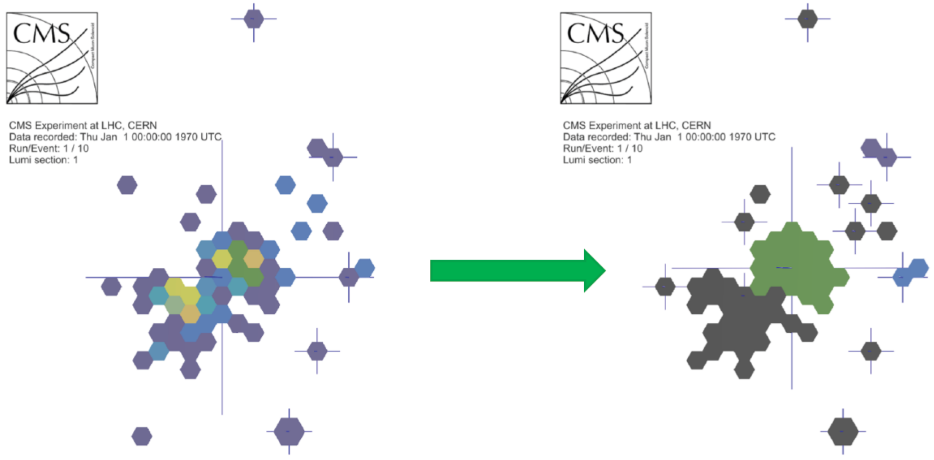
- **A**bstraction **L**ibrary for **Pa**rallel **K**ernel **A**cceleration
  - Developed and maintained at **HZDR** (Helmholtz-Zentrum-Dresden-Rossendorf) and **CASUS** (Center for Advanced Systems Understanding)
- C++ header-only library (*currently on C++17*)
- Supports a wide range of compilers (g++, clang, …)
- Several backends supported
  - CPU serial and parallel execution (std::thread or TBB)
  - NVIDIA GPU (CUDA)
  - AMD GPU (HIP/ROCm)
  - Intel GPU and FPGAs (SYCL) *under development*
- For more information, check **Jan Stephan's poster** "**Performance portability with alpaka**" on Thursday

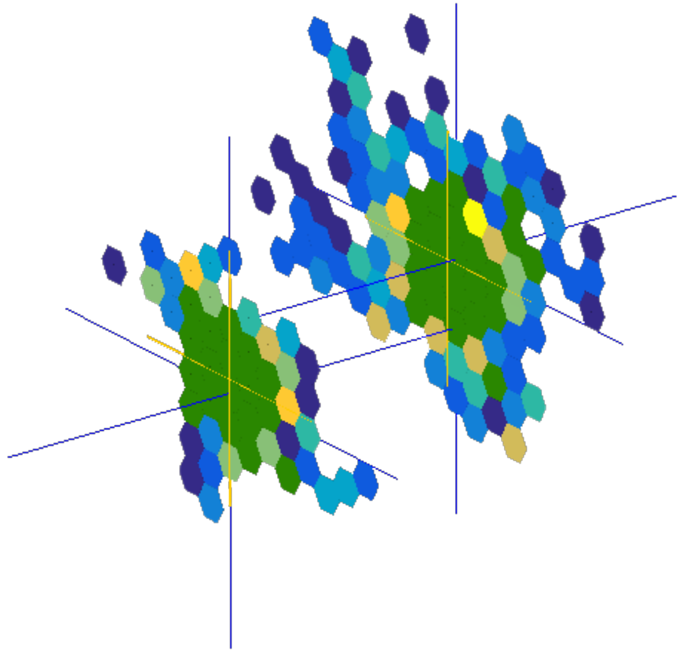
A real application: the CLUE algorithm

CLUstering of Energy (CLUE): fast 2D clustering algorithm developed for the future CMS-HGCAL detector

- Based on energy density
- Builds small clusters (~10 RecHits)
- **Fully ported to GPU (CUDA)**
- Uses a **tiled** data structure that fully exploits the detector granularity and allows fast querying of neighbor cells

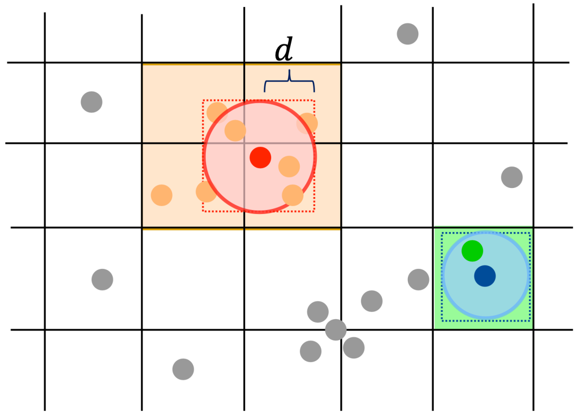






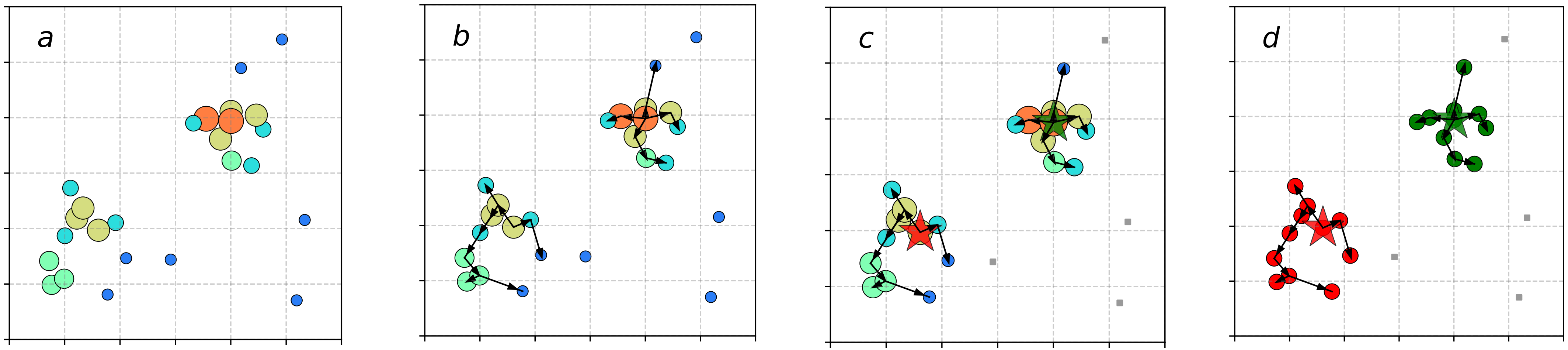
M. Rovere, Z. Chen, **A. Di Pilato**, F. Pantaleo, C. Seez, *CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High Energy Physics*, Frontiers in Big Data, **3**, 2020.

CLUE procedure



Step 0: *arrange input data in "tiles" (spatial indexing)*
a. *calculate local energy density*
b. *find nearest higher and calculate its distance*
c. *find seeds and outliers*
d. *assign cluster indices*

Each of these steps can be written as a function (or kernel) and perform the same operation on each point



M. Rovere, Z. Chen, **A. Di Pilato**, F. Pantaleo, C. Seez, *CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High Energy Physics*, Frontiers in Big Data, **3**, 2020.

Porting CLUE from CUDA to Alpaka - 1

```cpp
class CLUEAlgoCUDA {
public:
  // constructor
  CLUEAlgoCUDA() = delete;
  explicit CLUEAlgoCUDA(float const &dc, float const &rhoc, float const &outlierDeltaFactor, cudaStream_t stream)
      : d_points{stream}, dc_{dc}, rhoc_{rhoc}, outlierDeltaFactor_{outlierDeltaFactor}, stream_{stream} {
    init_device();
  }

  ~CLUEAlgoCUDA() = default;

  void makeClusters(PointsCloud const &host_pc);

  PointsCloudCUDA d_points;

  LayerTilesCUDA *hist_;
  cms::cuda::VecArray<int, maxNSeeds> *seeds_;
  cms::cuda::VecArray<int, maxNFollowers> *followers_;

private:
  float dc_;
  float rhoc_;
  float outlierDeltaFactor_;
  cudaStream_t stream_ = nullptr;
  cms::cuda::device::unique_ptr<LayerTilesCUDA[]> d_hist;
  cms::cuda::device::unique_ptr<cms::cuda::VecArray<int, maxNSeeds>> d_seeds;
  cms::cuda::device::unique_ptr<cms::cuda::VecArray<int, maxNFollowers>[]> d_followers;

  // private methods
  void init_device();

  void setup(PointsCloud const &host_pc);
};
```

```cpp
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class CLUEAlgoAlpaka {
  public:
    // constructor
    CLUEAlgoAlpaka() = delete;
    explicit CLUEAlgoAlpaka(float const &dc,
                            float const &rhoc,
                            float const &outlierDeltaFactor,
                            Queue stream,
                            uint32_t const &numberOfPoints)
        : d_points{stream, numberOfPoints},
          queue_{std::move(stream)},
          dc_{dc},
          rhoc_{rhoc},
          outlierDeltaFactor_{outlierDeltaFactor} {
      init_device();
    }

    ~CLUEAlgoAlpaka() = default;

    void makeClusters(PointsCloud const &host_pc);

    PointsCloudAlpaka d_points;

    LayerTilesAlpaka<Acc1D> *hist_;
    cms::alpakatools::VecArray<int, maxNSeeds> *seeds_;
    cms::alpakatools::VecArray<int, maxNFollowers> *followers_;

  private:
    Queue queue_;
    float dc_;
    float rhoc_;
    float outlierDeltaFactor_;

    std::optional<cms::alpakatools::device_buffer<Device, LayerTilesAlpaka<Acc1D>[]>> d_hist;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNSeeds>>> d_seeds;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNFollowers>[]>> d_followers;

    // private methods
    void init_device();

    void setup(PointsCloud const &host_pc);
  };
}  // namespace ALPAKA_ACCELERATOR_NAMESPACE
```

User-defined namespace that contain all the needed symbols (Platform , Device, Queue, BufferType)

Pointers to device memory passed to kernels

Executes the task (similar to cudaStream)

alpaka buffers don't have a default constructor

Porting CLUE from CUDA to Alpaka - 2

```
class PointsCloudCUDA {
public:
  PointsCloudCUDA() = delete;
  explicit PointsCloudCUDA(cudaStream_t stream, int nPoints)
      // input variables
      : x{cms::cuda::make_device_unique<float[]>(nPoints, stream)},
        y{cms::cuda::make_device_unique<float[]>(nPoints, stream)},
        layer{cms::cuda::make_device_unique<int[]>(nPoints, stream)},
        weight{cms::cuda::make_device_unique<float[]>(nPoints, stream)},
        // result variables
        rho{cms::cuda::make_device_unique<float[]>(nPoints, stream)},
        delta{cms::cuda::make_device_unique<float[]>(nPoints, stream)},
        nearestHigher{cms::cuda::make_device_unique<int[]>(nPoints, stream)},
        clusterIndex{cms::cuda::make_device_unique<int[]>(nPoints, stream)},
        isSeed{cms::cuda::make_device_unique<int[]>(nPoints, stream)},
        view_d{cms::cuda::make_device_unique<PointsCloudCUDAView>(stream)} {
    auto view_h = cms::cuda::make_host_unique<PointsCloudCUDAView>(stream);
    view_h->x = x.get();
    view_h->y = y.get();
    view_h->layer = layer.get();
    view_h->weight = weight.get();
    view_h->rho = rho.get();
    view_h->delta = delta.get();
    view_h->nearestHigher = nearestHigher.get();
    view_h->clusterIndex = clusterIndex.get();
    view_h->isSeed = isSeed.get();

    cudaMemcpyAsync(view_d.get(), view_h.get(), sizeof(PointsCloudCUDAView), cudaMemcpyHostToDevice, stream);
  }
```

```
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class PointsCloudAlpaka {
  public:
    PointsCloudAlpaka() = delete;
    explicit PointsCloudAlpaka(Queue stream, int nPoints)
        //input variables
        : x{cms::alpakatools::make_device_buffer<float[]>(stream, nPoints)},
          y{cms::alpakatools::make_device_buffer<float[]>(stream, nPoints)},
          layer{cms::alpakatools::make_device_buffer<int[]>(stream, nPoints)},
          weight{cms::alpakatools::make_device_buffer<float[]>(stream, nPoints)},
          //result variables
          rho{cms::alpakatools::make_device_buffer<float[]>(stream, nPoints)},
          delta{cms::alpakatools::make_device_buffer<float[]>(stream, nPoints)},
          nearestHigher{cms::alpakatools::make_device_buffer<int[]>(stream, nPoints)},
          clusterIndex{cms::alpakatools::make_device_buffer<int[]>(stream, nPoints)},
          isSeed{cms::alpakatools::make_device_buffer<int[]>(stream, nPoints)},
          view_d{cms::alpakatools::make_device_buffer<PointsCloudAlpakaView>(stream)} {
      auto view_h = cms::alpakatools::make_host_buffer<PointsCloudAlpakaView>(stream);
      view_h->x = x.data();
      view_h->y = y.data();
      view_h->layer = layer.data();
      view_h->weight = weight.data();
      view_h->rho = rho.data();
      view_h->delta = delta.data();
      view_h->nearestHigher = nearestHigher.data();
      view_h->clusterIndex = clusterIndex.data();
      view_h->isSeed = isSeed.data();

      alpaka::memcpy(stream, view_d, view_h);
    }
```

Allocate memory for alpaka buffers

Allocate memory for host view

Copy view from host to device

Porting CLUE from CUDA to Alpaka - 3

```
void KernelComputeHistogram(std::array<LayerTilesSerial, NLAYERS> &d_hist, PointsCloudSerial &points) {
  for (unsigned int i = 0; i < points.n; i++) {
    // push index of points into tiles
    d_hist[points.layer[i]].fill(points.x[i], points.y[i], i);
  }
};
```

CPU serial: loops over all the points

```
__global__ void kernel_compute_histogram(LayerTilesCUDA* d_hist, pointsView* d_points, int numberOfPoints) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < numberOfPoints) {
    // push index of points into tiles
    d_hist[d_points->layer[i]].fill(d_points->x[i], d_points->y[i], i);
  }
} // kernel
```

GPU CUDA: each thread execute the same instruction with a different point

```
struct KernelComputeHistogram {
  template <typename TAcc>
  ALPAKA_FN_ACC void operator()(const TAcc &acc,
                                LayerTilesAlpaka<Acc1D> *d_hist,
                                pointsView *d_points,
                                uint32_t const &numberOfPoints) const {
    // push index of points into tiles
    cms::alpakatools::for_each_element_in_grid(
        acc, numberOfPoints, [&](uint32_t i) { d_hist[d_points->layer[i]].fill(d_points->x[i], d_points->y[i], i); });
  }
};
```

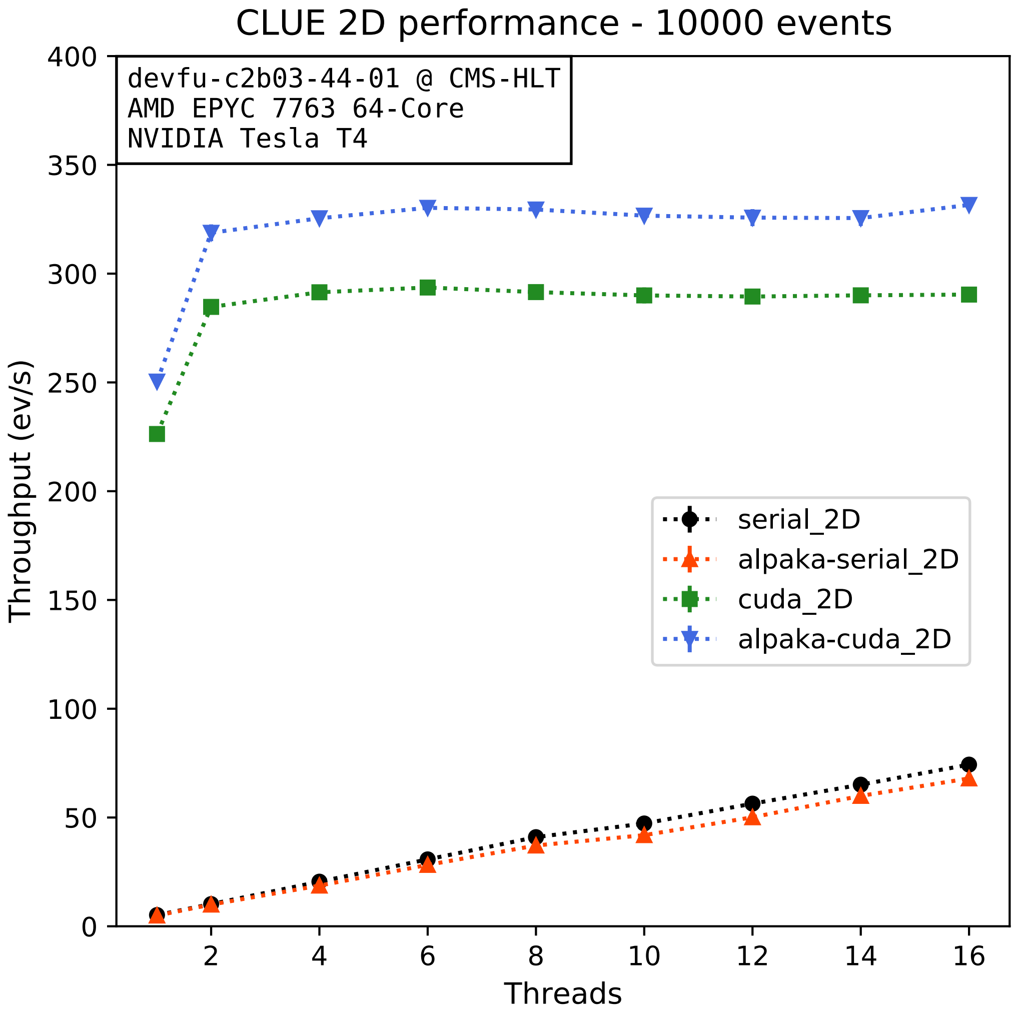
Called when launching the kernel

Equivalent to CUDA __global__

CPU/GPU alpaka: same as CUDA, with a user-defined helper function that accounts for an additional "***elements***" abstraction layer

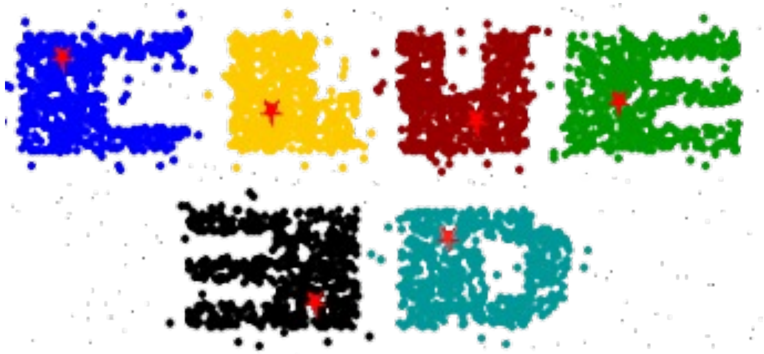
- Work division organized in Grids-Blocks-Threads-Elements

CLUE - Performance plot

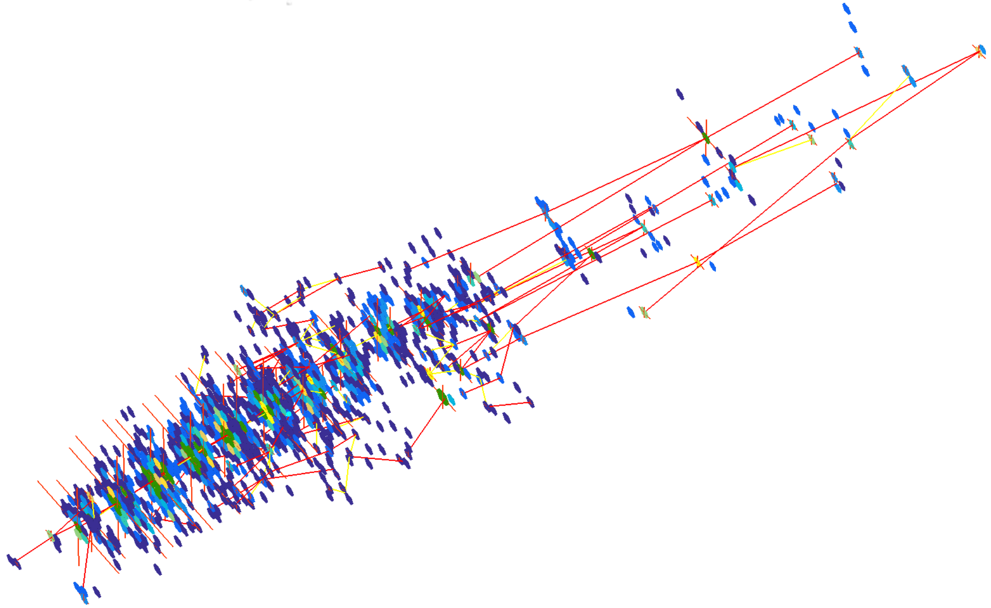


- **Alpaka** with the **serial** backend ***scales linearly*** with the number of threads (concurrent events), ***the same way*** as the **native serial** implementation
- **Alpaka** with the **cuda** backend has the ***same scaling*** of the **native cuda** implementation. Two points are under investigation:
  - Other applications **do not show** that alpaka is faster than cuda
  - It seems that I/O operations and the computing capability of the GPU are limiting the scaling for threads > 4

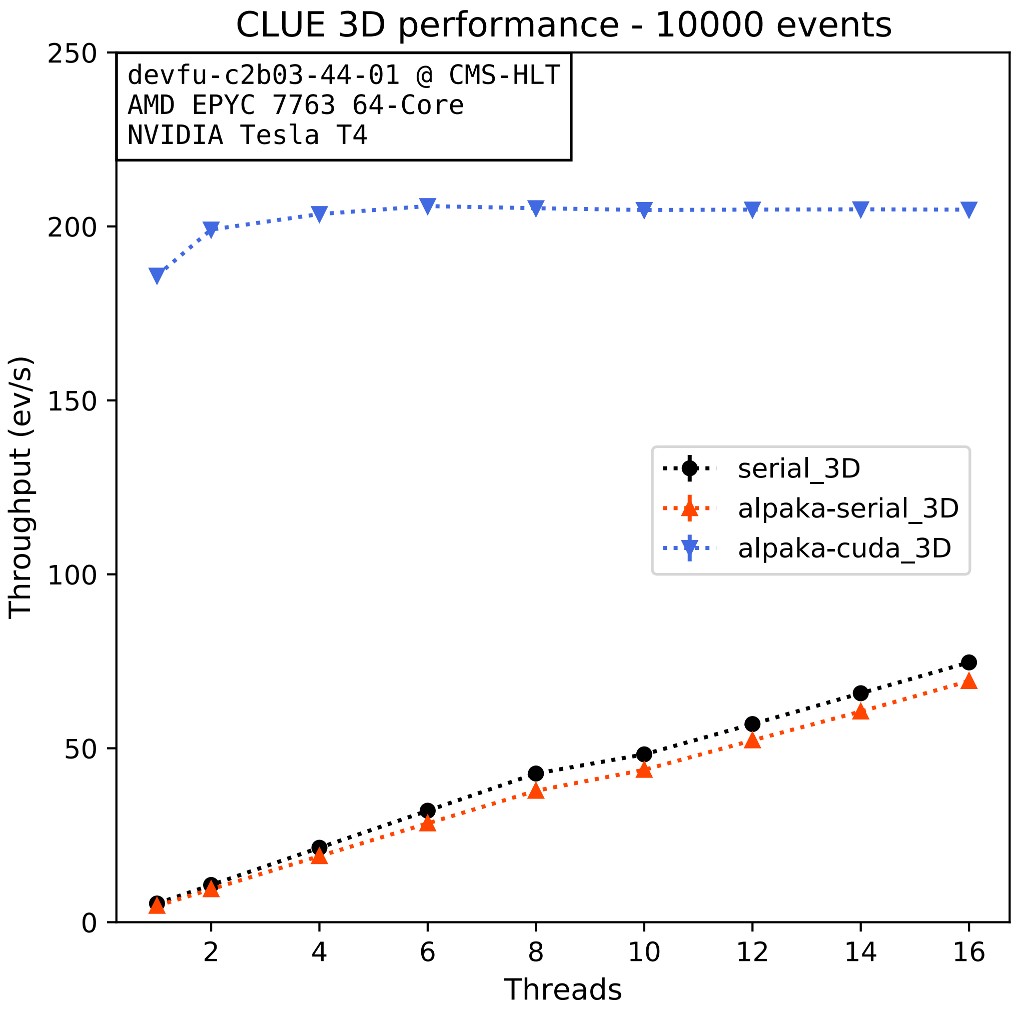
CLUE 3D (WIP)



- **3D version of the CLUE algorithm** to reconstruct particle showers in multi-layer high granularity calorimeters
- Builds 3D objects ***starting from clusters*** built with CLUE 2D
- Serial implementation currently used by the HGCAL reconstruction framework (TICL) in CMSSW
- **Ported to alpaka and can run on GPU now!**
- For more information, check **Wahid Redjeb's poster "The TICL reconstruction at the CMS Phase-2 High Granularity Calorimeter Endcap"** on Thursday



CLUE 3D – Performance plot



- **Alpaka** with the **serial** backend ***scales linearly*** with the number of threads (concurrent events) ***the same way*** as the **native serial** implementation
- **Alpaka** with the **cuda** backend provides a a high throughput of **~200 events/second**
  - Compared with serial and the same number of threads (i.e. 2), throughput is more than 20 times higher
  - Also for CLUE 3D, throughput on GPU seems limited by I/O operations

Work in progress and future plans

- CLUE has been ported to another performance portability library: **SYCL/oneAPI** (credits to ***Luca Ferragina*** and ***Juan Jose Olivera Loyola***)
  - Performance under study
  - CLUE 3D expected to be ported as well
  - For more information, check **Aurora Perego's poster** "**Experience in SYCL/oneAPI for event reconstruction at the CMS experiment**" on Tuesday
- A python library named **CLUEstering** (credits to ***Simone Balducci*** and ***Alessandro Mancini***) has been developed
  - Generalization of CLUE to N dimensions
  - Python binding to C++ serial implementation
  - Expected binding to C++ alpaka implementation in future

Conclusions

- The alpaka performance portability library is **an interesting solution** in the era of heterogeneous computing
  - ***Write the code once, compile it, and run it on different backends!***
  - Performance **close to native implementations**
  - New backends are planned and/or in development (i.e. SYCL)
- CLUE represents a **useful testbed** for performance portability solutions
  - Simple application
  - Tests have been made with both alpaka and SYCL/oneAPI
- CLUE 3D is the **first algorithm**, within the HGCAL-TICL reconstruction framework, that has been **ported directly from serial C++ to alpaka**
  - Optimizations still ongoing

Thanks for your attention

CLUE repository: heterogeneous-clue

CLUE original paper: CLUE

email: cms-patatrack@cern.ch

Backup

Porting to Alpaka: what to know

- Programming strategy *inspired by* CUDA
  - Easy porting CUDA-to-alpaka
  - Same way of organizing the work division – ***Grids-Blocks-Threads*** + additional abstraction layer ***Elements*** that can be exploited for vectorization
- Performance is close to the native backend
  - No overhead with respect to native CUDA or HIP/ROCm
- Alpaka objects behave like shared_ptrs → must be passed by value or const reference
- native buffers (vectors, arrays, …) must be ported to alpaka buffers, which **don't have a default constructor**

Kernel launch comparison

```
kernel_compute_histogram<<<gridSize, blockSize, 0, stream_>>>(d_hist.get(), d_points.view(), host_pc.x.size());
```

CUDA baseline

```
auto WorkDiv1D = cms::alpakatools::make_workdiv<Acc1D>(gridSize, blockSize);
alpaka::enqueue(
    queue_,
    alpaka::createTaskKernel<Acc1D>(WorkDiv1D, KernelComputeHistogram(), hist_, d_points.view(), d_points.n));
```

alpaka: kernels are enqueued in task objects